2025 IMEKO TC-6 International Conference on
Metrology and Digital Transformation - **M4DConf 2025**
Benevento, Italy, September 3-5, 2025

dcclib – Consolidation of Digital Calibration Certificate (DCC) tools into a unified Python library

Jan Loewe[1*], Justin Jagieniak[1], Shanna Schönhals[1]

[1]*Physikalisch-Technische Bundesanstalt (PTB), Bundesallee 100, 38116 Braunschweig, Germany*
[*]*Corresponding author: jan.loewe@ptb.de*

***Abstract* – The increasing adoption of Digital Calibration Certificates (DCCs) has led to the development of different tools and libraries, each addressing different aspects and needs of DCC creation and processing workflows. These tools and libraries differ in their functionality, interfaces and programming languages used, resulting in a fragmented landscape with an inconsistent user and developer experience. This work presents *dcclib*, a library that provides a standardized interface for basic DCC functionality by consolidating the functionalities from existing tools and libraries into a single, easy-to-use, and modular Python library. *dcclib* is designed to be extensible and can be used as the basis for more advanced DCC tools and libraries. Its functionality is accessible through an interface that can be used in Python scripts, a command line interface (CLI) and a REST API.**

I. INTRODUCTION

The Digital Calibration Certificate (DCC) is an XML-based document and data exchange format for calibration certificates. It has been developed since 2017 by an international consortium from the international metrology community coordinated by Physikalisch-Technische Bundesanstalt (PTB) in close collaboration with industry stakeholders. It aims to provide calibration results in a structured, machine-readable, and machine-actionable format, facilitating automated processing and increased interoperability across systems [1]. A DCC consists of four main areas: a regulated administrative data section, partially regulated measurement results, unregulated comments, and additionally a human-readable document [2]. Due to its XML-based structure, and because it is intended to eliminate manual human intervention of handling calibration data, the DCC workflow requires dedicated tools and libraries to create, validate, and process DCCs. Such middleware is essential for making full use of the benefits of the DCC and enabling integration into automated workflows.

A. Current situation

A range of tools has been developed to support the creation and processing of DCCs. These tools address specific use cases and have emerged from projects such as GEMIMEG [3] and GEMIMEG-II [4]. However, the landscape is characterized by fragmentation, with tools implemented in various programming languages, including Python, JavaScript, Java, and Rust, and exhibiting significant variations in software quality. In addition, most tools are limited in scope and functionality, meaning that different tasks often require different tools.

One example is the library *pyDCC*, which provides basic functionality for extracting administrative data and measurement results from DCCs [5]. This library was developed by partners of the GEMIMEG-II consortium and is available on GitHub under the MIT License [6]. By definition, it is a read-only library and it is out of its scope to provide other functionalities. For instance, it lacks support for other crucial aspects such as the generation of human-readable outputs or evaluating formulae. Graphical tools like the *GEMIMEG-Tool* offer broader features such as generating and modifying DCC, schema validation and XSLT-based rendering [7] but do not cover all aspects of the DCC lifecycle either. The result is an inconsistent and often inefficient tooling environment.

B. Motivation

The fragmented state of existing tooling has led to a clear need for consolidation. Multiple stakeholders, including national metrology institutes and commercial calibration laboratories, have expressed the desire for more integrated and robust tooling solutions that can be integrated in production environment workflows.
This work addresses this demand by designing and implementing a unified software library that consolidates selected core functionalities currently spread across various existing tools. In addition to functional consolidation, ensuring a consistently high software quality is a key motivation for this work.

II. DCCLIB

A. *Overview*

The library *dcclib* is designed to standardize the available functionalities from existing tools and libraries into a single, easy-to-use Python library. As shown in Figure 1, the functionality of *dcclib* is accessible through a Python interface, a command line interface (CLI) and a REST API. The CLI and REST API are built on top of the libraries public interface, allowing users to access its modules in other programming languages or environments. Scripts can use the CLI to automate tasks such as validating examples in a CI/CD pipeline, while the REST API allows for remote access and integration into other systems.

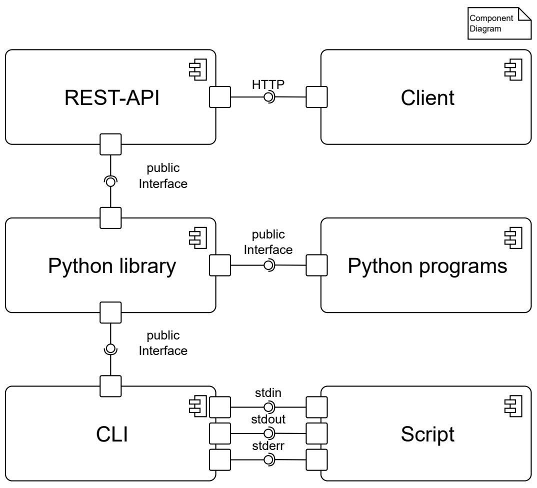

Fig. 1. Component diagram showing the library and its interfaces.

B. *Modules*

All functionality of the library is structured into modules, each of which contains classes for specific tasks. The five modules with a description are shown in Table 1. This design leaves room for extension and allows for the addition of new classes, for example to support new formats in the *conversion* module or to extract additional data from DCCs in the *extraction* module.
The *lxml* library is used by all modules to provide the `ElementTree` API for XML processing. *xmlschema* is used to convert XML to JSON. For XSLT transformations *saxonche* is used. Signatures are applied and checked using *signxml*.
In some cases the modules use each other to provide a more complex functionality. For instance, the *validation* module utilizes the XSLT class from the *transformation* module to execute Schematron validation using *SchXslt*.

Table 1. Module overview

Name	Description
conversion	Convert DCCs into different formats (e.g., JSON)
extraction	Extract key data from DCCs (e.g., formulae, attachments)
signature	Verify and apply digital signatures
validation	Validate structure and content of DCCs (e.g., against the XSD or Schematron)
transformation	Apply transformations (e.g., XSLT) to create human-readable outputs

C. *Implementation*

Python was chosen as the implementation language, due to its overall popularity and widespread use in data science [8]. Most of the existing tools and libraries that are considered for the consolidation are written in Python, making it easier to port existing code to the new library.
For the implementation of the CLI the library *click* was used, because it provides annotations to define the command line interface in a declarative way and is POSIX-compliant [9]. Each module is accessible via dedicated subcommands, for example validating a DCC against the XSD or Schematron with commands as shown in Figure 2. These commands take arguments like the file to transform (`<dcc_file>`) and flags such as the Schematron file to use (`--schematron <schematron_file>`). Each flag has a short and long version, for example `-o` and `--output` for the output file.

```
dcclib transform xslt [-o,
--output <output_file>] <xslt_file>
<dcc_file>
```

```
dcclib validate schematron [-s,
--schematron <schematron_file>]
<dcc_file>
```

Fig. 2. Example of the CLI usage.

The REST API is built using the library *APIFlask*, which extends *Flask* to provide a simple way to create RESTful APIs [10]. It automatically generates an OpenAPI specification, which documents the API and can be used to generate client and server libraries in different programming languages [11]. Similar to the CLI, the REST API provides endpoints for each module. The REST design rules by Masse (2011) [12] are followed to achieve a consistent and intuitive API design. Because of this and the stateless nature of REST, the DCCs need to be temporarily stored on the server to process them. Figure 3 exemplarily shows three endpoints together with their HTTP methods. The first endpoint is used to upload and temporarily store a DCC on the server. With the second endpoint, DCCs can be validated against the XSD. Files that are attached to a DCC can be extracted using the third endpoint.

```
POST  /dccs
POST  /dccs/<id>/validate/xsd
GET   /dccs/<id>/extract/files
```

Fig. 3. Example of the REST API usage.

D. *Formula Module*

The formula module is the most sophisticated component of the library. Internally, it is divided into two parts, each handling a specific task: one part is responsible for extracting formulae and variables from the DCC, while the other handles the conversion of Content *MathML* formulae into *SymPy* expressions.

Formula extraction

In the first step, the formula is extracted from the XML document. A machine-interpretable formula is typically provided in Content *MathML* [13], which represents mathematical expressions in Polish notation. This format is algorithmically efficient and easily parsed by machines.

In a DCC, formulas may also appear in Presentation *MathML* or LaTeX. However, since these formats are primarily intended for visual rendering rather than computational processing, they are disregarded during extraction.

Variable extraction via IDs

Content *MathML* references variables using an `xref` attribute, which links to unique identifiers (IDs). These IDs correspond to quantities defined elsewhere in the DCC. Due to the DCC's structured quantity hierarchy, extracting the actual numerical values is straightforward.

If a variable is part of a list structure within the DCC's quantity definition, it is preserved in the formula object. This ensures that the program handles the data correctly. A validation check ensures that either only a single value is provided per variable, in which case it is applied to every element of other variable lists, and all lists have the same length (as required in a valid DCC).

Conversion from Content MathML to SymPy

The conversion process transforms Content *MathML* into a *SymPy*-compatible expression, primarily by parsing the Polish notation into a string that *SymPy* [14] can interpret. When mathematical errors, invalid formulae and other problems are encountered, exceptions are thrown. These can be handled with try-catch-clauses.
The current implementation does not support calculations involving units and uncertainties, as the *SymPy* library provides limited or no support for these features. However, due to the modular design of *dcclib*, it is feasible to use a different backend for mathematical computations in the future. One option is the *dccQuantities* library, which supports arithmetic operations with both units and uncertainties [15].

Recursive parsing approach

Due to the nested nature of Polish notation, the conversion employs recursion, starting with the deepest (innermost) elements:

- **Function mapping:** A dictionary of function pointers (e.g., plus, sin, log) maps *MathML* operators to their corresponding *SymPy* functions.
- **Variable handling:** Variables are passed as parameters to these functions.
- **List support:** If a variable is a list, the parser iterates over each element, recursively processing them individually. The results are aggregated into a list for output.

The recursion unwinds outward, combining intermediate results until the full *SymPy* expression is constructed. This method ensures accurate representation of complex mathematical operations while maintaining computational efficiency. Multi-dimensional arrays or matrices are not supported in the current implementation, but support for them in the future is feasible.

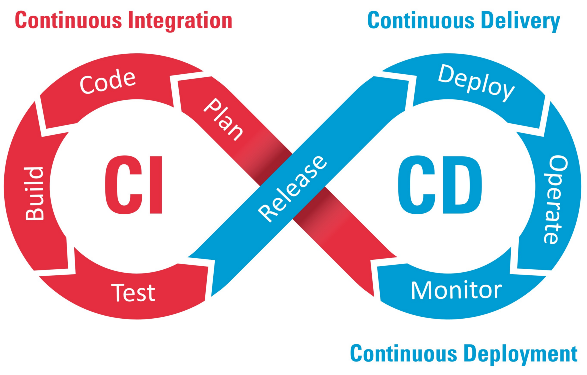

Fig. 4. Continuous integration and continuous delivery workflow.

E. *Quality Assurance*

To ensure the quality of the code, unit-, integration- and end-to-end tests are implemented using *pytest*. These tests cover all modules and functionalities of the library and are automatically run on every change using GitLab CI/CD, in the phase Test in Figure 4. For example, in the formula extraction and conversion process, the tests verify that the extracted formulae are correctly converted into *SymPy* expressions and that the calculated values match the expected results.

F. Distribution

The library and its interfaces are distributed through several channels to provide broad accessibility to its functionalities. The library itself and the CLI are available on the GitLab package registry and can be installed via package managers like *pip* or *uv*.
Docker images are provided for the REST API and the CLI on the GitLab container registry. This allows users to run the REST API and CLI in isolated environments without the need for a local installation. Another use case is using the Docker image for the CLI in CI/CD pipelines for example, to validate examples against the XSD or Schematron.
For use cases where no Python installation is available, the CLI is also distributed as a standalone executable which is built using the library *PyInstaller*.
All of these distribution methods are performed after testing and building the library and its interfaces in the Deploy phase of the CI/CD pipeline, as shown in Figure 4.

G. Limitations

Although *dcclib* provides a consolidated approach to DCC processing, some limitations must be acknowledged.

Conversion limitations

Currently the conversion module only supports conversion to JSON. This can be extended in the future to support other formats, such as CSV or YAML, but is not yet implemented. More importantly, the XML to JSON conversion process has inherent limitations, particularly the loss of element ordering information. This only happens when differently named nodes are on the same level in the XML hierarchy, which are then grouped together in arrays in the JSON output.

REST API Security

Currently, the REST API does not implement any security measures such as authentication or rate limiting. This limitation is known and can be addressed by deploying the REST API behind a reverse proxy that handles these aspects. The REST API also requires temporary storage of the DCCs to process it. This is achieved by storing the DCCs in a temporary directory on the server, from which they are deleted after a configurable time period (five minutes by default). At present, there are no measures in place to stop denial-of-service attacks caused by the uploading of large files or a high volume of files in a short period of time. In its current state, the REST API is intended for non-public use cases, such as integration into internal workflows or systems.

III. CONCLUSION

A. Summary

This work presented the approach to a consolidated toolset that provides core functionalities to work with DCCs. After analysing the current state of available tooling options that revealed a fragmented landscape, a modular approach has been developed with the focus on functional consolidation as well as consistently high software quality. The result of the work is the library *dcclib* that is structured into functional modules that address at the moment five core functionalities required for processing DCCs. The work includes implementation aspects such as the choice of Python as programming language, testing as an important part of quality assurance as well as the interfaces that are built-in components of the libraries to enable integration in existing workflows and environments. The aspect of distribution is covered as well addressing different distribution pathways including GitLab, Docker images and executables.

B. Outlook

As the development and introduction of the DCC progresses, additional functionalities will come into focus to facilitate the work with these documents. Thus, future work will focus on the extension of *dcclib* to support additional DCC functionalities, such as enhanced formula evaluation and more extensive format conversion options. Also, functionalities and libraries to facilitate handling of tables in the DCC or working with quantities could be incorporated and provided as part of the consolidated library. Further developments could also include the integration of a graphical user interface (GUI) to make the library even more accessible to non-technical users which is important for better understanding and easier uptake of the DCC.
The library is planned to be released as open source software, enabling the community to contribute to its development and improvement. Their contributions are highly appreciated and will be facilitated through a public Git repository. The packages will be distributed via Python Package Index (PyPI) and the Docker images via *Docker Hub*, making the work available to a wider community through standard channels.

IV. CITATIONS AND REFERENCES

[1] Siegfried Hackel et al. "The fundamental architecture of the DCC". In: *Measurement: Sensors* 18 (2021), p. 100354. ISSN: 2665-9174. DOI: 10.1016/j.measen.2021.100354.

[2] Siegfried Hackel et al. "The Digital Calibration Certificate". In: *Metrologie für die Digitalisierung von Wirtschaft und Gesellschaft* (2018). DOI: 10.7795/310.20170403.

[3] *GEMIMEG-I*. URL: https://www.gemimeg.ptb.de/en/gemimeg-i/ (visited on 07/15/2025).

[4] *GEMIMEG-II*. URL: https://www.gemimeg.ptb.de/en/gemimeg-home/ (visited on 07/15/2025).

[5] Siegfried Hackel et al. “The Digital Calibration Certificate (DCC) for an End-to-End Digital Quality Infrastructure for Industry 4.0”. In: *Sci* 5.1 (2023). ISSN: 2413-4155. DOI: 10.3390/sci5010011.

[6] Tim Ruhland Andreas Tobola and Benedikt Seeger. *Python library supporting automated processing of digital calibration certificates (DCC)*. URL: https://github.com/siemens/pydcc (visited on 07/15/2025).

[7] Héctor Laiz et al. “2nd international DCC-Conference 01-03 March 2022 Proceedings”. In: *PTB-OAR*. Physikalisch-Technische Bundesanstalt (PTB). 2022. DOI: 10.7795/820.20220411.

[8] JetBrains. *The State of Developer Ecosystem*. Tech. rep. JetBrains s.r.o., 2023. URL: https://www.jetbrains.com/lp/devecosystem-2023/ (visited on 03/24/2025).

[9] *Why click? — Click Documentation*. URL: https://click.palletsprojects.com/en/stable/why/ (visited on 03/24/2025).

[10] *APIFlask — APIFlask Documentation*. URL: https://apiflask.com/ (visited on 03/24/2025).

[11] Darrel Miller et al. *OpenAPI Specification v3.1.0*. Feb. 2021. URL: https://spec.openapis.org/oas/v3.1.0.html.

[12] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O’Reilly Media, 2011. ISBN: 978-1-4493-1990-8.

[13] Robert R Miner, Patrick D F Ion, and David Carlisle. *Mathematical Markup Language (MathML) Version 3.0 2nd Edition*. W3C Recommendation. https://www.w3.org/TR/2014/REC-MathML3-20140410/. W3C, Apr. 2014.

[14] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[15] *dccQuantities*. URL: https://gitlab1.ptb.de/digitaldynamicmeasurement/dccQuantities (visited on 07/15/2025).