ProM4Py- A Python Wrapper For The ProM Framework

Henrik Kaemmerling[1], Eduardo Goulart Rocha[2,1,*] and Wil M.P. van der Aalst[1,2]

[1]Process And Data Science (PADS) Chair - RWTH Aachen University, Aachen, Germany
[2]Celonis Labs GmbH, Munich, Germany

Abstract
Process mining is a young research discipline combining process and data science to analyze processes. Part of the success of process mining can arguably be attributed to the availability of open-source tools, datasets and standards that facilitate the exchange of ideas. The current leading process mining frameworks, ProM and PM4Py, present different tradeoffs in terms of scalability, ease-of-use, and feature completeness. We present ProM4Py, a Python wrapper for the ProM framework combining the scalability and feature-richness of ProM with PM4Py's ease-of-use. ProM4Py seamlessly integrates with PM4Py, such that existing PM4Py scripts can be migrated to it with minimal code changes.

Keywords
Process Mining Tooling, ProM, PM4Py, Python

Metadata description	Value
Tool name	ProM4Py
Current version	0.1.0
Legal code license	MIT
Languages, tools and services used	Python / PM4Py, JAVA / ProM
Supported operating environment	GNU/Linux , Microsoft Windows
Download and Screencast	https://zenodo.org/doi/10.5281/zenodo.13622550

1. Introduction

Open-source tools play a pivotal role in the development of process mining. By centralizing existing algorithms and making them easily accessible, these tools facilitate the rapid exchange of ideas. Currently, ProM and PM4Py are the leading academic frameworks for process mining. Both tools strike different trade-offs in terms of scalability, provided features, and ease-of-use.

The ProM framework [1] is almost as old as the field. As such, it contains the reference implementation of many foundational process mining algorithms. ProM is based on a loosely coupled plugin architecture. Contributors add their plugins as packages, which are then bundled

ICPM Doctoral Consortium and Demo Track 2024, October 14-18, 2024, Kongens Lyngby, Denmark
*Corresponding author.
henrik.kaemmerling@rwth-aachen.de (H. Kaemmerling); e.goulartrocha@celonis.com (E. Goulart Rocha); wvdaalst@pads.rwth-aachen.de (W. M.P. van der Aalst)
0009-0007-1259-1054 (H. Kaemmerling); 0009-0000-1184-1188 (E. Goulart Rocha); 0000-0002-0955-6940 (W. M.P. van der Aalst)

together and distributed as part of the framework. ProM is written in JAVA, which makes it highly efficient, but unsuitable for rapid prototyping. Furthermore, ProM does not integrate with the machine learning and data science ecosystems, which are predominantly Python-based.

In comparison, PM4Py [2] is a more recent addition to the process mining toolset. PM4Py is developed with a focus on flexibility and ease of prototyping. As a Python-based library, PM4Py integrates well with the broader ecosystem of machine learning and data science tools. However, this added flexibility comes at the cost of computational efficiency. Moreover, PM4Py does not implement all the algorithms available in ProM, or, in some cases, both frameworks disagree on the implementation of the same algorithm.

In this demo paper, we present ProM4Py, a Python wrapper for the ProM framework, aiming to fully and seamlessly integrate ProM's CLI API into PM4Py. While there have been attempts in the community to expose ProM's functionality to other environments [3, 4]. As far as we are aware, this is the first attempt at combining the PM4Py and ProM frameworks.

We demonstrate how ProM4Py can be used to, with minimal code changes, speed up common process mining workflows that were previously implemented with PM4Py. The tool is currently in early development and while the evaluation demonstrates the feasibility and practicality of the approach, further work is needed to improve its robustness and usability.

2. Integrating the ProM and PM4Py Frameworks

ProM4Py has three main requirements:

1. **(usability)** PM4Py users can easily learn how to use ProM4Py
2. **(usability)** PM4Py scripts can be ported to ProM4Py with minimal changes
3. **(maintainability)** ProM4Py can be easily extended to incorporate new ProM plugins

It follows from the first two requirements that ProM4Py must offer an API that is similar to PM4Py and that PM4Py and ProM4Py's data structures must be interoperable. The third requirement implies that ProM4Py must provide a generic solution to support ProM plugins.

The resulting tool consists of a JAVA backend and a Python client. The JAVA backend is implemented in ProM as a new PluginContext (ProM4PyContext) and runs on a separate process. At startup, the ProM4PyContext initializes the ProM framework, loads available plugins, and starts a HTTP server implementing a REST API that exposes ProM's functionality. The Python client is responsible for establishing a connection with the ProM backend, coordinating API calls, and coverting Python objects from/to ProM, such that the user never interacts with the REST API. To illustrate ProM4Py's usability, Listing 1 provides an example code demonstrating how to use ProM4Py to mine a PM4Py event log using ProM's inductive miner plugin.

To use ProM4Py, one must first create a ProM4Py client object (line 2), through which methods can be called. This only requires passing the path where ProM4Py's JAVA client is installed. If an instance of the backend service is already running, the client will refer to that one. Otherwise, a new instance of the service is started as a separate process. ProM4Py exposes Python functions corresponding to each supported ProM plugin (line 6). Notice that a PM4Py log is passed as parameter to the function. When calling a plugin, ProM4Py takes care of converting PM4Py objects into their ProM counter-parts.

```
1  # Create ProM4Py instance
2  prom4py = ProM4Py(prom4py_script_path)
3  # Load the EventLog using PM4Py
4  log = pm4py.read_xes(log_file_path)
5  # Mine the event log using ProM's inductive miner with noise threshold = 0.2
6  net = prom4py.mine_petri_net_with_inductive_miner_with_parameters(log, 0.2)
7  # Extract the resulting net and visualize it with PM4Py
8  net = prom4py.create_accepting_petri_net(net[0]).extract()
9  pm4py.view_petri_net(net.net, net.initial_marking, net.final_marking)
10 # Explicitly terminate the engine
11 prom4py.terminate()
12 # CachedProMObjects are always garbage-collected when the script terminates
```

Listing 1: Sample code to discover a process model from a PM4Py event log.

ProM4Py avoids transferring any data from JAVA to Python. Instead, plugin calls always return a reference (called CachedProMObject) to the JAVA object representing the computation. CachedProMObjects can be converted to their respective Python objects using the `extract()` method (line 8), as long as an appropriate serializer is implemented on both sides (see Section 2.2). The user can choose whether the backend process should also be terminated (line 11), or kept alive to speed-up the startup of subsequent script executions. Last, upon termination (or process abort) ProM4Py's client cleans up the objects in the backend's cache store (see Section 2.1).

Notice that ProM4Py and PM4Py code can be freely mixed, and changing from a PM4Py function to its ProM4Py equivalent consists of a mere change in the name of the callback.

2.1. Architecture

Figure 1 depicts ProM4Py's architecture, highlighting the roundtrip process involved in discovering a process model. When a ProM4Py method is called (step 1), the Python client first checks whether its input are Python objects or CachedProMObjects. In the first case, it first transfers the objects to ProM. This is done by serializing the object and calling ProM4Py's `/cache` endpoint (step 2), which de-serializes the object and stores it into ProM4Py's `CacheStore` (step 3). The `CacheStore` returns the object's key, which is forwarded to the Python client (steps 4 and 5). In case the inputs are already CachedProMObjects, these steps are skipped.

Then, the Python client issues a call to the `/plugin` endpoint with the name of the plugin and a list of parameters cache keys (step 6). The `/plugin` endpoint offers a standard interface to call all existing ProM CLI plugins. Upon request of a plugin execution, ProM4Py retrieves the JAVA objects corresponding to the plugin's arguments from the `CacheStore` (steps 7 and 8). It then uses the plugin's name and the parameters' typing information to determine the appropriate method to be called. The method is then called (steps 9 and 10). Its results are also stored in the `CacheStore` (steps 11 and 12). The cache keys are returned to the Python client (step 13), which creates a CachedProMObject and returns to the user (step 14).

This choice of architecture brings a few advantages and challenges. On the positive side, using CachedProMObjects minimizes the amount of data copying, which can be a potential bottleneck. Furthermore, the `/plugin` endpoint's generic interface means that ProM4Py can,

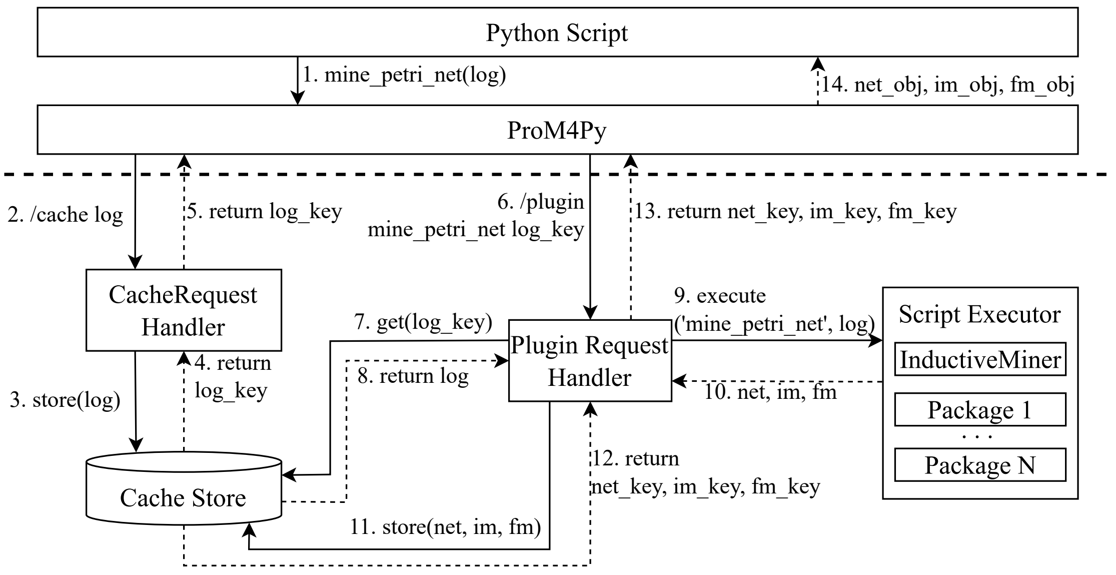

Figure 1: Overview of ProM4Py's architecture and roundtrip of `mine_petri_net` plugin.

in principle, support any ProM CLI plugin (see Section 2.2). On the negative side, since there are two separate processes, the system may reach inconsistent states such as orphan ProM processes, or ProM4Py objects pointing at non-existing instances.

2.2. Levels of Plugin Support and Compatibility

We distinguish between two levels of plugin support. A *fully supported* plugin is a plugin for which its input and output types can be freely interchanged between both runtimes. A *partially supported* plugin is a plugin that cannot serialize at least one of its inputs or outputs types. Among them, we distinguish between *uni-directional support* (for which its inputs can be serialized, but not the output, or vice-versa), and *fully-internal plugins*, when neither its inputs nor its outputs can be serialized. Since ProM4Py's API accepts CachedProMObjects as arguments, it is still possible to interact with these plugins by using other ProM plugins to read/write inputs from/to the file system, and convert between internal data-structures in ProM.

The notion of a fully/partially supported plugin assumes that the plugin is compatible with the schema from Figure 1, i.e. it assumes that parameters can be independently parsed and that adding support for them is merely a matter of implementing the required serialization methods. However, some plugins are *incompatible* with this schema. Supporting such plugins requires an extra implementation effort and result in a user-unfriendly API. Unfortunately, ProM's alignment package, arguably one its most used plugins, falls into this category.

ProM's alignment plugins require a mapping from transitions in the Petri net to event classes in the event log. However, the mapping relies on the ID of the JAVA object of the Petri net's transitions, which is only assigned once the Petri net is loaded into ProM. Therefore, the approach of serializing all input arguments separately does not work. Instead, ProM4Py must

Table 1
Execution time (in ms) of each computation step for PM4Py and ProM4Py. For ProM4Py, we also count the time to extract the alignment results from ProM to Python

BPIC-2020	PM4Py					ProM4Py				
	DD	ID	PL	PTC	RFP	DD	ID	PL	PTC	RFP
load log	2255	3667	5153	886	1768	802	1024	1207	235	448
discovery	125	861	8971	530	93	725	851	1063	286	473
align log	637	16771	263436	2488	628	405	3032	61314	827	355
extract alignments	-	-	-	-	-	253	20	1014	99	33
Σ	3016	21300	277561	3904	2489	1958	5228	64598	1447	1308

implement additional functionality to create this mapping. As a result, a custom alignment end-point was developed to explicitly support this plugin. Luckily, most ProM plugins are compatible with the schema from Figure 1, such that the majority of ProM's functionality can be supported without resorting to such workarounds.

3. Leveraging ProM4Py for Process Mining

We demonstrate how to use ProM4Py as a replacement for PM4Py in a common process mining workflow consisting of mining a process model and aligning the event log with the discovered model. For the evaluation, we use the BPI Challenge 2020 event log [5], which is split into 5 sublogs: Domestic Declarations (DD), International Declarations (ID), Permit Logs (PL), Prepaid Travel Costs (PTC), Requests For Payment (RFP). We use the the inductive miner-infrequent invariant with a noise of 0.1. We use PM4Py's `VERSION_STATE_EQUATION_A_STAR` alignment variant and ProM's `a_star_plugin` (LP-based replayer) replay algorithm, which implements the same marking equation-based a-star algorithm [6]. All experiments are run single-threaded on an Intel(R) Xeon(R) E-2276M CPU running Ubuntu 22.04. Table 1 shows a comparison of the execution time for both steps.

The first thing to notice is that loading the event log is consistently faster in ProM compared to Python. Loading the log is a bottleneck in simpler input scenarios such as DD and RFP, where the subsequent computation steps are relatively inexpensive. Generally, ProM4Py implements all computation steps more efficiently than PM4Py, the only exception being the discovery steps for DD and RFP. This can be partially explained by the usage of native DataFrame APIs in PM4Py's inductive miner implementation. However, during our experiments, we observed that PM4Py's inductive miner produces different results from those of ProM, i.e. the implementations of the two frameworks diverge. Thus, even without providing a performance improvement, ProM4Py can still be useful to provide access to the reference implementation of an algorithm.

When considering the alignment computation, we observe that performing this step in ProM is significantly (up to five times) faster than in Python and that the overhead associated with transferring the alignment results is, in most cases, negligible. Overall, ProM4Py is between 2 to 4 times faster than PM4Py. Since we used the same algorithms in both scenarios, this difference is attributed to the inherent performance differences between Java and Python, as well as other implementation-specific details. But notice that ProM also implements more efficient alignment plugins which are not available in PM4Py[7].

4. Maturity of the Tool and Future Development

We presented ProM4Py, a Python wrapper for the ProM framework. We demonstrate how the library exposes existing ProM plugins through an easy-to-use Python interface that can be seamlessly integrated with PM4Py. The experiments show the usefulness of the approach in combining the performance and feature-richness of ProM with PM4Py's ease-of-use. ProM4Py is still in its early stage of development, but it already supports common process mining tasks such as discovering process models, aligning event logs, and essentially any other plugin that is currently available via ProM's CLI interface.

We believe that it is possible to improve the integration between the Python and JAVA run times by using a standardized (and faster) data exchange format [8]. Further points of improvement include leveraging Python's dynamic typing to transparently manage ProMCachedObjects (making the extract() method optional), and, if possible, refactoring ProM's *incompatible* plugins to make them fit our computation schema. Ideally, the library should be fully integrated into the PM4Py and ProM's code-bases, but the licensing structure and governance models of both projects make it challenging.

Acknowledgments

We thank the Alexander von Humboldt (AvH) Stiftung for supporting our research.

References

[1] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, W. M. P. van der Aalst, The ProM framework: A new era in process mining tool support, in: Applications and Theory of Petri Nets, 2005. doi:10.1007/11494744_25.

[2] A. Berti, S. J. van Zelst, W. M. P. van der Aalst, Process mining for python (PM4Py): Bridging the gap between process- and data science (2019). arXiv:1905.06169.

[3] W. M. P. van der Aalst, A. Bolt, S. J. van Zelst, RapidProM: Mine your processes and not just your data, CoRR (2017). arXiv:1703.03740.

[4] H. Kourani, S. J. van Zelst, B.-D. Lehmann, G. Einsdorf, S. Helfrich, F. Liße, PM4KNIME: Process mining meets the KNIME analytics platform, in: ICPM Doctoral Consortium/Demo, 2022. URL: https://ceur-ws.org/Vol-3299/Paper14.pdf.

[5] B. F. van Dongen, BPI challenge 2020: Request for payment (2020). doi:10.4121/uuid:895b26fb-6f25-46eb-9e48-0dca26fcd030.

[6] A. Adriansyah, B. F. van Dongen, W. M. P. van der Aalst, Conformance checking using cost-based fitness analysis, in: EDOC, 2011. doi:10.1109/EDOC.2011.12.

[7] B. F. van Dongen, Efficiently computing alignments - using the extended marking equation, in: Business Process Management, 2018. doi:10.1007/978-3-319-98648-7_12.

[8] Google LLC, Protocol buffers: Google's data interchange format, 2008. URL: https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html.