

Towards Process Mining Use Case Map Models with PM4Py-UCM

Daniel Amyot 

School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario, Canada
damyot@uottawa.ca

Abstract—Given the increasing amount of data available in organizational systems, there is an opportunity for early requirements engineering (RE) activities to be better based on evidence than ever before. Process mining (PM) has been used for over two decades to discover and analyze as-is process models from event logs extracted from such data, with outputs often in the form of Petri Nets, directly-follows graphs, or BPMN models. This paper aims to make Use Case Map (UCM) models, from ITU-T’s User Requirements Notation (URN), a first-class output of process discovery, so that mined behavior can be used in URN-based modeling, analysis, and management activities. This paper contributes and illustrates PM4Py-UCM, an open-source extension to the existing PM4Py Python library. This new tool contributes 1) a UCM discovery pipeline, 2) hierarchical decomposition strategies producing nested UCM models, 3) configurable performer mappings for UCM and BPMN visualizations, and 4) an exporter to a URN tool (jUCMNav) that preserves the mined model under round-trip. Using public and synthetic event logs, the paper showcases how the same behavior is rendered under different performer abstractions and decomposition strategies, and discusses how PM can become a practical instrument for model-driven RE.

Artifact—<https://github.com/ProcessMining-uOttawa/pm4py-ucm>

Index Terms—Process mining, User Requirements Notation, Use Case Maps, model-driven requirements engineering, jUCMNav, PM4Py

I. INTRODUCTION

Process mining (PM), which enables the automated discovery and analysis of process models from event logs, has matured into a discipline that delivers data-driven evidence of how organizations actually execute their processes [1]. In particular, PM is known to support the discovery of process models in various notations, some limited to sequencing and alternatives (e.g., Directly-Follows Graphs – DFGs) and others also support concurrency (e.g., Petri nets – PNs [2], and OMG’s Business Process Model and Notation – BPMN [3]). Model elements are often annotated with frequency, time, or cost annotations, hence enabling performance analysis and the elicitation of improvement requirements. Discovered models, often labeled *as-is models*, can also be compared to expected or prescribed models during conformance analysis, to assess differences and infer alignment requirements. A few methodologies have emerged over the years to take advantage of PM in a process improvement context, with PM^2 being the best-known one [4].

Dozens of PM tools now exist on the market, most being cloud-based and commercial [5]. Among the open-source PM solutions available, one stands out as increasingly popular: PM4Py [6]. This Python library is used by practitioners and researcher alike due to its free access, the availability of its code, its extensibility, and the fact that it can run on local computers (an important capability for organizations that want to avoid sharing event logs on the cloud, often for privacy reasons).

From a requirements engineering angle, PM can be seen as a data-based elicitation technique [7] that aligns well with crowd-based requirements engineering when the source event logs capture user interactions with systems. Dąbrowski et al. [8] exploited that relationship to propose a method combining requirements engineering, process mining, and crowd-sourcing to discover the underlying processes of crowds from event logs, and detect misalignments with expected system behavior. More recently, Ghasemi and Amyot [9,10] explored the use of goal models as a filtering mechanism for preserving traces in an event log that satisfy a related collection of measurable goals. Bouhidel et al. [11] suggested a requirements-driven methodology that emphasizes specificity and iterative refinement when discovering models and presenting results, while calling for the customization of PM tools and analytical assets.

This paper answers this call by proposing a tool-oriented bridge between the process mining ecosystem and the requirements engineering ecosystem around the Use Case Map (UCM) notation, part of the User Requirements Notation (URN) standard [12]. The UCM notation, originally proposed by Buhr in the 1990’s [13], is combined with the Goal-oriented Requirement Notation (GRL) in URN, enabling the automated analysis of goal-process alignment, something that PNs and BPMN do not support on their own, out of the box. URN’s UCM notation also supports the binding of process activities (called *responsibilities*, akin to BPMN activities) to two-dimensional hierarchical structures of components (e.g., systems, system modules, users, and other types of *performers*), going beyond the stricter pool representation in BPMN.

In order to close the existing gap between process mining and the UCM/URN ecosystem, this paper extends PM4Py with support for the UCM notation, leading to the PM4Py-UCM tool. More precisely, the contributions of this paper are:

- **C1.** An event-log-to-UCM discovery pipeline with a documented mapping from PM4Py’s intermediate structures to UCM constructs. A Web interface to a deployed version of the pipeline (with sample event logs) is available online¹.
- **C2.** Configurable hierarchical decomposition strategies producing nested UCM models, also with support for BPMN (visualization only).
- **C3.** Performer-aware bindings (i.e., responsibilities connected to their performing components), with visualization for UCM and BPMN models that can be exported to PNG files.
- **C4.** A bi-directional importer/exporter that supports the jUCMNav file format (.jucm, XML-based).

The rest of this paper is organized as follows. Section II provides background information on the UCM notation and on process mining with PM4Py. Section III then introduces PM4Py-UCM’s architecture and discovery pipeline while Section IV explores and illustrates RE-related capabilities. Sections V and VI discuss related work and limitations, respectively, while Section VII concludes.

II. BACKGROUND

A. Use Case Map Notation and jUCMNav

Use Case Maps, standardized by the International Telecommunication Union (ITU-T) [12], are a requirements notation used for modeling and analyzing processes and scenarios. As summarized in Table I, the main UCM concepts share many similarities with those of BPMN [3], especially around start/end points, activities (responsibilities in UCM), sequencing, choice, concurrency, decomposition, and performers (components in UCM, pools in BPMN). This common subset is often the one targeted as output by process mining algorithms. BPMN also offers concepts not supported by UCMs, including message flows, object flows, and many more types of start/end events, which are often used when BPMN models specify executable automations. Meanwhile, UCMs offer an executable semantics based on a simple data model that supports scenario-based testing and visualization as part of the language. Being part of URN, UCMs are also integrated with and traceable to GRL goal models. These concepts unique to UCMs are more useful for requirements engineering activities than the ones unique to BPMN; however, they are not targeted by the current version of PM4Py-UCM.

Two decades of experience in using URN for different requirements engineering activities, including goal/process modeling and alignment, integration with requirements management systems, regulatory compliance and intelligence, process adaptation and improvement, value co-creation, and advanced extensions for feature modeling, aspect-oriented modeling, and model slicing, have recently been summarized [14] and could hence be made available to requirements engineers interested in PM-based process discovery. This is part of the added-value offered by the support of UCM models in a PM context.

TABLE I: Simple Comparison of Process Notations

Concept	BPMN	Petri Nets	DFG	UCM
Star/End	Yes	Yes	Yes	Yes
Activity	Activity/Task	Place	Activity	Responsibility
Sequence	Yes	Yes	Yes	Yes
Choice	Yes	Yes	Yes	Yes
Concurrency	Yes	Yes		Yes
Decomposition	Call Activity	Subnet		Stub
Performer	Pool/Lane			Component
Artifact	Object/Data Store	Col. token		Object
Message flows	Yes			
Links to Goals				Yes
Data Model				Yes
Test Definitions				Yes
Standardized	OMG	ISO		ITU-T

Many of these functionalities are currently supported by jUCMNav [15], an Eclipse-based environment that is the de facto standard for URN modeling and analysis. jUCMNav uses an XML-based file format for URN models, which is a serialization of the objects instantiating the tool’s metamodel. This open-source tool is freely available online².

B. Process Mining and PM4Py

Process mining uses event logs as input, and uses a mining algorithm that produces a process model (often with annotations) as output. An event log is generally produced by transforming event-related information found in information systems. A log is a list of events where each event must minimally contain information about its *case* identifier (customer, patient visit, sales transaction, etc.) capturing the process instance, together with the *activity* name and its *timestamp*. Other optional attributes can be used for enabling filtering and the computation of annotations (e.g., average activity costs) on the resulting model. Of particular here are performer-oriented attributes such as *roles* (e.g., client or claims officer) and particular *resources* (e.g., Nancy or Gary) playing such roles. Roles and resources are candidate attributes for representation as components in UCM models. Event logs are often stored as Comma-Separated Value (CSV) files or as standard XES files [16], where XES uses XML to provide clearer semantic definitions of the attributes.

Different mining algorithms exist, which offer different trade-offs [1]. Some are quick but do support concurrency: α -miner is precise but does not provide much abstraction and leads to complex models, whereas the heuristic-miner and fuzzy-miner algorithms provides some abstraction and better understandability. Others are slower but support concurrency detection (leading to simpler models, albeit less precise), including the inductive-miner and the split-miner. The *inductive-miner* algorithm [17,18] is of particular interest here as it generates well-nested process trees, amenable to representation with BPMN and UCM, and with a simple path towards support for hierarchical process decomposition. This algorithm first builds a DFG and then identifies a *cut* (a structural split) in that graph that matches basic workflow patterns, namely:

¹Web interface demo: <https://pm4py-ucm.streamlit.app/>

²<https://github.com/JUCMNAV/jUCMNavPlus/>

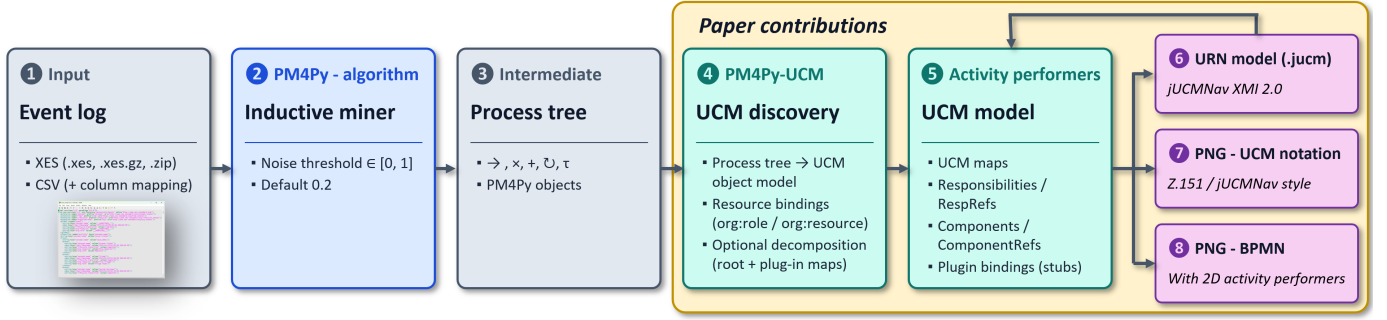


Fig. 1: The PM4Py-UCM discovery pipeline.

- Sequence ($\rightarrow$): Activity A must finish before activity B starts.
- Exclusive Choice ($\times$): Either activity A or activity B happens, but not both.
- Inclusive Choice ($\vee$): Activity A or activity B happens, or both happen.
- Parallel ($\wedge$) and Interleaving ($\circ$): Activities A and B happen concurrently, or in any order.
- Loop ($\odot$): An activity can be repeated multiple times.
- Silent (τ): An invisible activity, often used to capture an internal routing mechanism or skip logic.

Note that the above activities can also be sub-processes. Then, the algorithm splits the log at that cut point, and recursively applies the same split-based decomposition on each sub-log.

Although dozens of PM solutions exist nowadays [5], PM4Py is relevant here as this is an open-source and extensible library that supports several mining algorithms (including the inductive-miner³), as well as existing transformations to BPMN models [6]. However, none of these tools, including PM4Py, supported the UCM notation before this paper.

III. PM4Py-UCM: DISCOVERY PIPELINE

The main architectural strategy behind PM4Py-UCM is to reuse PM4Py for discovering a process tree, and then provide a transformation to a UCM object model (implemented as Python objects) that can be used for UCM visualization (akin to the existing BPMN ones) and model export to jUCMNav (.jucm file). The *UCM object model*, available online⁴, describes the various types of UCM path elements and their attributes, as well as how they are grouped in UCM maps, how stubs are decomposed into sub-maps, and how responsibilities are bound to components.

Figure 1 gives an overview of the discovery pipeline, which is this paper's first contribution (C1 in the introduction). Steps 1, 2, and 3 are performed with existing PM4Py features, steps 4 and 5 are where the main contributions of this paper reside, whereas artefacts 6, 7, and 8 show the different outputs generated. From an event log in CSV or XES format 1, PM4Py's inductive-miner algorithm is called 2 to produce a process tree 3. The algorithm has a threshold parameter that

can be set, to enable the mining of concurrency to be more or less tolerant to noise in the event log. That algorithm is used as is in the pipeline.

In step 4, the UCM discovery step converts the mined process tree to a UCM model that instantiates, using the mapping in Table II, the object model highlighted earlier. At this point, there is neither process decomposition nor performer binding done; this is in a way akin to current PM4Py capabilities that generate BPMN models from process trees [6].

TABLE II: Mapping from Process Tree Constructs to UCM Constructs

Process Tree	UCM Construct
Activity A	RespRef referencing a Responsibility A
$\rightarrow$	Children chained with EmptyPoint connectors (some being collapsed by a model simplifier)
$\times$ and $\vee$	OrFork $\rightarrow$ branches $\rightarrow$ OrJoin
$\wedge$ and $\circ$	AndFork $\rightarrow$ branches $\rightarrow$ AndJoin
$\odot$	OrJoin $\rightarrow$ do $\rightarrow$ OrFork with [redo] back-edge and [exit] forward edge
τ	Direct NodeConnection with no responsibility

The second contribution of this paper (C2), also found in step 4, focuses on the hierarchical decomposition of process models. The decomposition algorithm starts with a process tree-level planning step (i.e., to pick which subtrees become UCM plug-in maps for parent stubs) followed by a conversion-time intercept (i.e., the cut handler diverts those subtrees into their own maps as they are converted). The planning decisions are influenced by two categories of parameters:

- 1) **Workflow patterns of focus** (top four rows of Table III): these Boolean options enable users to decide which sequencing operators should be considered as potential cut points for decomposition. The first one is for the root map only (to handle long sequences of activities) whereas the others apply to all levels of decomposition.
- 2) **Size-based requirements** (bottom three rows of Table III): these numerical parameters enable users to set minimum and maximum numbers of elements (activities/stubs) per map, to better handle complexity. They can influence the decomposition depth.

As a convenience, several predefined decomposition modes (Table IV) are offered to users:

³See documentation of `discover_process_tree_inductive`.

⁴UCM object model available on GitHub in PNG and SVG.

TABLE III: Process Decomposition Parameters

Parameter	Default	Meaning
on_root_sequence	True	Each child of a top-level $\rightarrow$ becomes a plug-in. Root map reads as a chain of phase stubs.
on_parallel	True	Each $\wedge$ branch becomes a plug-in. AND-fork/join vertical-expansion cost is replaced by a single stub per branch.
on_alternative	True	Each $\times/V$ branch becomes a plug-in. OR-fork/join stays on the parent map; alternative bodies move into per-branch plug-ins.
on_loop	True	Each $\circlearrowleft$ operator's expansion becomes a plug-in. Parent map reads as forward flow with one stub for the iteration.
max_leaves_per_map	20	Cap: over-sized maps recursively force-cut the largest operator-subtree until the cap is met. Can override the above rules.
min_leaves_to_decompose	4	Floor: subtrees smaller than this stay inlined regardless of rules.
balance_ratio	0.2	Sibling share threshold under $\rightarrow$ and $\wedge$. A child needs at least this fraction of the parent's leaves to be pulled out independently.

TABLE IV: Process Decomposition User Modes

Mode	Definition
None / "off"	No decomposition.
"auto"	All workflow-related rules set to True, with max_leaves_per_map=20 and min_leaves_to_decompose=4.
"aggressive"	Same as "auto" but with max_leaves_per_map=10.
dict	A dictionary entry that provides specific values for any combination of the seven parameters in Table III.

Hierarchical UCM models can now be visualized and exported in PNG format [7](#), and exported valid .jucm files readable by jUCMNav [8](#). At this point, a simple script that uses PM4Py-UCM to mine a log in XES, decompose it aggressively, and save it both as a diagram and as a jUCMNav file is as follows:

```
import pm4py
import pm4py_ucm

log = pm4py.read_xes("log.xes")
ucm = pm4py_ucm.discover_ucm_inductive(log,
    decomposition="aggressive")

pm4py_ucm.save_vis_ucm(ucm, "diagram_ucm.png")
pm4py_ucm.write_ucm(ucm, "log.jucm")
```

Note that PM4Py also enables users to create process trees and UCM models programmatically (i.e., through Python method calls), which is useful for testing purposes. In addition, jUCMNav files can be modified and re-imported again as Python UCM model (see arrow from [6](#) to [5](#) in Figure 1), ensuring support for round-trip engineering. This essentially defines contribution C4.

Step [5](#) is concerned with contribution C3. At this point, UCM components are still not identified in the process model. The strategy here is to harvest roles and resources from an event log, assess what activities are related to them, and inject a simplified mapping (each activity being bound to one component, or none) into the UCM model.

```
# One-shot: mine and bind in a single call.
ucm = pm4py_ucm.discover_ucm_inductive(log,
    parameters={"resource_attribute":
        ["org:role", "org:resource"],
    })
pm4py_ucm.write_ucm(ucm, "log.jucm")
```

This performer-binding information can also be provided manually, through Python method calls; this capability is again useful for testing, but also for considering other sources of mapping information outside the log. The performer binding can use the role, resource, or any other attribute from the log as a source of UCM components.

As PM4Py-UCM is currently limited to one component per activity, and as the same activity in a log can be performed by many roles or resources, several options are offered when extracting performer bindings from a log (see Table V), which uses an example where activity *Act* with 10 events, performed by Alice (5 $\times$), Bob (3 $\times$), Carol (2 $\times$, first in log order).

TABLE V: Activity $\rightarrow$ Performer Aggregation Strategies

Strategy	Definition	Output
mode (default)	Most-frequent performer, with ties broken lexicographically. Activity left unbound if modal share $< \text{min_support}$.	Alice *
first	Performer of the first event for the activity, in log order.	Carol
unbound	Performer iff exactly one distinct value is observed; activity is omitted otherwise.	(omitted)
all	Composite "p ₁ +p ₂ +..." listing every distinct performer.	Alice+Bob+Carol

* Omitted if $\text{min_support} > 0.5$ (modal share is 5/10).

Once injected into the UCM object model, this performer binding information is used to create UCM components as well as bindings between activities and their containing components.

Going back to model generation, the components are also visualized in the PNG bitmaps [7](#) and exported to the .jucm files [6](#). Component colors are also selected based on the hashed value of the component name, so that the component remains colored the same way across generated models. Decomposed processes are stacked vertically with meaningful name inferred for stubs and their plug-ins.

As the UCM object model is in many ways similar to BPMN's in its support for activity sequences, choices, concurrency, decomposition, and performers (as seen in Table I), PM4Py-UCM also offers a BPMN-based hybrid PNG visualization [8](#), where the BPMN syntax is used to describe processes whose activities are bound and shown visually as UCM components instead of as pools and lanes. Activities are shown in yellow, decomposed activities in pink with a + annotation, gateways are in green, and the same coloring approach used for UCM models is also used for BPMN models. However, no real BPMN model (in XMI) is generated for now.

IV. ILLUSTRATION OF RE-RELEVANT CAPABILITIES

Two event logs will be used to illustrate the main capabilities discussed in the previous section, one about an issue tracking process (synthetic log with 100,008 events, 11,284 cases, and 9 activities), and the other about a claims payment process (subset of an anonymized real log with 78,126 events, 5,600 cases, and 25 activities). Both are available online on the PM4Py-UCM GitHub repository⁵ and are used in two provided

⁵<https://github.com/ProcessMining-uOttawa/pm4py-ucm/>

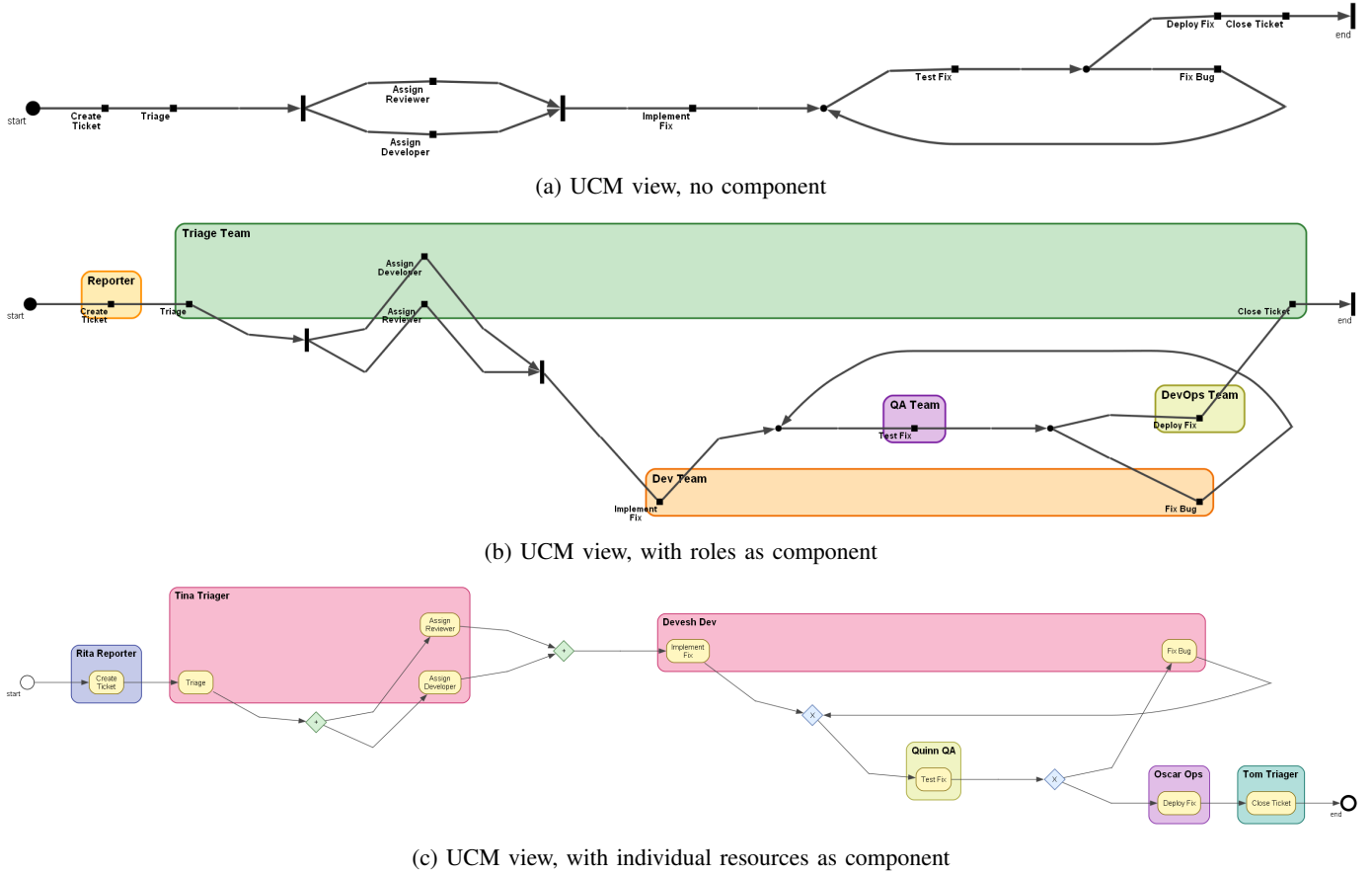


Fig. 2: Models mined from the issue tracking event log.

demo artifacts: a tutorial in *Jupyter Notebook*, and a Web-based interface built in Python on top of *Streamlit* [19]. The main RE scenario here pertains to the data-driven elicitation of a process model, and especially "who does what, and when".

A. Initial Elicitation via Process Discovery

The simplest PM4Py-UCM process model that can be extracted from an event log ① is one without decomposition and without components. Figure 2a shows such a model generated from the issue tracking event log (using inductive mining ② and our UCM discovery procedure ④), visualized as a UCM ⑦. It clearly shows 9 activities (shown here with small squares on a path), and sequence, concurrency, and loop constructs that correspond to the log's process tree ③:

(Create Ticket, Triage, $\wedge$ (Assign Reviewer, Assign Developer), Implement Fix, $\circ$ (Test Fix, Fix Bug), Deploy Fix, Close Ticket)

B. Performer Mappings

Performers can be extracted from the log (using the *mode* option, see Table IV), and injected to the UCM model ⑤. Figure 2b shows an example where roles were used as data source for the performer mapping. Each of the 9 activities is now bound to a UCM component. Some components (roles here) are responsible for multiple activities. Such a view

definitely lacks from PM tools as it shows on the same model the *who* aspect of "who does what, and when". RE practitioners are interested in the systems and users performing activities, not just the sequencing of the latter.

Figure 2c shows another example, using the hybrid BPMN-with-components visualization discussed earlier ⑧. In addition, resources (e.g., Tina Triager) have been used as a source of performers. This binding decision leads to differences with Figure 2b. For example, although Tina Triager and Tom Triager are both members of the Triage Team, evidence shows they specialize in different activities (and might not be fully interchangeable). If further elicitation about these activities is necessary, then it would be important to have them both involved in discussions.

C. Hierarchical Decomposition

The variety of decomposition options discussed in Section III suggests that no decomposition is canonical, and that a variety of views might be required by RE practitioners to enable effective understanding and communication. This will be illustrated using the claims payment log. Figure 3 shows the UCM model (and the Streamlit-based Web interface for PM4Py-UCM) generated, with role-based performer mapping, but without decomposition. This model contains 88 nodes and is too complex to visualize or understand as is.

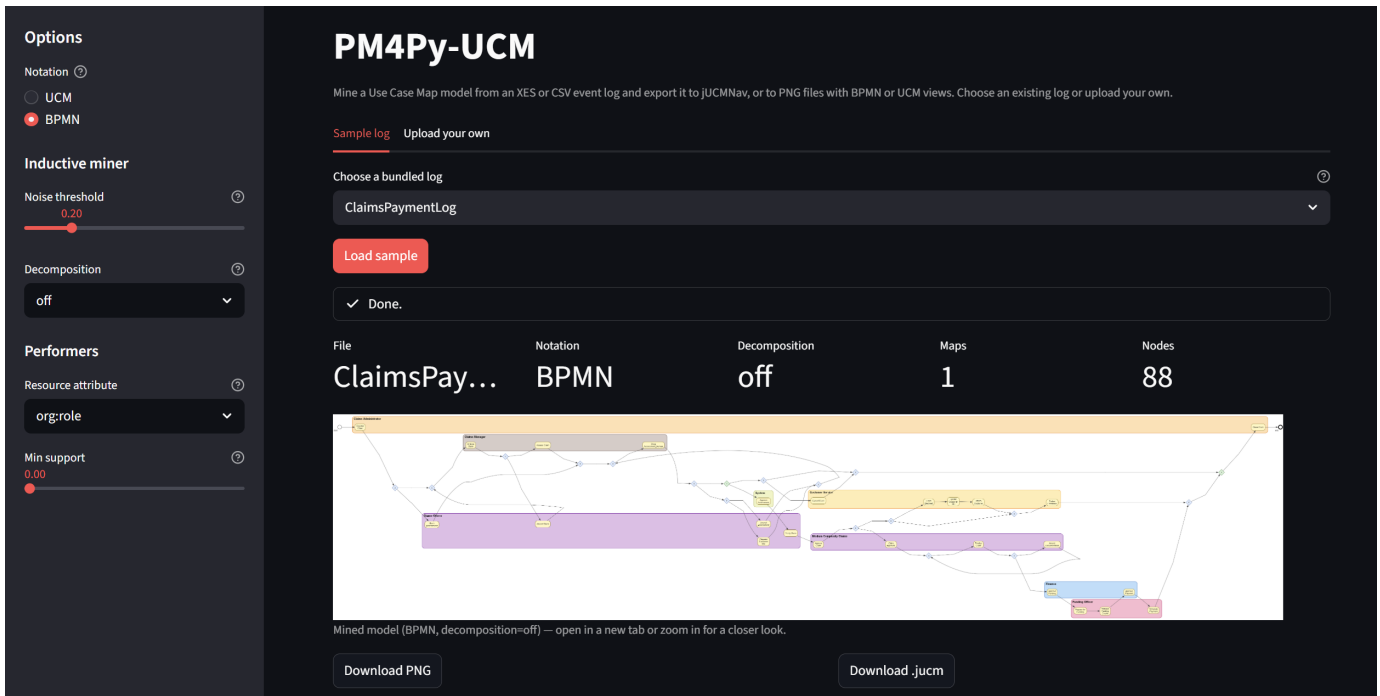


Fig. 3: Claims payment: UCM view without decomposition (from within the Web interface).

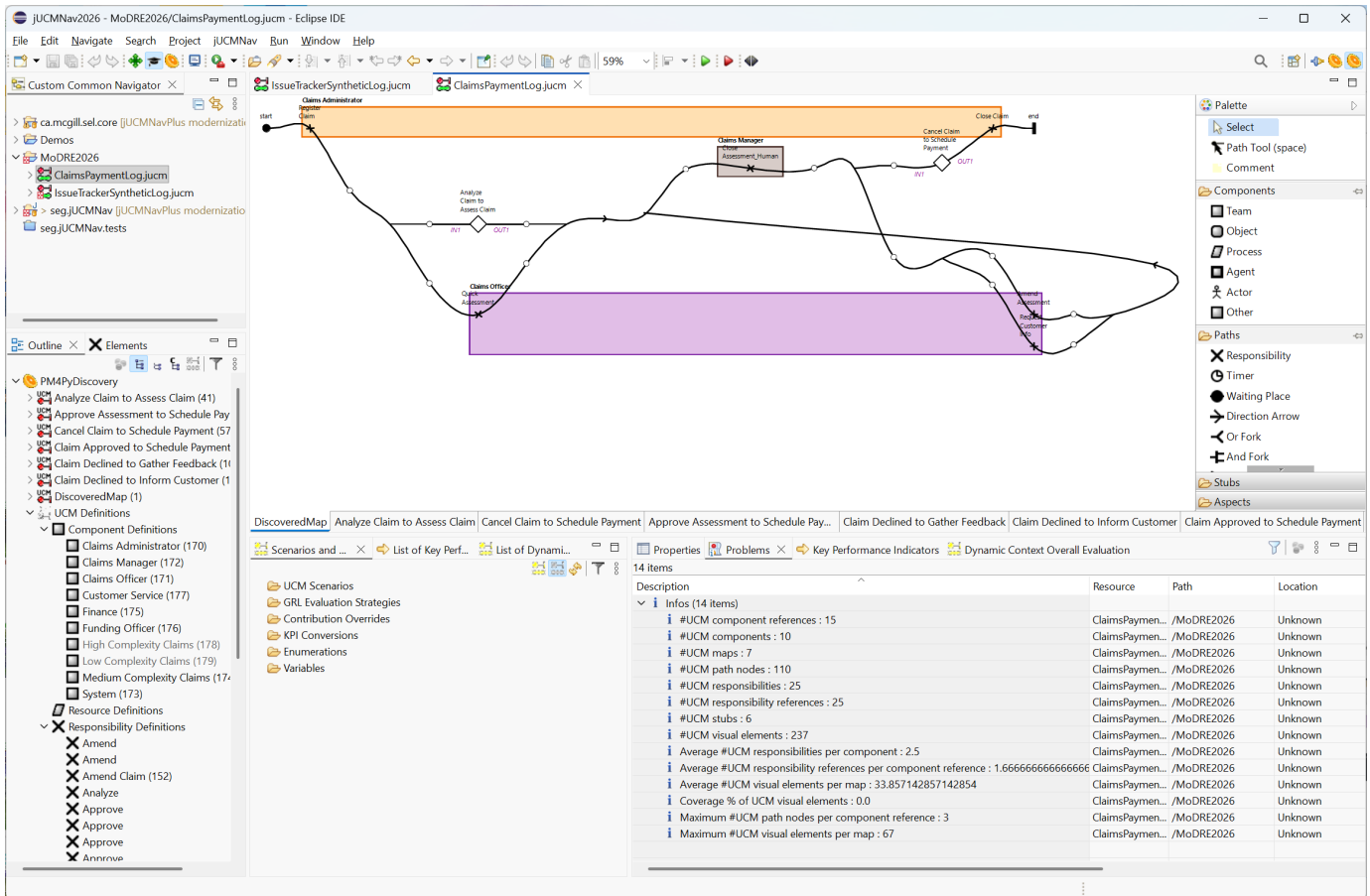


Fig. 4: Claims payment: UCM model exported by PM4Py-UCM to a .jucm file and opened in jUCMNav (<https://github.com/JUCMNAV/jUCMNavPlus>), with model metrics computed. Note that all 7 maps are there.

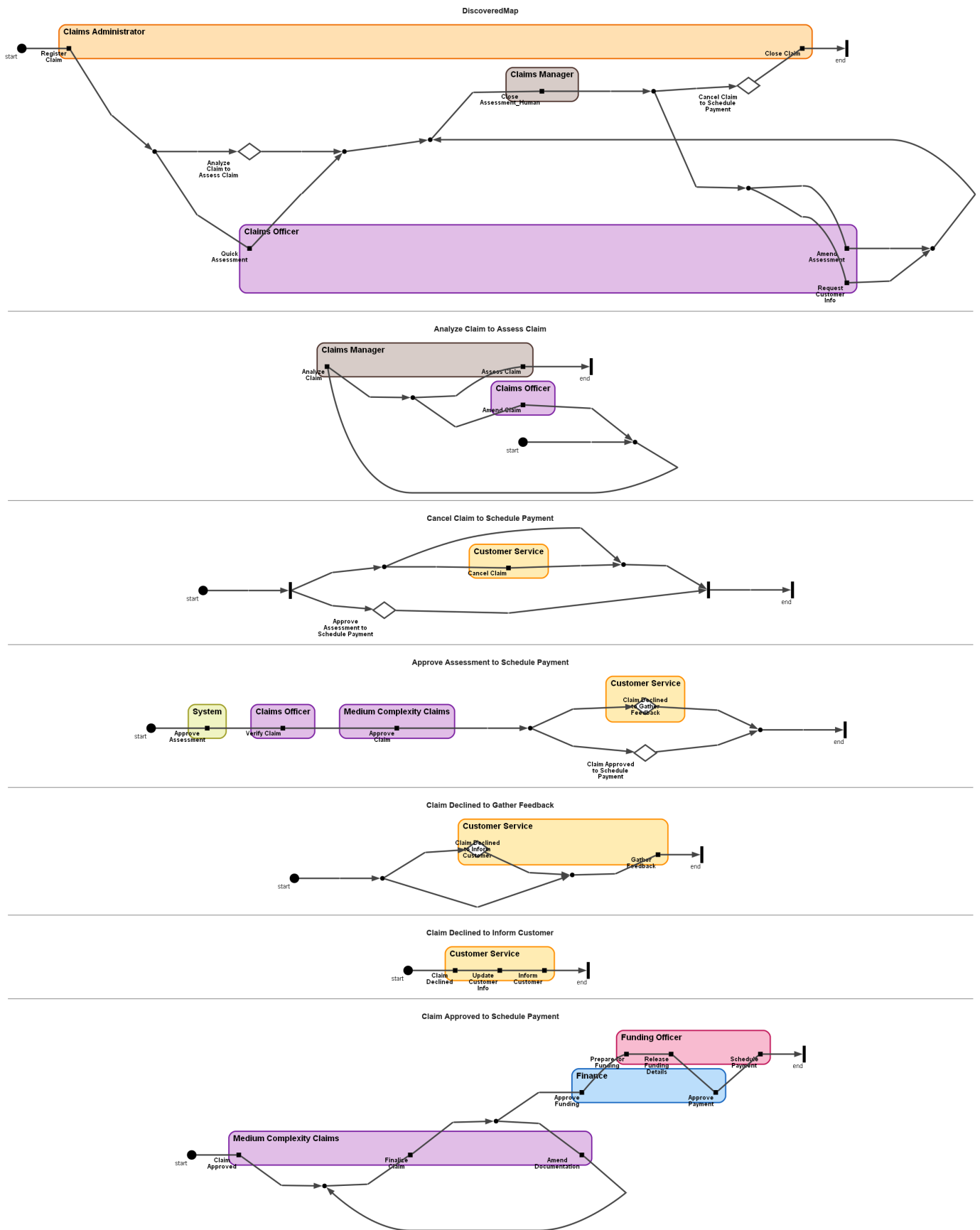


Fig. 5: Claims payment: UCM view with decomposition.

Using the decomposition ⑤ in mode auto (Table IV) splits this model into 5 maps, but with a root map that remains overly complex. However, by providing more aggressive settings regarding the number of elements per map (`min_leaves_to_decompose=3` and `max_leaves_per_map=7`), the number of maps becomes higher (7) but the root map becomes much easier to understand. The generated PNG file, which names and stacks all the plug-in maps (sub-processes for their respective stubs), is illustrated in Figure 5. The UCM stubs (represented as white diamonds) are the containers for sub-maps. Note also how the component colors remain consistent across the processes.

D. Round-Trip to jUCMNav for URN-Level Analysis

Mined UCM models can be exported to .jucm files, compliant with jUCMNav’s file format ⑥. For example, Figure 4 shows jUCMNav with the root map of the claims payment model with decomposition from the previous section. This is a fully editable model, that can be extended by a requirements engineer to trace its process elements to GRL goal model elements and assess alignment and consistency through jUCMNav’s OCL rules [20]. Different metrics can also be computed (again through predefined OCL queries), as shown on the bottom-right of the figure. Such an import also opens the door to scenario-based testing, test generation [21], and other URN-oriented capabilities briefly discussed in Section II-A.

Modifications to the model can also be re-imported to the UCM object model in Python, e.g., to explore other performer mappings or to visualize the results as BPMN diagrams (e.g., should stakeholders be more familiar with BPMN than UCM).

VI. RELATED WORK

Different categories of related work are worth exploring here.

A. Process Mining for Requirements Engineering

The intersection of process mining and RE has attracted growing attention, with papers arguing that mined evidence ought to feed back into requirements artifacts [9]. In particular, Ghasemi surveyed the literature looking at the connections between process mining, goal modeling, and goal/intention mining [22], while questioning the rationality and interpretability of mined models for RE activities. This worked led to a goal-driven log filtering approach, so PM models could focus on the cases that meet certain indicator-based objectives [10]. As this applies to event logs manipulation prior to PM, this is complementary to PM4Py-UCM.

Existing PM pipelines typically discover BPMN, DFG, or Petri-net models [6] and rely on a subsequent, often manual, translation step to obtain RE artifacts from these models. Aysolmaz et al. [23] however proposed a semi-automated approach for the generation of natural-language requirements documents from process models (mined or not), which could again complement the output of PM4Py-UCM.

Our contribution differs in that the mined process is itself represented in a standardized RE notation (UCM/URN), which

opens the door to a vast amount of other modeling and analysis activities unavailable for the other notations [14].

B. Hierarchical and Decomposed Process Discovery

The inductive-miner family of algorithms [17,18] produces block-structured process trees that admit a hierarchical reading, and trace-clustering approaches partition logs into behavioral variants amenable to separate discovery [24,25]. PM4Py-UCM differs in two respects here: decomposition strategies are selected along *RE-meaningful* axes (process constructs, understandability) rather than fitness or precision objectives, and the resulting hierarchy remains available as UCM stubs-plugins constructs in a model editor rather than as a process tree consumed only by mining tools.

Ghawi [26] also proposed combining process decomposition with the inductive-miner algorithm, but this was mainly done manually while targeting Petri Net models. PM4Py-UCM goes beyond this work by supporting decomposition automation via the extension of a popular library, and also via support for performer bindings.

C. Organizational and Role Mining

Song and van der Aalst systematized organizational mining as a sub-discipline of process mining [27] and introduced social-network discovery from event logs [28]. PM4Py-UCM differs in that performers are projected directly *into* the requirements model (as components in UCM), where they can provide requirements engineering with combined activity/performer insights (as discussed in Section IV-C, rather than being rendered as a separate organizational graph orthogonal to the process model. PM4Py-UCM’s support for a hybrid BPMN+components visualization also enables discussing the tool’s outputs with stakeholders familiar with BPMN but unfamiliar with UCM.

D. UCM and URN Tooling

To our knowledge, no prior tool offers a log-driven entry point into the UCM modeling and analysis chain. PM4Py-UCM offers that entry point, with stable round-trip engineering and mined model elements represented as first-class jUCMNav citizens. Prior work by Pourshahid et al. [29] has argued for URN as a (manual) business-process notation; PM4Py-UCM provides the empirical, automated, log-driven counterpart to that line of research.

VI. LIMITATIONS

This first version of PM4Py-UCM has many limitations that can lead to future work:

- 1) The inductive-miner algorithm is constraining process models to being well nested; this simplifies decomposition, but at the same time this may not reflect the complexity of non-well-behaved processes. Other algorithms such as split-miner [30] (unavailable in PM4Py) could likely relax that constraint while ensuring deadlock-free concurrency in the generated models, but at the cost of a more complex decomposition solution.

- 2) One activity could be performed by many roles or resources. The proposed aggregation strategies (Table V) are covering extreme cases (most frequent, first, all) but better trade-offs will be needed in practice. In particular, decomposition could allow sub-process activities to be mapped to components different from their parent maps. This would however require decomposition and performer-binding to be combined (they are performed separately in the current PM4Py-UCM).
- 3) The decomposition parameters and user modes are tightly coupled to the notion of process trees and might not be sufficient in practice. Already, with ad hoc testing on a number of event logs, the “aggressive” mode shows not to be aggressive enough. Better evidence-based fine-tuning of modes is needed.
- 4) Decomposition strategies are pragmatic (in their exploitation of process tree structures) but heuristic in nature. Clustering approaches [24,25] could lead to other types of decompositions that would be better based on the semantics of a process rather than solely on its structure. Proper decomposition metrics could also be used to compare their effectiveness [31].
- 5) UCM is a rich notation where some constructs (timers, failure points, dynamic stubs, component types, etc.) are not distinguished by the current approach. Investigating the feasibility and usefulness of inferring them from event logs is left to future work.
- 6) Boolean conditions on OR-forks in the UCM model are not inferred from the current PM approach, which limits the executability of mined models, e.g., for scenario execution. Decision mining [32,33] could potentially supplement the current approach to extract such conditions.
- 7) Over 150 unit and regression test cases were developed for PM4Py-UCM. However, the library and its Web interface have not yet been used by others. There is a need for proper usability testing and empirical validation, both of the API and its parameters and of the Web interface. Limitations in practical settings, especially around adoption, need to be properly identified and addressed.

VII. CONCLUSION

This paper proposed PM4Py-UCM, and extension of the popular PM4Py library as a means to support the Use Case Maps notation as an output of a common process mining algorithm, as well as enable the data-driven and automated creation of UCM models in a requirements engineering context. Four important contributions were defined in Section III and illustrated in Section IV:

- **C1.** An event-log-to-UCM discovery pipeline (Figure 1).
 - **C2.** Configurable decomposition strategies, producing hierarchical UCM models that can be visualized (using the UCM and BPMN syntaxes) and exported as PNG bitmaps (Figure 5).
 - **C3.** Configurable performer-aware bindings of activities to components (Figure 2), with performers extracted from logs or provided manually.
 - **C4.** A bi-directional importer/exporter that supports the jUCMNav file format (`.jucm`), enabling round-trip engineering (Figure 4).
- In addition to the library itself, the GitHub repository also contains documentation, a Jupyter Notebook tutorial on the tool’s API, and a deployed Web-based interface supporting common usages of the new library (Figure 3, <https://pm4py-ucm.streamlit.app/>).
- The many limitations identified in Section VI can lead to useful research areas focusing on a better exploitation of process mining in a requirements engineering context.

ACKNOWLEDGMENTS

This work is supported by an NSERC Discovery grant titled “Requirements-Oriented Process Mining”. The author acknowledges the use of Claude Code (by Anthropic), with its Opus 4.7 model, to assist in developing, debugging, and deploying the code used in PM4Py-UCM.

REFERENCES

- [1] W. van der Aalst, *Data Science in Action*. Springer, 2016, pp. 3–23, doi: 10.1007/978-3-662-49851-4_1.
- [2] ISO, “ISO/IEC 15909-1:2019 – Systems and software engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation,” 2019. [Online]. Available: <https://www.iso.org/obp/ui/#iso:std:iso-iec:15909:-1:en>
- [3] OMG, “Business Process Model and Notation (BPMN), version 2.0.2,” 2014. [Online]. Available: <https://www.omg.org/spec/BPMN/2.0.2>
- [4] M. L. van Eck, X. Lu, S. J. J. Leemans, and W. M. P. van der Aalst, “PM²: a process mining project methodology,” in *Advanced Information Systems Engineering, CAiSE 2015*. Springer, 2015, pp. 297–313, doi: 10.1007/978-3-319-19069-3_19.
- [5] C. A. Kesici, N. Ozkan, S. Taşkesenlioglu, and T. G. Erdogan, “A systematic literature review of studies comparing process mining tools,” *International Journal of Information Technology and Computer Science*, vol. 14, no. 5, pp. 1–14, 2022, doi: 10.5815/ijitcs.2022.05.01.
- [6] A. Berti, S. van Zelst, and D. Schuster, “PM4Py: A process mining library for Python,” *Software Impacts*, vol. 17, 2023, doi: 10.1016/j.simpa.2023.100556.
- [7] S. Lim, A. Henriksson, and J. Zdravkovic, “Data-driven requirements elicitation: A systematic literature review,” *SN Computer Science*, vol. 2, 2021, doi: 10.1007/s42979-020-00416-4.
- [8] J. Dąbrowski, F. M. Kifetew, D. Muñante, E. Letier, A. Siena, and A. Susi, “Discovering requirements through goal-driven process mining,” in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, 2017, pp. 199–203, doi: 10.1109/REW.2017.61.
- [9] M. Ghasemi, “What requirements engineering can learn from process mining,” in *2018 1st International Workshop on Learning from other Disciplines for Requirements Engineering (D4RE)*, 2018, pp. 8–11, doi: 10.1109/D4RE.2018.00008.
- [10] M. Ghasemi, D. Amyot, and W. Van Woensel, “Goal-oriented process mining: A scalability experiment,” in *2025 IEEE 33rd International Requirements Engineering Conference Workshops (REW)*, 2025, pp. 304–313, doi: 10.1109/REW66121.2025.00046.
- [11] S. I. Bouhidel, M. M. Bouhamed, G. Diaz, and N. Belala, “A requirement-driven method for process mining based on model-driven engineering,” *Computer Standards & Interfaces*, vol. 97, p. 104108, 2026, doi: 10.1016/j.csi.2025.104108.
- [12] ITU-T, “Recommendation Z.151 (10/2018): User requirements notation (URN) — language definition,” International Telecommunication Union, Tech. Rep., 2018. [Online]. Available: <https://www.itu.int/rec/T-REC-Z.151-201810-I/en>

- [13] R. J. A. Buhr, "Use case maps as architectural entities for complex systems," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, pp. 1131–1155, 1998, doi: 10.1109/32.738343.
- [14] D. Amyot, O. Akhigbe, M. Baslyman, S. Ghanavati, M. Ghasemi, J. Hassine, L. Lessard, G. Mussbacher, K. Shen, and E. Yu, "Combining goal modelling with business process modelling: Two decades of experience with the user requirements notation standard," *Enterprise Modelling and Information Systems Architectures (EMISAJ)-International Journal of Conceptual Modeling*, vol. 17, pp. 2:1–38, 2022, doi: 10.18417/emisa.17.2.
- [15] J.-F. Roy, J. Kealey, and D. Amyot, "Towards integrated tool support for the User Requirements Notation," in *System Analysis and Modeling: Language Profiles. SAM 2006*, ser. LNCS 4320. Springer, 2006, pp. 198–215, doi: 10.1007/119511.
- [16] M. T. Wynn, W. van der Aalst, E. Verbeek, and B. D. Stefano, "The IEEE XES standard for process mining: Experiences, adoption, and revision [society briefs]," *IEEE Computational Intelligence Magazine*, vol. 19, no. 1, pp. 20–23, 2024, doi: 10.1109/MCI.2023.3333141.
- [17] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from event logs – A constructive approach," in *Application and Theory of Petri Nets and Concurrency*, ser. LNCS, vol. 7927. Springer, 2013, pp. 311–329, doi: 10.1007/978-3-642-38697-8_17.
- [18] —, "Discovering block-structured process models from event logs containing infrequent behaviour," in *Business Process Management Workshops*, ser. LNBIP, vol. 171. Springer, 2014, pp. 66–78, doi: 10.1007/978-3-319-06257-0_6.
- [19] M. Khorasani, M. Abdou, and J. Fernández Hernández, *Streamlit for Web Development: Build and Scale Secure Python-Powered Apps with Streamlit*. Apress Berkeley, 2025. doi: 10.1007/979-8-8688-1826-4.
- [20] O. Akhigbe, D. Amyot, A. A. Anda, L. Lessard, and D. Xiao, "Consistency analysis for User Requirements Notation models," in *iStar 2016, Ninth International i* Workshop*, vol. CEUR-WS Vol-1674, 2016, pp. 43–48. [Online]. Available: https://ceur-ws.org/Vol-1674/iStar16_pp43-48.pdf
- [21] N. Kesserwan, R. Dssouli, J. Bentahar, B. Stepien, and P. Labrèche, "From use case maps to executable test procedures: a scenario-based approach," *Software and Systems Modeling*, vol. 18, no. 2, pp. 1543–1570, 2019, doi: 10.1007/s10270-017-0620-y.
- [22] M. Ghasemi and D. Amyot, "From event logs to goals: a systematic literature review of goal-oriented process mining," *Requirements Engineering*, vol. 25, no. 1, pp. 67–93, 2020, doi: 10.1007/s00766-018-00308-3.
- [23] B. Aysolmaz, H. Leopold, H. A. Reijers, and O. Demirörs, "A semi-automated approach for generating natural language requirements documents based on business process models," *Information and Software Technology*, vol. 93, pp. 14–29, 2018, doi: 10.1016/j.infsof.2017.08.009.
- [24] M. Song, C. W. Günther, and W. M. P. van der Aalst, "Trace clustering in process mining," in *Business Process Management Workshops*, ser. LNBIP, vol. 17. Springer, 2009, pp. 109–120, doi: 10.1007/978-3-642-00328-8_11.
- [25] M. Trabelsi, S. E. Boukhetta, D. Mondou *et al.*, "Towards explainable clustering in process mining and sequential pattern mining: A formal framework," in *Conceptual Knowledge Structures*. Springer, 2025, pp. 143–160, doi: 10.1007/978-3-032-03364-2_9.
- [26] R. Ghawi, "Process discovery using inductive miner and decomposition," 2016. [Online]. Available: <https://arxiv.org/abs/1610.07989>
- [27] M. Song and W. M. P. van der Aalst, "Towards comprehensive support for organizational mining," *Decision Support Systems*, vol. 46, no. 1, pp. 300–317, 2008, doi: 10.1016/j.dss.2008.07.002.
- [28] W. M. P. van der Aalst, H. A. Reijers, and M. Song, "Discovering social networks from event logs," *Computer Supported Cooperative Work (CSCW)*, vol. 14, no. 6, pp. 549–593, 2005, doi: 10.1007/s10606-005-9005-9.
- [29] A. Pourshahid, D. Amyot, L. Peyton, S. Ghanavati, P. Chen, M. Weiss, and A. J. Forster, "Business process management with the user requirements notation," *Electronic Commerce Research*, vol. 9, no. 4, pp. 269–316, 2009, doi: 10.1007/s10660-009-9039-z.
- [30] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and A. Polyvyanny, "Split miner: automated discovery of accurate and simple business process models from event logs," *Knowledge and Information Systems*, vol. 59, no. 2, pp. 251–284, May 2019, doi: 10.1007/s10115-018-1214-x.
- [31] F. Milani, M. Dumas, R. Matulevičius, N. Ahmed, and S. Kasela, "Criteria and heuristics for business process model decomposition: review and comparative evaluation," *Business & Information Systems Engineering*, vol. 58, no. 1, pp. 7–17, 2016, doi: 10.1007/s12599-015-0413-1.
- [32] E. Elhami, W. Van Woensel, and D. Amyot, "Discovering the why of process models – a systematic literature review on decision mining," *SSRN*, p. 6749580, 2026, doi: 10.2139/ssrn.6749580.
- [33] A. Rozinat and W. M. P. van der Aalst, "Decision mining in ProM," in *Business Process Management*. Springer, 2006, pp. 420–425, doi: 10.1007/11841760_33.