BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES,
DOI: 10.24425/bpasts.2025.155043

A Hierarchical Process Model Discovery Algorithm for CRS Systems

Andrzej Stroiński[1]* **Dariusz Dwornikowski**[2] **Anna Kobusińska**[1] **Jerzy Brzeziński**[1]

[1] Institute of Computing Science, Poznan University of Technology [2] Bitropy

Abstract. An increasing number of distributed systems are currently being developed according to the REST paradigm, supporting a diverse range of services and applications. However, analyzing potential errors and deviations in their operation has become progressively more difficult due to both the scale of processed data and the proliferation of available services. In this context, process mining offers a valuable approach. By analyzing event logs collected from such systems, it is possible to derive process models that represent application behavior in distributed environments. These models support the identification and remediation of errors as well as the optimization of system performance. This article introduces a mechanism for representing process models of Communication Resource Systems (CRS) developed within the REST paradigm using process algebra. In addition, we propose an algorithm for discovering such models, enabling the identification of both local processes executed within individual services and the interactions that occur between them.

Key words: Process mining; data discovery; process modelling; distributed systems; CRS systems

1. INTRODUCTION

RESTful Web services have become the dominant architectural choice for both emerging startups and major enterprises on the Web. Several factors contribute to this trend. The simplicity of REST lowers the entry barrier for service development, while its lightweight nature makes system management more straightforward compared to "heavy" SOAP-based systems (SOAP —Simple Object Access Protocol). Furthermore, RESTful web services align seamlessly with the fabric of the Web, as they are natively integrated with the ubiquitous HTTP protocol that underpins Internet communication. REST (Representational State Transfer) is also particularly well-suited to the development of both internal and external APIs. This is especially relevant in microservices architectures, where applications are composed of small, loosely coupled, and replaceable services that communicate through language-agnostic APIs. In such environments, microservices interact with one another through HTTP and RESTful APIs. In the following discussion, we refer to this class of systems, designed according to ROA (Resource-Oriented Architecture) or REST principles, as Communicating Resource Systems (CRS).

CRSs are inherently granular, API-driven, and composed of numerous independent services. Their complexity does not primarily stem from the functionality of individual components but from the intricate interactions, dependencies, and coordination across many small and relatively simple services via APIs. This phenomenon can be described as a "complexity shift". While the behavior of a single service may be understandable and verifiable against its design, assessing the system as a whole raises more difficult questions. From a global perspective, determining whether the system behaves according to its intended design or whether its overall behavior conforms to the APIs it exposes, becomes a significant challenge.

We argue that process mining techniques, particularly conformance checking [1], offer a promising direction for addressing this challenge. Conformance checking focuses on identifying deviations between the observed system behaviour, reconstructed from event logs, and a predefined reference model. Such a model may be provided directly by an expert or derived from API specifications and documentation. However, existing process mining approaches [1], including both process discovery and conformance checking, generally operate under the assumption that the process of interest is flat (a so-called 2D analysis). In this view, even if the process is composed of multiple sub-processes, these remain indistinguishable and are not explicitly represented in the discovered model. While this assumption is often adequate for enterprise-level business processes, it becomes limiting when analyzing systems characterized by inherent process composition or hierarchical structures. Recently, the emergence of object-centric event logs (OCEL) [2] has drawn increasing attention, together with more advanced discovery techniques capable of exploiting such data [3–6]. These methods enable the capture of more complex and multidimensional processes. Nevertheless, they still lack explicit semantics for describing communication patterns and process composition, which are crucial for understanding the behaviour of distributed and service-oriented systems.

For Communicating Resource Systems (CRS), more suitable modelling approaches are required. These systems consist of numerous small processes, hierarchically composed and interacting to realize larger, global business processes. Consequently, it is essential to adopt modeling techniques capable of capturing their distinctive characteristics, namely compositionality, modularity, the presence of subprocesses, and communication. In our previous work on process mining for communicating systems built on REST principles [7, 8], we demonstrated that classical Petri Nets are inadequate for fully representing such systems. Specifically, Petri Nets lack compositionality, are inherently flow-oriented rather than behavior-oriented, and cannot directly model communication, which is

*e-mail: andrzej.stroinski@put.poznan.pl

a fundamental building block of every distributed system.

We therefore argue that symbolic process calculi are better suited for modelling CRSs. By design, they are compositional, naturally support the representation of subprocesses, emphasize communication and behavioral aspects, and can be readily adapted to specific domains. Collectively, these features reduce the representational bias of models employed in process mining [1, 9].

In this article, we introduce *ROC*, a process calculus designed to model Communicating Resource Systems (CRS), including distributed systems based on REST, RESTful business processes, ROA systems, Web Services, and related architectures. *ROC* represents a natural evolution of the earlier RA_s algebra proposed in [10], which provided an elementary calculus but lacked mechanisms for data passing and explicit HTTP communication. In contrast, *ROC* is fully equipped to represent systems that rely on HTTP as their underlying protocol. We further propose the AMA-WC (Algebra Miner Algorithm with Communication) algorithm, which enables the discovery of CRS process models from their execution histories. Such histories are typically captured in the event logs of CRS components (e.g., services). Unlike an ideal model, which represents the intended system behaviour, the discovered model reflects the actual behaviour observed during execution. The AMA-WC algorithm derives process models expressed in the *ROC* calculus.

The remainder of this article is organized as follows. Section 2 provides a brief overview of related work. Section 3 introduces the *ROC* calculus, presenting its syntax, semantics, and illustrative examples. Section 4 describes the AMA-WC algorithm for discovering *ROC* models from event logs. The evaluation of the proposed algorithm and analysis of the obtained results is presented in Section 5. Finally, Section 6 concludes the paper with a summary of contributions.

2. RELATED WORK

Regarding conformance checking, no prior work explicitly addresses resource-oriented systems such as REST or ROA. Existing studies mainly concern SOAP-based Web Services. For example, [11] extracts behavioural models from BPEL definitions for comparison with logs, though the approach lacks hierarchy and explicit communication. Related work verifies choreography in SOA systems [12], uses reference architectures [13], or applies event calculus [14]. More recent contributions, such as [15], propose constraint-based conformance with CxWSDL. However, these approaches remain tied to SOAP and emphasize the flow perspective. Privacy-aware federated conformance checking [16] enables cross-organizational analysis but similarly neglects the resource perspective.

Process calculi for SOA systems have also been widely studied [17–23], though these treat services as black boxes and lack support for hierarchical composition. A comprehensive survey is available in [21]. For resource-based systems, the most relevant work is [24], which applies temporal logic to verify RESTful properties, but without behavioural modeling or hierarchy.

Process mining has been applied in distributed systems, but few algorithms explicitly address hierarchical and communicative processes. In [25], the authors propose a method for distributed discovery of CRS models by analyzing component interactions, experimentally demonstrating its effectiveness in uncovering communication structures. Applications to Web services are explored in [26], which highlights the use of event logs for process design and monitoring, and in [27], which introduces a hybrid genetic service mining (HGSM) method for large-scale, distributed logs.

Adaptation to resource-oriented systems is discussed in [7], which emphasizes the need to extend SOAP-focused methods to RESTful architectures, presenting initial work on process discovery in this domain. The problem of evolving Web APIs is addressed in [28], where usage-log mining reveals behavioural patterns guiding API evolution, with applications in education and healthcare. Finally, [29] proposes hierarchical process tree discovery, distinguishing business from event logs, emphasizing the importance of identifying hierarchical relationships that arise from task dependencies during execution. The proposed method, based on hierarchical process trees, supports the discovery of models with subprocess relationships. However, this work focuses solely on hierarchical structures, without addressing communication or message-passing aspects.

Several studies propose domain-agnostic algorithms for process discovery, but they fail to capture the hierarchical dependencies characteristic of CRS systems. In [30], different Petri net classes are evaluated for modelling collaboration processes, with a meta-model and design guidelines proposed. Building on this, [31] presents Collaboration Miner (CM), which identifies collaboration processes directly from event logs without predefined assumptions. Another approach, fuzzy mining [32], simplifies unstructured models by using adaptive, multi-perspective views, which apply to domains such as healthcare, logistics, and web services.

In addition, [33] surveys methods for handling complex processes but does not address compositionality or inter-process communication.

In summary, existing work advances conformance checking and process discovery in SOAP-based or domain-agnostic settings, but a gap persists for CRS and REST systems. Current approaches are limited to Petri nets or flow-oriented models, highlighting the need for algebra-based methods as a foundation for new frameworks in process analysis and conformance checking.

3. RESOURCE ORIENTED CALCULUS (*ROC*)

The primary entities in CRSs are resources, which are accessed through method invocations transmitted over a communication protocol. Upon invocation, a resource executes a sequence of activities that constitute its behaviour or process, defining the actions required in response to a request. Consequently, CRS resources are inherently reactive: they engage in activities only when explicitly invoked. In practical terms, resources can be understood as programs that are executed upon method calls.

Figure 1 presents a graphical representation of the CRS model. The boxes are resources, and the arcs represent communication among them. The primary characteristic of the system is that resources are hierarchical, meaning that resources can be placed within other resources. Some distinguished resources are *top resources*, e.g. *id1,id3,id4*, some of them are *sub-resources* of other resources, e.g. `id2` in `id1`, `id7` in `id6`. In the latter example, `id2` is also a direct sub-resource of `id1` because it is directly underneath it in the hierarchy. *Sub-resources* cannot belong to more than one top resource. In addition, top resources and all resources belonging to the same "level" in one enclosing top resource have to be uniquely named. So, in CRS, resources are uniquely addressable, and every resource has its name or id, e.g. `id1`. Hierarchy in addressing is expressed similarly to the URI standard as paths. In the given example, the path for the resource `id7` would be `id5/id6/id7`.

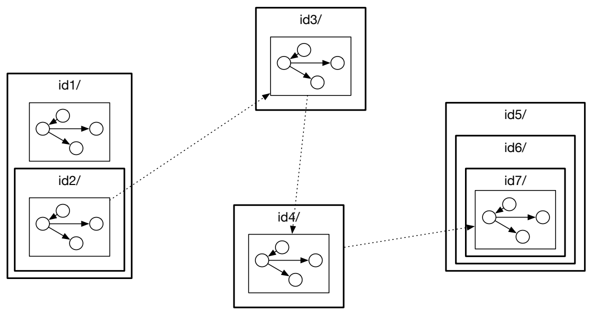


Fig. 1. Model of a communicating resource system

The communication model of CRS is synchronous, analogous to HTTP, meaning that a corresponding response is immediately sent after every request. Accordingly, each communication activity in CRS is represented as a request–response pair. Communication is both directed and addressed: when a resource issues a request to another resource, it must explicitly identify the target resource along its path. For example, a resource `/client/` may send a request *req* to a resource `/user/`. Since resources in CRS are reactive, system execution is always initiated by an external message. The originator of this initiating message is referred to as the client. A client may either be an internal component of the CRS (i.e., another resource) or an assumed external entity.

In the following, we formally define the CRS model and its properties.

Def. 3.1 ***CRS model***
CRS is a tuple $\langle R, I, A, \alpha, \eta, \theta, \lhd \rangle$, *where:*
R *— is a set of resources,*
I *— is a set of resource identifiers (e.g. URI address).*
A *— is a set of all activity names that resources can execute.*
α *— is a relation that assigns activities (*A*) that can be performed by a resource* r, $\alpha \subseteq (R \times A)$.
η *— is a surjective function* $\eta : R \rightarrow I$ *that assigns identifiers from* I *to resources from* R.
θ *— is a surjective function* $\theta : OPS \rightarrow RETS$ *that assigns return codes from a set of response codes* $RETS$ *to request methods activities from* $OPS \subseteq A$.
$\lhd$ *— is strict partial order hierarchy relation* $\lhd$ *where* $\lhd \subseteq (R \times R)$.

The model in Def. 3.1 was loosely inspired by [24], in addition to it, let us denote $id(r) = i$ for $(r,i) \in \eta$ and $r \in R, i \in I$. The hierarchy is expressed by the relation $\lhd$, and based on that, we will now define concepts related to and resulting from that: *sub-resource* and *super resource* (Def. 3.2), *Top resource* (Def. 3.3), *Unique belonging* (Def. 3.4), *Direct sub-resource* and *Direct super resource* (Def. 3.5).

Def. 3.2 ***Sub-resource and super resource***
Whenever $(r_1, r_2) \in \lhd$ *for some* $r_1, r_2 \in R$ *and* $r_1 \neq r_2$, *we say that* r_1 *is a* ***sub-resource*** *of* r_2, *and we write* $r_1 \lhd r_2$. *We will call* r_2 *a* ***super resource*** *of* r_1.

Def. 3.3 ***Top resource***
A resource t *is a* ***top resource*** *if* $\forall r \in R : (t, r) \notin \lhd$. *A set of top resources is denoted as* $\mathscr{T}$, $\mathscr{T} \subseteq R$.

Def. 3.4 ***Unique belonging***
Unique belonging *denotes that* $\forall r_1, r_2, r_3 \in R$ *if* $r_1 \lhd r_2$ *and* $r_1 \lhd r_3$ *and* $r_2, r_3 \in \mathscr{T}$ *then* $r_2 = r_3$.

Def. 3.5 ***Direct sub-resource***
$\forall r_1, r_2 \in R$*: if* $r_1 \lhd r_2, r_1 \neq r_2$ *and* $\nexists r_3 \in R$*:* $r_1 \lhd r_3 \lhd r_2$, *then we call* r_1 *a direct sub-resource of* r_2 *and write* $r_1 \lhd_1 r_2$

Def. 3.6 ***Top resource naming uniqueness***
$\forall r_1, r_2 \in \mathscr{T}, r_1 \neq r_2$ *and* $i_1, i_2 \in I$, *if* $id(r_1) = i_1, id(r_2) = i_2, r_1 \neq r_2$ *then* $i_1 \neq i_2$

Def. 3.7 ***Direct super resource naming uniqueness***
$\forall r_1, r_2, r_3 \in R$, *if* $r_1 \lhd_1 r_3$ *and* $r_2 \lhd_1 r_3$ *and* $r_1 \neq r_2$ *and* $id(r_1) = i_1$ *and* $id(r_2) = i_2$ *then* $i_1 \neq i_2$

Def. 3.8 ***Resource addressing***
Let us define a path function:

$$path(r_1) = \begin{cases} id(r_1) & \text{if } r_1 \in \mathscr{T} \\ path(r_2) \cdot \text{"/"} \cdot id(r_1) & \text{if } r_1 \leq_1 r_2 \end{cases}$$

where $\cdot$ *is a string concatenation.*

A communication in CRS can be defined as a quadruple:

Def. 3.9 ***Communication in CRS.***
A communication in CRS is a tuple $\langle path(R_s), path(R_d), o, r \rangle$, *written* $R_s \xrightarrow{o,r} R_d$, *where:*
$path(R_S)$ *— is the path address of the requesting resource, called source resource (*$R_S \in R$*),*
$path(R_d)$ *— is the path address of the requested resource called destination resource (*$R_d \in R$*),*
$o \in OPS$ *— is the operation invoked by* R_s *on* R_d,
$r \in RETS$ *— is a return code to the operation* o,

An example of a communication in CRS, as defined in Def. 3.9, could be $/client/ \xrightarrow{READ,OK} /counters/counter$. In this hypothetical example, some resource `client` sends `READ` request to the `/counters/counter`, resulting in a return code `200 OK`.

In the case of *ROC*, the sets *OPS* and *RETS* are fixed to correspond to the methods and return codes of the HTTP protocol. Specifically, *OPS* includes the standard HTTP methods, e.g., $GET, POST, PUT, DELETE, HEAD$. For brevity, we do not enumerate the full set of 36 HTTP return codes as specified in RFC7231; instead, we assume that *RETS* encompasses all of them (e.g., 202 Accepted, 200 OK, 404 Not Found, 301 Moved Permanently, etc.).

ROC introduces a possibility of simple data passing in a communication. For this, we employ notation inspired by that in [20]. Let *Var* be a set of all variables ranged over by *x, y, z* and *Val* be a set of values over by *v*. In order to represent tuples we exploit notations $\vec{x} = \langle x_0, x_1, \ldots, y_i \rangle$ and $\vec{y} = \langle y_0, y_1, \ldots, y_i \rangle$. To access a particular variable in a tuple, we exploit $\rightarrow$ notation, e.g variable x_2 in $\vec{x}$ is $\vec{x} \rightarrow x_2$.

A hierarchical nature of resource-based systems is expressed by the id guard $@l$, which allows to embedding other guards, making it possible to express $P := l[m[n[\ldots]]]$. In the example, l is a top resource id ($id(t_0) = l$ for some $t_0 \in \mathscr{T}$), $@m, @n$ are sub-resource ids ($id(r_0) = m, id(r_1) = n, r_0, r_1 \in R, r_0 \leq_t t_0, r_1 \leq_t t_0, r_1 \leq_1 r_2$). Now, let's define the syntax of the *ROC* using BNF grammar:

$$
\begin{aligned}
&a, b, c, \dots \in A, \quad \bar{o} \in OPS, \\
&r \in RETS \quad l, m, n, \dots \in I \quad x, y, z \in Var \\
&P, Q ::= @l[P] \,|\, \mathbf{stop} \,|\, P + Q \,|\, P \,||\, Q \,|\, a.P \,|\, comm \\
&comm ::= \bar{o}@m(\vec{x}, \vec{y}).P \,|\, o_{\downarrow}(\vec{x}).P \,|\, \overline{o_{\uparrow r}}(\vec{x}).P
\end{aligned}
$$

stop denotes the null process, i.e., a process that exhibits no behaviour. The operator $P + Q$ represents alternative composition (choice), meaning that either P executes or Q. The prefixing construct $a.P$ indicates that the process first executes action a and then continues as P. For brevity, we permit omission of **stop** in terminating processes; for example, $a.b.c$ is syntactically equivalent to $a.b.c.\mathbf{stop}$. Finally, $P \,||\, Q$ denotes parallel composition, where the executions of P and Q proceed in an interleaved manner.

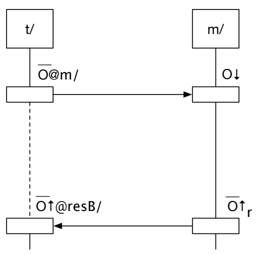


Fig. 2. Communication in *ROC*

SEQ

$$
\frac{}{t[a.P \xrightarrow{a} stop]} \qquad \frac{}{t[a.P \xrightarrow{a} P]}
$$

ALT

$$
\frac{t[P \xrightarrow{a} P']}{t[P+Q \xrightarrow{a} P']} \qquad \frac{t[Q \xrightarrow{a} Q']}{t[P+Q \xrightarrow{a} Q']}
$$

$$
\frac{t[P \xrightarrow{a} stop]}{t[P+Q \xrightarrow{a} stop]} \qquad \frac{t[Q \xrightarrow{a} stop]}{t[P+Q \xrightarrow{a} stop]}
$$

PAR

$$
\frac{t_i[P \xrightarrow{a} P']}{t_i[P] \,||\, t_j[Q] \xrightarrow{a} t_i[P'] \,||\, t_j[Q]} \; i \neq j
$$

$$
\frac{t_i[Q \xrightarrow{a} Q']}{t_i[P] \,||\, t_j[Q] \xrightarrow{a} t_i[P] \,||\, t_j[Q']} \; i \neq j
$$

$$
\frac{t_j[Q \xrightarrow{a} stop]}{t_i[P] \,||\, t_j[Q] \xrightarrow{a} t_i[P]} \; i \neq j \qquad \frac{t_i[P \xrightarrow{a} stop]}{t_i[P] \,||\, t_j[Q] \xrightarrow{a} t_j[Q]} \; i \neq j
$$

COM

$$
\frac{}{t[\bar{a}@m(\vec{x},\vec{y}).P \xrightarrow{\bar{a}@m(\vec{v}/\vec{x},\vec{y})} a_{\uparrow}@m(\vec{y}).P]} \; m \not\leq t
$$

$$
\frac{t_i[P \xrightarrow{\bar{a}@m(\vec{v}/\vec{x})} P'] \quad t_j[Q \xrightarrow{a_{\downarrow}(\vec{v}/\vec{x},\vec{y})} Q']}{t_i[P] \,||\, t_j[Q] \xrightarrow{\tau} t_i[P'] \,||\, t_j[Q']} \; m \leq t_j, i \neq j
$$

$$
\frac{t_i[P \xrightarrow{a_{\uparrow}@m(\vec{y})} P'] \quad t_j[Q \xrightarrow{\bar{a}_{\uparrow r}(\vec{x})} Q']}{t_i[P] \,||\, t_j[Q] \xrightarrow{\tau} t_i[P'] \,||\, t_j[Q']} \; m \leq t_j, i \neq j
$$

Fig. 3. Transition rules for *ROC*

Communication in *ROC* is inspired by HTTP and is therefore synchronous: each request is blocked until a corresponding response with a return code is returned. The receiving process may perform arbitrary internal computations between receiving the request and issuing the response. This mechanism enables *ROC* to represent subprocesses and orchestration, where one process invokes another and thereby initiates an entire processing chain. $\bar{o}@m(\vec{x}, \vec{y})$ is a request that sends an operation and is blocked until a response comes, to a response getting operation $\overline{o_{\uparrow}}@m$ operation. m is an "address" of the requested remote resource (e.g. */patients/*, or */user/111*); $\vec{x}$ is a request structure, with request body and headers, whereas $\vec{y}$ is where the response structure (again body and headers, and a response code) will be written to. $o_{\downarrow}(\vec{x})$ is where a request is received and $\overline{o_{\uparrow r}}(\vec{z})$ is the response to the request with r return code, which will finally unblock the sender. We will exploit _ to denote an empty data structure. A simplified communication diagram can be seen in Figure 2. Data structures are omitted for simplicity.

Figure 3 presents Labeled Transition Systems (LTS) semantics of *ROC* calculus, $t_0, t_1 \in \mathscr{T}, a \in A, o \in OPS, r \in RETS$. Most rules state that transitions can happen in a single top resource. Within it, however, behaviour is still free to shift from one sub-resource to another. SEQ are rules governing prefixing, ALT rules deal with the situation where a process makes a non-deterministic choice inside a top resource. PAR rules define behaviour for parallel composition similarly to ALT rules, considering all possible cases. Finally, the COM rule covers communication, where two processes in two different resources communicate on a channel, resulting in a silent activity defined as τ.

Example 3.1 presents a simple load balancer that, upon an incoming request *GET* from some assumed Client (not covered in this example), directs the request to one of two servers, $serv_1$ or $serv_2$.

Example 3.1 *A simple load balancer*

$$LB := lb[GET_{\downarrow}(\_).\overline{GET}@serv_1(\_, resp).save(resp).\overline{GET_{\uparrow 200\ OK}}(resp2)$$
$$+ GET_{\downarrow}(\_).\overline{GET}@serv_2(\_, resp).save(resp).\overline{GET_{\uparrow 200\ OK}}(resp2)]$$
$$Server_1 := serv_1[GET_{\downarrow}(\_).process.\overline{GET_{\uparrow 200\ OK}}(resp)]$$
$$Server_2 := serv_2[GET_{\downarrow}(\_).process.\overline{GET_{\uparrow 200\ OK}}(resp)]$$
$$System := Client \,||\, LB \,||\, (Server_1 \,||\, Server_2)$$

4. *ROC* ALGEBRA MODEL DISCOVERY

In this section, we introduce the Algebra Miner Algorithm with Communication Mining (AMA-CM), an extension of Guido Schimm's algorithm (GS) [34, 35]. The algorithm comprises two main components: (i) the discovery of local processes (AMA-WC-LOC, Section 2), and (ii) their composition into a global process through communication (AMA-WC-GLB, Section 3). Below, we briefly outline the key differences between AMA-WC and the original GS algorithm:

- Whereas GS discovers models in a general-purpose process algebra, AMA-WC is tailored to the more expressive *ROC* calculus. This enables explicit representation of process composition via inter-process communication (see Section 3); for example, task delegation can be modelled as request–response interactions.
- AMA-CM (Section 3) yields a compositional system model: multiple interacting local process models are composed into a global model. Consequently, it identifies both process composition and the communication between processes.
- AMA-WC operates on the *ROC* calculus using an extended graph notation, formally captured by communication blocks with explicit sender and receiver nodes (see Section 3).
- GS assumes lifecycle-aware logs (explicit start/end events for each activity) to infer concurrency. In contrast, AMA-WC accommodates the reality that most CRS logging (and typical service frameworks, e.g., [36–39]) records point events at occurrence time only. Accordingly, AMA-WC introduces alternative concurrency definitions (see Def.s 4.1, 4.2). In addition, we formalize algebra graphs and present a step-by-step construction method.

1. *Formal notation and theoretical foundations*

To describe the event log, we exploit the notation from [40]. L is an event log of a system, and is a multi-set of event sequences (σ), e.g. $L = \{\sigma_1^7, \sigma_2^2, \ldots, \sigma_n^m\}$, where the n-th sequence occurs m times in L. Event sequences consist of ordered events (e), e.g. $\sigma = \langle e_1, e_2, e_3, e_4, e_5 \rangle$. Furthermore, each event can consist of multiple data fields, where one is distinguished as an identifier of the event. We assume the id's uniqueness in L. To select a specific event field, we use $\#_{field\_name}(e)$, e.g. $\#_{id}(e_1) = first\_event$, means event e_1 has *id* set to $first\_event$. We distinguish two types of events: local resource events (LRE), and communication resource events (CRE). The difference between them is the minimal set of available attributes that must be recorded. LRE needs information about the executing resource, as well as the event id. In addition, if the ordering relation in the event log is not fulfilled, then timestamps must also be present. CRE, in addition to attributes present in LRE, needs information about the source and destination of communication. Such data is extremely difficult to get in practice; however, an approach to this problem can be found in our previous work [7].

2. *PART 1: Algebra Miner Algorithm with communication mining — local process discovery*

In the GS algorithm, it is assumed that the event log is represented in interval form. In this representation, each activity is described by two partial events: a start event, which marks the initiation of the activity, and an end event, which marks its completion. Interval logs facilitate straightforward detection of parallelism: informally, if the time intervals of two activities overlap, the activities are considered parallel. Consequently, for two activities to be classified as parallel, the conditions specified in Def. 4.1 must be satisfied.

Def. 4.1 (Interval activity parallel relation) *Let* $a, \; b \in E$ *(where* E *are events set in* L*),* $a||b \iff \#(a)_{start} \succ \#(b)_{start} \succ \#(a)_{end} \vee \#(a)_{start} \succ \#(b)_{end} \succ \#(a)_{end} \vee \#(a)_{start} \succ \#(b)_{start} \succ \#(b)_{end} \succ \#(a)_{end}$
In addition: $a||b = b||a$

This relation, however, does not apply to CRS systems due to the way modern application servers record logs. Typically, each resource generates only a single log entry per invocation, containing information about the request sent and the response received from the remote resource. The same limitation applies to standard local application logs, which typically do not record event durations or start–end event pairs. Consequently, it is necessary to adopt the classical ordering relation introduced in [40] (Def. 4.2) and apply it to the generation of trace clusters. Under this relation, two events a and b are considered parallel if, in at least one trace, a directly precedes b, and in at least one other trace, b directly precedes a.

Def. 4.2 (Classical activity parallel relation) *Let* $a, \; b \in E$, $a||b \iff a \succ_L b \wedge b \succ_L a$ *(where* E *are events set in* L*)*

Listing 1 AMA-WC: discovery of local process

function AMA-WC-LOC(L)
$el = remove\_redundant\_traces(L)$
$init\_clusters = gen\_init\_clusters(el)$
$ord\_set = gen\_ord\_set(el)$
$pll\_set = gen\_pll\_set(ord)$
$clusters = gen\_pll\_clusters(init\_clusters, pll\_set)$
$clusters = gen\_init\_algebra\_model(clusters)$
$algebra = reduce\_model(clusters)$

With this background, we now introduce the general idea of AMA-WC-LOC, illustrated in Listing 1. The procedure begins by processing the event log L to remove redundant traces, i.e., traces that occur more than once. The result of this step is an event log without repeated traces, denoted as el (Figure 4, l. 2). In the second step, initial trace clusters are generated based on all non-redundant traces (Figure 4, l. 3). A simple implementation of this procedure is presented in Listing 2.

In this implementation, we iterate over traces in the event log (Listing 2, l. 3), creating a new Node for each event (l. 7). For each trace, a new cluster is generated (l. 4). If the current event is the first event of the trace (l. 8), the node is added to the cluster (l. 9). For subsequent events, we update the outgoing nodes of the previous node (*prev.out*) and the incoming nodes of the current node (*curr.in*) accordingly (l. 11–13). Finally, the completed cluster is added to the set of clusters (l. 15).

Listing 2 Generating initial trace clusters

```
1:  function GEN_INIT_CLUSTERS(el)
2:      clusters = set()
3:      for all σ ∈ el do
4:          curr_cluster = new cluster()
5:          prev = None
6:          for all e ∈ σ do
7:              curr = new Node(e)
8:              if e == first(σ) then
9:                  curr_cluster.addNode(curr)
10:             else
11:                 prev.out.add(curr)
12:                 curr.in.add(prev)
13:                 curr_cluster.addNode(curr)
14:             prev = curr
15:         clusters.add(curr_cluster)
16:     return clusters
```

The next step of AMA-WC-LOC (Listing 1, l. 4) involves computing the set of tuples (a,b), where each tuple denotes that event a directly precedes event b in at least one trace. Based on this set (ord_set) and Def. 4.2, the algorithm derives the set of parallel events (pll_set) (l. 5). Subsequently, in l. 6, parallel clusters are generated and redundant ones are removed. As shown in Listing 3, the algorithm iterates over previously generated clusters (l. 2) and the ordering relations within those clusters (l. 3). The purpose of this iteration is to determine whether the dependency between two events represents a sequential or a parallel relation. Following Def. 4.2, if the reverse ordering relation is found in other traces (and thus in other clusters), the two events are recognized as parallel. Accordingly, in l. 4, the algorithm checks whether the events are parallel; if so, a new cluster is created where the tested events are connected by a parallel relation rather than a sequential one (l.s 5–9). The original cluster is then removed.

An example of this transformation is presented in Figure 4, l. 6. In this case, clusters emphB and C are both transformed into their parallel counterparts, B' and C'. Since B' and C' are identical, the duplicates are removed, leaving a single cluster, denoted as D.

In l. 9 of Listing 3, redundant duplicate clusters are removed.

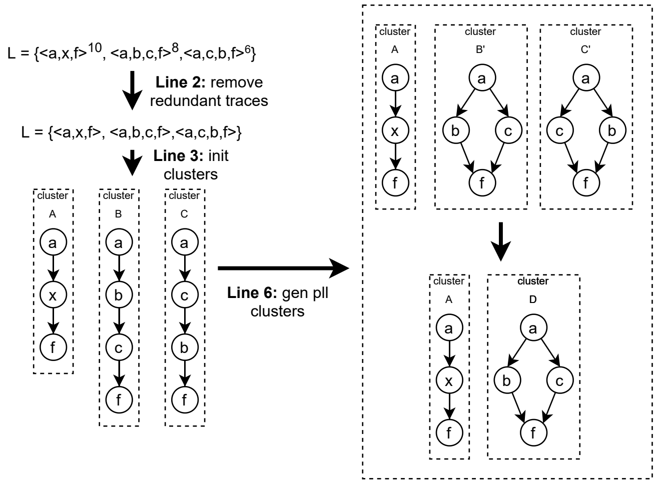


Fig. 4. AMA-WC-LOC: preparing clusters for initial algebra

Listing 3 Generate parallel clusters

```
1:  function GEN_PLL_CLUSTERS(clusters, pll_set)
2:      for all c ∈ clusters do
3:          for all e1 ≻ e2 ∈ c do
4:              if (e1, e2) ∈ pll_set then
5:                  c' = c.copy()
6:                  e'1.out = e'2.out
7:                  e'2.in = e'1.in
8:                  clusters.add(c')
9:                  clusters.del(c)
        clusters = remove_duplicate_clusters(clusters)
10:     return clusters
```

Finally, in l.s 7 and 8 of Listing 1, a workflow algebra model is discovered from the generated clusters. These steps correspond directly to Steps 4 and 5 of the original GS algorithm [34] and remain unchanged. However, for the sake of completeness, we briefly describe their implementation in our framework using algebra graphs and present the semantics of this structure for subsequent use. To this end, we introduce several definitions of the workflow algebra model: 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, and 4.10.

Def. 4.3 (Algebra graph) *Alpha graph AG is a tuple* $\langle N, E\rangle$, *where N is a set of algebra graph nodes and E is a set of directed edges.*

Def. 4.4 (Algebra graph node) *An algebra graph node is a tuple* $\langle id, IN, OUT, RELATED\rangle$, *where id is a unique node identifier, IN is a set of input nodes, OUT is a set of output nodes, and RELATED is a node related to the current node.*

Def. 4.5 (Algebra simple node (sequence)) *Algebra sequence node is a tuple seq =* $\langle id, IN, OUT, RELATED\rangle$, *where* $|IN| = 1$, $|OUT| = 1$, $|RELATED| = 0$. *The semantics of this node is as follows: execute this node action, when iff all seq.IN nodes have already been executed and go to the next (n.OUT) node.*

Def. 4.6 (Algebra parallel start node (APSN)) *Algebra parallel start node is a tuple* $pll_{start} = \langle id, IN, OUT, RELATED\rangle$, *where* $|IN| = 1$, $|OUT| = x$ $(x \geq 1)$, *RELATED = n (n is APEN node). The semantics of this node is as follows: execute this node action, when* *iff all pll_{start}.IN nodes have already been executed and go to the all (pll_{start}.OUT) node at the same time.*

Def. 4.7 (Algebra parallel end node (APEN)) *Algebra parallel end node is a tuple $pll_{end} = \langle id, IN, OUT, RELATED\rangle$, where $|IN| = x$ and $(x \geq 1)$, $|OUT| = 1$, RELATED = n (n is APSN node). The semantics of this node is as follows: execute this node action, when iff all pll_{end}.IN nodes have already been executed and go to the next (pll_{end}.OUT) node.*

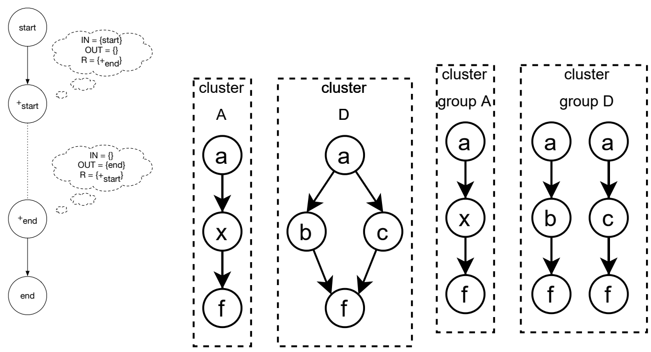
Def. 4.8 (Algebra alternative start node (AASN)) *Algebra alternative start node is a tuple $alt_{start} = \langle id, IN, OUT, RELATED\rangle$, where $|IN| = 1$, $|OUT| = x$ $(x \geq 1)$, RELATED = n (n is AAEN node). The semantics of this node is as follows: execute this node action, when iff all alt_{start}.IN nodes have already been executed and go to only one node from the set $alt_{start}.OUT$.*

Def. 4.9 (Algebra alternative end node (AAEN)) *Algebra alternative end node is a tuple $alt_{end} = \langle id, IN, OUT, RELATED\rangle$, where $|IN| = x$ and $(x \geq 1)$, $|OUT| = 1$, RELATED = n (n is AASN node). The semantics of this node is as follows: execute this node action, when iff only one node from set alt_{end}.IN nodes have already been executed and go to the next (alt_{end}.OUT) node.*

Def. 4.10 (Algebra graph edge) *Alpha graph edge is a pair (a, b), where $a \in E \wedge b \in E \wedge a \in b.IN \wedge b \in a.OUT$*

Listing 4 Generate initial algebra model

1: **function** GEN_INIT_ALGEBRA_MODEL(*clusters*)
2: alg = new Algebra()
3: alg.N += new Node(*start*, ∅, ∅, ∅)
4: alg.N += new Node($+_{1_{start}}$, ∅, ∅, ∅)
5: start.OUT += $+_{1_{start}}$
6: $+_{1_{start}}$.IN += *start*
7: alg.E += (start, $+_{1_{start}}$)
8: alg.first = start
9: alg.N += new Node(*end*, ∅, ∅, ∅)
10: alg.N += new Node($+_{1_{end}}$, ∅, ∅, ∅)
11: end.IN += $+_{1_{end}}$
12: $+_{1_{end}}$.OUT += end
13: alg.E += ($+_{1_{end}}$,end)
14: alg.last = end
15: $+_{1_{end}}$.R = $+_{1_{start}}$
16: $+_{1_{start}}$.R = $+_{1_{end}}$
17: clusters_groups += gen_all_traces(clusters)
18: **for all** $cg \in clusters\_groups$ **do**
19: $||_{cg.idx_{start}}$ = new Node($||_{cg.idx_{start}}$, $\{+_1\}$, ∅, ∅)
20: $+_1$.OUT += $||_{cg.idx_{start}}$
21: $||_{cg.idx_{start}}$.IN += $+_1$
22: alg.E += ($+_1$, $||_{cg.idx_{start}}$)
23: **for all** $c \in cg$ **do**
24: seq = gen_seq_events(c)
25: alg.N += nodes(seq)
26: alg.E += edges(seq)
27: first(seq).IN += $||_{cg.idx_{start}}$
28: $||_{cg.idx_{start}}$.OUT += first(seq)
29: alg.E += ($||_{cg.idx_{start}}$,first(seq))
30: last(seq).OUT += $||_{cg.idx_{end}}$
31: $||_{cg.idx_{start}}$.IN = last(seq)
32: alg.E += (last(seq),$||_{cg.idx_{end}}$)
33: **return** *clusters*



(a) first steps of alg. generation **(b)** before generation clusters groups **(c)** after generation clusters groups

Fig. 5. Graphical notation of clusters

The initial algebra model is constructed from the clusters obtained in Step I (Listing 4). In line 2, a new algebra instance is created. Subsequently, the starting node of the discovered process (l. 3) and the first node (an alternative node) are generated. The purpose of this construction is to associate each possible ordering of events, as derived from the clusters, with an alternative node. This is realized by creating the node (l. 4. The ordering between these two nodes is then established by an edge (l. 7) and recorded in the appropriate sets (l. 5 and 6). Analogous operations are performed for the termination nodes of both the alternative node and the process (l. 9–14). Specifically, the process end node and the closure of the alternative node are added. Additionally, lines 15–16 record relationships between nodes, specifying which nodes serve as closures of others. The current state of the algebra model at this stage is illustrated in Figure 5a.

All possible traces are generated from the clusters (l. 17). These traces are additionally grouped by their originating clusters, emphasizing that they correspond to parallel executions of events. The outcome of this step is illustrated in Figures 5b and 5c, where the left side depicts clusters before execution and the right side shows the clusters after execution. For each cluster group, a parallel node is then generated (l. 19) and connected to the alternative node (l. 20–22). Each cluster is subsequently translated into a sequence of events (l. 24), resulting in a sequence of sequence nodes. The function *first* returns the initial node of the sequence, while *last* returns the final node. These nodes are then connected to the respective parallel start and end nodes. Finally, an ending alternative node is created and linked to the parallel end nodes of all clusters, followed by the creation of the global process end node. The resulting algebra graph is presented in Figure 6a.

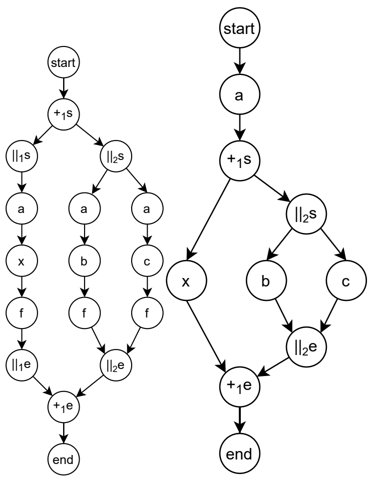
Listing 5 Reduce model

```
1: function REDUCE_MODEL(algebra)
2:     changed = True
3:     while changed do
4:         changed = False
5:         (changed, algebra) = exclude_before_pll(alg)
6:         (changed, algebra) = exclude_after_pll(alg)
7:         (changed, algebra) = exclude_before_alt(alg)
8:         (changed, algebra) = exclude_after_alt(alg)
9:     return algebra
```

The next step of the algorithm, following the generation of the initial algebra model (Figure 6a), is its reduction (Listing 5). The first operation excludes nodes that appear immediately before parallel complex nodes (l. 5). Specifically, the goal is to identify identical nodes (whether complex or simple) that directly succeed the parallel start node and remove them before it.

Three possible cases of this transformation are illustrated in Figures 7–8. In the first case (Figure 7a), all succeeding nodes are identical. Here, the redundant nodes can be straightforwardly excluded, as shown in Figure 7b.

The second case arises when there are multiple identical nodes preceding the parallel start node, but at least one additional node differs (Figure 7c). In this situation, subsets of identical succeeding nodes (denoted as $subset1$) must be identified. For each such subset, a new parallel complex node ($||2start$) is introduced and placed between the nodes of the subset (x_1,x_1) and the original parallel start node ($||1start$). The new parallel start node must also be paired with a corresponding end node ($||2end$), and the appropriate connections are added. To complete this, the algorithm identifies paths (e.g., x_1, y_1) leading from the newly added node ($||2start$) to the last nodes (k, k) before the original parallel end node ($||1end$).



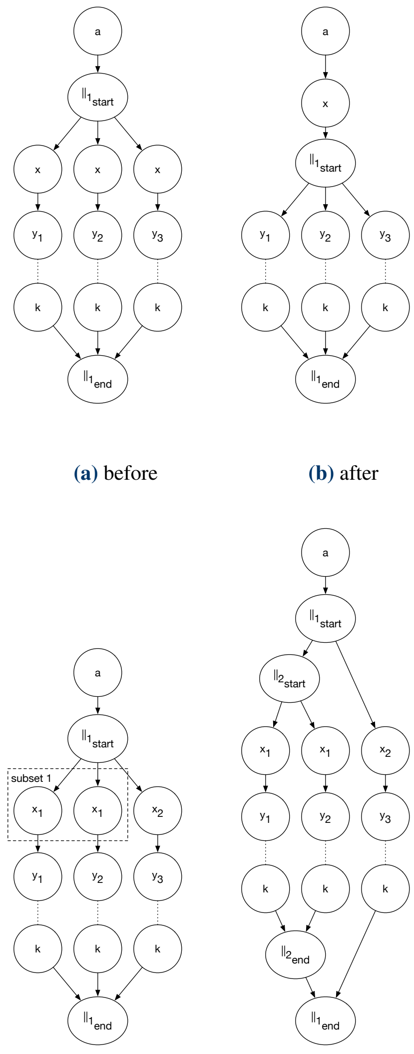
(a) initial model **(b)** final model

Fig. 6. Algebra graph model

The same principle is applied to exclude nodes following the parallel end node (l. 6); however, only examples are provided without detailed discussion. The variants are illustrated in Figures 9a, 9c, and 8a. In this case, instead of analyzing the successors of the parallel start node, the algorithm considers the predecessors of the parallel end node. In the first variant (Figure 9a), the transformation produces the result shown in Figure 9b. The second variant, which involves identifying subsets of identical events, is illustrated in Figure 9c with the result shown in Figure 9d. The third variant is analogous to the case of excluding nodes before the parallel start node (Figure 8).

The same reduction steps are applied to alternative algebra nodes instead of parallel ones (l. 7 and 8).

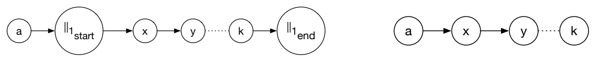
All of the above steps are executed iteratively in a loop (l. 3) until no further structural changes occur in the algebra graph. This is necessary because even a small transformation may enable additional reductions. An example of the resulting graph is presented in Figure 6b.



(a) before **(b)** after

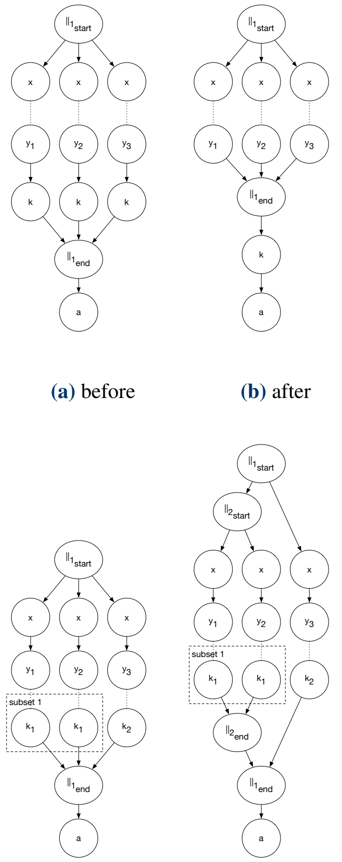
(c) before **(d)** after

Fig. 7. Exclusion before parallel start node (a and b variant I), (c and d variant II)



(a) before **(b)** after

Fig. 8. Exclusion before parallel start node and exclusion after parallel end node (variant III)



(a) before **(b)** after

(c) before **(d)** after

Fig. 9. Exclusion after parallel end node (a and b variant I), (c and d variant II)

3. PART 2: Algebra Miner algorithm with communication mining — AMA-WC-GLB

In this part of the algorithm, we demonstrate how to derive a hierarchical *ROC* model applicable to CRS. To achieve this, we extend AMA-WC-LOC by introducing an additional pre-processing step and a post-processing step.

3.1. Step I: CRS system event collection and session reconstruction The first step of AMA-WC-GLB is to collect event logs from the CRS and link the corresponding invocations of different resources into coherent process instances. To this end, we assume that system resources are capable of recording two types of events: local events (Def. 4.11) and communication events (Def. 4.12).

Def. 4.11 (Local resource event) *Local resource event is a tuple $LRE = \langle event\_id, res\_id, context \rangle$, where:*
$event\_id$ — is a locally resource unique event identifier,
res_id — is a globally unique resource identifier,
$context$ — is a locally unique local process case identifier.

Def. 4.12 (Communication resource event) *Communication resource event (CRE) is a tuple:*
$\langle event\_id, res\_id, context, h\_ctx, source\_uri, dest\_uri \rangle$ where:
$event\_id$ — is a locally unique resource event identifier
res_id — is a globally unique resource identifier
$context$ — is locally unique local process case identifier
h_ctx — context of resource that invoked resource that stores that data
$source\_uri$ — communication source resource URI address
$dest\_uri$ — communication destination resource URI address
and $event\_id \neq \emptyset$, $res\_id \neq \emptyset$, $context \neq \emptyset$, $source\_uri \neq \emptyset$, $dest\_uri \neq \emptyset$

The event log of a CRS system contains both types of events. By applying the context-ordering relation defined in Def. 4.13 together with our earlier session reconstruction algorithm for CRS [7], we can associate resource invocations with the corresponding local events executed during their processing (i.e., the local process of a resource, Def. 4.14). This integration produces global process instances (Def. 4.15), resulting in the construction of the global system event log (*GEL*).

Def. 4.13 (Context ordering relation) *$a \succ_{ctx} b$ iff there is trace in event log where $\#_{ctx}(a) = \#_{hctx}(b) \wedge \#_{res}(a) = resA \wedge \#_{res}(b) = resB$, where $resA, resB \in Res \wedge resA \neq resB$, where Res is a set of all resources in the system, and $\#_{ctx}(e) = A$ is a value of field ctx of event e is A*

Def. 4.14 (Local process) *Local process is a process executed at one resource of CRS during its invocation.*

Def. 4.15 (Global process) *Global process is a process executed in the system that involves communication between multiple resources of that system.*

3.2. Step II and III: mining the local process of each resource In this stage, we extract the local processes of each resource in the system using AMA-WC-LOC (see Section 2). First, as shown in Listing 6, the global event log obtained in Step I is reorganized into a collection of resource-specific logs (l. 2). Then, for each resource represented in the *GEL*, we apply AMA-WC-LOC and add the resulting algebra to the set of CRS algebras (*AG*) (l. 3).

Listing 6 Mining local processes

```
1: function MINE_LOCAL_PROCESSES(GEL)
2:     RL[] = organize_log_into_res(EL)
3:     for all rl ∈ RL do
4:         AG += AMA-WC-LOC(rl)
5:     return AG
```

The example of a result of this step is shown in Figure 10.

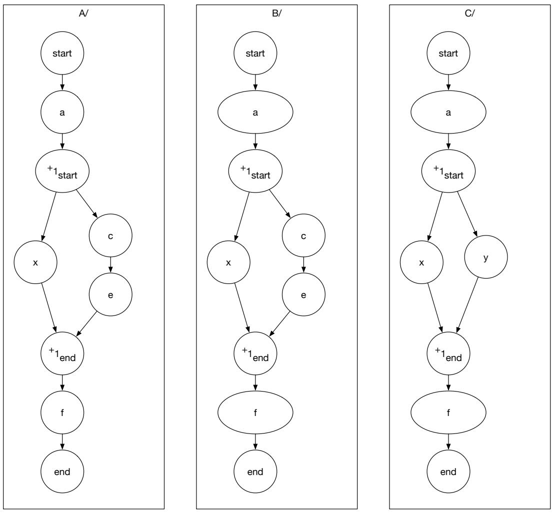


Fig. 10. Discovered model without communication

3.3. Step IV: Mining communication

Finally, communication events are identified based on the local invocation relation (Def. 4.16), the global invocation relation (Def. 4.17), and the approach introduced in our earlier work [8, 25].

Def. 4.16 (Local invocation) *$a \succ_{l.inv} b$ iff a is an event of sending message x and b is an event of receiving a response to the message x where $\#_{res}(a) = \#_{res}(b)$*

The local invocation relation (Def. 4.16) represents events corresponding to the invocation of an external resource during execution and the subsequent receipt of a response, from the perspective of the invoking resource. This relation connects the two events (nodes in the algebra) by a direct arc, since a response cannot occur without a preceding request. As a result, the local process model can be analyzed independently of the global process, i.e., with communication abstracted away.

Def. 4.17 (Global invocation) *$a \succ_{g.inv} b$ iff a is an event of sending message x and b is an event of receiving the message x, where $\#_{res}(a) \neq \#_{res}(b)$*

The global invocation relation (Def. 4.17) specifies the direction of workflow during external resource invocations, enabling the connection of interacting processes within the system. For example, $a \succ_{g.inv} b$ denotes that a is a message-sending event generated by one resource, while b is the corresponding receiving event at another resource. Here, $\#_{res}(e)$ refers to the resource property of event e in the log.

Within the algebraic model, the algorithm identifies communication events at the invoker side and groups them into communication blocks (Def. 4.18). Sequential nodes (Def. 4.4) are then replaced with *recACN* nodes (Def. 4.19) and *resACN* nodes (Def. 4.20) at the receiver side. In addition, the communication block and both communication nodes are connected by a communication edge between them ($ACG.CE$) (Def. 4.22).

Def. 4.18 (Communication block) *Communication block denotes the invocation and response handling events and is a tuple $cb = \langle id, n_{send}, n_{receive}, DATA \rangle$, where $n \in SEQ$ and SEQ is a set of sequence nodes of algebra and id is the identifier of the event in the form of a concatenation of the operation used to invoke and destination which is invoked by this communication. The semantics of this block is as follows: a message is sent when n_{send} is executed, and a response is received when $n_{receive}$ is executed. The $DATA_{send}$ and $DATA_{receive}$ contain information about sent and received codes, headers, body, etc.*

Def. 4.19 (Algebra communication node (receiver) – recACN)
*Algebra communication node (receiver) node, which denotes the entry point for invocation by some remote resource. recACN is a tuple $\langle id, IN,$
$OUT, RELATED, DATA \rangle$, where |IN| = 1, |OUT| = 1, |RELATED| = 1, |commIN| = 1. The semantics of this node is as follows: execute this node action, when iff all seq.IN nodes have already been executed, and the message was sent by node commIN (this node was executed). DATA element contains all information sent in the HTTP request, like headers, body, etc.*

Def. 4.20 (Algebra communication node (responder) – resACN)
Algebra communication node (responder) node is a tuple $seq = \langle id, IN, OUT, RELATED, commOUT, DATA \rangle$, where |IN| = 1, |OUT| = 1, |RELATED| = 0, |commOUT| = 1. The semantics of this node is as follows: execute this node action and send a message to node $commOUT$. DATA element contains all information sent in the HTTP request, like response code, headers, body, etc.

Def. 4.21 (*ROC* process algebra graph – RAG) *ROC process algebra graph is a process algebra graph where some of the sequential nodes are also: recACN and resACN nodes.*

Def. 4.22 (Algebra communication graph – ACG)
Algebra communication graph is a directed graph structure $\langle A, B, CE \rangle$, where A is a set of ROC process algebras graphs (RAG), B is a set of communication blocks, and CE is a set of communication edges in the form (a, b) which denotes that node a is sending the message to node b. Here, the element of set A without any incoming edges from set CE is considered the starting point of the system processing described by ACG and is denoted ACG.start.

Listing 7 presents the $mine_{g}lobal_{p}rocess$ procedure used in this step. First, the algorithm searches for communication entries in each resource log (l. 4) within the algebra groups (*ACG*) (l. 3). When the condition in l. 5 is satisfied (i.e., the local invocation relation holds), a communication block is added to the algebra graph (l. 6).

The semantics of the method are as follows: $gMTH(a)$ retrieves the HTTP invocation method of event a; $destURI(a)$ retrieves the destination URI of event a; and $gNode(a, alg)$ returns the node with identifier a from algebra alg. Next, the algorithm verifies that the global ordering relations are satisfied within the algebra communication graph (*ACG*). For any pair of events that fulfil this relation, an edge is created in the *ACG* to represent the corresponding communication.

Listing 7 Mining global process

```
procedure mine_global_process(AG)
    ACG.A = AG
    for alg ∈ ACG do
        for rl ∈ alg do
            if a ≻_{l.inv} b ∧ a ∈ rl ∧ b ∈ rl then
                ACG.B += { gMTH(a) + "@" + destURI(a), gNode(a, alg), gNode(b, alg)}
    for alg_1 ∈ ACG do
        for alg_2 ∈ ACG do
            for n_1 ∈ alg_1.N do
                for n_2 ∈ alg_2.N do
                    if a ≻_{g.inv} b then
                        ACG.CE += (gNode(a, alg_1), gNode(b, alg_2))
return ACG
```

After applying this step to model 10, the resulting Algebra communication graph *ACG* shown in graphical notation is depicted in the Figure 11.

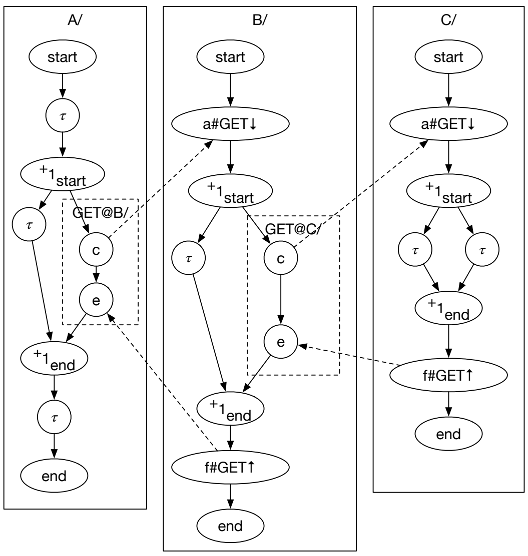


Fig. 11. Discovered model with communication

5. EVALUATION AND TESTS

In this section, we present the evaluation of the proposed algorithm along with a performance analysis of the developed method.

To assess its effectiveness, we designed dedicated test environments (Figures 12, 13, and 15), where event logs were generated in accordance with the contextual logging framework introduced in [7].

It is essential to note that AMA-WC requires complete event logs [40], meaning that no noise is allowed to be present. Consequently, all incomplete local and global processes were excluded from the evaluation. It is worth noting that there are studies that address noise and errors in the log, e.g., [41, 42], which makes it possible to apply this method also in such situations.

The experiments were conducted on a machine with the following configuration: Intel Core i7-5820K CPU @ 3.30 GHz, 16 GB RAM, 1 TB SSD storage, and Windows x64 operating system. Each test involved constructing a hierarchical process composed of communicating local processes, generating the corresponding context logs, and subsequently filtering and transforming these logs into CSV format. The prepared logs were then used as input to the algorithm.

1. Process quality and complexity

Let us first discuss the quality and complexity of process model discovery in AMA-WC, using as an example a simple global process composed of four subprocesses (see Fig. 12).

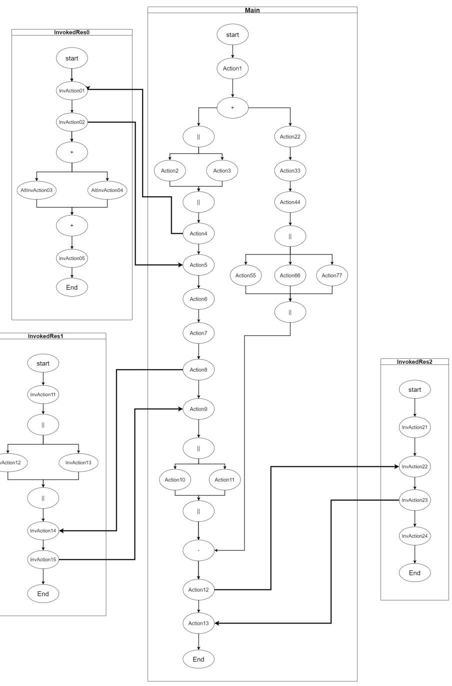


Fig. 12. Hierarchical CRS process discovered by AMA-CM

The experimental setup consisted of a central process invoking three subordinate processes. Each process incorporated standard control-flow constructs supported by the target process algebra, including parallelism, exclusive choice, and communication. The objective of this experiment was to demonstrate that AMA-CM can accurately discover the designed process algebra, independently identify local processes, and, by exploiting contextual information, reconstruct the communication between them.

It is important to emphasize that, when provided with a complete event log of the CRS system, the proposed method is capable of discovering the exact process model expressed in the *ROC* process algebra. The procedure for obtaining such logs was outlined in the introduction to Section 5.

From the perspective of computational complexity, the discovery of local models is grounded in Schimm's algorithm, which exhibits exponential complexity. In CRS systems, however, the processes executed by individual resources are typically not highly complex; the overall complexity arises primarily from the cooperation and interaction of these processes within the global system. The key factor is therefore the complexity of the communication discovery step. This step is implemented using dictionary operations, which have an average-case complexity of $O(1)$ and a worst-case complexity of $O(n)$, where n is the number of dictionary elements.

As a result, the communication discovery step exhibits a multi-linear time complexity: linear in each parameter, but dependent on the product of these parameters.

In the following sections, we examine how the communication context in two distinct composition scenarios influences process discovery time, and we compare communication discovery time with the time required for global process discovery.

2. TEST 1: Performance tests for orchestrated process

In the first test, a process was designed to orchestrate all other processes within the example CRS system (Figure 13).

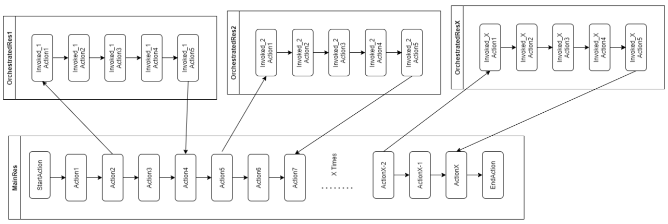


Fig. 13. Test 1: Orchestrated process example.

The purpose of this test was to evaluate a scenario in which a single process sequentially invokes subsequent processes. To minimize the influence of local process complexity on the performance of global process discovery, straightforward, non-branching processes were deliberately chosen.

Each experiment was executed ten times, and the results were averaged.

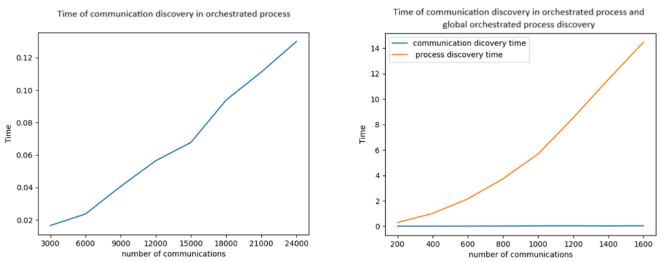


Fig. 14. Test 1: Time for communication discovery (left) and time of global process discovery (right).

As illustrated in Figure 14, the algorithm as a whole exhibits exponential complexity, whereas the communication discovery step remains linear and accounts for only a small fraction of the total execution time.

3. TEST 2: Performance tests for nested communication

The second test (Figure 15) examined a scenario with nested local processes.

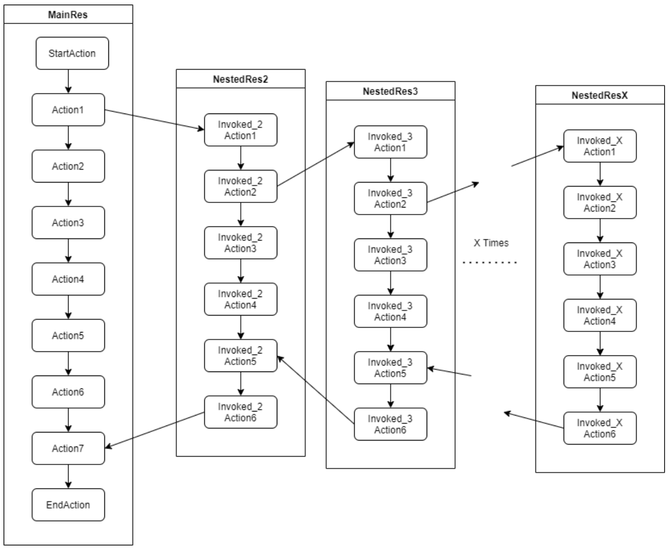


Fig. 15. Test 2: Nested process example.

In this setup, each process is invoked by another process and, in turn, invokes an additional one, resulting in a strictly nested structure where successive processes are embedded within one another. As in the previous experiment, each test was executed ten times, and the results were averaged.

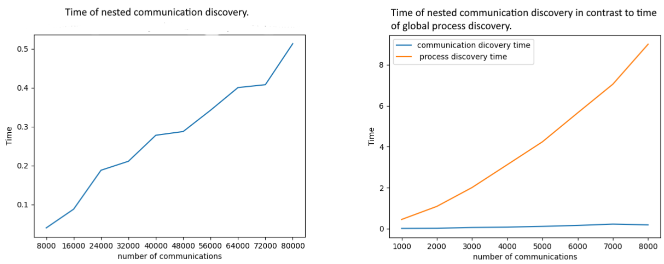


Fig. 16. Test 2: Time for communication discovery (left) and time of global process discovery (right).

The results (Figure 16) confirm that, in this scenario as well, the time required for communication discovery constitutes only a small fraction of the total process discovery time.

4. Test summary

The performance and complexity analysis demonstrates that incorporating composition and communication does not substantially increase the time required for process discovery using the proposed hierarchical algorithm. On the contrary, the approach reduces discovery time compared to scenarios in which the global process is mined as a single entity, without decomposition or communication. In AMA-WC, the exponential complexity applies only to local processes rather than the global process, which considerably lowers the overall time required to discover the global CRS process. Moreover, the discovery of individual local processes can be distributed across multiple resources, further decreasing the total discovery time. Only the step of identifying inter-process connections needs to be performed in a centralized manner.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the *ROC* process calculus, a formal language for modelling communicating resource systems. The calculus captures both the hierarchical structure of such systems and the communication between their resources.

We then proposed an extension of the GS algorithm, enabling the discovery of process models expressed in *ROC*and supporting non-interval event logs. This enhancement broadens applicability to systems implemented with widely used technologies. A further extension allowed us to reconstruct complex processes composed of multiple interacting subprocesses by explicitly modelling communication. While the approach assumes access to contextual logs, prior studies show that such logs can be derived without reimplementing the system.

Overall, the proposed method advances process discovery in CRS systems by explicitly addressing communication and composition. Although our work is centered on CRS and REST systems, the approach is transferable to other domains such as organizational workflows, logistics, and scheduling—provided contextual logs are available. The increasing adoption of OCEL further simplifies this requirement, making contextual log extraction feasible through pre-processing alone. Designing such a pre-processing algorithm, validated with a practical case study, is a promising avenue for future work.

Another direction for further research is extending AMA-WC to incorporate event frequency, quantifying the strength of dependencies between activities. Such a heuristic extension would improve robustness against noise and enable the analysis of typical rather than only idealized process behavior.

7. ACKNOWLEDGMENTS

This work was supported by the Polish National Science Center under Grant No.DEC-2012/05/N/ST6/03051 and by the Poznan University of Technology.

The authors gratefully acknowledge the support of Adam Godziński in enhancing the algorithm implementation and conducting the tests.

ChatGPT (OpenAI) was employed for minor language editing, such as text proofreading, correcting grammar and typographical errors. No AI tools were used in the conceptualisation or analysis of the research.

REFERENCES

[1] W. M. Van Der Aalst and J. Carmona, *Process mining handbook*. Springer Nature, 2022.

[2] R. A. University, "Object-centric event log 2.0," https://ocel-standard.org, 2024, accessed: 2025-07-15.

[3] B. Knopp, M. Pourbafrani, and W. M. van der Aalst, "Discovering object-centric process simulation models," in *2023 5th International Conference on Process Mining (ICPM)*. IEEE, 2023, pp. 81–88.

[4] B. Xiu, G. Li, and Y. Li, "Discovery of object-centric behavioral constraint models with noise," *IEEE Access*, vol. 10, pp. 88 769–88 786, 2022.

[5] G. Li, R. M. de Carvalho, and W. M. van der Aalst, "Automatic discovery of object-centric behavioral constraint models," in *International conference on business information systems*. Springer, 2017, pp. 43–58.

[6] W. M. van der Aalst and A. Berti, "Discovering object-centric petri nets," *Fundamenta informaticae*, vol. 175, no. 1-4, pp. 1–40, 2020.

[7] A. Stroiński, D. Dwornikowski, and J. Brzeziński, "Resource mining: Applying process mining to resource-oriented systems," in *Business Information Systems*. Springer International Publishing, 2014, pp. 217–228. [Online]. Available: https://doi.org/10.1007/978-3-319-06695-0_19

[8] A. Stroiński, D. Dwornikowski, and J. Brzeziński, "Restful web service mining: simple algorithm supporting resource-oriented systems," in *Web Services (ICWS), 2014 IEEE International Conference on*. IEEE, 2014, pp. 694–695. [Online]. Available: https://doi.org/10.1109/ICWS.2014.106

[9] W. van der Aalst, "On the representational bias in process mining," in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*, June 2011, pp. 2–7. [Online]. Available: https://doi.org/10.1109/WETICE.2011.64

[10] D. Dwornikowski, A. Stroinski, and J. Brzezinski, "Conformance checking of communicating resource systems with ras calculus," in *2015 IEEE International Conference on Services Computing*, 2015, pp. 759–764. [Online]. Available: https://doi.org/10.1109/SCC.2015.109

[11] W. M. P. v. d. Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance checking of service behavior," *ACM Trans. Internet Technol.*, vol. 8, no. 3, pp. 13:1–13:30, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1361186.1361189

[12] W. M. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H. Verbeek, "Choreography conformance checking: An approach based on bpel and petri nets," *BPM Center Report BPM-05-25, BPMcenter.org*, 2005.

[13] R. Weinreich and G. Buchgeher, "Automatic reference architecture conformance checking for soa-based software systems," *2014 IEEE/IFIP Conference on Software Architecture*, vol. 0, pp. 95–104, 2014. [Online]. Available: https://doi.org/10.1109/WICSA.2014.22

[14] M. Rouached, W. Gaaloul, W. van der Aalst, S. Bhiri, and C. Godart, "Web service mining and verification of properties: An approach based on event calculus," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 4275, pp. 408–425. [Online]. Available: https://doi.org/10.1007/11914853_25

[15] C.-A. Sun, A. Fu, J. Jia, M. Li, and J. Han, "Improving conformance of web services: A constraint-based model-driven approach," *ACM Transactions on the Web*, vol. 17, no. 2, pp. 1–36, 2023.

[16] M. Rafiei, M. Pourbafrani, and W. M. van der Aalst, "Federated conformance checking," *Information Systems*, vol. 131, p. 102525, 2025.
[17] H. T. Vieira, L. Caires, and J. C. Seco, "The conversation calculus: A model of service-oriented computation," in *Programming Languages and Systems*. Springer, 2008, pp. 269–283. [Online]. Available: https://doi.org/10.1007/978-3-540-78739-6_21
[18] A. Lapadula, R. Pugliese, and F. Tiezzi, "A calculus for orchestration of web services," in *Programming Languages and Systems*. Springer, 2007, pp. 33–47. [Online]. Available: https://doi.org/10.1007/978-3-540-71316-6_4
[19] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti, "Sessions and pipelines for structured service programming," in *Formal Methods for Open Object-Based Distributed Systems*. Springer, 2008, pp. 19–38. [Online]. Available: https://doi.org/10.1007/978-3-540-68863-1_3
[20] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro, "Sock: a calculus for service oriented computing," in *Service-Oriented Computing–ICSOC 2006*. Springer, 2006, pp. 327–338. [Online]. Available: https://doi.org/10.1007/11948148_27
[21] D. Dwornikowski, A. Stroiński, and J. Brzeziński, "Towards a process calculus fo REST: current state of the art," *Foundations of computing and decision sciences*, vol. 40, no. 4, pp. 237–265, 2015. [Online]. Available: http://dx.doi.org/10.1515/fcds-2015-0015
[22] M. M. Chara Eddine, "A comparative study of formal approaches for web service oriented architecture," *Network and Communication Technologies*, vol. 5, no. 2, p. 15, 2020. [Online]. Available: http://dx.doi.org/10.5539/nct.v5n2p15
[23] W. Ju, "A new process algebra more suitable for formal specification," in *Proceedings of the 4th International Conference on Computer Science and Software Engineering*, 2021, pp. 103–106.
[24] U. Klein and K. S. Namjoshi, "Formalization and automated verification of RESTful behavior," in *Computer Aided Verification*. Springer, 2011, pp. 541–556. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_43
[25] A. Stroiński, D. Dwornikowski, and J. Brzeziński, "A distributed discovery of communicating resource systems models," *IEEE Transactions on Services Computing*, vol. 12, no. 2, pp. 172–185, 2019. [Online]. Available: https://doi.org/10.1109/TSC.2016.2609913
[26] W. Van Der Aalst, "Service mining: Using process mining to discover, check, and improve service behavior," *IEEE transactions on services Computing*, vol. 6, no. 4, pp. 525–535, 2012.
[27] Y. Tang, T. Li, R. Zhu, C. Liu, and S. Zhang, "A hybrid genetic service mining method based on trace clustering population," *IEICE TRANSACTIONS on Information and Systems*, vol. 105, no. 8, pp. 1443–1455, 2022.
[28] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, "Web api evolution patterns: A usage-driven approach," *Journal of Systems and Software*, vol. 198, p. 111609, 2023. [Online]. Available: http://dx.doi.org/10.1109/TSC.2012.25
[29] M. Leemans, "Hierarchical process mining for scalable software analysis," 2018.
[30] J.-V. Benzin and S. Rinderle-Ma, "Petri net classes for collaboration mining: Assessment and design guidelines," in *International Conference on Process Mining*. Springer, 2023, pp. 449–461. [Online]. Available: https://doi.org/10.1007/978-3-031-56107-8_34
[31] J.-V. Benzin and S. Rinderle-Ma, "Collaboration miner: Discovering collaboration petri nets (extended version)," *arXiv preprint arXiv:2401.16263*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.16263
[32] C. W. Günther and W. M. Van Der Aalst, "Fuzzy mining–adaptive process simplification based on multi-perspective metrics," in *International conference on business process management*. Springer, 2007, pp. 328–343. [Online]. Available: https://doi.org/10.1007/978-3-540-75183-0_24
[33] M. Imran, M. A. Ismail, S. Hamid, and M. H. N. M. Nasir, "Complex process modeling in process mining: A systematic review," *IEEE Access*, vol. 10, pp. 101 515–101 536, 2022.
[34] G. Schimm, "Mining exact models of concurrent workflows," *Computers in Industry*, vol. 53, no. 3, pp. 265–281, 2004. [Online]. Available: https://doi.org/10.1016/j.compind.2003.10.003
[35] G. Schimm, "Generic linear business process modeling," in *Conceptual Modeling for E-Business and the Web*. Springer, 2000, pp. 31–39. [Online]. Available: https://doi.org/10.1007/3-540-45394-6_4
[36] M. Grinberg and T. P. Projects, *Flask Documentation*, The Pallets Projects, https://flask.palletsprojects.com/.
[37] Django Software Foundation, *Django Documentation*, Django Software Foundation, https://docs.djangoproject.com/.
[38] Rocket Contributors, *Rocket Web Framework Documentation*, Rocket Project, https://rocket.rs/guide/v0.5/.
[39] Spring.io Team, *Spring Boot Reference Documentation*, VMware, https://docs.spring.io/spring-boot/docs/current/reference/html/.
[40] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, Incorporated, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-19345-3
[41] H. M. Marin-Castro and E. Tello-Leal, "Event log preprocessing for process mining: a review," *Applied Sciences*, vol. 11, no. 22, p. 10556, 2021.
[42] A. Koschmider, K. Kaczmarek, M. Krause, and S. J. van Zelst, "Demystifying noise and outliers in event logs: Review and future directions," in *International Conference on Business Process Management*. Springer, 2021, pp. 123–135.