IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 July 2023, accepted 6 October 2023, date of publication 16 October 2023, date of current version 8 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3324742

RESEARCH ARTICLE

Extending Graph-Based Process Discovery for Hierarchical Block Detection and Incomplete Concurrent Relationship Handling

INDRA WASPADA[1,3], RIYANARTO SARNO[1], (Senior Member, IEEE), ENDANG SITI ASTUTI[2], HANUNG NINDITO PRASETYO[1,4], AND RADEN BUDIRAHARJO[1,5]

[1]Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology, Institut Teknologi Sepuluh Nopember, Surabaya 60111, Indonesia
[2]Department of Business Administration, Brawijaya University, Malang 65145, Indonesia
[3]Department of Informatics, Faculty of Science and Mathematics, Diponegoro University, Semarang 50275, Indonesia
[4]Department of Information System Diploma, School of Applied Science, Telkom University, Bandung 40257, Indonesia
[5]Department of Information System, Faculty of Industrial Technology, Institut Teknologi Nasional, Bandung 40124, Indonesia

Corresponding author: Riyanarto Sarno (riyanarto@if.its.ac.id)

This work was supported in part by the Indonesian Ministry of Education and Culture under Penelitian Terapan Unggulan Perguruan Tinggi (PTUPT) Program, in part by Institut Teknologi Sepuluh Nopember (ITS) under project scheme of the Publication Writing and Intellectual Property Rights Incentive Program (Insentif Publikasi dan Hak Kekayaan Intelektual), and in part by the Diponegoro University Scholarship Program.

ABSTRACT The flow of business process activities in an organization can run simply in the form of sequential or complex in the form of choice and parallel. In the discovery process, parallel conditions can be deduced from the concurrent flows between activities. Concurrent patterns can be identified by tracing pairs of sequences in opposite directions between two activities. Long parallel blocks allow incomplete concurrent relationship (ICR) phenomena when the observed data only provides a one-way sequence flow against conditions that should be concurrent. All state-of-the-art algorithms starting from Graph-based Process Discovery (GPD), Inductive Miner (IM), and Split Miner (SM), detect ICR as a sequence so that the quality of the resulting model decreases. This paper proposes the GPD++ algorithm for block detection of complex structures by combining algorithms from GPD and SM and handling ICR for each detected parallel block. The experiment compared IM, SM, and GPD++ using real-life data from the container loading and unloading process during import activities and some public data. The experimental results show that only GPD++ has successfully detected and corrected ICR to obtain the best fitness and precision values. In addition, GPD++ is also the only algorithm that can find hierarchies in the same type of complex structure, resulting in a block model with consistent split-join pairs.

INDEX TERMS Graph-based process discovery, graph database, incomplete concurrent relationship, block hierarchy, GPD++.

I. INTRODUCTION

Automatic discovery of process models from event logs makes a significant contribution to many organizations, from finance, health, industry, government, and many other fields, in capturing descriptions of business process execution models from real-world activities.

Automatic discovery of the process models from the event logs significantly contributes to organizations, from finance, health, industry, government, and many others, in capturing a model description of the execution of business processes in the field [1]. One of the methods for obtaining this model is by integrating event log data into a directly following graph (DFG) model [2], [3]. The DFG model is still primitive because it only displays sequence relationships, so it does not yet have a semantic representation for choice and parallel branching. Therefore, algorithms have been developed to

The associate editor coordinating the review of this manuscript and approving it for publication was Zhangbing Zhou.

obtain high-level models with simulation capabilities based on formal mathematics, such as Petri Net [4], or models with industry-standard notation, such as BPMN [5]. The Petri Net model can be converted to BPMN easily so that these two models become the standard models most widely used as outputs from process discovery techniques [6].

The inductive miner (IM) algorithm [7] traces the DFG and cuts to get Sequence, XOR, and AND blocks. The resulting output is a tree structure that can be converted to a Petri Net model. The split miner (SM) algorithm [3] uses a specific approach in detecting split and join parts. The first stage of this algorithm detects all split points along with their hierarchies and types of branches. Then proceed with the second stage to determine the join using the refined process structure tree (RPST) algorithm. The GPD algorithm [8], [9] utilizes the cypher algorithm on the Neo4j graph database to detect split and join parts and uses concurrent path frequency values to distinguish between AND and OR. However, these state-of-the-art process discovery algorithms have not been able to detect process flows that represent incomplete concurrency, which will reduce the quality of the resulting model.

In [10] we developed GPD+, which proposes the parallel block concept to recognize ICR. However, the GPD+ method still has weaknesses because it cannot detect branching hierarchies and requires the availability of concurrency flows on the join part in order to detect (parallel) blocks.

We propose the development of the GPD++ algorithm by combining SM hierarchy detection techniques and graph pattern tracing algorithms to obtain block structures. The location and type of join relation (XOR, AND) can be determined based on the detected block. Thus the contributions we provide include the following:

1. Propose a split-join block abstraction that contains components like a split point, entrance point, region, exit point, and join point.
2. Modifying and developing the split discovery algorithm in SM by utilizing the graph database to find split points, split-join blocks, and their block hierarchies.
3. Propose a technique for detecting incomplete concurrent relationships (ICR) by utilizing the parallel block abstraction model. Two activities that are in different regions in the same parallel block and have sequence paths that are directly connected are declared detected as ICR flows.

The experiment was carried out using real-life proprietary data from the business process of import activities at the container terminal (TPS) and public data. The TPS data contains data from green-line, yellow-line, and red-line container activities. In this case, the green and yellow line data contains the ICR. In addition, the green path data presents a challenge to detect the hierarchy of two AND blocks. Meanwhile, the red and yellow paths challenge hierarchical variations between XOR and AND, and contain skip paths. The public data used are BPIC12 [11] and BPIC17 [12] because they have long parallel block patterns containing ICR.

The experimental results show that GPD++ succeeded in finding a split-join block hierarchy including blocks of the same type. GPD++ can recognize and improve ICR paths so as to get a model with the highest quality compared to IM and SM.

The paper structure is constituted as follows. Section II contains a catalog of related works. The third section defines and explains the concepts that underpin our research. The fourth section describes the fundamentals of the proposed method. The fifth section provides the experimental results and analyses the findings. The last Section gives a conclusion and overview of future works.

II. RELATED WORK

This section discusses research related to the work carried out including the process discovery method, its quality dimensions, and graph-based process mining.

A. PROCESS DISCOVERY METHODS

A data-driven automated method called Process Discovery can locate, map, and record the activities of current business processes. The received data is then automatically analyzed so that it can be automatically recommended in the process modeling or workflow [13]. An automated representation of the business processes is created, leaving a digital trace in the system. When launching a digital transformation program as well as when enhancing the efficiency of current processes, the discovery and analysis of an organization's business processes can be used to pinpoint important problem areas. Process Discovery is essential for enhancing the caliber of business process management, together with business process modeling [14]. The process discovery algorithm is still being developed. According to the requirements of the resulting process model, there are numerous new process discovery techniques. However, based on prior research, the output comparison algorithms compatible with Petri Net for creating process models in the form of Petri Nets are Inductive Miner and Split Miner, which are the best in terms of fitness and precision.

Modern process discovery methods include the Split Miner (SM) [3], Inductive Miner (IM) [2], [7] algorithms. Sequence slicers, XOR, AND, and loops are some of the methods used by IM to divide event logs. In the absence of a flawless cut, this algorithm compares the value of the best cut to be chosen when selecting a cutter. Although the process model produced frequently sacrifices precision levels for high fitness values. SM, on the other hand, begins the discovery process by creating a directly follows graph (DFG) as a middle model. SM determines which portions are separated and united based on DFG. The properties of each split and join are examined in the following step to determine the best branching configuration. This method results in a process model produced by the SM technique that is slightly less fit but has a higher precision value than IM.

Since IM and SM must load every log into memory, working with large amounts of data is limited. The Graph-based Invisible Task (GIT) approach was presented by Sarno et al. [8], [14] for the purpose of identifying invisible tasks using

a graph database. Additionally, Waspada et al. [9] created a Graph-based Process Discovery (GPD) to improve GIT so that it can operate more broadly by focusing on the frequency on the edge and taking concurrency and frequency into account in the detection of exclusive OR (XOR), parallel (AND), and inclusive OR (OR) patterns. In the sections below, we'll go over the GPD techniques employed in [9]. The following algorithms are carried out in nine steps:

1. Load the event log as a set of nodes for a graph. Trace and process model nodes are now loaded into the graph database using the event log data as two different types of nodes.
2. Employ definition 1 to help to create a directly-follows graph (DFG) and its frequency. The matching of a consecutive pair of nodes in each unique case id results in the creation of a DFG in the graph database. After then, a relationship known as a sequence relationship, which connects the two nodes, is established.
   Definition 1 (Directly-Follows Graph [15]): With an event log L where $L \in \mathbb{B}(\mathcal{A}^*)$, the directly-follows graph of L is written as $G(L) = \left(A_L, \mapsto_L, A_L^{start}, A_L^{end}\right)$ with:
   - $A_L = \{a \in \sigma \mid \sigma \in L\}$ is the set of activities in L
   - $\mapsto_L = \{(a, b) \in A \times A \mid a >_L b\}$ is the directly follows relation,
   - $A_L^{start} = \{a \in A \mid \exists_{\sigma \in L^a} = first(\sigma)\}$ is the set of start activities, and
   - $A_L^{end} = \{a \in A \mid \exists_{\sigma \in L^a} = last(\sigma)\}$ is the set of end activities

   A directly-follows graph $G(L)$.$a \mapsto_L b$ exists if a was directly followed by b somewhere in any sub log L.
   The first step is to develop a DFG model for the process model as well as the trace model. Second, the frequency counter for each relationship that grows each time is combined in the graph model. According to definition 2 of this value, it is referred to in [3] as directly-follows frequency (DFF).
   Definition 2 (Directly-Follows Frequency): Given an event log ε, and two events label l_1,l_2 ε L, the directly-follows frequency between l_1 and l_2 ($|l_1 \rightarrow l_2|$) is $\left|\left\{\left(e_i, e_j\right) \in \varepsilon x \varepsilon | e_i^l = l_1 \wedge e_j^l = l_2 \wedge \exists_t \in \varepsilon\, [\exists e_x \varepsilon t \left[e_x = e_i \wedge e_{x+1} = e_j\right]]\right\}\right|$
3. Counting each node's frequency connection. The DFF value in each relationship is utilised to calculate the input frequency (*ifr*) and output frequency (*ofr*).
4. Determine a relationship that is ongoing. When node A and node B are directly connected to one another and vice versa, concurrent relations are discovered. Because they both function simultaneously in reality, this circumstance can occur in models. It is important to note that the concurrent relations pattern in the graph process model resembles the short-loops pattern. We make use of the notion of short-loop and concurrent relationships found in [3].
   Definition 3 (Short-Loop): Given two activities, x and y, x short-loop ($x \circlearrowright y$) exist iff meets the requirement in (1) and (2) [3].

$$|\boldsymbol{x} \rightarrow \boldsymbol{x}| = \mathbf{0} \wedge |\boldsymbol{y} \rightarrow \boldsymbol{y}| = \mathbf{0} \tag{1}$$

$$|x \longleftrightarrow y| + |x \longleftrightarrow y| \neq \mathbf{0} \tag{2}$$

   The rule in Condition 1 imposes the restriction that a and b may not be contained in a self-loop. Condition 2 guarantees the short-loop pattern $x \circlearrowright y$.
   Definition 4 (Concurrent): Given activities x and y, they are concurrent $x \parallel y$ iff meet conditions in (3), (4), and (5) [3].

$$|x \rightarrow y| > \mathbf{0} \wedge |y \rightarrow x| > \mathbf{0} \tag{3}$$

$$|x \longleftrightarrow y| + |y \longleftrightarrow x| = \mathbf{0} \tag{4}$$

$$\frac{||x \rightarrow y| - |y \rightarrow x||}{|x \rightarrow y| + |y \rightarrow x|} < \boldsymbol{\varepsilon} \quad (\boldsymbol{\varepsilon} \in [\mathbf{0}, \mathbf{1}]) \tag{5}$$

   Based on definition 4 and three requirements, namely conditions 3, 4, and 5. It is necessary to correlate the pattern in the process model to the pattern in the execution trace in order to determine whether or not two activities with opposite relations are concurrent. Condition 3 is the most essential antecedent for xy. Preventing a brief loop constitutes the fourth requirement. When the frequency of concurrent relationships in both directions must satisfy a specified threshold (threshold,), Condition 5 must be met.
5. Discover Split and Join. The rules to discover Split and Join only work on sequential relationship. The Split and Join is a foundation for the next step to detect the complex structures.
6. Discover XOR pattern. This approach is built with the restriction that there cannot be any concurrent relationships between branch nodes in the XOR relation.
7. Discovery AND pattern. The detection of AND and OR need concurrent pair in their branches. The AND pattern can be distinguished when the frequency of connections (*ifr* or *ofr*) in the branches is identical.
8. Discover OR pattern. The algorithm for locating OR relationships is created by identifying concurrent relationships in branches and branching patterns with gateway nodes having a higher connection frequency value than each branch node.
9. Remove any connections that are concurrent. From the process model, all concurrent relationships were eliminated.

B. QUALITY DIMENSIONS FOR PROCESS DISCOVERY

The first categorization, process discovery, is perhaps the most difficult task in the process mining field. The main goal of this task is to record a system's behaviour by obtaining its event log. The outcomes from the task are model processes [15]. Different algorithms are employed to translate the logs into representative models to create these models. The condition that the algorithms' output models be "representative" provides few specifics regarding how the implementations should be carried out. When creating models, four quality

criteria should be taken into account in order to attain "representativeness" [1]. The criteria are specifically [15], [16]:

1. Fitness: The fitness dimension typically allows for event log-observed behaviours to be repeated in the models created. This criterion has a value that ranges from 0 to 1 as its measurement.
2. Precision: This dimension might be considered one of the criteria for assessing a model that has been developed. The accuracy criterion is concerned with how an event log's recorded actions should be represented in a model.
3. Generalization: This dimension indicates that a created model must generalize to account for behaviours that are not recorded in an event log. Scores for generalization range from 0 to 1, with scores closer to 1 producing an underfitting model.
4. Simplicity: This aspect of process discovery adheres to the idea that a model should be as straightforward as possible. By comparing the size of a model's process tree to the number of activities in an event log, one can determine the model's level of simplicity.

C. GRAPH-BASED PROCESS MINING

Process mining has a close relationship with graph theory [17]. This is because the event log as the main data input in process mining is a component of a business process that records behavior-related activities. The relationship between these activities can be accurately represented by the graph model.

Sarno et al. [18] initiated process discovery utilizing graph-based mining in a graph database setting. The study proposes a Graph-based invisible task (GIT) to identify a process model containing an invisible task. Next, Sarno et al. proposed a way to store ERP Event logs directly in a graph database, so that process discovery does not require event log format conversion. In terms of time complexity and computation time, the GIT algorithm outperforms other discovery methods [14].

III. FUNDAMENTALS

In this section, terms and concepts relating to process mining, event log analysis, Petri nets, and graph databases are covered.

A. PROCESS MINING

Process mining is a method or activity used to explore and examine event data in order to learn from and understand the insights produced while processes are being executed [19]. The specifics of process discovery tasks include process discovery, modeling, monitoring, and optimization [20]. The benefits of engaging in the process mining include the following [21]:

1. Process discovery. A process that is used to analyze event logs by transforming them into process models.
2. Conformance checking. An investigation into the discrepancies between the model and what actually occurs. The organization will be able to identify the necessary items with the help of this stage in order to enhance current procedures.
3. Throughput/bottleneck analysis. Determine potential process bottlenecks by performing an activity to calculate the intensity of event implementation.

TABLE 1. Sample of an event log.

Case	Activity	Resource	Timestamp
...	...	...	...
101	Check item stock (a)	Joe	17-10-2021:10.02
102	Check item stock (a)	Alex	17-10-2021:11.32
103	Check item stock (a)	Sue	17-10-2021:11.45
101	Classify as high (b)	Mary	17-10-2021:15.06
103	Classify as low (c)	Sue	17-10-2021:16.10
...	...	...	...
102	Issue purchase order (f)	Alex	18-10-2021:16.01
101	Issue purchase order (f)	Joe	18-10-2021:12.00
...	...	...	...

TABLE 2. Sample of XES format structure for an event log.

Case	Activity	Resource	Timestamp
101	Check item stock (a)	Joe	17-10-2021:10.02
101	Classify as high (b)	Mary	17-10-2021:15.06
...	...	...	...
101	Issue purchase order (f)	Joe	18-10-2021:12.00
102	Check item stock (a)	Alex	17-10-2021:11.32
...	...	...	...
102	Issue purchase order (f)	Alex	18-10-2021:16.01
103	Check item stock (a)	Sue	17-10-2021:11.45
103	Classify as low (c)	Sue	17-10-2021:16.10
...	...	...	...

B. EVENT LOG

An event log is data resulting from the execution of business processes that are stored in the form of activities and accompanying information. For example, in Table 1 provides an illustration of the results of recorded event logs. Each row contains an event that describes a particular process instance. XES format (eXtensible Event Stream) is typically used to store event logs. According to its case id, XES successively arranges each event in a single trace. In Table 2, a straightforward example of the XES format of the event log in Table 1 is shown.

C. PROCESS MODEL

A process model provides an overview of the execution of a process. This paper uses the Business Process Modeling Notation (BPMN) and Petri Net model. BPMN is the most widely used business process modeling notation. BPMN has support from multiple vendors and has been standardized by OMG. Petri Net supports simulation and is compatible with many analytical techniques [15].

Definition 5 (Petri Net): A Petri net is a tuple $N = (P, T, F, \alpha, m_i, m_f)$ *where* P *is a finite set of places,* T *is a finite set of transitions,* $F \subseteq (P \times T) \cup (T \times P)$ *is a finite set of arcs as flow relation,* $\alpha : T \rightarrow A$ *is the transition mapping function to labels.*

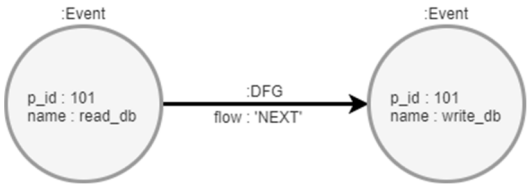


FIGURE 1. Two nodes with Event label and a relationship with DFG type.

D. GRAPH DATABASE

Graph database models are defined as the data structure in which schema and instances are modeled as graphs and graph-oriented operations are performed on graphs. It has recently received a great deal of attention in both research and industry due to its capacity to manage vast quantities of data and networks. This paper employs Neo4j GDBMS and the graph query language Cypher to implement the graph.

Figure 3 illustrates a straightforward graph model. For instance, the cypher syntax (x:Event p_id: 102)-[r:DFG {Flow:'NEXT'}]->(y) declares the variable x for nodes with the label:Event and the variable r for relations with the label:DFG. Additionally, the p_id attribute is declared for node x, and the Flow attribute is declared for relationship r.

IV. THE PROPOSED METHOD

GPD+ still has several weaknesses, namely not being able to detect the split-join block structure hierarchy, not being able to detect invisible tasks, and requiring concurrent exit nodes to be able to recognize the block structure. To correct this weakness, a generalization of the split-join block pattern abstraction (hereinafter referred to as block pattern) is proposed in Figure 2 (which is a generalization of the parallel block pattern in GPD+).

Through this approach, choice blocks (XOR) or parallel blocks (AND) can be built. In addition, this method will also support block hierarchy discovery.

Based on the illustration in Figure 2, the minimum component requirements that must be met to build a split-join block are introduced, namely:

1. The block pattern has a Split-node, the Split-nodes are connected to pairs of Entrances nodes, and the pairs of Exits nodes are connected to a Join-node.
2. The Entrance node has a path that ends at the Exit node. Exit nodes are connected to Join-Gate nodes.
3. The nodes that are on the path between Entrance nodes and Exit nodes are in a region. The Entrance and Exit nodes also part of the region.
4. Blocks are declared parallel if the split-node part is connected to Entrance nodes connected to each other concurrently.
5. In a parallel block, every edge that is connected across regions is concluded as a concurrent relation.

A. REAL-LIFE CASE WITH SPLIT-JOIN BLOCK HIERARCHY PATTERN AND CONTAINING ICR

Most process model discovery algorithms depart from the directly follows graph (DFG) representation. An overview of

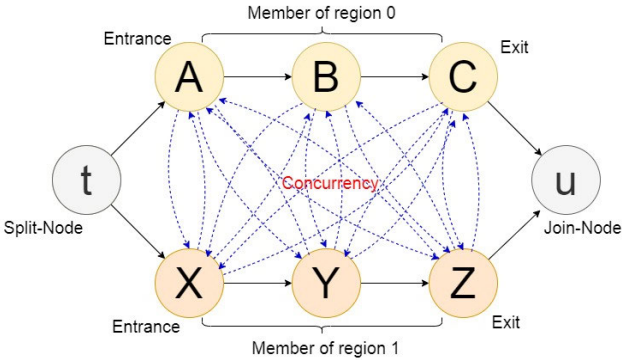


FIGURE 2. Split-join block pattern abstraction.

the DFG containing the split-join block hierarchy and the ICR can be seen in Figure 3.

In the DFG in Figure 3, the red ellipse shows that multiple concurrent relationships can potentially constitute a parallel block hierarchy. The figure also shows an incomplete concurrent relationship (ICR). The Graph-based Process Discovery (GPD+) method still detects the split and join points separately so that it seems they have no relationship semantically. One of the drawbacks of this approach is that it does not support detecting ICRs between the split (entrance) and join (exit) sections. Inductive Miner (IM) and Split Miner (SM) also cannot detect ICR, so it will be detected as an additional split or join, reducing the quality of the resulting model.

As a solution, the GPD++ algorithm is proposed with the ability to detect block hierarchies, is more flexible in recognizing long parallel block patterns, and is more reliable in detecting and handling ICR patterns.

B. GRAPH-BASED PROCESS DISCOVERY PLUS PLUS (GPD++)

GPD++ consists of the main algorithm for detecting split and join blocks presented in Algorithm 1 and the algorithm for finding the block hierarchy and its types (XOR or AND) in Algorithm 2 and Algorithm 3. GPD++ utilizes the detected block structure to identify ICR in blocks parallel (long).

As an illustration of how Algorithm 1 works in finding split-join blocks, an example of a DFG model with one split node ($|T| = 1$) and one join node is given in figure 4. The input for Algorithm 1 is the DFG model with split-node and join-node information. The results of the first iteration of finding blocks, namely Algorithm 1 in the 14th row (finding the XOR block) and the 15th row (finding the AND block) are presented in Figure 5. It appears that there are AND-split gates and XOR-split gates that have been inserted between the split points -node (t) and its direct successors (s1, s2, s3, s4). The red rectangles indicate the blocks found, starting after the split-node and ending before the join-node. Based on the identified block, the position of the joint gate can be determined. However, the AND-join gate and XOR-join gate at this stage have not been installed in the model, but only the gate name and location are recorded (connected to any node).

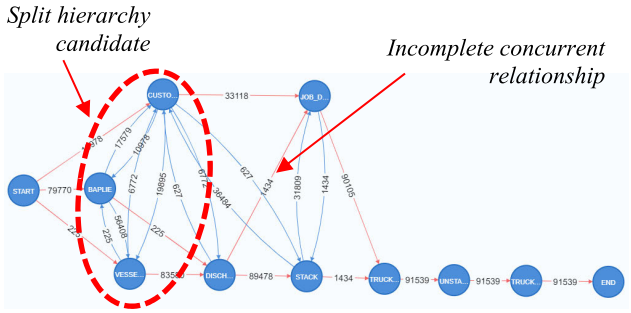


FIGURE 3. Directly follows graph (DFG) representation of the event log on the green line of import activities at the container terminal.

Algorithm 1 Graph-Based Process Discovery Plus Plus (GPD++)

Input: DFG modal in graph database with split and join nodes

```
1  function GPD++ (DFG, startNode, counter)
2      T ← getAllSplitNodes
3      While |T| > 0 :
4          t ← closestSplitNode(startNode, T)
5          S ← direct successors of t
6          C ← ∅
7          F ← ∅
8          for s1 ∈ S do
9              C[s1] ← {s1}
10             F[s1] ← ∅
11             for s2 ∈ S do
12                 If (s2 ≠ s 1 ∧ s 2||s1) then add s2 to F[s1]
13         While |S| > 1
14             discoverXORsplitJoinBlock(DFG, t, S, C, F, counter)
15             discoverANDsplitJoinBlock(DFG, t, S, C, F, counter)
16         remove t from T
```

The installation of the joint gate is carried out after all the split gates have been installed.

The result of the first iteration, which adds an XOR-split gate and an AND-split gate, becomes the reference for discovering the next block. The second iteration succeeded in finding another XOR block, the hierarchical parent block of the two previous blocks, as shown in Figure 6.

After the insertion of the second XOR gate, the number of direct succession from the split point is only 1 ($|S| = 1$), indicating that the iteration is complete (line 13 of Algorithm 1). The last step is to insert all the join gates according to the split-join block structure found so that the final result is presented in Figure 7.

C. COMPLEXITY

The input of GPD++ is the DFG model. Let n be the number of events in the event log. The construction of the DFG model is run by matching the event nodes from each trace so that the complexity of constructing the DFG is O(n).

Let m be the number of unique task names. The finding of choice blocks (Algorithm 2) and parallel blocks (Algorithm 3) is $O(m^3)$ respectively. GPD++ allows running both Algorithm 2 and Algorithm 3 m iteratively (in two nested loops for split node tracing and direct successor tracing), so the complexity of GPD++ (Algorithm 1) is $O(m^5)$.

Algorithm 2 Choice (XOR) Split-Join Block Discovery With Hierarchy For GPD++

Input: DFG modal in graph database with split and join nodes

```
1  function discoverXORsplitJoinBlock(DFG, t, S, C, F, counter)
2      do
3          X ← ∅
4          For s1 ∈ S do
5              C_u − C[s1]
6              for s2 ∈ S do
7                  if F_s1 = F_s2 ∧ s 1 ≠ s2 then
8                      Add s2 to X
9
10             If X ≠ ∅ then
11                 add s1 to X
12                 break
13         If X ≠ ∅ than
14             U ← getAllJoinNodes
15             P ← allPathVariantsFromEntranceToExit(A, U)
16             If P = ∅ than
17                 g^X = ∅
18                 return g^X
19             for p in P :
20                 V ← getValidBlocks (p)
21             for u in V :
22                 While true :
23                     V, status ← mergeBlocks (u, V)
24                     if status is not true then break
25             H− getTheNextHierarchies(V)
26             for h in H :
27                 g^X ← insertXORsplitGateway (t, h, counter)
28                 Add u to U^X          // add all XORjoin later
29                 counter = counter + 1
30                 e ← getEntrancePairs(h)
31                 for i in range(|e|):
32                     C_u ← C[e[i]]
33                     F_i ← F[e[i]]
34                     remove e[i] from S
35                     remove e[i] from C
36                     remove e[i] from F
37                 Add g^X to S
38                 C[g^X] ← C_u
39                 F[g^X] ← F_i
```

D. DATA SET AND PREPROCESSING

GPD++ testing makes use of two data set types. The first type of data is private real-world data regarding import activities at container terminals. Real-life data is obtained from the Terminal Operations System (TOS) by querying it. The obtained results do not conform to the required event record structure. Therefore, we preprocess to conform to the standard format for Extensible Event Stream (XES). Final event log results are saved in CSV format for import into the Neo4j graph database.

From this data, there are three groups of data with different business process flows: green line data, red line data, and yellow line data. Green line if it does not require physical inspection by customs or quarantine. Red line if the container needs to be inspected by customs. Yellow line if it requires inspection by the quarantine department. These three data

Algorithm 3 Paralel (AND) Split-Join Block Discovery With Hierarchy for GPD++

Input: DFG model in graph database with split and join nodes

```
function discoverANDsplitJoinBlock(DFG, t, S, C, F, counter)
    do
        A ← ∅
        for s1 ∈ S do
            CF_s1 ← C[s1] ∪ F[s1]
            for s2 ∈ S do
                CF_s2 ← C[s2] ∪ F[s2]
                if CF_s1 = CF_s2 ⋀ s1 ≠ s2 then
                    Add s2 to A
            If A ≠ ∅ then
                add s1 to A
                break
        If A≠ ∅ then
            U ← getAllJoinNodes
            P ← allPathVariantsFromEntranceToExit(A, U)
            for p in P :
                V ← getValidBlocks(p)
                for v in V :
                    icrHandler(v)    // Incomplete Concurrent
                      Relationship
            for u in V :
                While true :
                    V, status ← mergeBlocks(u, V)
                    if status is not true then break
            ℋ ← getTheNextHierarchies(V)
            for h in ℋ :
                g+ ← insertANDsplitGateway(t, h, counter)
                Add u to U+        // add all ANDjoin later
                counter = counter + 1
                e ← getEntrancePairs(h)
                for i in range(|e|):
                C_u ← C[e[i]]
                F_i ← F[e[i]]
                remove e[i] from S
                remove e[i] from C
                remove e[i] from F
            Add g+ to S
            C[g+] ← C_u
            F[g+] ← F_i
```

have conditions that require the ability to detect split-join block hierarchies. In addition, the green line and yellow line data contain ICR cases. Meanwhile, red line data requires the ability to detect hierarchies with blocks of the same type (XOR block hierarchy in XOR blocks, AND block hierarchies in AND blocks).

The next type of data is public real-life data from BPIC12[1] and BPIC17.[2] These two public data are chosen because both contain long parallel patterns. Data goes through a preprocessing process to select a combination of two sub-processes that contain parallel (long) blocks. The result of BPIC12 data selection is BPIC12_AO data with code A indicating the flow of business processes related to Application activities, and O is Offer activities. Meanwhile, from BPIC17, BPIC17_AW is obtained with code A for Application activities and W for Work_item activities related to the Application.

[1]https://data.4tu.nl/articles/_/12689204/1
[2]https://data.4tu.nl/articles/_/12696884/1

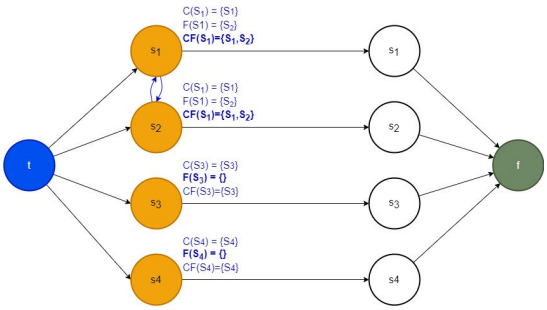


FIGURE 4. An example of a directly follows graph (DFG) model with one split-node and one join-node.

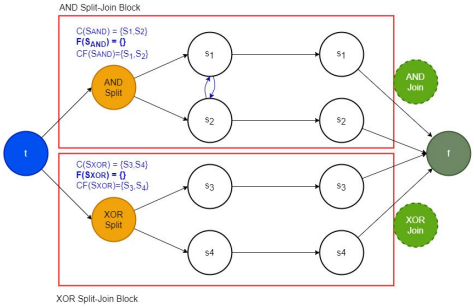


FIGURE 5. The first iteration of the block discovery algorithm gets one AND block and one XOR block.

E. EXPERIMENTAL SETTING

As a preparatory step, we are using pm4py to load the event log data as an event log object. Then the event log is grouped based on its variant. Each variant has a frequency value. Variants with very small frequencies are assumed to be noise so they need to be filtered. After that, the event log that has been cleaned of noise is saved in csv format. The data in csv form is then loaded into the Neo4j using the first stage of the GPD algorithm which produces a trace model and a DFG model.

After the DFG model has been built, the trace model can be removed. By using this DFG model, the GPD++ algorithm starts working.

To test the ability of GPD++ as a process discovery algorithm, an experiment was carried out by comparing it with IM and SM. Performance quality is presented in terms of fitness, precision, and fscore.

V. RESULT AND DISCUSSION

In this section, the experimental results are presented. Then, based on these results an analysis was carried out to compare the capabilities of GPD++ against state-of-the-art process discovery.

Of the five types of data used in the experiment, data from the TOS:green line will be used as an example for discussion. Experiments with the IM algorithm were carried out using the IM plugin on ProM 6.11 tools. Meanwhile, experiments with the SM algorithm were carried out through the Apromore application.

IM discovery results are the Petri Net model which can be seen in Figure 8. The results of discovery using SM are the BPMN models presented in Figure 9.

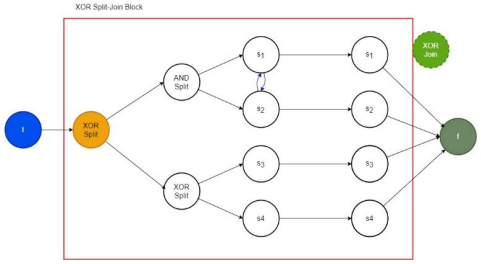


FIGURE 6. The second iteration succeeds in identifying another XOR block which is the hierarchical parent of the AND block and XOR block.

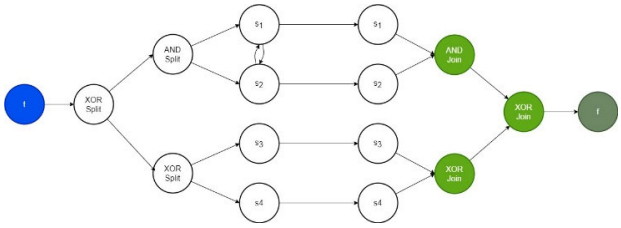


FIGURE 7. After all the split gates are found, then all the join gates are inserted according to the locations that were recorded when the block was found.

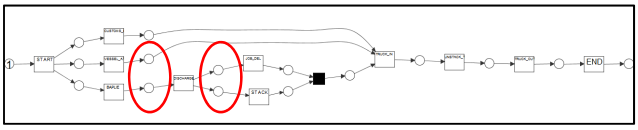


FIGURE 8. Discovery results using inductive miner algorithm.

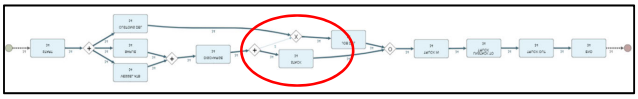


FIGURE 9. Discovery results using split miner algorithm.

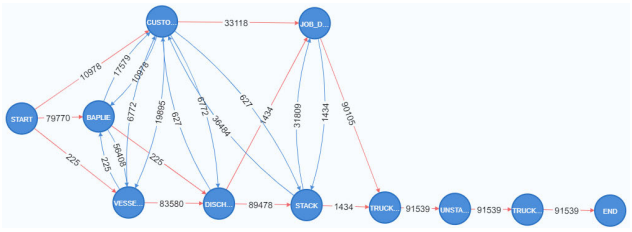
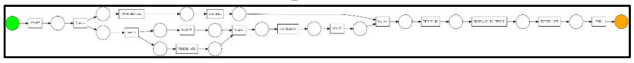


FIGURE 10. Directly follows graph representation of green line real-life data import container activities.

For the GPD++ algorithm, data is first loaded into the Neo4j database and compiled and assembled as a directly follows graph (DFG) model so that the DFG model is obtained in Figure 10. Then by using algorithms 2 and Algorithm 3, the Petri Net model is obtained on the basis Neo4j data as shown in Figure 10(a). Nodes in red are representations of places in the Petri Net. This model is then transformed with the help of PM4Py into a Petri Net object and saved in PNML format. The PNML file can be displayed as a Petri Net as presented in Figure 10 (b).

Furthermore, the model generated through automatic discovery using the IM, SM, and GPD++ algorithms is

(a) Discovery result in graph-based Petri Net Representation



(b) Discovery result in Graphical Representation from the PNML file

FIGURE 11. Discovery results using GPD++.

TABLE 3. Experimental results of finding a model containing hierarchy AND split and join, and incomplete concurrent patterns using real-life data.

Algorithm	Quality measurement	Fitness	Precision	fscore
IM (Leemans et al., 2013)	Alignment-based	1.00	0.897	0.945
	Markovian-based	1.00	0.220	0.358
SM (Augusto et al., 2019)	Alignment-based	-	-	
	Markovian-based	1.00	0.020	0.043
GPD++	Alignment-based	**1.00**	**0.905**	**0.950**
	Markovian-based	**1.00**	**0.350**	**0.524**

TABLE 4. Comparison results IM, SM, GPD++.

Data	Algoritma	Fitness	Precision	fscore
TOS: Green line	IM	1,000	0,8979	0,946
	SM	*)	*)	*)
	GPD++	1,000	0,905	**0,950**
	IM	1,000	0,22	0,360
	SM	1,000	0,02	0,039
	GPD++	1,000	0,35	**0,518**
TOS: Yellow line	IM	**0,999**	**0,869**	**0,929**
	SM	*)	*)	*)
	GPD++	**0,999**	**0,869**	**0,929**
	IM	**0,705**	**0,685**	**0,694**
	SM	0,911	0,021	0,041
	GPD++	**0,705**	**0,685**	**0,694**
TOS: Red line	**IM**	**0,996**	**0,996**	**0,996**
	SM	**0,996**	**0,996**	**0,996**
	GPD++	**0,996**	**0,996**	**0,996**
BPIC12_AO	IM	**0,980**	0,541	0,697
	SM	0,803	0,960	0,875
	GPD++	0,925	**0,929**	**0,927**
BPIC17_AW	IM	0,630	**0,736**	0,678
	SM	0,532	0,648	0,584
	GPD++	**0,654**	0,734	**0,692**

measured for its fitness and precision values as a basis for observing the quality of the resulting model. In the case of this green line, the model produced using SM is not sound, so it cannot be evaluated using an alignment-based meter. Therefore, a marker based on Markovian is also used. The measurement results are presented in Table 3.

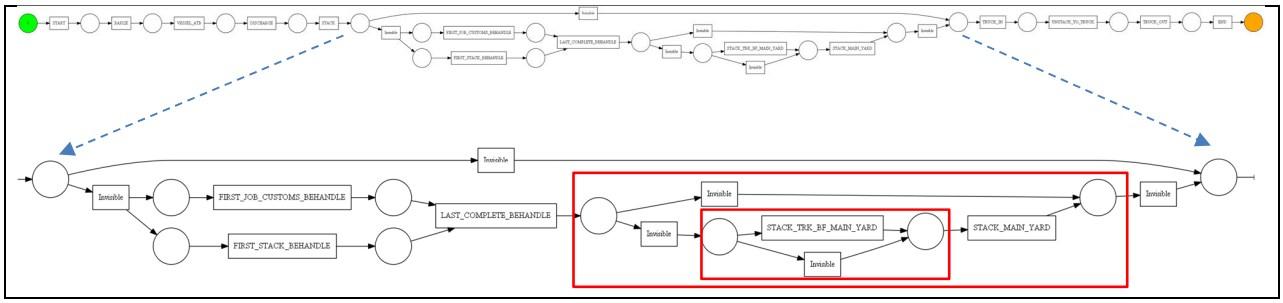


FIGURE 12. Findings of the red line process model using GPD++.

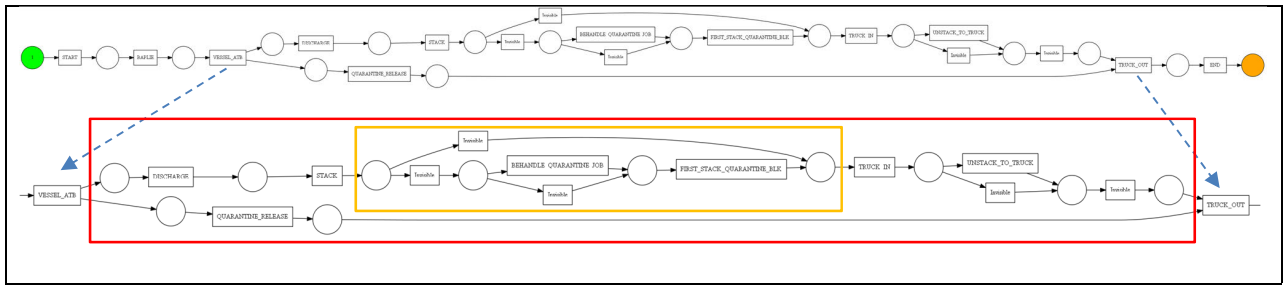


FIGURE 13. Findings of the yellow line process model using GPD++.

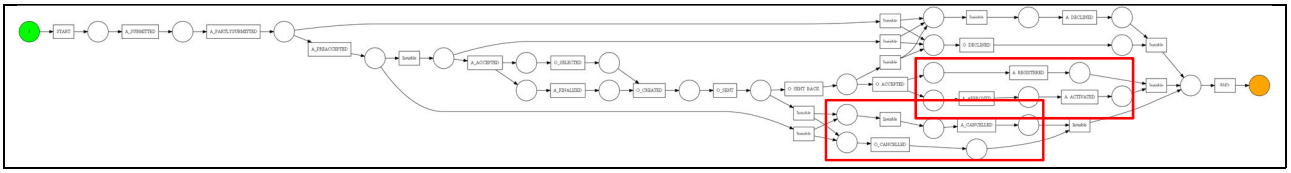


FIGURE 14. Findings process model BPIC12_AO using GPD++.

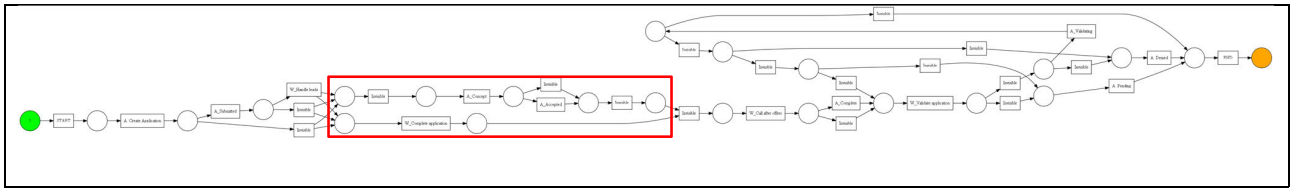


FIGURE 15. Findings of BPIC17_AW process model using GPD++.

In the same way, an experiment was also carried out using data for the red line, yellow line, BPIC12_AO, and BPIC17_AW. The findings of the red line process model in Figure 12 show that GPD++ can build a split-join XOR block hierarchy of the same type (XOR within XOR). Even though there is no ICR problem in the XOR block, the display of a model with a symmetrical block hierarchy (same number of arc-splits and arc-joins) can identify invisible tasks and also provide clarity for the analyst in understanding the structure of the model.

The result of the discovery of the model from the yellow line in Figure 13 shows that GPD++ successfully handles ICR so that it is able to produce long parallel blocks properly, besides that it also appears that GPD++ can detect skip conditions and handle them by adding an invisible task.

The findings of the BPIC12_AO model in Figure 14 show that GPD++ can detect complex block hierarchies, including parallel blocks. But the problem is the flow from outside the block (alien flow) that enters the parallel block. Flow like this should not be allowed to occur in parallel blocks. Alien flow will have an impact on the missing token so that it will reduce the fitness value.

The model obtained from the BPIC17_AW data in Figure 15 shows that GPD++ can recognize and build loop blocks. It can also be observed that the emergence of alien flows from XOR-split to parallel blocks causes GPD++ to fail to build AND-join gates.

A summary of the results of comparing the quality of the models produced by IM, SM, and GPD++ on several data sets is presented in Table 4. The rows in the table that are colored gray are the results of markovian measurements.

VI. CONCLUSION

In this paper, we propose an improvement from GPD+, namely GPD++, which can build block structures based on the DFG model of a process model. ICR conditions can be

detected and corrected based on this block structure, especially in parallel blocks. Besides that, GPD++ supports skip identification and handles it by inserting an invisible task. Our contributions through this research include: proposing a split-join block abstraction, an algorithm based on a cypher query language to build split-join blocks, and an algorithm to detect and handle ICR.

The experimental results comparing the quality of the models produced by IM, SM, and GPD++ against some real-life data containing long blocks show that, in general, GPD++ gives the best results, especially in terms of the fscore value.

Future research will include a study to enhance GPD++'s for dealing with alien flow and complex loop structures.

REFERENCES

[1] W. M. P. van der Aalst, "Process discovery from event data: Relating models and logs through abstractions," *WIREs Data Mining Knowl. Discovery*, vol. 8, no. 3, May 2018, Art. no. e1244.

[2] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Scalable process discovery with guarantees," in *Enterprise, Business-Process and Information Systems Modeling* (Lecture Notes in Business Information Processing), vol. 214. Cham, Switzerland: Springer, 2015, pp. 4–5.

[3] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, and A. Polyvyanyy, "Split miner: Automated discovery of accurate and simple business process models from event logs," *Knowl. Inf. Syst.*, vol. 59, no. 2, pp. 251–284, May 2019.

[4] T. Murata, "Petri nets: Properties, analysis and applications," *Astrophys. J.*, vol. 542, no. 2, pp. L115–L118, 2000.

[5] M. Dumas, M. L. Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management*, vol. 64, no. 1. Berlin, Germany: Springer, 2018.

[6] W. M. P. van der Aalst, "A practitioner's guide to process mining: Limitations of the directly-follows graph," *Proc. Comput. Sci.*, vol. 164, pp. 321–328, Jan. 2019.

[7] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from incomplete event logs," in *Application and Theory of Petri Nets and Concurrency*. Cham, Switzerland: Springer, 2014, pp. 91–110.

[8] R. Sarno and K. R. Sungkono, "A survey of graph-based algorithms for discovering business processes," *Int. J. Adv. Intell. Informat.*, vol. 5, no. 2, p. 137, Jul. 2019.

[9] I. Waspada, R. Sarno, and K. Sungkono, "An improved method of parallel model detection for graph-based process model discovery," *Int. J. Intell. Eng. Syst.*, vol. 13, no. 2, pp. 127–139, Apr. 2020.

[10] I. Waspada, R. Sarno, E. S. Astuti, H. N. Prasetyo, and R. Budiraharjo, "Improving parallel pattern discovery from directly follows graph model," in *Proc. 5th Int. Seminar Res. Inf. Technol. Intell. Syst. (ISRITI)*, Dec. 2022, pp. 769–774.

[11] B. van Dongen, "BPI challenge 2012," Version 1. 4TU.ResearchData, 2012. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2012/12689204, doi: 10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.

[12] B. van Dongen, "BPI challenge 2017," Version 1. 4TU.ResearchData, 2017. [Online]. Available: https://data.4tu.nl/articles/dataset/BPI_Challenge_2017/12696884, doi: 10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b.

[13] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Scalable process discovery and conformance checking," *Softw. Syst. Model.*, vol. 17, no. 2, pp. 599–631, Jul. 2016.

[14] R. Sarno, K. R. Sungkono, M. Taufiqulsa'di, H. Darmawan, A. Fahmi, and K. Triyana, "Improving efficiency for discovering business processes containing invisible tasks in non-free choice," *J. Big Data*, vol. 8, no. 1, pp. 1–17, Dec. 2021.

[15] W. van der Aalst, *Process Mining: Data Science in Action*. Berlin, Germany: Springer, 2016.

[16] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," *Int. J. Cooperat. Inf. Syst.*, vol. 23, no. 1, Mar. 2014, Art. no. 1440001.

[17] S. Esser and D. Fahland, "Multi-dimensional event data in graph databases," *J. Data Semantics*, vol. 10, nos. 1–2, pp. 109–141, 2021.

[18] R. Sarno, K. Sungkono, R. Johanes, and D. Sunaryono, "Graph-based algorithms for discovering a process model containing invisible tasks," *Int. J. Intell. Eng. Syst.*, vol. 12, no. 2, pp. 85–94, Apr. 2019.

[19] A. Seeliger, A. S. Guinea, T. Nolle, and M. Mühlhäuser, "ProcessExplorer: Intelligent process mining guidance," in *Business Process Management* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2019, pp. 216–231.

[20] W. van der Aalst, "Using process mining to bridge the gap between BI and BPM," *Computer*, vol. 44, no. 12, pp. 77–80, Dec. 2011.

[21] L. Reinkemeyer, "Process mining in action," in *Process Mining in Action: Principles, Use Cases and Outloook*. Cham, Switzerland: Springer, 2020.

INDRA WASPADA is currently pursuing the Ph.D. degree in computer science with Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia. He is also a Lecturer and a Researcher with the Department of Computer Science, Universitas Diponegoro, Semarang. His current research mainly focuses on process mining. He is also interested in data mining and business process management.

RIYANARTO SARNO (Senior Member, IEEE) received the Ph.D. degree, in 1992. He is currently a Professor with the Informatics Department, Institut Teknologi Sepuluh Nopember (ITS). He is interested in research projects about machine learning, the Internet of Things, knowledge engineering, enterprise computing, and information management. He has researched process mining for a period of five years. Being the author of more than five books and over 300 scientific articles, led him incorporated in the top 2% world ranking scientist by Standford University, in 2020.

ENDANG SITI ASTUTI is a Full Professor with Brawijaya University. She is also a member of the PPIKID Team and the UB Lecturer Certification Assessor. Her research interests include business administration, information system management, perceptions of digital technology, leadership, entrepreneurship, e-commerce, and knowledge management.

HANUNG NINDITO PRASETYO received the B.Sc. degree from the Mathematics Study Program, Universitas Pendidikan Indonesia (UPI), in 2003, and the M.E. degree in informatics from Institut Teknologi Bandung, in 2013. He is currently pursuing the Ph.D. degree in computer science with Institut Teknologi Sepuluh Nopember, Surabaya. He is also a Faculty Member with Telkom University. His research interests include databases, business processes, process mining, and the development of applications and information systems.

RADEN BUDIRAHARJO is currently pursuing the Ph.D. degree with the Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia. He is also a Faculty Member with the Department of Information Systems, Institut Teknologi Nasional (Itenas), Bandung, Indonesia. His current research interests include process mining, data mining, machine learning, IT governance, and other topics related to information systems and computer science fields of study.