Received: 27 November 2023 | Revised: 23 January 2024 | Accepted: 2 February 2024 | Published online: 4 February 2024

RESEARCH ARTICLE

Journal of Data Science and Intelligent Systems
2024, Vol. 00(00) 1–8
DOI: 10.47852/bonviewJDSIS42022166

BON VIEW PUBLISHING

Generating LTL Formulas for Process Mining by Example of Trace

Kota Komatsu[1] and Hiroki Horita[1,*]

[1]*Graduate School of Science and Engineering, Ibaraki University, Japan*

Abstract: Process mining enables efficient and exhaustive analysis of business processes based on event data. Process mining tools such as linear temporal logic (LTL) Checker allow users to verify temporal properties of business processes for traces by providing a description based on LTL. However, it is difficult for many users to understand and use LTL-based mathematical notation. Therefore, there is a need for a method to describe temporal properties even for those who are not familiar with mathematical notation. Several methods have been proposed to automatically generate logical expressions to address these issues, but there are still problems related to the data format and the limited scope of the expressions that can be described. In this study, we proposed a method based on the satisfiability problem (SAT) for event logs in eXtensible Event Stream format used in process mining, and it is verified how well it can automatically generate temporal properties in business processes. We conducted experiments using two types of event logs to demonstrate the effectiveness of the proposed method.

Keywords: process mining, business process, linear temporal logic, satisfiability problem

1. Introduction

Process mining Dumas et al. (2018) is a data analysis method that visualizes event log data, which represent the execution history of business processes, to support the analysis and improvement of business operations.

Linear temporal logic (LTL) Checker can verify whether a property is true for each trace that constitutes the event log of a business process, using the LTL language based on LTL (Pnueli, 1977). That is, the event log can be divided into two groups of traces, one satisfying the property to be checked and the other not satisfying the property, and the event log can be analyzed from various viewpoints. For example, by using a process discovery algorithm (Ferreira & Gillblad, 2009; Van derWerf et al., 2009), a business process model can be automatically generated from event logs to visualize the flow of business processes. By comparing business process models created from event logs that satisfy certain properties with business process models created from event logs that do not satisfy certain properties, it may be possible to understand the influence of certain properties on business processes. Various other analysis methods can be applied after the event log is split, such as goal-oriented process mining (Ghasemi & Amyot, 2020) and conformance checking (Dunzer et al., 2019). LTL Checker assumes that the user writes and understands correct logical expressions. However, it is difficult for many users to understand and use LTL-based mathematical notation, and it is difficult to correctly describe the temporal properties of business processes. Therefore, a method for describing logical expressions is required for those who are unfamiliar with logic.

There are several techniques (Chesani et al., 2022; Horita et al., 2016; Neider & Gavran, 2018) that automatically generate logical expressions by giving examples of traces where certain properties are true and others where they are false. However, Chesani et al. (2022) and Horita et al. (2016) are limited in the logical expressions it can generate. Their method can only generate logical expressions according to a predefined template. In addition, Neider and Gavran (2018) do not target the data format of event logs in process mining. Their method did not support xes, a common data format in process mining.

In this study, we propose a method to generate LTL expressions containing various modal logic operators for eXtensible Event Stream (XES) event logs, a common data format in process mining. In this study, the event logs were converted into a format compatible with methods based on the satisfiability problem (SAT) for generating logical expressions (Neider & Gavran, 2018). The description method of the traces given as input values of Neider and Gavran (2018) is different from the general event log description method and is written in a language on the alphabet {0,1}. Therefore, we implemented a tool for one hot encoding of event logs into a sequence of numbers on the alphabet {0,1}. After conversion to a sequence of numbers using this tool, the user can select several positive and negative traces from the event log and generate LTL expressions containing various modal logic operators by using SAT-based methods (Neider & Gavran, 2018).

The structure of this paper is as follows. Section 2 introduces the basic knowledge and related studies. Section 3 describes the proposed method. Section 4 presents the results and a discussion of the evaluation experiments using the proposed method. Section 5 summarizes the study and discusses future works.

2. Preliminaries and Related Research

This section describes the knowledge and research related to this study.

*Corresponding author: Hiroki Horita, Graduate School of Science and Engineering, Ibaraki University, Japan. Email: hiroki.horita.is@vc.ibaraki.ac.jp

2.1. Linear temporal logic

LTL (Pnueli, 1977) is any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time (for example, "I am always hungry", "I will eventually be hungry", or "I will be hungry until I eat something"). LTL allows us to express future events in terms of logical expressions such as whether a certain condition is eventually true.

LTL uses the variables $p_1, p_2, \ldots$, the general logical operators $\neg$, $\wedge$, $\vee$, $\rightarrow$, and the modal logical operator $\bigcirc$(next), $\Box$(globally), $\Diamond$(finally), $\mathcal{U}$(until), $\mathcal{R}$(release). Since the three logical operators $\bigcirc$, $\Box$, $\Diamond$ are unary if φ is a logical expression, then $\bigcirc\varphi$ is also a logical expression. Since the two logical operators $\mathcal{U}$ and $\mathcal{R}$ are binary operators, if φ and ψ are a logical expression, then $\varphi\mathcal{U}\psi$ is also a logical expression. The logical operators of LTL are listed in Table 1.

Table 1
LTL modal logic operator

LTL formula	Explanation of formula
$\bigcirc\phi$	ϕ has to hold at the next state
$\Box\phi$	ϕ has to hold on the entire subsequent path
$\Diamond\phi$	ϕ eventually has to hold
$\psi\mathcal{U}\phi$	ψ has to hold at least until ϕ becomes true, which must hold at the current or a future position
$\psi\mathcal{R}\phi$	ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever

2.2. Event log

Process mining tools such as ProM (Process Mining Framework) use information system logs. A log consists of several traces, and the traces consist of several events. An activity is also a label that defines the specific task included in an event. The general event log in this paper refers to XES , which is an event data storage and management format defined as an open standard by the IEEE (Gunther & Verbeek, 2014). Figure 1 shows an example of an XES document. Each trace and event store a literal attribute with the key "concept:name". Each trace is tagged with <trace> and the caseID is stored as an attribute value. Each event is tagged with <event> and the activity is stored as an attribute value. The attribute value is represented by a letter of the alphabet (A, B, C, and D). For example, a trace with caseID 1 indicates that events A, B, C, and D are executed sequentially.

Figure 1
Example of XES document

```
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="1.0" xes.features="nested-attributes "openxes.version="1.0RC7">
    <global scope="trace"><string key="concept:name" value="__INVALID__"/></global>
    <global scope="event"><string key="concept:name" value="__INVALID__"/></global>
    <trace>
        <string key="concept:name" value="1"/>
        <event><string key="concept:name" value="A"></event>
        <event><string key="concept:name" value="B"></event>
        <event><string key="concept:name" value="C"></event>
        <event><string key="concept:name" value="D"></event>
    </trace>
    <trace>
        <string key="concept:name" value="2"/>
        <event><string key="concept:name" value="A"></event>
        <event><string key="concept:name" value="C"></event>
        <event><string key="concept:name" value="B"></event>
        <event><string key="concept:name" value="D"></event>
    </trace>
</log>
```

2.3. LTL checker

LTL Checker (van der Aalst et al., 2005) uses LTL-based notation to describe the temporal properties desired by the user and automatically verifies that the traces to be verified satisfy the desired properties.

The use of LTL Checker is explained with an example. Table 2 shows the event transitions for each trace in Figure 1. Table 3 shows the properties desired by the user in the business process, and LTL Checker can be used to verify that each trace in Table 2 satisfies the properties in Table 3. As a result of the actual verification of each property, the trace with caseID 1 satisfies all properties, but the trace with caseID 2 does not satisfy the property "If B is executed, C will eventually be executed". In other words, the property "$\Box$(B $\rightarrow$ $\Diamond$C)" in Table 3 can be said to classify traces with caseID 1 as true and traces with caseID 2 as false.

Table 2
Example of event log

caseID	Event transition
1	A → B → C → D
2	A → C → B → D

Table 3
Examples of properties to be verified

LTL formula	Explanation of formula
$\Diamond$A	A has to hold at the next state
$\Box$(B $\rightarrow$ $\Diamond$C)	If B is executed, C will eventually be executed

2.4. Related work

There are several methods to automatically generate logical expressions in process mining.

Horita et al. (2016) and Chesani et al. (2022) can automatically generate logical expressions from the event log. However, Horita et al. (2016) can only use some operators, which limits what can be described. In addition, Chesani et al. (2022) can only generate logical expressions that follow the template of the DECLARE model (Maggi, 2013; Pesic, 2008).

In addition, there are several studies (Camacho & McIlraith, 2019; Gaglione et al., 2021; Neider & Gavran, 2018; Raha et al., 2022) that aim to explain the temporal behavior of the system in order to increase the interpretability of the system. In Neider and Gavran (2018) and Gaglione et al. (2021), a learning algorithm is proposed for logical expressions based on the satisfiability problem (SAT). Given an input value, a sample S consisting of positive and negative examples, the method can automatically generate a temporal property satisfying the description as a logical expression of the minimum size. However, this method does not target the XES event logs used in the process mining.

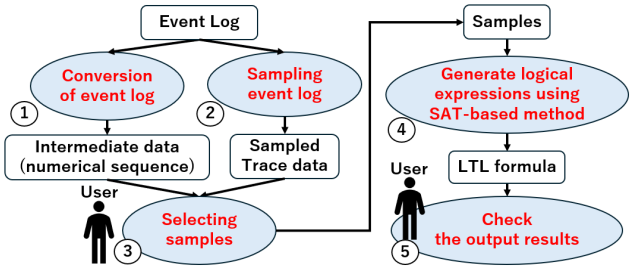
This study deals with the issue of limited description of generated LTL formulas in previous studies (Horita et al., 2016; Chesani et al., 2022) and the issue of not supporting event logs in XES in previous study (Neider & Gavran, 2018).

3. Proposed Method

In this section, we explain a SAT-based method (Neider & Gavran, 2018) for dealing with general event logs and explain

how to investigate the expressive ability of the logical expressions generated using this method. Figure 2 shows an overview of this method. In Figure 2, the rounded rectangles represent the input/output elements in the proposed method and the ovals represent the tasks that handle the elements. The tasks marked with a user icon indicate that they include tasks that require manual operation by the user. Each task is executed in the order of 1 to 5 (1 and 2 are executed in arbitrary order).

Figure 2
Overview of the proposed method



3.1. Conversion of event logs

The SAT-based logical expression generation method Neider and Gavran (2018) uses sample $\mathcal{S}$ as the input value, so it is necessary to convert the event log to sample $\mathcal{S}$. Sample $\mathcal{S}$ consists of positive and negative traces. Positive traces are those that satisfy the properties of the logical expression they generate, and negative traces are those that violate the properties of the logical expression they generate. The description format of Sample S is one hot encoding of the activities in each trace, separated by ";". At each timestamp separated by ";" in the trace, an event that was executed is true (1) and an event that was not executed is false (0). For example, the event log of Table 2 with one hot encoding is shown in Table 4. A single trace can be distinguished by a line break, and multiple traces can be distinguished into positive or negative traces.

Table 4
Numerical sequence converted using Table 2 as input

caseID	Event transition	Numerical sequence for each trace
1	A → B → C → D	1,0,0,0; 0,1,0,0; 0,0,1,0; 0,0,0,1
2	A → C → B → D	1,0,0,0; 0,0,1,0; 0,1,0,0; 0,0,0,1

In this study, we implemented a tool for one hot encoding of event logs for each trace. When the tool is given an event log in the XES format as input, it can output as intermediate data a sequence of numbers on the alphabet {0,1}, which is the real data constituting the sample $\mathcal{S}$ (Task 1 in Figure 2).

In order to do one hot encoding of the event log for each trace, this study introduced a tuple T_i = <*TraceID*, *Encoded_T*> for each trace and a tuple A_j = <*ActID*, *ActivityName*, *Encoded_A*> for each activity, and variables *SumOfTrace* and *SumOfActivity*. Both *i* and *j* are natural numbers, where $i \in$ *TraceID* and $j \in$ *ActID*, respectively. Table 5 shows a summary of each variable. A concrete conversion method using these variables is shown in Algorithm 1. In Algorithm 1, "←" indicates substitution, "+" indicates addition in the case of numbers, and "+" indicates connection in the case of strings. In Algorithm 1, when *Encoded_T* of the *i*-th trace is referenced, it is written as T_i. *Encoded_T*.

Table 5
Variables used in Algorithm 1

Variable name	Details
TraceID	Trace identifier (natural number)
Encoded_T	Numeric sequence of traces
ActID	Activity identifier (natural number)
Encoded_A	Numeric sequence of activity
ActivityName	Activity name
SumOfTrace	Total number of traces
SumOfActivity	Total number of activities

3.1.1. Read information about traces, events, and activities

First, when the event log is given as input, the identifier of each trace (*TraceID*), the total number of traces (*SumOfTrace*), and the total number of activities (*SumOfActivity*) are read (line 1). Each trace and event information are read with the <trace> tag and <event> tag. The <event> tag assigns an identifier (*ActID*) to the activity that is the attribute value. The <trace> tag records the identifier of each trace (*TraceID*). After reading the event log, the total number of traces (*SumOfTrace*) and the total number of activities (*SumOfActivity*) are recorded.

Algorithm 1
Conversion method from event log to numerical sequence

Algorithm 1 Conversion method from event log to numerical sequence
Input: Event log(.xes)
output: Tuple T_{Vi} ($0 \leq i <$*SumOfTrace*)

```
1: The information about traces, events, and activities is read.
2:
3: for each Activity A_j do
4:    n ← 0
5:    A_j.Encoded_A ← []
6:    while n <SumOfActivity do
7:       if n = A_j.ActID then
8:          A_j.Encoded_A ← A_j.Encoded_A + "1" //connection
9:       else
10:         A_j.Encoded_A ← A_j.Encoded_A + "0" //connection
11:      end if
12:      if n != SumOfActivity then
13:         A_j.Encoded_A ← A_j.Encoded_A + "," //connection
14:      end if
15:      n ← n + 1 //addition
16:   end while
17: end for
18:
19: i ← 0
20: while i < SumOfTrace do
21:    T_i.Encoded_T ← []
22:    for each Event do
23:       for each Activity A_j do
24:          if EventName = A_j.ActivityName then
25:             j ← A_j.ActID
26:          end if
27:       end for
28:       T_i.Encoded_T ← T_i.Encoded_T + A_j.Encoded_A //connection
29:       if Event is not in the last state then
30:          T_i.Encoded_T ← T_i.Encoded_T + ";" //connection
31:       end if
32:    end for
33:    i ← i+ 1 //addition
34: end while
```

3.1.2. Convert each activity to a numerical sequence

After reading the information required for one hot encoding, perform the conversion to a numerical sequence for each activity (*Encoded_A*) (lines 3–17). For example, given the event logs in Table 2 as input, the numerical sequence for each activity (A, B, C, D) is converted, as shown in Table 6. As a specific action, first, for each activity, list 0 until the total number of activities (*SumOfActivity*). However, if counter *n* matches the identifier (*ActID*), it lists 1 instead of 0. In the case of Table 6, identifiers are assigned to activities A, B, C, and D as A ← 0, B ← 1, C ← 2, D ←3. Since the total number of activities (*SumOfActivity*)

Table 6
Numerical sequence of activities converted using Table 2 as input

Identifier	Activity	Numerical sequence for each activity
1	A	1,0,0,0
2	B	0,1,0,0
3	C	0,0,1,0
4	D	0,0,0,1

is 4, the numerical sequence for each activity consists of three zeros, and only the part corresponding to the identifier (0 on the left end and *SumOfActivity*-1 on the right end) consists of ones. In this way, a numerical sequence *Encoded_A* can be defined for each activity.

3.1.3. Convert each trace to a numerical sequence

After converting each activity to a numerical sequence (*Encoded_A*), the event log is read again and converted into a numerical sequence for each trace (lines 19–34).

For each event in the trace, identify the activity name (A_j.*ActivityName*) and refer to the corresponding activity identifier $j = ActID$. Referring to A_j.*Encoded_A* from the identifier j, the order of execution of the events is considered as timestamps and distinguished by ";". By repeating this process, the entire event log can be one hot encoded for each trace. The final output is $T_{\forall i}$.*Encoded_T* ($0 \leq i < SumOfTrace$), which is a pair of the trace identifier (*TraceID* = i) and the actual numerical sequence (*Encoded_T*). With this conversion process, the input event log is performed in one hot encoding for each trace.

Table 4 is the result of one hot encoding of the event log in Table 2 based on the numerical sequence in Table 6. In addition, all traces converted by this tool are finite.

3.2. Sampling event log and selection of samples

When selecting the sample, the event log is sampled (Task 2 in Figure 1). This study uses the sampling technique proposed by Bernard and Andritsos (2021). This is a method of sampling representative traces using Earth Mover's Distance (EMD), a measure of dissimilarity between two multidimensional distributions. In this study, we use this method because of its ability to specify the number of traces to be sampled and to sample traces that are representative of the entire event log.

Bernard and Andritsos (2021) have three methods: a normal method, an Expected Occurrence Reduction method that preselects simple representatives, and a Euclidean method that works in Euclidean space. Since the dataset size handled in the evaluation experiments is not large, the difference in execution time between these three methods is small. Therefore, the normal method was used in this study. The number of traces sampled in this study was set to 10.

The user arbitrarily selects positive and negative traces from the intermediate data generated in Section 3.1 to construct the input values (sample $\mathcal{S}$) (Task 3 in Figure 2).

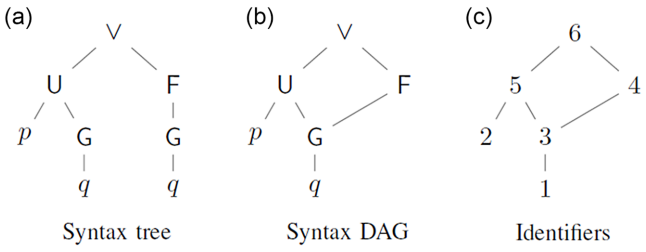
3.3. Generate logical expressions using SAT-based method

The satisfiability problem (SAT) is the problem of determining whether there exists a true-false assignment of φ to a variable in conjunctive normal form such that φ is true given a logical expression φ. The satisfiability problem is one of the NP-complete problems. The SAT solver is a tool to quickly determine whether a propositional logical expression is satisfiable, and if so, it can output its true/false assigned value.

The SAT-based learning algorithm (Neider & Gavran, 2018) is an algorithm that outputs an LTL formula that satisfies the description of a given sample $\mathcal{S}$. The SAT-based learning algorithm is characterized by its ability to learn logical expressions of minimal size and its independence from templates such as DECLARE (Maggi et al., 2011). To determine the satisfiability of an LTL formula using the SAT solver, Neider and Gavran (2018) defined constraints that restrict the syntax of the logical expression and constraints that define the LTL semantics of the expression. We say that a sample $\mathcal{S}$ is satisfiable if we can correctly classify each trace of the samples as positive or negative in the SAT-based learning algorithm.

For syntactic constraints, Neider and Gavran (2018) focused on the tree structure of the LTL formula, defining labels for each node and parent–child relationships among nodes. Figure 3 (Neider & Gavran, 2018) shows an example of a syntax tree and syntax DAG for the LTL formula ($p\,\mathcal{U}\,\square q)\vee(\Diamond(\square q))$. Syntax DAG (a directed acyclic graph) is one of the regular syntactic representation methods in which common subexpressions are shared. In Figure 3(c), the identifier of the root of the syntax DAG of the LTL formula is $n \in \mathbb{N}\backslash\{0\}$, and if the internal node is $i \in \{1,2, \ldots, n\}$, the unique identifier i is assigned so that the identifier of its children is smaller than i. The syntax constraints focused on the syntax DAG are as follows: (1) Each node has one label. (2) Each node (except node 1) has a left child and a right child. However, if a node represents a unary operator or an atomic proposition, the specific child can be ignored. (3) Node 1 is labeled with an atomic proposition. From these constraints, a syntactically correct LTL formula can be generated as a candidate solution.

Figure 3
Syntax tree, syntax DAG, and identifiers of the syntax DAG for the LTL formula ($p\,\mathcal{U}\,\square q)\vee(\Diamond(\square q))$



For semantics constraints, Neider and Gavran (2018) introduced constraints on the modal logic operators in Table 1 to evaluate the satisfiability of LTL. Constraints on the semantics for evaluating LTL formulas are as follows: (1) atomic propositions, (2) negation ($\neg$), (3) logical disjunction ($\vee$), (4) next ($\bigcirc$), (5) until ($\mathcal{U}$), (6) logical conjunction ($\wedge$), (7) implication ($\rightarrow$), (8) globally ($\square$), and (9) finally ($\Diamond$). By giving these constraints to the SAT solver, the satisfiability of LTL formula can be evaluated. Detailed definitions of each constraint are given in the paper by Neider and Gavran (2018).

For example, consider the output of the solution when the trace with caseID 1 in Table 4 is positive, the trace with caseID 2 is negative, and the operators (constraints) that can be used are restricted to $\Diamond$ and $\mathcal{U}$. The size (search depth) n of the expression has a default value of 0, and the activities are assigned variables based on their identifiers, such as A←"x0", B←"x1", C←"x2", D←"x3". List these variables and the operators used. Then, the constraints are input to the SAT solver to determine their satisfiability. Specifically, it starts with a value of n of 1 and randomly generates logical expressions of size 1 as candidate

solutions based on sample description, syntax, and semantic constraints. When the size is 1, the syntax constraint makes the generated expression an atomic proposition, so "x0", "x1", etc., are possible candidates for the solution. If there is no candidate solution, it is determined to be unsatisfiable and the value of n is increased by 1, and a candidate solution is generated again. In this example, when the value of n is 3, the candidate solution "x0 $\mathcal{U}$ x1" that is determined to be satisfiable is output as the solution.

By giving the sample $\mathcal{S}$ selected in Section 3.2 to the SAT-based learning tool implemented by Neider and Gavran (2018), the logical expression of the smallest size can be generated automatically (Task 4 in Figure 2). In this study, we assume the automatic generation of logical expressions in the actual work and do not manipulate the parameters related to execution.

3.4. Check the output results

It is necessary to manually record and analyze the logical expressions generated in Section 3.3. This corresponds to Task 5 in Figure 2. In the evaluation experiment, 10 sample traces are labeled as positive and negative examples by the true or false of a known LTL formula, and the SAT-based method is used to learn LTL formulas using these as input. Using this learned expression, the entire data set is determined to be true or false by LTL Checker, and the percentage of traces whose true/false results are consistent with the known LTL formulas is evaluated.

4. Evaluation Experiments

This section describes the purpose and overview of the experiment, the tools, and event logs used in the experiment, shows the results of the experiment, and finally discusses the results.

4.1. Purpose and overview of the experiments

A SAT-based method of generating logical expressions is used by those not familiar with LTL-based mathematical notation to describe the temporal properties of event logs using various LTL logic operators. Specifically, the event log in XES format is converted to the input format used by SAT-based methods to automatically generate temporal properties. We also investigate the percentage of correct answers that can be calculated from the known LTL formulas and generated LTL formulas in order to verify what properties are generated. To show that the method proposed by Neider and Gavran (2018) can be used for event logs in XES format, this study conducted experiments following the proposed method in Section 3.

The experiment was conducted as follows.

(1) Prepare an event log.
(2) Convert the event logs into the languages on the {0, 1} alphabet using the conversion algorithm described in Section 3.1.
(3) Sample 10 traces of the prepared event logs using sampling method (Bernard & Andritsos, 2021).
(4) Manually classify the 10 traces into true traces that satisfy certain properties and false traces that do not satisfy certain properties.
(5) Generate logical expressions from the true and false trace groups using Neider and Gavran's method (Neider & Gavran, 2018).
(6) Classify the event log of (1) into true traces and false traces using the generated LTL formula.
(7) Compare the results of (6) with the results of classifying the event log in (1) using the correct LTL formula (known LTL formula) to obtain the percentage of correct answers.

In this experiment (4), 4 DECLARE templates (Maggi et al., 2011; Pesic, 2008) and 5 common LTL patterns (Dwyer et al., 1998) were targeted. The 4 DECLARE templates we selected are existence and absence constraints from the existence template, and response and co-existence constraints from the relationship template. These are used as known LTL formulas in this experiment because, compared to other templates, they show basic properties such as existence and execution order with respect to single events and between events. The 5 general LTL patterns we selected were among the 9 LTL patterns used by Neider & Gavran (2018) in their evaluation experiments, which did not overlap with the DECLARE template and whose expressions were not redundant. For each execution, the duration was set to 30 min.

4.2. Tools and event logs used in the experiments

In this experiment, we used ProM and SAT-based methods to learn logical expressions, a tool for sampling traces, and a transformation tool implemented in this study. The event logs are classified by LTL Checker, a ProM plug-in, and samples are selected using a sampling tool and a conversion tool. Experiments were conducted to generate logical expressions using SAT-based methods with those samples as input values.

Two datasets, "exercise5.xes" and "Sepsis.xes", were used in the experiment. These files are available to anyone on the Web. "exercise5.xes" is the artificial event log of the review process of papers by various reviewers handled in the ProM tutorials, with 2297 events, 100 traces, and 20 activities. "Sepsis.xes" is the real-life event log of sepsis cases recorded by the enterprise resource planning system of a hospital, with 15190 events, 1050 traces, and 16 activities.

4.3. Results of experiments

The results of the runs are shown in Tables 7 and 8. The "Generated LTL formula" shows the logical expressions generated by each execution. For the 9 logical expressions in this experiment, only two of the generated LTL formulas matched the known LTL formulas: existence (Tables 7 and 8, line 1) and absence (Tables 7 and 8, line 2), which are DECLARE templates. For the other 7 logical expressions, there were many cases in which the generated LTL formulas appeared events that were not related to the events selected by the known LTL formulas. For example, in execution against "exercise5.xes", the known LTL formulas are those related to "reject" such as $\Box(\neg$ reject $\vee \Diamond$(reject $\wedge$ time-out X)), but the resulting output expression $\Diamond$accept is related to "accept". All the executions could be performed within the duration, and all of them lasted less than 1 min.

Using the generated LTL formulas in Tables 7 and 8, we verified how the generated LTL formulas with a small number of traces classify the original traces. Specifically, the entire dataset was determined to be true or false by LTL Checker using the learned expressions, and the percentage of traces whose true/false matches with the known LTL formulas was quantitatively evaluated as the "percentage of correct answers". Table 9 shows the distribution of the percentage of correct answers for all logical expressions generated in this experiment.

As a result, the logical expressions generated by each execution showed a high concordance rate. excercise5.xes" showed 7 verifications with a 100% concordance rate. There were 5 verifications for "Sepsis.xes" that showed a concordance rate of 100%. The lowest concordance rate was 63% for the "Sepsis.xes" verification.

Table 7
Execution results for "exercise5.xes"

Classification	Known LTL formula	Generated LTL formula	Correct answers	Correct answers (%)
DECLARE	◇time-out X	◇time-out X	100	100%
	¬(◇time-out X)	¬(◇time-out X)	100	100%
	◇get review 1 ∧ ◇get review X	(time-out 1 → get review X) $\mathcal{U}$ get review X	100	100%
	□(time-out 2 → ◇time-out X)	◇(reject ∨ get review 2)	79	79%
LTL pattern	◇collect reviews → (¬get review 1 $\mathcal{U}$ collect reviews)	◇time-out 1	100	100%
	□(invite reviewers → □(¬get review X))	□(¬invite additional reviewer)	100	100%
	□(¬reject ∨ ◇(reject ∧ time-out X))	◇accept	100	100%
	◇accept → (get review X $\mathcal{U}$ accept)	◇reject	100	100%
	□(time-out X → □invite additional reviewer)	¬(◇invite additional reviewer)	72	72%

Table 8
Execution results for "Sepsis.xes"

Classification	Known LTL formula	Generated LTL formula	Correct answers	Correct answers (%)
DECLARE	◇IV Liquid	◇IV Liquid	1050	100%
	¬(◇IV Liquid)	□(¬IV Liquid)	1050	100%
	◇CRP ∧ ◇Release A	◇Release A	673	64%
	□(CRP → ◇Release A)	◇Release A	667	63%
LTL pattern	◇Release A → (¬ER Sepsis Triage $\mathcal{U}$ Release A)	□(¬Release A)	1049	99%
	□(ER Triage → □(¬Release A))	□(¬Release A)	1050	100%
	□(¬Release A ∨ ◇(Release A ∧ ER Registration))	□(¬Release A)	1050	100%
	◇IV Liquid → (ER Triage $\mathcal{U}$ IV Liquid)	¬(◇IV Liquid)	1018	96%
	□(Release A → □IV Liquid)	□(¬Release A)	1050	100%

Table 9
Distribution of percentage of correct answers for generated LTL formula

Percentage range of correct answers	exercise5.xes	Sepsis.xes
0–60%	0	0
61–80%	2	2
81–99%	0	2
100%	7	5

4.4. Discussion

4.4.1. Known LTL formula and generated LTL formula

The results of the experiments (Tables 7 and 8) show that the automatic generation of logical expressions using a sample consisting of 10 traces was able to generate logical expressions that were consistent with known LTL formulas under some conditions. In 5 LTL patterns, the generated LTL formulas were sometimes written as logical expressions of smaller size than the known LTL formulas. One possible reason why the generated LTL formulas do not match the known LTL formulas is that some traces in the positive trace are filled with vacuity (Kupferman & Vardi, 2003). For example, when the event log is verified using a logical expression containing an implication (→) such as "A → B", any trace in which event A does not exist is considered true. In this case, we say that the logical expression is vacuously satisfied for a trace where event A does not exist. Therefore, in this experiment, the sample selection was performed without considering the vacuity detection, which may have resulted in a large number of candidates for the automatically generated LTL formulas. This has a significant effect when generating a known LTL formula, as in this experiment, but is expected to have almost no effect when generating an unknown LTL formula from a freely given trace. The user's specification of events to be verified may also have a significant effect on the automatic generation of logical expressions. In this study, we used "Mine Petri net with Inductive Miner" plug-in for ProM, to specify events in the execution order that indicate the desired property with reference to the Petri net diagrams generated from each dataset. However, depending on the event to be specified, it is not possible to generate a logical expression indicating the desired property.

4.4.2. Percentage of correct answers to generated LTL formulas

The results of the experiments (Tables 7, 8, and 9) show that the LTL formulas generated by the proposed method have a high percentage of correct answers. In particular, the formula that resulted in 100% showed the same classification results as the known LTL formula when using the LTL Checker, but with different properties. Therefore, even the generation of logical expressions using samples with a small number of traces is likely to generate logical expressions that characterize the original traces. If the total number of traces in the event log is large, it is unlikely that the percentage of correct answers will be 100% because many unique traces will be included.

4.4.3. Selecting traces

The number of traces sampled in our experiment was very small (10 traces) and may not have included enough information to characterize the entire event log. In this study, the EMD value for each dataset was 0.16 for "exercise5.xes" and 0.31 for "Sepsis.xes"

(a smaller EMD value means that the sample trace is more representative of the entire dataset). By clearly defining the standard for the value of EMD, there is a possibility to specify the number of traces to be given as a sample for each size of the dataset. Thus, if the number of traces to be given for each data set is to be changed, it is necessary to define more clearly the "smallness" of the number of traces.

4.4.4. Practicality of the proposed method

From Sections 4.4.1 and 4.4.2, it can be seen that logical expressions generated with inputs consisting of fewer than the original number of traces do not always show the desired temporal properties, but they do characterize the original traces. Therefore, it cannot be said that strict verification is possible, but LTL Checker can be used to roughly grasp the execution trends of business processes.

The number of examples traces in this paper is 10. This is for ease of use. It is believed that more accurate logic formulas can be generated by increasing the number of example traces. On the other hand, it is time-consuming to provide many examples. How to balance effectiveness and practicality is an issue for the future.

5. Conclusion

In this paper, we implemented a tool to convert XES format event logs into input values called Sample $\mathcal{S}$ for a SAT-based logical expression generation method and proposed a method to generate logical expression for use in the LTL Checker. Using this method, logical expressions of temporal properties can be generated from event logs of business processes.

In the evaluation experiments, we investigated the expressive power of logical expressions that can be generated from example event logs in XES. We found that it is likely that a sample consisting of fewer traces than the total number of traces can be used to generate logic expressions that characterize the original traces. We were also able to show the characteristics of the undesirable results.

In the future, we should investigate the logical formulas generated by the sample using the person in charge, work time, etc., and improve the algorithm considering the vacuity of the trace.

Funding Support

This work is supported in part by JSPS KAKENHI under Grants 24K20756.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by any of the authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

The data that support the findings of this study are openly available in [GitHub] at https://github.com/ivan-gavran/samples2LTL, https://github.com/gaelbernard/sampling; in [Prom Tools] at http://promtools.org/doku.php and in [4TU.ResearchData] at https://data.4tu.nl/

The data are also available from the corresponding author Hiroki Horita (hiroki.horita.is@vc.ibaraki.ac.jp) upon reasonable request.

References

Bernard, G., & Andritsos, P. (2021). Selecting representative sample traces from large event logs. In *3rd International Conference on Process Mining,* 56–63.

Camacho, A., & McIlraith, S. A. (2019). Learning interpretable models expressed in linear temporal logic. *Proceedings of the International Conference on Automated Planning and Scheduling, 29*, 621–630.

Chesani, F., Di Francescomarino, C., Ghidini, C., Grundler, G., Loreti, D., Maggi, F. M., . . . , & Tessaris, S. (2022). Shape your process: Discovering declarative business processes from positive and negative traces taking into account user preferences. In *International Conference on Enterprise Design, Operations, and Computing,* 217–234.

Dumas, M., La Rosa, M., Mendling, J., & Reijers, H. A. (2018). *Fundamentals of business process management.* Germany: Springer.

Dunzer, S., Stierle, M., Matzner, M., & Baier, S. (2019). Conformance checking: A state-of-the-art literature review. In *Proceedings of the 11th International Conference on Subject-oriented Business Process Management,* 1–10.

Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1998). Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice,* 7–15.

Ferreira, D. R., & Gillblad, D. (2009). Discovering process models from unlabelled event logs. In *Business Process Management: 7th International Conference,* 143–158.

Gaglione, J. R., Neider, D., Roy, R., Topcu, U., & Xu, Z. (2021). Learning linear temporal properties from noisy data: A MaxSAT-based approach. In *Automated Technology for Verification and Analysis: 19th International Symposium,* 74–90.

Ghasemi, M., & Amyot, D. (2020). From event logs to goals: A systematic literature review of goal-oriented process mining. *Requirements Engineering*, *25*(1), 67–93.

Gunther, C. W., & Verbeek, H. M. W. (2014). XES - Standard definition. *BPM Reports, 1409*.

Horita, H., Hirayama, H., Hayase, T., Tahara, Y., & Ohsuga, A. (2016). Process mining approach based on partial structures of event logs and decision tree learning. In *2016 5th IIAI International Congress on Advanced Applied Informatics,* 113–118.

Kupferman, O., & Vardi, M. Y. (2003). Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer*, *4*, 224–233.

Maggi, F. M. (2013). Declarative process mining with the declare component of ProM. In *Proceedings of the BPM Demo Sessions 2013,* 1021.

Maggi, F. M., Mooij, A. J., & Van der Aalst, W. M. (2011). User-guided discovery of declarative process models. In *2011 IEEE Symposium on Computational Intelligence and Data Mining,* 192–199.

Neider, D., & Gavran, I. (2018). Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design,* 1–10.

Pesic, M. (2008). *Constraint-based workflow management systems: Shifting control to users*. PhD Thesis, Eindhoven University of Technology.

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science,* 46–57.

Raha, R., Roy, R., Fijalkow, N., & Neider, D. (2022). Scalable anytime algorithms for learning fragments of linear temporal logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 263–280.

van der Aalst, W. M., de Beer, H. T., & van Dongen, B. F. (2005). Process mining and verification of properties: An approach based on temporal logic. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences,* 130–147.

van derWerf, J. M. E., van Dongen, B. F., Hurkens, C. A., & Serebrenik, A. (2009). Process discovery using integer linear programming. *Fundamenta Informaticae*, *94*(3–4), 387–412.

How to Cite: Komatsu, K. & Horita, H. (2024). Generating LTL Formulas for Process Mining by Example of Trace. *Journal of Data Science and Intelligent Systems*. https://doi.org/10.47852/bonviewJDSIS42022166