Article

Specification Mining over Temporal Data †

Giacomo Bergami *, Samuel Appleby † and Graham Morgan ‡

School of Computing, Faculty of Science, Agriculture and Engineering, Newcastle University, Newcastle Upon Tyne NE4 5TG, UK; s.appleby2@newcastle.ac.uk (S.A.); graham.morgan@newcastle.ac.uk (G.M.)
* Correspondence: giacomo.bergami@newcastle.ac.uk
† This paper is an extended version of our paper published in IDEAS’23: Proceedings of the International Database Engineered Applications Symposium Conference.

Abstract: Current specification mining algorithms for temporal data rely on exhaustive search approaches, which become detrimental in real data settings where a plethora of distinct temporal behaviours are recorded over prolonged observations. This paper proposes a novel algorithm, Bolt2, based on a refined heuristic search of our previous algorithm, Bolt. Our experiments show that the proposed approach not only surpasses exhaustive search methods in terms of running time but also guarantees a minimal description that captures the overall temporal behaviour. This is achieved through a hypothesis lattice search that exploits support metrics. Our novel specification mining algorithm also outperforms the results achieved in our previous contribution.

Keywords: verified artificial intelligence; specification mining; knowledge bases; temporal logic

1. Introduction

Verified artificial intelligence from the perspective of formal methods (Figure 1) concerns the specification, design, and verification of systems represented in mathematical specifications [1]. These all consider a model of the system to be verified \( S \), a model of the environment \( E \), and the property to be verified \( \Phi \). When merely focusing on the system to be verified in context-free scenarios, we can also ignore \( E \) and assume that \( S \) will contain all the relevant information of interest. The application of such techniques requires the representation of both \( S \) and \( \Phi \) in a formalism for which efficient decision processes exist, either for verifying the compliance of a system to a specification (formal verification [2], Section 2.3.1) or for producing a system that abides by a given specification (formal synthesis [3,4]). The ability to determine the former also allows us to mine specifications \( \Phi \) from the system represented in our data \( S \) (specification mining [5], Section 2.3.2). Under these assumptions, formal verification determines whether a system \( S \) satisfies the property \( \Phi \) through a yes or a no answer (\( S \models \Phi \)). A negative output is accompanied by a counterexample that shows how \( \Phi \) is falsified, while a positive answer also includes a certificate of correctness.

![Figure 1. Pipeline for Verified Artificial Intelligence for Temporal Data.](https://www.mdpi.com/journal/computers)
The wide adoption of such techniques makes finding efficient algorithms for solving specification mining tasks more pressing. Relational databases, usually providing more efficient query plans than other NoSQL representations [6], provide suitable representations for storing and querying collections of temporal information, thus including numerical time series [7], temporal updates of entities, and relationships represented in the relational model [8], as well as recording multiple traces as logs. In particular, traces result from observing a completed real-world process, providing a temporally ordered sequence of activities being audited [2,9]. Recent research shows how temporal data can always be represented as traces, as *time series segmentations* discretise multi-varied time series into discrete, fully observable, linear, and distinct events, identifying the different states in which a system is transiting [10], as well as the variations (increases or decreases) of the values associated with time series [11]. Such a segmentation also allows to optimise pattern searches over temporal processes [10]. In the context of temporal data, long-time proposed declarative languages provide both human-readable representations for temporal behaviours as clauses/templates [12] as well as suitable query languages for expressing formal verification tasks [2,13]. Adopting such languages is extremely beneficial for specification mining tasks, as they fairly restrict the sets of all possible temporal behaviours to certain interesting patterns of interest, thus allowing a restriction of the space search.

Cybersecurity datasets provide ideal use case scenarios for temporal data represented as logs. Those are often represented as a sequence of observable events that are audited while collecting information concerning the attacker’s strategy [2]. As such, sequences are temporal; this makes it the perfect use-case scenario for sequential pattern mining and association rule mining, which then become the basis for attack prediction tasks [14]. This makes them the ideal testbed to verify whether current state-of-the-art specification mining algorithms for temporal data can deal with greater amounts of distinct behaviours and data to be analysed (Section 5.2 vs. Section 5.3). Lagraa et al. [15] show that it is also possible to represent entire malicious log-in sequences as Workflow Nets, a procedural representation of temporal data, mined via sequential process mining algorithms, such as Alpha+++ [16]: Workflow Nets are then used as predictors for malicious log-in attempts through replaying live sequences. Çatak et al. [17,18] also show that it is possible to audit and log malware system call invocations to represent malicious software as a sequence of subsequent instruction invocations. The resulting dataset can then be exploited to train a long short-term memory neural network for distinguishing between eight different types of malware: trojan, backdoor, downloader, worm, spyware, adware, dropper, and virus. The primary distinction between earlier methods and this recent one lies in the resultant specification’s interpretability. While the new approach elucidates the disparities between various malware types post-facto using explainers, such as SHAP [19], the aforementioned approaches inherently extract a white-box classifier. Tools such as Workflow Nets, sequential pattern mining, or association rules deliver explainable results, motivating the rationale behind a specific classification outcome.

Specification mining over logs stipulates the return of temporal behaviour fulfilled by most traces in a log; such a majority constraint is usually assessed through a minimum support threshold $\theta$. This allows the practitioner to impose early stopping conditions to avoid testing underrepresented temporal behaviour. Despite the adoption of such a constraint, state-of-the-art specification mining algorithms [20,21] still test all the traces in the log without using the minimum support threshold as an earlier stopping condition (Sections 3.3 and 6.3), thus introducing sensible overhead while testing unnecessary temporal behaviour. This is a desired feature when dealing with real data. Furthermore, the aforementioned approaches are hindered by the exhaustive generation and testing of declarative clauses instantiated over each frequently occurring event type [20,21], thus being heavily hindered by the curse of propositionalisation. As a result, existing approaches do not adequately scale up to the size of big and real-world datasets, as the running time becomes quite explosive in the long run (Section 5.3). As the adoption of a declarative language to express the temporal behaviour of interest considerably reduces the space
search for the temporal behaviour to be returned as a specification, we can efficiently visit our search space by organising the candidate’s temporal behaviour to be tested as a lattice (Sections 2.2 and 4.3), where the descent is pruned using the aforementioned \( \theta \) as an early stopping condition. As a consequence of the missed opportunity to exploit faster search approaches, current algorithms do not ensure the provision of specifications of compact sizes. This feature is essential due to the profusion of mined declarative clauses which are later utilized as features to describe temporal data sequences in learning tasks [21] (Section 2.1). Such an approach can easily lead to trained models that overfit the training dataset [22], while training data with such an explosive amount of features requires an exponential number of data samples with respect to the number of such features (https://youtu.be/QZ0DtNFdDko, accessed on 10 September 2023).

This paper proposes Bolt2, an extension of our previous specification mining algorithm Bolt [5], which improves the former in terms of specification size reduction, providing a comparable running time to our initial solution. Our controlled experiments show that our heuristic algorithm can find the most specific behaviour with the best support if compared to other algorithms returning all the possible clauses (Section 5.1), while adopting a general-to-specific lattice search to traverse the declarative clause space, combined with pruning and heuristic assumptions for inference, the algorithm significantly surpasses existing exhaustive search methods. This makes it ideal for mining data-sparse real-world temporal datasets (Sections 5.2 and 5.3). As any mined specification can then be decomposed into minimal features to be fed to a classifier, we postulate that an ideal specification mining algorithm for temporal data shall return a minimal specification that “adequately” describes the temporal data of interest.

2. Related Work

In this section, we use the notation from [23] for uniformly describing data mining and machine learning problems.

2.1. Machine Learning

Machine learning studies systems that enhance their behaviour over time through experience. This experience consists of a set of examples \( \mathcal{L}_e \) in which we look for regularities or patterns fitting a specific model. Such a set of examples might be labelled with a \( f: \mathcal{L}_e \rightarrow \mathcal{Y} \) function, associating each example to a target classification class \( y \in \mathcal{Y} \). We can, therefore, define a (labelled) dataset as a collection of pairs \( D = \{ (x_0, y_0), \ldots, (x_{m-1}, y_{m-1}) \} \), associating each data point \( x_i \) to a class \( f(x_i) = y_i \in \mathcal{Y} \). We can define the set of all the possible training outcomes (i.e., hypotheses) as \( \mathcal{L}_h := \mathcal{Y}^{\mathcal{L}_e} = \{ h \mid h: \mathcal{L}_e \rightarrow \mathcal{Y} \} \), among which, we are interested in a function \( \hat{f} \) that minimises the classification loss when compared to the expected output \( f \). The training problem can then be summarised as follows:

\[
ML(loss, D, \mathcal{L}_h) := \arg \min_{h' \in \mathcal{L}_h} \text{loss}(h', D)
\]

When using decision trees [24] as use case examples for explainable machine learning models, each path \( \pi := r \rightarrow p_1 \rightarrow \cdots \rightarrow p_m \) from the root \( r \) toward a leaf \( p_m \) node associated with a class \( y \in \mathcal{Y} \) describe a classification rule \( r \land \land_{p_j \in \pi} p_j = y \), where each predicate might be (or might not be) negated, depending on the direction followed while traversing the learned model. As each branch provides alternative specifications for a class, we can extract an explanation for each class \( y \in \mathcal{Y} \) as a finitary disjunction of all the rule tails associated with the same head \( y \).

Deviant model learning (DML) [21] generates specifications (a.k.a. models) that uniquely characterise classes of traces present in a log through a mixture of declarative and procedural (i.e., patterns) features that are mined from the data (Section 2.2). It assumes that the log associates each trace \( \sigma \) to a specific class \( y(\sigma) \in \mathcal{Y} \). As the authors represent the presence/absence of a feature in a trace \( \sigma \) as a number \( \bar{x}_\sigma[\ell] \), each trace can be represented as an \( n \)-dimensional embedding \( \bar{x}_\sigma \in \mathbb{R}^n \), where \( n \) is the overall number of features,
e.g., when a Declare clause \( c_\ell \) is chosen as the \( \ell \)-th feature, a value of \( \vec{x}_\sigma[\ell] = -1/0/n \) denotes that the trace \( \sigma \) does not satisfy, vacuously satisfies, or satisfactorily activates \( c_\ell \) \( n \) times. For the features providing declarative characterisations, DML extracts those via the ADM+S specification mining algorithm [21] (Section 2.3.2). Any improvement in the specification mining phase ensuring model minimality will also minimise both the risk of model overfitting and the curse of dimensionality, as discussed in the introduction of this paper. Therefore, we aim to improve the preliminary mining phase before effectively discussing how to cope with the learning phase, which will be part of our future contribution. As the experiments in this paper are biased toward learning descriptions of deviant specifications expressed as conjunctive declarative clauses, no evidence of the possibility of returning specifications in disjunctive normal form is provided.

2.2. Data Mining

Contrary to machine learning algorithms, data mining algorithms do not limit themselves to the provision of one single hypothesis as they rather return all hypotheses that satisfy a quality criterion \( Q \). Given \( h \in \mathcal{L}_h \) and two classes \( \mathcal{Y} = 2 = \{0, 1\} \), we can define a cover function \( \text{co}(h, D) = \{x \in D | h(x) = 1\} \) as the function identifying the set of all items that are covered by the hypothesis \( h \) from \( D \). Data mining algorithms are often interested in returning only the hypotheses exhibiting frequent patterns. We can then define the minimum frequency criterion over a minimum threshold \( \theta \) as follows:

\[
Q_\theta(h, D) := |\text{co}(h, D)| \geq \theta
\]

We capture all possible data mining algorithms with the following mathematical formulation:

\[
\text{DM}(Q, D, \mathcal{L}_h) := \{h \in \mathcal{L}_h | Q(h, D) \text{ holds}\}
\]

Under the previous assumptions, one natural way to structure the hypothesis search space is to employ a generality relation \( \preceq \). We state that a hypothesis \( h_2 \) is more general than \( h_1 \) if all the examples covered by \( h_1 \) are also covered by \( h_2 \), expressed as

\[
h_2 \preceq h_1 \iff \text{co}(h_1, D) \subseteq \text{co}(h_2, D)
\]

Despite the exhaustive generation of all possible hypotheses in \( \mathcal{L}_h \) satisfying \( Q \) being inefficient (Algorithm 1), we can enhance efficiency by pruning the majority of non-valid hypotheses. For binary classification problems, the set of all possible hypotheses is \( 2^{\mathcal{L}_e} \), so each possible predicted positive example in \( \varphi(\mathcal{L}_e) \) represents the outcome of one single hypothesis. Given that there is an isomorphism between the set of hypotheses and the set of all possible data subsets \( \varphi(\mathcal{L}_e) \) (Lemma A1 in Appendix A), we can directly search the hypotheses by navigating the lattice space by \( \preceq \) using a general-to-specific visit. After proving that all the generalisations of hypotheses will satisfy the same quality criterion when a minimum-frequency \( Q_\theta \) is used (Lemma A2), it follows, by duality, that none of the specialized hypotheses not meeting this criterion will satisfy it either.

Algorithm 1: Brute-force hypothesis generation [23].

1: function ENUMERATE\((Q, D, \mathcal{L}_h)\)
2: \[ H \leftarrow \emptyset \]
3: \[ \text{for all } h \in \mathcal{L}_h \text{ do if } Q(h, D) \text{ then } H \leftarrow H \cup \{h\} \]
4: return \( H \)

While simplistic mining algorithms [20,21,25] enumerate and test all possible hypotheses in \( \mathcal{L}_h \) (Algorithm 1), more efficient ones [5,26,27] represent the whole hypothesis space as a lattice induced by the \( \preceq \) relationship while traversing from the most general hypothesis (\( \top \), Algorithm 2 Line 2) toward the most specific ones, while still validating the quality conditions, which are then returned (Line 7). These last solutions require the
General-to-specific lattice search \[23\]  
Algorithm 2 

To ensure specification size reduction, our aim is to return the $S$-processing operator, as in \[20\]. Moreover, we intend to design a quality criterion. This previous contribution defined a specialization $\rho$ of transactions as a frequent metric $\varsigma$ greater or equal to $\theta$ transactions. Given a minimum support threshold $\theta$, an item $i$ is frequent if its support is greater or equal to $\theta$. We can express the representativeness of an itemset in the whole set as the number of traces where $i$ appears:  

$$ \varsigma'(i) = |\{ t \in T \mid i \subseteq t \}| $$

(1)  

With respect to temporal model mining from linear finite logs of finite traces $\mathcal{E}$, our previous contribution defined a specialization $\rho$ for declarative dataless clauses through logical entailment \[5\]. With respect to this, our current contribution relies on a more solid theoretical foundation related to the adequacy of our quality criterion of choice (Section 3.4) and some improved branch-and-bound heuristic conditions (Section 3.3); it is adopted to fasten the search while making more refined assumptions about the data (Section 6.3).  

To ensure specification size reduction, our aim is to return the $S$-borders of such clauses rather than return all clauses with different support types that do not use max as a post-processing operator, as in \[20\]. Moreover, we intend to design a quality criterion. This criterion will ensure that the most refined candidate neither compromises the support of any more general clause nor falls below a specified minimum support threshold.  

**Algorithm 2 General-to-specific lattice search \[23\]**  

1. function $\text{GENERAL\_TO\_SPECIFIC}_{\mathcal{T} \in \mathcal{L}_{\mathcal{D}}}(\mathcal{Q}, D, \mathcal{L}_h)$  
2. Queue $\leftarrow \{ T \}$  
3. $H \leftarrow \emptyset$  
4. while Queue $\neq \emptyset$ do  
5. $h \leftarrow \text{pop}(\text{Queue})$  
6. if $\mathcal{Q}(h, D)$ then  
7. $H \leftarrow H \cup \{ h \}$  
8. Queue $\leftarrow \text{Queue} \cup \rho(h)$  
9. end if  
10. end while  
11. return $H$

2.2.1. Frequent Pattern Mining  

Given a set of all possible transactions $T$, and a set of occurring items $I = \bigcup_{t \in T} t$, an itemset is an element of $i \in \varphi(I)$. Support $\varsigma(i)$ of any itemset $i$ is defined as the ratio between the number of transactions $T$ being supersets of $i$ and the total number of transactions. Given a minimum support threshold $\theta$, an item $i$ is frequent if its support is greater or equal to $\theta$. We can express the representativeness of an itemset in the whole set of transactions as a frequency metric $\varsigma'(i)$ defined as the number of traces where $i$ appears:  

$$ \varsigma'(i) = |\{ t \in T \mid i \subseteq t \}| $$

We denote $\varsigma$ as the normalisation of such support by providing a ratio between $\varsigma'(i)$ and the overall number of available transactions $|T|$.  

We focus our attention on two frequent itemset mining algorithms. The Apriori \[25\] is a brute-force mining algorithm that begins by calculating an $F_1$ table, which associates each item in $I$ with its support, denoted as $\varsigma'(i)$. For each $i$-th iteration step from 1 to $|I|$, the algorithm prunes each frequent itemset table $F_i$ containing all of the itemsets of length $i$ using a minimum-support threshold $\theta$. If $F_i$ is not empty, we might compute the itemsets of length $i + 1$ as $F_{i+1} := F_i \times F_i$. In the worst-case scenario, this algorithm...
runs in exponential time: \(O(2^{|I|})\). FP-Growth \([26]\) improves the time complexity of the former algorithm by merging the itemset generation phase with the itemset pruning via the minimum support threshold, considerably reducing the overall time complexity. This enhancement exploits an item support table and the representation of frequent patterns as a prefix tree. The latter guarantees a compact in-memory representation of the transactions, and offers an efficient general-to-specific approach when generating frequent candidate itemsets.

2.2.2. Association Rule Mining

An association pattern \([29]\) \(X \rightarrow Y\) states that “if \(X\) occurs in the current transaction \(t \in T\), then it is likely that \(Y\) will occur, too”. We refer to \(X\) as the antecedent or tail and to \(Y\) as the consequence or head. Each frequent itemset \(Z\) can be considered as a rule \(\varnothing \rightarrow Z\), stating that \(Z\) is a possible configuration within our dataset. Association rule mining can be generalised through general-to-specific algorithms \([23]\), refining each possible frequent itemset \(i\) expressed as \(\varnothing \rightarrow i\) to the set of all possible rules \(X \rightarrow Y\), where \(X \cup Y = i\) and \(X \cap Y = \varnothing\). In particular, if we perform this for each itemset, we obtain a correct confidence-based pruning algorithm \([29]\) by visiting the lattice of all possible rules descending from \(i\), where \(\varnothing \rightarrow i\) (\(i \rightarrow \varnothing\)) is its top (bottom). Support and confidence metrics quantify the strengths of such rules from different perspectives. Support \(s\) determines how often a rule is “applicable” to the transactions in \(T\), while confidence \(c\) determines how frequently items in \(Y\) appear in “transactions containing \(X\)”. These can be defined as follows:

\[
s(X \rightarrow Y) = \frac{c'(X \cup Y)}{|T|} \quad c(X \rightarrow Y) = \frac{c'(X \cup Y)}{c'(X)}
\]

(2)

2.3. Verified Artificial Intelligence for Temporal Data

The present paper concerns specification mining for temporal data, where properties to be verified \(\Phi\) are expressed in LINEAR TIME LOGIC FOR FINITE TRACES (LTL\(_f\)) or the extensions. The LTL\(_f\) formalism \([3]\) extends modal logic to include linear, discrete, and future-oriented time, where each event represents a single relevant instant of time. LTL\(_f\) quantifies only on events reported in the trace, so each event of interest is fully observable, as the events represented in each trace are totally ordered. The syntax of a well-formed LTL\(_f\) formula \(\phi\) is defined as follows:

\[
\phi ::= a | \neg \phi | \phi \lor \phi' | \phi \land \phi' | \square \phi | \diamond \phi \mid \phi \lor \phi' \quad (3)
\]

where \(a \in I\) represents an atomic proposition, \(\square\phi\) determines that the condition \(\phi\) must hold from the subsequent temporal step, \(\diamond\phi\) requires that such a condition must always hold from the current instant of time, \(\diamond\phi\) specifies that the current condition must happen at any time in the future, and \(\phi \lor \phi'\) shows that \(\phi\) must hold until the first occurrence of \(\phi'\) does. The remainder are interpreted as classical logical connectives, where the implication connective \(\phi \Rightarrow \phi'\) can be interpreted as \(\neg \phi \lor (\phi \land \phi')\) \([30]\). The semantics are usually defined in terms of the first-order logic \([3]\) for a given trace \(\sigma^j\) at a current time \(j\) (e.g., for event \(\sigma^i_j\)).

Under the LTL\(_f\) framework, any given event in a sequence of interest is immediately followed by a single possible future event. This enforces a specific representation of systems, denoted as \(\mathcal{E}\), as a collection of sequences of distinguishable events \(\sigma^j \in \mathcal{E}\) (i.e., a trace) \([2]\). Each trace \(\sigma^j\) is a temporally ordered and finite sequence of distinct events \(\sigma^j = \sigma^j_1 \ldots \sigma^j_n\). Each event \(\sigma^j_i = (k_i, \delta^j_i)\) pairs an activity label \(k_i \in I\), describing the specific actions performing/occurring in a given event, a payload \(\delta^j_i\), and a finite function mapping each key to a single value. In this context, the formal verification determines which traces in a log satisfy an LTL\(_f\) formula \([2]\) while formal synthesis concerns the generation of traces abiding by an LTL\(_f\) formula \([4]\).
A transaction \( t \in T \) differs from the trace representation \( \sigma \); the former presents items in the form of sets, and the latter captures a sequence of temporally ordered events. Events might still be represented as items through their associated activity labels; therefore, \( I \) denotes the set of all activity labels. The conversion of a trace to a transaction comes at the cost of losing temporal information.

2.3.1. Formal temporal verification

In the context of LTL\(_t\) and temporal data, a counterexample of temporal verification is usually expressed as a non-empty edit sequence for a specific trace invalidating \( \Phi \), thus showing which events have to be removed or added to make a trace compliant to \( \Phi \) [31]. On the other hand, the certificate of correctness associated with traces often reflects the association between the atomic propositions in the formula and the events satisfying such conditions [2,30], and are often referred to as activation \( B \), target \( B \), and correlation \( \Theta \) conditions (see the following paragraph). As the present paper focuses on positive specification mining, we are only going to consider the latter aspect.

‘Declare’ [12] expresses properties \( \Phi \) as conjunctions of clauses providing a specification (or model) \( \Phi \), where each clause is an instantiation of a specific temporal template. A Declare template differs from customary association rules as the latter enforces no temporal occurrence pattern between the conditions at the head and the tail of the rule. Declare can be assessed over a finite multiset of finite traces.

Templates express abstract parameterised temporal properties (first column of Table 1), where semantics can be expressed in terms of the LTL\(_t\) (third column). The provided definitions align with the prevailing literature’s interpretations, subject to \( a \)-renaming (Section 6.4). We assert that a trace \( \sigma^t \) satisfies a Declare clause \( c \) if it meets the associated semantic representation in LTL\(_t\); we denote this as \( \sigma^t \models [c] \). We also state that a trace \( \sigma \) conforms to a (conjunctive) Declare specification (or model) \( \Phi \) if the trace satisfies each of the clauses \( \Phi = \{ c \}_{l \in \sigma, n \in N} \) composing the aforementioned specification, i.e., \( \sigma^t \models \Phi \iff \forall c \in \Phi [c] \). This verification process is also referred to as conformance checking in the business process management literature. We also state that an LTL formula \( \phi \) implies \( \phi' \) if all the traces satisfying the former also satisfy the latter. We denote this as \( \phi \Rightarrow \phi' \), where we state that \( \phi' \) is more general than \( \phi \) by interpreting the coverage set as the set of all traces in the log satisfying a given declarative clause (Lemma A3).

The implication notion can also be extended to declarative clauses in terms of their associated LTL\(_t\) semantics; Figure 2 represents the entailment lattice of such declarative clauses, expressed as a Hasse diagram [32], and depicting the relationship among such clauses under the assumption of non-zero confidence (Section 6.1).

![Figure 2](image-url)

**Figure 2.** The lattice represents all clauses that entail each other with non-zero confidence, visualized as a Hasse diagram.

If a Declare specification contains one clause that predicates over event payloads as an activation/target/correlation condition, we refer to this as *data-aware* specification; otherwise, we refer to this as *dataless*. We state that the log is *dataless* if each event has an empty payload, otherwise, it is *data-aware*. For data-aware specifications, we can identify correlation conditions denoting disjunction conditions across target/activation conditions,
as well as correlations between them. In this paper, we only consider dataless specifications and logs; thus, we omit the true data predicates associated with the activation/target conditions and only provide the activity labels. Therefore, throughout the rest of the paper, we denote each event $c^j$ by only its associated activity label, $A$ when all activity labels are considered as one character, we might represent each trace as a string of activity labels.

**Example 1.** Our current logs of interest are dataless; thus, having events that contain empty payload functions $\emptyset$, we use $c^j = ABC$ as the shorthand for $c^j = \langle A, \emptyset \rangle \langle B, \emptyset \rangle \langle C, \emptyset \rangle$.

Table 1. A set of dataless Declare templates considered in the paper. The clauses are marked according to their implementations in the respective Bolt/Bolt2 algorithmic implementations. Surround is our novel clause that, to the best of our knowledge, was first introduced in this paper. To simplify the characterisation of such clauses, we always assume that the first argument of the clause provides its activation $A$, while the second argument always provides a target $B$ or an alternative activation $A'$.

<table>
<thead>
<tr>
<th>Exemplifying Clause ($c_j$)</th>
<th>Natural Language Specification for Traces</th>
<th>LTL$_c$ Semantics ([$c_j$])</th>
<th>Bolt</th>
<th>Bolt2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Init</strong>($A$)</td>
<td>The trace should start with an activation</td>
<td>$A$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>End</strong>($A$)</td>
<td>The trace should end with an activation</td>
<td>Last($A$)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Exists</strong>($A$, $n$)</td>
<td>Activations should occur at least $n$ times</td>
<td>$\left{ \diamond (A \land \diamond (\exists (A, n + 1))) \right}$ $n &gt; 1$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Absence</strong>($A$, $n + 1$)</td>
<td>Activations should occur at most $n$ times</td>
<td>$\neg [\exists (A, n + 1)]$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Choice</strong>($A$, $A'$)</td>
<td>At least one of the two activation conditions must appear.</td>
<td>$\diamond A \lor \diamond A'$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>ExclChoice</strong>($A$, $A'$)</td>
<td>Only one activation condition must happen.</td>
<td>$[\text{Choice}(A, A')] \land [\text{NotCoExistence}(A, A')]$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>RespExistence</strong>($A$, $B$)</td>
<td>The activation requires the existence of the target.</td>
<td>$\diamond A \Rightarrow \diamond B$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>CoExistence</strong>($A$, $B$)</td>
<td>RespExistence, and vice versa.</td>
<td>$[\text{RespExistence}(A, B)] \land [\text{RespExistence}(B, A)]$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Precedence</strong>($A$, $B$)</td>
<td>Events preceding the activation should not satisfy the target</td>
<td>$\neg B \lor A$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>ChainPrecedence</strong>($A$, $B$)</td>
<td>The activation is immediately preceded by the target.</td>
<td>$\Box(\Box A \Rightarrow B)$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Response</strong>($A$, $B$)</td>
<td>The activation is either followed by or simultaneous to the target.</td>
<td>$\Box(A \Rightarrow \diamond B)$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>ChainResponse</strong>($A$, $B$)</td>
<td>The activation is immediately followed by the target.</td>
<td>$\Box(A \Rightarrow \Box B)$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Succession</strong>($A$, $B$)</td>
<td>The target should only follow the activation.</td>
<td>$[\text{Precedence}(A, B)] \land [\text{Response}(A, B)]$</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td><strong>ChainSuccession</strong>($A$, $B$)</td>
<td>Activation immediately follows the target, and the target immediately precedes the activation.</td>
<td>$[\text{ChainPrecedence}(B, A)] \land [\text{ChainResponse}(A, B)]$</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Surround</strong>($A$, $B$)</td>
<td>The activation is immediately followed by, and immediately preceded by, the target.</td>
<td>$[\text{ChainPrecedence}(A, B)] \land [\text{ChainResponse}(A, B)]$</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

Declarative clauses, therefore, involve the instantiation of declarative templates with activation and target conditions; by restricting our consideration to dataless specifications, such conditions restrict the possible activity label events by the one expressed by the activation (and target) conditions $A$ (and $B$). For example, $\text{Precedence}(A, B)$ has $A$ as an activation condition, as an occurrence of the first $A$ requires checking whether no preceding $B$ shall occur. We denote $\text{ActLeaves}(c)$ as the set of traces containing at least one event satisfying an activation condition in $c$ [2]. The clause activation might require the investigation of a possible target condition, where temporal occurrence is regulated by the template semantics.
To demonstrate this, consider the following example: while $\text{Response}(A, B)$ requires that the target conditions always follow the activations, $\text{RespExistence}(A, B)$ only requires its existence across the trace, and holds no temporal constraints as the former. Some of the clauses listed in Table 1 entail others, due to the stricter variation in the “temporal strictness” or the composition of multiple clauses. The previous example focuses on stricter temporal variation, while $\text{CoExistence}(A, B)$ entails both $\text{RespExistence}(A, B)$ and $\text{RespExistence}(B, A)$.

We state that a trace $\sigma^i$ vacuously satisfies a Declare clause $c$ if the trace satisfies such a clause without providing an event that satisfies the clause’s activation condition (i.e., $\sigma^i \not\models [c] \land \sigma^i \notin \text{ActLeaves}(c)$). We can similarly frame the concept of vacuous violation. It occurs when a trace fails to satisfy a declarative clause, represented as $(\sigma^i \not\models [c] \land \sigma^i \notin \text{ActLeaves}(c))$.

Declare walks in the footsteps of association rule mining by providing definitions of confidence and support metrics associated with declarative clauses. Still, the current literature provides disparate definitions of those metrics, depending on the specific context of interest \[2,20\], exploiting different clause “applicability” notions. Our previous work \[2\] considers “applicability” in terms of mere clause satisfiability, thus considering traces either vacuously or non-vacuously, satisfying a given declarative clause, while straightforwardly considering the traces containing the activation conditions as the denominator for the confidence score:

$$\text{SUPP}_\mathcal{E}(c) = \frac{|\{\sigma^i \in \mathcal{S} | \sigma^i \models [c]\}|}{|\mathcal{S}|} \quad \text{CONF}_\mathcal{E}(c) = \frac{|\{\sigma^i \in \mathcal{S} | \sigma^i \models [c]\}|}{|\text{ActLeaves}(c)|}$$

As the notion of $\text{SUPP}$ is directly related to the satisfiability of a given declarative clause, despite its inclusion of the vacuously satisfied traces, this makes it the ideal candidate for the basis of an anti-monotonic quality criterion (Lemma A4) for implementing the lattice search within a general-to-specific heuristic algorithm for the hypothesis search. Still, this definition of confidence is not normalised between zero and one, as the number of traces potentially satisfying the clause are much larger than the ones satisfying and providing an activation condition to the clause.

By restricting the notion of “applicability” to the sole traces containing activation conditions, we can have the notion of restrictive support and confidence \[20\], ensuring that the clause will fully represent the data if the evidence for both activation and (possibly) target conditions are given:

$$\text{RSUPP}_\mathcal{E}(c) = \frac{|\{\sigma^i \in \mathcal{S} \cap \text{ActLeaves}(c) | \sigma^i \models [c]\}|}{|\mathcal{S}|} \quad \text{RCONF}_\mathcal{E}(c) = \frac{|\{\sigma^i \in \mathcal{S} \cap \text{ActLeaves}(c) | \sigma^i \models [c]\}|}{|\text{ActLeaves}(c)|}$$

Still, these two definitions each come with an intrinsic limitation. First, we can easily observe that $\text{RSUPP}$ is not anti-monotonic in the same way as $\text{SUPP}$ and, therefore, cannot be chosen as a quality criterion candidate for traversing the declarative hypothesis space (Section 6.2). Second, the authors showcasing $\text{RCONF}_\mathcal{E}$ assumed that, for $\text{Precedence}(A, B)$, B should have been an activation condition, while this should have never really been the case, as the clause only postulates that B shall not appear before the first A, but nothing is stated on whether Bs shall occur after an occurrence of A. For this particular clause, there is a requirement for a redefined declarative confidence that ranges between 0 and 1. This new definition aims to provide a more restrictive understanding of confidence, especially in the presence of arbitrary activations or vacuous conditions within the trace. These force us to propose a novel specification of this restrictive confidence, which is a topic we will delve into in Section 3.1.

$\text{KnoBAB}$ \[2\] is a columnar-based relational database, built with the prime purpose of optimizing formal verification for temporal data represented as event logs. This was achieved through the provision of relational semantics to the $\text{LTL}_\mathcal{E}$ operators, namely $\text{x}_\mathcal{E}\text{LTL}_\mathcal{E}$,
while providing a one-to-one mapping between the provided temporal operators and the LTL$_t$ logical operators. The encoding of specific temporal operators for the relational model was mainly due to the inadequacy of the relational query plans and physical operators to efficiently compute such tasks over customary relational models [30]. The definition of xLTL$_t$ operators facilitates the expression of formal verification tasks as relational database queries. These queries not only return the trace that satisfies the specification but also provide details on the events that activate, target, and correlate with one another. xLTL$_t$ also allows the identification of query plan optimisation strategies. First, we minimise access to the data tables by fusing entire sub-expressions, providing the same temporal behaviour, including the query plan leaf nodes that directly access the temporal information stored as relational tables. Second, we can further pre-process this query plan to achieve trivial intra-query parallelism. As the aforementioned query plan transformation converts an abstract syntax tree into a direct acyclic graph, we can now run a topological sort algorithm to determine the operators’ order of execution after representing them as nodes within the query plan. This allows the additional transformation of the aforementioned direct acyclic graph into a dependency graph, represented as a layered graph. As each node belonging to a given layer shares no computational dependencies with the other, we can also define a scheduling order for the xLTL$_t$ operators, where the execution can be trivially parallelised, achieving straightforward intra-query parallelism. Third, we guarantee efficient data access through the provision of indexed relational tables containing temporal, payload, and event-counting information. The results provided in our previous research [2] show that the system achieves significant performance gains against state-of-the-art dataless and data-aware temporal formal verification systems, either ad hoc solutions or competing relational representations and queries expressing formal verification tasks directly in SQL.

We will now describe in greater detail the relational table being exploited to store the log traces’ temporal behaviour, as both Bolt and the current algorithm leverage such architecture. Temporal data are loaded and processed into three types of tables depicted in Figure 3: CountingTable$_{\mathbb{E}}$, ActivityTable$_{\mathbb{E}}$, and AttributeTable$_{\mathbb{E}}$(s).

![Figure 3: KnoBAB representation of a log $S$ describing a healthcare scenario [2]. Colours indicate the respective traces: yellow/orange/pink labels refer to $o_1/o_2/o_3$ respectively. Pointers to the n-th record of the ActivityTable are denoted as #n, while NULL pointers refer to no record.](image)

The CountingTable$_{\mathbb{E}}$ compactly stores the number of occurrences $f_i(A)$ of a (count cell) for each activity label $A$ of a (activityId cell) occurring in a trace $o^i$ of a (trace cell) as a 64-bit unsigned integer, which is also used to sort the tables in ascending order. We also show the absence of events associated with a given activity label by setting the count field for a specific trace to zero. The space complexity associated with this table is $O(|\Sigma| \cdot |\mathbb{E}|)$, while the occurrence count of a given activity for a specific trace can be retrieved in the $O(1)$ time.

<table>
<thead>
<tr>
<th>Counting Table</th>
<th>Activity Table</th>
<th>Attribute Table [query]</th>
<th>Attribute Table [feature]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivityId</td>
<td>Trace</td>
<td>Count</td>
<td>ActivityId</td>
</tr>
<tr>
<td>_trace_paylad</td>
<td>1</td>
<td>1</td>
<td>#1_trace_paylad</td>
</tr>
<tr>
<td>_trace_paylad</td>
<td>2</td>
<td>1</td>
<td>#2_TRACE_paylad</td>
</tr>
<tr>
<td>_trace_paylad</td>
<td>3</td>
<td>1</td>
<td>#3_TRACE_paylad</td>
</tr>
<tr>
<td>Referral</td>
<td>1</td>
<td>1</td>
<td>#4_TRACE_paylad</td>
</tr>
<tr>
<td>Referral</td>
<td>2</td>
<td>1</td>
<td>#5_trace_paylad</td>
</tr>
<tr>
<td>Mastectomy</td>
<td>1</td>
<td>1</td>
<td>#6_TRACE_paylad</td>
</tr>
<tr>
<td>Mastectomy</td>
<td>2</td>
<td>0</td>
<td>#7_TRACE_paylad</td>
</tr>
<tr>
<td>Mastectomy</td>
<td>3</td>
<td>0</td>
<td>#8_TRACE_paylad</td>
</tr>
<tr>
<td>FollowUp</td>
<td>1</td>
<td>1</td>
<td>#9_TRACE_paylad</td>
</tr>
<tr>
<td>FollowUp</td>
<td>2</td>
<td>0</td>
<td>#10_TRACE_paylad</td>
</tr>
<tr>
<td>Mastectomy</td>
<td>1</td>
<td>0</td>
<td>#11_TRACE_paylad</td>
</tr>
<tr>
<td>Mastectomy</td>
<td>2</td>
<td>0</td>
<td>#12_TRACE_paylad</td>
</tr>
<tr>
<td>Mastectomy</td>
<td>3</td>
<td>0</td>
<td>#13_TRACE_paylad</td>
</tr>
</tbody>
</table>
As the table’s records are sorted by the activity label, trace ID, and event ID by storing \( \text{ActivityTable} \) This is very different from customary formal methods, where a formula defines the correct property, and another for those that deviate from it [21]. Still, doing so when data are \( \delta \) payload information \( \sigma \) preceding preserves this. Each row \( \langle \) adequate to identify two distinct classes: one for traces conforming to the expected temporal between behaviour that adheres to a formal specification and that which does not, it is expected behaviours over all possible inputs. When we are only interested in distinguishing classification label \( y \) from which specifications are extracted. Data points \( x \) for mining such clauses in Section 4.3.3.

**Example 3.** ChainPrecedence(\text{FollowUp, Mastectomy}) in our running example requires accessing all the activation conditions through the primary index of such a table, thus obtaining \( \text{ActivityTable}_{\delta} \), \( \text{primary\_index}(\text{FollowUp}) = (\#8, \#9) \). Now, we can iterate the table’s records, starting from no. 8 until no. 9 included. By exploiting the \( q \) \text{Prev} pointers, we can determine that no. 8 has a preceding \text{mastectomy} event while no. 9, also belonging to a different trace, has a preceding \text{lumpectomy} event, thus violating the aforementioned clause. This leads to the main optimisation of mining ChainPrecedence and ChainSuccession clauses This assumption will be used implicitly in the code for mining such clauses in Section 4.3.3.

The secondary index of the \( \text{ActivityTable}_{\delta} \) maps each trace the ID to the first and last events. This is especially beneficial for declarative mining as we can directly return the first and last events of each trace, and are able to quickly mine \text{Init/End} clauses.

**Example 4.** With reference to our running example, determining whether \( \text{End(\text{FollowUp})} \) requires accessing the last event for all traces, as we obtain \( \text{ActivityTable}_{\delta} \), \( \text{secondary\_index}(\sigma^1) = (\#1, \#8) \), \( \text{ActivityTable}_{\delta} \), \( \text{secondary\_index}(\sigma^2) = (\#2, \#5) \), and \( \text{ActivityTable}_{\delta} \), \( \text{secondary\_index}(\sigma^3) = (\#3, \#9) \), only the first and the last events are associated with a \text{FollowUp} event, while the second is a \text{Referral}. This leads to the main optimisation associated with Algorithm 5 for mining \text{Init/Ends} clauses.

\( \text{AttributeTable}_{\delta} \)’s guarantee faithfully reconstructing the entire log information by associating the payload information for each event in the activity table. In particular, we generate one table per key in the data payload. As our current algorithm does not rely on payload information \( \delta \) stored in this table, we refer to [2] for further information.

2.3.2. Temporal Specification Mining

In traditional machine learning (Section 2.1), data provide the common specifications from which specifications are extracted. Data points \( x \) are usually associated with a classification label \( y \), directly associating the data to the expected class of specifications. This is very different from customary formal methods, where a formula defines the correct expected behaviours over all possible inputs. When we are only interested in distinguishing between behaviour that adheres to a formal specification and that which does not, it is adequate to identify two distinct classes: one for traces conforming to the expected temporal property, and another for those that deviate from it [21]. Still, doing so when data are
biased, limited, or labelled from non-experts might be challenging [1]. Specification mining attempts to bridge the gap between specifications and data by inferring the former from the latter.

In business process mining literature, specification mining is often referred to as declarative process (or model) mining. As in specification mining, we are interested in extracting specifications, where the language for the specification is expressed as a Declare clause. Apriori declarative mining (ADM) [20] represents an advancement in state-of-the-art declarative temporal specification mining algorithms [33] by limiting the instantiation of all possible declarative clauses to the itemsets being returned by an Apriori algorithm; such clauses are then further pruned through declarative minimum support thresholds, while prior solutions [33] instantiated every Declare clause with all conceivable templates, leading to a quadratic combination set. This constraint is also evident in the available implementation of Apriori declarative mining + Fisher score (ADM+S) in Python [21], where the actual implementation instantiates all the templates for all the possible activity labels being available on the current data. In fact, both [33] and [21] become easily intractable in real-world logs describing thousands of activities/events. ADM also further creams off infrequent declarative clauses through confidence and support. Notwithstanding the aforementioned improvements for candidate clause pruning, such approaches exploit no efficient heuristics for mining clauses. By only exploiting frequent itemset mining algorithms with support filtering for generating and testing all possible declarative clauses on log traces, the authors achieve suboptimal algorithms that generate and test all the candidate clauses (propositionalisation [25]), which are further creamed off through an additional filtering step.

Figure 4 provides a clear picture concerning the impossibility of generating, testing, and returning all possible clauses for realistic datasets containing many distinct events through exhaustive search and propositionalisation within a reasonable amount of time. As an upper bound to the total number of possible clauses to be returned, we can consider instantiating all possible pairs of activity labels for the 11 binary clauses and instantiating the 4 unary clauses with \( n = 0 \) (Absence) or \( n = 1 \) (Exists), thus obtaining the following:

\[
UB(I) = 11 \cdot (|I|^2 - |I|) + 4|I| \tag{7}
\]

For the lower bound, we can limit the number of clauses for the 3 symmetric clauses of Choice, ExclChoice, and CoExistence to \(||I| - 1|/2\) combinations, thus obtaining \( LB(I) = \frac{1}{2}(19|I|^2 - 11|I|) \). We can easily see that, for hundreds of activity labels, as in realistic cybersecurity datasets, naïve algorithms are required to test an order of one million clauses, which cannot be analysed in a feasible amount of time. This clearly postulates the need for an efficient search algorithm limiting the search to the clauses effectively being representative of the data of interest.

Our previous implementation of dataless declarative process mining, i.e., Bolt [5], was designed to efficiently mine the twelve clauses listed in Table 1; however, it excluded the two newly introduced clauses, Succession and ChainSuccession, and our novel proposal introduced in the present paper, Surround (Section 3.2). Bolt outperforms all state-of-the-art dataless declarative mining competitors by leveraging KnoBAB’s aforementioned indexed tables while avoiding costly formal verification computations by only detecting vacuity and violation conditions without the need for combining multiple \( x\text{LTL}_f \) operators, as they can generate considerable overhead due to the intermediate results being generated by the implementation of the relational operators. Moreover, declarative clauses are generated using a general-to-specific search, starting from frequent clauses being directly inferable from association rule mining, such as RespExistence and CoExistence, temporally refining them by starting the visit of the lattice in Figure 5. To avoid generating uninteresting atemporal behaviour when temporal behaviour might be identified, we generate Choice and ExclChoice clauses by combining unary infrequent patterns. Our current solution further improves this by discarding the return of such clauses if any more informative temporal behaviour is returned (Section 4.3.5).
Figure 4. Lower bound and upper bounds to the maximum number of clauses returnable by a specification mining algorithm, given several maximum activity labels.

Figure 5. Lattice implemented for the clause traversal in Bolt [5], where the parts in blue show the results from frequent itemset and association rule mining.

In our previous contribution, we compared our proposed algorithm to [5] TopN Declare [30] using KnoBAB’s formal verification pipeline, Apriori declarative mining (ADM) after re-implementation on top of KnoBAB [20], and Apriori declarative mining + Fisher score (ADM+S) implemented in Python [21]. Through experiments, we demonstrated that we were both scalable with regard to the trace length and log size, while consistently outperforming the running time for all former competitors. TopN Declare was poor at mining all expected clauses from a log, as just considering all possible pairs among frequently occurring activity labels does not guarantee returning all the relevant clauses, and the others suffered greatly by producing explosive and redundant specifications, achieving hundreds of clauses, even with a strict support threshold $\theta$, as these solutions are affected by the curse of propositionalisation required to instantiate and test all possible clauses that might be generated from some mined binary frequent patterns. Bolt avoids the computational
overhead from prior approaches due to the reduction of the specification mining task to a conformance checking one and its associated data allocation footprint by directly scanning relational tables without the need for generating intermediate results, as enforced by the adoption of query operators. Different from previous mining approaches that required instantiating all possible declarative clauses over all possible frequent itemsets through extensive clause propositionalisation, Bolt exploits a preliminary branch-and-bound solution for pruning irrelevant or “redundant” clauses, avoiding testing to abide by some irrelevant clauses that are not represented within the data (Example 13). The frequent support and confidence of itemsets serve as heuristics for determining whether specific clauses are frequent or appear in the specification, thus narrowing down the testing of temporal patterns to only the relevant instances.

Despite Bolt outperforming state-of-the-art temporal specification mining algorithms while mining less superfluous and more descriptive specifications, this approach can still be significantly improved. First, despite Bolt being able to prune many redundant clauses that do not aptly describe the log compared to other solutions, the outputted specification is still far from minimised, as there still exist redundant clauses that imply each other. Improving this requires avoiding returning Choice clauses for temporal declarative clauses that have already been considered if appearing with greater or equal support. Second, this preliminary solution still returns multiple symmetric clauses, which are deemed as equivalent, e.g., CoExistence(A, B) is completely equivalent to CoExistence(B, A), as they refer to the same set of log traces. Third, we still return multiple clauses entailing each other, not fully exploiting the lattice descent for ensuring specification size minimality via the computation of an S-border. Any desired improvement of Bolt tackling all these limitations with minimal performance costs will provide specifications returning more specific clauses while being less verbose than Bolt, while outperforming all other competitors by at least one order of magnitude. Our methodology section (Section 4) shows how our novel solution addresses all such limitations.

3. Theoretical Contributions

This section provides preliminary theoretical definitions that are needed before introducing our improved algorithm discussed in the methodology section (Section 4). After enhancing the restrictive definition of confidence (Section 3.1), we describe our newly proposed clause, Surround (Section 3.2), where we envision pruning heuristics jointly with ChainSuccession to avoid costly violation checks (Section 3.3). Finally, we discuss a notion of generality and quality criteria for shaping the S-border to ensure specification minimality (Section 3.4). Table 2 summarises the notation used throughout the paper.

Table 2. Table of notation for symbols $\chi \in \mathcal{T}$ defined as $(\chi := \mathcal{E})$ or characterised by $(\mathcal{E}(\chi)) \mathcal{E}$.

<table>
<thead>
<tr>
<th>Symbol ($\chi$)</th>
<th>Definition ($\mathcal{E}$)</th>
<th>Type ($\mathcal{T}$)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>Set Theory for Data Mining</td>
<td>Set</td>
<td>An empty set contains no items.</td>
</tr>
<tr>
<td>$(\preceq, S)$</td>
<td>A partially ordered set (poset) is a relational structure for which $\preceq$ is a partial ordering over $S$ [32]. $\preceq$ over $S$ might be represented as a lattice, referred to as the Hasse diagram.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\complement$</td>
<td>$\mathcal{U}\setminus\mathcal{C}$</td>
<td>Set</td>
<td>Complement set: given a universe $\mathcal{U}$, the complement returns all of the elements that do not belong to $\mathcal{C}$.</td>
</tr>
<tr>
<td>$</td>
<td>\mathcal{C}</td>
<td>$</td>
<td>$\sum_{c \in \mathcal{C}} 1$</td>
</tr>
<tr>
<td>$\wp(\mathcal{C})$</td>
<td>${T</td>
<td>T \subseteq \mathcal{C}}$</td>
<td>Set</td>
</tr>
</tbody>
</table>
Table 2. Cont.

<table>
<thead>
<tr>
<th>Symbol (χ)</th>
<th>Definition (E)</th>
<th>Type (T)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Y^X )</td>
<td>( { f : X \rightarrow Y } )</td>
<td>Set</td>
<td>Set exponentiation.</td>
</tr>
<tr>
<td>( I_B(x) )</td>
<td>( \begin{cases} 1 &amp; B(x) \ 0 &amp; \text{oth.} \end{cases} )</td>
<td>D ( \rightarrow {0,1} )</td>
<td>Indicator function.</td>
</tr>
<tr>
<td>( \bowtie^b )</td>
<td>Def. 2</td>
<td>Relationship</td>
<td>Bolt2 generality relationship.</td>
</tr>
<tr>
<td>( Q^b )</td>
<td>Def. 3</td>
<td>Predicate</td>
<td>Bolt2 quality criterion.</td>
</tr>
</tbody>
</table>

Frequent Patterns and Association Rules

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l )</td>
<td>Finite set of occurring atoms, also providing a finite set of activity labels.</td>
</tr>
<tr>
<td>( i )</td>
<td>A possible itemset ( i ), or a transaction ( l ).</td>
</tr>
<tr>
<td>( T )</td>
<td>The set of all possible transactions.</td>
</tr>
<tr>
<td>( \zeta(i) )</td>
<td>Itemset support for ( i ) with respect to some transactions ( T ).</td>
</tr>
<tr>
<td>( \zeta(i) )</td>
<td>Normalised itemset support for ( i ) w.r.t some transactions ( T ).</td>
</tr>
<tr>
<td>( T \rightarrow H )</td>
<td>Rule</td>
</tr>
<tr>
<td>( s(T \rightarrow H) )</td>
<td>Equation (2)</td>
</tr>
<tr>
<td>( c(T \rightarrow H) )</td>
<td>Equation (2)</td>
</tr>
</tbody>
</table>

**3.1. Enhanced Restrictive Confidence**

During our extension of the declarative metrics, we found that the definition of restrictive confidence in Section 2.3.1 did not encompass all the possible formulations of activations and targets in Declare clauses. This is mainly due to the semantics of Precedence\( (A,B) \), as well as the "correct" interpretation of what an activation for such a clause should be. Here, \( A \) is our activation, as in LTL\(_f\), we need to scan the first \( A \)s of each trace and check for \( \neg B \) in the past. Precedence is distinct from the other declarative clauses.
as the lack of activation does not guarantee vacuous satisfiability because it can be violated without any activation at all via \( \square (\neg B) \). By considering this, we obtain our upgraded metric by considering the cases where the trace was either activated or vacuously violated.

**Definition 1.** Enhanced restrictive confidence expresses the ratio between traces in \( \mathcal{S} \) both satisfying and activating a declarative clause \( c \) of the overall number of traces, either activating the declarative clause (without necessarily satisfying it) or not satisfying the clause at all.

\[
\text{RCONF}_{\mathcal{S}^+}(c) = \frac{|\{ \sigma_i \in \mathcal{S} \cap \text{ActLeaves}(c) \mid \sigma_i \models [c] \}|}{|\text{ActLeaves}(c) \cup \{ \sigma_i \in \mathcal{S} \mid \sigma_i \not\models [c] \}|} \quad (8)
\]

**Example 5.** Consider the log \( \mathcal{S} = \{ \sigma^1, \sigma^2, \sigma^3 \} \) with traces \( \sigma^1 = \text{ABB} \), \( \sigma^2 = \text{CAA} \), and \( \sigma^3 = \text{CCB} \). Let \( c \) be a clause \( \text{Precedence}(A, B) \). \( \sigma^1 \) is activated and satisfied with both activation and target conditions, and \( \sigma^2 \) is satisfied, containing an activation only. However, \( \sigma^3 \) provides a case that is not considered by the other clauses; we have no activation, \( A \), but the clause is still violated. We consider \( \sigma^3 \) as vacuously violating \( c \) as it violates \( \square (\neg B) \), while the others are non-vacuously satisfied. For Precedence, we have \( \text{RCONF}_{\mathcal{S}}(c) = 1.0 \). However, this does not consider the vacuous violation from trace \( \sigma^3 \).

By using the novel formulation above, we obtain the expected \( \text{RCONF}_{\mathcal{S}^+}(c) = \frac{2}{3} \).

### 3.2. Surround

We propose a novel Declare clause, Surround, providing a unique temporal behaviour. We might argue that Surround is not required due to the existence of ChainSuccession, as they both describe similar temporal behaviours. Still, the latter does not prescribe that an event might be both preceded and followed by a given one as the former. Surround describes the following unique temporal constraint: every activation is immediately preceded by and immediately followed by the target. In the context of data-aware specifications and logs, we can exploit this clause to express the idea that the occurrence of an activation event should not affect the post-conditions of the event that should then match the pre-conditions, thus adequately describing invariant behaviour.

**Example 6.** Consider the following log:

\[ \mathcal{S} = \{ \sigma^1, \sigma^2, \sigma^3 \} \quad \sigma^1 = \text{CBAB}, \sigma^2 = \text{BBB}, \sigma^3 = \text{CCBBA} \]

We have that \( \text{ChainSuccession}(A, B) \) has zero support as all the traces violate the constraint, while \( \text{Surround}(A, B) \) has a perfect support. In fact, the latter is defined as \( \text{ChainResponse}(A, B) \land \text{ChainPrecedence}(B, A) \), while the former is defined as \( \text{ChainResponse}(A, B) \land \text{ChainPrecedence}(B, A) \).

We can formally prove that \( \text{ChainSuccession}(A, B) \neq \text{Surround}(A, B) \) by showing that there exists at least one trace differentiating the behaviours of the two clauses (Lemma A5).

### 3.3. Pruning Heuristics

The following pruning heuristics aim to avoid computing costly trace scanning and testing for specific temporal patterns if we can efficiently test a numerical condition that encompasses the violation of a clause by a substantial number of traces within the log. Formulating such heuristics as conditions to be checked on the ActivityTable\(\mathcal{S} \) will not be deemed as adequately efficient as it would require extra scans that will still boil down to the punctual detection of dissatisfying traces, we aim to determine early stopping conditions on the CountingTable\(\mathcal{S} \) when applicable. We identify two heuristics, one for ChainSuccession and the other for Surround.

We now discuss the novel improved heuristics, replacing the ones from our previous paper. Any trace satisfying \( \text{ChainSuccession}(A, B) \) should have the same number of As as B. However, we also need to consider that B might also occur at the beginning of a trace, for which no \( \text{ChainPrecedence}(B, A) \) is activated. This provides a new bound for the number
of $A$s within a trace, which should be one less than the number of $B$s. This leads to the following lemma:

**Lemma 1.** A trace $\sigma^i$ violates not-vacuously ChainSuccession($A, B$) upon violation of the following constraint: $\#_{\sigma^i}B - 1_{v^i_{\sigma^i}=(B, B)} \leq \#_{\sigma^i}A \leq \#_{\sigma^i}B$.

Consider that $1_P$ is the indicator function returning 1 upon verification of the predicate $P$, and 0 otherwise. We can infer a similar constraint for Surround($B, A$) in the worst-case scenario, where activation never occurs at the start of a trace, events labelled as $B$ must occur both before and after event $A$. Conversely, in the best-case scenario, when $A$ occurs at the start of the trace, there is no need for a preceding $B$ event before such $A$:

**Lemma 2.** A trace $\sigma^i$ violates non-vacuously Surround($A, B$) upon violation of the following constraint: $\#_{\sigma^i}A \leq \#_{\sigma^i}B \leq 2 \cdot \#_{\sigma^i}A$.

Correctness proofs for both lemmas are presented in Appendix C.

### 3.4. Clause Generality and Quality Criterion for Mining

The generality relationship of choice for our mining algorithm formalises the notion implicitly provided in our previous paper [5]. This notion is required to guarantee the correctness of our mining algorithm and avoid returning evidence from the data due to vacuous satisfiability.

**Definition 2 (Bolt2 Generality).** Given a log $\mathcal{S}$, we state that a declarative clause $c'$ is more general than $c$ (and we denote this as $c' \preceq^b c$) if $c$ implies $c'$, while both are activated by at least one trace in $\mathcal{S}$:

$$c' \preceq^b c \Leftrightarrow c \Rightarrow c' \land RCONF_{\mathcal{S}}^b(c) > 0 \land RCONF_{\mathcal{S}}^b(c') > 0$$

Observe that this definition follows from the correlation between the coverage and clause implication (Lemma A3) while also guaranteeing the usage of the SUPP metric as an anti-monotonic metric for general-to-specific algorithms for lattice descent (Lemma A4 and Corollary A1). Due to this, our algorithm is also going to exploit this metric as an early stopping condition, thus avoiding further refining clauses that might provide infrequent support from the data. Furthermore, generating preliminary clauses from non-zero confidence association rules guarantees the generation of refined clauses with non-zero enhanced restrictive support, while also guaranteeing the traversal of the lattice from the most specific hypothesis to a more general one (Section 6.2). This definition enables a specific definition for a refinement function $\rho$ using $\preceq^b$ as a generality relation.

We then leverage the former definition of generality for specifying our quality criterion for our general-to-specific hypothesis (i.e., clause) search. Before saying so, we want to state that clause $c$ is “stronger” (in our previous paper, this notion was referred to as “consumption”) than $c'$ if (i) the coverage of the former is greater than the one of the latter, (ii) all the traces accepted by $c$ are also accepted by $c'$ (i.e., $c \Rightarrow c'$), and (iii) its support is lower-bounded by a minimum support threshold $\theta$. The first property favours clauses covering most of the traces from the log while still satisfying the minimal quality criterion. Given this requirement, we provide a quality criterion for a clause $c$ over a log $\mathcal{S}$, dictating that $c$ be stronger than any possible clause $c'$ for which $c$ is a refinement.

**Definition 3 (Bolt2 quality criterion).** A clause $c$ satisfies the quality criterion over a log $\mathcal{S}$ if this appears stronger than any clause for which it is a direct refinement.

$$Q^b(c, \mathcal{S}) \Leftrightarrow (\forall c'. c \in \rho(c') \Rightarrow \text{SUPP}_{\mathcal{S}}(c') \leq \text{SUPP}_{\mathcal{S}}(c)) \land \text{SUPP}_{\mathcal{S}}(c) \geq \theta \land RCONF_{\mathcal{S}}^b(c) > 0$$

Observe that this quality criterion can also be further simplified after observing that SUPP is anti-monotonic (Corollary A2).
Example 7. By leveraging the same log from Example 6, SUPP(Response(\(D, B\))) = \(\frac{2}{3}\), as \(\sigma^2\) contains no \(B\) events following \(D\), and the remaining clauses are vacuously satisfied, while the support of RespExistence(\(D, B\)) is 1.0 as \(\sigma^2\) contains a \(B\) event as well as a \(D\). In our proposed approach, we advocate returning the latter instead of the former, as it provides better data coverage, thus minimising the risk of returning clauses overfitting the temporal behaviour of traces.

4. Methodology

Our proposed specification mining algorithm Bolt2 (Algorithm 3) can be seen as the repeated application of the general-to-specific hypothesis search (Algorithm 2) for each frequent pair of activity labels \(\{A, B\}\) frequently occurring in the log that generates binary declarative clauses (Algorithm 6 Line 6), while unary patterns are treated separately (Algorithm 5). We also extend the mining algorithm with pruning and inference heuristics, discarding costly tests of clauses not being representative of the log. This log then describes the data \(D\) of interest for the algorithm. At the same time, the hypothesis space \(L_h\) for our declarative clause is derived from the instantiation of all possible templates from Table 1 with both \(\{A, B\}\) and \(\{B, A\}\). The generality relation \(\preceq^b\), essential for the refinement operator \(\rho\), as well as the quality criterion \(Q^b\), are defined in Section 3.4.

Algorithm 3 Bolt2

1: function BOLT2(\(\mathcal{G}, \theta\))
2: \(P \leftarrow \text{GENERATEFREQUENTITEMSETS}(\mathcal{G}, \theta)\) \(\triangleright \) Algorithm 4
3: ChoiceFilter, \(\Phi' \leftarrow \text{GENERATEUNARYCLAUSES}(P)\) \(\triangleright \) Algorithm 5
4: return \(\Phi' \cup \text{BINARYCLAUSEMINING}(P, \text{ChoiceFilter}, \theta)\) \(\triangleright \) Algorithm 6

We now describe the specific implementation of such an algorithm. This accepts a log \(\mathcal{G}\) loaded and indexed as a knowledge base in KnoBAB, as well as a minimum support threshold \(\theta\), and returns a list of declarative clauses to be considered as part of a conjunctive declarative specification \(\Phi\). As with our previous mining algorithm, Bolt [5], we avoid mining clauses that are completely vacuously satisfied through association rule confidence. This is further enhanced by our notion of SUPP while ensuring that at least one trace also activates the associated clause. This is extremely important for ensuring the correctness of our proposed lattice descent and ascent (Section 6.1).

We now motivate the adoption of this novel algorithm in light of the limitations of Bolt, as discussed in Section 2.3.2.

First, regarding Choice and ExclChoice, we generate such clauses by either pairing infrequent unary label with events having other activity labels or by returning a (Excl)Choice clause if no suitable temporal binary clause was not already returned. We also change the policy for returning Init and End clauses by returning them only if they satisfy the whole log.

Second, for symmetrical clauses \(c(A, A') \Leftrightarrow c(A', A)\), where both arguments are to be considered as activations, such as Choice, ExclChoice, and CoExistence, we also consistently prefer one of the two options by preferring the activity label appearing before the other one as the first argument of the clause.

Third, we further reduce the amount of returned clauses by discarding any Choice clauses if any other more refined clause is returned, as well as using the quality metric discussed in Section 3.4. To achieve this, we strictly abide by the entailment lattice described in Figure 6. In particular, considering the non-zero support constraint, the general-to-specific traversal of the lattice exploits the refinement relationship \(\rho\) using the associated generality relation \(\preceq^b\) from the previous section. Implementation-wise, this is achieved by always preferring the most refined clause at the same support conditions and avoiding returning the more general clauses if a clause with greater support is already returned.
The algorithm proceeds as follows: After generating the frequent itemsets with a maximum size 2 via FP-Growth (Section 4.1), we generate Existence/Absence clauses for certain unary patterns and prepare the generation of binary clauses by combining infrequent unary patterns or directly exploiting the mined binary patterns (Section 4.2). The last phase effectively generates such binary clauses through a preliminary heuristic phase via association rule mining, allowing further pruning of declarative clauses to be tested and considered (Section 4.3). Algorithm 3 shows the mutual dependencies of such tasks regarding input/output parameters.

### 4.1. Frequent Pattern Mining

In our first step described in Algorithm 4, we transform the temporal data stored in KnoBAB as transactions that are suitable for our implementation of the FP-Growth algorithm. This is rewritten from scratch due to errors in the previous library (https://github.com/integeruser/FP-growth, accessed on 10 August 2023) of choice, where bugs lead to inaccurate support computations, thus enforcing an estimate of the overall support through approximations. This phase now generates frequent itemsets suitable for the next association rule-mining phase.

**Algorithm 4 Running FP-Growth over KnoBAB**

```plaintext
1: function GENERATEFREQUENTITEMSETS(Σ, θ)
2:   ζ ← \[A → \{\{σ|}_{j} \geq \theta(i,j)\}\] _i∈I
3:   T ← \[i \in I \mid \#_σ(i) > 0\] _i∈Σ
4: return FPGROWTH(ζ,T,θ,ℓ = 2)
```

As applying the FP-Growth algorithm over a log requires discarding the temporal information associated with a trace, as transactions merely list the occurrence of items/activity labels, we exploit our CountingTable as a transactional representation of our log. Any transaction \(t_i\) associated with a temporal trace \(σ^t\) will contain an item \(j\) if \(σ^t\) has at least one occurrence of \(j\), i.e., \(j \in t_i \Leftrightarrow (σ^t_j) > 0\) (Line 3). Given this correspondence, we can then determine the item frequency table \(ζ'\) associated with our log as required by the FP-Growth algorithm by counting the traces containing at least one event with a given activity label \(A\) (Line 2). Such an algorithm can then be run with a minimum support threshold \(θ\), consistent with previous literature. As we are only interested in unary and binary clauses, we generate frequent itemsets that have a maximum length of \(ℓ = 2\) and support at least \(θ\). The algorithm then returns the frequently mined itemsets \(π_i \in P\) with support \(ζ'(π_i)\) (Line 4).

### 4.2. Unary Clause Mining and Binary Clauses Pre-processing

The phase described in Algorithm 5 combines the infrequent unary itemsets in \(P\) for later generating Choice and ExclChoice clauses, while directly yielding unary declarative...
patterns for certain itemsets. The clauses being generated in this phase share the same certain (100%) values for SUPP, RSUPP, and RCONF$^+$. 

**Algorithm 5 Generating unary clauses.**

1: function GENERATEUNARYCLAUSES$(P)$
2: ChoiceFilter ← $\emptyset$; $\Phi$ ← $\emptyset$
3: for all $\langle \pi_i = \{A\}, \varsigma(\pi_i) \rangle \in P$ do
4:   if $\varsigma(\pi_i) = 100\%$ then
5:     $n$ ← $\text{arg min}_s \#_s A > 0$
6:     $N$ ← $\text{arg max}_s \#_s A > 0$
7:     $\Phi$ ← $\Phi \cup \{\langle \text{Exists}(A, n), 100\% \rangle, \langle \text{Absence}(A, N + 1), 100\% \rangle\}$
8:   else
9:     ChoiceFilter ← ChoiceFilter $\cup \{A\}$
10: end if
11: end for
12: Init ← $\{\}$; Ends ← $\{\}$
13: for all $\sigma^i \in \Sigma$ do
14:   $\sigma^i_1 = \langle A, \delta \rangle$; $\sigma^{|i|}_d = \langle B, \delta \rangle$ \hspace{1cm} $\triangleright$ Retrieving this information through secondary index
15:   Init$[A]$ ← Init$[A]$ + 1
16:   Ends$[B]$ ← Ends$[B]$ + 1
17: end for
18: for all $A \in I$ s.t. Init$[A] = |\Sigma|$ do
19:   $\Phi$ ← $\Phi \cup \{\langle \text{Init}(A), 100\% \rangle\}$
20: end for
21: for all $B \in I$ s.t. Ends$[B] = |\Sigma|$ do
22:   $\Phi$ ← $\Phi \cup \{\langle \text{End}(B), 100\% \rangle\}$
23: end for
24: return ChoiceFilter, $\Phi$

We now consider the unary frequent itemsets $\pi_i$ with support $\varsigma(\pi_i)$ generated by the FP-Growth algorithm. For the events represented in every trace (Line 4), we perform a linear scan of the CountingTable to obtain the minimum ($n$) and maximum ($N$) non-zero occurrences of the event in every single trace, thus guaranteeing that the event will appear in the trace at least $n$ times and shall never occur in the trace more than $N$ times (Line 7). Next, we can easily infer Init and End clauses by scanning the secondary index of the ActivityTable, associating with each trace the starting and ending event in it (Line 13). We then count the number of traces, starting (Line 15) or ending (Line 16) with a specific activity label. We assert that it is likely that traces will start (end) with an activity label of $A$ if all traces do so (Lines 19 and 22). This differs from our previous implementation, where it was sufficient that at least $\theta \cdot |\Sigma|$ traces started (or ended) with a given activity label to return such associated clauses.

For unary patterns that do not occur in all traces, we might observe that they have no strong correlation with any other binary pattern; therefore, we might want to combine those in a later step described in Section 4.3.5, so as to generate Choice and ExclChoice with greater support and confidence than the one associated with each single infrequent unary pattern. This also differs from ADM, where binary clauses are only generated from frequent binary itemsets. As we show in Section 5.1, this is detrimental to the detection of ExclChoice patterns.

### 4.3. Binary Clause Mining

The generation of binary declarative clauses from frequent binary itemsets can be summarised as follows: (i) after inferring some candidate association rules from each item (Section 4.3.1), (ii) we run the pruning heuristics (Section 3.3), determining whether we can avoid the testing of time-consuming clauses (Section 4.3.2). After this, (iii) we attempt to refine the mined clauses with other ones that provide stricter temporal behaviours
(Section 4.3.3). (iv) If none of these more refined clauses has greater or equal support than the one associated with the association rules in (i), we return the latter or we further relax them, as (Excl)Choice (Section 4.3.4). This algorithm mainly differs from Bolt in these two last points, as this does not extensively exploit the notions of generality and quality for the lattice descent. (v) Last, we generate distinct (Excl)Choice clauses through a combination of pairs of infrequent unary patterns not being represented in any of the clauses being generated in the previous steps (Section 4.3.5). All these steps are summarised in Algorithm 6.

Algorithm 6 Binary clause mining, except from (Excl)Choice

1: $\text{CSviol}(i; A, B) \leftarrow \#_{v,i} A - 1\sigma_{v,i}^{c} = \langle B, \beta \rangle > \#_{v,i} A \vee \#_{v,i} B < \#_{v,i} A$
2: $\text{SViol}(i; A, B) \leftarrow \#_{v,i} A > \#_{v,i} B \vee 2 \cdot \#_{v,i} A < \#_{v,i} B$
3: function $\text{BINARYCLAUSEMINING}(\mathcal{P}, \text{ChoiceFilter}, \text{Beginnings}, \theta)$
   $\triangleright$ Cross-Branch Entailment (ChainSuccession)
4:   $\text{Beginnings} \leftarrow [A \mapsto \{q \in \mathcal{S}|q^{|l|} = \langle A, \delta \rangle \}] |_{A \in \mathcal{L}}$
5:   $\Phi \leftarrow \emptyset$
6:   for all $(\pi_i = \langle A, B \rangle, \xi(\pi_i)) \in \mathcal{P}$ do $\triangleright \Phi \leftarrow \Phi \cup \max(\text{DM}(Q^{|l|}, L, L_h))$
   $\triangleright$ (i) Algorithm 7
7:     $c_{r}, r, \theta', \langle \mathbf{A}, \mathbf{B} \rangle \leftarrow \text{ASSOCIATIONRULESFORDECLARE}(\langle A, B \rangle, \text{Beginnings}, \theta)$
   $\triangleright$ (ii) Computing Pruning Heuristics
8:     $\text{global notChainSucc}_{A, B} \leftarrow 1 - \sum_{i \in \mathcal{L}} \frac{1_{\text{CSviol}}(\langle A, B \rangle)}{|\mathcal{L}|} < \theta'$
9:     $\text{global notChainSucc}_{B, A} \leftarrow 1 - \sum_{i \in \mathcal{L}} \frac{1_{\text{CSviol}}(\langle B, A \rangle)}{|\mathcal{L}|} < \theta'$
10:    $\text{global notSurr}_{A, B} \leftarrow 1 - \sum_{i \in \mathcal{L}} \frac{1_{\text{CSviol}}(\langle A, B \rangle)}{|\mathcal{L}|} < \theta'$
11:    $\text{global notSurr}_{B, A} \leftarrow 1 - \sum_{i \in \mathcal{L}} \frac{1_{\text{CSviol}}(\langle B, A \rangle)}{|\mathcal{L}|} < \theta'$
   $\triangleright$ (iii) Algorithm 8
12: $\Phi' \leftarrow \text{TEMPORALREFINEMENT}(r, \mathcal{S}, \theta')$
   $\triangleright$ (iv) Post-TEMPORALREFINEMENT (with Algorithm 14)
13: if $\Phi' = \emptyset$ and $(\theta + \epsilon \geq 1 \text{ or not } \text{SINGLECHOICEEXCLGEN}(\langle A, B \rangle, \mu, \mu^{-1}, \theta))$ then
14:    $\Phi \leftarrow \Phi \cup \{\epsilon\}$
15: else
16:    $\Phi \leftarrow \Phi \cup \Phi'$
17: end if
18: end if
19: return $\text{CHOICEEXCLGEN}(\Phi, \text{ChoiceFilter}, \theta)$

4.3.1. Association Rule Mining for Declarative Clauses

Algorithm 7 runs a simplified version of the association rule mining algorithm for each binary pattern $\langle A, B \rangle$ generated by Algorithm 4. This step provides a heuristic inference for determining what clauses need to be mined as a refinement of the RespExistence($A, B$) representing an association rule $A \rightarrow B$, as inferred from $\langle A, B \rangle$.
Computers 2023, 12, 185

Algorithm 7 Determining declarative clauses from association rule mining.

1: function ASSOCIATIONRULESFORDECLARE((A, B), θ)
2: p ← (A, B)
3: if c(A → B) > c(B → A) and c(A → B) ≥ θ then
4: c ← RespExistence(A, B); r ← {A → B}
5: θ′ ← max{θ, SUPP(RespExistence(A, B))}
6: else if c(A → B) < c(B → A) and c(B → A) ≥ θ then
7: c ← RespExistence(B, A); r ← {B → A}
8: θ′ ← max{θ, SUPP(RespExistence(B, A))}
9: p ← (B, A)
10: else if c(A → B) = c(B → A) ≥ θ then
11: c ← CoExistence(A, B); r ← {A → B, A → B}
12: θ′ ← max{θ, min{SUPP(RespExistence(A, B)), SUPP(RespExistence(B, A))}}
13: end if
14:▷ Cross-Branch Entailment (ChainSuccession)
15: if r = {T → H} and Beginnings[H] > 0 then
16: r ← {A → B, B → A}
17: end if
18: return c, r, θ′, p

We compute the strengths of two candidate association rules, A → B and B → A, in terms of their confidence. If the confidence of one dominates the other, we choose the strongest, either A → B (Line 4) or B → A (Line 7). As such rules entail that the existence of the antecedent requires the consequent, this associative rule immediately represents a RespExistence. Different from our previous algorithm, this candidate clause is generated but not yielded yet as returning this clause requires ensuring that no “stronger” clause exists. Still, this candidate clause restricts the generation of subsequent candidate declarative rules, as highlighted in Figure 6. This also differs from our previous implementation where we always assumed that the rule’s antecedent will always correspond to an activation and the consequence is always a target, as now A → B is associated with a Precedence(B,A) clause (Section 6.3). If, on the other hand, no rule dominates the other (Line 11), we can only assume that both the activation and target coexist while we assess declarative rules, where one of the two appears to be an activation and the other is a target, thus entailing an A ↔ B pattern representing a CoExistence and two association rules: A → B and B → A.

The generation of a candidate clause enforces the update of the minimum support threshold θ as θ′, as we later want to return more refined clauses than the ones described here only if they have declarative support that is at least equal to one of the clauses that are mined; θ′ is set to the maximum between our previous θ and the support of the currently mined clause (Lines 5 and 8). As the subsequent refinement step (Algorithm 8, Section 4.3.3) will separately consider two distinct interpretations of RespExistence jointly with their associated threshold, we always consider the support of the two associated RespExistence clauses rather than the one from CoExistence (Line 12). This differs from our previous implementation, where we only considered θ as a minimum support threshold in the subsequent refinement step. This restriction will allow us to avoid further undesired behaviour being mined.

Still, we can observe that the heuristic approach described so far without considering the relaxation proposed in Line 15 can be too strict and might discard some relevant declarative clauses:

Example 8. Let us consider the following log:

\[ \mathcal{S} = \{ c^1, c^2, c^3 \} \quad c^1 = AB, c^2 = BD, c^3 = CABAB \]

FPGrowth can yield the frequent itemset \{A, B\}; we now want to generate some association rules. As the confidence of A → B is greater than the one for B → A, we might return only the former,
Algorithm 8 Lattice descent for the mined association rules in \( r \)

1: \textbf{procedure} ASSOCIATIONRULEREFINEMENT(\( A, B, \theta' \))
2: \hspace{1em} PRECEDENCERESPONSE(\( A,B,\theta' \)) \hspace{1em} \( \triangleright \) Algorithm 9
3: \hspace{1em} CHAINPRECEDENCERESPONSE(\( A,B,\theta' \)) \hspace{1em} \( \triangleright \) Algorithm 10

4: \hspace{1em} \textbf{for all} Tail \( \rightarrow \) Head \( \in \) \( r \) \textbf{do}
5: \hspace{2em} ASSOCIATIONRULEREFINEMENT(Tail,Head,\( \theta' \))
6: \hspace{1em} \textbf{end for}
7: \hspace{1em} \textbf{if} \( |r| = 1 \) \textbf{then return} SINGLEBRANCHMINING(\( A',B',\Sigma,\theta' \)) \hspace{1em} \( \triangleright \) Algorithm 11
8: \hspace{1em} \textbf{end for}
9: \hspace{1em} \( \triangleright r = \{ A' \rightarrow B', B' \rightarrow A' \} \)
10: \hspace{1em} \( C \leftarrow \) SINGLEBRANCHMINING(\( A',B',\Sigma,\theta' \))
11: \hspace{1em} \textbf{return} CROSSBRANCHENTAILMENT(\( C; B',A',\Sigma,\theta' \)) \hspace{1em} \( \triangleright \) Algorithm 12

Thus potentially returning only a ChainResponse\( (A,B) \) clause. Still, we can observe that \( \sigma^2 \) vacuously satisfies both ChainResponse\( (A,B) \) and ChainPrecedence\( (B,A) \). The vacuousness of the former is already encompassed by the other traces satisfying both ChainResponse\( (A,B) \) and contributing to the support of \( A \rightarrow B \); the vacuousness of the latter implies an absence of \( A \) event being labelled in the data. This prevents us from mining \( B \rightarrow A \) and returning a ChainSuccession\( (A,B) \).

To alleviate the aforementioned issue, we, therefore, consider both possible association rules for adequately mining the ChainSuccession clauses from the data if there exists at least one trace in the log starting with the activity label associated with the rule’s head (Line 15).

As this step avoids the immediate return of either RespExistence or CoExistence, we need to first check if any refined clauses stronger than these can be returned instead; this helps reduce the specification size.

4.3.2. Computation of Pruning Heuristics

Before refining the clauses mined in the previous stage, we want to compute the pruning heuristics in Section 3.3 to avoid time-consuming checks required by ChainSuccession and Surround in Algorithm 10 by merely exploiting the CountingTable. \( \theta' \) reflects the minimum tolerated number of traces for any candidate stronger than \( c \) from Line 7 in Algorithm 6 should satisfy, as any refinement of \( c \) having more violations than \( c \) cannot be deemed as a clause stronger than \( c \) and, therefore, shall not be considered in the final mined specification. Upon violating the heuristic constraints of a specific trace, we can infer that either ChainSuccession (Line 1) or Succession (Line 2) is violated for a specific trace in the log; to compute the number of non-violating clauses, we decrease one by the ratio of the number of violating clauses on the overall number of traces within the log. We can then determine that ChainSuccession\( (A,B) \) shall never be returned as a clause consuming \( c \) if the aforementioned value is strictly less than \( \theta' \) (Line 8). Similar considerations can be performed for Surround clauses (Line 10–11) and ChainSuccession\( (B,A) \) (Line 9).

This phase significantly differs from our previous implementation, as the heuristics are now computed before attempting to refine the clause while also considering Surround.

4.3.3. Temporal Refinements

Algorithm 8 describes the attempt to refine either RespExistence or CoExistence with more precise temporal behaviour. For each association rule \( A \rightarrow B \), we attempt to descend the lattice from RespExistence\( (A,B) \) (Line 6), irrespective of this being associated with a CoExistence\( (A,B) \). We exploit the quality criterion in Definition 3 for the lattice descent.
Algorithm 9 Scanning Precedence(B,A) and Response(A,B) for association rule A → B

1: procedure PrecedenceResponse(A, B, \( \theta' \))
2:   \( \mathit{it}_A \leftarrow \mathsf{SORT}_t(\{\mathit{ct}_{A,i} \mid \mathit{ct}_{A,i} \in \mathcal{E}, \mathit{ct}_{A,i} = \mathit{A}\}); \mathit{ct}_{A,i} \leftarrow \mathsf{CURRENT}(\mathit{it}_A) \)
3:   \( \mathit{it}_B \leftarrow \mathsf{SORT}_t(\{\mathit{ct}_{B,i} \mid \mathit{ct}_{B,i} \in \mathcal{E}, \mathit{ct}_{B,i} = \mathit{B}\}); \mathit{ct}_{B,i} \leftarrow \mathsf{CURRENT}(\mathit{it}_B) \)
4:   \( \mathit{retain} \leftarrow \mathsf{false}; \mathit{retain}_a \leftarrow -1 \)
5:   \( \mathit{vac}_{A,B} \leftarrow \{1, \ldots , i - 1\}; \mathit{p}_A \leftarrow i \)
6:   while \( \mathit{ct}_{A,i} \neq \mathit{ct}_{B,i} \) do
7:     \( \triangleright \) Start scanning from the first A event in trace \( \mathit{ct}_{A,i} \)
8:     if \( 1 - |\mathit{viol}_{A,B}^P|/|\mathcal{E}| < \theta' \) and \( 1 - |\mathit{viol}_{B,A}^P|/|\mathcal{E}| < \theta' \) then break
9:     end if
10:    if \( \mathit{retain} \) and (\( \mathit{ct}_{A,i}^* = \mathit{ct}_{B,i}^* \) or \( i \geq \mathit{retain}_a \)) then
11:       \( \mathit{retain} \leftarrow \mathsf{false}; \mathit{retain}_a \leftarrow -1 \)
12:    end if
13:    if \( \mathit{p}_A \neq i \) then
14:       \( \mathit{vac}_{A,B} \leftarrow \mathit{vac}_{A,B} \cup \{\mathit{p}_A + 1, \ldots , i - 1\}; \mathit{p}_A \leftarrow i \)
15:    end if
16:    if \( \mathit{ct}_{A,i}^* = \mathit{ct}_{B,i}^* = \mathit{ct}_{A,i} \) then \( \triangleright \) A’s on their own
17:       \( \mathit{act}_{A,B}^* \leftarrow \mathit{act}_{A,B}^* \cup \{i\}; \mathit{viol}_{A,B}^* \leftarrow \mathit{viol}_{A,B}^* \cup \{i\} \)
18:    else if \( \mathit{ct}_{A,i}^* = \mathit{ct}_{B,i}^* = \mathit{ct}_{B,i} \) then \( \triangleright \) B’s on their own
19:       \( \mathit{act}_{B,A}^* \leftarrow \mathit{act}_{B,A}^* \cup \{i\}; \mathit{act}_{A,B}^* \leftarrow \mathit{act}_{A,B}^* \cup \{i\} \)
20:    end if
21:    if \( \mathit{ct}_{B,i} = \mathit{ct}_{A,i} \) then
22:       \( \mathit{act}_{B,A}^* \leftarrow \mathit{act}_{B,A}^* \cup \{i\}; \mathit{act}_{A,B}^* \leftarrow \mathit{act}_{A,B}^* \cup \{i\} \)
23:    end if
24:    if \( \mathit{ct}_{B,i} = \mathit{ct}_{A,i} \) then
25:       \( \mathit{act}_{B,A}^* \leftarrow \mathit{act}_{B,A}^* \cup \{\alpha \mid i \leq \alpha \leq |\mathcal{E}|, \#\sigma \neq \mathit{B} = 0\} \)
26:    end if
27: else if \( \mathit{ct}_{A,i}^* = \mathit{ct}_{B,i}^* = \mathit{ct}_{B,i} \) then \( \triangleright \) Presence of both
28:    end if
29:    if \( x > j \) then \( \triangleright \) violation
30:       \( \mathit{viol}_{A,B}^* \leftarrow \mathit{viol}_{A,B}^* \cup \{i\} \)
31:       for all \( \mathit{ct}_{A,i} \) s.t. \( \mathit{ct}_{A,i}^* = \mathit{ct}_{B,i} \geq j \) do
32:          while \( k < j \) and \( i = k \) do
33:             \( \triangleright \) for \( \mathit{ct}_{A,i} \) and \( \mathit{ct}_{B,i} \) both in trace \( \mathit{ct}_{B,i} \)
34:                \( \mathit{act}_{A,B}^* \leftarrow \mathit{act}_{A,B}^* \cup \{i\}; \mathit{act}_{B,A}^* \leftarrow \mathit{act}_{B,A}^* \cup \{i\} \)
35:             \( \mathit{viol}_{A,B}^* \leftarrow \mathit{viol}_{A,B}^* \cup \{i\} \); break
36:          end while
37:       \( \triangleright \) for \( \mathit{ct}_{A,i} \) in trace \( \mathit{ct}_{A,i} \)
38:       end if
39:    end if
40:   end for
41:   if \( \mathit{ct}_{A,i} \neq \mathit{ct}_{B,i} \) then
42:      \( \triangleright \) at the end of all A events
43:      \( \mathit{vac}_{A,B} \leftarrow \mathit{vac}_{A,B} \cup \{\mathit{p}_A + 1, \ldots , |\mathcal{E}|\} \)
44:   \end{algorithm}
For the clauses built upon refinements of \( \text{RespExistence}(A,B) \) for the association rule \( A \rightarrow B \) are \( \text{Response}(A,B) \) and \( \text{Precedence}(B,A) \) (both on Line 2), as well as \( \text{ChainResponse}(A,B) \) and \( \text{ChainPrecedence}(A,B) \) (both computed on Line 3), we can restrict the computations of the activated \( (act^*_A,B) \), violated \( (viol^*_A,B) \), and vacuous traces \( (vac_A,B) \) to only the aforementioned clauses \(*\). Observe that all satisfying traces constitute the complement of the violating traces \( (sat^*_A,B) = \overline{viol^*_A,B} \). This is different from ADM+S [21], where such sets are computed all the time for all clauses, and not only the basic building blocks, thus providing significant space and time overheads.

After this, we can then express the scoring functions for the declarative clauses as follows:

\[
\text{SUPP}_\Theta(\ast(A,B)) = \frac{|sat^*_A,B|}{|\Theta|}, \quad \text{RSUPP}_\Theta(\ast(A,B)) = \frac{|sat^*_A,B \cap act^*_A,B|}{|\Theta|},
\]

\[
\text{REConf}_\Theta(\ast(A,B)) = \frac{|sat^*_A,B \cap act^*_A,B|}{|act^*_A,B \cup viol^*_A,B|}.
\]

If the mining heuristic only provides one association rule \( \text{Tail} \rightarrow \text{Head} \), we only perform the lattice descent for the rules that are derivable from this (Line 7). Otherwise, after caching the clauses associated with the \( \text{Tail} \rightarrow \text{Head} \) rule in \( C \) (Line 10), we finalise the lattice descent for the other \( \text{Head} \rightarrow \text{Tail} \) rule while testing descendant clauses across all its specialisations (Line 11).

This mining step provides the following improvements when compared to the existing competing approaches:

(a) We exploit early stopping conditions via both pruning heuristics, as in Sections 3.3 and 4.3.2, as well as testing how many traces violate the condition associated with a specific declarative clause \(*\). For the latter, if we know that at least \( \tau \) traces violate \(*\), such that \( 1 - \frac{\tau}{|\Theta|} < \theta' \), we can stop testing the remaining traces as we have already found a violation to the minimum support threshold \( \theta' \). This is one of the main differences with the algorithms reducing specification mining to formal verifications with an exhaustive search, as they test all the traces without any stopping condition, either because they still test one trace at a time (ADM+S [21]) or because multiple traces are assessed from the log by design (xtLTLf-based algorithms, re-implementation of ADM [20], or TopN Declare [2]).

(b) We test multiple clauses for each trace by merging compatible clause-testing conditions, minimising both data access and testing for violations, activations, and vacuity conditions. Different from our temporal formal verification pipeline [2], and only ensuring this for declarative clauses that share common sub-expressions, we show that we can further improve this by merging clauses that also have fairly distinct temporal behaviour (Precedence and Response). We further enhance the computations of ChainPrecedence and ChainResponse clauses by accessing the immediately preceding or following events through the \( \text{ActivityTable}_x \)'s \( \text{Prev} \) and \( \text{Next} \) pointers without the need to use an \( \ominus \phi \) operator, thus further reducing the data access to the knowledge base. Observe that other algorithmic implementations for formal verification, such as ADM+S [21], do not offer even the basic optimizations that our temporal verification pipeline does [2].

When compared to our previous heuristic algorithm Bolt [5], this phase mainly differs in the following aspects:

(c) We interpret \( \text{Precedence}(A,B) \) as a candidate clause for association rule \( B \rightarrow A \) rather than one for \( A \rightarrow B \) (see the discussion in Section 6.3).

(d) We refine the \( \text{(CHAIN)PRECEDENCE\_RESPONSE} \) to better take into account the vacuous conditions associated with ChainPrecedence for events occurring at the beginning of the trace.
Despite the temporal semantics of Computers 2023, 12, 185, A (Line 2) and We can easily determine that the presence of Activation detection
As the traces violating a clause complement the traces satisfying it, Violation detection
discuss how each condition is collected for each trace \( \sigma \)
candidate-activating events (Lines 47 and 48). To show the correctness of our approach, we are violated (Lines 7 and 8). When this occurs, we skip to the following trace containing the indexing phase. We extensively check for all such events in a trace until both clauses ActivityTable access the \( x_t \) activating, violating, and vacuously satisfying/violating either of the two clauses within A requires that each occurrence of A needs a future presence of a B; we can test for traces activating, violating, and vacuously satisfying/violating either of the two clauses within the same Algorithm 9. Similar to the implementation of our \( \pi\text{LTL}_f \) operators [2], we access the ActivityTable \( \sigma \) for returning all the events associated with the activity labels \( \lambda \) (Line 2) and \( \beta \) (Line 3), pre-sorted by trace and event in lexicographical order during the indexing phase. We extensively check for all such events in a trace until both clauses are violated (Lines 7 and 8). When this occurs, we skip to the following trace containing candidate-activating events (Lines 47 and 48). To show the correctness of our approach, we discuss how each condition is collected for each trace \( \sigma^\iota \) in our log.

**Activation detection** We can easily determine that the presence of \( \lambda \)-s is the minimum requirement for the activation of \( \text{Response}(\lambda, \beta) \); this encompasses traces containing either just \( \lambda \)-labelled events (Line 16) or those alongside \( \beta \)-labelled ones (Line 37). Similarly, \( \text{Precedence}(\beta, \lambda) \) is activated when at least one \( \beta \) appears in the trace; this encompasses traces containing only \( \beta \)-labelled events (Line 29) as well as ones containing both types of labels (Line 38). Observe that a clause’s activation does not necessarily entail its satisfaction.

**Violation detection** As the traces violating a clause complement the traces satisfying it, we collect only violating traces. A trace containing only \( \lambda \) events provides a violation for both clauses of interest, \( \text{Response}(\lambda, \beta) \) and \( \text{Precedence}(\beta, \lambda) \), as the former contains no \( \beta \) events of interest (Line 16) while the latter violates the requirements that no \( \lambda \)-s must occur if no \( \beta \) is present in the trace (Line 17). A trace containing both types of events violates \( \text{Precedence}(\beta, \lambda) \) if the first occurring \( \beta \equiv \sigma^\iota_k \) occurs after the first existing \( \lambda \equiv \sigma^\iota_j \), violating \( \neg \beta \land \lambda \) (Line 39), while \( \text{Response}(\lambda, \beta) \) is violated the first time (break) we cannot find a \( \beta \)-labelled \( \sigma^\iota_k \) event occurring after the current \( \lambda \)-labelled event \( \sigma^\iota_j \) of interest (Line 43), even after proactively scanning for such a desired \( \beta \) event, as in Line 42.

**Vacuity Detection** While scanning all the \( \lambda \)-labelled events sorted by trace and event IDs, the traces vacuously satisfying \( \text{Response}(\lambda, \beta) \) are the ones where no \( \lambda \) appears, which are all the traces preceding the traces containing the first occurrences of such event types (Line 5), the ones after the last trace where \( \lambda \) occurs (Line 50), as well as all the traces occurring between the previous (\( p_i \)) and current (\( i \)) traces containing \( \lambda \)-labelled events (Line 13). Regarding \( \text{Precedence}(\beta, \lambda) \), we shall have either vacuously satisfied or vacuously violated traces if we detect no \( \beta \) events within the trace. This includes all traces between the current one containing only \( \lambda \) (\( i \)) and the next one containing at least one \( \beta \) (\( i, \text{Line 21} \)); when we observe a trace \( i \) containing all \( \beta \)-labelled events, all remaining traces \( a \) up to a subsequent trace containing at least one event \( \lambda \) (if any), should not have any \( \beta \)-labelled events (\( \#_{\infty} \beta = 0, \text{Line 33} \)). For efficiency purposes, in the first scenario, we avoid inserting these values multiple times by detecting whether some vacuous traces are already retained in the past.
Algorithm 10 Scanning ChainPrecedence(B,A) and ChainResponse(A,B) for rule A → B

1: procedure ChainPrecedenceResponse(A, B, θ′)
2: \( i_{λ} \leftarrow \text{SORT}_{j}(\{ σ_{j}^i | σ_{j}^i ∈ σ^i, σ^i ∈ S, σ_{j}^i = A \}); σ_{j}^i ← \text{CURRENT}(i_{λ}) \)
3: forward_ ← false; forward_p ← false; \( p_a ← 1 \)
4: start ← \( σ_{j}^i \)
5: \( \text{vac}_{A,B}^p ← \text{vac}_{A,B}^r; \text{vac}_{A,B}^r ← \text{vac}_{A,B} \)
6: while \( σ_{j}^i \neq \text{forward} \) do
   \( \triangleright \) Start scanning from the first A event in trace \( σ^i \)
7: if \( 1 - |\text{viol}_{B,A}^r|/|S| < θ′ \) and \( 1 - |\text{viol}_{B,A}^p|/|S| < θ′ \) and \( \text{notSurr}_{A,B} \) and
8: \( (\text{notChainSucc}_{A,B} \text{ or } 1 - |\text{viol}_{B,A}^r \cup \text{viol}_{B,A}^p|/|S| < θ′) \) and
9: \( (\text{notChainSucc}_{B,A} \text{ or } 1 - |\text{viol}_{B,A}^r \cup \text{viol}_{B,A}^p|/|S| < θ′) \) then break
10: if \( i \neq p_a \) then
11:   forward_ ← false; forward_p ← false; \( p_a ← i \)
12: end if
13: if not forward_ and \( (j = |σ^i| \text{ or } σ_{j+1}^i \neq B) \) then
14:   \( \text{viol}_{A,B}^r ← \text{viol}_{A,B}^r \cup \{i\}; \text{forward}_r ← \text{true} \)
15: end if
16: if not forward_p and \( j > 1 \text{ and } σ_{j-1}^i \neq B \) then
17:   \( \text{viol}_{A,B}^p ← \text{viol}_{A,B}^p \cup \{i\}; \text{forward}_p ← \text{true} \)
18: end if
19: if \( σ_{j}^i = \text{start} \text{ or } (σ_{j}^i ← \text{PREV}(i_{λ}) \text{ and } i \neq i) \) then
20:   \( \text{act}_{A,B}^r ← \text{act}_{A,B}^r \cup \{i\} \)
21:   if \( j > 1 \text{ or } \#_{A,A} > 1 \) then
22:     \( \text{act}_{A,B}^p ← \text{act}_{A,B}^p \cup \{i\} \)
23:   else if \( j = 1 \text{ and } \#_{A,A} = 1 \) then
24:     \( \text{vac}_{A,B}^p ← \text{vac}_{A,B}^p \cup \{i\} \)
25: end if
26: end if
27: \( p_a ← i; \text{NEXT}(i_{λ}); σ_{j}^i ← \text{CURRENT}(i_{λ}) \)
28: while forward_ and forward_p and \( p_a \neq i \) do \( \text{NEXT}(i_{λ}); σ_{j}^i ← \text{CURRENT}(i_{λ}) \)
29: end while
30: end while

ChainPrecedence (∗ = cp) and ChainResponse (∗ = cr), Algorithm 10 determines the coverage of both ChainPrecedence and ChainResponse over our log of interest. This leverages the following main optimisations: The use of pruning heuristics to avoid costly trace and log scans (Lines 7–9) and extend vacuously satisfied conditions from the previous algorithm to minimise the memory footprint (Line 5 vs. 24). Unlike the last step, we only scan the events associated with the activation conditions for both clauses while accessing the immediately preceding and following events to test the satisfaction of the target condition. While iterating over all A-labelled events as candidate activation conditions for both clauses, we extensively check for all such events in a trace (Line 27) until both clauses are violated. When this occurs, we skip to the following trace containing candidate-activating events (Line 28).

Similar to the previous step, we show the correctness of such an algorithm by explaining our rationale for detecting trace activating, violating, or vacuously satisfying of each clause.

**Activation detection** We determine the presence of activations at each first occurrence of A-labelled event \( σ_{j}^i \) per trace \( σ^i \). As any occurrence of A in a trace is a valid activation for a ChainResponse, we add it straight away (Line 19). Concerning activations for ChainPrecedence(A,B), \( σ_{j}^i \) is a plausible candidate activation if it occurs after the first
...event of the trace \((j > 1)\); otherwise, if this is the first event of the trace \((j = 1)\), trace \(\sigma^j\) can have a future activation only if there are further \(A\)-labelled events in \(\sigma^j\ (\#_{A}A > 1, \text{ Line 22})\).

**Violation detection** A trace \(\sigma^j\) violates the ChainResponse\((A,B)\) whenever we find \(A\)-labelled event at the end of the trace \((j = |\sigma^j|)\) or that does not have the following \(B\)-labelled event (Line 14); once a violation in the trace is detected, we can skip testing for further violations. Similarly, a trace will violate the ChainPrecedence\((A,B)\) if we find an event occurring after the first event with a non-\(B\)-labelled event, as a preceding one (Line 17). We also test these conditions for each new trace of interest (Line 11).

**Vacuity detection** As previously mentioned, this step inherits all vacuous traces being computed for the previous set of clauses, Precedence and Response. While ChainResponse\((A,B)\) shares the same set of vacuously satisfying traces as Response\((A,B)\), ChainPrecedence\((A,B)\) extends the former by adding all the traces, starting with the sole occurrence of an \(A\)-labelled event (Line 24).

---

**Algorithm 11** Single-branch mining.

```plaintext
1: function SINGLEBRANCHMINING\((A,B,\Sigma, \theta')\)
2: if \(|\text{sat}_{A,B}^{\theta'}| \geq \theta'|\Sigma| \text{ or } |\text{sat}_{A,B}^{\theta'}| \geq \theta'|\Sigma|\) then
3:    if \(|\text{act}_{A,B}^{\theta'}| \geq \theta'|\Sigma|\) then
4:        if \(|\text{sat}_{A,B}^{\theta'}| \geq \theta'|\Sigma|\) \text{ and } \text{sat}_{A,B}^{\theta'} \geq \text{sat}_{A,B}^{\theta'} \text{ and } \text{act}_{A,B}^{\theta'} \neq \emptyset \text{ and } \text{sat}_{A,B}^{\theta'} \cap \text{act}_{A,B}^{\theta'} \neq \emptyset\) then
5:            yield ChainResponse\((A,B)\)
6:        else if \(\text{act}_{A,B}^{\theta'} \neq \emptyset \text{ and } \text{sat}_{A,B}^{\theta'} \cap \text{act}_{A,B}^{\theta'} \neq \emptyset\) then
7:            yield Response\((A,B)\)
8:    end if
9: end if
10: if \(|\text{sat}_{B,A}^{\theta'}| \geq \theta'|\Sigma|\) \text{ and } \text{act}_{B,A}^{\theta'} \neq \emptyset \text{ and } \text{sat}_{B,A}^{\theta'} \cap \text{act}_{B,A}^{\theta'} \neq \emptyset\) then
11:    yield Precedence\((B,A)\)
12: end if
13: end if
14: if \(|\text{sat}_{A,B}^{\theta'}| \geq \theta'|\Sigma|\) \text{ and } \text{act}_{A,B}^{\theta'} \neq \emptyset \text{ and } \text{sat}_{A,B}^{\theta'} \cap \text{act}_{A,B}^{\theta'} \neq \emptyset\) then
15:    \(\text{act}_{A,B}^{\theta'} \leftarrow \text{act}_{A,B}^{\theta'} \cup \text{sat}_{A,B}^{\theta'} \cap \text{act}_{A,B}^{\theta'} \text{ and } \text{sat}_{A,B}^{\theta'} \leftarrow \text{sat}_{A,B}^{\theta'} \cap \text{act}_{A,B}^{\theta'}\) \triangleright \text{Surround}
16: if \(|\text{sat}_{A,B}^{\theta'}| \geq |\text{sat}_{A,B}^{\theta'}|\) \text{ and } \text{sat}_{A,B}^{\theta'} \geq |\text{sat}_{A,B}^{\theta'}|\) \text{ and } \text{act}_{A,B}^{\theta'} \neq \emptyset \text{ and } \text{sat}_{A,B}^{\theta'} \cap \text{act}_{A,B}^{\theta'} \neq \emptyset\) then
17:    yield Surround\((A,B)\)
18: else
19:    yield ChainPrecedence\((A,B)\)
20: end if
21: end if
```

Single-branch mining. We collected all relevant information \((\text{vac}_{A,B}^{\theta'}, \text{viol}_{A,B}^{\theta'}, \text{act}_{A,B}^{\theta'}, \text{act}_{A,B}^{\theta'}\)) associated with the basic constituent clauses \(\star\); we will now discuss the subroutine in Algorithm 11 to determine the insertion of each clause in the final specification. When dealing with one single association rule per frequent itemset (e.g., either \(A \rightarrow B\) or \(B \rightarrow A\) for a frequent itemset \(\{A, B\}\)), we return neither Succession nor ChainSuccession clauses, as they require the return of both association rules to be returned. As ChainResponse entails Response, we can avoid testing the return of the former if the latter violates the quality criterion (Line 3). Moreover, we produce the former only if its support is at least equal to the one of the latter (Line 5); otherwise, the latter is returned instead (Line 7). We produce a Precedence if, as per the previous cases, this has support greater or equal to the given minimum threshold \(\theta'\), while guaranteeing that this clause is not vacuously satisfied for all traces in the log (Line 11). Finally, we return a Surround clause, if this support is not less than one of the associated ChainResponse and ChainPrecedence clauses (Line 17); otherwise, ChainPrecedence is returned (Line 19) alongside ChainResponse, as per previous testing.
Cross-branch entailment. Cross-branch entailment is triggered by the co-presence of two specular association rules for a single frequent itemset. In this scenario, we might have a descendant clause stronger than the ancestor ones stemming from different association rules. We cannot simply add clauses to the specification upon the first refinement attempt, and we need to retain the information of all descendants to compute the metrics of the entailing clause correctly.

**Algorithm 12** Cross-branch entailment (1/2).

```plaintext
1: function CROSSBRANCHENTAILMENT(C, B, a, θ)
2:   if |sat_{A,B}^C| < θ |Ω| and |sat_{A,B}^B| < θ |Ω| and |sat_{A,B}^A| < θ |Ω| and |sat_{A,B}^F| < θ |Ω| then
3:     return C
4:   end if
5:   surrYield ← false
6:   if |sat_{A,B}^F| ≥ θ |Ω| and act_{A,B}^F ∉ Ω and sat_{A,B}^F ∩ act_{A,B}^F ≠ ∅ then
7:     act_{A,B}^F ← act_{A,B}^F ∪ act_{A,B}^F; sat_{A,B}^F ← sat_{A,B}^F ∩ act_{A,B}^F > Surround
8:     if |sat_{A,B}^F| ≥ |sat_{A,B}^F| and |act_{A,B}^F| ≥ |act_{A,B}^F| and
9:        surrYield ← true
10:   end if
11:   act_{A,B}^F ← act_{A,B}^F ∪ act_{A,B}^F; sat_{A,B}^F ← sat_{A,B}^F ∩ sat_{A,B}^F > ChainSucc.(A,B)
12:   if |act_{A,B}^F| ≥ θ |Ω| and |act_{A,B}^F| ≥ |act_{A,B}^F| and
13:      surrYield ← true
14: end if
15: else
16:     if not surrYield then
17:       yield ChainPrecedence(B,A)
18:     end if
19:     if Surround(A, B) ∉ C and ChainResponse(A, B) ∈ C then
20:       yield ChainResponse(A,B)
21:     end if
22: end if
23: end if
```

**Example 9.** Consider the following log:

\[ \Gamma = \{ \sigma^1, \sigma^2, \sigma^3 \} \quad \sigma^1 = CBAB, \sigma^2 = BBAB, \sigma^3 = BABB \]

ChainResponse(A, B), satisfying all the traces in the log, is a constituent of both Surround(A, B) and ChainSuccession(A, B). As Surround also requires the verification of ChainPrecedence(A, B), the inclusion in the final specification of the former can be determined while running TEMPORAL-REFIMENENT \((A, B, \theta')\) for \(A \rightarrow B\), while determining this for ChainSuccession(A, B) requires also invoking TEMPORAL-REFIMENENT \((B, A, \theta')\), as its other constituent clause (ChainPrecedence(B, A)) can only be determined in this additional step. Furthermore, the provision of ChainSuccession(A, B) in the final specification requires the mining of both association rules \(A \rightarrow B\) and \(B \rightarrow A\), to activate both refining steps. Observe that this is also the case in the current example, as both association rules share the same confidence score.

The resulting procedure is described in Algorithms 12 and 13. This works as follows: if the candidate clauses that are mined from the second association rule yield no additional clause, we then return all the clauses that are preemptively cached in C (Line 3). Otherwise, we extend the former algorithm for single-branch mining to finally return the clauses from either the cache or the newly visited branch. We produce the specular (but not symmetric) Surround clause (Surround(B,A)) similar to the previous branch (Line 9), while remembering whether this was inserted so as to avoid the return of duplicate Chain clauses (Line 10). We
can return ChainSuccession(A,B) (and ChainSuccession(B,A)) if such a clause is at least non-vacuously satisfied by at least one trace in the log, and has support that is greater or equal to one of the constituent clauses (Lines 14 and 29, respectively). If this condition is not met, we can return each constituent if and only if corresponding Surround clauses are not previously returned in both branches (Line 17 or 20, and Line 32 or 32) or the other constituent delivered from the first lattice visit is actually cached in C (Line 20 and 35). Similarly, we can return Succession(B,A) (or Succession(A,B)) only if this clause is non-vacuously supported by at least one trace in the log and its associated support is greater or equal to each one of its constituents (Line 41 or 53). Otherwise, we return each constituent if either are mined in the current step (Lines 43 or 55) or were already cached in the previous step (Line 55 or 57). We can easily observe that this algorithm considers all possible combinations from the lattice represented in Figure 2.

4.3.4. Temporal Refinement Post-processing

Despite the previous attempt at refining the clauses initialised in Section 4.3.1, we might not have found an adequate refinement. In this situation, we want to add RespExistence or a CoExistence instead if and only if a (Excl)Choice clause has better support (i.e., strictly greater) than the previous. Although this condition might seem to contradict the assumptions of the S-border, please consider that we start directly with the lattice descent from the RespExistence and CoExistence clauses and not from the top of the lattice, represented by (Excl)Choice clauses. Furthermore, the price we are willing to pay for discarding RespExistence in favour of a more general clause that doesn’t offer entailment information, such as Choice, is only increased data coverage. Lines 13–17 in Algorithm 6 describe this second last step.

**Example 10.** Let us consider the following log:

\[ S = \{ \sigma^1, \sigma^2, \sigma^3, \sigma^4, \sigma^5 \} \]

\[ \sigma^1 = BA, \sigma^2 = AB, \sigma^3 = A, \sigma^4 = \sigma^5 = B \]

After mining an A \( \rightarrow \) B as \( c(A \rightarrow B) > c(B \rightarrow A) \), we generate RespExistence(A,B) with support of 0.8. However, Choice(A,B) has greater support than the previous clause, as this one is satisfied by each trace in the log. So, we can consider generating this last clause instead of considering RespExistence(A,B).

4.3.5. (Excl)Choice Generation

Finally, we generate further declarative clauses from the paring non-certain unary patterns if they do not lead to the generation of other declarative clauses. The steps for achieving this are summarised in Algorithm 14. We first collect all distinct pairs of activity labels that occur in any of the mined clauses (Line 12) while ensuring that only one instance between \( \langle A, B \rangle \) and \( \langle B, A \rangle \) is retained using a non-symmetric predicate \( \sqsubset \) (e.g., a string lexicographical order on the activity labels). For each pair \( \langle A, B \rangle \) (Line 15), we preliminary generate a candidate declarative clause if the ratio between the number of traces of either A or B on the overall log size is greater or equal to the mining parameter \( \theta \) (Line 3). We return a simple Choice clause if there exists at least one clause sharing both event IDs and ExclChoice if all traces containing A-labelled events do not contain B-s, and vice-versa. We then populate a global map \( M \) associating the currently mined clausal support and the first activation of the mined clause (e.g., A) to the clauses exhibiting such value. Map \( M \) was preliminarily filled by the post-TemporalRefinement phase (Algorithm 6 Line 13).
Algorithm 13 Cross-branch entailment (2/2).

24: if $|\text{sat}^{p}_{B,A}| \geq \theta'|\mathcal{S}$ or $|\text{sat}^{p}_{A,B}| \geq \theta'|\mathcal{S}$ then
25: \hspace{1em} if $|\text{sat}^{cr}_{B,A}| \geq \theta'|\mathcal{S}$ then
26: \hspace{2em} if $|\text{sat}^{cr}_{B,A}| \geq \theta'|\mathcal{S}$ and $|\text{sat}^{cr}_{B,A}| \geq |\text{sat}^{p}_{B,A}|$ and
27: \hspace{3em} \text{act}^{cr}_{B,A} \leftarrow \text{act}^{cr}_{B,A} \cap \text{act}^{p}_{B,A}$; $\text{sat}^{cr}_{B,A} \leftarrow \text{sat}^{cr}_{B,A} \cap \text{act}^{p}_{B,A}$ \hspace{1em} \triangleright \text{ChainSucc.}(B,A)
28: \hspace{2em} if $|\text{act}^{cr}_{B,A}| \geq \theta'|\mathcal{S}$ and $|\text{act}^{cr}_{B,A}| \geq |\text{act}^{p}_{B,A}|$ and
29: \hspace{3em} $|\text{act}^{p}_{B,A}| \neq \emptyset$ and $\text{sat}^{cr}_{B,A} \setminus \text{act}^{cr}_{B,A} \neq \emptyset$
30: \hspace{2em} \text{yield} \hspace{0.5em} \text{ChainSucc.}(B,A)
31: \hspace{1em} \text{else} \hspace{0.5em} \text{if not surrYield} then
32: \hspace{2em} \text{yield} \hspace{0.5em} \text{ChainResponse}(B,A)
33: \hspace{1em} \text{end if}
34: \hspace{1em} if $\text{Surround}(A,B) \notin C$ and $\text{ChainPrecedence}(A,B) \in C$ then
35: \hspace{2em} \text{yield} \hspace{0.5em} \text{ChainPrecedence}(A,B)
36: \hspace{1em} \text{end if}
37: \hspace{1em} \text{else if $|\text{act}^{p}_{B,A}| \neq \emptyset$ and $\text{sat}^{p}_{B,A} \cap \text{act}^{p}_{B,A} \neq \emptyset$ then}
38: \hspace{2em} $\text{act}^{p}_{B,A} \leftarrow \text{act}^{p}_{B,A} \cup \text{act}^{p}_{B,A}$; $\text{sat}^{p}_{B,A} \leftarrow \text{sat}^{p}_{B,A} \cap \text{act}^{p}_{B,A}$ \hspace{1em} \triangleright \text{Succ.}(B,A)
39: \hspace{2em} if $|\text{act}^{p}_{B,A}| \geq \theta'|\mathcal{S}$ and $|\text{act}^{p}_{B,A}| \geq |\text{act}^{p}_{B,A}|$ and
40: \hspace{3em} $|\text{act}^{p}_{B,A}| \neq \emptyset$ and $\text{sat}^{p}_{B,A} \setminus \text{act}^{p}_{B,A} \neq \emptyset$
41: \hspace{2em} \text{yield} \hspace{0.5em} \text{Succession}(B,A)
42: \hspace{1em} \text{else}
43: \hspace{2em} \text{yield} \hspace{0.5em} \text{Response}(B,A)
44: \hspace{2em} if $\text{Precedence}(B,A) \in C$ then
45: \hspace{3em} \text{yield} \hspace{0.5em} \text{Precedence}(B,A)
46: \hspace{2em} \text{end if}
47: \hspace{1em} \text{end if}
48: \hspace{1em} \text{end if}
49: \hspace{1em} \text{end if}
50: \hspace{1em} \text{end if}
51: \hspace{1em} \text{end if}
52: \hspace{1em} \text{end if}
53: \hspace{1em} \text{end if}
54: \hspace{1em} \text{else}
55: \hspace{2em} \text{yield} \hspace{0.5em} \text{Precedence}(A,B)
56: \hspace{2em} if $\text{Response}(B,A) \in C$ then
57: \hspace{3em} \text{yield} \hspace{0.5em} \text{Response}(B,A)
58: \hspace{2em} \text{end if}
59: \hspace{1em} \text{end if}
60: \hspace{1em} \text{end if}
61: \hspace{1em} \text{end if}
After fully populating \( M_s \), we now return some of the clauses collected in \( M \) for the final specification; we prefer returning (Excl)Choice clauses with perfect (1.0) support associated with a given activity label \( A \), if any (Line 18); if not, we resort to returning those with non-perfect support (Line 23).

After this final step, we directly return the conjunctive declarative specification to the user with the novel (Excl)Choice clauses that are mined (Algorithm 6 Line 19, Algorithm 3 Line 4).

Note that this is different from our previous Bolt, as at this stage, we add any (Excl)Choice clause without evaluating their quality. Furthermore, we do not abstain from including these if another temporally superior clause with the same activity label is considered.

5. Results

Given the competing approaches (ADM, ADM+S, and TopN Declare), all rely on propositionalisation for generating declarative clauses for a conjunctive specification; we completely rerun these algorithms under the same circumstances as the previous paper [5]. As Bolt2 supports more declarative clauses than our predecessor, Bolt, we extend our prior re-implementation of the aforementioned competing algorithms to keep all the declarative clauses supported by Bolt2, such as ChainSuccession, Succession, and Surround. In ADM+S, we also implement the CoExistence clause, as this was not considered in our previous implementation or benchmark [5]. As in our previous contribution, we upgrade ADM to compute declarative clause support through KnoBAB, thus requiring the need to load

Algorithm 14 Generating (Excl)Choice from pair-wise infrequent behaviour

1: function SINGLECHOICEEXCLGEN\((A,B), \mu, \mu^{-1}, \theta, \phi\)
2: \( s \leftarrow rs \leftarrow rc^+ \leftarrow \frac{\mu(A) \cup \mu(B)}{|L|} \)
3: if \( s \geq \theta \) then
4: if \( \mu(A) \cap \mu(B) = \emptyset \) then
5: \( M[A][s] \leftarrow M[A][s] \cup (\text{ExclChoice}(A,B), s, rs, rc^+) \)
6: else
7: \( M[A][s] \leftarrow M[A][s] \cup (\text{Choice}(A,B), s, rs, rc^+) \)
8: end if
9: end if
10: return \( s \geq \theta \)

11: function CHOICEEXCLGEN\(\Phi, C, \theta\)
12: \( \text{Considered} \leftarrow \{ (A,B) \in C \times C \mid A \sqsubseteq B, c(A,B) \notin \Phi \} \)
13: \( \mu \leftarrow [A \mapsto \{ i, \exists \sigma^j = \langle A, \delta \rangle \land \sigma^i \in L \} \}_{A \in I} \)
14: \( \mu^{-1} \leftarrow \{ i \mapsto \{ A \in I \mid \# \rho_A > 0 \} \}_{i \in L} \)
15: for all \( \langle A, B \rangle \in \text{Considered} \) do \( \text{SINGLECHOICEEXCLGEN}(A,B), \mu, \mu^{-1}, \theta) \)
16: for all \( \langle A, M_2 \rangle \in M \) do
17: if \( M_2[1.0] \neq \emptyset \) then
18: for all \( \langle c(A,B), s, rs, rc^+ \rangle \in M_2[1.0] \) s.t. \( \langle A, B \rangle \notin \text{Considered} \)
19: \( \text{Considered} \leftarrow \text{Considered} \cup \{ A, B \} \)
20: \( \Phi \leftarrow \Phi \cup \{ \langle c(A,B), s, rs, rc^+ \rangle \} \)
21: end for
22: else
23: for all \( \langle c(A,B), s, rs, rc^+ \rangle \in M_2[s] \) s.t. \( \langle A, B \rangle \notin \text{Considered} \land M_2[s] \neq \emptyset \) do
24: \( \text{Considered} \leftarrow \text{Considered} \cup \{ A, B \} \)
25: \( \Phi \leftarrow \Phi \cup \{ \langle c(A,B), s, rs, rc^+ \rangle \} \)
26: end for
27: end if
28: end for
29: return \( \Phi \)
the entire log in KnoBAB. For ADM+S, we upgrade the original Python codebase while keeping the original trace representation format, where each trace in the log is a dictionary that maps each activity label to the event ID where a given activity label occurs. For all candidates, we retain the θ frequent itemset support as well as declarative support SUPP, as our algorithm exploits SUPP for its lattice descent and, thus, tests all algorithms under the same conditions.

For algorithm validation, we repeat our experiments alongside state-of-the-art specification mining algorithms for temporal data. Additionally, we compare them with our prior algorithm, Bolt, thus showing how the enhanced temporal specification mining algorithm provides greater specification size reduction when compared to our previous heuristic solution (Section 5.1). After this, we provide loading and mining time benchmarks on datasets designed for Declare process mining and compact trace sizes (Section 5.2), as well as real data sequences containing considerable amounts of events within each trace (Section 5.3).

Our benchmarks were conducted on a Dell Precision mobile workstation 5760 running Ubuntu 22.04. The specifications of the machine include an Intel® Xeon(R) W-11955M CPU @ 2.60GHz × 16, 64GB DDR4 3200MHz RAM, with 500 GB of free disk space.

5.1. Algorithm Validation

To make a fair comparison, we benchmark Bolt2 as well as all competing approaches using the same dataset provided in our previous paper [5]. We use a synthetic dataset (https://osf.io/nsqcd/, accessed on 10 September 2023) generated through the ground-truth specification $\Phi^*$, described in the first column of Table 3, thus ensuring that all traces abide by the specification requested by $\Phi^*$ and that the whole resulting log will contain, at most, $|I| = 9$ activity labels. The resulting dataset is composed of 1000 traces containing 10 events each; we mine the clauses with minimum support of $\theta = 0.1$, thus obtaining a conjunctive specification $\Phi$ of all clauses with minimum declarative support of $\theta$. In $\Phi_{\geq 0.999}$, we keep the ones whose declarative support is at least 0.999 to mitigate the effect of inaccuracies from our previous implementations; this discards any infrequent behaviour not expressed in all traces. To better test the ability of the algorithm to generate compact specifications, no subsequent metric, such as the Fisher score to further prune the mined clauses, is considered.

Table 3. Mined specifications with frequent itemset support $\theta = 0.1$ on the same Synthetic Data, where all algorithms, except our previous Bolt, support ChainSuccession, Succession, Surround, and CoExistence. $\Phi^*$ denotes the expected specification.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ChainPrecedence(b, c)</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>ChainResponse(d, e)</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>Choice(f, g)</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>Exists(b, l)</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>Init(e)</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>Precedence(c, b)</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>RespExistence(h, i)</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>RespResponse(e, f)</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$</td>
<td>\Phi_2</td>
<td>$</td>
<td>36</td>
<td>197</td>
<td>46</td>
</tr>
<tr>
<td>$</td>
<td>\Phi_{\geq 0.999}$</td>
<td>148</td>
<td>46</td>
<td>40</td>
<td>120</td>
</tr>
<tr>
<td>Mining time (ms)</td>
<td>$5.95 \times 10^1$</td>
<td>$2.11 \times 10^0$</td>
<td>$2.03 \times 10^0$</td>
<td>$1.98 \times 10^2$</td>
<td>$3.55 \times 10^3$</td>
</tr>
</tbody>
</table>
Table 3 presents the results of the comparison; ticks (and crosses) denote the (missed) return of a declarative clause expected from the original specification, and “stronger” clauses with non-zero confidence are returned alongside expected clauses. Such results show that the current implementation (Bolt2) further improves the specification size reduction results from our previous heuristic search (Bolt) by slightly reducing the number of clauses in $\Phi_{\geq 0.999}$, while the size of the entire specification is a quarter of the one provided by Bolt. This also comes to a comparable mining time. This shows that the restrictions and additional tests described in Section 4.3 for reducing the specification size are effective and do not come at the cost of an increased running time. Furthermore, Bolt2 returns a Succession clause that is “stronger” than both Response and RespExistence; on the other hand, this clause is not currently supported by Bolt. Given our previous considerations, as well as the fact that our previous implementation is less expressive in terms of declarative templates being instantiated as clauses, we retain Bolt2 and discard Bolt in all following experiments.

While TopN Declare’s specification size is less than both Bolt and Bolt2, the former fails at providing most of the expected behaviour while also being less performant than Bolt. Due to this, we discard TopN Declare from all following benchmarks.

We now compare our best implementation, Bolt2, with the remaining two competitors, ADM and ADM+S. While the previous exploits pair candidate pruning using the Apriori algorithm, while avoiding any lattice descent algorithm by simply instantiating all possible pairs with all available declarative templates, the latter simply instantiates all declarative clauses over all possible pairs. This is shown by the fact that ADM+S returns a total number of clauses that is exactly $\text{UB}(I)$ (see Equation (7)), i.e., the maximum number of expected clauses that an algorithm can return. This is motivated by the fact that this algorithm still tests all the clauses and uses $\theta$ as a mere post-processing step for creaming off the final clauses. While the latter approach ensures the generation of all possible candidates to be tested, the former discards infrequent behaviour. In fact, satisfying an ExclChoice$(h,a)$ clause essentially demands the absence of any frequent pair $\langle h, a \rangle$, as the declarative pattern stipulates that $a$ and $h$ should not co-occur in the same trace. On the other hand, both Bolt and Bolt2 mitigate this problem by generating pairs for (Excl)Choice clauses, merging frequent unary patterns that infrequently occur from the data. Furthermore, we show that both competitors return very large specifications with many uncertain behaviours, while our lattice descent algorithm, along with the adoption of a quality criterion, ensures the return of the most general clause providing the best support. This allows the reduction of the overall specification size, to become as close as possible to the number of expected clauses with perfect support. On the other hand, competing approaches do not exploit this principle within the specification mining algorithm, thus returning an explosive amount of clauses. This is also reflected in the number of clauses with perfect support, where the considerable size can be explained in terms of the provision of both implying and being-implied clauses. This adversely affects the overall running time, which is substantially prolonged due to unnecessary candidate testing. This delay can be mitigated through early stopping conditions, such as heuristic computations, as well as leveraging efficient lattice descent to avoid testing clauses whose support is anticipated to be lower than the ones currently under consideration.

By comparing these results with the ones where ChainSuccession, Succession, and Surround are not included among the Declare templates to be tested, ADM generates 512 clauses, of which, 119 have perfect support in $1.22 \times 10^2$ ms, while ADM+S generates 612 clauses, of which, 450 have exact support in $1.67 \times 10^2$ ms. The decreased running time in comparison with the ones provided in Table 3 clearly shows that an increased amount of clauses to be tested leads to an increase in running time. Furthermore, while the running time increase for ADM is almost negligible, the running time for ADM+S is almost tripled; this is in line with our previous observations [2], where an optimised query plan eases the running time for formal verification by avoiding recomputing multiple sub-expressions to be tested in the mining phase. This cannot be leveraged by the ADM+S pipeline, as each clause has to be computed separately. This consideration is further leveraged in the implementation of
Bolt2, where some clauses are also tested within the same for loop, despite not formally sharing any \( \text{LTL}_f \) sub-expression similar to Algorithms 9 and 10.

This final comparison strengthens our previous considerations in [5], showing how the curse of propositionalisation affects both the running time and the specification size by leading to an increase in mined clauses and the overall running time.

5.2. Mining and Loading Times for Process Mining Datasets

We now want to test the execution times of the mining algorithms under the same boundary conditions assumed by the original authors of these state-of-the-art algorithms, i.e., a rather limited total number of distinct activity labels (at most, in the order of 50). We restrict our analysis to Bolt2, ADM, and ADM+S. We also consider some datasets from our previous paper (https://osf.io/nsqcd/, accessed on 10 September 2023), where we considered both a synthetic dataset generated from the same declarative specification (i.e., model) \( \Phi^* \), and a real-world process mining dataset that described the purchase order handling process for some of the 60 subsidiaries part of a large multinational company (BPIC2019 [34]). To conduct mining benchmarks, while considering the log size as a dimension of our analysis, we sample both datasets, generating three and five sub-datasets, respectively. This downsizing strategy generates some possible subsets \( \mathcal{S}^j_i \in \wp(\mathcal{S}) \) of \( \mathcal{S} \) of increasing size \( |\mathcal{S}^j_i| = 10^i \):

\[ \forall 1 \leq i \leq (N - 1), \quad \mathcal{S}^j_i \subseteq \mathcal{S}^j_{i+1} \quad \text{where} \quad N = \lceil \log_{10} |\mathcal{S}| \rceil \]

We start our analysis by describing a controlled experiment from the synthetic dataset, where we set up both the maximum log size and ensure that all traces have the same length. In particular, we return logs of 1000 traces, where trace lengths are in \( \zeta \in \{10, 15, 20, 25, 30\} \). We run the algorithms on different log sizes \( \mathcal{S}^j_i \) (x-axis) and trace lengths \( \zeta \) (symbol shape and line type) while keeping a minimum support threshold of \( \theta = 0.9 \). Figure 7a shows the running time for loading and indexing data for Bolt2, ADM is implemented on top of KnoBAB, and ADM+S. As we can see, both Bolt2 and ADM exhibit similar loading times as they both use KnoBAB as a knowledge base for storing temporal data while also exploiting the same physical representations for both datasets. Despite ADM+S not relying on sophisticated data representations, we observe that this is slower than loading data when compared to KnoBAB by at least one order of magnitude. After further investigation, we find that the main issue is related to the Python library for parsing XES data and OpyenXES (https://github.com/OpyenXES/OpyenXES, accessed on 10 September 2023), while KnoBAB relies on a customary parser and lexer generated by ANTLR4 (https://github.com/antlr/antlr4, accessed on 10 September 2023) [35] from the given grammar in the Backus–Naur form. By comparing this plot with the mining times in Figure 7b, we note that Bolt2′s data loading time dominates its mining time, while for the competing approaches, we can provide the opposite consideration.

Concerning the mining times in Figure 7b, our proposed solution is, at most, three orders of magnitude more efficient than our slowest competitor (ADM+S over \(|\mathcal{S}| = 10^1 \) and \( \zeta = 10 \)) and at least one-and-a-half orders of magnitude more efficient than our fastest competitor (ADM over \(|\mathcal{S}| = 10^3 \) and \( \zeta = 10 \)). All algorithms exhibit polynomial time complexity with respect to the log size, mainly because the log (in the worst-case scenario) needs to be accessed to test each binary itemset, leading to the generation of a declarative clause. ADM+S, while able to prune potential candidates, tests each declarative clause per trace; furthermore, it cannot leverage efficient data scanning operations via an indexed relational database. The main difference comes from large additive constants that heavily affect the computation time. Such experiments show that our previous solution to lower the testing times of declarative clauses against a log via query plans [2] can only be lapped by ad hoc heuristics further pruning redundant or infrequent clauses, as well as promptly yielding, clauses without the need for “intermediate results”, leading to allocation overhead.
We now discuss our experiments on the BPIC2019 dataset. Figure 8a shows the overall trace length distribution for each sample. We can observe that the logs contain fairly short traces; there are few traces with a length of $10^3$, and the majority of traces have, at most, 100 events. The biggest sample ($|S_4| = 10^4$) contains, at most, 41 distinct activity labels, requiring testing of, at most, $UB(I) = 18.204$ clauses, while the smallest ($|S_1| = 10$) contains 10 distinct activity labels, where we can test, at most, $UB(I) = 1030$ clauses. The resulting data are also provided (https://osf.io/qsqcd/, accessed on 10 September 2023).
in the aforementioned dataset. For this, we also consider variations of the minimum support threshold ($\theta \in \{0.1, 0.25, 0.5, 0.9\}$) as customary for mining algorithms benchmarks. Data loading times in Figure 8b show similar data loading trends to those presented in our previous experimental setting, where loading times across OpyenXES and KnoBAB solutions mainly differed by one order of magnitude. These results do not consider $\theta$ thresholds, as this only affects the mining algorithm. Mining times in Figure 8c show that all algorithms exploiting the minimum support threshold for candidate pruning through a frequent itemset algorithm (ADM and Bolt2) exhibit an inverse correlation between the minimum support threshold $\theta$ and mining time, as a lesser mining time requires the identification of a great number of items leading to the subsequent generation of declarative clauses. The experiments clearly show that ADM+S exploits $\theta$ only as a post-processing feature by filtering out all the clauses appearing as infrequent after generating all the possible patterns, as $\theta$ does not affect the running time at all. Despite ADM+S exhibiting a linear time complexity, its running time consistently upper-bounds the running times of the two other competing algorithms exhibiting polynomial time complexities; among these, Bolt2 consistently provides the fastest running time, being two orders of magnitude less than the slowest competitor (ADM).

Figure 8. Cont.
Figure 8. Benchmarks on the BPIC2019 dataset. (a) Trace length PDFs on the different samples. Colours refer to each specific sample size $|L|$. (b) Loading time in ms. Colours show the algorithm names. (c) Mining time in ms. Colours show the algorithm names.

For datasets with $|S^x| > 10^4$, ADM+S exceeds the 64GB of available primary memory and fails without yielding results, after running for about 3 h and 15 min. This shows the inadequacy of ADM+S when testing several clauses from relatively compact sizes of activity labels. ADM+S utilises the pandas Python library for data-frame representation, where combinations of candidates are represented in a sparse matrix to be more efficiently held in memory and reduce the overall memory footprint. Even so, the memory required to store the embedding information is insufficient.

5.3. Real-World Datasets

We now test our algorithm alongside competing approaches at their boundaries, by increasing the total number of distinct activity labels as well as the total trace lengths. Both parameters, in the worst-case scenario, lead to an increase in the number of clauses to be tested. The former is directly dependent on the size of $I$ and the frequent patterns that can be mined from the data due to $UB$, while the latter will require a longer time to effectively test all the clauses being considered. Usual business process management datasets contain restricted amounts of distinct activity labels, while temporal datasets from the artificial intelligence field, such as the ones for cybersecurity, contain larger varieties of distinct possible activities $I$. GNU/Linux systems can have between 300 and 500 system calls, requiring the testing of just under $3 \times 10^6$ clauses, while other operative systems, such as Windows, provide 2000 possible system calls (https://www.knowledgehut.com/blog/web-development/system-calls-in-os (last accessed: 14 March 2023)), requiring the testing of several clauses in the order of $5 \times 10^7$. Considering that cybersecurity datasets often track and audit a significantly larger number of operations compared to typical business process scenarios, it becomes evident that testing and generating all possible clauses in a reasonable timeframe is nearly unfeasible. This further motivates our approach advocating for testing only the clauses that are considered the most representative and are supported by existing data.

For this, we consider a cybersecurity dataset [18] that represents malware in terms of a sequence of Windows operating system API calls being invoked. The authors distinguish between eight main types of malware: trojan, backdoor, downloader, worm, spyware, adware, dropper, and virus. Each trace is then associated with a specific label identifying the specific class to which the malware belongs. The authors represent the dataset as a tab-separated file. The data loading phase in KnoBAB potentially supports the loading...
of logs serialized in multiple formats, including the tab-separated (TAB, .tab) format for this dataset and the de facto log standard extensible event stream (XES, .xes) used for the previous datasets. We use different data parsers, which are still linked to the same data loading primitives; thus, the main differences in loading times are related to the parsing times, and not to the loading and indexing times. For this reason, we discard the data loading analysis for this dataset.

Even in this scenario, we sample (https://osf.io/69q8h/, accessed on 10 September 2023) the original dataset in different logs. We consider logs with increasing sizes of $|S_i| = 9^i$. Figure 9a represents the trace length distribution for each sampled dataset. Different from the previous ones, this exhibits longer traces that appear more frequently for each subsample, as the longest trace contains more than $10^6$ events; while sampling the logs, we also attempt to guarantee that each log has a similar trace length distribution despite the skewed trace length distribution. The biggest sample ($|S_4| = 6552$) contains 278 distinct activity labels, while the smallest ($|S_1| = 9$) contains 132 distinct activity labels. Therefore, the data pose major challenges for both ADM and ADM+S algorithms, which heavily rely on propositionalisation approaches to generate all of the possible declarative candidates for the given candidate itemsets of choice. We run the following algorithms with a minimum support threshold of $\theta = 0.9$.

By comparing the mining times from Figure 9b with our previous experiments, we observe that the combined provision of longer traces and longer traces overall affects the running times of both ADM and ADM+S, as they are now at least four orders of magnitude less efficient than our mining algorithm, Bolt2. Despite our implementation of ADM being exploited, the optimised conformance-checking algorithm for formal verification, as discussed in [2], makes this implementation more competitive than ADM+S in terms of running time; both implementations run out of memory in the third sample ($|S_3| = 9^3$), showing the inadequacy of directly exploiting current formal temporal verification algorithms, such as conformance checking for specification mining. For ADM, this is mainly due to the amount of intermediate results associated with multiple events being considered, while for the latter, this can be ascribed to the wasteful memory representation provided by pandas. Concerning Bolt2, the running time plateau between the two last samples can be partially ascribed to a lesser variation in the number of distinct activity labels (258 vs. 278) and partially to the effectiveness of the candidate-pruning algorithm, as well as the early stopping conditions, avoiding testing unnecessary clauses that are not representative of the data. This last stress test shows the effectiveness of the envisioned heuristic search algorithm for dealing with big data scenarios when compared to competing solutions, as our mining algorithm consistently takes less than one second of mining time when compared to existing solutions that take more than 15 min to compute before running out of memory.

We now conduct a complete series of benchmarks over Bolt2, where we vary the log size and the minimum support threshold to empirically assess the properties of our novel proposed approach. Figure 10a,b show that Bolt2 is not significantly affected by the variations of log and minimum support threshold sizes, as in all such circumstances, our algorithm mines a specification without running out of memory (while doing so in a reasonable amount of time). In Figure 10a, we observe that the aforementioned running time plateau is only observable for $\theta \in \{0.9, 0.5\}$, while this effect is less evident for lower minimum support thresholds. This shows the effectiveness of the early stopping conditions, avoiding testing for further clauses through the provided heuristics; it clearly shows that no further temporal behaviour is actually tested. For the remaining minimum support values, we register a decrease in the running time as it deviates from the initial polynomial time trend.

By correlating these results with the size of the mined specification reported in Figure 10c, we observe that these running times are not necessarily correlated with a decreased amount of clauses being collected (please refer to the Pearson correlation coefficient in the Figure). In fact, for $\theta = 0.9$, we can still observe an increase in running time.
per log size, which is then reflected in an overall decrease in the number of clauses being returned, thus showing that the running time has a stronger correlation with the size of the log while having the strongest inverse correlation with the minimum support threshold, as per expectations. However, the constant decline in the number of clauses returned, despite the use of logs providing a constant increase in \(|I|\), should not arouse particular suspicions in the reader. On the contrary, these highlight the fact that, for sufficiently high support values, we attempt to obtain clauses that maximise trace coverage. This tendency, especially when considering traces with significantly varied behaviour, often leads to the exclusion of numerous clauses not supported across all traces.

**Figure 9.** Benchmarks on the cybersecurity dataset. (a) Trace length PDFs of the different samples. Colours refer to each specific sample size \(|L|\). (b) Mining time in ms. Colours show the algorithm names.
These results suggest to practitioners that to retrieve a larger number of clauses supporting the majority of instances within a log, one could partition the aforementioned log into sub-logs. By then running Bolt2 for each of these sub-logs, we can merge all the clauses mined for each bin in a single conjunctive specification. This is still a viable solution, as the presented algorithm is quite efficient even for considerably small values of the minimum support value.

![Figure 10. Cont.](image-url)
6. Discussion

In the subsequent section, following our discussions on the reasons why Bolt2 can be considered as faster than competing approaches, this second-last section discusses some of the assumptions that we adopted in the ideation of our specification mining algorithm (see Sections 6.1 and 6.2); we provide disambiguations between similar concepts that might be easily confused (see Sections 6.3 and 6.4). Finally, we postulate some future works, addressing the current limitations of deviant model learning algorithms and discussing how we aim to overcome these limitations within our present framework (Section 6.5).

6.1. Non-Zero Confidence and Activated Traces

Observe that our lattice described in Figure 2 works if and only if we consider that all the clauses tested or returned by the algorithm anytime are always activated at least once in our dataset. This is ensured from the very beginning by the adoption of association rule mining, as the provision of an \( X \rightarrow Y \) rule entails the presence of an event \( X \) in at least one trace, evaluating these rules by their confidence scores. This ensures that \( \text{RespExistence} \) (or \( \text{CoExistence} \)) is non-vacuously satisfied by at least one trace in the log.

\( \text{RespExistence}(A, B) \) does not generally imply \( \text{Choice}(A, B) \). In particular, the only case where \( \text{RespExistence}(A, B) \) is satisfied but \( \text{Choice}(A, B) \) is not is when neither \( A \) nor \( B \) are in the trace. But, if we return \( \text{RespExistence}(A, B) \), as per our mining algorithm, it is because \( A \) appears in the trace at least once, and that is because we are generating such clauses from the frequent itemsets through the confidence measure as discussed in the previous paragraph. So, in these scenarios, returning a \( \text{RespExistence}(A, B) \) from a mined rule \( A \rightarrow B \).
entails the satisfaction of Choice(A, B) by at least one trace in the specification. This can be shown as follows:

\[
c(A \rightarrow B) > 0 \iff \frac{\mathcal{C}(A \cup B)}{\mathcal{C}(A)} > 0
\]

\[
\iff \zeta'(A) > 0 \land \zeta'(A \cup B) > 0
\]

\[
\iff \{\sigma^i \in \mathcal{S}|\#_\sigma^i A > 0\} \neq \emptyset \land \{\sigma^i \in \mathcal{S}|\#_\sigma^i A > 0 \land \#_\sigma^i B > 0\} \neq \emptyset
\]

\[
\Rightarrow \{\sigma^i \in \mathcal{S}|\#_\sigma^i A > 0 \lor \#_\sigma^i B > 0\} \neq \emptyset
\]

\[
\Rightarrow \{\sigma^i \in \mathcal{S}|\sigma^i = \text{Choice}(A, B)\} \neq \emptyset
\]

\[
\Rightarrow \text{SUPP}_\phi(\text{Choice}(A, B)) > 0
\]

When returning a declarative clause candidate, our quality criterion (Definition 3) enforces this by ensuring that clauses non-vacuously satisfied by at least one trace are considered as candidates for our specification.

### 6.2. SUPP\(_\phi\) vs. RSUPP\(_\phi\)

During our benchmarks and tests, we found that the choice of support metric (Section 2.3.1), either SUPP\(_\phi\) or RSUPP\(_\phi\), has a significant impact on the lattice search. The former allows for vacuously satisfying traces, while the latter will only consider those containing activation conditions.

**First**, given that activations may vary on the lattice descent (due to Precedence), the support of a descendant clause may be greater than its ancestors.

**Example 11.** Consider the following log and clauses:

\[
\mathcal{G} = \{\sigma^1, \sigma^2, \sigma^3\} \quad \sigma^1 = CD \quad \sigma^2 = AB \quad \sigma^3 = A
\]

In the log above, we obtain \(B \rightarrow A\) as a frequent association rule with certain confidence entailing the Declare clause \(\phi = \text{RespExistence}(B, A), \text{SUPP}_\phi(\phi) = 1.0\) as per a previous discussion. Upon refinement of the former declarative clause, we might generate a \(\phi' = \text{Precedence}(A, B)\), with the same support as the former, as all traces are either activated and satisfied, albeit vacuously for the latter in the case of \(\sigma_3\). These two clauses differ in their activation conditions, as the former is activated by any occurrence of a \(B\) while the latter triggers the checks upon detecting an activation \(A\). Within the same example, \(\sigma_3\) only activates (and satisfies) \(\text{Precedence}(A, B)\), but not \(\text{RespExistence}(B, A)\). This leads to considerably different restrictive supports for the two clauses, i.e., RSUPP\(_\phi\)(\(\phi\)) = 0.33 and RSUPP\(_\phi\)(\(\phi'\)) = 0.66.

We can see that if we pruned our clauses using restrictive support as quality criterion for the general-to-specific lattice visit with a minimum threshold of 0.6, we might not have been able to generate any of its descendants with greater restrictive support, as the lattice descent would have been stopped at the coarser clause with lesser support. This is an undesired behaviour violating the anti-monotonicity requirement of a good quality criterion. This postulates the adoption of our previous notion of SUPP over RSUPP, as the former can be used as an early stopping condition to avoid returning clauses that are underrepresented in a log due to the anti-monotonicity (Lemma A4). As RSUPP still enables greater discriminative and pruning power while incorporating the requirement of having a support strictly greater than zero while ensuring at least one activation is available, future works will find new ways to leverage this novel notion of support with a novel algorithmic approach.

**Second**, the relaxed notion of support avoids an overabundance of atemporal and “acorrelational” behaviour (e.g., (Excl)Choice), not allowing adequate temporal discrimination across traces. This is a desired behaviour, as our previous experiment on cybersecurity data showed that this often occurs within the data.
Example 12. Let us now consider the following log:

\[ \mathcal{G} = \{ \sigma^1, \sigma^2, \sigma^3 \} \quad \sigma^1 = BA, \sigma^2 = AB, \sigma^3 = B. \]

By considering \( \text{SUPP}_{\mathcal{G}} \) for our lattice descent, we never obtain a \( \text{Choice}(A, B) = \text{Choice}(B, A) \), as \( \text{RespExistence}(B, A) \) has a \( \text{SUPP}_{\mathcal{G}} \) of 1.0 due to \( \sigma^3 \). Alternatively, by considering \( \text{RSUPP}_{\mathcal{G}} \), we now obtain a \( \text{Choice} \) instead, as the latter clause is not activated by \( \sigma_3 \), reducing the \( \text{RSUPP}_{\mathcal{G}} \) to \( \frac{2}{3} \). This then guarantees the return of temporal or association-based behaviour when possible.

Future works will analyse whether returning additional Choice clauses will actually benefit a better classification task, thus further motivating the adoption of \( \text{RSUPP} \) in lieu of \( \text{SUPP} \).

6.3. Activation and Target vs. Heads and Tails in Association Rules

Our heuristic for associating a declarative rule to a mined association rule can be summarised by Equation (9), stripping any temporal information from the LTL\(_f\) semantics of a declarative clause while retaining only the activation and target conditions of reference. This works under the assumption that an implication \( \Rightarrow \) in LTL\(_f\) is the main separator between the activation condition, appearing in the premise of the rule, and the target one, appearing as a consequence of the implication. Concerning \( U \), we consider the detection of the second argument of the operator as an activation, thus interpreting the first argument as a target condition, as all the remaining events preceding the activation must satisfy this condition. We can easily observe that \( \text{CoExistence}(A, B) \) will then return both \( A \rightarrow B \) and \( B \rightarrow A \), while \( \text{Choice}(A, B) \) returns \( \{A, B\} \), which can also be interpreted as the former.

\[
\text{ERASURE}_{A, B}(\phi) = \begin{cases} 
T & \text{TC}(A, B) \\
\text{ERASURE}_{A, B}(\phi') \rightarrow \text{ERASURE}_{A, B}(\phi'') & (\phi' \rightarrow \phi'' \lor \phi' \rightarrow \phi''') \land \\
\text{ERASURE}_{A, B}(\phi') \land \text{ERASURE}_{A, B}(\phi'') & (\phi' \rightarrow \phi'' \lor \phi' \rightarrow \phi''') \\
\text{ERASURE}_{A, B}(\phi') & \phi' \rightarrow \phi'' \lor \phi' \rightarrow \phi''' \\
\emptyset & \text{oth.}
\end{cases}
\tag{9}
\]

Consider that this formulation completely discards negated information, as the negation of a condition cannot be straightforwardly translated into a specific activity label of interest. This includes the interpretation of \( \text{Precedence}(A, B) \), which provides no association rule associated with its LTL\(_f\) semantics. For this, we cannot consider \( A \) as a potential activation condition, as the semantics of this rule stipulate that the occurrence of \( A \) does not strictly require a future occurrence of \( B \) as in the Response. On the other hand, we can observe that, when a \( B \) occurs, we need to check that such \( B \)s shall always occur after \( A \)-s, but not before those. Due to this, we can associate such a clause to an association rule \( B \rightarrow A \). This differs from our previous implementation [5], where we associated such a clause to \( A \rightarrow B \) instead. The following example further contextualises the intuition for this association rule:

Example 13. Let us consider the following log:

\[ \mathcal{G} = \{ \sigma^1, \sigma^2, \sigma^3 \} \quad \sigma^1 = A, \sigma^2 = AA, \sigma^3 = BC \]

A specification mining algorithm exploiting an exhaustive search, such as \( \text{ADM+S} \), might generate \( \text{Exists}(A, 1) \), as well as \( \text{Precedence}(A, B) \), as the first two traces trivially satisfy this condition, and no \( B \) appears in the data. Nevertheless, the data do not provide sufficient evidence on how to interpret the occurrence of \( B \) alongside \( A \), as the traces where \( A \)s occur solely contain \( A \)s.
The peculiar case of Precedence stresses the fact that the notion of the activation/target condition cannot be straightforwardly mapped to a notion of the head or tail of an association rule, as the presence of a negation heavily changes the game. Future works will provide a better formulation of our ERASURE function to encompass negated LTL formulae; this will further assist in the mining of additional negated declarative clauses [2].

6.4. ChainPrecedence: $\alpha$-Renaming

Our ChainPrecedence($A$, $B$) has undergone $\alpha$-renaming; previous declarative literature studies denote the semantics of the former as follows: $B$ is immediately preceded by $A$ [36], with the LTL$_f$ formalisation as $\Box(\Diamond B \Rightarrow A)$. Our definition, after $\alpha$-renaming, is $A$ immediately preceded by $B$ ($\Box(\Diamond A \Rightarrow B)$). This is mainly to provide clarity to the non-practitioner by always representing $A$ as the activation condition and $B$ as the target condition. To maintain consistency among the activation labels and the semantics, our $\alpha$-renaming flips the semantic labels, such that for an association $A \rightarrow B$, we mine the clause ChainPrecedence($A$, $B$), with the LTL$_f$ formalization flipped as $\Box(\Diamond A \Rightarrow B)$.

6.5. Limitations of Current Deviant Model Learning Algorithms

Our previous experiments showed that both ADM implemented in KnoBAB as well as ADM+S implemented in Python are severely affected by exceptionally long traces in a log, as the mining tasks are both time-expensive and run out of memory for larger logs. This discards any further attempt to use pandas and scikit-learn in Python for mining and learning deviant models from real data, thus requiring attempts to re-implement this pipeline from scratch in C++. In the long run, this language is proven to be better when ensuring memory-saving strategies.

Even by ensuring this, exploiting just one decision tree for exporting either explanations or deviant features from the different classes, as in [21], might provide specifications that are extremely biased toward a single candidate providing perfect purity. Observe that this heuristic assumption will likely create specifications that overfit the training data, as the testing data might, in the worst-case scenario, not satisfy such a clause. This, on real data not necessarily biased toward declarative temporal representations as the datasets of choice in [21], might lead to specifications with low precision. Our future works will attempt to mitigate this potential overfitting risk by exploiting ensemble methods that guarantee the generation of multiple decision trees that can then be combined as one single explainable specification, such as random forests or their variants.

Future works will attempt to tackle these limitations, requiring improvements in our formal verification pipeline [2], as we might still need to test clauses appearing across different logs, where distinct clauses might appear in distinct logs. The decision trees can return conjunctive specifications, as outlined in [21], as well as generate specifications in disjunctive normal form; we will assess the benefits of defining novel specification mining algorithms that return specifications in disjunctive normal form.

7. Conclusions and Future Works

The present paper proposes a specification mining algorithm for temporal data, Bolt2, improving our previous algorithm Bolt [5] specification (i.e., model) size reduction while retaining similar running times. Benchmarks show that Bolt2 outperforms competing approaches in terms of the size of the specification being returned in the data loading and mining times by at least one order of magnitude, and in the best-case scenario, in real temporal data, by four/five orders of magnitude. These improvements can be ascribed to the heuristics of choice for navigating the hypothesis/clause search spaces that provide early stopping conditions, while guaranteeing a compact number of clauses via a quality criterion, ensuring that clause coverage does not decrease during a lattice descent. This advances the field of specification mining, as for the first time in this field, we introduce efficient lattice search algorithms, which were mainly known in data mining and relational learning literature. Notwithstanding the aforementioned, a controlled experiment
demonstrates that our algorithm effectively captures all the relevant and expected temporal features present in synthetic datasets, while competing approaches are significantly impeded by testing a vast number of clauses without early stopping conditions across the entire log, thereby failing to bypass clauses for traces that do not meet specific temporal requirements.

Future works will consider returning specifications expressed in disjunctive normal forms as well as specification mining algorithms that consider more than one specification at once from the data. To achieve this, we need to enhance our previous formal verification pipeline further to assess the validity of mined clauses across different logs efficiently. We will also consider extending the refinement operator to extend dataless clauses with ones that support data predicates. After achieving this, we will then conduct further experiments to test whether the S-border visit and the early pruning of infrequent behaviour will lead to the extraction of several relevant features to be used in deviant model (i.e., specification) learning algorithms. Our previous results on Bolt ensure that we might still be able to increase the recall with a negligible loss of execution time, which could worsen as the distinct activities contained in the data increase. Subsequent versions must, therefore, consider increasing recall while maintaining a reasonable execution time. Preliminary considerations in this paper might suggest that this can be trivially achieved by running the algorithm on multiple partitions of the same log (Section 5.3). Experiments over ADM show that our current implementation of the $xt\text{LTL}_f$ operators can be further improved; this will be crucial for testing mined clauses across distinct logs, as required by deviant model learning algorithms. This will then come to the further benefit of existing temporal formal verification techniques in KnoBAB [2].

Our future extensions of this algorithm will also consider the adoption of further Declare clauses that were not considered at this current stage, such as AltPrecedence, AltResponse, AltSuccession, and some other negative clauses. Regarding the first three clauses, these will still require a complete rewriting of the algorithm to hardcode the testing algorithm, including the scanning of the column database for seeking temporal correlations between activation and target conditions, as well as hardcoding further $Q^b$ test conditions to determine the inclusion of the tested clauses. This is strictly required to achieve good optimisation, as the current clause-rewriting solutions via $xt\text{LTL}_f$ cannot subsume the presented early stopping conditions while embedding clause testing containing disparate $xt\text{LTL}_f$ operators, such as Precedence and Response. To automate this approach, future works will assess the possibility of further atomising the $xt\text{LTL}_f$ operators [2] to combine the iteration over activation and target conditions that do not necessarily share the same $xt\text{LTL}_f$ operators. Achieving this will also enable us to define a generic general-to-specific mining algorithm that might be completely independent of the intended hypothesis language of choice. Concerning the negative clauses, we will consider the possibility of generating these from the ones discarded by the specification as they had low support.

Finally, we are also planning to further extend our dataless mining algorithm in order to support data conditions. Preliminary observations also show that achieving this will require exploiting our formal verification pipeline to extract activation/target payloads; this, as per previous observations, shows a need to further optimise this pipeline to handle real-data use case scenarios more efficiently. We will also ponder the possibility of replacing FPGrowth with more up-to-date mining algorithms, such as Q-Eclat [37].

**Author Contributions:** Conceptualization, G.B.; methodology, G.B.; software, S.A., G.B.; validation, S.A.; formal analysis, G.B.; investigation, S.A.; resources, G.B. and G.M.; data curation, G.B.; writing—original draft preparation, S.A. and G.B.; writing—review and editing, G.B. and G.M.; visualization, G.B.; supervision, G.B. and G.M.; project administration, G.B.; funding acquisition, G.B. All authors have read and agreed to the published version of the manuscript.

**Funding:** Samuel Appleby’s work is supported by Newcastle University.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.
Data Availability Statement: The dataset associated with the presented experiments is available online at https://osf.io/nsqcd/ and https://osf.io/69q8h/ (accessed on 10 September 2023).

Conflicts of Interest: The authors declare no conflict of interest.

Sample Availability: The most up-to-date version of KnoBAB is available on GitHub: https://github.com/datagram-db/knobab (accessed on 10 September 2023).

Abbreviations
The following abbreviations are used in this manuscript:

- DAG: direct acyclic graph
- KnoBAB: knowledge base for alignments and business process modelling
- LTL\text{f}: linear temporal logic over finite traces
- RDBMS: Relational Database Management System
- XES: extensible event stream
- x\text{t}LTL\text{f}: extended linear temporal logic over finite traces

Appendix A
This section focuses on the lemmas pertaining to the foundations of data mining algorithms.

Lemma A1. Given two hypotheses, $h_1 := 1_{B_1}$ and $h_2 := 1_{B_2}$, for binary decision problems, we have that $h_2 \preceq h_1$ if and only if $B_1 \subseteq B_2$.

Proof. For any subset $S \subseteq T$ of $T$, if we assume that the intersection $S \cap T$ is equal to $S$ (H), then we can provide the following proof:

\[
\begin{align*}
  h_2 \preceq h_1 & \iff \operatorname{co}(h_1, D) \subseteq \operatorname{co}(h_2, D) \\
                      & \iff (B_1 \cap D) \subseteq (B_2 \cap D) \\
                      & \iff (B_1 \cap D) \cap (B_2 \cap D) = (B_1 \cap D) \\
                      & \iff (B_1 \cap B_2) \cap D = B_1 \cap D \\
                      & \iff B_1 \cap B_2 = B_1 \\
                      & \iff B_1 \subseteq B_2
\end{align*}
\]

Lemma A2. The minimum frequency quality criterion is anti-monotonic with respect to $D$.

Proof. By Lemma A1, we rewrite $h_1 \preceq h_2$ as $B_2 \subseteq B_1$ (H1). Also, the min-frequency quality criterion for $h_2$ becomes $|B_2 \cap D| \geq \theta$, implying that we know a real number $x \in \mathbb{R}^+$ for which $|B_2 \cap D| = \theta + x$ (H2). If we now expand the min-frequency definition for $h_1$, we now have to prove that $|B_1 \cap D| \geq \theta$ also holds. By expanding H1, we know a set $S$ for which $|(B_2 \cup S) \cap D| = |(B_2 \cap D) \cup (S \cap D)| \geq \theta$. We can now prove

\[
\begin{align*}
  |(B_2 \cup S) \cap D| = |(B_2 \cap D) \cup (S \cap D)| \geq \theta & \iff \exists y \in \mathbb{R}^+ . |B_2 \cap D| + y = \theta \\
                                                                     & \iff \exists y \in \mathbb{R}^+ . \theta + x + y = \theta \\
                                                                     & \iff \theta \geq \theta
\end{align*}
\]

Lemma A3. If $\phi'$ is more general than $\phi$, then $\phi$ implies $\phi'$ by considering each LTL\text{f} formula as a hypothesis, in which coverage is defined as the set of traces in a log satisfying such a formula.
Proof.
\[\phi' \preceq^b \phi \iff \text{co}(\phi, \Sigma) \subseteq \text{co}(\phi', \Sigma) \iff \{\sigma \in \Sigma | \sigma \models \phi\} \subseteq \{\sigma \in \Sigma | \sigma \models \phi'\} \iff \phi \Rightarrow \phi'\]

□

**Lemma A4.** If clause \(c_2\) is more general than \(c_1\) \((c_2 \preceq^b c_1)\), then the support of the latter is lesser or equal to the support of the former.

**Proof.**
\[
c_2 \preceq^b c_1 \iff \{\sigma \in \Sigma | \sigma \models c_1\} \subseteq \{\sigma \in \Sigma | \sigma \models c_2\}
\]
\[
\Rightarrow \frac{|\{\sigma \in \Sigma | \sigma \models c_1\}|}{|\Sigma|} \leq \frac{|\{\sigma \in \Sigma | \sigma \models c_2\}|}{|\Sigma|}
\]
\[
\iff \text{SUPP}(c_1) \leq \text{SUPP}(c_2)
\]

□

**Corollary A1.** If clause \(c_1\) is a refinement of \(c_2\), then its support is less or equal to the support of \(c_2\).

**Proof.** By definition of \(\rho\), \(c_2 \preceq^b c_1\). Therefore, from Lemma A4, it derives that \(\text{SUPP}(c_1) \leq \text{SUPP}(c_2)\). □

**Corollary A2.** Our quality metric for Bolt2 can be rewritten by stating that the support of the clause of interest shall be equal to each one of its direct ancestors for which it is a direct refinement.

**Proof.** As per Corollary A1, we have that \(h \in \rho(h')\) entails \(\text{SUPP}(h) \leq \text{SUPP}(h')\). As our definition of the metric requires that \(\text{SUPP}(h') \leq \text{SUPP}(h'')\) for each \(h''\), then by antisymmetry, we have \(\text{SUPP}(h') = \text{SUPP}(h)\). Then, the quality criterion can be rewritten as follows:
\[
Q^b(c, \Sigma) \iff (\forall c'. c \in \rho(c') \Rightarrow \text{SUPP}(c') = \text{SUPP}(c)) \land \text{SUPP}(c) \geq \theta \land \text{RCONF}_{\Sigma}(c) > 0
\]

□

**Appendix B**

This section focuses on the lemmas regarding the expressiveness of each single clause.

**Lemma A5.** Clauses \(\text{ChainSuccession}(A, B)\) and \(\text{Surround}(A, B)\) do not entail each other.

**Proof.** We can state that two declarative clauses are equivalent if they satisfy the same set of traces; thus, showing that two clauses are not equivalent requires finding a trace that is satisfied by the former but not by the latter, and vice versa. Therefore, showing their inequality requires finding only one trace distinguishing their temporal behaviour. For example, consider the following traces:

\[\sigma^1 = ABABB \quad \sigma^2 = BB\]

First, we can show that there exists a trace satisfied by a Surround that violates a ChainSuccession. We can observe that \(\sigma^1\) satisfies Surround\((A, B)\) as the first \(A\) is not activated by \(\square(\Box A \Rightarrow B)\) but not ChainSuccession\((A, B)\). This is due to the temporal requirement \(\square(\Box B \Rightarrow A)\) stemming from ChainPrecedence\((B, A)\) of ChainSuccession, which is violated by the last \(B\) occurring in the trace. Last, we can show that there exists a trace satisfying the ChainSuccession violating a Surround. \(\sigma^2\) demonstrates this; \(A/\Box A\) is never activated for Surround\((A, B)\), which is vacuously satisfied, while \(\square(\Box B \Rightarrow A)\) is activated but violated for ChainSuccession\((A, B)\). □
Appendix C

This section provides some lemmas motivating the correctness of our heuristics.

Proof (Lemma 1). Satisfaction of ChainSuccession(A,B) requires that both ChainPrecedence(B,A) and ChainResponse(A,B) must be satisfied. If both defining clauses are not vacuously satisfied, then A and B must occur in equal numbers, as all the B-s occurring in the trace must be located after the first event due to □B while A-s might occur freely. We observe that having A-s and B-s in equal numbers is not a sufficient condition for a ChainSuccession to occur, as there might be an arbitrary number of such events within a given trace without them necessarily being ordered in the trace as dictated by the aforementioned clause. Still, the violation of such a condition indicates the violation of the clause, as there is no adequate number of events to be paired consecutively.

Nevertheless, when there exists one B event at the beginning of the trace, this does not activate ChainPrecedence(A,B) as this is formulated as □(□B → A), as only a B that does not occur in the first position of the trace will activate the latter clause. Therefore, in this situation, we need to exclude—at most—one occurrence of B from the count. This exclusion results in a decrease in the count, thereby providing the lower bound for our constraint.

Proof (Lemma 2). Satisfaction of Surround(A,B) requires that both ChainPrecedence(A,B) and ChainResponse(A,B) must be satisfied. If both defining clauses are not vacuously satisfied, then the number of B-s doubles the number of A-s; as for each A occurring after the first event of the trace, there shall be a B both preceding and following it. Similar to the proof of the previous lemma, having events in that number is not a sufficient condition for the Surround to be verified, as simple counting does not prescribe any temporal occurrence of such events. Nevertheless, the violation of such a condition manifests the impossibility of Surrounding.

Nevertheless, when there exists at least one A event occurring at the beginning of the trace, this does not activate the ChainPrecedence(A,B), as this is formulated as □(□A → B), as only a B that does not occur in the first position of the trace will activate such a clause. In this situation, we might expect the Bs in the same number of A-s. (Bs can also be shared by As, i.e., the B event satisfying □(A ⇒ □B) might be the same event as □(□A ⇒ B) for the following A).

References
Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.