information

Article

Quickening Data-Aware Conformance Checking through Temporal Algebras †

Giacomo Bergami *, Samuel Appleby and Graham Morgan

School of Computing, Faculty of Science, Agriculture and Engineering, Newcastle University, Newcastle Upon Tyne NE4 5TG, UK

* Correspondence: giacomo.bergami@newcastle.ac.uk

† This paper is an extended version of our paper: Appleby, S.; Bergami, G.; Morgan, G. Running Temporal Logical Queries on the Relational Model. In Proceedings of the IDEAS'22, 26th International Database Engineered Applications Symposium, Budapest, Hungary, 22–24 August 2022.

Abstract: A temporal model describes processes as a sequence of observable events characterised by distinguishable actions in time. Conformance checking allows these models to determine whether any sequence of temporally ordered and fully-observable events complies with their prescriptions. The latter aspect leads to Explainable and Trustworthy AI, as we can immediately assess the flaws in the recorded behaviours while suggesting any possible way to amend the wrongdoings. Recent findings on conformance checking and temporal learning lead to an interest in temporal models beyond the usual business process management community, thus including other domain areas such as Cyber Security, Industry 4.0, and e-Health. As current technologies for accessing this are purely formal and not ready for the real world returning large data volumes, the need to improve existing conformance checking and temporal model mining algorithms to make Explainable and Trustworthy AI more efficient and competitive is increasingly pressing. To effectively meet such demands, this paper offers KnoBAB, a novel business process management system for efficient Conformance Checking computations performed on top of a customised relational model. This architecture was implemented from scratch after following common practices in the design of relational database management systems. After defining our proposed temporal algebra for temporal queries ($\texttt{xt}\text{LTL}_\text{f}$), we show that this can express existing temporal languages over finite and non-empty traces such as LTL_f. This paper also proposes a parallelisation strategy for such queries, thus reducing conformance checking into an embarrassingly parallel problem leading to super-linear speed up. This paper also presents how a single $\texttt{xt}\text{LTL}_\text{f}$ operator (or even entire sub-expressions) might be efficiently implemented via different algorithms, thus paving the way to future algorithmic improvements. Finally, our benchmarks highlight that our proposed implementation of $\texttt{xt}\text{LTL}_\text{f}$ (KnoBAB) outperforms state-of-the-art conformance checking software running on LTL_f logic.

Keywords: logical artificial intelligence; knowledge bases; query plan; temporal logic; conformance checking; temporal data mining; intraquery parallelism

Citation: Bergami, G.; Appleby, S.; Morgan, G. Quickening Data-Aware Conformance Checking through Temporal Algebras. *Information* **2023**, *14*, 173. https://doi.org/10.3390/info14030173

Academic Editor: Peter Revesz

Received: 14 November 2022
Revised: 3 March 2023
Accepted: 5 March 2023
Published: 8 March 2023

1. Introduction

(Temporal) conformance checking is increasingly at the heart of ARTIFICIAL INTELLIGENCE activities: due to its logical foundation, assessing whether a sequence of distinguishable events (i.e., a *trace*) does not conform to the expected process behaviour (*process model*) reduces to the identification of the specific unfulfilled temporal patterns, represented as logical *clauses*. This leads to Explainable AI, as the process model's violation becomes apparent. Clauses are the instantiation of a specific behavioural pattern (i.e., *template*) that expresses temporal correlation between actions being carried out (*activations*) and their expected results (*targets*). These, therefore, differ from traditional association rules [1], as they can also describe complex temporal requirements: e.g., whether the target should

immediately follow (ChainResponse) or precede (ChainPrecedence) the activation, if the former might happen any time in the future (Response), or if the target should have never happened in the past (Precedence). These temporal constraints can be fully expressed in a LINEAR TEMPORAL LOGIC OVER FINITE TRACES (LTL_f) and its extensions; this logic is referred to as linear as it assumes that, in a given sequence of events of interest, only one possible future event exists immediately following a given one. This differs from stochastic process modelling, where each event is associated with a probabilistic distribution of possibly following events [2,3]. Such a formal language can be extended to express correlations between activations and targets through binary predicates correlating data payloads. Events are also associated with either an action or a piece of state information represented as an *activity label*. Collections of traces are usually referred to as *log*.

Despite its theoretical foundations, state-of-the-art conformance checking techniques for entire logs expose sub-optimal run-time behaviour [4]. The reasons are the following: while performing conformance checking over relational databases requires computing costly aggregation conditions [5], tailored solutions do not exploit efficient query planning and data access minimisation, thus requiring scanning the traces multiple times [6]. Efficiency becomes of the uttermost importance after observing that conformance checking's run-time enhancement has a strong impact on a whole wide range of practical use case scenarios (Section 1.1). To make conformance checking computations efficient, we synthesise temporal data derived from a system (be it digital or real) to a simplified representation in the Relational Database model. In this instance, we use $\texttt{xt}LTL_f$, our proposed extension of LTL_f, to represent process models. While the original LTL_f merely asserts whether a trace is *conformant* to the model, our proposed algebra returns all of the traces satisfying the temporal behaviour, as well as activated and targeted events. As a temporal representation in the declarative model provides a point-of-relativity in the context of correctness (i.e., time itself may dictate if traces maintain correctness throughout the logical declarations expressed by the model), the considerations of such temporal issues significantly increase the checking requirement. This is due to a need to visit logical declarations for correctness in the context of each temporal instance.

This paper extends our previous work [4], where we clearly showed the disruptiveness of the relational model for efficiently running temporal queries outperforming state-of-the-art model checking systems. While our original work [4] provided just a brief rationale behind the success of KnoBAB (The acronym stands for KNOWLEDGE BASE FOR ALIGNMENTS AND BUSINESS PROCESS MODELLING). The Business Process Mining literature often uses the term Knowledge Base differently from customary database literature: while in the former, the intended meaning is a customary relational representation for trace data, in the latter, we often require that such representation provides a machine-readable representation of data in order to infer novel facts or to detect inconsistencies., this paper wants to dive deep into each possible contribution leading to our implementation success.

1. As an extension from our previous work, we fully formalise the logical data model (Section 3.1) and characterise the physical one (Section 4) in order to faithfully represent our log. This will prelude the full formalisation of the $\texttt{xt}LTL_f$ algebra;
2. Contextually, we also show for the first time that the $\texttt{xt}LTL_f$ algebra (Section 3.2) can not only express declarative languages such as Declare [7] as in our previous work but can express the semantics of LTL_f formula by returning any non-empty finite trace satisfying the latter if loaded in our relational representation (see Appendix A.2). We also show for the first time a formalisation for data correlation conditions associated with binary temporal operators;
3. Differently from our previous work, where we just hinted at the implementation of each operator with some high level, we now propose different possible algorithms for some $\texttt{xt}LTL_f$ operators (Section 6), and we then discuss both theoretically (Supplement II.2) and empirically (Section 7.1) which might be preferred under different trace length ϵ or log size $|\mathcal{L}|$ conditions. This leads to the definition of hybrid algorithms [8];

4. Our benchmarks demonstrate that our implementation outperforms conformance checking techniques running on both relational databases (Section 7.2) and on tailored solutions (Section 7.5) when customary algorithms are chosen for implementing $\texttt{xt}\text{LTL}_\text{f}$ operators;
5. Finally, this paper considerably extends the experimental section from our previous work. First, we show (Section 7.3) how the query plan's execution might be parallelised, thus further improving with super-linear speed-up our previous running time results. Then, we also discuss (Section 7.4) how different data accessing strategies achievable through query rewriting might affect the query's running time.

Figure 1 provides a graphical depiction of this paper's table of contents, with the mutual dependencies between its sections. Appendices and Supplements start from p. 50.

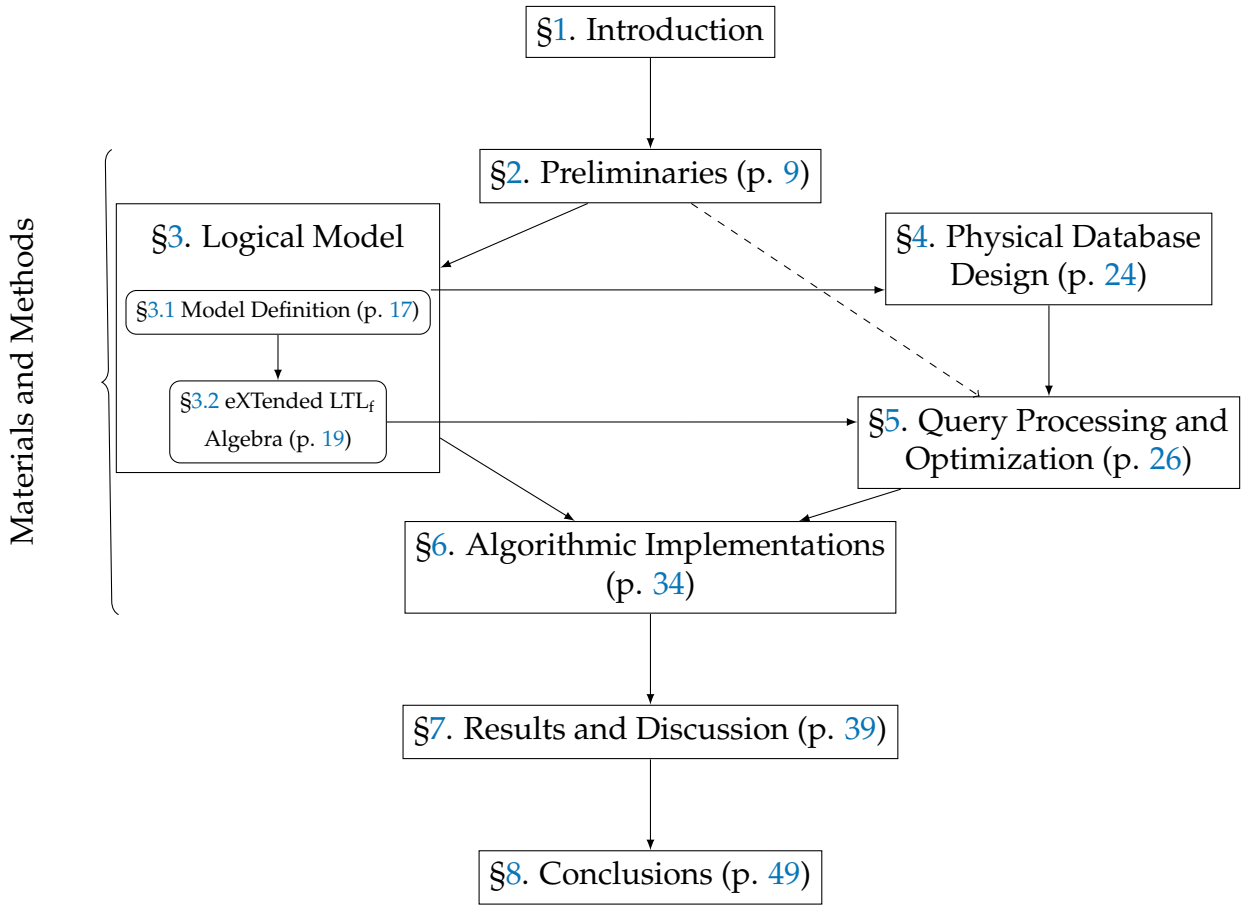


Figure 1. Table of Contents.

Figure 2 provides a bird-eye view of the overall KnoBAB architecture: in the upper half, we show how a log is loaded in our business process management system as a series of distinct tables providing some activity statistics (CountingTable) and full payload information (AttributeTable) in addition to reconstructing the unravelling of the events as described by their traces (ActivityTable). On the other hand, the lower half shows the main steps of the query engine transforming a declarative model into a DAG query plan accessing the previously-loaded relational tables. The most recent version of our system is on GitHub (https://github.com/datagram-db/knobab as accessed the 5 March 2023). When not explicitly stated, all the links were last accessed the 5 March 2023.

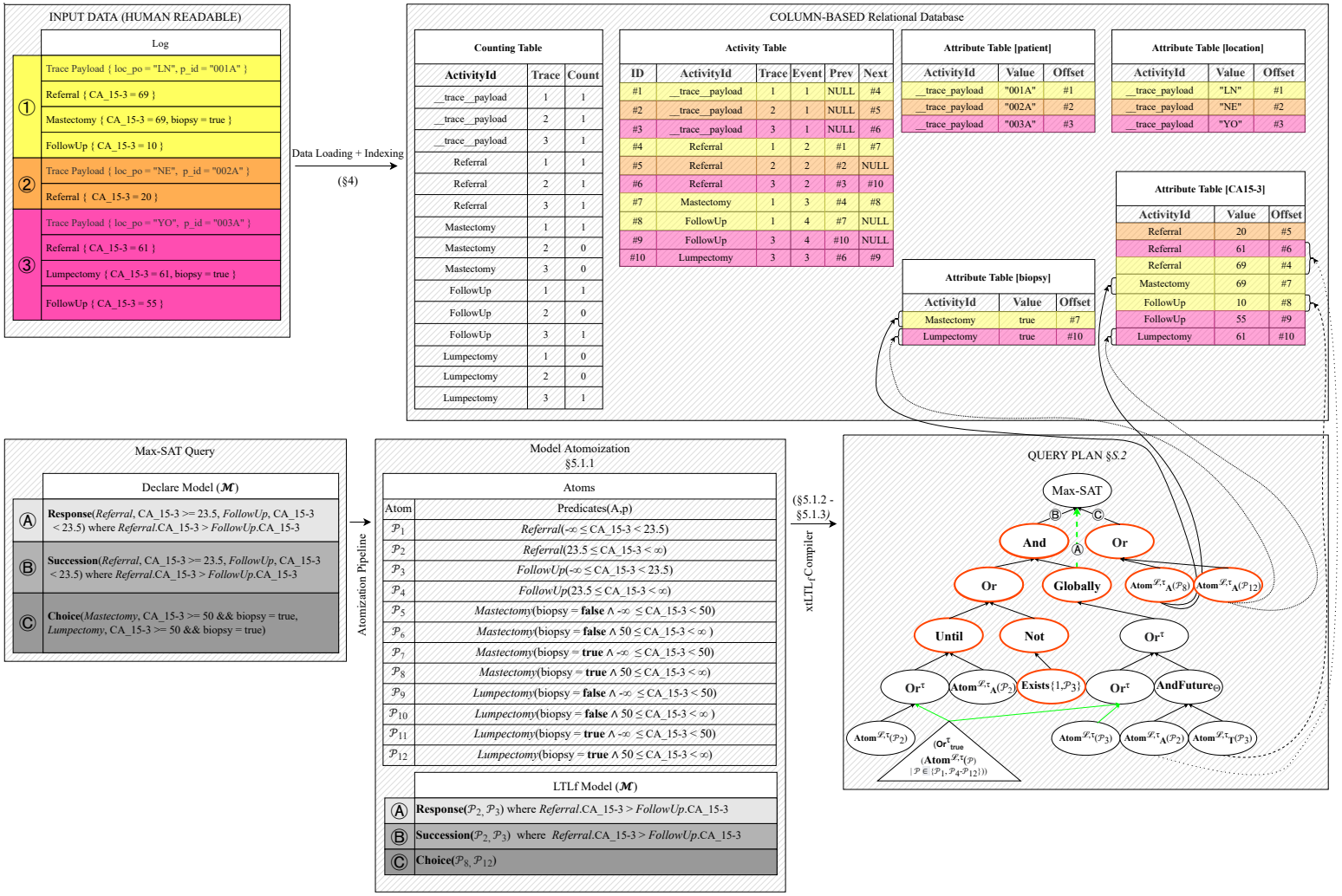


Figure 2. KnoBAB Architecture for Breast Cancer patients. Each trace ①–③ represents one single patient's clinical history, represented with unique colouring, while each Declare clause Ⓐ–Ⓒ prescribes a temporal condition that such traces shall satisfy. Please observe that the atomisation process does not consider data distribution but rather partitions the data space as described by the data activation and target conditions. In the query plan, green arrows indicate access to shared sub-queries as in [9], and thick red ellipses indicate which operators are untimed.

1.1. Case Studies

The present section shows a broad-ranging set of real case studies requiring efficient conformance checking computations in LTL_f*. This, therefore, motivates the need for our proposed approach in a practical sense.*

1.1.1. Cyber Security

Intrusion detection for cyber security aims at auditing an environment for identifying potential flaws that can be remedied and fixed later. While *anomaly-based* approaches raise an alarm if the observed behaviour differs significantly from the expected one, *signature-based* approaches check whether attack patterns might be recognised from the environment. The latter are often used to mitigate the high false-alarm rates of the former [10]. Expected behaviour might be encoded as process models expressed in LTL_f, which, when violated, lead to the detection of an attack: such a language can be directly exploited to represent several different kinds of attacks, such as Denial Of Service, Buffer Overflows, and Password Guessing [10]. In his dissertation [11], Ray shows how malware can be detected by determining LTL_f formulae discriminating between system–calls patterns generated by malicious software from expected run-time behaviour. Recent developments [12,13] showed that it is possible to perform prediction (and therefore reasoning) on potentially infinite sequences by analysing a finite subsequence of the overall behaviour within a sliding window; Buschjäger et al. [12] predict future events not covered by the sliding window by correlating them to the patterns observed in such a window. By associating a positive label to each finite subsequence preceding or containing an attack, and a negative one otherwise, we can also extract temporal models detecting subsequences containing attacks [14]. This entails that real-time verification boils down, to some extent, to offline monitoring, as we guarantee that it is sufficient to analyse currently-observed behaviours to predict and detect an attack. The learned model, once validated, can be exploited in the aforementioned real-time verification systems [10].

Example 1. *The Cyber Kill Chain® framework (https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html as accessed the 5 March 2023) allows the identification and prevention of intrusion activities on computer systems. This framework is based on a military tactic simply known as a kill chain (https://en.wikipedia.org/wiki/Kill_chain, 5 March 2023), which breaks down the attack into the following phases: target identification, marshalling and organizing forces towards the target, starting an attack, and target neutralisation. Lockheed Martin reformulated these steps to be transferred to the IT world and redirected the attack against a virtual target. These phases were reformulated as follows:*

Reconnaissance (*rec*): *An attacker observes the situation from the outside in order to identify targets and tactics. As the attacker mainly collects information regarding the system's vulnerabilities, this is the hardest part to detect.*

Weaponisation (*weap*): *After gathering the information, the cybercriminal implements his strategy through a software artefact. This detection will have greater chances of success in the future after post-mortem analysis, when either a temporal model is mined over the collected attack data or the strategy is directly inferred from available artefacts (e.g., malaware).*

Payload or Delivery (*del*): *The cybercriminal devises a way to infiltrate the host system that hides the previously produced artefact (e.g., a Trojan). This must sound as harmless as possible to fool the system.*

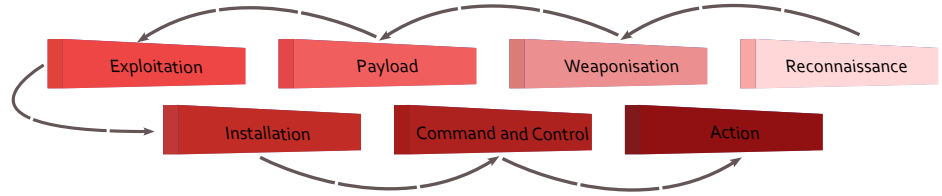
Exploitation (*expl*): *The cybercriminal exploits the system's vulnerabilities and infiltrates it through the previous "cover". At this stage, the defensive system should raise the alarm if any kind of unusual behaviour is detected while increasing the security level.*

Installation (*inst*): *The weapon escapes the payload and gets installed into the host computer system. At this point, any kind of suspected behaviour might be detected by malicious system calls.*

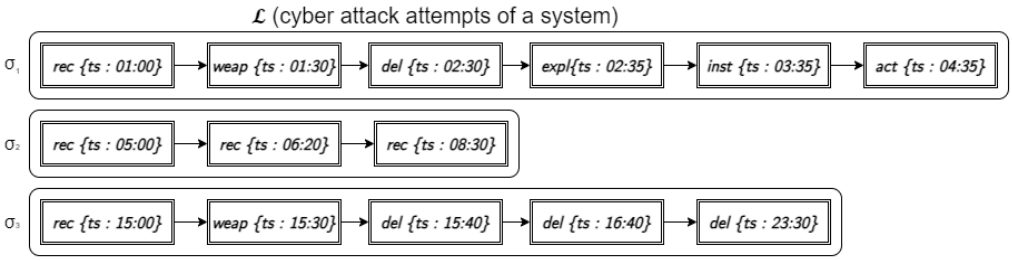
Command & Control (*comm*): *The weapon establishes a communication with the cybercriminal for receiving orders from the attacker. The system should detect any kind of suspicious network communication and should attempt to break the communication channel.*

Action (*act*): *The intruder starts the attack on the system. At this stage, the attack should be more evident, and the* ***Industrial IoT Shields*** *(iiot_sh), such as network devices protection, should be activated.*

*Figure 3a describes the actions (and therefore activity labels) of interest. Having defined the actions that should be monitored, records of activities can be stored as traces within a log. This is represented in Figure 3b, where we define three distinct attacks as distinct traces ($\sigma_1, \sigma_2, \sigma_3$). Each trace contains the event information leading up to the completion of an attack attempt (which may be (un)successful). Data payload information is also considered, and here this is provided as the unique timestamp (**ts**) associated with each event. Trace payload information is not simulated here but is described and applied in Example 2.*



(**a**)



(**b**)

Figure 3. We can express a cyber-security scenario by considering (**a**) possible situations in a Cyber Kill Chain, than are then (**b**) represented in the activity labels' names associated to the events.

A temporal model might describe a completely successful attack. The occurrence of the aforementioned phases can be described through a temporal declarative language Declare [7], where each constraint is an instantiated Declare clause (see Table 1). Our declarative language should be able to state the following requirements: Ⓐ all reconnaissance events should be followed by a weaponisation, Ⓑ there should be no IoT shields in place, and Ⓒ either command and control or action should occur.

On blockchains, each trace event represents a proper blockchain event, thus including function or event invocations issued by one or more smart contracts. In particular, smart contracts are sets of conditions specified in self-executing programs [15], which include protocols within which the parties will fulfil some promises [16]. Given that smart contracts can also be seen as postconditions activated upon the occurrence of specified pre-conditions [17], they are also exploited as security measures reducing malicious and accidental exceptions [15]. As per previous considerations, we can directly encode the smart contract premises in LTL_f*, as well as represent the whole smart contract as a whole* LTL_f *formula under the assumption that the blockchain guarantees its execution [17]. Therefore, we can perform post-mortem analysis checking whether a given run-time abides by the rules imposed by the system.*

Table 1. Declare templates illustrated as exemplifying clauses. $A \wedge p$ ($B \wedge q$) represents the *activation* (*target*) condition, A (B) denotes the activity label, and p (q) is the data payload condition.

Type	Exemplifying Clause (c_l)	Natural Language Specification for Traces	LTL$_f$ Semantics ($[\![c_l]\!]$)
Simple	Init(A, p)	The trace should start with an activation	$A \wedge p$
	Exists(A, p, n)	Activations should occur at least n times	$\begin{cases} \Diamond(A \wedge p \wedge \bigcirc([\![\text{Exists}(A, p, n-1)]\!])) & n > 1 \\ \Diamond(A \wedge p) & n = 1 \end{cases}$
	Absence$(A, p, n+1)$	Activations should occur at most n times	$\neg[\![\text{Exists}(A, p, n+1)]\!]$
	Precedence(A, p, B, q)	Events preceding the activations should not satisfy the target	$\neg(B \wedge p)\ \mathcal{W}\ (A \wedge p)$
(Mutual) Correlation	ChainPrecedence(A, p, B, q)	The activation is immediately preceded by the target.	$\Box(\bigcirc(A \wedge p) \Rightarrow (B \wedge q))$
	Choice(A, p, A', p')	At least one of the two activation conditions must appear.	$\Diamond(A \wedge p) \vee \Diamond(A' \wedge p')$
	Response(A, p, B, q)	The activation is either followed by or simultaneous to the target.	$\Box((A \wedge p) \Rightarrow \Diamond(B \wedge q))$
	ChainResponse(A, p, B, q)	The activation is immediately followed by the target.	$\Box((A \wedge p) \Rightarrow \bigcirc(B \wedge q))$
	RespExistence(A, p, B, q)	The activation requires the existence of the target.	$\Diamond(A \wedge p) \Rightarrow \Diamond(B \wedge q)$
	ExclChoice(A, p, A', p')	Only one activation condition must happen.	$[\![\text{Choice}(\texttt{A}, p, \texttt{A'}, p')]\!] \wedge [\![\text{NotCoExistence}(\texttt{A}, p, \texttt{A'}, p')]\!]$
	CoExistence(A, p, B, q)	RespExistence, and vice versa.	$[\![\text{RespExistence}(\texttt{A}, p, \texttt{B}, q)]\!] \wedge [\![\text{RespExistence}(\texttt{B}, q, \texttt{A}, p)]\!]$
	Succession(A, p, B, q)	The target should only follow the activation.	$[\![\text{Precedence}(\texttt{A}, p, \texttt{B}, q)]\!] \wedge [\![\text{Response}(\texttt{A}, p, \texttt{B}, q)]\!]$
	ChainSuccession(A, p, B, q)	Activation immediately follows the target, and the target immediately preceeds the activation.	$\Box((A \wedge p) \Leftrightarrow \bigcirc(B \wedge q))$
	AltResponse(A, p, B, q)	If an activation occurs, no other activations must happen until the target occurs.	$\Box((A \wedge p) \Rightarrow (\neg(A \wedge p)\ \mathcal{U}\ (B \wedge q)))$
	AltPrecedence(A, p, B, q)	Every activation must be preceded by an target, without any other activation in between	$[\![\text{Precedence}(\texttt{A}, p, \texttt{B}, q)]\!] \wedge \Box((A \wedge p) \Rightarrow \bigcirc(\neg(A \wedge p)\ \mathcal{W}\ (B \wedge q))$
Not.	NotCoExistence(A, p, B, q)	The activation nand the target happen.	$\neg(\Diamond(A \wedge p) \wedge \Diamond(B \wedge q))$
	NotSuccession(A, p, B, q)	The activation requires that no target condition should follow.	$\Box((A \wedge p) \Rightarrow \neg\Diamond(B \wedge q))$

1.1.2. Industry 4.0

Smart factories enable the collection and analysis of data through advanced sensors and embedded software for better decision-making. These enable monitoring each phase of the entire production process in both real-time and domain-specific applications where the safety of both autonomous cyber-physical systems as well as human workers is at stake [18]. This is of the uttermost importance, as both humans and machines cooperate in the same environment where a minimal violation of safety requirements might damage the overall production process, thus reflecting in maintenance costs. This calls for logical-based formal methods providing correctness guarantees [19]. Run-time verification [19] and prediction [13] have started gaining momentum against customary static analysis tools: in fact, *real* complex systems such as factories are often hard to predict and analyse before execution. As run-time verification can be deployed as a permanent testing condition on the environment, Mao et al. [19] show that this approach is complete, thus reducing the complicated model-checking problem into a simpler conformance checking one. PROGRAMMABLE LOGIC CONTROLLERS (PLC) are at the heart of this mechanism, where controllers can make decisions over previously-observed events. PLC work is similar to smart contracts in the previous scenario: at each "scan cycle", the controllers perceive through sensors the status change of the environment (e.g., variations of temperature and pressure). This information is then fed to the internal logic, which, on the other hand, might decide to intervene directly in the environment by sending signals to some actuators (e.g., controlling the pressure and temperature on the system). Due to the similarity of PLC to smart contracts, these might also exploit LTL$_f$ for determining security requirements: when a safety condition is violated, the PLC might activate an alarm while ensuring that the system works within safe operation ranges [19]. Please observe that ptLTL, also defined in [19], is a version of LTL allowing reasoning on past events so as to avoid semi-decidable computations for traces of infinite length, might be still represented through an equivalent LTL$_f$formula evaluated over a finite sliding window [13] bounded by the first and the latest event. Please observe that the difference between LTL and LTL$_f$ is that only the latter considers traces of finite length.

In some other industrial scenarios, we might be interested in detecting unexpected variations in time series reflecting the fluctuation of some perceived variables (e.g., variations in temperature and pressure). The latest developments [13] showed that (industrial) time series could also be represented as traces: we might assign to each event an activity label ϖ if the current event has a data payload whose values upper bound the ones from

immediately preceding event's payload, and $\neg\varnothing$ otherwise. Consequently, we can encode disparate data variation patterns in LTL_f reflecting different types of data volatility or steep increases/decreases [13]. This shows how LTL_f can also represent *anomaly-based* problems by reducing them to the identification of anomaly patterns [20].

1.1.3. Healthcare

A medical process describes clinical-related procedures as well as organisational management ones (e.g., registration, admission, and discharge) [21]. The renowned openEHR (https://www.openehr.org/ accessed the 5 March 2023) standard distinguishes the former in four main archetypes: an observation, recording patients' clinical symptoms (e.g., body temperature, blood pressure); an evaluation, providing preliminary diagnosis and assessing the patient's health based on the former results; and an instruction, the execution of the treatment plan proposed by a physician (e.g., prescribing, examining, and testing). An action describes the way to intervene or treat medical patients according to the treatment plan (e.g., drug administration, blood matching). Once encoded as such, each process representing an instantiation of a medical process, i.e., a patient's clinical course, can be then collected and represented in a log. As such, each action is going to be represented as a distinct activity label of a given event [22] that might contain relevant payload information recording the outcome of the clinical procedures, as well as demographical information related to the patient [21] for future socio-clinical analyses [23].

Declarative temporal languages such as Declare can then be exploited to provide a descriptive approach specifying temporal constraints among activities without strictly enforcing their order of completion, thus restricting the order of application of a specific set of activities [21]. As these models come with temporal semantics expressed in LTL_f, these are, for all intents and purposes, process models. As such, these might be applied to detect discrepancies between clinical guidelines, expressed by the aforementioned model, and the actual process executions collected in a log. This is of the utmost concern as often deviations represent errors compromising the patient recovery [22], which, if efficiently and identified in advance, lead to an increased patient satisfaction as well a reduction of healthcare costs (e.g., due to mismanagement) [21].

Example 2. *To minimise costs and unrequired procedures, only ill patients should receive treatment. Thus, sufferers not receiving treatment (false negatives) and non-sufferers receiving treatment (false positives) need to be minimised. Figure 2 proposes a simplified scenario where we consider two event payload keys:* CA 15-3 *(cancer antigen concentration in a patient's blood) and* biopsy *(biopsies should be taken before any procedure is acted upon). Our model targets only breast cancer patients with successful therapies that describe a medical protocol and the desired patients' health condition at each step.* Ⓒ *states that two possible surgical operations for breast tumours are mastectomy or lumpectomy if the biopsy is positive and the CA-13.5 is way above ($\geq$ 50) the guard level, being 23.5 units per mL, and* Ⓐ–Ⓑ *any successful treatment should decrease the CA-13.5 levels, which should be below the guard level; such correlation data condition is expressed via a* Θ *condition (introduced by a* where*). A twinned negative model (not in Figure) might better discriminate healthy patients from patients where the therapy was unsuccessful. Novel situations can be represented as a log. For example, in Figure 2, we have three patients:* ① *a cancer patient with a successful mastectomy,* ② *a healthy patient, and* ③ *an unsuccessful lumpectomy, thus suggesting that the patient might still have some cancerous cells. Given the aforementioned model, patient* ① *will satisfy the model as the surgical operation was successful,* ② *will not satisfy the model because neither a mastectomy nor a lumpectomy was required ($\mathcal{M}$ is only fulfilled for successful procedures), and* ③ *will not satisfy the target condition, even though the correlation condition was met. Our model of interest should only return* ① *as an outcome of the conformance checking process.*

2. Preliminaries

eXtensible Event Stream (XES). This paper relies on temporal data represented as a temporally ordered sequence of events (*trace* or *streams*), where events are associated with at most one action described by a single *activity label* [24]. In this paper, we formally characterize payloads as part of both events and traces while, in our previous work, we only considered payloads from events [25].

Given an arbitrarily ordered set of keys K and a set of values V, a **tuple** [26] is a finite function $p\colon K \to V$ (also $p \in V^K$), where each key is either associated with a value in V or is undefined. After denoting $\perp$ as a `null` element missing from the set of values ($\perp \notin V$), we can express that κ is not associated with a value in p as $p(\kappa) = \perp$, thus $\kappa \notin \mathrm{dom}(p)$. An empty tuple ε has an empty domain.

(Data) *payloads* are tuples, where values can represent either categorical data or numerical data. An *event* σ_j^i is a pair $\langle \mathsf{a}, p \rangle \in \Sigma \times V^K$, where Σ is a finite set of activity labels, and p is a finite function describing the data payload. A *trace* σ^i is an ordered sequence of distinct events $\sigma_1^i, \ldots, \sigma_n^i$, which is distinguished from the other traces by a case id i; n represents the trace's length ($n = |\sigma^i|$). If a payload is also associated with the whole trace, this can be easily mimicked by adding an extra initial event containing such a payload with an associated label of `__trace_payload`. A *log* $\mathcal{L}$ is a finite set of traces $\{ \sigma^1, \ldots, \sigma^m \}$. In this paper, we further restrict our interest to the traces containing at least one event, as empty traces are meaningless as they are not describing any temporal behaviour of interest. Finally, we denote as $\beta\colon \Sigma \leftrightarrow \{ 1, \ldots, |\Sigma| \}$ the bijection mapping each activity label occurring in the log to an unique id.

Example 3. *The log $\mathcal{L}$ in Figure 2 comprises three distinct traces $\mathcal{L} = \{ \sigma^1, \sigma^2, \sigma^3 \}$. In particular, the second trace comprises two events $\sigma^2 = \sigma_1^2 \sigma_2^2$, where the first event represents the trace payload, and therefore $\sigma_1^2 = \langle$ `__trace_payload`, $p \rangle$ having $p(loc\_po) = NE$ and $p(p\_id) = 002A$. The other event is $\sigma_2^2 = \langle$ `Referral`, $\tilde{p} \rangle$, where payload $\tilde{p}$ is only associated with the CA-13.5 levels as $\tilde{p}(CA\text{-}13.5) = 20$. Similar considerations can be carried out for the other log traces.*

Linear Temporal Logic over finite traces ($\mathrm{LTL_f}$). $\mathrm{LTL_f}$ is a well-established extension of *modal logic* considering the possible worlds as finite traces, where each event represents a single relevant instant of time. The time is thereby linear, discrete, and future-oriented. This entails that that the events represented in each trace are totally ordered and, as $\mathrm{LTL_f}$ quantifies only on events reported in the trace, all the events of interest are fully observable. The *syntax* of an well-formed $\mathrm{LTL_f}$ formula φ is defined as follows:

$$\varphi ::= \mathsf{a} \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi \,\mathcal{U}\, \varphi' \tag{1}$$

where $\mathsf{a} \in \Sigma$. Its *semantics* is usually defined in terms of First Order Logic [27] for a given trace σ^i at a current time j (e.g., for event σ_j^i) as follows:

- An event satisfies the **activity label** a iff. its activity labels is a: $\sigma_j^i \vDash \mathsf{a} \Leftrightarrow \sigma_j^i = \langle \mathsf{a}, p \rangle$;
- An event satisfies the **negated** formula iff. the same event does not satisfy the non-negated formula: $\sigma_j^i \vDash \neg\varphi \Leftrightarrow \sigma_j^i \nvDash \varphi$;
- An event satisfies the **disjunction** of $\mathrm{LTL_f}$ sub-formulæ iff. the event satisfies one of the two sub-formulæ: $\sigma_j^i \vDash \varphi \vee \varphi' \Leftrightarrow \sigma_j^i \vDash \varphi \vee \sigma_j^i \vDash \varphi'$;
- An event satisfies the **conjunction** of $\mathrm{LTL_f}$ formulæ iff. the event satisfies all of the sub-formulæ: $\sigma_j^i \vDash \varphi \wedge \varphi' \Leftrightarrow \sigma_j^i \vDash \varphi \wedge \sigma_j^i \vDash \varphi'$;
- An event satisfies a formula at the **next** step iff. the formula is satisfied in the incoming event if present: $\sigma_j^i \vDash \bigcirc\varphi \Leftrightarrow i < |\sigma_j| \wedge \sigma_{j+1}^i \vDash \varphi$;
- An event **globally** satisfies a formula iff. the formula is satisfied in all the following events, including the current one: $\sigma_j^i \vDash \Box\varphi \Leftrightarrow \forall j \leq x \leq |\sigma^i|.\sigma_x^i \vDash \varphi$;

- An event **eventually** satisfies a formula iff. the formula is satisfied in either the present or in any future event: $\sigma^i_j \vDash \Diamond\varphi \Leftrightarrow \exists j \leq x \leq |\sigma^i|.\sigma^i_x \vDash \varphi$;
- An event satisfies φ **until** φ' holds iff. φ holds at least until φ' becomes true, which must hold at the current or a future position: $\sigma^i_j \vDash \varphi\ \mathcal{U}\ \varphi' \Leftrightarrow \exists j \leq y \leq |\sigma^i|.\sigma^i_y \vDash \varphi' \wedge \left(\forall x \leq z < y.\sigma^i_z \vDash \varphi\right)$

Other operators can be seen as syntactic sugar: Weak-Until is denoted as $\varphi\ \mathcal{W}\ \varphi' := \varphi\ \mathcal{U}\ \varphi' \vee \Box\varphi$, while the implication can be rewritten as $\varphi \Rightarrow \varphi' := (\neg\varphi) \vee (\varphi \wedge \varphi')$. Generally, binary operators bridge activation and target conditions appearing in two distinct sub-formulæ. The semantics associated with activity labels, consistently with the literature on business process execution traces [25], assumes that, in each point of the sequence, one and only one element from Σ holds. We state that a trace σ^i is *conformant* to an LTL_f formula iff. it satisfies it starting from the first event: $\sigma^i \vDash \varphi \Leftrightarrow \sigma^i_1 \vDash \varphi$, and is *deviant* otherwise [25]. The Declare language described in the next paragraph provides an application for such logic. As relational algebra describes the semantics for SQL [28,29], LTL_f is extensively applied [30] as a semantics for formally expressing temporal and human-readable declarative constraints such as Declare.

At the time of the writing, different authors proposed several extensions for representing data conditions in LTL_f. The simplest extensions are *compound conditions* $\mathsf{a} \wedge q$, which are the conjunction of data predicate $q \in \textbf{Prop}$ to the activity label a [25]. Nevertheless, this straightforward solution is not able to express correlation conditions in the data which might be relevant in business scenarios [31], as representing correlations as single atoms requires decomposing the former into disjunctions of formulae [32]. Despite prior attempts to define a temporal logic expressing correlation conditions, no explicit formal semantics on how this can be evaluated was provided [6]. This poses a problem to the current practitioner, as this hinders the process of both checking formally the equivalence among two languages expressing correlation conditions, as well as providing a correct implementation of such an operator. We, on the other hand, propose a relational representation of $\texttt{xt}\text{LTL}_\text{f}$, where the semantics of all of the operators, thus including the ones requiring correlation conditions, is clearly laid out and implemented.

Declare. Temporal declarative languages pinpoint highly variable scenarios, where state machines provide complicated graph models that can be hardly understandable by the common business stake-holder [33]. Among all possible temporal declarative languages, we constrain our interest to Declare, originally proposed in [7]. Every single temporal pattern is expressed through *templates* (i.e., an abstract parameterised property: Table 1 column 2), which are parametrised over activation, target, or correlation conditions. Template names induce the semantic representation in LTL_f $[\![c_l]\!]$ of each model clause c_l. Therefore, a trace σ^i is conformant to a Declare clause iff. it satisfies its associated semantic representation in LTL_f ($\sigma^i \vDash c_l \Leftrightarrow \sigma^i \vDash [\![c_l]\!]$). At this stage, *activation* (and *target*) *conditions* are predicates $\mathsf{A} \wedge p$ (and $\mathsf{B} \wedge q$) in such a clause asserting required properties for the events' activity label (A and B) and payload (p and q). An event in a given trace *activates* (or *targets*) a given clause if they satisfy the activation (or target) condition. Please observe that neither activation nor target conditions postulate the temporal (co)occurrence between activating or targeting events, as this is duty is transferred to the specific LTL_f semantics of the clause. A trace *vacuously satisfies* a clause if the trace satisfies the clause despite no event in the trace satisfied the activation condition. After this, we state that a trace *non-vacuously satisfies* the declarative clause if the trace satisfies the clause and one of the following conditions is satisfied:

- The clause provides no target condition and it exists at least one activating event;
- The clause provides a target condition but no binary (payload) predicate Θ, and the declarative clause establishes a temporal correlation between (at least one) activating event and (at least one) targeting one;
- The clause provides both a target condition and a binary predicate Θ, while the activating and targeting events satisfying the temporal correlation as in the previous

case also satisfy a binary Θ predicate over their payloads; in this situation, we state that the activating and targeting event *match* as they jointly satisfy the *correlation condition* Θ.

Finally, the presence of activating events is a necessary condition for non-vacuous satisfiability.

We can then categorize each Declare template from [30] through these conditions and the ability to express correlations between two temporally distant events happening in one trace: simple templates (Table 1, rows 1–3) only involving activation conditions; (mutual) correlation templates (rows from 4 to 15), which describe a dependency between activation and target conditions, thus including correlations between the two; and negative relation templates (last 2 rows), which describe a negative dependency between two events in correlation. Despite these templates possibly appearing quite similar, they generate completely different finite state machines, thus suggesting that these conditions are not interchangeable (http://ltlf2dfa.diag.uniroma1.it/, 5 March 2023). Figure 4 exemplifies the behavioural difference between two clauses differing only on the template of choice.

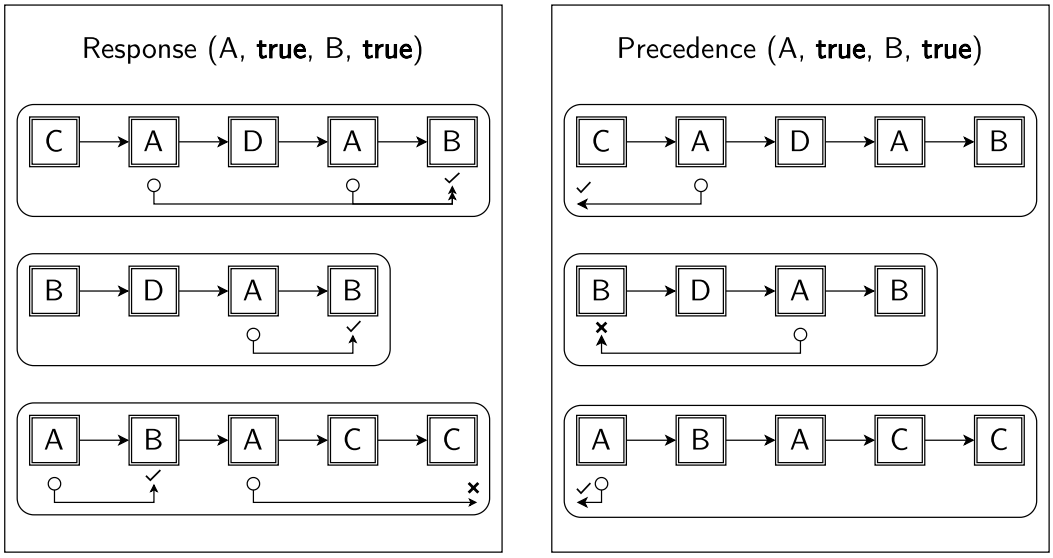


Figure 4. Two exemplifying clauses distinguishing Response and Precedence behaviours. Traces are represented as temporally ordered events associated with activity labels (boxed). Activation (or target) conditions are circled here (or ticked/crossed). Ticks (or crosses) indicate a (un)successful match of a target condition. For all activations, there must be an un-failing target condition; for precedence, we shall consider at most one activation. These conditions require the usage of multiple join tests per trace.

A Declare Model is composed of a set of clauses $\mathcal{M} = \{ c_l \}_{l \leq n, n \in \mathbb{N}}$ which have to be contemporarily satisfied in order to be true. A trace σ^i is conformant to a model $\mathcal{M}$ iff. such a trace satisfies each LTL_f formula $[\![c_l]\!]$ associated with the model clause $c_l \in \mathcal{M}$. Consequently, a Declare model can be represented as a finitary conjunction of the LTL_f representation of each of its clauses, $[\![\mathcal{M}]\!] := \bigwedge_{c_l \in \mathcal{M}} [\![c_l]\!]$: for this, the MAXIMUM-SATISFIABILITY PROBLEM (Max-SAT) for each trace counts the ratio between the satisfied clauses over the whole model size. This consideration can be extended later on to also data predicates through predicate atomisation [25], as discussed in the next paragraph.

Relational Models and Algebras. The *relational model* was firstly introduced by Codd [34] to compactly operate over tuples grouped into tables. Such tables are represented as mathematical n-ary relations $\Re$ that can be handled through a relational algebra. Upon the effective implementation of the first RELATIONAL DATABASE MANAGEMENT SYSTEMS (RDBMS), such algebra expressed the semantics of the well-known declarative query language, SQL. The rewriting of SQL in algebraic terms allowed the efficient execution

of the declarative queries through abstract syntax tree manipulations [28]. Our proposed $\texttt{xt}\text{LTL}_\text{f}$ (Section 3.2) takes inspiration from this historical precedent, in order to run conformance checking and temporal model mining queries over an relational representation of the log via relational tables (Section 3.1).

More recently, *column-oriented DBMS* such as MonetDB [35] proposed a new way to store data tables: instead of representing these per row, these were stored by column. There are several advantages to this approach, including better access to data when querying only a subset of columns (by eliminating the need to read columns that are not relevant) as well as discarding null-valued cells. This is achieved by representing each relation $\Re(\texttt{id}, A_1, \ldots, A_n)$ in the database schema as distinct binary relations $\Re_{A_i}(\texttt{id}, A_i)$ for each attribute A_i in $\Re$. As this decomposition guarantees that the full-outer natural join $\mathbb{D}\!\!\bowtie_{1 \leq i \leq n} \Re_{A_i}$ over the decomposed tables is equivalent to the initial relation $\Re$, we can avoid representing `NULL` values in each single binary relation, thus limiting our space allocation to the values effectively present in the data. We therefore took inspiration from this intuition for representing the payload information, thus storing one single table per payload attribute. To further optimise the query engine, it is also possible to boost the query performance by guaranteeing that the results always have a fixed schema, mainly listing the record ids satisfying the query conditions [36]. As we will see while introducing our temporal operators (Section 3.2), we will also guarantee that each operator returns the output in the same schema, thus guaranteeing time and memory optimality.

Finally, the *nested relational model* [37] extends the relational model by relaxing its first normal form (1NF), thus allowing table cells to contain tables and relations as values. Relaxing this 1NF allows for storing data in a hierarchical way in order to access an entire sub-tree with a single read operation. We will leverage this representation for our intermediate result representation, in order to associate multiple activation, target, or correlation conditions to one single event, thus including any relevant future event occurring after it.

Common Subquery Problem. Query caching mechanisms [38] are customary solutions for improving query runtime by holding partially-computed results in temporary tables referred to as materialised views, under the assumption that the queries sharing common data are pipelined [39]. Recently, Kechar et al. [9] proposed a novel approach that can also be run when queries are run contemporarily: it is sufficient to find the shared subqueries before actually running them so that, when they are run, their result is stored into materialised views thus guaranteeing that these are computed at most once.

Example 4. *Figure 2 shows how this idea might be transferred to our use case scenario requiring running multiple declarative clauses:* Response *is both a subquery of* Succession *as well as a distinct declarative clause of interest. Green arrows indicate operators' output shared among operators expressed in our proposed* $\texttt{xt}\,\text{LTL}_f$ *extension of* $\texttt{xt}\,\text{LTL}_f$*. Please also observe that operators with the same name and arguments but marked either with activation, target, or no specification are considered different as they provide different results, and therefore are not merged together. This includes distinctions between timed and untimed operators, which will be discussed in greater detail in Section 3.2.*

To further minimize tables' access times, it is possible to take this reasoning to its extreme by minimising the data access per data predicate in order to avoid accessing the same table multiple times. In order to do so, we need to partition the data space according to the queries at our disposal as in our previous work [25]. This process can be eased if we assume that each payload condition p and p' for the declarative clauses within a model $\mathcal{M}$ is represented in Disjunctive Normal Form (DNF) [40]: in this scenario, data predicates q are in DNF if they are a disjunction of one or more conjunctions of one or more data intervals referring to just one payload key.

Example 5. *The model illustrated in Figure 3a and discussed in former Example 1 comes with data conditions associated with neither activation nor target conditions. Therefore, no atomisation process is performed. Thus, each event in a log might just be distinguished by its activity label [25].*

Given an LTL_f expression φ containing compound conditions, we denote $\mathcal{D}_\varphi$ as the set of distinct compound conditions in φ. We refer to the items in $\mathcal{D}_\varphi$ as *atoms* iff. for each pair of distinct compound conditions in it, they never jointly satisfy any possible payload p (More formally, $\forall p.\forall \mathsf{a} \in \Sigma.\forall \mathsf{a} \wedge q, \mathsf{a} \wedge q' \in \mathcal{D}_\varphi.(q \neq q') \Rightarrow (q(p) \Rightarrow \neg q'(p)))$. Ref. [25] shows a procedure showing how any formula φ can be rewritten into an equivalent one φ' by ensuring that $\mathcal{D}_{\varphi'}$ contains atoms. This can be achieved by constructing $\mathcal{D}_{\varphi'}$ first from φ (Algorithm 1), and then converting each compound conditions in φ as disjunctions of atoms in $\mathcal{D}_{\varphi'}$, thus obtaining φ'.

Algorithm 1 Atomisation: $\mathcal{D}_\varphi$-encoding pipeline.

```
1:  global μ ← {}; ad ← {}; ak ← {}

2:  procedure CollectIntervals(a, DNF)          ▷ DNF:= ⋁_{1≤i≤n} ⋀_{1≤k≤m(i)} low_{i,k} ≤ k_{i,k} ≤ up_{i,k}
3:      for all conj ∈ DNF and low ≤ k ≤ up ∈ conj do
4:          μ(a, k).put([low, up])
5:      end for

6:  procedure CollectIntervals(M)               ▷ M := ⋀_{1≤i≤|M|} clause_i(A, p, B, p')
7:      for all clause_i(A, p, B, p') ∈ M do
8:          if p ≠ True then CollectIntervals(A, p)
9:          if p' ≠ True then CollectIntervals(B, p')
10:     end for

11: procedure D_φ-encoding( )
12:     for all a ∈ Σ do
13:         for all k ∈ K do
14:             μ(a, k) ←SegmentTree(μ(a, k))
15:         end for
16:         for all partition ∈ ⨉_{k∈K} μ(a, k).elementaryIntervals() do   ▷ partition := (low_k ≤ k ≤ up_k)_{k∈K}
17:             p_i ←new atom()
18:             p_i := a ∧ partition
19:             ak(a).put(p_i)
20:             for all low_k ≤ k ≤ up_k ∈ partition do
21:                 ad(a, low_k ≤ k ≤ up_k).put(p_i)
22:             end for
23:         end for
24:         if ak(a) = ∅ then
25:             ak(a) ← {a}
26:         end if
27:     end for
```

We collect all the conjunctions referring to the same payload key into a map $\mu(\mathsf{a}, \kappa)$ (Line 4). After doing so, we can construct a Segment Tree [41] from the intervals in $\mu(\mathsf{a}, \kappa)$, thus identifying the *elementary intervals* partitioning the collected intervals (Line 14). These elementary intervals also partition the payload data space associated with events for each activity label a. This can be achieved by combining each elementary interval in each dimension κ for a (Line 16) and then associating it with a new atom representing such a partition (Line 18) that is then guaranteed to be an atom by construction. This entails that each interval $low_\kappa \leq \kappa \leq up_\kappa$ will be characterised by the disjunction of all of the atoms p_i comprising such interval (Line 21). Given this, we can then associate to each activation condition A that is associated with an activation payload condition p the disjunction of atoms that are collected by the following formula:

$$\mathsf{Atom}_{\mu,ad}(\mathsf{A},p) := \bigcup_{conj\in p} \bigcap_{(low\leq\kappa\leq up)\in conj} \bigcup_{I\in\mu(\mathsf{A},\kappa).\texttt{findElementaryIntervals}(low,up)} ad(\mathsf{A},I) \quad (2)$$

If we assume that the dimension of $\mu(\mathsf{a},\kappa)$ for each $\mathsf{a}\in\Sigma$ and $\kappa\in K$ is at most m, our implementation available at https://github.com/datagram-db/knobab/blob/main/include/yaucl/structures/query_interval_set/structures/segment_partition_tree.h (5 March 2023) builds such trees in $\sum_{1\leq i\leq m}\log(i)+m\in O(m\cdot\log(m))$ time, as we first insert the intervals into the data structure and then we guarantee to minimise the tree representation, requiring a linear visit cost to the whole tree data structure. The time complexity of $\mathcal{D}_\varphi$-ENCODING() is $m|K|(1+\log m+|\Sigma|)\in O(m|K||\Sigma|)$.

Example 6. *Each distinct payload conditions associated with either activation or target conditions in Figure 2 can be expressed as one single atom, as there are no overlapping data conditions associated with the same activity label, and each data condition can be mapped into one single elementary interval associated with an activity label. The next example will provide another use case example and a different model on the same dataset leading to a decomposition of payload conditions into a disjunction of several atoms. Table 2 shows the partitioning of the data payloads associated with each activity label in the log by the elementary interval of interest.*

Table 2. Definition of the atoms from Figure 2 in terms of partitioning over the elementary intervals.

Referral	*CA-15.3* <23.5	*CA-15.3* ≥ 23.5
	p_1	p_2
Mastectomy	*CA-15.3* <50	*CA-15.3* ≥ 50
biopsy = **false**	p_5	p_6
biopsy = **true**	p_7	p_8
FollowUp	*CA-15.3* <23.5	*CA-15.3* ≥ 23.5
	p_3	p_4
Lumpectomy	*CA-15.3* <50	*CA-15.3* ≥ 50
biopsy = **false**	p_9	p_{10}
biopsy = **true**	p_{11}	p_{12}

Example 7. *Let us suppose to return all the false negative and false positive* Mastectomy *cases that are not caused by data imputation errors. For this, we want to obtain all of the negative biopsies having CA15.3 levels greater than the guard level of 50 and positive biopsies having CA15.3 below the same threshold. Under the assumption that biopsy values were imputed through numerical numbers thus leading to more imputation errors, we are ignoring cases where both CA15.3 and biopsy values are out of scale, that is, we want to ignore the data where CA15.3 levels are negative or above 1000, and where the biopsy values are neither true (*1.0*) nor false (*0.0*). For this, we can outline the following model:*

$$\begin{aligned}\mathcal{M}' = \{&\mathit{Choice}(\mathsf{Mastectomy}, \mathit{biopsy}=0.0\wedge \mathit{CA15.3}\geq 50, \mathsf{Mastectomy}, \mathit{biopsy}=1.0\wedge \mathit{CA15.3}<50),\\ &\mathit{Absence}(\mathsf{Mastectomy}, \mathit{CA15.3}>1000\vee \mathit{CA15.3}<0),\\ &\mathit{Absence}(\mathsf{Mastectomy}, \mathit{biopsy}\neq 1.0\vee \mathit{biopsy}\neq 0.0)\}\end{aligned} \quad (3)$$

This implies that we are interested in decomposing the intervals pertaining to both CA-15.3 and biopsy into elementary intervals: Table 3a shows that only CA-15.3 <50 *and CA-15.3* ≥ 50 *are decomposed into two elementary intervals, as the former also includes the range CA-15.3* <0*, while the latter also includes CA-15.3* >1000*. Elementary intervals not occurring in the initially collected ones are not reported in this graphical representation. Table 3b shows the partitioning*

of the Mastectomy *data payload induced by the elementary intervals of interest; the former data conditions can be now rewritten after Equation (2) in the Supplement as follows:*

1. $\bigvee \mathit{Atom}_{\mu,ad}(\text{Mastectomy}, \mathit{biopsy}{=}0.0 \wedge \mathit{CA15.3} \geq 50) = p_{12} \vee p_{17}$
2. $\bigvee \mathit{Atom}_{\mu,ad}(\text{Mastectomy}, \mathit{biopsy}{=}1.0 \wedge \mathit{CA15.3} < 50) = p_4 \vee p_9$
3. $\bigvee \mathit{Atom}_{\mu,ad}(\text{Mastectomy}, \mathit{CA15.3} > 1000 \vee \mathit{CA15.3} < 0) = p_1 \vee \cdots \vee p_5 \vee p_{16} \vee \cdots \vee p_{20}$
4. $\bigvee \mathit{Atom}_{\mu,ad}(\text{Mastectomy}, \mathit{biopsy} \neq 0.0 \vee \mathit{biopsy} \neq 1.0) = p_1 \vee p_3 \vee p_5 \vee p_6 \vee p_8 \vee p_{10} \vee p_{11} \vee p_{13} \vee p_{15} \vee p_{16} \vee p_{18} \vee p_{20}$

where each atom is defined as a conjunction of compound conditions defined upon the previously collected elementary intervals. Some examples are then the following:

- $p_1 := \mathit{biopsy} < 0 \wedge \mathit{CA\text{-}15.3} < 0$
- $p_2 := \mathit{biopsy} = 0 \wedge \mathit{CA\text{-}15.3} < 0$

This decomposition will enable us to reduce the data access time while scanning the tables efficiently.

Table 3. Intermediate steps to generate distinct atoms for the Referral data predicates from Example 7.

(**a**) Interval decomposition in *basic intervals* $\mu(\text{Mastectomy}, \cdot)$.

$\mu(\text{Mastectomy}, \mathit{CA\text{-}15.3})$			
$\mathit{CA\text{-}15.3} < 0$	$\mathit{CA15.3} < 0$		
$\mathit{CA\text{-}15.3} < 50$	$\mathit{CA15.3} < 0$,	$0 \leq \mathit{CA\text{-}15.3} < 50$	
$\mathit{CA\text{-}15.3} \geq 50$	$50 \leq \mathit{CA15.3} \leq 1000$,	$\mathit{CA\text{-}15.3} > 1000$	
$\mathit{CA\text{-}15.3} > 1000$	$\mathit{CA15.3} > 1000$		
$\mu(\text{Mastectomy}, \mathit{biopsy})$			
$\mathit{biopsy} = 0$	$\mathit{biopsy} = 0$		
$\mathit{biopsy} = 1$	$\mathit{biopsy} = 1$		
$\mathit{biopsy} \neq 0$	$\mathit{biopsy} < 0$,	$0 < \mathit{biopsy} < 1$,	$\mathit{biopsy} = 1$, $\mathit{biopsy} > 1$
$\mathit{biopsy} \neq 0$	$\mathit{biopsy} < 0$,	$\mathit{biopsy} = 0$,	$0 < \mathit{biopsy} < 1$, $\mathit{biopsy} > 1$

(**b**) Atom generation by partitioning the data space $\bigtimes_{\kappa \in K} \mu(\text{Mastectomy}, \kappa)$.`elementaryIntervals()` with $K = \{\, \mathit{biopsy}, \mathit{CA\text{-}15.3} \,\}$.

	$\mathit{biopsy} < 0$	$\mathit{biopsy} = 0$	$0 < \mathit{biopsy} =< 1$	$\mathit{biopsy} = 1$	$\mathit{biopsy} > 1$
$\mathit{CA15.3} < 0$	p_1	p_2	p_3	p_4	p_5
$0 \leq \mathit{CA15.3} < 50$	p_6	p_7	p_8	p_9	p_{10}
$50 \leq \mathit{CA15.3} \leq 1000$	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}
$\mathit{CA15.3} > 1000$	p_{16}	p_{17}	p_{18}	p_{19}	p_{20}

Further Notation. We represent relational tables as a sequence of records indexed by id as per the physical relational model: given a relational table T, $T[i]$ represents the i-th record in T counting from 1. We denote $f = [x \mapsto y, z \mapsto t]$ as a finite function such that $f(x) = y$ and $f(z) = t$. Table 4 collects the notation used throughout the paper.

Table 4. Table of Notation for symbols $\chi \in \mathcal{T}$ defined as ($\chi := \mathcal{E}$) or characterised by ($\mathcal{E}(\chi)$) $\mathcal{E}$.

Symbol (χ)	Definition ($\mathcal{E}$)	Type ($\mathcal{T}$)	Comments
		Set Theory	
$\emptyset$		Set	An empty set contains no items.
$(\preceq, S)$			A partially ordered set (*poset*) is a relational structure for which $\preceq$ is a partial ordering over S [40]. $\preceq$ over S might be represented as a lattice, referred to as the Hasse diagram.
$\top_S$	$\forall a \in S . a \preceq \top_S$	S	Given a poset $(\preceq, S)$, $\top_S$ is the unique greatest element of S.
$\complement C$	$\mathcal{U} \backslash C$	Set	*Complement* set: given an universe $\mathcal{U}$, the complement returns all of the elements that do not belong to C.
$\bigtimes_{\kappa \in K} f(\kappa)$	$f(\kappa_1) \times \cdots \times f(\kappa_n)$	$\mathrm{dom}(f)^{\lvert K \rvert}$	*Generalised cross product* for ordered sets K where $\kappa_1 \prec \cdots \prec \kappa_n$
$\lvert C \rvert$	$\sum_{c \in C} 1$	$\mathbb{N}$	The *cardinality* of a finite set indicates the number of contained items.
$\wp(C)$	$\{T \mid T \subseteq C\}$	Set	The *powerset* of C is the set whose elements are all of the subsets of C.
		XES Model & $\mathrm{LTL_f}$	
Σ		Set	Finite set of activity labels
K		Set	Finite set of ordered (payload) keys, κ
V		Set	Finite set of (payload) values
p	$[\kappa_1 \mapsto v_1, \dots]$	V^K	Tuple (or finite function) mapping keys $\kappa_1 \in K$ to values in $v_1 \in V$
$\perp$	$\perp \notin V$		`NULL` value
σ^i_j	$\langle p, a \rangle$	$\Sigma \times V^K$	Event
σ^i	$\sigma^i_1, \dots, \sigma^i_n$	Sequence	Trace, sequence of temporarily ordered events.
$\mathcal{L}$	$\{\sigma^1, \dots, \sigma^m\}$	Set	Log, set of traces.
β		$\Sigma \leftrightarrow \{1, \dots, \lvert\Sigma\rvert\}$	Bijection mapping each activity label to its unique identifier.
φ	Equation (1)	Expression	An $\mathrm{LTL_f}$ expression.
$\vDash$			$\Gamma \vDash \varphi$ denotes that φ is *satisfied* for the world/environment Γ.
		$\mathtt{xt}\mathrm{LTL_f}$	
ψ	Section 3.2	Expression	eXTended $\mathrm{LTL_f}$ Algebra expression.
$A(k)/T(k)/M(h,k)$		ω	*Marks* associated with activation/target/matching conditions.
ρ	$\{ \langle i, j, L \rangle, \dots \}$	$\Omega = \{\wp(\mathbb{N} \times \mathbb{N} \times S) \mid S \in \wp(\omega)\}$	*Intermediate representation* returned by each $\mathtt{xt}\mathrm{LTL_f}$ operator
$T[i]$		$T[i] \in T$	Accessing the i-th record of a sequence T.
$\Theta(x,y)$		Binary Predicate	*Correlation condition* between activated and targeted events.
$\Theta^{-1}(y,x)$	$\Theta(x,y)$	Binary Predicate	*Inverted/Flipped* correlation condition.
True		Binary Predicate	Always-true binary predicate.
$E^i_\Theta(M_1, M_2)$	Equation (S1)	Algorithm 7	*Existential matching condition* for which there exists at least one event in M_1, M_2 providing a match.
$A^i_\Theta(M_1, M_2)$	Equation (S2)	Algorithm 9	*Universal matching condition* returning a non-empty set if each event expressed in the maps M_1, M_2 provides a match.
$\mathcal{T}^{F,i}_\Theta(M_1, M_2)$	Equation (S3)	$\mathcal{T}^{F,i}_\Theta(M_1, M_2) \in \wp(\omega) \cup \{\mathbf{False}\}$	*Testing functor* returning **False** iff., despite the maps containing activated and targeted events, the matching condition $F^i_\Theta(M_1, M_2)$ is empty. It returns $F^i_\Theta(M_1, M_2)$ otherwise.
		Pseudocode	
$\uparrow$			Null pointer or terminated iterator.
Iterator(ρ)		Pointer	On ρ non-empty, it returns the iterator pointing to the first record in ρ
`current`(it)		Dereference	Element pointer by the pointer/iterator it.
LowerBound(d, b, e, v)		Binary Search	Given a beginning b and end e iterator range within a sequential and sorted data structure by increasing order, LowerBound returns either the first location in this range pointing at a value greater or equal to v or e otherwise.
UpperBound(d, b, e, v)		Binary Search	Given a beginning b and end e iterator range within a sequential and sorted data structure by increasing order, UpperBound returns either the first location in this range pointing to a value strictly less to v or e otherwise.
		Time Complexity	
ϵ		$\mathbb{N}$	Maximum trace length.
ℓ		$\mathbb{N}$	Maximum length of the third component of the intermediate representation.

https://en.cppreference.com/w/cpp/algorithm/lower_bound as accessed the 5 March 2023. https://en.cppreference.com/w/cpp/algorithm/upper_bound as accessed the 5 March 2023.

3. Logical Model

Differently from our previous work [4], we provide a full definition of the (logical) model, thus describing the relational schema and how such tables are instantiated in order to fully represent the original log $\mathcal{L}$ (Section 3.1). This is a required preliminary step, as this will provide the required background to understand the definitions for the $\mathtt{xt}\,\mathrm{LTL}_f$ *operators (Section 3.2). These operators,*

differently from the LTL_f *ones, are defined over the aforementioned model and assess the satisfiability of multiple traces loaded in such a model.*

The discussion on how such tables are loaded and indexed is postponed when discussing the physical model (Section 4), as well as the different algorithms associated with the different operators (Section 6).

3.1. Model Definition

KnoBAB provides a tabular (i.e., relational) representation of the log $\mathcal{L}$, in order to efficiently query it through tailored relational operators ($\texttt{xtLTL}_\text{f}$). If the log does not contain data payloads, the entire log can be represented in two relational tables, $\text{CountingTable}_\mathcal{L}$(`Activity`, `Trace`,`Count`) and $\text{ActivityTable}_\mathcal{L}$(`Activity`,`Trace`,`Event`,`Prev`,`Next`). While the former can efficiently assess how many events in the same given trace share the same activity label, the latter allows a faithful reconstruction of the activity label associated with the traces. In particular, we use the former to assess whether a trace contains a given activity label at all. Such tables are then defined as follows:

Definition 1 (CountingTable)**.** *Given a log* $\mathcal{L}$*, the* $\textit{CountingTable}_\mathcal{L}$(`Activity`,`Trace`,`Count`) *counts for each trace in* $\mathcal{L}$ *how many times each activity label occurs. More formally:*

$$\textit{CountingTable}_\mathcal{L} = \left[\langle \beta(\mathsf{a}), i, |\{\sigma^i_j \in \sigma^i | \sigma^i_j = \langle \mathsf{a}, p \rangle\}| \rangle \;\middle|\; \mathsf{a} \in \Sigma, \sigma^i \in \mathcal{L} \right]$$

A record $\langle \beta(\mathsf{a}), i, n \rangle$ *states that the i-th trace from the log* $\sigma^i \in \mathcal{L}$ *contains n occurrences of* a*-labelled events with id* $\beta(\mathsf{a})$*.*

Definition 2 (ActivityTable)**.** *Given a log* $\mathcal{L}$*, the* $\textit{ActivityTable}$ (`Activity`,`Trace`,`Event`,`Prev`,`Next`) *lists all of the possible events occurring in each log trace, where* `Prev` (π) *and* `Next` (ϕ) *are offsets pointing to the row representing the immediately preceding or following event in the trace if any. More formally:*

$$\textit{ActivityTable}_\mathcal{L} = \left[\langle \beta(\mathsf{a}), i, j, \pi, \phi \rangle \;\middle|\; \mathsf{a} \in \Sigma, \sigma^i \in \mathcal{L}, \sigma^i_j \in \sigma^i, \sigma^i_j = \langle \mathsf{a}, p \rangle \right]$$

A record $\langle \beta(\mathsf{a}), i, j, \pi, \phi \rangle$ *states that the j-th event of the i-th log trace* ($\sigma^i_j \in \sigma^i$, $\sigma^i \in \mathcal{L}$) *has an activity label* a *and that its preceding and following events (if any) are respectively located on the* π*-th and* ϕ*-th record of the same table. Each record of this table should also satisfy the following integrity constraints:*

- $(j = 1 \wedge \pi = \bot) \vee (\exists h, \pi', \phi' . \langle h, i, j-1, \pi', \phi' \rangle \in \textit{ActivityTable}_\mathcal{L}[\pi])$;
- $(j = |\sigma^i| \wedge \phi = \bot) \vee (\exists h, \pi', \phi' . \langle h, i, j+1, \pi', \phi' \rangle \in \textit{ActivityTable}_\mathcal{L}[\phi])$

Please observe that `Prev` and `Next` are computed after bulk inserting while loading and indexing the data (see LoadingAndIndexing from Algorithm 2). If a log is associated with either trace or event payloads, we must store for each payload the values associated with keys k in an $\text{AttributeTable}^k_\mathcal{L}$(`Activity`,`Value`,`Offset`), where `Offset` points to the event described in the $\text{ActivityTable}_\mathcal{L}$.

Definition 3 (AttributeTable)**.** *Given a log* $\mathcal{L}$*, for each attribute* $\kappa \in K$ *associated with at least one value in a payload, we define a table* $\textit{AttributeTable}^\kappa_\mathcal{L}$(`Activity`,`Value`,`Offset`) *associating each value to the pertaining event's payload as follows:*

$$\textit{AttributeTable}^\kappa_\mathcal{L} = \left[\langle \beta(\mathsf{a}), p(\kappa), \pi \rangle \;\middle|\; \sigma^i \in \mathcal{L}, \sigma^i_j \in \sigma^i, \sigma^i_j = \langle \mathsf{a}, p \rangle, p(\kappa) \neq \bot \right]$$

A record $\langle \beta(a), v, \pi \rangle$ *states that the event* $\sigma^i_j = \langle \mathsf{a}, p \rangle$ *stored in the* $\textit{ActivityTable}$ *associated with the* π*-th offset contains a payload* p *associating* κ *to a value* v ($p(\kappa) = v$)*.*

Please observe that, similarly to the former table, the offset π is also computed while loading and indexing the data: this is discussed in greater detail in Section 4.2.2.

Algorithm 2 Populating the Knowledge Base (Section 4.2)

procedure BULKINSERTION($\mathcal{L}$)
  $\Sigma, K \leftarrow \emptyset$
  for all $\sigma^i \in \mathcal{L}$ **do**
    $\Sigma \leftarrow \Sigma \cup \{\mathsf{a}\}$
    for all $\sigma^i_j = \langle \mathsf{a}, p \rangle \in \sigma^i$ **do**
      $\texttt{CountBulkMap}[\beta(\mathsf{a})][i] = \texttt{CountBulkMap}[\beta(\mathsf{a})][i] + 1$
      $\texttt{ActToEventBulkVector}[\beta(\mathsf{a})].\texttt{put}(\langle i, j \rangle)$
      $\texttt{TraceToEventBulkVector}[i][j] = j$
      for all $\kappa \in \mathrm{dom}(p)$ **do**
        $K \leftarrow K \cup \{\kappa\}$
        $\texttt{AttBulkMap}_k[\beta(\mathsf{a})][p(\kappa)].\texttt{put}(\langle i, j \rangle)$
      end for
    end for
  end for

procedure LOADINGANDINDEXING($\mathcal{L}$)
  $\texttt{actTableOffset} \leftarrow 1$
  for all $\beta(\mathsf{a}) \in \{1, \ldots, |\Sigma|\}$ **do**
    $\text{ActivityTable}_{\mathcal{L}}.\texttt{primary\_index}[\beta(\mathsf{a})] \leftarrow \texttt{actTableOffset}$
    for all $\sigma^i \in \mathcal{L}$ **do**
      $\text{CountingTable}_{\mathcal{L}}.\texttt{load}(\langle \beta(\mathsf{a}), i, \texttt{CountBulkMap}[\beta(\mathsf{a})][i] \rangle)$
    end for
    for all $\langle i, j \rangle \in \texttt{ActToEventBulkVector}[\beta(\mathsf{a})]$ **do**
      $\text{ActivityTable}_{\mathcal{L}}.\texttt{load}[\langle \beta(\mathsf{a}), i, j, \uparrow, \uparrow \rangle]$
      $\texttt{TraceToEventBulkVector}[i][j] = \texttt{actTableOffset}$
      $\texttt{actTableOffset} \leftarrow \texttt{actTableOffset} + 1$
    end for
  end for
  for all $\kappa \in K$ **and** $\beta(\mathsf{a}) \in \{1, \ldots, |\Sigma|\}$ **do**
    $\texttt{begin} \leftarrow |\text{AttributeTable}^{\kappa}_{\mathcal{L}}|, \quad \texttt{map} \leftarrow \{\}$
    for all $\langle \nu, lst \rangle \in \texttt{AttBulkMap}_k[\beta(\mathsf{a})]$ **and** $\langle i, j \rangle \in lst$ **do** ▷ $\sigma^i_j = \langle \mathsf{a}, p \rangle$ with $\nu = p(\kappa)$
      $\texttt{offset} \leftarrow \texttt{TraceToEventBulkVector}[i][j]$
      $\text{AttributeTable}^{\kappa}_{\mathcal{L}}.\texttt{load}(\langle \beta(\mathsf{a}), \nu, \texttt{offset} \rangle)$
      $\text{AttributeTable}^{\kappa}_{\mathcal{L}}.\texttt{secondary\_index}[\texttt{offset}] \leftarrow |\text{AttributeTable}^{\kappa}_{\mathcal{L}}|$
    end for
    $\text{AttributeTable}^{\kappa}_{\mathcal{L}}.\texttt{primary\_index}[\beta(\mathsf{a})] \leftarrow \langle \texttt{begin}, |\text{AttributeTable}^{\kappa}_{\mathcal{L}}| \rangle$
  end for
  for all $\sigma^i \in \mathcal{L}$ **and** $\sigma^i_j \in \sigma^i$ **do**
    $\texttt{curr} \leftarrow \texttt{TraceToEventBulkVector}[i][j]$
    if $j = 1$ **then**
      $\text{ActivityTable}_{\mathcal{L}}.\texttt{secondary\_index}[i] \leftarrow \langle \texttt{curr}, \texttt{TraceToEventBulkVector}[i][|\sigma^i|] \rangle$
    else
      $\text{ActivityTable}_{\mathcal{L}}[\texttt{curr}](\texttt{Prev}) \leftarrow \texttt{TraceToEventBulkVector}[i][j-1]$
    end if
    if $j < |\sigma^i|$ **then**
      $\text{ActivityTable}_{\mathcal{L}}[\texttt{curr}](\texttt{Next}) \leftarrow \texttt{TraceToEventBulkVector}[i][j1]$
    end if
  end for

function RECONSTRUCTLOG($\mathcal{L}$)
  $\mathcal{L}' \leftarrow \emptyset$
  for all $\langle i, \langle \texttt{begin}, \texttt{end} \rangle \rangle \in \text{ActivityTable}_{\mathcal{L}}.\texttt{secondary\_index}$ **do**
    $\varsigma^i \leftarrow []; \quad j \leftarrow 1$
    repeat
      $r \leftarrow \text{ActivityTable}_{\mathcal{L}}[\texttt{begin}]$
      $\mathsf{a} \leftarrow \beta^{-1}(r(\texttt{Activity}))$
      $p \leftarrow \{\}$
      for all $\kappa \in K$ **s.t.** $\exists o.\ \langle \texttt{begin}, o \rangle \in \text{AttributeTable}_k.\texttt{secondary\_index}$ **do**
        $p(\kappa) \leftarrow \text{AttributeTable}_k[o](\texttt{Value})$ ▷ $\text{AttributeTable}_k[o](\texttt{Offset}) = \texttt{begin}$
      end for
      $\varsigma^i_j \leftarrow \langle \mathsf{a}, p \rangle; \quad \sigma^i.\texttt{put}(\varsigma^i_j)$
      $\texttt{begin} \leftarrow r(\texttt{Next}); \quad j \leftarrow j + 1$
    until $\texttt{begin} \neq \uparrow$
    $\mathcal{L}'.\texttt{put}(\varsigma^i)$
  end for
  return $\mathcal{L}'$

Example 8. *Figure 2 provides a graphical depiction of the tables storing our data. The records are also sorted by ascending order induced by the first three cells of each record, as required by our Physical Database Design (Section 4). For representation purposes, the first cell of each row shows the activity label* a *rather than its associated unique id* $\beta(\mathsf{a})$.

3.2. *eXTended* LTL$_f$ *Algebra (*$\mathtt{xt}$LTL$_f$*)*

We extend the operators provided in our previous work [4] into more minimal ones, thus better describing the data access on the relational model. Furthermore, we provide a full formal characterisation for each of these operators via their access to the aforementioned relational tables. Please observe that, similarly to the relational algebra, each $\mathtt{xt}$LTL$_f$ *operator might come with different possible algorithms [42], which are discussed in Section 6.*

Our operators, assessing the behaviour of non-empty traces, come in two flavours: timed and untimed. While the former are marked by a τ and return all of the traces' events for which a given condition holds, the latter guarantee that such a condition will hold any time from the beginning of the trace. Furthermore, these operators assess the satisfiability of all the log traces simultaneously and not only one trace at a time as per LTL$_\mathrm{f}$.

Each $\mathtt{xt}$LTL$_\mathrm{f}$ operator returns a nested relational table ρ with schema IntermediateResult(`Trace,Event,MarkList(Mark)`) implemented as an ordered set of triplets $\langle i, j, L\rangle$, where each triplet states that an event σ^i_j from trace σ^i satisfies a condition specified by the returning operator. If L (`MarkList(Mark)`) is not empty, the current event σ^i_j might have observed events σ^i_k and σ^i_h satisfying either an activation ($A(k) \in L, k \geq j$), a target ($T(k) \in L, k \geq j$), or a correlation condition ($M(h,k) \in L, k, h \geq j$). The nested relation L is implemented as a vector ordered by *mark* type and referenced event id. ρ is implemented as a vector and sorted by increasing `Trace` and `Event` id, as sorted vectors guarantee efficient intersection and union operations, as well as efficient event counting within the same trace through linear scanning. Binary operators associated with a non-**True** binary predicate Θ return matching/correlation conditions $M(h,k) \in L$ if at least one activation and one target condition were matched, depending on the definition of the operator. As we are going to see next, if the output comes from a base operator, as defined in the next section, L might contain a single activation or target corresponding to the immediately returned event.

3.2.1. Base Operators

First, we discuss the base operators directly accessing the tables. These might have an associated marker specifying whether the event of interest is considered an activation (A) or a target (T) condition; if none is required, the mark can be omitted from the operator. The $\mathsf{Activity}^{\tau}(\mathsf{a})^{\mathcal{L}}_{A/T}$ operator lists all of the events associated with an activation label a. As the $\mathsf{ActivityTable}_{\mathcal{L}}$ directly provides this information, this operator is defined as follows:

$$\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a}) = \{\ \langle i, j, \{A/T(j)\}\rangle \mid \exists \pi, \phi.\ \langle \beta(\mathsf{a}), i, j, \pi, \phi\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \}$$

We can also make similar considerations for single *elementary interval* representable as an LTL$_\mathrm{f}$ *compound condition* $\mathsf{a} \wedge \mathit{lower} \leq \kappa \leq \mathit{upper}$, which can be run as a single range query over an $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$. As each of its records has an offset π to the $\mathsf{ActivityTable}_{\mathcal{L}}$, this resolves the trace id and event id information required for the intermediate result. This operator can therefore be formalised as follows:

$$\mathsf{Compound}^{\mathcal{L},\tau}_{A/T}(\mathsf{a}, \kappa, [\mathit{lower}, \mathit{upper}]) = \Big\{\ \langle i, j, \{A/T(j)\}\rangle \ \Big|\ \exists \pi, \pi', \phi, v.\ \mathit{lower} \leq v \leq \mathit{upper}, \langle \beta(\mathsf{a}), v, \pi\rangle \in \mathsf{AttributeTable}^{\kappa}_{\mathcal{L}},\ \mathsf{ActivityTable}_{\mathcal{L}}[\pi] = \langle \beta(a), i, j, \pi', \phi\rangle\ \Big\}$$

If we want to list all of the initial (or terminal) events of a trace, we can directly access the ActivityTable and provide a linear scan over the number of the possible traces through its associated secondary index. If we are not interested in whether the trace starts with a specific activity label, then we can define the $\mathsf{First}^{\mathcal{L},\tau}_{A}$ (and $\mathsf{Last}^{\mathcal{L},\tau}_{A}$) operators as follows:

$$\mathsf{First}^{\mathcal{L},\tau}_{A} = \{\ \langle i, 1, \{A(1)\}\rangle \mid \exists \mathsf{a}, \phi.\ \langle \beta(\mathsf{a}), i, 1, \bot, \phi\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \}$$

$$\mathsf{Last}^{\mathcal{L},\tau}_{A} = \Big\{\ \langle i, |\sigma^i|, \{A(|\sigma^i|)\}\rangle \ \Big|\ \exists \mathsf{a}, \pi.\ \langle \beta(\mathsf{a}), i, |\sigma^i|, \pi, \bot\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \Big\}$$

On the other hand, Init (and Ends) are the specific refinements of the former operators if we are also interested in retrieving events with a specific activity label. These can be defined as follows:

$$\mathsf{Init}_A^{\mathcal{L}}(\mathsf{a}) = \{\ \langle i, 1, \{A(1)\}\rangle \mid \exists \phi.\ \langle \beta(\mathsf{a}), i, 1, \bot, \phi\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \}$$

$$\mathsf{Ends}_A^{\mathcal{L}}(\mathsf{a}) = \left\{\ \langle i, 1, \{A(|\sigma^i|)\}\rangle \;\middle|\; \exists \pi.\ \langle \beta(\mathsf{a}), i, |\sigma^i|, \pi, \bot\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \right\}$$

Given a natural number n, $\mathsf{Exists}(\mathsf{a}, n)_A^{\mathcal{L}}$ lists the traces containing at least n events with an activity label a. As $\mathsf{Absence}(\mathsf{a}, n)_A^{\mathcal{L}}$ is the substantial negation of the former, this lists the traces containing at most $n-1$ events with an activity label a. Please observe that these operators directly provide the formal semantics for the homonym Declare template. As the CountingTable precisely contains the counting information required to solve this query, these operators can be formalised as follows for $n \in \mathbb{N}_{>0}$:

$$\mathsf{Exists}_A^{\mathcal{L}}(\mathsf{a}, n) = \{\ \langle i, 1, \{A(1)\}\rangle \mid \exists m \geq n.\ \langle \beta(\mathsf{a}), i, m\rangle \in \mathsf{CountingTable}_{\mathcal{L}}\ \}$$

$$\mathsf{Absence}_A^{\mathcal{L}}(\mathsf{a}, n) = \{\ \langle i, 1, \{A(1)\}\rangle \mid \exists m < n.\ \langle \beta(\mathsf{a}), i, m\rangle \in \mathsf{CountingTable}_{\mathcal{L}}\ \}$$

The following paragraph shows how these last two operators can be generalised for counting the salient event information returned by any sub-expression returning an operand ρ.

3.2.2. Unary Operators

The unary $\mathtt{xtLTL_f}$ operators come in two flavours: the first ones extend some of the former operators for compound conditions or atoms not necessarily associated with activity labels, while the second ones directly extend the unary operators from $\mathrm{LTL_f}$.

Base Operators' generalisations. We extend the definition of Init/Ends or Exists/Absence for any possible set of events of interest listed in an intermediate result ρ, not necessarily associated with the same activity label. We first define Exists and Absence operator as such: instead of exploiting the counting table, we now actually need to count the events returned in ρ for each trace and return an intermediate result triplet iff. they satisfy the counting condition. These can be then defined as follows for $n \in \mathbb{N}_{>0}$:

$$\mathsf{Exists}_n(\rho) = \left\{\ \langle i, 1, \cup_{\langle i,j,L_j\rangle \in \rho} L_j\rangle \;\middle|\; n \leq |\{\langle i, j, L'\rangle \in \rho\}|\ \right\}$$

$$\mathsf{Absence}_n(\rho) = \left\{\ \langle i, 1, \cup_{\langle i,j,L_j\rangle \in \rho} L_j\rangle \;\middle|\; n > |\{\langle i, j, L'\rangle \in \rho\}|\ \right\}$$

Similarly, while the operators accessing the CountingTable (Exists/Absence) return the result by linearly scanning such a table, their generalised counterparts require scanning their operand ρ as returned from a subexpression of choice, and then creaming them off depending on how many events per trace were in ρ. As we might observe, we might exploit the previously provided operators when we want to evaluate conditions only associated with activity labels, while we might need to exploit the former if we are interested in results associated with compound conditions whose evaluation is returned in ρ.

Finally, we refine Init and Ends for a given operand ρ, to keep only the events at the beginning or end of a given trace:

$$\mathsf{Init}(\rho) = \{\ \langle i, j, L\rangle \in \rho \mid j = 1\ \}$$

$$\mathsf{Ends}(\rho) = \left\{\ \langle i, 1, L\rangle \;\middle|\; \langle i, |\sigma^i|, L\rangle \in \rho\ \right\}$$

Further details on our intended notion of these operators' generality if compared to the corresponding base operators can be found in Appendix A.1.

$\mathrm{LTL_f}$extensions. The unary $\mathtt{xtLTL_f}$ operators work differently from the corresponding ones in $\mathrm{LTL_f}$: while the latter compute the semantics from the first occurring operator

appearing in the formula towards the leaves, the former assume to receive intermediate results from the leaves.

This structural difference also imposes an explicit distinction between timed and untimed operators. This is required as each operator is completely agnostic from the semantics associated with the upstream operator, and therefore the downstream operator has to combine the incoming intermediate results appropriately. This motivates why LTL_f operators do not have to provide such an explicit distinction from their syntactical standpoint.

Such a premise motivates the counter-intuitive definition of the timed Next^τ operator if compared to the homonym in LTL_f: as this needs to return the events for which desired temporal constraints happen immediately after them, it needs to assume that the desired forthcoming temporal behaviour is the one received as an input ρ, for which all the events preceding the ones listed in ρ are the ones of interest. As per the previous statement, it also follows that this operator shall never possess an equivalent untimed flavour. From these considerations, Next^τ is formally defined as follows:

$$\mathsf{Next}^\tau(\rho) = \{\ \langle i, j-1, L\rangle \mid \langle i, j, L\rangle \in \rho, j > 1\ \}$$

where L fulfils the role of preserving the information of the events satisfying an activation, target, or correlation condition independently from the event stated in the second component of the intermediate representation record. Therefore, $\langle i, j, L\rangle$ shall be interpreted as follows: σ^i_j witnesses the satisfaction of any activation, target, or correlation condition by the events collected in L.

We now discuss the definition of "globally". As per previous considerations, checking that all of the events in a trace satisfy a given condition corresponds to retrieving all of the events satisfying such a condition, for then counting if the length of the returned events corresponds to the trace length. Similarly, the timed version of the same operator shall test the same condition for each possible event and return the points in the trace after which the desired condition always happens in the future. These operators are therefore defined as follows:

$$\mathsf{Globally}^\tau(\rho) = \left\{ \langle i, j, \cup_{\substack{j \le k \le |\sigma^i| \\ \langle i,k,L_k\rangle \in \rho}} L_k\rangle \;\middle|\; \langle i, j, L_j\rangle \in \rho,\ |\sigma^i| - j + 1 = \left|\{\langle i, k, L_k\rangle \in \rho | j \le k \le |\sigma^i|\}\right| \right\}$$

$$\mathsf{Globally}(\rho) = \left\{ \langle i, 1, \cup_{\langle i,j,L_k\rangle \in \rho} L_k\rangle \;\middle|\; |\sigma^i| = \left|\{\langle i, k, L_k\rangle \in \rho\}\right| \right\}$$

The operators expressing the eventuality that a condition shall happen in the future undergo similar considerations, with the only difference that these do not require to test that all of the trace events from a given point in time will satisfy a given condition, as it suffices that at least one event will satisfy it. The Future operator with its timed counterpart are then formally defined as follows:

$$\mathsf{Future}^\tau(\rho) = \left\{ \langle i, j, \cup_{\substack{j \le k \le |\sigma^i| \\ \langle i,k,L_k\rangle \in \rho}} L_k\rangle \;\middle|\; \exists h \ge j, L.\ \langle i, h, L\rangle \in \rho \right\}$$

$$\mathsf{Future}(\rho) = \left\{ \langle i, 1, \cup_{\langle i,k,L_k\rangle \in \rho} L_k\rangle \;\middle|\; \exists j, L.\ \langle i, j, L\rangle \in \rho \right\}$$

Timed and untimed negations are implemented dissimilarly by design. While the timed negation returns all of the events that are in the log but which were not returned in the previous computation ρ, the untimed version returns the traces containing no events associated with the provided input. These operators are therefore defined as follows:

$$\mathsf{Not}^\tau(\rho) = \{\ \langle i, j, \emptyset\rangle \mid (\nexists L.\ \langle i, j, L\rangle \in \rho) \wedge \exists \alpha, \pi, \phi.\ \langle \alpha, i, j, \pi, \phi\rangle \in \mathsf{ActivityTable}_\mathcal{L}\ \}$$

$$\mathsf{Not}(\rho) = \{\ \langle i, 1, \emptyset\rangle \mid (\nexists j, L.\ \langle i, j, L\rangle \in \rho) \wedge \exists \alpha, j, \pi, \phi.\ \langle \alpha, i, j, \pi, \phi\rangle \in \mathsf{ActivityTable}_\mathcal{L}\ \}$$

3.2.3. Binary Operators

Differently from the LTL_f binary operators, the $\texttt{xt}\text{LTL}_\text{f}$ binary operators are specifically tailored to express data correlation conditions Θ between activation and target payloads. This requires that one of the two operands, either ρ or ρ', returns activated events while the other provides targeted ones. Supplement I discusses the formal definition of predicates assessing whether an event $\langle i, j, L\rangle \in \rho$ matches with another event $\langle i, j', L'\rangle \in \rho'$ on the basis of their matched and activated events in L and L'. After this, we have the definition of our required binary operators.

The until operators work similarly to the other LTL_f-derived unary operators. The timed until returns all of the events within the trace satisfying the until condition, expressed by returning all of the "activated" events σ^i_j listed in the right operand (as they trivially satisfy the until condition) alongside all of the "targeted' events σ^i_j from the left operand with $k < j$ at a distance $j - k + 1$ from the second operand's event while guaranteeing that all the events in $\sigma^i_k, \ldots, \sigma^i_{j-1}$ appear in the first operand while satisfying the matching condition within this temporal window. The untimed version of this operator performs such considerations only from the beginning of the trace. These are defined as follows:

$$\begin{aligned}\mathsf{Until}^\tau_\Theta(\rho_1, \rho_2) = &\rho_2 \cup \\ &\Big\{ \langle i, k, \tau\rangle \,\Big|\, \exists j > k.\, \langle i, j, L\rangle \in \rho_2, (\forall k \le h < j.\, \langle i, h, L\rangle \in \rho_1), \\ &\qquad \tau := \mathcal{T}^{A,i}_\Theta([k \mapsto L]_{k\le h<j}, [h \mapsto L_h]_{k\le h<j, \langle i,h,L_h\rangle\in\rho_1}), \tau \neq \textbf{False}\Big\}\end{aligned}$$

$$\begin{aligned}\mathsf{Until}_\Theta(\rho_1, \rho_2) = &\{\, \langle i, j, L\rangle \in \rho_2 \mid j = 1 \,\} \cup \\ &\Big\{ \langle i, 1, \tau\rangle \,\Big|\, \exists j > 1, L.\, \langle i, j, L\rangle \in \rho_2, (\forall 1 \le k < j.\, \langle i, k, L_k\rangle \in \rho_1), \\ &\qquad \tau := \mathcal{T}^{A,i}_\Theta([k \mapsto L]_{i\le k<j}, [k \mapsto L_k]_{i\le k<j, \langle i,k,L_k\rangle\in\rho_1}), \tau \neq \textbf{False}\Big\}\end{aligned}$$

where $\mathcal{T}^{A,i}_\Theta$ performs (Please see Supplement I for more details.) the correlation tests and returns the set of the matches if any and, if no match was successful, it returns **False**. Differently from $\mathsf{Until}^\tau_\Theta$ and Until_Θ, the rest of the binary operators assume to receive "activated" (or "targeted") events from the left (right) operand. The timed conjunction states that a join condition effectively happens in a given event σ^i_j if both operands return such an event and their associated activation and target conditions match. Thus, we only care for activation and target conditions at the same event σ^i_j. For its untimed counterpart, we state that a trace satisfies the conjunction of events if at least one activation condition from the left operand matching with a target from the right operand, if any, exists; this corresponds to coalescing the activations and target conditions on the first event while requiring that at least one of them occurs. These two operators can then be defined as follows:

$$\mathsf{And}^\tau_\Theta(\rho_1, \rho_2) = \Big\{ \langle i, j, \tau\rangle \,\Big|\, \exists L_1, L_2.\, \langle i, j, L_1\rangle \in \rho_1, \langle i, j, L_2\rangle \in \rho_2, \tau := \mathcal{T}^{E,i}_\Theta([j \mapsto L_1], [j \mapsto L_2]), \tau \neq \textbf{False}\Big\}$$

$$\begin{aligned}\mathsf{And}_\Theta(\rho_1, \rho_2) = \Big\{ \langle i, 1, \tau\rangle \,\Big|\, &\exists j, j', L, L'.(\langle i, j, L\rangle \in \rho_1 \wedge \langle i, j', L'\rangle \in \rho_2), \\ &\tau := \mathcal{T}^{E,i}_\Theta([1 \mapsto \cup\{L_j |\, \langle i, j, L_j\rangle \in \rho_1\}], [1 \mapsto \cup\{L_j |\, \langle i, j, L_j\rangle \in \rho_2\}]), \\ &\tau \neq \textbf{False}\Big\}\end{aligned}$$

The disjunctive version of the timed conjunctive operator returns either the result of the conjunctive operator or the events that did not temporally match from each respective

operator. The only difference with its untimed version is that the latter merges all potential activation or target conditions from either of the two operands:

$$\begin{aligned}\mathsf{Or}^{\tau}_{\Theta}(\rho_1,\rho_2) = \mathsf{And}^{\tau}_{\Theta}(\rho_1,\rho_2) \cup \{\ \langle i,j,L\rangle \in \rho_1 \mid \nexists L'.\, \langle i,j,L'\rangle \in \rho_2\ \} \\ \cup \{\ \langle i,j,L\rangle \in \rho_2 \mid \nexists L'.\, \langle i,j,L'\rangle \in \rho_1\ \}\end{aligned}$$

$$\begin{aligned}\mathsf{Or}_{\Theta}(\rho_1,\rho_2) = \mathsf{And}_{\Theta}(\rho_1,\rho_2) \cup \{\ \langle i,1,\cup\{L|\exists j.\, \langle i,j,L\rangle \in \rho_1\}\rangle \mid \nexists j,L'.\, \langle i,j,L'\rangle \in \rho_2\ \} \\ \cup \{\ \langle i,1,\cup\{L|\exists j.\, \langle i,j,L\rangle \in \rho_2\}\rangle \mid \nexists j,L'.\, \langle i,j,L'\rangle \in \rho_1\ \}\end{aligned}$$

As we will see, the choice of characterizing Or with an E^i_{Θ} match while coalescing the activation and target conditions on the first trace event allows us to express the Choice template from Declare with one single operator while preserving its expected $\mathrm{LTL_f}$ semantics.

3.2.4. Derived Operators

Similarly to relational algebra, we can now compose some frequently occurring operators together for enhancing the overall time complexity associated with the execution of frequently appearing subqueries in Declare.

Appendix A.3 will show that computing these operators is equivalent to computing their semantically equivalent $\mathtt{xtLTL_f}$ expression containing multiple operators.

$$\begin{aligned}\mathsf{AndFuture}^{\tau}_{\Theta}(\rho_1,\rho_2) = \Big\{ \langle i,j,\tau\rangle \ \Big|\ \exists L.\, \langle i,j,L\rangle \in \rho_1, (\exists L', k \geq j.\, \langle i,k,L'\rangle \in \rho_2), \\ \tau := \mathcal{T}^{E,i}_{\Theta}([j \mapsto L], [j \mapsto \cup_{h\geq j, \langle i,h,L_h\rangle \in \rho_2} L_h]), \tau \neq \textbf{False}\Big\}\end{aligned}$$

$$\begin{aligned}\mathsf{AndGlobally}^{\tau}_{\Theta}(\rho_1,\rho_2) = \Big\{ \langle i,j,\tau\rangle \ \Big|\ \exists L.\, \langle i,j,L\rangle \in \rho_1, (\forall |\sigma^i| \geq k \geq j. \exists L'.\, \langle i,k,L'\rangle \in \rho_2), \\ \tau := \mathcal{T}^{A,i}_{\Theta}([j \mapsto L], [j \mapsto \cup_{h\geq j, \langle i,h,L_h\rangle \in \rho_2} L_h]), \tau \neq \textbf{False}\Big\}\end{aligned}$$

For easing the pseudocode readability, we can also define an $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}(p_i)$ operator computing the conjunction of all of the compound conditions characterizing each atom:

$$\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}(p_i) = \mathop{\mathsf{And}^{\tau}_{\textbf{True}}}_{\kappa\in K} \mathsf{Compound}^{\mathcal{L},\tau}_{A/T}(\mathsf{a},\kappa,[low_\kappa, up_\kappa]) \text{ s.t. } p_i := \mathsf{a} \wedge \bigwedge_{\kappa\in K} low_\kappa \leq \kappa \leq up_\kappa$$

Properties of the $\mathtt{xtLTL_f}$ Algebra. We furnish the previous definitions with some formal proofs, which, so as not to burden the reader, are postponed to the Appendix A. We show that $\mathtt{xtLTL_f}$ is as expressive as traditional $\mathrm{LTL_f}$, as we can show that each $\mathrm{LTL_f}$ expression evaluated over a finite and non-empty trace σ corresponds to an $\mathtt{xtLTL_f}$ expression evaluated over the representation of such a trace within the proposed logical model; as the proofs of Lemmas A5 and A6 in Appendix A.2 are constructive, they show the translation process from $\mathrm{LTL_f}$ formulæ to equivalent $\mathtt{xtLTL_f}$ expressions.

Next, we also show that the timed and untimed operators correspond to the intended semantics: that is, for each timed operator having a corresponding untimed operator if the former states that the timed formula is satisfied by the i-th trace starting from time j, it follows that the sub-trace of i starting from time j will satisfy the corresponding untimed formula. This shows the correctness of the untimed operators concerning their timed definitions (Lemma A7).

In Appendix A.3, we show that the Declare template Choice can be fully implemented by exploiting an untimed Or operator (Corollary A1) while the latter still abides to the rules of $\mathrm{LTL_f}$ semantics. We also motivate the need of the derived operators in terms of equivalence to the intended $\mathtt{xtLTL_f}$ expressions (Lemmas A9 and A10) as well as in terms of improved computational complexity (Section 6.4) and run time (Section 7.1). The latter

is discussed after describing the physical model in more detail alongside the algorithms associated with each operator, which is introduced in the following section.

4. Physical Database Design

This section shows how the defined model (Section 3.1) is represented in primary memory in terms of indices and data structures (Section 4.1). We also illustrate the algorithm loading a log in such representation of choice (Section 4.2).

4.1. Primary Memory Data Structures

At the time of the writing, KnoBAB is primarily an in-memory database. This is a common assumption in the conformance checking domain where most of the log datasets are quite compact and nicely fit in primary memory.

In order to be both memory and time efficient in our operations, the sub-record referring to the first three columns of both the $\mathsf{CountingTable}_{\mathcal{L}}$ and the $\mathsf{ActivityTable}_{\mathcal{L}}$ are fully stored in primary memory as an unsigned 64-bit unsigned integer, while the `Prev` and `Next` are more efficiently stored as pointers to the table records rather than being an offset. After sorting the $\mathsf{CountingTable}_{\mathcal{L}}$, we directly obtain the occurrence of each activity label a within the log by accessing the records in the range $[|\mathcal{L}| \cdot (\beta(\mathsf{a}) - 1) + 1,\ |\mathcal{L}| \cdot \beta(\mathsf{a})]$.

Indexing data structures, on the other hand, eases the access to the $\mathsf{ActivityTable}_{\mathcal{L}}$, as different traces might have different lengths, and activity labels might be differently distributed among the traces. Therefore, we exploit a clustered and sparse primary index for determining which is the first event associated with a given activity label; as the traces in such a table are represented as a doubly linked list, its secondary index maps each trace-id to a block that, in turn, points to the head (first event of the trace) and the tail (last event of the trace) of such a doubly linked list.

The deduplication of trace and event payloads in distinct $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ for each key κ follows the prescriptions of the query and memory-efficient representation of columnar-based storages [35]. In our implemnetation, such tables are sorted in ascending order by their three first columns. Each $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ is also associated with two indices: the clustered and sparse primary index maps each activity label's id $\beta(\mathsf{a})$ to the records referring to values contained in a-labelled events, and a dense secondary index associates an $\mathsf{ActivityTable}_{\mathcal{L}}$ record offset to an $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ record offset if and only if the event described in $\mathsf{ActivityTable}_{\mathcal{L}}$ has a payload containing a value associated with a key κ. While data range queries leverage the former, the latter is used for reconstructing the payload associated with a given event when identified by its offset in the $\mathsf{ActivityTable}_{\mathcal{L}}$. A relevant use case for doing so is the reconstruction of the event payload information while performing the Θ correlation condition, as well as reconstructing the original log leading to the loading of the internal database. RECONSTRUCTLOG function in Algorithm 2 shows the computation of the latter.

Example 9. *With reference to Figure 2, let us consider some events with activity label* Mastectomy *associated with an unique id* $\beta(\mathsf{Mastectomy}) = 3$. *The offsets for accessing the records in the* $\mathsf{CountingTable}_{\mathcal{L}}$ *defining the number of events per trace with such a label is* $[3 \cdot (3-1)+1, 3 \cdot 3] = [7, 9]$.

The $\mathsf{ActivityTable}_{\mathcal{L}}$*'s primary index allows the access to the first record within the table recording a* Mastectomy *event, i.e.,* $\mathsf{ActivityTable}_{\mathcal{L}}.\texttt{primary\_index}[\beta(\mathsf{Mastectomy})] = 7$ *; the index implicitly returns the last event associated with such an activity label by decreasing the offset to the following activity label by one, i.e.,* $\mathsf{ActivityTable}_{\mathcal{L}}.\texttt{primary\_index}[\beta(\mathsf{Mastectomy}) + 1] - 1 = 7$*: please remember that, if the activity label is such that* $\beta(\mathsf{a}) = |\Sigma|$*, then the final offset to be considered corresponds to the* $\mathsf{ActivityTable}_{\mathcal{L}}$ *size. This indicates that there exists only one event throughout the whole log associated with such an activity label. We will exploit this mechanism for returning the events associated with* $\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{Mastectomy})$*. As the seventh record of such a table refers to the third event of the first trace,* $\mathsf{Activity}^{\mathcal{L},\tau}_{A}(\mathsf{Mastectomy})$ *will then return* $\{\ \langle 1, 3, \{A(3)\}\rangle\ \}$.

Finally, we discuss how we can leverage $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$*'s primary indices for returning results associated with an* $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$ *operator. Let us consider atom* p_{12}*: we can see that this is associated with the elementary intervals* $\mathsf{Lumpectomy} \wedge \mathit{biopsy} = \textbf{\textit{true}}$ *and* $\mathsf{Lumpectomy} \wedge 50 \leq \mathit{CA\_15.3} < +\infty$*. By definition of the operator of interest, we then have that:*

$$\mathsf{Atom}^{\mathcal{L},\tau}_{A}(p_{12}) = \mathsf{And}^{\tau}_{\textbf{\textit{True}}}(\mathsf{Compound}^{\mathcal{L},\tau}_{A}(\mathsf{Lumpectomy} \wedge \mathit{biopsy} = \textbf{\textit{true}}), \mathsf{Compound}^{\mathcal{L},\tau}_{A}(\mathsf{Lumpectomy} \wedge \mathit{CA\_15.3} \geq 50))$$

The first $\mathsf{Compound}$ *operator will access the primary index from* $\mathsf{AttributeTable}^{\mathit{biopsy}}_{\mathcal{L}}$ *while the second one will access the one from* $\mathsf{AttributeTable}^{\mathit{CA\_15.3}}_{\mathcal{L}}$*. Then, the primary index of each table maps each activity to the offsets of the first and last record:* $\mathsf{AttributeTable}^{\mathit{biopsy}}_{\mathcal{L}}$.`primary_index` $[\beta(\mathsf{Lumpectomy})] = \langle 2,2 \rangle$ *and* $\mathsf{AttributeTable}^{\mathit{CA\_15.3}}_{\mathcal{L}}$.`primary_index`$[\beta(\mathsf{Lumpectomy})] = \langle 7,7 \rangle$*. Then, within these returned record offsets, we perform range queries respectively looking for records satisfying* $\mathit{biopsy} = \textbf{\textit{true}}$ *and* $50 \leq \mathit{CA\_15.3} < +\infty$*. All of the* $\mathsf{ActivityTable}_{\mathcal{L}}\kappa$*'s records satisfying these conditions point to the tenth record of the* $\mathsf{ActivityTable}_{\mathcal{L}}$ *referring to the third event of the third trace. Therefore,* $\mathsf{Atom}^{\mathcal{L},\tau}_{A}(p_{12})$ *returns* $\{ \langle 3,3,\{A(3)\} \rangle \}$*.*

4.2. Populating the Database

We discuss two subsequent steps for loading a log in our proposed relational model: we preliminarily sort the data by activity label id, event id, and values (Section 4.2.1) for then loading the sorted record in the tables while generating their primary and secondary indices (Section 4.2.2). These are computed in quasi-linear time with respect to the full log size.

4.2.1. Bulk Insertion

KnoBAB uses BULKINSERTION to pre-load the tables' data into an intermediate representation by pre-sorting it according to the ascending order induced by the first column of the tables of interest. Algorithm 2 shows the loading of the following three maps referring to the aforementioned tables. *(i)* `CountBulkMap` counts the occurrence of each activity label per track, implying that the absence of a trace identifier for a given $\beta(\mathsf{a})$ value presupposes the absence of a given activity label a within a trace; as the name suggests, we use this to later on populate the CountingTable. *(ii)* The `ActToEventBulkVector` prepares the insertion of sorted data in $\mathsf{ActivityTable}_{\mathcal{L}}$ by associating an activity label to each event and its associated trace containing it. *(iii)* Similarly to the `ActToEventBulkMap`, the $\mathtt{AttBulkMap}_k$ associates to each key κ the values $p(\kappa)$ for each event σ^i_j with payload p and activity label a, in order to prepare the insertion of sorted records in $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$. Please observe that, by construction, the set of pairs associated with each activity id $\beta(\mathsf{a})$ is already sorted by increasing trace and event id.

We also pre-allocate a `TraceToEventBulkVector` map (represented as a vector of vectors) which will later associate each event trace to an offset on the $\mathsf{ActivityTable}_{\mathcal{L}}$ where such event is stored. KnoBAB will later use this to calculate `Prev` and `Next` in the ActivityTable. After this, KnoBAB knows the number of the traces within the log $|\mathcal{L}|$, the length $|\sigma^j|$ for each trace σ^j, and the number of distinct activity labels $|\Sigma|$ is known, as well as their associated unique id $\beta(\mathsf{a})$ for each $\mathsf{a} \in \Sigma$. We can show that this procedure might be computed in quasi-linear time with respect to the full log size (Lemma S1).

4.2.2. Loading and Indexing

We continue our discussion with LOADINGANDINDEXING. *First*, we can iterate over the activity labels in ascending order of appearance (Line 17). All the tables including the $\mathsf{CountingTable}_{\mathcal{L}}$ have activity ids $\beta(\mathsf{a})$ as their first cell: by further iterating by increasing trace id, we can immediately orderly store the records in $\mathsf{CountingTable}_{\mathcal{L}}$ (Line 20).

Second, we start populating the $\mathsf{ActivityTable}_{\mathcal{L}}$ (Line 23) where each record is associated with an increasing offset of the table (Line 25). We can populate its primary index in order to point at the record representing the first event of the first trace with the currently considered

activity label. We store such information in the pre-allocated `traceToEventBulkVector` (Line 24), in order to later set the currently null pointer (↑) `Next` and `Prev` fields.

Third, we start populating each $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ for each key $\kappa \in K$ associated with at least one value in a payload: as per the previous discussion, each record associates the offset of event $\sigma^i_j = \langle \mathsf{a}, p \rangle$ in the $\mathsf{ActivityTable}_{\mathcal{L}}$ with a value $\nu = p(\kappa)$ and the activity label id $\beta(\mathsf{a})$ (Line 32). We also populate its secondary index by associating each event offset in the $\mathsf{ActivityTable}_{\mathcal{L}}$ to the current position in the $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ (Line 33). The last iteration finally populates each $\mathsf{ActivityTable}_{\mathcal{L}}$'s secondary index (Line 50) and sets the `Next` (Line 45) and `Prev` (Line 42) fields through the offset via `TraceToEventBulkVector`. After this, the relational database is fully loaded in primary memory. The overall time complexity grows linearly to the whole log representation (Lemma S2).

5. Query Processing and Optimisation

This section shows how a declarative model $\mathcal{M}$ is compiled to a query plan consisting of $\mathtt{xt}\,\mathrm{LTL}_f$ operators (Section 5.1) so it can be run (Section 5.2) on top of the primary memory data described in the previous section.

5.1. Query Compiler

The conversion of a declarative model $\mathcal{M}$ into its corresponding $\mathtt{xt}\,\mathrm{LTL}_f$ query plan is structured into three main phases. First, the atomisation pipeline calls the preliminary $\mathcal{D}_\varphi$-encoding from [25] for rewriting the data predicates appearing in each declarative clause as a disjunction of mutually exclusive atoms (Section 5.1.1). Second, we (ii) rewrite each Declare constraint as a $\mathtt{xt}\,\mathrm{LTL}_f$ formula from which we obtain a preliminary query plan represented as a Direct Acyclic Graph *(DAG) (Section 5.1.2). Third, we compute the scheduling order for the operators' execution over the DAG, thus preparing the execution to a potential parallel evaluation of the query (Section 5.1.3).*

5.1.1. Atomisation Pipeline

The atomisation pipeline (Algorithm 3) represents each activation and target condition as a set of disjunct atoms or activity labels. KnoBAB can always be configured in two ways: to either fully represent each possible activation (or target) condition with activity label a as a disjunction of atoms (or activity labels) if there exists at least one declarative clause where a is also associated with a non-trivial payload condition (strategy=`AtomizeEverything`), or to restrict atomisation to data conditions appearing in a clause (strategy=`AtomizeOnlyOnDataPredicates`). Both can be set through the AtomisationPipeline procedure in Algorithm 3. The $\mathcal{D}_\varphi$-encoding step guarantees that each activation or target condition will be associated with at least one atom or activity label. While the former approach will maximise the access to the $\mathsf{AttributeTable}_{\mathcal{L}}$, the latter will maximise the access to the $\mathsf{ActivityTable}_{\mathcal{L}}$. Correlation conditions do not undergo this rewriting step. We discuss the effects of each different strategy on the query runtime via empirical benchmarks in Section 7.4. We can show that this step has a polynomial complexity with respect to the model, key set, and element intervals' maximum size (Lemma S3).

Example 10. *With reference to Figure 3a, we might observe that, as no activation or target is ever associated with payload conditions, the atomisation pipeline will never express each activation or target condition as a disjunction of atoms, as no elementary interval is collected. Therefore, these will be only associated with activity labels.*

Example 11. *With reference to Example 6, Figure 2 shows the atomised version of the declarative model, where each activation and target condition is associated, in this case, with just one atom.*

Algorithm 3 Atomisation Pipeline (Section 5.1.1)

1: **procedure** AtomisationPipeline($\mathcal{M}$, strategy)
2: CollectIntervals($\mathcal{M}$) ▷ See Algorithm 1
3: $\mathcal{D}_\varphi$-encoding( ) ▷ See Algorithm 1
4: **for all** $\text{clause}_l(\mathsf{A}, p, \mathsf{B}, p')$ where $\Theta \in \mathcal{M}$ **do**
5: **if** p=**True and** (strategy=`AtomizeOnlyOnDataPredicates` **or** $ak(\mathsf{A}) = \{\mathsf{A}\}$) **then** ▷ Defining S_A for clause_l
6: clause_l.`left` $\leftarrow \{\mathsf{A}\}$
7: **else**
8: clause_l.`left` $\leftarrow ak(\mathsf{A}) \cap \mathsf{Atom}_{\mu,ad}(\mathsf{A}, p)$ ▷ Equation (2)
9: **end if**
10: **if** p′=**True and** (strategy=`AtomizeOnlyOnDataPredicates` **or** $ak(\mathsf{B}) = \{\mathsf{B}\}$) **then** ▷ Defining S_T for clause_l
11: clause_l.`right` $\leftarrow \{\mathsf{B}\}$
12: **else**
13: clause_l.`right` $\leftarrow ak(\mathsf{B}) \cap \mathsf{Atom}_{\mu,ad}(\mathsf{B}, p')$ ▷ Equation (2)
14: **end if**
15: **end for**

Example 12. *Continuing with Example 7, where we discussed the outcome of the $\mathcal{D}_\varphi$-encoding phase for a model $\mathcal{M}'$ in Equation (3), we obtain the following atomisation:*

$$\{\mathsf{Choice}_1(\mathit{left} = \{p_{12}, p_{17}\}, \mathit{right} = \{p_4, p_9\}),$$
$$\mathsf{Absence}_2(\mathit{left} = \{p_1, \ldots, p_5, p_{16}, \ldots, p_{20}\}, n = 1),$$
$$\mathsf{Absence}_3(\mathit{left} = \{p_1, p_3, p_5, p_6, p_8, p_{10}, p_{11}, p_{13}, p_{15}, p_{16}, p_{18}, p_{20}\}, n = 1)\}$$

5.1.2. Query Optimiser

The query optimiser consists of three steps: (i) loading the `xt` LTL_f *formulæ associated with each declarative clause at warm-up, (ii) exploiting the outcome of the Atomisation Pipeline to instantiate the* `xt` LTL_f *formulæ, (iii) and coalescing the single* `xt` LTL_f *into one compact abstract syntax DAG. Our query plan will not be represented as a tree as we merge as many nodes computing the same result as possible, thus computing the same sub-expression at most once.*

First, we load the translation map `xtTemplates` (Table 5) at warm-up through an external script providing the temporal semantics associated with the clauses of interest via partially-instantiated xtLTL_f expressions. Such representation also supports negated activation or target conditions, thus avoiding the need to compute a Not operator stripping the information of either activation or target conditions. These are marked in the previous table via set complementation, $\complement$. At the time of the writing, the scripts provide the xtLTL_f semantics for Declare templates. Future investigations will express other temporal declarative languages such as [43] in xtLTL_f, as well as other LTL_f extensions including "past" operators [17,19].

Second, we exploit the aforementioned map to convert each declarative clause into its xtLTL_f semantics ψ. If the clause is met for the first time, we proceed with its instantiation by recursively visiting ψ until the leaves are reached: at this level, we potentially replace the activation and target placeholders with the associated set of atoms. Disjunctions of atoms and activity labels associated with leaf nodes as returned by the previous pipeline are minimised by ensuring that each shared $\mathsf{Or}_{\mathbf{True}}$ computation across all of the atoms and activity labels is computed at most once. If an atom is met, we decompose it into its defining compound conditions (Line 14), thus guaranteeing that each compound condition is evaluated via $\mathtt{Compound}^{\mathcal{L},\tau}_{A/T}$ at most once across all of the atoms occurring in the xtLTL_f formula when running the query (Section 5.2.1).

Third, we complete the process by coalescing shared disjunct sub-expressions via a map (*queryCache*) guaranteeing that all of the equivalent sub-expressions are all replaced by just one instance of these. Finally, we associate each sub-expression referring to each clause to the final query operator representing the expression's root (*queryRoot*), either presenting an aggregation or a conjunctive query.

Table 5. Declare templates illustrated as their associated $\texttt{xt}\text{LTL}_\text{f}$ semantics. S_A (and S_T) denote the disjunction of collected atoms and activity labels (represented as sets) associated with the activation (and target) condition. The Atomisation Pipeline will return these sets. For declarative clauses that can be directly represented as $\texttt{xt}\text{LTL}_\text{f}$ operators, we might have two different possible operators depending on the atomisation result.

Exemplifying Clause (c_l)	**$\texttt{xt}\text{LTL}_\text{f}$ Semantics** $S_A = \{\mathsf{A}\}, S_T = \{\mathsf{B}\}$ $\mathsf{A}, \mathsf{B} \in \Sigma$ **Otherwise (e.g., Atomisation)**
$\mathsf{Init}(S_A)$	$\mathsf{Init}^{\mathcal{L}}_A(\mathsf{A})$ $\quad$ $\mathsf{Init}(S_A)$
Exists(S_A, n)	$\mathsf{Exists}^{\mathcal{L}}_A(\mathsf{A}, n)$ $\quad$ $\mathsf{Exists}_n(S_A)$
Absence$(S_A, n+1)$	$\mathsf{Absence}^{\mathcal{L}}_A(\mathsf{A}, n)$ $\quad$ $\mathsf{Absence}_{n+1}(S_A)$
Precedence(S_A, S')	$\mathsf{Or}_{\mathbf{True}}(\mathsf{Until}(\complement S', S_A), \textbf{Absence}(S', 1))$
$\mathsf{ChainPrecedence}(S_A, S_T)$ **where** Θ	$\mathsf{Globally}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\mathsf{Last}^{\mathcal{L},\tau}, \mathsf{Next}^{\tau}(\complement S_T)), \mathsf{And}^{\tau}_{\Theta}(\mathsf{Next}^{\tau}(S_A), S_T)))$
$\mathsf{Choice}(S_A, S_{A'})$	$\mathsf{Or}_{\mathbf{True}}(S_A, S_{A'})$
Response(S_A, S_T) **where** Θ	$\mathsf{Globally}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\complement S_A, \mathsf{AndFuture}^{\tau}_{\Theta}(S_A, S_T)))$
$\mathsf{ChainResponse}(S_A, S_T)$ **where** Θ	$\mathsf{Globally}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\complement S_A, \mathsf{And}^{\tau}_{\Theta}(S_A, \mathsf{Next}^{\tau}(S_T))))$
RespExistence(S_A, S_T) **where** Θ	$\mathsf{Or}_{\mathbf{True}}(\textbf{Absence}(S_A, 1), \mathsf{And}_{\Theta}(S_A, S_T))$
$\mathsf{ExclChoice}(S_A, S_{A'})$	$\mathsf{And}_{\mathbf{True}}(\mathsf{Or}_{\mathbf{True}}(\textbf{Exists}(S_A, 1), \textbf{Exists}(S_{A'}, 1)), \mathsf{Or}_{\mathbf{True}}(\textbf{Absence}(S_A, 1), \textbf{Absence}(S_{A'}, 1)))$
$\mathsf{CoExistence}(S_A, S_T)$ **where** Θ	$\mathsf{And}_{\mathbf{True}}(\textbf{RespExistence}(S_A, S_T)$ **where** Θ, **RespExistence**$(S_{A'}, S_{T'})$ **where** $\Theta^{-1})$ s.t. $S_{A'} = S_T$ and $S_{T'} = S_A$
$\mathsf{Succession}(S_A, S_T)$ **where** Θ	$\mathsf{And}_{\mathbf{True}}(\textbf{Precedence}(S_A, S'), \textbf{Response}(S_A, S_T)$ **where** $\Theta)$ s.t. $S' = S_T$
$\mathsf{ChainSuccession}(S_A, S_T)$ **where** Θ	$\mathsf{Globally}(\mathsf{And}^{\tau}_{\mathbf{True}}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\mathsf{Last}^{\mathcal{L},\tau}, \mathsf{Next}^{\tau}(\complement S_{T'})), \mathsf{And}^{\tau}_{\Theta^{-1}}(\mathsf{Next}^{\tau}(S_{A'}), S_{T'})),$ $\mathsf{Or}^{\tau}_{\mathbf{True}}(\complement S_A, \mathsf{And}^{\tau}_{\Theta}(S_A, \mathsf{Next}^{\tau}(S_T))))))$ s.t. $S_{A'} = S_T$ and $S_{T'} = S_A$
$\mathsf{AltResponse}(S_A, S_T)$ **where** Θ	$\mathsf{Globally}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\complement S_A, \mathsf{And}^{\tau}_{\Theta}(S_A, \mathsf{Next}^{\tau}(\mathsf{Until}^{\tau}_{\mathbf{True}}(\complement S_A, S_T)))))$
$\mathsf{AltPrecedence}(S_A, S_T)$ **where** Θ	$\mathsf{And}_{\mathbf{True}}(\textbf{Precedence}(S_A, S_T), \mathsf{Globally}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\complement S_A, \mathsf{And}^{\tau}_{\Theta}(S_A, \mathsf{Next}^{\tau}(\mathsf{Or}^{\tau}_{\mathbf{True}}(\mathsf{Until}^{\tau}(\complement S_A, S_T), \mathsf{Globally}^{\tau}(\complement S_A))))))))$
$\mathsf{NotCoExistence}(S_A, S_T)$ **where** Θ	$\mathsf{Not}(\mathsf{And}_{\Theta}(S_A, S_T))$
$\mathsf{NotSuccession}(S_A, S')$	$\mathsf{Globally}(\mathsf{Or}_{\mathbf{True}}(\complement S_A, \mathsf{AndGlobally}^{\tau}_{\mathbf{True}}(S_A, \complement S_T)))$

Example 13. *The model in Figure 3a, when compiled and associated with a conjunctive query, might produce the following xLTLf expression:*

$$\mathit{And}_{\mathit{True}}\Big(\mathit{Globally}\Big(\mathit{Or}^{\tau}(\mathit{Not}^{\tau}(\mathit{Activity}^{\mathcal{L},\tau}(\mathsf{rec})), \mathit{AndFuture}^{\tau}_{\mathit{True}}(\mathit{Activity}^{\mathcal{L},\tau}_{A}(\mathsf{rec})), \mathit{Activity}^{\mathcal{L},\tau}_{T}(\mathsf{weap}))\Big)\Big),$$
$$\mathit{And}_{\mathit{True}}\Big(\mathit{Absence}(\mathsf{iiot\_sh}, 1),$$
$$\mathit{Or}_{\mathit{True}}(\mathit{Activity}^{\mathcal{L}}(\mathsf{comm}), \mathit{Activity}^{\mathcal{L}}(\mathsf{act}))\Big)\Big)$$

We might observe that this expression cannot be further minimised, as there are neither shared atoms nor sub-expression in common. This can neither be achieved by rewriting $\mathit{Not}^{\tau}(\mathit{Activity}^{\mathcal{L},\tau}(\mathsf{rec}))$ *as* $\mathit{Or}^{\tau}_{\mathit{True}\ \mathsf{a}\neq\mathsf{rec}}\ \mathit{Activity}^{\mathcal{L},\tau}(\mathsf{a})$*, as the* comm *and* act *atoms associated with the choice clause are untimed, while the former rewriting only included timed* Activity *operators. As these two different flavours of operators do not necessarily return the same result, these nodes are not merged.*

Example 14. *With reference to Example 11 and Table 1, as the* Response *clause was associated with the same activation and target condition to* Succession*, the former is indeed a subquery of the latter. For this reason, these queries are fused together, thus guaranteeing that the result for* Response *is computed at most once. As the query root requires the computation of Max-SAT, this one is always going to be linked to the sub-expression being the representation of an original declarative clause. Green arrows in Figure 2 indicate operators' output shared among operators.*

Example 15. *This last example shows the effect of the reduction of the number of shared timed union operators at the leaf level. By recalling the atomised model discussed in Example 12, we need to represent each set of atoms as a timed disjunction of* Atom *operators. While doing so, we observe that* Choice *and the first* Absence *condition share atoms* p_4 *and* p_{17}*, while the two* Absence *clauses share all the atoms in* $\{p_1, p_3, p_5, p_{16}, p_{18}, p_{20}\}$*. Not ensuring that the timed unions associated with these last elements are computed only once will result in both multiple data access to our relational tables, as well as a considerable increase in run time as union operations are run twice. The detection and minimisation of such kind of shared sub-queries cannot be merely computed through a simple*

caching mechanism, thus requiring a more sophisticated algorithm for determining the maximal common subset shared among all of the possible sets of atoms (and potentially activity labels).

Algorithm 4 provides additional details on the implementation of such an approach. Line 11 refers to the *first* phase and shows the point in the code where we associate each negated leaf with the complementary set of atoms appearing after the decomposition process. With respect to the *second* phase, Line 65 shows the rewriting of the Declare clause into an intermediate $\texttt{xtLTL}_\text{f}$ by recursively visiting it in each of its operands until the leaves are reached (Line 5). If during this visit we meet a binary operator marked as being the "tester" for the correlation condition, we associate to it the Θ coming from the declarative clause (Line 4); otherwise, the operator keeps the default **True**. Concerning the leaves, for unary clauses, we consider the sole activation condition, while for binary clauses, we might also consider target conditions. If the leaf node is associated with an S_A (or S_T) containing more than one activity label or atom, we need to keep track of all of these while representing such a leaf as a disjunction of such atoms

(Lines 18–25). Next, we optimize each disjunction of atoms and activity labels in order to minimize the number of shared union computations (Line 48); such optimisation is performed after fully visiting the $\texttt{xtLTL}_\text{f}$ expression, thus ensuring that each appearing disjunction is actually collected (Line 69).

Line 14 shows where we collect atoms representing compound conditions while guaranteeing that its associated $\texttt{Compound}^{\mathcal{L},\tau}_{A/T}$ operator is computed only once, as well as decomposing it in its constituent compound conditions.

Finally, the method PUTINCACHE extends the *queryCache* map by guaranteeing that each distinct disjunction of atoms is also represented at most once within the query plan.

Example 16. *Figure 5 showcases the result of the application of such an algorithm while generating unique* $\texttt{xt}\,\text{LTL}_f$ *expressions. Such an algorithm also guarantees the non-repetition of single-leaf operators appearing in different clauses. Its upper box shows a query plan where common union operations are shared across sub-trees by representing each sub-tree at most once. These are actually represented in the query plan as opposed to the evaluation associated with the atoms, which is discussed in Supplement III.1.*

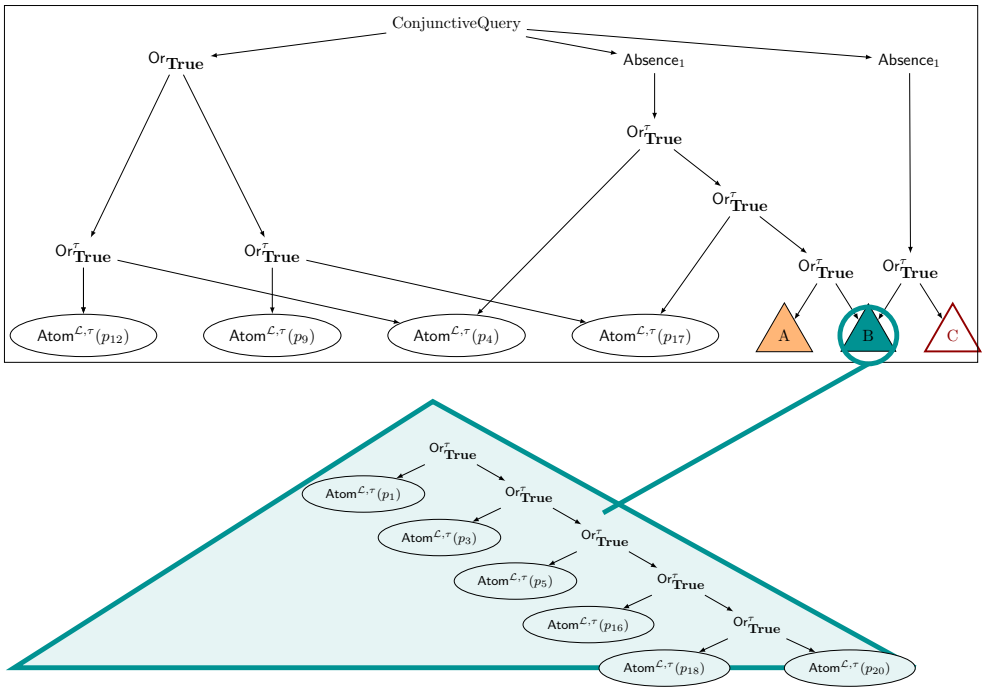


Figure 5. In-depth representation of the query plan associated with the model described in Example 15.

Algorithm 4 Query Optimiser

global *declare*2xtLTL$_f$ $\leftarrow \{\}$; *queryCache* $\leftarrow \{\}$; *collectUnions* $\leftarrow \{\}$; $\mathcal{Q} \leftarrow \{\}$; *atomQ* $\leftarrow \emptyset$
global keyToLabelToSortedIntervals $\leftarrow \{\}$; $\mathcal{S}_\Sigma \leftarrow \{\}$; *Results* $\leftarrow \{\}$

function INSTANTIATE(ψ, Θ, S_A, S_T)
  if ψ.hasTheta **then** ψ.theta $\leftarrow \Theta$
  if ψ.arg$= \emptyset$ **then** ▷ ψ is a leaf
    if ψ.isActivation **or** ψ.isNeither **then**
      ψ.atom $\leftarrow S_A$
    else if ψ.isTarget **then**
      ψ.atom $\leftarrow S_T$
    end if
    if ψ.negated **then** ψ.atom $\leftarrow \complement\ \psi$.atom ▷ Complementing the atoms from the universe set upon negation
    for all *atom* $\in \psi$.atom **do**
      if *atom* $\in \bigcup_{a \in \Sigma} ak(a)$ **then** ▷ The atom is generated from $\mathcal{D}_\varphi$-encoding
        RETRIEVEINTERVALS(*atom*)
      else *atomQ*.put(*atom*)
      end if
    end for
    if $|\psi$.atom$| > 1$ **then**
      disj $\leftarrow \emptyset$
      for all *atom* $\in \psi$.atom **do**
        $\psi' \leftarrow$**new** xtLTL$_f$ ()
        ψ'.atom $= \{atom\}$
        disj.put(*atom*)
      end for
      collectUnions[*disj*].put(ψ)
    else
    end if
  else
    for all arg $\in \psi$ **do**
      arg $\leftarrow$INSTANTIATE(arg, Θ, S_A, S_T)
    end for
  end if

procedure COLLECTUNIONS( ) ▷ DAG over the leaves undergoing union operations.
  for all $\langle atomSet, \psi' \rangle \in$ FINITARYSETOPERATIONS(*collectUnions*, $\mathsf{Or}_{\mathbf{True}}$) **do** ▷ Algorithm S4
    for all $\psi \in$ *collectUnions*[*atomSet*] **do**
      queryCache[ψ] $\leftarrow \psi'$
    end for
  end for

procedure PUTINCACHE(ψ)
  if $\exists \psi'. \langle \psi, \psi' \rangle \in$ *queryCache* **then**
    return ψ'
  else
    for all arg $\in \psi$.args **do**
      arg $\leftarrow$PUTINCACHE(arg)
    end for
    $\psi' \leftarrow$ **new** xtLTL$_f$ ()
    $\psi' \leftarrow \psi$
    queryCache[ψ] $\leftarrow \psi'$
    return ψ'
  end if

procedure RETRIEVEINTERVALS(p_i) ▷ $p_i :=$ a $\wedge$ *partition*
  for all $low_\kappa \leq \kappa \leq up_\kappa \in$ *partition* **do** ▷ $p_i = \bigwedge_{\kappa \in K} low_\kappa \leq \kappa \leq up_\kappa$
    if $\exists h. \langle low_\kappa \leq \kappa \leq up_\kappa, h \rangle \in$ keyToLabelToSortedIntervals[κ][a] **then**
      $\mathcal{S}_\Sigma[p_i]$.put(h)
    else
      Results.put($\emptyset$)
      $\mathcal{S}_\Sigma[p_i]$.put($|Results|$)
      keyToLabelToSortedIntervals[κ][a].put($\langle low_\kappa \leq \kappa \leq up_\kappa, |Results| \rangle$)
    end if
  end for

function QUERYOPTIMISER($\mathcal{M}$,*queryRoot*)
  for all clause$_l$(A, p, B, q) where $\Theta \in \mathcal{M}$ **do**
    if $\exists \psi$: xtLTL$_f$. $\langle$clause$_l$(A, p, B, q) where $\Theta, \psi \rangle \in$ *declare*2xtLTL$_f$ **then** $\mathcal{Q}$.push(ψ)
    else
      $\psi \leftarrow$ INSTANTIATE(xtTemplates[clause$_l$], Θ, clause$_l$.left, clause$_l$.right)
      $\mathcal{Q}$.push(ψ)
    end if
  end for
  COLLECTUNIONS( )
  queryRoot.args $\leftarrow \{$ PUTINCACHE(ψ) $\mid \psi \in \mathcal{Q} \}$
  return *queryRoot*

5.1.3. Enabling Intraquery Parallelism

The query scheduler (Algorithm 5) takes as an input the query compiled in the previous phase and returns the scheduling order for achieving *intraquery parallelism* [42]. The previously generated expression might not be considered as an abstract syntax tree, rather than an abstract syntax DIRECT ACYCLIC GRAPH (DAG) rooted in the entry-point operator *queryRoot*, as we guarantee that sub-expressions appearing multiple times are replaced by unique instances of them.

Therefore, we can freely represent the query plan as a DAG $\mathcal{G}$ in our pseudocode notation, where each root operator in ψ is a single node while edges connect parent operators to the siblings' (ψ.`args`) root operator. Graph edges induce the execution order, where any ancestor node needs to be run after all of its immediate siblings. A reversed topological sort (Line 3) induces the order in which the operations should be run. To know which of these operators can be run contemporarily (i.e., scheduled together [44]) as they share no interdependencies, we compute for each node its maximum distance from *queryRoot* (Line 6). This generates a *layer*ing [45] guaranteeing that all of the nodes at the same levels share no mutual dependencies (Line 10). This enables the level-wise parallelisation of the tasks' execution (also referred to as Intraquery Parallelism [42]), thus showing how such a problem can be reduced into an embarrassingly parallel problem by parallelising the computation of each operator in the same given layer. This procedure runs in linear time with respect to the number of operators appearing in the $\texttt{xt}\text{LTL}_f$ query plan (Lemma S4). We benchmark query plan parallelisation with different task scheduling policies in Section 7.3.

Algorithm 5 Query Scheduler (Section 5.1.3)

```
1:  function QUERYSCHEDULER(G)
2:      layer ← {}
3:      V ← REVERT(TOPOLOGICALSORT(G))
4:      for all ψ ∈ V do
5:          for all ψ' ∈ ψ.args do
6:              ψ'.distance ← max(ψ'.distance, ψ.distance + 1)
7:          end for
8:      end for
9:      for all ψ ∈ V do
10:         layer[ψ.distance].put(ψ)
11:     end for
12:     return layer
```

Example 17. *The DAG in Figure 2 depicts a query plan, where operators' dependencies are suggested as arrows starting from the ancestors. The graph is also already represented as a layered graph, as all of the nodes having the same maximum distance from the query root are aligned horizontally. We might observe that none of the nodes within each layer shares dependencies.*

5.2. Execution Engine

The execution engine (Algorithm 6) runs the previously compiled query (Section 5.1) on top of the relational model populated from the XES log (Section 4.2). The computation will start from the DAG query leaves directly accessing the relational database (Section 5.2.1) for then propagating the results until the root of the DAG is reached (Section 5.2.2). At this point, we can perform the final conjunctive or aggregation queries (Section 5.2.3).

At each stage, we exploit a functor $\mathcal{A}$ associating to each $\texttt{xt}\,\text{LTL}_f$ operator an algorithm which will take the result from the ψ's operands as an input while returning the expected output by formal definition in an intermediate result ρ. This abstraction enables the separation between $\texttt{xt}\,\text{LTL}_f$ syntax and multiple possible algorithmic implementations. Some algorithmic implementations for such operators are discussed in Section 6.

Algorithm 6 Execution Engine (Section 5.2)

1: **function** EXECUTIONENGINE(*layer*, $\mathcal{L}$, $\mathcal{A}$)
2: **for all** $\psi \in atomQ$ **(parallel) do** ψ.`result` $\leftarrow \mathcal{A}(\psi)$
3: RUN$\mathcal{D}_\varphi$-ENCODINGATOMS($\mathcal{L}$) ▷ Algorithm S5
4: **for all** $\langle distance, \Psi \rangle \in layer$ **do**
5: **for all** $\psi \in \Psi$ **(parallel) do**
6: **if** $\psi.atom = \{p_i\}$ **and** $p_i \in \bigcup_{\mathtt{a} \in \Sigma} ak(\mathtt{a})$ **then**
7: ψ.`result` $\leftarrow \mathcal{A}(\text{ATOM}^{\mathcal{L},\tau}_{\cdot})(\psi)$ ▷ Algorithm S5
8: **else if** ψ.`atom` $= \{\mathtt{a}\} \wedge \mathtt{a} \in \Sigma$ **then**
9: **continue** ▷ Already run in Line 2
10: **else**
11: ψ.`result` $\leftarrow (\mathcal{A}(\psi))(\{\psi'$.`result`$|\psi' \in \psi$.args$\})$
12: **end if**
13: **end for**
14: **end for**
15: *queryRoot* $\leftarrow$ *layer*$[0]$
16: **return** *queryRoot*.`result`

For this step, we will not discuss the computational complexity of evaluating the query plan as this is heavily dominated by the computation of every single operator, the model of choice, and the log size. For this reason, we only conducted empirical analysis by benchmarking the run time of the whole execution engine, where models either only contain $\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}$ *(Section 7.2) operators or mainly* $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$ *ones (Section 7.5).*

5.2.1. Basic Operators' Execution

Among all of the possible DAG node leaves, we first (Line 2) execute the leaves either *(i)* directly associated with an activity label, or *(ii)* First and Last. For the former *(i)*, each activity label a is run through its correspondent $\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathtt{a})$ operator, whether either A or T or none are going to be set depending on the fact that such atom refers to an activation (ψ.`isActivation`) or target (ψ.`isTarget`) condition, or whether the associated result should be ignored as a whole (**ψ.`isNeither`**). For the latter *(ii)*, we directly access the data tables and retrieve the data from them. As the tables are already sorted by trace and event id, no further post-processing besides the insertion of activation or target label in the nested component L of the intermediate representation is required.

Next, we evaluate the intermediate result associated with each atom generated by the $\mathcal{D}_\varphi$-encoding (Line 3). Intuitively (Please refer to Supplement III.1 for a more in-depth discussion with pseudocode.), this requires three subsequent phases. *First*, we obtain the compound conditions grouped by key and activity label as collected at query compile time, and we exploit them to pipeline multiple range queries over each $\mathsf{AttributeTable}^{k}_{\mathcal{L}}$. The associated results are cached. *Second*, we compute the results for each atom by intersecting the previously cached results before actually computing the actual $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$. This also guarantees that shared intersections are run at most once across all of the previously cached results. *Third*, we exploit the former result to compute the $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$ operator at the leaf level on our DAG, while associating either an activation or a target mark in L depending on the prior definition of our leaf-level operator.

5.2.2. Results Propagation

After running the basic operators and their derived counterparts (e.g., $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$), the only $\mathtt{xtLTL_f}$ operators that KnoBAB runs are the ones not accessing the relational tables. KnoBAB implements three different $\mathcal{A}$-s which are only sharing the implementation for the aforementioned operators: one set is either strictly abiding by the formal definition and completely ignoring the fact that the intermediate results are provided as an ordered set of tuples or providing slower algorithms overall, one will leverage appropriate data representation, thus outperforming the former operations, while the other will implement hybrid algorithms for selecting the best performant implementation depending on the data conditions through hybrid algorithms. An in-depth discussion of how different operators might have different algorithmic implementations is postponed to a specific section (Section 6).

While computing these, we associate a temporary primary-memory cache (We can completely free each intermediate cache if we are not computing a CONFIDENCE query and if the furthest ancestor has already accessed it, or if the cache is unassociated with any activation required by CONFIDENCE.) to each intermediate representation being computed (ψ.`result`).

5.2.3. Conjunctive and Aggregation Queries

The first version of KnoBAB supports the CONJUNCTIVE QUERY of the model as well as three aggregation queries: MAX-SAT, CONFIDENCE, and SUPPORT. While the former requires a further untimed $\mathsf{And}_{\mathbf{True}}$ among all the intermediate results associated with the computation to each clause, the aggregation requires just an iteration over the provided results. The conjunctive query is formulated as follows:

$$\text{CONJUNCTIVEQUERY}(\rho_1, \ldots, \rho_n) = \mathsf{And}_{\mathbf{True}}(\rho_1, \ldots \mathsf{And}_{\mathbf{True}}(\rho_{n-1}, \rho_n))$$

The Max-SAT will calculate the ratio of the intermediate results ρ_l associated with each clause c_l, over the total number of model clauses $|\mathcal{M}|$. $\mathsf{ActLeaves}(\rho_l)$ is the untimed union of the intermediate results yielded by activation conditions for the Declare clause $c_l \in \mathcal{M}$. For c_l, the CONFIDENCE represents the ratio between the number of traces returned by ρ_l and the total number of traces that contain activation conditions. When the same numerator is on the other hand divided by the total log traces, we have SUPPORT. Following the computation of each ρ_l per clause c_l, the aggregation functions can be expressed as follows:

$$\text{Max-SAT}(\rho_1, \ldots, \rho_n) = \left(\frac{|\{\, l \mid \exists j, L. \langle i, j, L \rangle \in \rho_l \,\}|}{|\mathcal{M}|} \right)_{\sigma^i \in \mathcal{L}}$$

$$\text{CONFIDENCE}(\rho_1, \ldots, \rho_n) = \left(\frac{|\{\, i \mid \exists j, L. \langle i, j, L \rangle \in \rho_l \,\}|}{|\mathsf{ActLeaves}(\rho_l)|} \right)_{c_l \in \mathcal{M}}$$

$$\text{SUPPORT}(\rho_1, \ldots, \rho_n) = \left(\frac{|\{\, i \mid \exists j, L. \langle i, j, L \rangle \in \rho_l \,\}|}{|\mathcal{L}|} \right)_{c_l \in \mathcal{M}}$$

The execution of such queries is performed in a non-parallel way, as each aggregation query will appear at the top of the query plan, and this will be associated with the latest execution run of the scheduler (Line 15). We then return and prompt the result associated with the root node of our query plan (Line 16).

Example 18. *As per previous discussions, the satisfaction of a model requires the satisfaction of all constituent clauses. The model described as the bottom table in Figure 6 is the result of further elaborating on the requirements from Example 1. This is only one example of a myriad of possible solutions, which can either be manually defined (as here), or generated through mining/learning techniques. Such model can be now used to compute the degree to which the model is satisfied, or per trace, each requiring different metrics. An example of a trace-wise metric is Max-SAT while Support and Confidence values can be computed per clause. By providing the trace metrics, we are able to analyse the scenarios with respect to the model, and therefore help provide insight into the exhibits of any backdoors in the software. On the contrary, providing model metrics allows us to establish the suitability of a model and its constituent clauses; for example, clauses with low Support but high Confidence may indicate a correlation between events. Finally, a conjunctive query will return all the traces satisfying all the model clauses. From Figure 6, it is evident that the only trace where a successful attack occurred is σ_1, as returned by the Conjunctive Query, providing the grounds that we have a suitable model. By exploiting the previous formulæ, we can compute the metrics as Table 6. These metrics may provide some insight of correlations between events. For example, clause Ⓑ had Support(Confidence) values as 1.0, while clause Ⓒ had 1/3 (1.0). This therefore indicates that the activation of the latter occurred much less than that of the former; however, every time the activation occurred, the clause was always fulfilled. Conclusions such as these can help to identify*

any weaknesses/strengths within the model and the system itself (here, the metrics obtained from Ⓒ may suggest that comm/act contain a correlation that needs investigating).

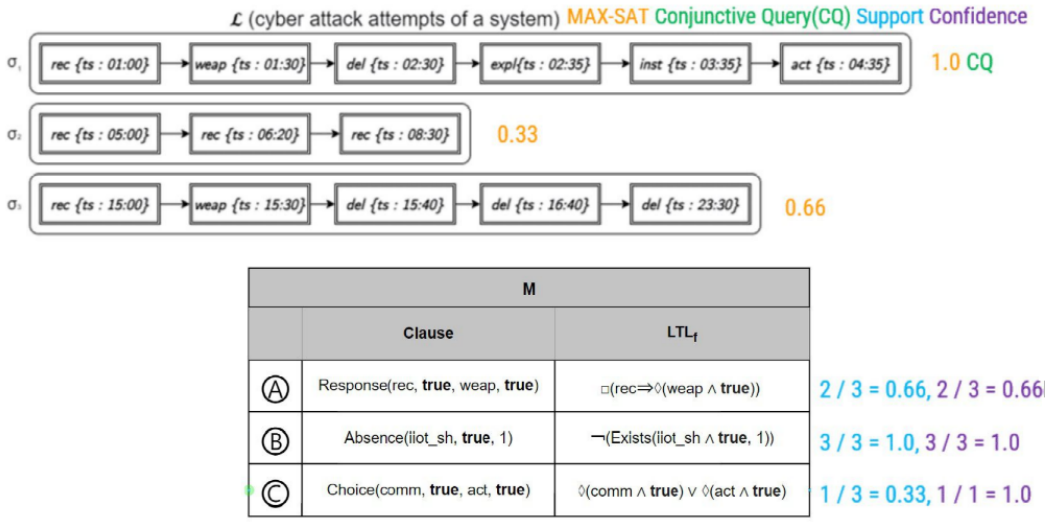


Figure 6. Assessing a high-level use case of an intrusion attack on a software system through a declarative model.

Table 6. Conjunctive and Aggregation queries for Figure 6.

(**a**) Metric calculations per trace.		
Trace	MAX-SAT	in Conjunctive Query
σ_1	$\frac{\lvert\{c_1,c_2,c_3\}\rvert}{\lvert M\rvert} = 1.0$	**true**
σ_2	$\frac{\lvert\{c_2\}\rvert}{\lvert M\rvert} = 1/3$	**false**
σ_3	$\frac{\lvert\{c_1,c_2\}\rvert}{\lvert M\rvert} = 2/3$	**false**
(**b**) Metric calculations per clause.		
Clause	Support	Confidence
Ⓐ	$\frac{\lvert\{\sigma_1,\sigma_3\}\rvert}{\lvert\mathcal{L}\rvert} = 2/3$	$\frac{\lvert\{\sigma_1,\sigma_3\}\rvert}{\lvert\{\sigma_1,\sigma_2,\sigma_3\}\rvert} = 2/3$
Ⓑ	$\frac{\lvert\{\sigma_1,\sigma_2,\sigma_3\}\rvert}{\lvert\mathcal{L}\rvert} = 1.0$	$\frac{\lvert\{\sigma_1,\sigma_2,\sigma_3\}\rvert}{\lvert\{\sigma_1,\sigma_2,\sigma_3\}\rvert} = 1.0$
Ⓒ	$\frac{\lvert\{\sigma_1\}\rvert}{\lvert\mathcal{L}\rvert} = 1/3$	$\frac{\lvert\{\sigma_1\}\rvert}{\lvert\{\sigma_1\}\rvert} = 1.0$

6. Algorithmic Implementations

In this section, we show how the relational model and the proposed intermediate result representation enable the definition of different operators boosting the query performance compared to an equivalent $\mathtt{xt}\,\mathrm{LTL}_f$ *expression obtained through the straightforward translation procedure entailed by the lemmas in Appendix A.2 (*LTL_f*-rewriting). Each subsection is going to discuss different possible algorithms for implementing some operators, as well as discussing its associated pseudocode and computational complexity.*

6.1. Timed and Untimed Or/And

Algorithm 7 shows the implementation of the timed version of the $\mathsf{And}^{\tau}_{\Theta}$ (Line 27) and $\mathsf{Or}^{\tau}_{\Theta}$ (Line 28) operators, for then generalising this concept for the implementation of the untimed And_{Θ}. We omit the discussion related to the implementation of the untimed Or_{Θ} operator for the sake of conciseness.

As we see from their formal definition, any binary $\texttt{xtLTL}_\text{f}$ operator supports Θ conditions. And (and Or) resembles a sorted set intersection (or union, Line 11), where we use both trace (i) and event (j) id information from the intermediate result triplet as preliminary equality condition for the match. We also use a Θ binary predicate to be tested over the activated and targeted events in the third component (L). The event shared among the operands is returned if either Θ is always true (Line 7) or, from this point in time, if there exists one activated future activated event (in a L coming from the left operand) as well as a targeted one (in a L coming from the right operand) satisfying the correlation (Line 4). The match is then represented as a marked correlation condition $M(h, k)$, which is then collected in the L associated with the returned event (Line 5).

For the untimed And_Θ operator, we require to return one single trace i as $\langle i, 1, L\rangle$ if either Θ is true and each operator has an event from σ^i, or if there exists at least one event per operand from the same trace performing the match. This can be implemented in two different ways: we can either group the records by trace id (Lines 31 and 32) and then scan the intermediate results' records (Line 38) associated with the same trace id (Line 36, SLOWUNTIMEDAND) or straightforwardly scan them by trace id without exploiting the preliminary aggregation (FASTUNTIMEDAND). This latter implementation is possible as the intermediate results records are already sorted, thus allowing the results' aggregation while scanning the intermediate results without the need for any preliminary aggregation. We show that the faster version is always faster than computing it with its slower counterpart in Corollary S1.

Similar considerations can be also applied for the untimed Or operation, for which we implemented equivalent SLOWUNTIMEDOR and FASTUNTIMEDOR, as we only need to pay an additional linear scan for the unmatched traces.

6.2. Choice and Untimed Or

We prelude our analysis of derived operators by firstly discussing the difference in computational complexity between providing the straightforward translation from LTL_f to $\texttt{xtLTL}_\text{f}$ and to exploiting equivalent expression rewriting in $\texttt{xtLTL}_\text{f}$. We remind the reader that the definition of Choice (see Table 1) states that either one condition or another should occur anytime in the trace.

This requirement can be interpreted in two distinct ways: by either returning all the traces satisfying the first condition or the second separately and then merging them, or instead collecting all of the events satisfying either the former or the latter condition while jointly scanning both operands, and then returning the traces where any one of them is met. After observing (Please also refer to the experiments in Section 7.1 for the empirical evidence of such theoretical claims.) that the SLOWUNTIMEDOR is actually slower than FASTUNTIMEDOR and that the latter actually implements the Choice declarative clause (Corollary A1), the time complexity of computing the LTL_f rewriting of Choice in its LTL_f-rewriting is almost equivalent to the time complexity of FASTUNTIMEDOR, as we can have an asymptotic constant speed-up in the best case scenario (Corollary S3). As the untimed Or_Θ behaves by computing a Future operator (Algorithm 8) on each of its operands, the computation of an additional Future operator for each of its operands becomes an omittable overhand.

6.3. Untimed Until(s)

We show how different data access policies for scanning the intermediate results affect the overall computational complexity as well as their associated run time. Algorithm 9 provides two possible variants for the untimed until:

Algorithm 7 $\texttt{xtLTL}_\text{f}$ pseudocode implementation for And_Θ and Or_Θ operators

function $\mathcal{T}^{E,i}_\Theta(L, L')$
  $L'' \leftarrow \emptyset$; *hasMatch*$\leftarrow \Theta =$ **True** ▷ (Explicitly) computing $\mathcal{T}^{E,i}_\Theta$
  if $\Theta \neq$ **True and** $L \neq \emptyset$ **and** $L' \neq \emptyset$ **then**
    for all $A(m) \in L$ **and** $T(n) \in L'$ **s.i.** $\Theta(m, n)$ **do**
      $L'' \leftarrow L'' \cup \{ M(m, n) \}$; *hasMatch*$\leftarrow$ **true**
    end for
  else
    $L'' \leftarrow L'' \cup L' \cup L$
  end if
  if *hasMatch* **then return** L'' **else return False**

function TIMEDINTERSECTION$_\Theta(\rho, \rho', isUnion)$
  $it \leftarrow$**Iterator**$(\rho), it' \leftarrow$**Iterator**(ρ')
  while $it \neq\uparrow$ **and** $it' \neq\uparrow$ **do**
    $\langle i, j, L\rangle \leftarrow \texttt{current}(it), \langle i', j', L'\rangle \leftarrow \texttt{current}(it')$
    if $i = i'$ **and** $j = j'$ **then**
      $tmp \leftarrow \mathcal{T}^{E,i}_\Theta(L, L')$
      if $tmp \neq$ **False then yield** $\langle i, j, tmp\rangle$
      $\texttt{next}(it); \texttt{next}(it')$;
    else if $i < i'$ **or** $(i = i'$ **and** $j < j')$ **then**
      if *isUnion* **then yield** $\langle i, j, L\rangle$ **end if**
      $\texttt{next}(it)$
    else
      if *isUnion* **then yield** $\langle i', j', L'\rangle$ **end if**
      $\texttt{next}(it')$
    end if
  end while

function $\text{AND}^\tau_\Theta(\rho, \rho')$ TIMEDINTERSECTION$_\Theta(\rho, \rho',$ **false**$)$

function $\text{OR}^\tau_\Theta(\rho, \rho')$ TIMEDINTERSECTION$_\Theta(\rho, \rho',$ **true**$)$

function SLOWUNTIMEDAND$_\Theta(\rho, \rho')$
  leftOperand $\leftarrow \{\}$; *rightOperand* $\leftarrow \{\}$
  for all $\langle i, j, L\rangle \in \rho$ **do** *rightOperand*$[i]$.$\texttt{put}(\langle i, j, L\rangle)$
  for all $\langle i, j, L\rangle \in \rho'$ **do** *rightOperand*$[i]$.$\texttt{put}(\langle i, j, L\rangle)$
  $it \leftarrow$**Iterator**$($*leftOperand*$), it' \leftarrow$**Iterator**$($*rightOperand*$)$
  while $it \neq\uparrow$ **and** $it' \neq\uparrow$ **do**
    $\langle i, R\rangle \leftarrow \texttt{current}(it)$; $\langle i', R'\rangle \leftarrow \texttt{current}(it')$
    if $i = i'$ **then**
      $L'' \leftarrow \emptyset$; *hasMatch*$\leftarrow \Theta =$ **True**
      for all $\langle i, j, L\rangle \in R$ **and** $\langle i, j', L'\rangle \in R'$ **do**
        $tmp \leftarrow \mathcal{T}^{E,i}_\Theta(L, L')$
        if $tmp \neq$ **False then**
          hasMatch$\leftarrow$ **true**; $L'' \leftarrow L'' \cup tmp$
        end if
      end for
      if *hasMatch* **then yield** $\langle i, 1, L''\rangle$;
    else if $i < i'$ **then** $\texttt{next}(it)$
    else $\texttt{next}(it')$
    end if
  end while

function FASTUNTIMEDAND$_\Theta(\rho, \rho')$
  $it \leftarrow$**Iterator**$(\rho), it' \leftarrow$**Iterator**(ρ')
  while $it \neq\uparrow$ **and** $it' \neq\uparrow$ **do**
    $\langle i, \iota, \lambda\rangle \leftarrow \texttt{current}(it)$; $\langle i', \iota', \lambda'\rangle \leftarrow \texttt{current}(it')$
    if $i = i'$ **then**
      $L'' \leftarrow \emptyset$; *canOptimize*$\leftarrow$ **false**
      $it_* \leftarrow it$
      while $it_* \neq\uparrow$ **do**
        $\langle i, j, L\rangle \leftarrow \texttt{current}(it_*)$; $it'_* \leftarrow it'$
        if not *canOptimize* **then**
          while $it'_* \neq\uparrow$ **do**
            $\langle i', j', L'\rangle \leftarrow \texttt{current}(it'_*)$
            $tmp \leftarrow \mathcal{T}^{E,i}_\Theta(L, L')$
            if $tmp \neq$ **False then**
              hasMatch$\leftarrow$ **true**; $L'' \leftarrow L'' \cup tmp$
            end if
            $\texttt{next}(it'_*)$
          end while
          if $\Theta =$ **True then** *canOptimize*$\leftarrow$ **true**
        else$L'' \leftarrow L'' \cup L$
        end if
        $\texttt{next}(it_*)$
      end while
      if *hasMatch* **then yield** $\langle i, 1, L''\rangle$;
      $it \leftarrow it_*; it' \leftarrow it'_*$;
    else if $i < i'$ **then** $\texttt{next}(it)$
    else $\texttt{next}(it')$
    end if
  end while

Algorithm 8 $\texttt{xtLTL}_\text{f}$ pseudocode implementation for Future and Globally

1: **function** FUTURE(ρ) ▷ $O(|\mathcal{L}|\epsilon^2)$
2: **for all** $\langle i, j, L\rangle \in \rho$ **do yield** $\langle i, j, \bigcup\{\, L' \mid \langle i, j', L'\rangle \in \rho \text{ and } j' \geq j \,\}\rangle$
3: **end for**
4: **function** GLOBALLY(ρ)
5: **for all** $\langle i, j, L\rangle \in \rho$ **do**
6: $E \leftarrow \{\, j' \mid \langle i, j', L'\rangle \in \rho \text{ and } j' \geq j \,\}$
7: **if** $|E| = \ell_t - j$ **then yield** $\langle i, j, \bigcup\{\, L' \mid \langle i, j', L'\rangle \in \rho \text{ and } j' \in E \,\}\rangle$ **end if**
8: **end for**

All optimisations happen when the activation condition coming from the second operand does not occur at the beginning of a trace (Lines 34 and 61). In the first variant, we calculate, for all of the events in the first operand starting from the beginning of the trace (Line 29, and Line 51 for the second variant), the position of the last activated event preceding the current target condition with a logarithmic scan with respect to the length of the first operand (Line 34). On the other hand, the second variant directly discards the traces not starting with a target condition (Line 59) and, otherwise, it moves the scan of the first operand—from that initial position—by an offset equal to the distance from the event preceding activation (Line 61): if that position does not correspond to an activation condition preceding the current activation condition, then we completely discard the trace (Line 65). The matching conditions between activations and target are implemented similarly (Lines 37–40 and 67–69). Lemma S7 shows that the second variant is better asymptotically only for bigger datasets.

6.4. Derived Operators

Our previous observation for the untimed Or_Θ led us to the definition of additional derived operators with the hope of easing the overall computational complexity. We walked in the same footsteps of relational algebra, where it was customary to merge multiple operators into one single new operator if the latter might be implemented through a more performant algorithm than computing an equivalent expression being the straightforward translation of LTL_f formulae into LTL_f (LTL_f *rewriting*).

For example, we can implement TIMEDANDFUTURE by extending the fast implementation of the timed AND operator, and considering all of the trace events from the second operand succeeding the events from the first operand within the same trace. Similar considerations can be carried out with TIMEDANDGLOBALLY, where in the former we need to count whether all of the events from the current time until the end of the trace are present in the rightmost operand, while in the latter we also need to skip the matched event from the rightmost operand and start scanning from the following ones.

For simplicity's sake, we postpone the discussion of these operands' pseudocode as well as the discussion of their computational complexity in Supplement II.2, where we show that these two operators might come with two different algorithms, for which there always exists one of them having a lower running time with respect to the equivalent $\texttt{xtLTL}_\text{f}$ expression containing no derived operators. We can show formally that, while the first implementation (*variant*) works better for smaller datasets, the second works better for reasonably long traces when the number of the traces is upper bounded by an exponential number of events (Corollary S2).

Algorithm 9 Two implementations for the untimed $\mathtt{xtLTL_f}$ Until_Θ.

function $A^i_\Theta(\langle it', bEnd\rangle, \langle it, aEnd\rangle)$
  $\langle i', j', L'\rangle \leftarrow \mathtt{current}(it')$; $L'' \leftarrow \emptyset$;
  if $\Theta \neq$ **True and** $L' \neq \emptyset$ **then**
    for all $A(k), M(k,k') \in L'$ **do**
      $aBeg \leftarrow it$
      while $aBeg \neq aEnd$ **do**
        $\langle i, j, L\rangle \leftarrow \mathtt{current}(aBeg)$
        if $L = \emptyset$ **then** $L'' \leftarrow L'' \cup L$
        else
          $anyMatch \leftarrow$ **false**
          for all $T(h) \in L$ **s.t.** $\Theta(\sigma^i_k, \sigma^i_h)$ **do** $anyMatch \leftarrow$ **true**; $L'' \leftarrow L'' \cup \{M(k,h)\}$
          end for
          if not *anyMatch* **then return False**
        end if
      end while
    end for
  else
    while $aBeg \neq aEnd$ **do**
      $\langle i, j, L\rangle \leftarrow \mathtt{current}(aBeg{+}{+})$; $L'' \leftarrow L'' \cup L$
    end while
    $L' \leftarrow L'' \cup L'$
  end if
  return L''

function $\textsc{UntimedUntil}^1_\Theta(\rho, \rho')$
  $it \leftarrow$**Iterator**(ρ), $it' \leftarrow$**Iterator**(ρ')
  while $it' \neq \uparrow$ **do**
    $\langle i', j', L'\rangle \leftarrow \mathtt{current}(it')$; $bend \leftarrow \textsc{UpperBound}(\rho', it', \uparrow, \langle i', |\sigma^{i'}|+1, \emptyset\rangle)$
    $it \leftarrow \textsc{LowerBound}(\rho, it, \uparrow, \langle i', 1, \emptyset\rangle)$
    $atLeastOneResult \leftarrow$ **false**; $L'' \leftarrow \emptyset$
    while $it' < bend$ **do**
      if $j' = 1$ **then**
        $atLeastOneResult \leftarrow$ **true**; $L'' \leftarrow L'' \cup L$; $it'{+}{+}$
      else
        $aEnd \leftarrow \textsc{UpperBound}(\rho, it, \uparrow, \langle i', j'-1, \top_\Omega\rangle)$
        if $it = aEnd$ **or** $\textsc{Distance}(aEnd - 1, it) + 1 \neq j' - 1$ **then break**
        else ▷ $i = i'$. Computing partial $\mathcal{T}^{A,i}_\Theta$
          $tmp \leftarrow A^i_\Theta(\langle it', bend\rangle, \langle it, aEnd\rangle)$
          $atLeastOneResult \leftarrow atLeastOneResult$ **or** $tmp \neq$ **False**
          if $tmp \neq$ **False then** $L'' \leftarrow L'' \cup tmp$;
          $it'{+}{+}$
        end if
      end if
    end while
    if *atLeastOneResult* **then yield** $\langle i, 1, L''\rangle$
    $it' \leftarrow bend$
  end while

function $\textsc{UntimedUntil}^2_\Theta(\rho, \rho')$
  $it \leftarrow$**Iterator**(ρ), $it' \leftarrow$**Iterator**(ρ')
  while $it' \neq \uparrow$ **do**
    $\langle i', j', L'\rangle \leftarrow \mathtt{current}(it')$; $bend \leftarrow \textsc{UpperBound}(\rho', it', \uparrow, \langle i', |\sigma^{i'}|+1, \emptyset\rangle)$
    $it \leftarrow \textsc{LowerBound}(\rho, it, \uparrow, \langle i', 1, \emptyset\rangle)$
    $atLeastOneResult \leftarrow$ **false**; $L'' \leftarrow \emptyset$
    while $it' < bend$ **do**
      if $j' = 1$ **then**
        $atLeastOneResult \leftarrow$ **true**; $L'' \leftarrow L'' \cup L$; $it'{+}{+}$
      else if $it = \uparrow$ **then break**
      else
        $\langle i, j, L\rangle \leftarrow \mathtt{current}(it)$;
        if $j > 1$ **then break**
        else
          $aEnd \leftarrow \textsc{MoveForward}(it, j'-1)$; ▷ $(it) + j' - 1$
          if $aEnd = \uparrow$ **then break**
          else
            $\langle i_e, j_e, L_e\rangle \leftarrow \mathtt{current}(aEnd)$
            if $i_e > i'$ **or** $j_e \neq j' - 1$ **then break**
            else ▷ $i = i' = i_e$. Computing partial $\mathcal{T}^{A,i}_\Theta$
              $tmp \leftarrow A^i_\Theta(\langle it', bend\rangle, \langle it, aEnd\rangle)$
              $atLeastOneResult \leftarrow atLeastOneResult$ **or** $tmp \neq$ **False**
              if $tmp \neq$ **False then** $L'' \leftarrow L'' \cup tmp$;
              $it'{+}{+}$
            end if
          end if
        end if
      end if
    end while
    if *atLeastOneResult* **then yield** $\langle i, 1, L''\rangle$
    $it' \leftarrow bend$
  end while

Table 7 shows the range of datasets used for benchmarking.

Table 7. Range of datasets used for benchmarking.

Competitor	Dataset	Traces $\|\mathcal{L}\|$	Events	Distinct Activities $\|\Sigma\|$
SQL Miner	BPIC 2011 (original)	1143	150,291	624
	BPIC 2011 (10)	10	2613	158
	BPIC 2011 (100)	100	12,195	276
	BPIC 2011 (1000)	1000	133,935	607
Declare Analyzer	BPIC 2012 (original)	13,087	262,200	24

7. Results and Discussion

Our benchmarks exploited a Razer Blade Pro on Ubuntu 20.04: Intel Core i7-10875H CPU @ 2.30 GHz–5.10 GHz, 16GB DDR4 2933 MHz RAM, 450 GB free disk space. All of our datasets used for benchmarking (synthetic data generation (Section 7.1), BPIC_2011 (Sections 7.2 and 7.3), BPIC_2012 (Sections 7.4 and 7.5) and our proposed cancer example (Section 1.1) are publicly available (https://dx.doi.org/10.17605/OSF.IO/2CXR7). Table 7 summarises these datasets' features.

7.1. Comparing Different Operators' Algorithms

We advocate that the choice of representing the intermediate representation as an ordered record set allows the exploitation of efficient algorithms through which we might avoid costly counting and aggregation operations [46]. From these comparisons, the operators fully assuming that the data are sorted greatly outperform naïve operators. Walking in the footsteps of relational algebra, we show that the computational complexity of so-called derived operators outperforms the computation of an equivalent expression evaluated through either naïve or fast algorithms. The experiments are discussed in order of presentation of the algorithms in the previous section.

To create a suitable testing environment, we synthetically generate data-less logs, where the trace and log lengths are increased 10-fold at a time from 10^1–10^4 , with the resulting sets $|\mathcal{L}| \in \{ 10, 100, 1000, 10, 000 \}$ $\epsilon \in \{ 10, 100, 1000, 10, 000 \}$, with the most extreme log consisting of 10^8 events. In some cases, we exceeded 16 GB of primary memory on the testing machine; in the following results (Figure 7–10), M+ denotes an out of memory exception. We chose to generate our data in place of using existing real-world logs (https://dx.doi.org/10.17605/OSF.IO/2CXR7) , as the controlled scenario allows for identifying the location and extent of any possible speed-ups. These data were up-sampled, guaranteeing that a given log configuration was always a subset of the larger. The data generation randomly assigned events from the universal alphabet ($\Sigma = \{ A, B, C, D, E \}$), up to the maximum length for the set in consideration, and we stored the resulting logs as tab-separated files.

Our operators consider correlations between timed events A and B, where the computed speed-up is *per* operator. Given this, we denote $\rho_1 = \mathsf{Activity}_A^{\mathcal{L},\tau}(A), \rho_2 = \mathsf{Activity}_T^{\mathcal{L},\tau}(B)$, prior to benchmarking, and we ignore the time required for accessing the data on the knowledge base, as the focus of the present benchmarks is solely on the operators. Details of how the custom clauses/derived operators are run are demonstrated in Table 8, while singular operators are run sequentially.

Table 8. Proposed operator semantics vs. traditional.

Operator	$\mathbf{LTL_f}$ Rewriting	Optimised
Choice	$\mathsf{Or}_\Theta(\mathsf{Future}(\rho_1), \mathsf{Future}(\rho_2))$	$\mathsf{Or}_\Theta(\rho_1, \rho_2)$
TimedAndFuture	$\mathsf{And}_\Theta(\rho_1, \mathsf{Future}^\tau(\rho_2))$	$\mathsf{AndFuture}^\tau_\Theta(\rho_1, \rho_2)$
TimedAndGlobally	$\mathsf{And}_\Theta(\rho_1, \mathsf{Globally}^\tau(\rho_2))$	$\mathsf{AndGlobally}^\tau_\Theta(\rho_1, \rho_2)$

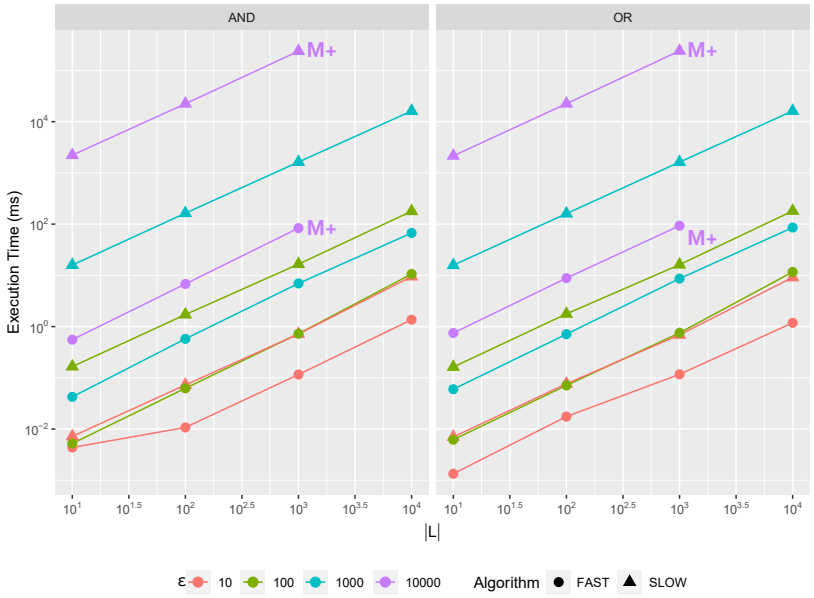


Figure 7. Results for the fast set operations Section 6.1 against the traditional logical implementation.

Untimed Or/And . The first group of experiments aim to challenge different possible algorithms for the same $\mathtt{xt}\mathrm{LTL}_\mathrm{f}$ operators, $\mathrm{And}_{\mathbf{True}}$ and $\mathrm{Or}_{\mathbf{True}}$, as discussed in Section 6.1. The outcome of such experiments is given in Figure 7: our experiments reveal that, in every case, the FAST- operators are *always* more performant than their logical counterparts. Our benchmark confirms the cost of overhead encumbered by the SLOW- implementation, which conforms **linearly** to increased log size, almost **polynomially** with trace length. This aggregation is upper bounded with a **quadratic** with respect to trace length ϵ (Lines 31 and 32); in the most extreme case ($\epsilon = 10^4$), the cost is over one order of magnitude versus the algorithm without aggregation. From now on, we always exploit our FAST- operators in place of the SLOW- equivalent for representing non-derived $\mathtt{xt}\mathrm{LTL}_\mathrm{f}$ operators, which usually suffer the cost caused by the preliminary aggregation as per previous experiments.

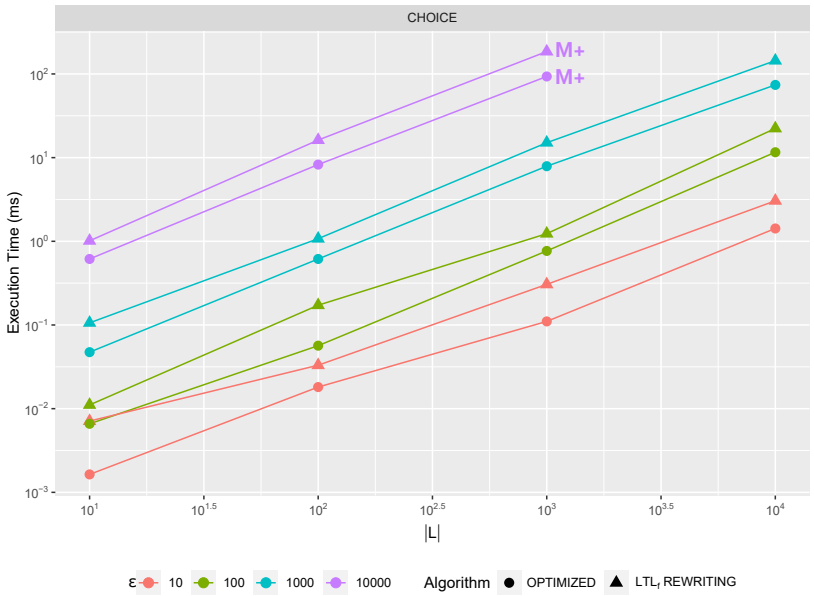


Figure 8. Results for the custom declarative clause implementations Section 6.2 against the traditional logical implementation.

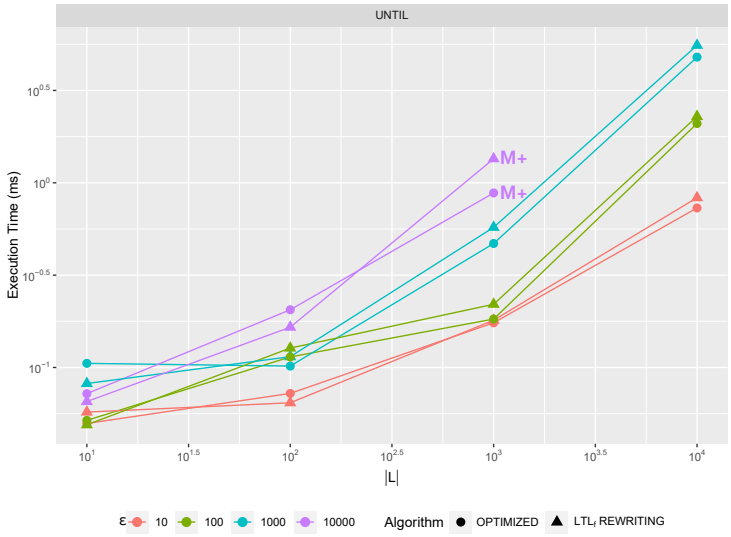


Figure 9. Results for the **UNTIL** operator (Section 6.3).

Choice and Untimed Or. The next set of experiments is to evaluate the customary declarative clause implementation, where we hypothesise reformulating the semantics associated with Choice to provide performance gains from the absence of preliminary aggregations via the UntimedFuture operator. In fact, the proposed optimisation derives from the omittance of the Future operators for ρ_1, ρ_2, which formally comply with the logical definition. For the untimed Future Section 3.2.2 operator, bounded scans can be exploited, as the data are sorted with respect to trace id, and all the events that satisfy ρ for the current trace id are included in the result. Therefore, we expect an overhead that grows linearly with log size. Figure 8 shows that, in the best case ($\epsilon = 10$), we gain 0.5 orders of magnitude in performance. The findings affirm that log size has a greater influence on computational overhead than trace length. For $\epsilon \geq 10^3$, the overhead resulting from the Future operators steadily increases while both the trace length ϵ and the log size $|\mathcal{L}|$ grows, albeit this is negligible in the logarithmic scale.

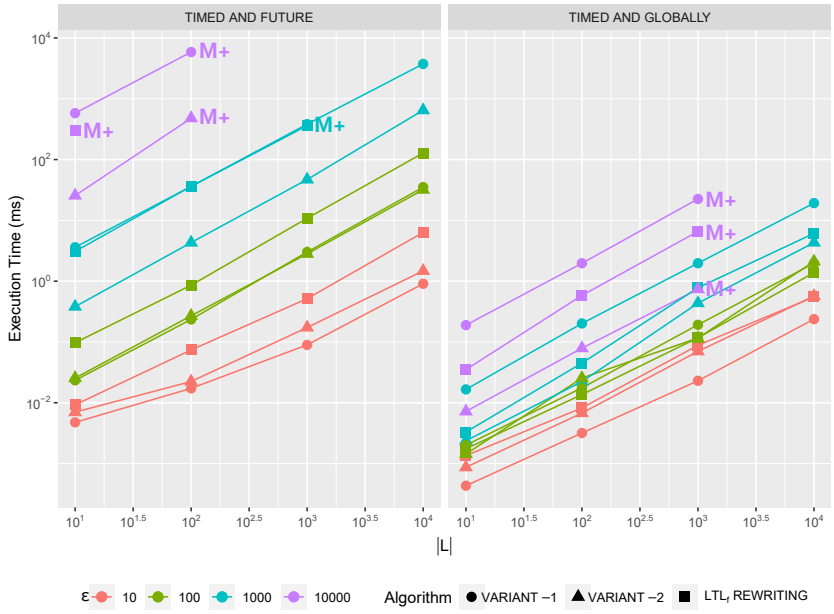


Figure 10. Results for the derived operators **TIMEDANDFUTURE** and **TIMEDANDGLOBALLY** Section 6.4. We include both variants of the fast implementations to analyse the environments where each thrive.

Untimed Until(s). Benchmarks from Figure 9 show that the first variant is almost always more performant than the second one for considerably short traces, while the latter becomes more efficient when ϵ increases. With significant increases to log size, the latter becomes more performant; when $|\mathcal{L}| = 10^4$, all cases show improved running times, regardless of ϵ. The plots also show that the operator's running time is polynomial with respect to the number of traces in the log, as a consequence of the increased scans within every single trace.

Derived Operators. The final set of experiments is to test whether the newly proposed derived operators achieve more optimised results than those from their LTL_f rewriting counterpart (Table 8). For example, TIMEDANDGLOBALLY can be optimised with the customary algorithms replacing one single operator with the execution of multiple pipelined operators. Computations from LTL_f rewriting demonstrate worse performance than the derived counterparts across all operators; in the most extreme case TIMEDANDGLOBALLY, there is over $10^{1.5}$ speed-up for $\epsilon = 10^4$. We were able to conclude that different impersonations to the internal data storage of the optimised algorithm may provide better results depending on the log size. As for UntimedUntil, we provide two implementations for **TIMEDANDGLOBALLY** and **TIMEDANDFUTURE**, VARIANT-1 (Algorihtm S1) and VARIANT-2 (Algorihtm S2), with the latter exploiting bounded reversed scans on the data.

TIMEDANDGLOBALLY: by merging the AND join operation with Globally, we only consider elements within the same trace **after** the first operand. The logical implementation performs these operations separately, and so cannot reap the benefits of a merged join [47]. Figure 10 shows that, in most cases, there is a linear performance gain with log size. VARIANT-2 aims to exploit potential gains from a reversed scan of a trace while VARIANT-1 provides a forwards scan for every activation. By performing a reverse scan, the latter is able to prune further events from any activations happening in the past, as the condition did not hold for the current time. For smaller trace lengths ($\epsilon \leq 10^1$), the VARIANT-1 demonstrates better performance than VARIANT-2. With increased trace length, the latter operators outperform the former, sometimes by over an order of magnitude ($\epsilon = 10^4$). In some cases, the VARIANT-1 performs slower than their LTL_f-rewriting counterparts ($\epsilon \geq 10^3$).

TIMEDANDFUTURE: the principal optimisation gains from this operator follow the same reasoning as **TIMEDANDGLOBALLY**; however, the implementations of the variants follow a unique approach. By exploiting the allocation of intermediate data structures in reverse, VARIANT-2 also provides improved performance for larger $|\mathcal{L}|$. As with **TIMEDANDGLOBALLY**, VARIANT-1 outperforms the former for smaller trace lengths.

We conclude that VARIANT-1 (VARIANT-2) of **TIMEDANDFUTURE** and **TIMEDANDGLOBALLY** outperform each other for small (large) trace lengths. In addition, the first variant of Until proves to be more performant than our second variant for smaller log lengths. We design a mechanism for always running the fastest algorithm under the previously-observed circumstances. We then need to calculate the average trace length and the log size at data loading time (this only needs to happen once per log). Then, at query time, the most optimal operator is chosen based on these values. We define a HYBRID TRACE QUERY THRESHOLD γ of $10^2/2$ (Lines 5 and 9) and a HYBRID LOG QUERY THRESHOLD η of $10^3/2$ (Line 1); values exceeding these thresholds will execute the operators more tailored towards large trace (log) sizes. The pseudocode provided as Algorithm 10 demonstrates how two different variants can be engulfed in one single parametric algorithm.

Algorithm 10 Hybrid Algorithms

function HYBRIDUNTIMEDUNTIL$^{\eta}_{\Theta}(\rho, \rho')$
  if $|\mathcal{L}| \geq \eta$ **then return** UNTIMEDUNTIL$^{2}_{\Theta}(\rho, \rho')$ ▷ Algorithm 9
  else return UNTIMEDUNTIL$^{1}_{\Theta}(\rho, \rho')$ ▷ Algorithm 9
  end if

function HYBRIDANDFUTURE$^{\gamma}_{\Theta}(\rho, \rho')$
  if $\epsilon > \gamma$ **then return** ANDFUTURE$^{2}_{\Theta}(\rho, \rho')$ ▷ Algorithm S1
  else return ANDFUTURE$^{1}_{\Theta}(\rho, \rho')$ ▷ Algorithm S2
  end if

function HYBRIDANDGLOBALLY$^{\gamma}_{\Theta}(\rho, \rho')$
  if $\epsilon \geq \gamma$ **then return** ANDGLOBALLY$^{2}_{\Theta}(\rho, \rho')$ ▷ Algorithm S1
  else return ANDGLOBALLY$^{1}_{\Theta}(\rho, \rho')$ ▷ Algorithm S2
  end if

7.2. Relational Temporal Mining

We now move from synthetic data, required to tune hybrid algorithms and thoroughly test our operators, towards real data benchmarks with no data payload conditions. We contextualise our experiments for data-intensive model mining operations that can also be run on a relational model. While doing so, we compare our runtimes both with hybrid operators with the one from the previous paper [4], as well as run times from the relational model with traditional SQL queries.

SQLMiner, provided by Schonig et al. [5], utilises database architectures for declarative process mining. We chose to test our hypothesis of engineering a custom database architecture against state-of-the-art traditional relational databases (**PostgreSQL 14.2**). For this set of experiments, we exploited the BPIC 2011 (Dutch academic hospital log) dataset (https://dx.doi.org/10.17605/OSF.IO/2CXR7), as used in [5]. This log contained data payload information, though the queries executed as [5] were comprised of data-less events. The original dataset was sampled into sub-logs containing 10, 100, and 1000 traces, and the sampling approach adopted the same behaviour as the synthetic dataset from the previous set-up, where each sub-log is guaranteed to be a subset of the greater ones. Increased sizes of datasets exhibited exponential increases in primary memory requirements and thus justifies our sampling approach. Schönig [48] provides the templated implementations for mining eight declarative clauses. As these are only templates, the models were instantiated from the resulting combinations of the five most occurring events. Therefore, we generated eight models, each consisting of 25 clauses. SQLMiner simulated this by creating a secondary Actions table, with each row containing the instantiated Declare template. SQLMiner provides the Support values associated with each clause. We extend this to also provide trace information, where each clause also contains the traces satisfying it. We also want to test our hypothesis that our proposed hybrid operator pipeline (Section 7.1) can outperform the pipeline set up from our previous work [4] that does not exploit the potential gains that can be made from picking the best algorithm according to the data conditions, and only uses our defined VARIANT-1 operators. The outcomes of these experiments are shown in Figure 11, where each plot represents the execution times for a given elected template, with the more complex queries located on the first row.

SQLMiner results. In the worst case, our running time is comparable with SQLMiner (Response). Even for this case, SQLMiner returns only the Support information, while KnoBAB also returns (for the same execution time) trace information. In SQL, providing the least possible query alterations to provide the trace information causes $10^{1.5}$ run time increase, thus demonstrating that we are more performant on the same conditions. Conversely, in the best case, we outperform SQLMiner by over five orders of magnitude. By exploiting efficient database design, our custom query plan can minimise data access and our computation avoided explicit computations of aggregations. In addition, guaranteeing that the intermediate results are always sorted allows for **linear** scanning cost for counting operations. Responded Existence is a clear candidate for demonstrating the gains

from custom database design: with access to our proposed $\mathsf{CountingTable}_{\mathcal{L}}$, our solution requires only a table look-up, while SQLMiner requires an aggregation requiring an **entire** scan of the Log table. Combining this with the extended $\mathtt{xtLTL}_{\mathrm{f}}$ operators allows for much more optimised query times; this is shown in the results, where KnoBAB is consistently at least two orders of magnitude more performant with queries returning trace information. As $|\mathcal{L}|$ increases beyond 10^2, the more complex queries were unable to finish to completion for SQLMiner, exceeding the 16 GB primary memory of the benchmarking machine.

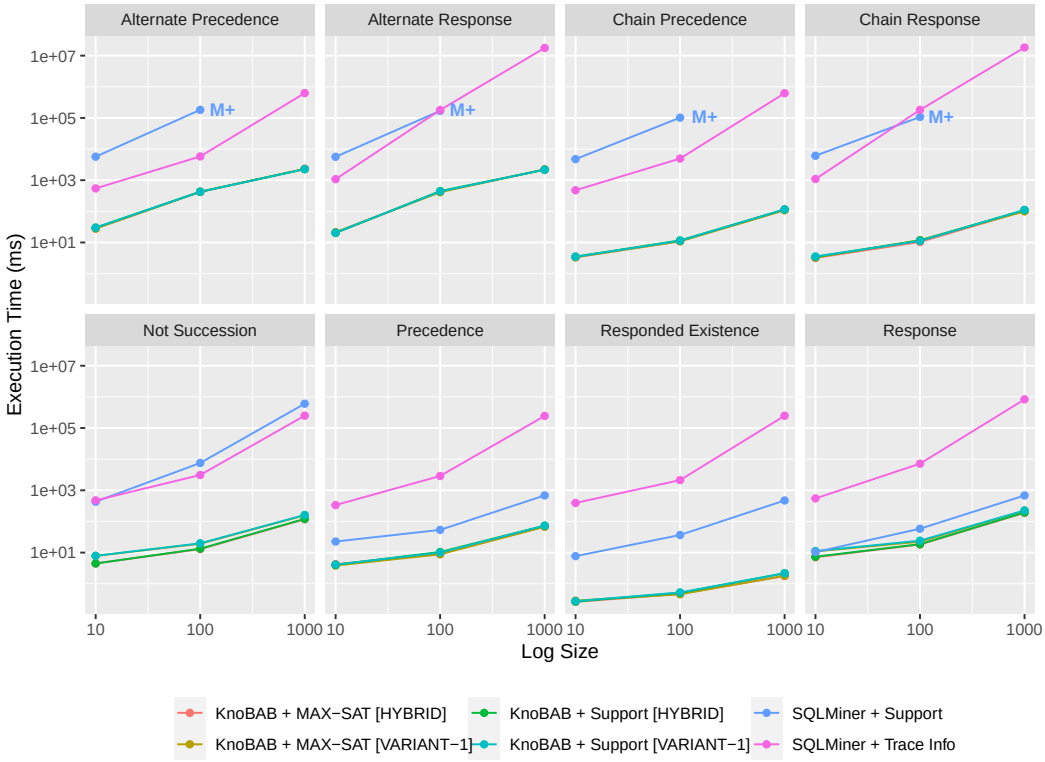


Figure 11. Results for relational temporal mining Section 7.2.

Pipeline results. The execution times for KnoBAB + Support and KnoBAB + MaxSAT are comparable, while there is much greater variation for SQLMiner + Support and SQLMiner + Trace Info. As support requires only an aggregation over intermediate results (Section 5.2.3), we guarantee that we suffer at most a cost proportional to the model size, so we expect a constant overhead based on model size. The large fluctuation in results for SQLMiner is a culprit of the query rewriting provided by the PostgreSQL query engine; in some cases, returning trace information yielded better results. In these experiments, we combined the alternate ensemble methods with our proposed HYBRID operators. The results demonstrate that, for most operators, there is a marginal improvement in time complexity. For NotSuccession and Response, the improvement is more apparent, with the former, for $|\mathcal{L}| = 10$ providing 20% improvement against VARIANT-1. The reader is encouraged to refer back to Figure 10 to explain this. The faster operators thrive with $|\mathcal{L}| > 10^3$, while, for traces within the region of 10^2, the gain is much less apparent. The BPIC_2011 dataset has a corresponding average trace length of ∼220: exploiting the VARIANT-2 operators within this region will therefore yield lesser benefit than much larger $|\mathcal{L}|$.

7.3. Query Plan Parallelisation

By keeping the immediately preceding experimental setting while considering the whole log as well as extending the model size, we now benchmark our solution in a multithreaded environment,

where we perform intra-query parallelism by running each operator laying in the same layer in parallel as per previous discussions.

The correctness of our proposed parallelisation approach is guaranteed by the fact that each thread in a given layer can operate independently with no interdependencies requiring costly mutual exclusions. In place of directly using the `pthread` C++ library on multiple tasks, we utilised a thread pool proposed by [49], to minimise the thread creation overhead, while feeding the pool with the tasks denoted by **for ...(parallel) do** statements in our pseudocode Algorithm 6. We extended the library to support both static and dynamic scheduling approaches proposed by the OpenMP specifications [50]; these are:

- **BLOCKED STATIC** : aims to balance the chunk sizes per thread by distributing any leftover iterations;
- **BLOCK-CYCLIC STATIC**. Does not utilise balancing as the former. Instead, work blocks are cyclically allocated over the threads;
- **GUIDED DYNAMIC**: aims to distribute large chunks when there is a lot of work still to be completed; tasks are split into smaller chunks as the work load diminishes;
- **MONOTONIC DYNAMIC**: uses a single centralised counter that is incremented when a thread performs an iteration of work. The schedule issues iterations to threads in an increasing manner.

In addition to these, we also implemented two different scheduling policies splitting the tasks to be run in parallel while estimating the running time that each operator will take depending on the size of its associated operands (if any).

- **TASK SIZE PREDICTION BLOCK STATIC** provides an estimation of work required per chunk. Then, these chunks are sorted in ascending work load, with the last providing the greatest amount of computation. Threads are then assigned chunks through a distribution algorithm, distributing the first and last chunk of the sorted work to the first thread, the second and penultimate to the second, etc.. The algorithm aims to distribute equal amounts of work to each thread, though assumes that the workload is strictly increasing while workload sizes are evenly distributed;
- **TASK SIZE PREDICTION UNBALANCED DYNAMIC**: unlike the former, we assume that the incoming work is not balanced. Instead, a chunk is taken, its work size estimated and assigned to a thread. Then, the next thread will recursively receive chunks until the summed work load is approximate to that of the former. The next thread is then pulled from the pool and the process repeated until all chunks are assigned.

For this set of experiments, we exploited the full BPIC 2011 (Dutch hospital log) dataset. We want to determine how varying the total number of threads affects execution time, and therefore use only the original dataset with no sampling. This also demonstrates the performance against the real-world scenario. Similarly to the previous mining approach in Section 7.2, we generated models from the most occurring events labels. Here, we extended the model size to consider the top 15 events for the same eight Declare templates, thus resulting in 225 clauses. Extending the model size as such allows a better scalability analysis on the large; in fact, a smaller model size would not be able to reap the benefits of the dissected query plan, as it becomes more likely that there will not be enough work to allocate; as more threads might be left idle in the pool, no speed-up can be achieved.

The results of our experiments are shown in Figure 12. Across all instances, the parallelisation pipeline (line with data-points) proves more performant than any single threaded executions (horizontal vertical bar). There also appears to be a great variation in speed-up for different scheduling policies; MONOTONIC DYNAMIC, TASK SIZE PREDICTION UNBALANCED DYNAMIC, and GUIDED DYNAMIC consistently perform worse than all others. In addition to this, the former schedules grant almost no gain with trace number, indicating that dynamic scheduling is not only less performant than static in our use case scenario, but also bears no potential gains by through thread scalability. This is especially true in the case of Alternate Precedence, where all static policies have improved performance by

at least an order of magnitude. Schedules also show different degrees of speed-ups. For the dynamic and BLOCK-CYCLIC STATIC schedules, increasing the number of threads has little effect on performance. In fact, adding threads proves to be **detrimental** in some cases (BLOCK-CYCLIC STATIC & Chain Precedence). Conversely, the other static schedules (BLOCKED STATIC and TASK SIZE PREDICTION BLOCK STATIC) achieve a super-linear speed-up [51–53], as the thread count increases. The greatest gains in performance were found for Alternate Precedence and Alternate Response with thread sizes of eight; there are over two orders of magnitude improvement against a single threaded instance, and almost the same speed up compared with the static schedules. As our problem is heavily bounded on data access and on the size of it, reducing the task allocation size will create an overall increase of cache misses, while these are minimised by associating each thread with a greater amount of tasks.

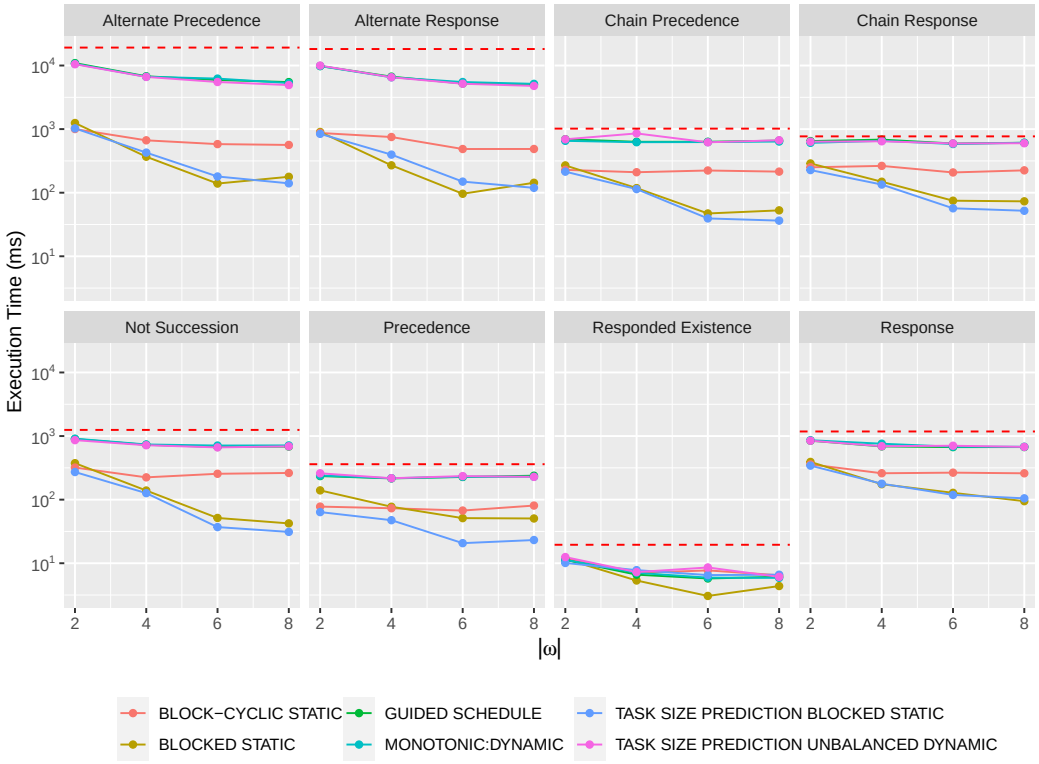


Figure 12. Results for parallelisation Section 7.3. ω indicates the set of threads in the thread pool, and the red dashed horizontal lines indicate running times for single threaded instances.

7.4. $\mathcal{D}_\varphi$-Encoding Atomisation Strategies

We now want to test how distinct query atomisation strategies affect the query run time. For this, we exploit a different dataset while we hardcoded some models suitable for highlighting such differences.

While the `AtomizeEverything` strategy guarantees that all activation and targets undergo the atomizaiton step if a clause is found that contains a data payload predicate, the `AtomizeOnlyOnDataPredicate` atomises only those conditions containing a data payload and considers the others as activity labels. As a consequence, the former is expected to have more weighted access to $\mathsf{AttributeTable}_{\mathcal{L}}$, while the latter to $\mathsf{ActivityTable}_{\mathcal{L}}$. We analyse the execution times over the same models M_1–M_5, where each model differs from the other in the number of clauses as well as in data conditions.

For these experiments, we exploited the full BPIC 2012 (Dutch loan company) dataset. This contained event/trace payload information and was comprised of activities occurring for a loan transaction. The models exploited are visualised in Supplement Table S1a. We define four models, increasing by five clauses, where each is a sub-model of the latter.

These clauses consisted of both data and data-less payload conditions, in order to adhere to our benchmarking hypothesis.

Results are shown in Figure 13 for both configurations, where there is a positive correlation between model size and execution time, with a constant increase with each additional set of clauses. For the smaller model size, `AtomizeEverything` outperforms `AtomizeOnlyOnDataPredicate`, though the former exhibits greater increases in running time as more clauses are added. This therefore suggests that accessing the $\mathsf{ActivityTable}_{\mathcal{L}}$ becomes more expensive than the $\mathsf{AttributeTable}_{\mathcal{L}}$ as the number of activation/target conditions increases. To explain this, the reader is encouraged to refer back to Supplement Table S1a, which defines the clauses that are added to each model, and therefore the new activities and atoms that may require decomposition. With increased model sizes, `AtomizeOnlyOnDataPredicate` suffers from duplicated memory access; as some events (e.g., A_SUBMITTED) are accessed in both tables: while returning the events satisfying an atom requires the access to the $\mathsf{AttributeTable}^{k}_{\mathcal{L}}$ for any given attribute k of interest, returning all of the events having a given activity label requires accessing the $\mathsf{ActivityTable}_{\mathcal{L}}$. The data access for the atomised queries may duplicate access to the $\mathsf{ActivityTable}_{\mathcal{L}}$, which becomes more costly as our model size increases. Conversely, `AtomizeEverything` will atomize A_SUBMITTED from q_1, as clauses q_2 and q_3 contain payload conditions. Therefore, these queries only ever access the $\mathsf{AttributeTable}_{\mathcal{L}}$, and the duplication of data access is removed. For the smaller model size M_1, this gain is less apparent as the duplicated data access becomes negligible.

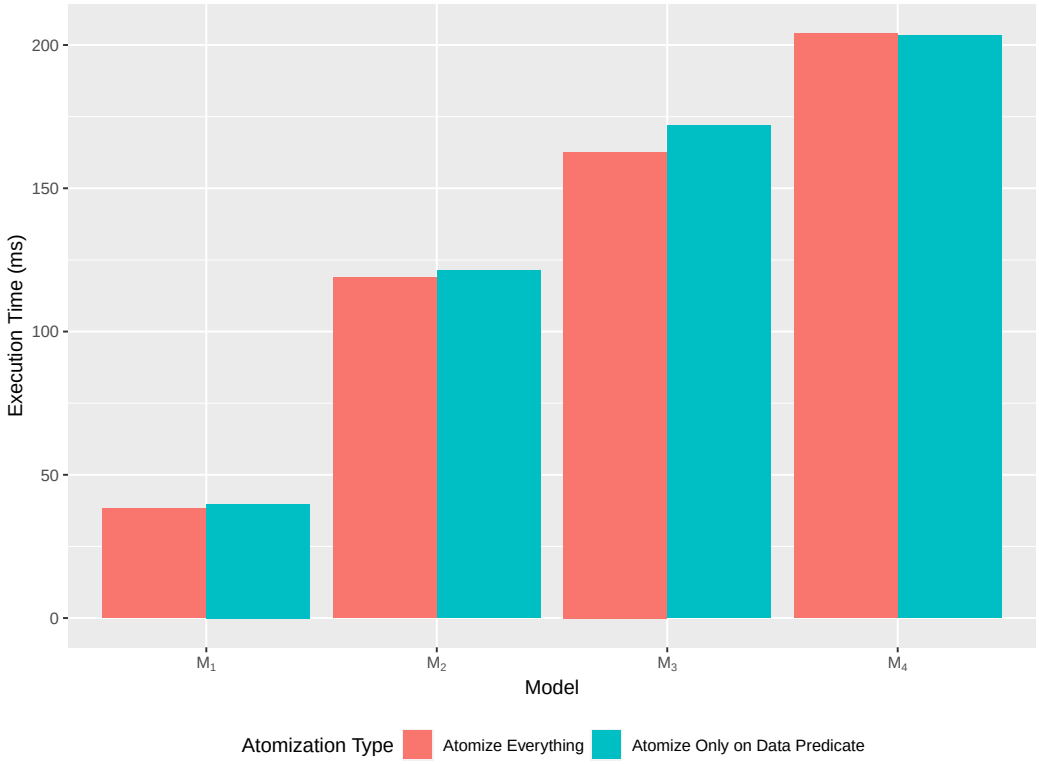


Figure 13. Running times over different models (Table S1a) for different atomisation strategies.

7.5. Data-Aware Conformance Checking

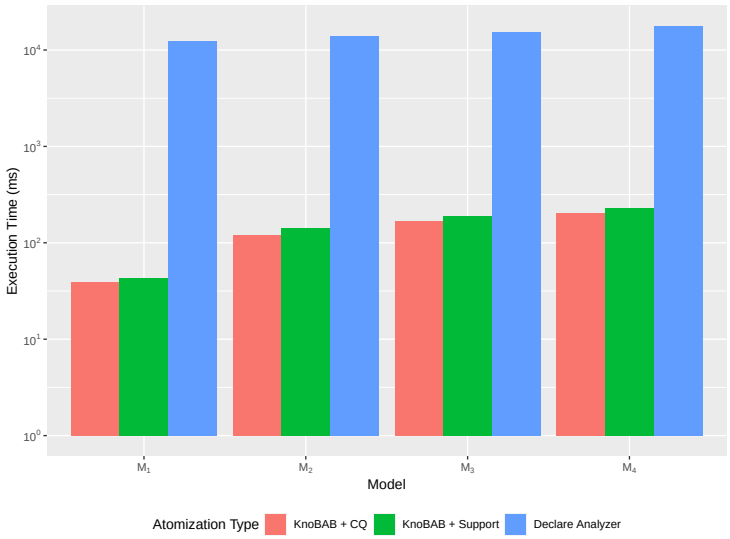
We now consider another state-of-the-art solution, Declare Analyzer [6] for conformance checking with payload information. This solution is tested against two different sets of models of increasing sizes, with each of them providing either the worst or the best case scenario for KnoBAB. These experiments exploit the same dataset as in the former experimental set-up, and also used in [6].

We represented the log for Declare Analyzer via MapDB (https://mapdb.org/), thus reflecting a relational model representation. The authors do not consider trace payloads, and therefore propose injecting trace payload as an extension of each event payload. On

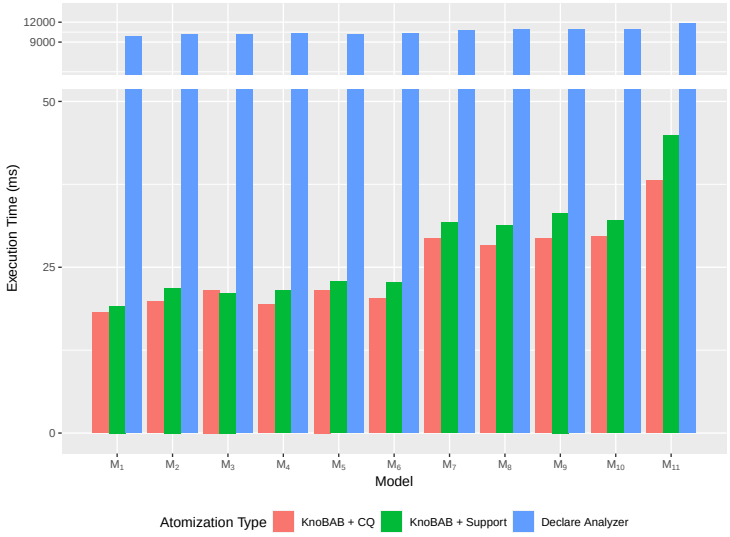
the other hand, KnoBAB injects the trace payload as a *unique* event at the beginning of the trace (Section 2), thus reducing the overhead of testing an activation/target condition *per event* while minimising data loading time. We wanted to investigate our solution's performance among the best/worst cases regarding the clauses of choice. Therefore, we provide two scenarios. The first scenario (SCENARIO 1), also described in our seminal paper [4], provides our *worst* case scenario models (Table S1a) where each additional set of clauses consist of entirely novel activity labels and clauses and, within each sub-model, each clause is distinguished by data payload conditions. Consequently, the query plan cannot exploit gains made from data access minimisation as every condition is considered a unique disjunction of atoms. Conversely, the second (SCENARIO 2) novel scenario describes our best case. We encourage the reader to refer to this, where activation and target conditions appear several times in different clauses (Table S2). Thus, there are many more instances where data access can be minimised; for example, the model $q_1 \wedge q_2 \wedge q_3 \wedge q_4 \wedge q_5$ considers the activity label A_SUBMITTED across five instances. Following strategies such as in [9], this can be reduced to one access. SCENARIO 1 (SCENARIO 2) results are shown from Figure 14a (Figure 14b). For either scenario, we average 2–3 orders of magnitude more performant than Declare Analyzer; even in the worst case (M_4), we are over an order of magnitude more performant. For both scenarios, we compute the following metrics: Conjunctive Query (CQ) and Support, to analyse any variations between the ensemble methods. KnoBAB + CQ outperforms KnoBAB + Support in all cases, where the cost increase is linear with model size.

SCENARIO 1. For Declare Analyzer, increases in model size results in a constant slope of 3.47×10^2 ms per model size, while our solution demonstrates an initial slope of 2×10^1 ms per model size, followed by a constant slope of 6×10^0 ms per model size. To explain this abrupt behaviour, the reader is encouraged to refer to Supplement Table S1a and the query plan from Figure 2. KnoBAB thrives when data access is minimised; if this cannot be achieved (due to the addition of novel activation/target conditions), potential gains cannot be exploited. **Every** clause from M_2 contains new activation/target labels/payload conditions compared to M_1. As a result, the number of atoms and leaves in the query plan is doubled. However, M_3 contains the activity label O_CANCELLED. This atom has already been considered in the previous model, and so data access is optimised. Therefore, the time increase from M_2 to M_3 is much less than that of the former. Subsequently, as M_3 is a sub-model of M_4, the same gains are seen here (M_4 contains entirely novel conditions). Overall, the results show that we are not bounded by model size unlike Declare Analyzer, which must perform an entire log scan per clause, while we can ignore irrelevant traces via bounding/indexing across our tabular representation available to the relational model. Still, our running times reflect the formal definition stated in Section 5.2.3, where queries still need to scan each model clause and therefore their expected running time is proportional to the model size.

SCENARIO 2. We now want to test whether clauses providing similar queries lead to lower running times. Here, the model sizes are smaller than the previous example, so as to demonstrate the potential optimisation from even small examples. The former contains only a single clause, while the latter consists of seven clauses. The slope between these models is 3.3×10^0 ms per model size, an order of magnitude less than the worst case scenario. To clarify the results, the reader is encouraged to compare the models q_1 vs. $q_1 \wedge q_2 \wedge q_3 \wedge q_4 \wedge q_5$. All atoms in the former are included in the latter, so we can have much greater data access minimisation, which these results confirm. Of course, a hand-made model is unlikely to contain such overlapping elements, but these results demonstrate the potential gains to be made, even for less bespoke scenarios such as data mining, where a huge amount of overlap might still occur while testing multiple clauses' combinations.



(**a**) Scenario 1 (Table S1a).



(**b**) Scenario 2 (Table S2).

Figure 14. Running times for data-aware conformance checking.

8. Conclusions

By summarizing the contributions of our paper, we showed how to express temporal logic through ad hoc temporal algebra ($\texttt{xtLTL}_\text{f}$) based on the relational model. The latter, defined both in its logical and physical model, has been suitably extended for log and operators' result representation. We showed how it is possible to load data on this model using suitable algorithms and how it is possible to represent a sequence of operations with a parallelisable query plan providing super-linear speed-up. As a new contribution to our previous work, we have also shown different implementations for the $\texttt{xtLTL}_\text{f}$ operators, thus showing how there is always a faster non-trivial implementation exploiting both the properties of the intermediate result representation as well as query rewriting. Our proposed solution, KnoBAB, leverages all of the aforementioned features, thus providing higher performance than current conformance checking and mining solutions, be it data or data-less.

This work encourages future KnoBAB developments and implementations, including more efficient data model mining algorithms and the use of views to reduce further the cost of allocating intermediate results. Furthermore, secondary memory representation of the

log according to the percepts of Near Data Processing is in its infancy. Future developments will explore the possibility of using KnoBAB to learn temporal models from data and the ability to fully support trace repair operations in order to make deviant traces compliant to the given model. For this, we will consider the possibility of integrating our relational system with the BCDM relational model [54], thus fully supporting operations such as insertions, updates, and deletions required for trace repairs in conformance checking [25]. Finally, our future work will also consider vectorial data as a specific data representation [32,55]: this will enable KnoBAB to fully support spatial data representation, thus aiming for full spatio-temporal representation [56,57]. This, along with more advanced model mining algorithms, will enable us to efficiently mine spatio-temporal patterns from logs. Finally, we will also investigate the possibility of transferring the definition of such algebraic operators when logs are represented as graphs [58,59], thus further improving the efficiency of graph-based query languages.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/info14030173/s1.

Author Contributions: *Conceptualisation*, S.A. and G.B.; *methodology*, G.B.; *software*, S.A. and G.B.; *validation*, S.A. and G.B.; *formal analysis*, G.B.; *investigation*, S.A.; *resources*, G.B. and G.M.; *data curation*, S.A.; *writing—original draft preparation*, S.A. and G.B.; *writing—review and editing*, G.B. and G.M.; *visualisation*, S.A.; *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.

Informed Consent Statement: Not applicable.

Data Availability Statement: The dataset associated with the presented experiments was available online the 5 March 2022: https://dx.doi.org/10.17605/OSF.IO/2CXR7.

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 (5 March 2022).

Abbreviations

The following abbreviations are used in this manuscript:

DAG	Direct Acyclic Graph
KnoBAB	KNOwledge Base for Alignments and Business process modelling
LTL_f	Linear Temporal Logic over finite traces
RDBMS	Relational Database Management System
XES	eXtensible Event Stream
$\texttt{xt}\text{LTL}_\text{f}$	eXTended Linear Temporal Logic over finite traces

Appendix A

We now show some equivalence and correctness lemmas.

Appendix A.1

First, we want to show the equivalence of some unary operators as generalisations of some of the base operators. We now show that $\mathsf{Init}^{\mathcal{L}}_{A/T}$ or $\mathsf{Ends}^{\mathcal{L}}_{A/T}$ can be subsumed by appropriate combinations of Init or Ends with $\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}$. As the former set of operators cannot express data conditions for the events while the former can by replacing $\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}$ with an arbitrary sub-expression with $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$, we can trivially conclude that the former are less general than the latter.

Lemma A1.

$$\forall a \in \Sigma. \quad \mathit{Init}^{\mathcal{L}}_{A/T}(a) = \mathit{Init}(\mathit{Activity}^{\mathcal{L},\tau}_{A/T}(a))$$

Proof. We can expand the definition of the left-hand side of the equation for any $\mathsf{a} \in \Sigma$ as follows:

$$\mathsf{Init}^{\mathcal{L}}_{A/T}(\mathsf{a}) = \{\ \langle i, 1, \{A/T(1)\}\rangle \mid \exists \phi.\ \langle \beta(\mathsf{a}), i, 1, \bot, \phi\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \}$$

The right-hand side of the equation can be rewritten as follows:

$$\mathsf{Init}(\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a})) = \{\ \langle i, 1, \{A/T(1)\}\rangle \mid \exists \pi, \phi.\ \langle \beta(\mathsf{a}), i, 1, \pi, \phi\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \}$$

The goal is immediately closed by choosing $\pi = \bot$, as any first event will have always an empty `Prev` pointer. □

Lemma A2.

$$\forall a \in \Sigma.\quad \mathit{Ends}^{\mathcal{L}}_{A/T}(a) = \mathit{Ends}(\mathit{Activity}^{\mathcal{L},\tau}_{A/T}(a))$$

Proof. We can expand the definition of the left-hand side of the equation for any $\mathsf{a} \in \Sigma$ as follows:

$$\mathsf{Ends}^{\mathcal{L}}_{A/T}(\mathsf{a}) = \left\{\ \langle i, 1, \{A/T(|\sigma^i|)\}\rangle \;\middle|\; \exists \pi.\ \langle \beta(\mathsf{a}), i, 1, \pi, \bot\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \right\}$$

The right-hand side of the equation can be rewritten as follows:

$$\begin{aligned}\mathsf{Ends}(\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a})) &= \left\{\ \langle i, 1, L\rangle \;\middle|\; \langle i, |\sigma^i|, L\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \right\}\\ &= \left\{\ \langle i, 1, \{A/T(|\sigma^i|)\}\rangle \;\middle|\; \exists \pi.\ \langle \beta(\mathsf{a}), i, |\sigma^i|, \pi, \bot\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \right\}\end{aligned}$$

The goal is immediately closed by choosing $\pi = \bot$, as any first event will always have an empty `Prev` pointer. □

On the other hand, as the $\mathsf{Exists}^{\mathcal{L}}_{A/T}$ and $\mathsf{Absence}^{\mathcal{L}}_{A/T}$ operators discard the activation and target marks of the associated events for the purposes of efficiency, we need to relax their notion of equivalence by ignoring the result being provided by the third component. Still, we can observe that they compute the same result trace-wise. Even in this scenario, as the former operators merely access the counting table for the purposes of efficiency, they cannot be generally exploited when the expression of data conditions is also required.

Lemma A3.

$$\forall a \in \Sigma. \forall \sigma^i \in \mathcal{L}. \exists L, L'.\ \langle i, 1, L\rangle \in \mathit{Exists}^{\mathcal{L}}_{A/T}(a, n) \Leftrightarrow \langle i, 1, L'\rangle \in \mathit{Exists}_n(\mathit{Activity}^{\mathcal{L},\tau}_{A/T}(a))$$

Proof.

$$\begin{aligned}\langle i, 1, L\rangle \in \mathsf{Exists}^{\mathcal{L}}_{A/T}(\mathsf{a}, n) &\Leftrightarrow \langle i, 1, L'\rangle \in \mathsf{Exists}_n(\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a}))\\ \exists m \geq n.\ \langle \beta(\mathsf{a}), i, m\rangle \in \mathsf{CountingTable}_{\mathcal{L}} &\Leftrightarrow n \leq |\left\{\ \langle i, j, L'\rangle \in \mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a})\ \right\}|\\ \left|\left\{\ \sigma^i_j \in \sigma^i \;\middle|\; \sigma^i_j = \langle \mathsf{a}, p\rangle\ \right\}\right| \geq n &\Leftrightarrow n \leq |\{\ \langle \beta(\mathsf{a}), i, j, \pi, \phi\rangle \in \mathsf{ActivityTable}_{\mathcal{L}}\ \}|\end{aligned}$$

□

Lemma A4.

$$\forall a \in \Sigma. \forall \sigma^i \in \mathcal{L}. \exists L, L'.\ \langle i, 1, L\rangle \in \mathit{Absence}^{\mathcal{L}}_{A/T}(a, n) \Leftrightarrow \langle i, 1, L'\rangle \in \mathit{Absence}_n(\mathit{Activity}^{\mathcal{L},\tau}_{A/T}(a))$$

Proof. By simply replacing the $m \geq n$ and $n \leq |S|$ for any set S conditions in the former lemma to $m < n$ and $n > |S|$. This boils down to:

$$\langle i, 1, L \rangle \in \mathsf{Absence}^{\mathcal{L}}_{A/T}(\mathsf{a}, n) \Leftrightarrow \langle i, 1, L' \rangle \in \mathsf{Absence}_n(\mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a}))$$
$$\exists m < n.\, \langle \beta(\mathsf{a}), i, m \rangle \in \mathsf{CountingTable}_{\mathcal{L}} \Leftrightarrow n > |\left\{ \langle i, j, L' \rangle \in \mathsf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a}) \right\}|$$
$$\left|\left\{ \sigma^i_j \in \sigma^i \;\middle|\; \sigma^i_j = \langle \mathsf{a}, p \rangle \right\}\right| < n \Leftrightarrow n > |\{ \langle \beta(\mathsf{a}), i, j, \pi, \phi \rangle \in \mathsf{ActivityTable}_{\mathcal{L}} \}|$$

□

Appendix A.2

Next, we want to show that $\mathtt{xt}\mathrm{LTL_f}$ is at least as expressive as $\mathrm{LTL_f}$. To support this claim, we need to prove the two following lemmas where, as $\mathrm{LTL_f}$ does not support explicit activation and target conditions with Θ correlation conditions over the payload data, we are always going to assume $\Theta =$ **True** and that the atomic operators are never associated with an activation/target label, thus always returning an empty third component of the intermediate result. As we might observe, the following lemma entails that, differently from standard $\mathrm{LTL_f}$ semantics applied to each event trace at a time, $\mathtt{xt}\mathrm{LTL_f}$ semantics returns all of the events for which the given temporal condition holds. This becomes very relevant for minimising the data access while scanning our relational representation of the log, as well as allowing better intermediate result reuse for any incoming sub-expression. The following lemma also entails a correspondence between timed $\mathtt{xt}\mathrm{LTL_f}$ operators and $\mathrm{LTL_f}$ formulae.

Lemma A5. *For each* $\mathrm{LTL_f}$ *formula* φ*, a timed* $\mathtt{xt}\,\mathrm{LTL_f}$ *expression* ψ^τ *evaluated over an intended relational model representing a log* $\mathcal{L}$ *of finite and non-empty traces exists for which the latter returns* $\langle i, j, L \rangle$ *iff.* $\sigma^i_j \vDash \varphi$*. More formally:*

$$\forall \sigma^i_j \in \sigma^i, \sigma^i \in \mathcal{L}.\, \forall \varphi \in \mathrm{LTL_f}.\, \exists \psi^\tau : \mathtt{timed\ xt}\mathrm{LTL_f}.(\langle i, j, L \rangle \in \psi^\tau \Leftrightarrow \sigma^i_j \vDash \varphi)$$

Proof. The constructive proof proceeds by structural induction over ψ^τ. We first need to consider a rewriting lemma stating that $\langle \beta(\mathsf{a}), i, j, \pi, \chi \rangle \in \mathsf{ActivityTable}_{\mathcal{L}}$ iff. a p exists such that $\sigma^i_j = \langle a, p \rangle$. Now, we can start the proof by induction.

$\varphi = \mathsf{a}$**:** By applying the aforementioned rewriting lemma (from now on simply referred to as *by construction of ActivityTable*), we can immediately close the goal by choosing $\psi^\tau = \mathsf{Activity}^\tau(\mathsf{a})$ as the model will only return data associated with the log of choice:

$$\langle i, j, L \rangle \in \mathsf{Activity}^\tau(\mathsf{a}) \Leftrightarrow \exists p. \sigma^i_j = \langle \mathsf{a}, p \rangle \Leftrightarrow \sigma^i_j \vDash \mathsf{a}$$

$\varphi = \mathsf{a} \wedge q$**:** If the compound condition is also atomic for which q can be expressed as an interval query $low \leq \kappa \leq up$ for some payload key κ, we can follow a similar proof from the former case and choose the atom $\psi^\tau = \mathsf{Compound}^\tau(\mathsf{a}, \kappa, low, up)$, thus closing the goal as follows:

$$\begin{aligned}\langle i, j, L \rangle \in \mathsf{Compound}^\tau(\mathsf{a}, \kappa, [low, up]) &\Leftrightarrow \exists p. \sigma^i_j = \langle a, p \rangle \wedge low \leq p(\kappa) \leq up \\ &\Leftrightarrow \sigma^i_j \vDash \mathsf{a} \wedge (low \leq \kappa \wedge \kappa \leq up)\end{aligned}$$

$\varphi = \bigcirc, \varphi'$**:** by inductive hypothesis, we know the ρ $\mathtt{xt}\mathrm{LTL_f}$ expression returning ρ, which contains $\langle i, j+1, L \rangle$ when $\sigma^i_{j+1} \vDash \varphi'$. For this, we choose as $\psi^\tau = \mathsf{Next}^\tau(\rho)$, which

also guarantees that j never exceeds the trace's length ($j \leq |\sigma^i|$). We can therefore expand the definition of our proposed operator by obtaining:

$$\begin{aligned}\langle i,j,L\rangle \in \mathsf{Next}^\tau(\rho) &\Leftrightarrow \langle i,j+1,L\rangle \in \rho \wedge 1 < j+1 \leq |\sigma^i| \\ &\overset{IH}{\Leftrightarrow} \sigma^i_{j+1} \vDash \varphi \wedge 0 < j < |\sigma^i| \\ &\Leftrightarrow \varphi^i_j \vDash \bigcirc, \varphi\end{aligned}$$

$\varphi = \Box\varphi'$**:** The application of the induction is similar to the former and, similarly to the former case, we also proceed by expanding the definition of the relational operator. We can hereby choose $\psi^\tau = \mathsf{Globally}^\tau(\rho)$ where the induction is applied over ρ and φ'. We can close the goal as follows:

$$\begin{aligned}\langle i,j,L\rangle \in \mathsf{Globally}^\tau(\rho) &\Leftrightarrow \langle i,j,L_j\rangle \in \rho \wedge |\sigma^i| - j + 1 = |\{\langle i,k,L_k\rangle \in \rho | j \leq k \leq |\sigma^i|\}| \\ &\Leftrightarrow \forall j \leq k \leq |\sigma^i|.\, \langle i,k,L_k\rangle \in \rho \\ &\overset{IH}{\Leftrightarrow} \forall j \leq k \leq |\sigma^i|.\sigma^i_k \vDash \varphi' \\ &\Leftrightarrow \sigma^i_j \vDash \Box\varphi'\end{aligned}$$

$\varphi = \Diamond\varphi'$**:** Similarly to globally, we obtain $\psi^\tau = \mathsf{Future}^\tau(\rho)$ for a ρ corresponding to φ' by inductive hypothesis.

$\varphi = \neg\varphi'$**:** Similarly to the previous unary operators, we choose as $\mathtt{xtLTL_f}$ operator $\psi^\tau = \mathsf{Not}^\tau(\rho)$ where the inductive hypothesis links ρ to φ'. We can therefore close the goal as follows:

$$\begin{aligned}\langle i,j,L\rangle \in \mathsf{Not}^\tau(\rho) &\Leftrightarrow \sigma^i_j \in \sigma^i \wedge \sigma^i \in \mathcal{L} \wedge \langle i,j,L\rangle \notin \rho \\ &\overset{IH}{\Leftrightarrow} \sigma^i_j \in \sigma^i \wedge \sigma^i \in \mathcal{L} \wedge \sigma^i_j \vDash \neg\varphi \\ &\Leftrightarrow \sigma^i_j \vDash \neg\varphi\end{aligned}$$

This is doable as stating $\langle i,j,L\rangle \in \psi^\tau \Leftrightarrow \sigma^i_j \vDash \varphi$ is equivalent to $\langle i,j,L\rangle \notin \psi^\tau \Leftrightarrow \sigma^i_j \nvDash \varphi$ where the latter can be rewritten as $\sigma^i_j \vDash \neg\varphi$.

$\varphi = \varphi' \wedge \varphi''$**:** As we have that two inductive hypotheses associate ρ' and ρ'' respectively to φ' and φ'', we choose the $\mathtt{xtLTL_f}$ formula $\psi^\tau = \mathsf{And}^\tau_{\mathbf{True}}(\rho',\rho'')$ to be associated with $\varphi' \wedge \varphi''$. For this $\mathtt{xtLTL_f}$ operator, we can state that a result $\langle i,j,\ell\rangle$ is returned by such an operator if and only if $\langle i,j,\emptyset\rangle \in \rho'$ and $\langle i,j,\emptyset\rangle \in \rho''$ per definition of operators never returning explicit activation or target condition. We close the goal as follows:

$$\begin{aligned}\langle i,j,L\rangle \in \mathsf{And}^\tau_{\mathbf{True}}(\rho,\rho') &\Leftrightarrow \langle i,j,\emptyset\rangle \in \rho \wedge \langle i,j,\emptyset\rangle \in \rho' \\ &\overset{IH}{\Leftrightarrow} \sigma^i_j \vDash \varphi \wedge \sigma^i_j \vDash \varphi' \\ &\Leftrightarrow \sigma^i_j \vDash \varphi \wedge \varphi'\end{aligned}$$

$\varphi = \varphi' \vee \varphi''$**:** We can firstly observe that $(A \wedge B) \vee (A \wedge \neg B) \vee (\neg A \wedge B)$ in the classical semantics is equivalent to $A \vee B$ for any possible proposition A and B (*OrRwLem*). After observing that the current operator is defined by extension of the previously proved one, we can exploit the previous one as a rewriting lemma. As we have that two inductive hypothesis associating ρ' and ρ'' respectively to φ' and φ'', we choose the $\mathtt{xtLTL_f}$ formula $\psi^\tau = \mathsf{Or}^\tau_{\mathbf{True}}(\rho',\rho'')$ to be associated with $\varphi' \vee \varphi''$. We close the goal as follows:

$$\langle i,j,L\rangle \in \mathsf{Or}^{\tau}_{\mathbf{True}}(\rho,\rho') \overset{IH}{\Leftrightarrow} \langle i,j,L\rangle \in \mathsf{And}^{\tau}(\rho,\rho') \vee (\sigma^i_j \vDash \varphi \wedge \sigma^i_j \nvDash \varphi') \vee (\sigma^i_j \vDash \varphi' \wedge \sigma^i_j \nvDash \varphi)$$
$$\Leftrightarrow \sigma^i_j \vDash \varphi \wedge \varphi' \vee (\sigma^i_j \vDash \varphi \wedge \sigma^i_j \nvDash \varphi') \vee (\sigma^i_j \vDash \varphi' \wedge \sigma^i_j \nvDash \varphi)$$
$$\overset{OrRwLem}{\Leftrightarrow} \sigma^i_j \vDash \varphi \vee \varphi'$$

$\varphi = \varphi' \; \mathcal{U} \; \varphi''$**:** as both the results from the third element of the intermediate results are always empty by construction and preliminary assumption, and we have inductive hypothesis associating ρ and ρ' respectively to φ and φ', we can immediately close the goal after choosing the $\mathtt{xt}\mathrm{LTL_f}$ formula $\psi^{\tau} = \mathsf{Until}^{\tau}_{\mathbf{True}}(\rho',\rho'')$ to be associated with $\varphi = \varphi' \; \mathcal{U} \; \varphi''$.

□

The next lemma is required for closing the generic lemma stated at the beginning of this sub-section, as $\mathrm{LTL_f}$ starts assessing the formulae from the beginning of each trace. We need to show that the former lemma applies to $\mathtt{xt}\mathrm{LTL_f}$ operators in a stricter version, which is the following one:

Lemma A6. *For each* LTL_f *formula* φ *satisfied from the beginning of the trace, it exists an* $\mathtt{xt}\,\mathrm{LTL}_f$ *expression* ψ *returning a* $\langle i,1,L\rangle$*, thus highlighting that the condition holds from the beginning of the trace. More formally:*

$$\forall \sigma^i \in \mathcal{L}. \forall \varphi \colon \mathrm{LTL}_f. \exists \psi \in \mathtt{xt}\mathrm{LTL}_f. (\sigma^i \vDash \varphi \Rightarrow \exists L.\, \langle i,1,L\rangle \in \psi)$$

Proof. Similarly to the previous lemma, as $\mathrm{LTL_f}$ cannot express activation and target conditions to be tested in Θ correlation conditions, we always choose $\Theta = \mathbf{True}$, and we decide to use base $\mathtt{xt}\mathrm{LTL_f}$ operators where none of these conditions is returned. Differently from the previous lemma, we now have to go by inductive structure over the $\mathrm{LTL_f}$ formulae rather than on the $\mathtt{xt}\mathrm{LTL_f}$ ones. We can therefore consider the following inductive cases:

$\varphi = \mathsf{a}$**:** By definition of the Init operator, it is sufficient to consider $\psi = \mathsf{Init}(\mathsf{a})$;

$\varphi = \mathsf{a} \wedge p$**:** Under the assumption that the compound condition corresponds to an atomic query with $p := low \leq \kappa \leq up$, we can formulate the former as follows: $\psi = \mathsf{Init}(\mathsf{Compound}^{\mathcal{L},\tau}(\mathsf{a},\kappa,low,up))$;

$\varphi = \bigcirc, \varphi'$**:** By rewriting this definition, this implies to prove that $\varphi^i_2 \vDash \varphi'$. As the Next^{τ} operator is a timed one and we cannot assess φ' from the beginning of the trace, we cannot exploit the inductive hypothesis for φ', but we need to apply the previously proven lemma for the conditions happening at any point in the trace. From the application of the previous lemma, we have that $\varphi^i_2 \vDash \varphi \Leftrightarrow \langle i,2,L\rangle \in \rho$ for some $\mathtt{xt}\mathrm{LTL_f}$ expression returning ρ. From this, it follows that $\langle i,1,L\rangle \in \mathsf{Next}^{\tau}(\rho)$. By its definition, Next^{τ} returns all events preceding the ones stated in ρ, while, for $\sigma^i \vDash \bigcirc, \varphi$, we are only interested in restricting all of the possible results of Next^{τ} to the ones also corresponding to the beginning of the trace. For this reason, we need to consider ψ as $\mathsf{And}^{\tau}(\mathsf{First}^{\tau}, \mathsf{Next}^{\tau}(\rho))$;

$\varphi = \Box\varphi'$**:** Similarly to the previous operator, φ' is timed and should be checked for all events σ^i_j of interest within the trace σ^i. Even in this case, we need to apply the previous lemma for φ', thus guaranteeing that an $\mathtt{xt}\mathrm{LTL_f}$ expression ρ exists containing $\langle i,j,L\rangle$ whenever $\sigma^i_j \vDash \varphi'$. As globally requires that all of the events satisfy φ', we have that $\mathsf{Globally}(\rho)$ responds by the intended semantics, and therefore we choose this as our ψ;

$\varphi = \Diamond\varphi'$**:** Similarly to the previous operator, we choose $\mathsf{Future}(\rho)$ when ρ is linked to the evaluation of φ' for any possible trace event by the previous lemma;

$\varphi = \neg\varphi'$**:** In this other scenario, we can directly apply the previous lemma, as the evaluation of φ' will always start from the beginning of the trace. After recalling that $\nexists x.P(x) \Leftrightarrow \forall x.\neg P(x)$, we rewrite the definition of φ while applying the inductive hypothesis for the present lemma over some ρ semantically linked to φ' as follows:

$$\sigma^i \vDash \neg\varphi \Leftrightarrow \sigma^i_1 \nvDash \varphi' \overset{IH}{\Leftrightarrow} \forall L.\, \langle i, 1, L\rangle \notin \rho$$

Per inductive hypothesis, ρ contains all of the records $\langle i, 1, L\rangle$ for which $\sigma^i_1 \vDash \varphi'$; as the untimed negation will return a record $\langle \iota, 1, \emptyset\rangle$ if and only if there is no event associated with the trace ι in the provided operand, we can choose $\psi = \mathsf{Not}(\rho)$ and close the goal as follows:

$$\langle i, 1, \emptyset\rangle \in \mathsf{Not}(\rho) \Leftrightarrow \forall j, L.\, \langle i, j, L\rangle \notin \rho \Leftrightarrow \forall L.\, \langle i, 1, L\rangle \notin \rho$$

$\varphi = \varphi'\ \mathcal{U}\ \varphi''$**:** Similarly to the former operators, both φ' and φ'' required a timed evaluation of the events along the trace of interest, for which we need to exploit the former lemma, thus obtaining timed $\mathtt{xt}\mathrm{LTL_f}$ expressions ρ' and ρ''. We can immediately close the lemma by choosing $\psi = \mathsf{Until}_{\mathbf{True}}(\rho', \rho'')$;

$\varphi = \varphi' \wedge \varphi''$**:** Similarly to the negation operator, we can directly apply the inductive hypothesis on φ' and φ'', as these sub-operators will also be assessed from the beginning of a trace; these will be associated respectively to the $\mathtt{xt}\mathrm{LTL_f}$ expressions ρ' and ρ'' having $\langle i, 1, \emptyset\rangle \in \rho'$ and $\langle i, 1, \emptyset\rangle \in \rho''$ as we exploit neither activation nor target conditions. As per construction ρ' and ρ'' will contain no record $\langle i, j+2, L\rangle$ for some natural number $j \geq 0$, we chose $\psi = \mathsf{And}_{\mathbf{True}}(\rho', \rho'')$;

$\varphi = \varphi' \vee \varphi''$**:** By exploiting similar consideration from the former operator, we chose $\psi = \mathsf{Or}_{\mathbf{True}}(\rho', \rho'')$ for some ρ' and ρ'' respectively associated by inductive hypothesis to φ' and φ''.

□

As a corollary of the two given lemmas, we have that $\mathtt{xt}\mathrm{LTL_f}$ is at least as expressive as $\mathrm{LTL_f}$, as any $\mathrm{LTL_f}$ formula can always be computed through an equivalent $\mathtt{xt}\mathrm{LTL_f}$ formula. This validates the decision from our previous work [4] where we expressed the semantics of each template in Declare through a correspondent $\mathtt{xt}\mathrm{LTL_f}$ expression. These were also checked through automated testing Appendix A.2. At this stage, we also want to ascertain that the untimed and timed operators work as expected, that is, that we can mimic the outcome of the timed operators over the timed ones if, for each event $\langle i, j, L\rangle$, we evaluate the corresponding untimed operator over the suffix $\sigma^i_j, \ldots, \sigma^i_{|\sigma^i|}$. This can be proven as follows:

Lemma A7. *For each timed* $\mathtt{xt}\,\mathrm{LTL}_f$ *operator* ψ^τ *containing a result* $\langle i, j, L\rangle$ *over a relational representation of* $\mathcal{L}$*, generate a log of suffixes* $\mathcal{L}' = \{\sigma^{i\oplus j}\}$*, where* $\sigma^{i\oplus j} := \sigma^i_j, \ldots, \sigma^i_{|\sigma^i|}$ *of* σ^i*, and each event is defined as* $\sigma^{i\oplus j}_k := \sigma^i_{j+k-1}$ *for each* $1 \leq k \leq |\sigma^i| - j + 1$*. For this, an* $\mathtt{xt}\,\mathrm{LTL}_f$ *expression* ψ *evaluated over the relational representation of* $\mathcal{L}'$ *always exists such that* $\langle i \oplus j, 1, L\rangle \in \psi$*.*

Proof. We prove the lemma by induction over ψ^τ by considering all of the timed operators having an untimed counterpart. Please observe that we discard the negation Not from our considerations, as we have previously mentioned that the timed and untimed versions of this serve different purposes. We also provide an implementation (https://github.com/datagram-db/knobab/blob/main/tests/ltlf_operators_test.cpp, 5 March 2023) of such proofs via automated testing.

$\psi^\tau = \mathbf{Activity}^{\mathcal{L},\tau}_{A/T}(\mathsf{a})$: This can be trivially closed by choosing $\mathsf{Init}^{\mathcal{L}'}_{A/T}(\mathsf{a})$;

$\psi^\tau = \textbf{Compound}^{\mathcal{L},\tau}_{A/T}(\mathsf{a}, k, \boldsymbol{low}, \boldsymbol{up})$: This can be trivially closed by choosing $\mathsf{Init}(\mathsf{Compound}^{\mathcal{L}'}_{A/T}(\mathsf{a}, k, [low, up]))$;

$\psi^\tau = \textbf{Globally}^\tau(\rho)$**:** After observing that $|\sigma^{i\oplus j}| = |\sigma^i| - j + 1$, we obtain the following condition by operator's expansion, where ρ' is evaluated over $\mathcal{L}'$ as per inductive hypothesis:

$$\begin{aligned}
\langle i, j, L\rangle \in \mathsf{Globally}^\tau(\rho) &\Leftrightarrow L := \cup_{\substack{j\le k\le |\sigma^i|,\\ \langle i,k,L_k\rangle\in\rho}} L_k \wedge\ |\sigma^i| - j + 1 = \left|\{\langle i,k,L_k\rangle \in \rho | j \le k \le |\sigma^i|\}\right| \\
&\Leftrightarrow L := \cup_{\substack{1\le k\le |\sigma^{i\oplus j}|,\\ \langle i\oplus j,k,L_k\rangle\in\rho}} L_k \wedge\ |\sigma^{i\oplus j}| = \left|\{\langle i\oplus j,k,L_k\rangle \in \rho | 1 \le k \le |\sigma^{i\oplus j}|\}\right| \\
&\Leftrightarrow L := \cup_{\langle i\oplus j,k,L_k\rangle\in\rho} L_k \wedge\ |\sigma^{i\oplus j}| = \left|\{\langle i\oplus j,k,L_k\rangle \in \rho\}\right| \\
&\Leftrightarrow \langle i\oplus j, 1, L\rangle \in \mathsf{Globally}(\rho');
\end{aligned}$$

$\psi^\tau = \textbf{Future}^\tau(\rho)$**:** By following similar consideration as per the former operator, we have:

$$\begin{aligned}
\langle i, j, L\rangle \in \mathsf{Future}^\tau(\rho) &\Leftrightarrow L := \cup_{\substack{j\le k\le |\sigma^i|\\ \langle i,k,L_k\rangle\in\rho}} L_k \wedge \exists h \ge j, L.\ \langle i,h,L_h\rangle \in \rho \\
&\Leftrightarrow L := \cup_{\substack{1\le k\le |\sigma^{i\oplus j}|\\ \langle i\oplus j,k,L_k\rangle\in\rho}} L_k \wedge \exists h \ge 1, L.\ \langle i,h,L_h\rangle \in \rho \\
&\Leftrightarrow L := \cup_{\langle i\oplus j,k,L_k\rangle\in\rho} L_k \wedge \exists h, L.\ \langle i,h,L_h\rangle \in \rho \\
&\Leftrightarrow \langle i\oplus j, 1, L\rangle \in \mathsf{Future}(\rho');
\end{aligned}$$

$\psi^\tau = \textbf{And}^\tau_\Theta(\rho_1, \rho_2)$**:** By rewriting the definition of the timed And operator, we obtain the following:

$$\begin{aligned}
\langle i, j, L\rangle \in \mathsf{And}^\tau_\Theta(\rho_1,\rho_2) \Leftrightarrow \exists L_1, L_2.\ &\langle i, j, L_1\rangle \in \rho_1 \wedge \langle i, j, L_2\rangle \in \rho_2 \wedge \\
&L := \mathcal{T}^{E,i}_\Theta([j \mapsto L_1], [j \mapsto L_2]) \wedge L \neq \textbf{False}
\end{aligned}$$

If And contains for both of its operands an event σ^i_j, it follows that there should be at least one match $\sigma^{i\oplus j}_1$ over the corresponding untimed operator $\mathsf{And}_\Theta(\rho', \rho'')$ evaluated over $\mathcal{L}'$. For the latter operator, we can therefore ensure that a j exists and a j' being $j = j' = 1$ and L as well as L' for which the following condition holds:

$$\begin{aligned}
\langle i, j, L\rangle \in \mathsf{And}^\tau_\Theta(\rho_1,\rho_2) \Rightarrow \exists L_1, L_2.\ &\langle i\oplus j, 1, L_1\rangle \in \rho' \wedge \langle i\oplus j, 1, L_2\rangle \in \rho'' \wedge \\
&L := \mathcal{T}^{E,i}_\Theta([1 \mapsto \{L_j |\ \langle i, j, L_j\rangle \in \rho'\}], [1 \mapsto \{L_j |\ \langle i, j, L_j\rangle \in \rho''\}]) \wedge \\
&L \neq \textbf{False} \\
\Leftrightarrow \langle i\oplus j, 1, L\rangle \in \mathsf{And}_\Theta(\rho', \rho'');
\end{aligned}$$

$\psi^\tau = \textbf{Or}^\tau_\Theta(\rho_1, \rho_2)$**:** As this operator is derived from the definition of And^τ_Θ, we can directly close the goal by the previous inductive step if the result represents a match between the elements of the first and second operand. If there were no events that might have been matched, the data come either from the first or from the second operand. As the two cases are symmetric, we just provide proof for the former case. In this situation, we have a $\langle i, j, L\rangle \in \mathsf{Or}^\tau_\Theta(\rho_1, \rho_2)$ corresponding to a $\langle i, j, L\rangle \in \rho_1$ for which there is no L' such that $\langle i, j, L'\rangle \in \rho_2$. If there still exists a j' and L' such that $\langle i, j', L'\rangle \in \rho_2$ for which there might be a match between L and L', then this case falls under the untimed And_Θ over $\mathcal{L}'$, and we still have some τ for which the latter returns $\langle i\oplus j, 1, \tau\rangle$; if match is never possible or no of such j' exists, then the untimed Or_Θ operator will return a $\langle i\oplus j, 1, \cup\{L | \exists k.\ \langle i\oplus j, k, L\rangle \in \rho_2\}\rangle$ by definition;

$\psi^{\tau} = \textbf{Until}^{\tau}(\rho_1, \rho_2)$**:** This is a mere rewriting exercise, as the untimed version of Until is a mere instantiation of the latter where only the case $k = 1$ is considered.

□

Appendix A.3

At this stage, we provide some rewriting lemmas motivating the introduction of derived operators. First, we want to show that the untimed $\mathsf{And}_{\Theta}(\rho, \rho')$ operator can also be exploited to compute $\mathsf{And}_{\Theta}(\mathsf{Future}(\rho), \mathsf{Future}(\rho'))$, thus motivating the peculiar definition of such operator with an existential interpretation over all of the possible matches in the future. We can formally prove this as follows:

Lemma A8.

$$\forall \rho, \rho'. \mathit{And}_{\Theta}(\mathit{Future}(\rho), \mathit{Future}(\rho')) = \mathit{And}_{\Theta}(\rho, \rho')$$

Proof. By expanding the definition of the operators, we obtain:

$$\begin{aligned}
\langle i, 1, L'' \rangle \in \mathsf{And}_{\Theta}(\mathsf{Future}(\rho), \mathsf{Future}(\rho')) \Leftrightarrow\ & \exists L, L'. \langle i, 1, L \rangle \in \mathsf{Future}(\rho) \wedge \langle i, 1, L' \rangle \in \mathsf{Future}(\rho') \\
& L'' := \mathcal{T}_{\Theta}^{E,i}([1 \mapsto \cup\{L_j |\, \langle i, j, L_j \rangle \in \rho_1\}], [1 \mapsto \cup\{L_j |\, \langle i, j, L_j \rangle \in \rho_2\}]), \\
& L'' \neq \textbf{False} \\
\Leftrightarrow\ & \exists j, j', L, L'. \langle i, j, L \rangle \in \rho \wedge \langle i, j', L' \rangle \in \rho' \\
& L'' := \mathcal{T}_{\Theta}^{E,i}([1 \mapsto \cup\{L_j |\, \langle i, j, L_j \rangle \in \rho_1\}], [1 \mapsto \cup\{L_j |\, \langle i, j, L_j \rangle \in \rho_2\}]), \\
& L'' \neq \textbf{False} \\
\Leftrightarrow\ & \langle i, 1, L'' \rangle \in \mathsf{And}_{\Theta}(\rho, \rho')
\end{aligned}$$

□

Please remember that the untimed And operator is also compliant with the $\mathrm{LTL_f}$ semantics as per our previous lemmas. We can therefore exploit the versatile definition of such operation to reduce the computational overhead provided by the additional and unrequired aggregation provided by Future. Given the previous lemma, we have as a Corollary that the semantics associated with the Choice Declare clause, i.e., $\mathsf{Or}_{\Theta}(\mathsf{Future}(\rho), \mathsf{Future}(\rho'))$, can equivalently be computed by $\mathsf{Or}_{\Theta}(\rho, \rho')$. The following proof motivates the choice of exploiting E_{Θ}^{i} as a correlation matching semantics for both And_{Θ} and Or_{Θ}.

Corollary A1.

$$\forall \rho, \rho'. \mathit{Or}_{\Theta}(\mathit{Future}(\rho), \mathit{Future}(\rho')) = \mathit{Or}_{\Theta}(\rho, \rho')$$

Proof. By expanding the definition of the untimed Or_{Θ}, we obtain:

$$\begin{aligned}
\mathsf{Or}_{\Theta}(\mathsf{Future}(\rho), \mathsf{Future}(\rho')) = \mathsf{And}_{\Theta}(\mathsf{Future}(\rho), \mathsf{Future}(\rho')) \cup \{\ \langle i, 1, \cup\{L | \exists j.\ \langle i, j, L \rangle \in \mathsf{Future}(\rho)\} \rangle \mid \nexists j, L'.\ \langle i, j, L' \rangle \in \mathsf{Future}(\rho')\ \} \\
\cup \{\ \langle i, 1, \cup\{L | \exists j.\ \langle i, j, L \rangle \in \mathsf{Future}(\rho')\} \rangle \mid \nexists j, L'.\ \langle i, j, L' \rangle \in \mathsf{Future}(\rho)\ \}
\end{aligned}$$

For the previous lemma, this becomes:

$$\begin{aligned}
\mathsf{Or}_{\Theta}(\mathsf{Future}(\rho), \mathsf{Future}(\rho')) = \mathsf{And}_{\Theta}(\rho, \rho') \cup \{\ \langle i, 1, \cup\{L | \exists j.\ \langle i, j, L \rangle \in \mathsf{Future}(\rho)\} \rangle \mid \nexists j, L'.\ \langle i, j, L' \rangle \in \mathsf{Future}(\rho')\ \} \\
\cup \{\ \langle i, 1, \cup\{L | \exists j.\ \langle i, j, L \rangle \in \mathsf{Future}(\rho')\} \rangle \mid \nexists j, L'.\ \langle i, j, L' \rangle \in \mathsf{Future}(\rho)\ \}
\end{aligned}$$

At this stage, we only need to test the contribution of the second component of the union, as the third one is symmetrical (ρ and ρ' are just inverted). As the elements of the second component of the union come from Future operators, we can rewrite such as follows:

$$\{\ \langle i, 1, \cup\{L |\, \langle i, 1, L \rangle \in \mathsf{Future}(\rho)\} \rangle \mid \nexists L'.\ \langle i, 1, L' \rangle \in \mathsf{Future}(\rho')\ \}$$

We can also observe that $\langle i, 1, L \rangle \in \mathsf{Future}(\rho)$ for a given L if there exist a j and L'' for which $\langle i, j, L'' \rangle \in \rho$. Similar considerations come from the negated counterpart ($\langle i, 1, L \rangle \notin \mathsf{Future}(\rho)$). For this expansion, we can therefore close our goal. □

The remaining lemmas show the correctness of the logical formulation of the derived operators, thus motivating their adoption when possible. These lemmas were also tested in our implementation (See the end of https://github.com/datagram-db/knobab/blob/main/tests/until_test.cpp, 5 March 2023). The supplementary materials (Section II) show that it is possible to implement such derived operators so that they are faster than their corresponding LTL_f rewriting counterpart.

Lemma A9.

$$\forall \rho, \rho'. \mathit{And}^{\tau}_{\Theta}(\rho_1, \mathit{Future}^{\tau}(\rho_2)) = \mathit{AndFuture}^{\tau}_{\Theta}(\rho_1, \rho_2)$$

Proof.

$$\begin{aligned}
\langle i, j, L\rangle \in \mathsf{And}^{\tau}_{\Theta}(\rho_1, \mathsf{Future}^{\tau}(\rho_2)) \Leftrightarrow\ & \exists L_1, L_2.\ \langle i, j, L_1\rangle \in \rho_1 \wedge \langle i, j, L_2\rangle \in \mathsf{Future}^{\tau}(\rho_2) \wedge \\
& L := \mathcal{T}^{E,i}_{\Theta}([j \mapsto L_1], [j \mapsto L_2]) \wedge L \neq \textbf{False} \\
\Leftrightarrow\ & \exists L_1, L_2.\ \langle i, j, L_1\rangle \in \rho_1 \wedge \exists h \geq j, L.\ \langle i, h, L\rangle \in \rho_2 \wedge \\
& L := \mathcal{T}^{E,i}_{\Theta}([j \mapsto L_1], [j \mapsto \cup_{\substack{j \leq k \leq |\sigma^i| \\ \langle i,k,L_k\rangle \in \rho}} L_k]) \wedge L \neq \textbf{False} \\
\Leftrightarrow\ & \langle i, j, L\rangle \in \mathsf{AndFuture}^{\tau}_{\Theta}(\rho_1, \rho_2)
\end{aligned}$$

□

Lemma A10.

$$\forall \rho, \rho'. \mathit{And}^{\tau}_{\Theta}(\rho_1, \mathit{Globally}^{\tau}(\rho_2)) = \mathit{AndGlobally}^{\tau}_{\Theta}(\rho_1, \rho_2)$$

Proof.

$$\begin{aligned}
\langle i, j, L\rangle \in \mathsf{And}^{\tau}_{\Theta}(\rho_1, \mathsf{Globally}^{\tau}(\rho_2)) \Leftrightarrow\ & \exists L_1, L_2.\ \langle i, j, L_1\rangle \in \rho_1 \wedge \langle i, j, L_2\rangle \in \mathsf{Globally}^{\tau}(\rho_2) \wedge \\
& L := \mathcal{T}^{E,i}_{\Theta}([j \mapsto L_1], [j \mapsto L_2]) \wedge L \neq \textbf{False} \\
\Leftrightarrow\ & \exists L_1.\ \langle i, j, L_1\rangle \in \rho_1 \wedge \langle i, j, L_j\rangle \in \rho_2 \wedge \\
& |\sigma^i| - j + 1 = \left|\{\langle i, k, L_k\rangle \in \rho | j \leq k \leq |\sigma^i|\}\right| \wedge \\
& L := \mathcal{T}^{E,i}_{\Theta}([j \mapsto L_1], [j \mapsto \cup_{\substack{j \leq k \leq |\sigma^i| \\ \langle i,k,L_k\rangle \in \rho}} L_k]) \wedge L \neq \textbf{False} \\
\Leftrightarrow\ & \exists L_1.\ \langle i, j, L_1\rangle \in \rho_1 \wedge \forall j \leq k \leq |\sigma^i|. \exists L'.\ \langle i, k, L_k\rangle \in \rho_2 \wedge \\
& L := \mathcal{T}^{E,i}_{\Theta}([j \mapsto L_1], [j \mapsto \cup_{\substack{j \leq k \leq |\sigma^i| \\ \langle i,k,L_k\rangle \in \rho}} L_k]) \wedge L \neq \textbf{False} \\
\Leftrightarrow\ & \langle i, j, L\rangle \in \mathsf{AndGlobally}^{\tau}_{\Theta}(\rho_1, \rho_2)
\end{aligned}$$

□

References

1. Agrawal, R.; Imieliński, T.; Swami, A. Mining Association Rules between Sets of Items in Large Databases. *SIGMOD Rec.* **1993**, *22*, 207–216. [CrossRef]
2. Bergami, G.; Maggi, F.M.; Montali, M.; Peñaloza, R. Probabilistic Trace Alignment. In Proceedings of the 2021 3rd International Conference on Process Mining (ICPM), Eindhoven, The Netherlands, 31 October–4 November 2021; pp. 9–16. [CrossRef]
3. Schön, O.; van Huijgevoort, B.; Haesaert, S.; Soudjani, S. Correct-by-Design Control of Parametric Stochastic Systems. In Proceedings of the 2022 IEEE 61st Conference on Decision and Control, Cancun, Mexico, 6–9 December 2022.
4. Appleby, S.; Bergami, G.; Morgan, G. Running Temporal Logical Queries on the Relational Model. In Proceedings of the International Database Engineered Applications Symposium (IDEAS'22), Budapest, Hungary, 22–24 August 2022; pp. 222–231.
5. Schönig, S.; Rogge-Solti, A.; Cabanillas, C.; Jablonski, S.; Mendling, J. Efficient and Customisable Declarative Process Mining with SQL. In *Advanced Information Systems Engineering, Proceedings of the 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, 13–17 June 2016*; Springer: Berlin/Heidelberg, Germany, 2016.
6. Burattin, A.; Maggi, F.M.; Sperduti, A. Conformance checking based on multi-perspective declarative process models. *Expert Syst. Appl.* **2016**, *65*, 194–211. [CrossRef]

7. Pesic, M.; Schonenberg, H.; van der Aalst, W.M.P. DECLARE: Full Support for Loosely-Structured Processes. In Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, Annapolis, MA, USA, 15–19 October 2007; pp. 287–300.
8. Musser, D.R. Introspective Sorting and Selection Algorithms. *Softw. Pract. Exp.* **1997**, *27*, 983–993. [CrossRef]
9. Bellatreche, L.; Kechar, M.; Bahloul, S.N. Bringing Common Subexpression Problem from the Dark to Light: Towards Large-Scale Workload Optimizations. In Proceedings of the 25th International Database Engineering & Applications Symposium, Montreal, QC, Canada, 14–16 July 2021.
10. Naldurg, P.; Sen, K.; Thati, P. A Temporal Logic Based Framework for Intrusion Detection. In Proceedings of the Formal Techniques for Networked and Distributed Systems—FORTE 2004: 24th IFIP WG 6.1 International Conference, Madrid, Spain, 27–30 September 2004; Núñez, M., Ed.; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3235, pp. 359–376.
11. Ray, I. Security Vulnerabilities in Smart Contracts as Specifications in Linear Temporal Logic. Master's Thesis, University of Waterloo, Waterloo, ON, Canada, 2021.
12. Buschjäger, S.; Hess, S.; Morik, K. Shrub Ensembles for Online Classification. In Proceedings of the the AAAI Conference on Artificial Intelligence 2022, Virtual, 22 February–1 March 2022; pp. 6123–6131.
13. Huo, X.; Hao, K.; Chen, L.; song Tang, X.; Wang, T.; Cai, X. A dynamic soft sensor of industrial fuzzy time series with propositional linear temporal logic. *Expert Syst. Appl.* **2022**, *201*, 117176. [CrossRef]
14. Bergami, G.; Francescomarino, C.D.; Ghidini, C.; Maggi, F.M.; Puura, J. Exploring Business Process Deviance with Sequential and Declarative Patterns. *arXiv* **2021**, arXiv:2111.12454.
15. Zhou, H.; Milani Fard, A.; Makanju, A. The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. *J. Cybersecur. Priv.* **2022**, *2*, 358–378. [CrossRef]
16. Szabo, N. Smart contracts: Building blocks for digital markets. *Extropy J. Transhumanist Thought* **1996**, *18* , 28.
17. Fionda, V.; Greco, G.; Mastratisi, M.A. Reasoning About Smart Contracts Encoded in LTL. In Proceedings of the AIxIA 2021—Advances in Artificial Intelligence: 20th International Conference of the Italian Association for Artificial Intelligence, Virtual Event, 1–3 December 2021; Springer International Publishing: Cham, Switzerland, 2021; pp. 123–136.
18. Bank, H.S.; D'souza, S.; Rasam, A. Temporal Logic (TL)-Based Autonomy for Smart Manufacturing Systems. *Procedia Manuf.* **2018**, *26*, 1221–1229. [CrossRef]
19. Mao, X.; Li, X.; Huang, Y.; Shi, J.; Zhang, Y. Programmable Logic Controllers Past Linear Temporal Logic for Monitoring Applications in Industrial Control Systems. *IEEE Trans. Ind. Informatics* **2022**, *18*, 4393–4405. [CrossRef]
20. Boniol, P.; Linardi, M.; Roncallo, F.; Palpanas, T.; Meftah, M.; Remy, E. Unsupervised and scalable subsequence anomaly detection in large data series. *Vldb J.* **2021**, *30*, 909–931. [CrossRef]
21. Xu, H.; Pang, J.; Yang, X.; Yu, J.; Li, X.; Zhao, D. Modeling clinical activities based on multi-perspective declarative process mining with openEHR's characteristic. *BMC Med. Inform. Decis. Mak.* **2020**, *20-S*, 303. [CrossRef]
22. Rovani, M.; Maggi, F.M.; de Leoni, M.; van der Aalst, W.M.P. Declarative process mining in healthcare. *Expert Syst. Appl.* **2015**, *42*, 9236–9251. [CrossRef]
23. Bertini, F.; Bergami, G.; Montesi, D.; Veronese, G.; Marchesini, G.; Pandolfi, P. Predicting Frailty Condition in Elderly Using Multidimensional Socioclinical Databases. *Proc. IEEE* **2018**, *106*, 723–737. [CrossRef]
24. De Giacomo, G.; Maggi, F.M.; Marrella, A.; Patrizi, F. On the Disruptive Effectiveness of Automated Planning for LTL*f*-Based Trace Alignment. In Proceedings of the AAAI Conference on Artificial Intelligence 2017, San Francisco, CA, USA, 4–9 February 2017.
25. Bergami, G.; Maggi, F.M.; Marrella, A.; Montali, M. Aligning Data-Aware Declarative Process Models and Event Logs. In *Business Process Management*; Springer International Publishing: Berlin/Heidelberg, Germany, 2021; pp. 235–251.
26. Bergami, G. A Logical Model for joining Property Graphs. *arXiv* **2021**, arXiv:2106.14766.
27. Zhu, S.; Pu, G.; Vardi, M.Y. First-Order vs. Second-Order Encodings for LTLf-to-Automata Translation. *arXiv* **2019**, arXiv:1901.06108.
28. Ceri, S.; Gottlob, G. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Trans. Software Eng.* **1985**, *11*, 324–345. [CrossRef]
29. Calders, T.; Lakshmanan, L.V.S.; Ng, R.T.; Paredaens, J. Expressive power of an algebra for data mining. *ACM Trans. Database Syst.* **2006**, *31*, 1169–1214. [CrossRef]
30. Li, J.; Pu, G.; Zhang, Y.; Vardi, M.Y.; Rozier, K.Y. SAT-based explicit LTLf satisfiability checking. *Artif. Intell.* **2020**, *289*, 103369. [CrossRef]
31. Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E. FoodBroker-Generating Synthetic Datasets for Graph-Based Business Analytics. In Proceedings of the 5th International Workshop, WBDB 2014, Potsdam, Germany, 5–6 August 2014.
32. Bergami, G. On Declare MAX-SAT and a finite Herbrand Base for data-aware logs. *arXiv* **2021**, arXiv:2106.07781.
33. Pichler, P.; Weber, B.; Zugal, S.; Pinggera, J.; Mendling, J.; Reijers, H.A. Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In Proceedings of the BPM 2011 International Workshops, Clermont-Ferrand, France, 29 August 2011; pp. 383–394.
34. Codd, E.F. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* **1970**, *13*, 377–387. [CrossRef]
35. Idreos, S.; Groffen, F.; Nes, N.; Manegold, S.; Mullender, K.S.; Kersten, M.L. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* **2012**, *35*, 40–45.

36. Boncz, P.A.; Manegold, S.; Kersten, M.L. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *Proc. VLDB Endow.* **2009**, *2*, 1648–1653.
37. Roth, M.A.; Korth, H.F.; Silberschatz, A. Extended Algebra and Calculus for Nested Relational Databases. *ACM Trans. Database Syst.* **1988**, *13*, 389–417. [CrossRef]
38. Wang, J.; Ntarmos, N.; Triantafillou, P. GraphCache: A Caching System for Graph Queries. In Proceedings of the International Conference on Extending Database Technology (EDBT) 2017, Venice, Italy, 21–24 March 2017; pp. 13–24.
39. Keller, A.M.; Basu, J. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB J.* **1996**, *5*, 35–47. [CrossRef]
40. Davey, B.A.; Priestley, H.A. *Introduction to Lattices and Order*, 2nd ed. ; Cambridge University Press: Cambridge, UK, 2002.
41. de Berg, M.; Cheong, O.; van Kreveld, M.J.; Overmars, M.H. *Computational Geometry: Algorithms and Applications*, 3rd ed.; Springer: Berlin/Heidelberg, Germany, 2008.
42. Elmasri, R.; Navathe, S.B. *Fundamentals of Database Systems*, 7th ed.; Pearson: Upper Saddle River, NJ, USA, 2015.
43. Polyvyanyy, A.; ter Hofstede, A.H.M.; Rosa, M.L.; Ouyang, C.; Pika, A. Process Query Language: Design, Implementation, and Evaluation. *arXiv* **2019**, arXiv:1909.09543.
44. Coffman, E.G.; Graham, R.L. Optimal Scheduling for Two-Processor Systems. *Acta Inform.* **1972**, *1*, 200–213. [CrossRef]
45. Sugiyama, K.; Tagawa, S.; Toda, M. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Trans. Syst. Man. Cybern.* **1981**, *11*, 109–125. [CrossRef]
46. Bergami, G. On Efficiently Equi-Joining Graphs. In Proceedings of the 25th International Database Engineering & Applications Symposium 2021, Montreal, QC, Canada, 14–16 July 2021.
47. Dittrich, J. *Patterns in Data Management: A Flipped Textbook*; CreateSpace Independent Publishing Platform: Charleston, SC, USA, 2016.
48. Schönig, S. SQL Queries for Declarative Process Mining on Event Logs of Relational Databases. *arXiv* **2015**, arXiv:1512.00196.
49. Shoshany, B. A C++17 Thread Pool for High-Performance Scientific Computing. *arXiv* **2021**, arXiv:2105.00613.
50. Klemm, M.; Cownie, J. 8 Scheduling parallel loops. In *High Performance Parallel Runtimes*; De Gruyter Oldenbourg: Berlin, Germany; Boston, MA, USA, 2021; pp. 228–258.
51. Ristov, S.; Prodan, R.; Gusev, M.; Skala, K. Superlinear speedup in HPC systems: Why and when? In Proceedings of the 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), Gdańsk, Poland, 11–14 September 2016; pp. 889–898.
52. Yan, B.; Regueiro, R.A. Superlinear speedup phenomenon in parallel 3D Discrete Element Method (DEM) simulations of complex-shaped particles. *Parallel Comput.* **2018**, *75*, 61–87. [CrossRef]
53. Nagashima, U.; Hyugaji, S.; Sekiguchi, S.; Sato, M.; Hosoya, H. An experience with super-linear speedup achieved by parallel computing on a workstation cluster: Parallel calculation of density of states of large scale cyclic polyacenes. *Parallel Comput.* **1995**, *21*, 1491–1504. [CrossRef]
54. Anselma, L.; Bottrighi, A.; Montani, S.; Terenziani, P. Extending BCDM to Cope with Proposals and Evaluations of Updates. *IEEE Trans. Knowl. Data Eng.* **2013**, *25*, 556–570. [CrossRef]
55. Bergami, G.; Bertini, F.; Montesi, D. Hierarchical embedding for DAG reachability queries. In Proceedings of the IDEAS 2020: 24th International Database Engineering & Applications Symposium, Seoul, Republic of Korea, 12–14 August 2020; Desai, B.C., Cho, W., Eds.; ACM: New York, NY, USA, 2020; pp. 24:1–24:10.
56. Revesz, P.Z. *Introduction to Databases—From Biological to Spatio-Temporal*; Texts in Computer Science; Springer: Berlin/Heidelberg, Germany, 2010.
57. Revesz, P. Geographic Databases. In *Introduction to Databases: From Biological to Spatio-Temporal*; Springer: London, UK, 2010; pp. 81–109.
58. Zaki, N.M.; Helal, I.M.A.; Awad, A.; Hassanein, E.E. Efficient Checking of Timed Order Compliance Rules over Graph-encoded Event Logs. *arXiv* **2022**, arXiv:2206.09336.
59. Rost, C.; Gómez, K.; Täschner, M.; Fritzsche, P.; Schons, L.; Christ, L.; Adameit, T.; Junghanns, M.; Rahm, E. Distributed temporal graph analytics with GRADOOP. *VLDB J.* **2022**, *31*, 375–401. [CrossRef]