Supplementary Material

Part

Table of Contents

Supplement I Matching Requirements.

Depending on the binary $\texttt{xt}\text{LTL}_\text{f}$ operator of choice, we might set a successful match if at least two events in a trace, one per operator, match anytime in the future (E^i_Θ), or if correlation conditions hold across activated and targeted events within the temporal window of interest (e.g., *Until*; A^i_Θ). Given some maps M_1 and M_2 associating events of interest from the same trace σ^i to a list L providing the activation and target conditions to be tested, we define such two distinct matching requirements as follows:

$$E^i_\Theta(M_1, M_2) = \Big\{ M[h,k] \;\Big|\; \begin{aligned} &\exists j \in \text{dom}(M_1) \cap \text{dom}(M_2), L_1 \in M_1(j), L_2 \in M_2(j). \\ &(A(h) \in L_1 \vee \exists h'.M[h,h'] \in L_1), \\ &(T(k) \in L_2 \vee \exists k'.M[k,k'] \in L_2),\ \Theta(\sigma^i_h, \sigma^i_k) \Big\} \end{aligned} \tag{S1}$$

$$A^i_\Theta(M_1, M_2) = \begin{cases} \emptyset & \exists j \in \text{dom}(M_1) \cap \text{dom}(M_2).E^i_\Theta([j \mapsto M_1(j)], [j \mapsto M_2(j)]) = \emptyset \\ E^i_\Theta(M_1, M_2) & \text{oth.} \end{cases} \tag{S2}$$

where i indicates the trace id (e.g., of σ^i) from which the events in L should be considered while reconstructing their associated payload while determining the binary predicate Θ.

After expressing the different matching semantics as such, we skip testing correlations when (e.g.) only one of the two operands will contain activation (or target) events. We distinguish the case where the activation and target conditions of interest are missing because some basic operators did not return those from the scenario where the correlation condition was tested but failed. While in the former we return either an empty set or a set of just activations or targets, in the latter we want to return a placeholder **False** to remember that the correlation condition was falsified. We can then define a *testing functor* as follows:

$$\mathcal{T}^{F,i}_\Theta(M_1, M_2) = \begin{cases} \text{cod}(M_1) \cup \text{cod}(M_2) & \text{dom}(M_1) = \emptyset \vee \text{dom}(M_2) = \emptyset \\ F^i_\Theta(M_1, M_2) & F^i_\Theta(M_1, M_2) \neq \emptyset \\ \textbf{False} & \text{oth.} \end{cases} \tag{S3}$$

where F subsumes one of the two matching semantics of choice, either E^i_Θ or A^i_Θ.

Supplement II Time Complexity

Supplement II.1 Pipeline Components

We now discuss the first two procedures from Algorithm 2.

Lemma S1. BULKINSERTION *can be computed in quasi-linear time with respect to the full log size (thus including the trace size, as well as the size of their associated payload).*

Proof. As most of the data structures are vectors (Lines 7 and 8) and the only two maps are a hashmap (Line 6) and a hashmap of ordered maps (Line 11), this phase has a quasi-linear time computational complexity with respect to the size of the log, thus showing that no additional overhead for pre-calculating the semantics associated to the declarative clauses is required in [60]. In fact, bulk inserting the data requires scanning all the traces and their associated events, as well as its possible associated payload, while the insertion in the $\texttt{AttBulkMap}_k$ performs a sorted insertion for each value $p(k)$ (Line 11); this pays a log-linear cost over the number of distinct values per key k. □

Lemma S2. LOADINGANDINDEXING *has linear time complexity with respect to the log size (trace and data payload inclusive).*

Proof. We show that the first outer for loop has a time complexity in $O(|\mathcal{L}|\epsilon)$. The initialization of the primary index of $\mathsf{ActivityTable}_{\mathcal{L}}$ requires a linear scan of every unique activity label of the log, with index assignments being performed in constant time (Line 18). The loading of the aforementioned table (minus assignments of `Prev`(`Next`)) requires a scan of every element of the log, thus providing a linear complexity with respect to the log size (inclusive of trace lengths, Line 23). The loading of the $\mathsf{CountingTable}_{\mathcal{L}}$ has complexity $O(|\Sigma||\mathcal{L}|)$ (Line 20), which is always dominated by loading of the $\mathsf{ActivityTable}_{\mathcal{L}}$, as it is guranteed that $|\Sigma| \leq |\mathcal{L}|\epsilon$. In fact, in the worst-case scenario, every single event has a unique activity label, and therefore $|\Sigma| = |\mathcal{L}|\epsilon = |\texttt{ActToEventBulkVector}|$. In the best-case scenario, there is only one activity label ($|\Sigma| = 1$) providing a linear complexity with log size.

In the best case scenario, no trace comes with a data payload, and therefore the entire outermost second loop is skipped as the set of keys is completely empty, and therefore no $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ will be initialised. Otherwise, loading an $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ associated to a key $\kappa \in K$ requires a linear scan over each activity label a from Σ. While doing so, we cals access the `AttBulkMap` for every unique activity label and payload key, from which we also scan all the events from *lst* associated to a same value ν. The initialisation of each $\mathsf{AttributeTable}^{k}_{\mathcal{L}}$ primary index is linear with respect to the number of unique activity labels (Line 35); the loading and secondary indexing of $\mathsf{AttributeTable}^{\kappa}_{\mathcal{L}}$ is bounded by the number of events associating a value ν to each key κ of interest. In the worst-case scenario, every event contains the same key labels. Overall, this provides a complexity dominated by the number of key-value associations in each event and therefore by $|K|$, as each event can contain an unlimited number of payload keys. This boils down to a worst-case time complexity of $O(|\mathcal{L}|\epsilon|K|)$.

We show that the last outer for loop also has a time complexity in $O(|\mathcal{L}|\epsilon)$. This includes the assignment of the `Prev` (`Next`) pointers and the initialisation of the $\mathsf{ActivityTable}_{\mathcal{L}}$'s secondary index , requiring only a linear scan of the log (trace inclusive). No other data structures are iterated, therefore we always perform computations in $O(|\mathcal{L}|\epsilon)$ time.

Overall, in the best case scenario no event is associated to a payload, and therefore the time complexity is dominated by the first two loops, which provide a time complexity linear to the log size, trace size included: $O(|\mathcal{L}|\epsilon)$. Otherwise, we have to scan each value associated to each event which, in the worst case scenario, requires a full scan of $|K|$ for each event. In this other scenario, the time complexity is heavily dominated by the data payload and therefore is $O(|\mathcal{L}|\epsilon|K|)$. □

Last, we discuss the time complexity for the query processing algorithms.

Lemma S3. AtomisationPipeline *(Algorithm 3) has polynomial time complexity with respect to the model, key-set, and elementary intervals' maximum size.*

Proof. In the best-case scenario, all the clauses have no payload conditions, for which the activation condition will only contain the activation (or target) activity label. For this, we only pay the linear cost of scanning the model, $O(|\mathcal{M}|)$. In the worst-case scenario, each clause has a non-trivial payload condition for activation and target (when available): this implies always computing an intersection over unordered sets, which has the computational cost of scanning the smaller set and checking which elements are contained in the larger, where each operation takes $O(1)$ time. We can assume that each p is described in disjunctive normal form and that its size is negligible if compared to the size of both the collected elementary intervals and the model. Given that usually $ak(\mathsf{B})$ contains all of the atoms referring to the data predicates to B and therefore $\mathsf{Atom}_{\mu,ad}(\mathsf{B},\kappa) \subseteq ak(\mathsf{B})$ for any $\kappa \in K$, we have a computational complexity of $O(m|K||\mathcal{M}|)$ under the assumption that each clause's data payload condition contains predicates for any payload key in $O(|K|)$ and that each interval has a size comparable to the maximum size of $m = \mu(\mathsf{a},\kappa)$ for $\mathsf{a} \in \Sigma$ and $\kappa \in K$. □

Lemma S4. QueryScheduler *(Algorithm 5) has a linear time complexity with respect to the number of operators appearing in the compiled query plan.*

Proof. If we exclude the root node, the branching factor of our graph is at most 2, as each operator (represented as a node in the graph) might contain at most two sub-expressions; therefore, we can estimate that the number of edges of $\mathcal{G}$ is linear with respect to the graph nodes. As the scheduler is represented through a vector of vectors and the distance computation requires visiting each edge associated with each non-leaf node, while filling the scheduler layer-wise just requires visiting each operator, the time complexity of this step is linear with respect to the operators appearing in the formula. □

Supplement II.2 Operators' algorithmic implementation

We now discuss the computational complexity of the $\mathtt{xtLTL_f}$ operators. Let us assume that ℓ is the size of the collected events in L that need to be matched. Let us also remember that the intermediate results ρ and ρ' are represented as vectors of triplets sorted by trace and event id. Let us also assume all traces have a maximum length of ϵ. In the worst case scenario, we have $|\rho| = |\rho'| = |\mathcal{L}|\epsilon$.

Lemma S5. *Future*(ρ) *has linear time complexity with respect to the operand size while* *Future*$^{\tau}(\rho)$ *has quadratic time complexity with respect to the maximum trace length size.*

Proof. The computational complexity associated with the untimed Future operator boils down to performing an aggregation for each trace represented in the operand as well as its represented event, where its collected activation/target conditions are all associated with the first event of the trace. Therefore, this has a linear time complexity with respect to the size of each operand, i.e. $|\rho| \in O(|\mathcal{L}|\epsilon)$.

On the other hand, its timed counterpart (Future$^{\tau}$) needs both to list all of the events in the operand and, for each of them, to associate to it also the activation and target conditions happening in the future, thus requiring a quadratic cost over the size of each trace. This boils down to $O(|\mathcal{L}|\epsilon^2)$. □

Lemma S6. TimedIntersection *has computational complexity in* $O(\ell^2|\mathcal{L}|\epsilon)$.

Proof. The time complexity of the TimedIntersection is in the worst-case scenario in $O(\ell^2|\mathcal{L}|\epsilon)$ if we assume that each matched event needs to be tested and that ρ and ρ' have non-empty Ls. If either Θ is always true or the Ls of the two operands are empty, the time complexity boils down to $O(|\mathcal{L}|\epsilon)$. □

Corollary S1. *For $\Theta = \textbf{\textit{True}}$,* FASTUNTIMEDAND$(\rho, \rho')$ *is faster than* SLOWUNTIMEDAND(ρ, ρ'). *For $\ell \approx 1$, the theoretical speed-up is proportional to the increase of both trace length and number of traces.*

Proof. The time complexity of the SLOWUNTIMEDAND is in the worst case scenario in $O(\ell^2|\mathcal{L}|\epsilon^2)$: as the former is a minor adaptation of TIMEDINTERSECTION where, nevertheless, we have to pay a quadratic cost for scanning the whole traces as in timed future (Future$^\tau$). This has also an additional cost of $O(|\mathcal{L}| \log |\mathcal{L}|\epsilon)$ gained while grouping the operands' results by trace id.

On the other hand, its FASTUNTIMEDAND avoids a quadratic cost if $\Theta = \textbf{True}$ by only collecting the activation and target conditions through a linear scan of both operands, thus boiling down its computational complexity to $O(2|\mathcal{L}|\epsilon\ell)$.

At this point, we can compute the speed-up of the allegedly slower version over the faster one by checking when the following condition holds:

$$\frac{\ell^2|\mathcal{L}|\epsilon^2 + 2|\mathcal{L}| \log |\mathcal{L}|\epsilon}{2|\mathcal{L}|\epsilon\ell} = \frac{\ell\epsilon}{2} + \frac{\log |\mathcal{L}|}{\ell} > 1$$

We can close the goal after observing that the aforementioned condition always holds, and that is heavily dominated by the average trace length. □

Corollary S2. *Computing* FASTUNTIMEDAND(ρ, ρ') *is more efficient than its equivalent* FASTUNTIMEDAND$(\textit{Future}(\rho), \textit{Future}(\rho'))$ *by a theoretical constant speed up.*

Proof. By expanding and composing the definitions of the former computational complexities, we get that the speed-up is almost constant for small ℓ values:

$$\frac{2|\mathcal{L}|\epsilon + 2\mathcal{L}\epsilon\ell}{2\mathcal{L}\epsilon\ell} = \frac{1}{\ell} + 1 > 1$$

□

Corollary S3. *Computing* FASTUNTIMEDOR(ρ, ρ') *is more efficient than Choice as per its original Declare semantics, i.e.* FASTUNTIMEDOR$(\textit{Future}(\rho), \textit{Future}(\rho'))$ *by a theoretical constant speed up.*

Proof. As when no traces match we have no additional matching costs as they are just collected in the final results with a linear scan, the worst-case scenario boils down to performing trace matches as per the previous corollary. □

Lemma S7. *For $\Theta = \textbf{\textit{True}}$,* UNTIMEDUNTIL$^2(\rho, \rho')$ *is faster than* UNTIMEDUNTIL$^1(\rho, \rho')$ *for greater log sizes.*

Proof. In both implementations of the untimed Until operator, for each log trace, we perform a logarithmic scan for reaching the end of each trace within the operand. Trace by trace, the computation time decreases with the number of the operand's traces being visited. This can be expressed as $\sum_{i=0}^{|\mathcal{L}|-1} \log(|\mathcal{L} - i|\epsilon) = \sum_{i=1}^{|\mathcal{L}|} \log(i\epsilon)$. Given $\sum_{i=0}^{n} \log i = \log(i!)$, we can rewrite the computational cost of such operation as follows:

$$\sum_{i=1}^{|\mathcal{L}|} \log(i\epsilon) = \sum_{i=1}^{|\mathcal{L}|} (\log i + \log \epsilon) = \log(|\mathcal{L}|!) + |\mathcal{L}| \log \epsilon \leq |\mathcal{L}|(\log |\mathcal{L}|\epsilon)$$

In its worst-case scenario, UNTIMEDUNTIL^1 always run the same scan over each trace's targeted events for each activated event in the second operand, while its Fast- counterpart performs constant access for each of the ϵ trace events. So, while UNTIMEDUNTIL^2 has an overall time complexity of $\epsilon|\mathcal{L}|$, the former has the upper bound of:

$$= \sum_{i=1}^{|\mathcal{L}|}(\log i + \sum_{j=1}^{\epsilon} \log j) \leq |\mathcal{L}| \log |\mathcal{L}| + |\mathcal{L}|\epsilon \log \epsilon = |\mathcal{L}| \log(|\mathcal{L}|\epsilon^{\epsilon})$$

Furthermore, for both implementations we pay at most a quadratic cost for scanning each targeted event that needs to hold until the activation is met, thus including the cost of inserting the matched conditions in the intermediate result set. This implies a cost of $O(\epsilon^2|\mathcal{L}|)$ despite $\Theta =$ **True**. As for the speed-up we can neglect the sub-operations having the same computational complexity[22], we can rewrite the asymptotic speed-up as:

$$\frac{|\mathcal{L}| \log(|\mathcal{L}|\epsilon^{\epsilon})}{\epsilon|\mathcal{L}|} = \frac{\log |\mathcal{L}|}{\epsilon} + \log \epsilon > 1$$

which is always true only for adequately large traces, $\varepsilon \gg 0$. We can see that if ϵ is fixed, the speed-up grows proportionally with the log size. As the equation is always true for non-empty datasets, the fast implementation is always faster than the slower counterpart. □

Corollary S4. *For $\Theta =$ **True**, the second variant of the (timed)* AndGlobally *(Algorithm S2) is theoretically more performant than the first one (Algorithm S1) when the number of traces is exponentially upper bounded by the maximum trace length size.*

Proof. In the second variant of the operator, for each trace, we always perform a logarithmic scan for both operators for determining the end of the trace. As seen in the previous lemma, this corresponds to $O(|\mathcal{L}| \log(|\mathcal{L}|\epsilon^{\epsilon}))$. While scanning the trace events backwards we have that, in the worst-case scenario, each event on the left operand corresponds an event on the right one. This operation, as well as the gradual backward creation of the result, boils down to an additional time complexity of $2\epsilon|\mathcal{L}|$.

In the first algorithm for the operator (referred to as "variant"), we scan all of the events per single trace, this reducing to $2|\mathcal{L}|\epsilon$, but, for each event in the left operand, we have to scan all of the events in the right one until the end of the trace, thus adding up to a quadratic computational complexity with respect to the trace size: $\epsilon|\mathcal{L}|$.

The asymptotic and theoretical speed-up can be then computed as follows:

$$\frac{\epsilon^2|\mathcal{L}|}{2|\mathcal{L}| \log(|\mathcal{L}|\epsilon^{\epsilon})} = \frac{\epsilon^2}{2(\log |\mathcal{L}| + \epsilon \log \epsilon)} > 1 \Leftrightarrow \frac{\epsilon^2}{2} - \epsilon \log \epsilon > \log |\mathcal{L}|$$
$$\Leftrightarrow |\mathcal{L}| < 2^{\epsilon^2/2 - \epsilon \log \epsilon} \leq \sqrt{2\epsilon^2}$$

We can therefore conclude that the first variant will be faster when traces are shorter, while the other one will be always faster for "reasonably" long traces. We can also observe that, when the log size is fixed, the speed-up grows most quadratically with respect to the maximum trace length. □

Supplement III Maximal Common Subset Problem

In order to minimise union and intersection operations over a sequence of atoms, we had to solve multiple times the Maximal Common Subset Problem[23]. This problem, different from the maximal common subsequence for sequences (e.g., strings) [61], can be stated as follows:

[22] $\frac{A+B}{A+C} > 1 \leftrightarrow B > C \leftrightarrow \frac{B}{C} > 1$ when $A + B > 0$, $A + C > 0$, and $A > 0$ are always true.

[23] This problem was coded as the `partition_sets` function in https://github.com/datagram-db/knobab/blob/main/include/yaucl/structures/set_operations.h.

Algorithm S1 $\texttt{xtLTL}_\text{f}$ pseudocode implementation for derived operators, First Variant

```
1: function TimedAndFuture$^1_\Theta(\rho, \rho')$
2:     $it \leftarrow$Iterator$(\rho), it' \leftarrow$Iterator$(\rho')$
3:     while $it \neq\uparrow$ and $it' \neq\uparrow$ do
4:         $\langle t, e, L\rangle \leftarrow \texttt{current}(it), \langle t', e'', \lambda\rangle \leftarrow \texttt{current}(it')$
5:         if $t = t'$ and $e \geq e''$ then
6:             $L'' \leftarrow \emptyset$; $hasMatch \leftarrow \Theta =$ True
7:             $it'_* \leftarrow it'$
8:             while $it'_* \neq\uparrow$ do
9:                 $\langle t', e', L'\rangle \leftarrow \texttt{current}(it'_*)$;
10:                if $t' \neq t$ then break;
11:                $tmp \leftarrow \mathcal{T}^{E,i}_\Theta(L, L')$                    ▷ Algorithm 7
12:                if $tmp \neq$ False then
13:                    $hasMatch \leftarrow$ true; $L'' \leftarrow L'' \cup tmp$
14:                end if
15:                $\texttt{next}(it'_*)$
16:            end while
17:            if $hasMatch$ then yield $\langle t, e, L''\rangle$;
18:            $\texttt{next}(it)$; $\texttt{next}(it')$;
19:        else if $t < t'$ or ($t = t'$ and $e < e'$) then
20:            $\texttt{next}(it)$
21:        else
22:            $\texttt{next}(it')$
23:        end if
24:    end while

25: function TimedAndGlobally$^1_\Theta(\rho, \rho')$
26:    $it \leftarrow$Iterator$(\rho), it' \leftarrow$Iterator$(\rho')$
27:    while $it \neq\uparrow$ and $it' \neq\uparrow$ do
28:        $\langle t, e, L\rangle \leftarrow \texttt{current}(it), \langle t', e'', \lambda\rangle \leftarrow \texttt{current}(it')$
29:        if $t = t'$ and $e = e''$ then
30:            $L'' \leftarrow \emptyset$; $hasMatch \leftarrow \Theta =$ True; $count \leftarrow 0$
31:            $it'_* \leftarrow it'$
32:            while $it'_* \neq\uparrow$ do
33:                $\langle t', e', L'\rangle \leftarrow \texttt{current}(it'_*)$
34:                if $t' \neq t$ then break;
35:                $tmp \leftarrow \mathcal{T}^{E,i}_\Theta(L, L')$                    ▷ Algorithm 7
36:                if $tmp \neq$ False then
37:                    $hasMatch \leftarrow$ true; $L'' \leftarrow L'' \cup tmp$; $count \leftarrow count+1$
38:                end if
39:                $\texttt{next}(it'_*)$
40:            end while
41:            if $hasMatch$ and $count = |\sigma^t| - e + 1$ then yield $\langle t, e, L''\rangle$;
42:            $\texttt{next}(it)$; $\texttt{next}(it')$;
43:        else if $t < t'$ or ($t = t'$ and $e < e'$) then
44:            $\texttt{next}(it)$
45:        else
46:            $\texttt{next}(it')$
47:        end if
48:    end while
```

Problem S1. *Given a set of sets $\mathcal{S} = \{ S_1, \ldots, S_n \}$, we want to compute the maximal common subsets $\Delta = \{ \delta_1, \ldots, \delta_k \}$ so that each set $S_\iota \in \mathcal{S}$ can be defined as the union of some pairwise disjoint sets $\delta_{\iota_1}, \ldots, \delta_{\iota_m}$ in Δ. This decomposition has to guarantee that there exists no other decomposition containing non-overlapping supersets of the sets given in Δ providing such decomposition.*

Please observe that, if all of the sets in $\mathcal{S}$ are pairwise disjoint, then $\mathcal{S} = \Delta$.

Example S1. *Given $\mathcal{S} = \{ \{1,2,3\}, \{2,3\}, \{4,5\}, \{1,2,3,4,5\}, \{2,3,4,5\} \}$, we obtain the maximal common subsets $\Delta = \{ \{2,3\}, \{1\}, \{4,5\} \}$. Given this, we might characterize the sets in $\mathcal{S}$ from the constituents in Δ, and therefore* decomposition *returned by the* MaximalCommonSubset *function allows the characterization of $\mathcal{S}$ as $\{ \delta_2 \cup \delta_1, \delta_1, \delta_3, \delta_2 \cup \delta_1 \cup \delta_3, \delta_1 \cup \delta_3 \}$ via the returned decomposition $\mathcal{I}$.*

Algorithm S3 sketches a solution to the maximal common subsets problem: by remembering the sets S_i in which each set item is contained (Line 4) and by subsequently remembering the items that are shared among the sets in $\mathcal{S}$ (Line 7), the latter map identifies

Algorithm S2 $\mathtt{xtLTL_f}$ pseudocode implementation for derived operators, Second Variant

```
1: function TimedAndFuture²_Θ(ρ, ρ′)
2:     if ρ′ = ∅ then return ∅                                              ▷ Yielding no records
3:     it ←Iterator(ρ), it′ ←Iterator(ρ′)
4:     while it ≠↑ do
5:         toBeReversed← ∅; ⟨i_c, j_c, L⟩ ← current(it); ⟨i′, j′, L′⟩ ← current(it′)
6:         if i_c > i′ or (i = i′ and j_c > j′) then
7:             it′←LowerBound(ρ, it′, ↑, ⟨i_c, j_c, ⊤_Ω⟩); if it′ =↑ break
8:         else if i < i′ then
9:             it←LowerBound(ρ, it, ↑, ⟨i′, 1, ⊤_Ω⟩);
10:        else
11:            if it′ =↑ break; tmp← ∅
12:            bend← UpperBound(ρ′, it′, ↑, ⟨i′, |σ^{i′}| + 1, ⊤_Ω⟩); aend← UpperBound(ρ, it, ↑, ⟨i_c, |σ^{i_c}| + 1, ⊤_Ω⟩)
13:            if current(it′ − 1).i ≠ i then
14:                it←aend;
15:                while(it ≠↑) and current(it).i = i do it++
16:                continue
17:            end if
18:            it←--bend; it′←--aend; ⟨i, j, L⟩ ← current(it); ⟨i′, j′, L′⟩ ← current(it′)
19:            for J ← j′ downto 1 by 1 do
20:                if bend≥it′ and J = j′ then
21:                    tmp← tmp ∪ L′; it′--; ⟨i′, j′, L′⟩ ← current(it′)
22:                end if
23:                while i = i′ and j > j′ do
24:                    aend--; ⟨i, j, L⟩ ← current(it).
25:                end while
26:                if i ≠ i′ then break
27:                if j < j_c or j < J then continue
28:                tmp ← T^{E,i}_Θ(L, L′)                                   ▷ Algorithm 7
29:                if tmp ≠ False then toBeReversed.add(⟨i, j, tmp⟩)
30:            end for
31:            forall ⟨i, j, L″⟩ ∈ Reversed(toBeReversed) do yield ⟨i, j, L″⟩
32:            it← aend; if it′ =↑ then break;
33:        end if
34:    end while

35: function TimedAndGlobally²_Θ(ρ, ρ′)
36:    if ρ′ = ∅ then return ∅                                              ▷ Yielding no records
37:    it ←Iterator(ρ), it′ ←Iterator(ρ′)
38:    while it ≠↑ do
39:        toBeReversed← ∅; ⟨i, j, L⟩ ← current(it); ⟨i′, j′, L′⟩ ← current(it′)
40:        if i > i′ or (i = i′ and j > j′) then
41:            it++; if it′ =↑ break
42:        else
43:            if it′ =↑ break; tmp← ∅
44:            bend← UpperBound(ρ′, it′, ↑, ⟨i′, |σ^{i′}| + 1, ⊤_Ω⟩); aend← UpperBound(ρ, it, ↑, ⟨i, |σ^{i}| + 1, ⊤_Ω⟩)
45:            it←--bend; it′←--aend; ⟨i, j, L⟩ ← current(it); ⟨i′, j′, L′⟩ ← current(it′)
46:            J ← j′
47:            for J ← j′ downto 1 by 1 do
48:                if bend≥it′ and Distance(it′,bend)= |σ^{i′}| − J + 1 then
49:                    tmp← tmp ∪ L′; it′--; ⟨i′, j′, L′⟩ ← current(it′)
50:                else break
51:                end if
52:                while i = i′ and j > j′ do
53:                    it--; ⟨i, j, L⟩ ← current(it).
54:                end while
55:                if i ≠ i′ then break
56:                if j < J then continue
57:                tmp ← T^{E,i}_Θ(L, L′)                                   ▷ Algorithm 7
58:                if tmp ≠ False then toBeReversed.add(⟨i, j, tmp⟩)
59:            end for
60:            forall ⟨i, j, L″⟩ ∈ Reversed(toBeReversed) do yield ⟨i, j, L″⟩
61:            it←aend; if it′ =↑ then break
62:        end if
63:    end while
```

the desired maximal common subsets. Therefore, each set S_i can be defined through the union of the maximal common subsets $\delta_k \in \Delta$ (Line 10).

The primary aim of this algorithm in KnoBAB is to minimize the computations of Or or And operators where the computation of further intermediate subsets is appropriate. As we might indeed see from the former example, the reconstruction of the original sets in $\mathcal{S}$ from their constituents in Δ implies evaluating the same unions multiple times, e.g. $\delta_2 \cup \delta_1$ and $\delta_1 \cup \delta_3$. This overhead can be limited as follows: first, we might sort the

Algorithm S3 Maximal Common Subsets

```
1:  function MaximalCommonSubset(𝒮)
2:      setsToSharedValues← {}; itemInSets ← {}; ℐ ← {}; k ← 1; Δ ← ∅
3:      for all S_i ∈ 𝒮 and item ∈ S_i do
4:          itemInSets[item].put(i)
5:      end for
6:      for all ⟨item, {i, j, ...}⟩ ∈itemInSets do
7:          setsToSharedValues[{i, j, ...}].put(item)                ▷ item∈ S_i ∩ S_j ∩ ...
8:      end for
9:      for all ⟨{i, j, ...}, {item_α, item_β, ...}⟩ ∈setsToSharedValues and ι ∈ {i, j, ...} do
10:         ℐ[ι].put(k)                                              ▷ δ_k ⊆ S_ι, S_ι ∈ 𝒮
11:         Δ.put({item_α, item_β, ...})                             ▷ δ_k = {item_α, item_β, ...}
12:         k ← k + 1
13:     end for
14:     return ⟨Δ, ℐ⟩

15: function MCSRefinement(Δ, ℐ)
16:     Δ′ ← ∅; k ← |Δ| + 1; elemMap ← {}
17:     Sort(ℐ, (⟨s, I⟩, ⟨s′, I′⟩) → I ⊆ I′)
18:     for all ⟨s, I⟩_i ∈ ℐ from i = 1 to |ℐ| s.t. |I| ≠ 1 do
19:         hasElem ← False;
20:         for all ⟨s′, I′⟩_j ∈ ℐ from j = |ℐ| − 1 to i + 1 s.t. |I′| ≠ 1 do
21:             if |I| = |I′| then break
22:             else if I ⊆ I′ then
23:                 if not hasElem then
24:                     hasElem ← True
25:                     Δ′.put(I)
26:                 end if
27:                 I′ ← I′\I ∪ {k}
28:             end if
29:         end for
30:         if hasElem then
31:             elemMap[i] ← k
32:             k ← k + 1
33:         end if
34:     end for
35:     return ⟨Δ, Δ′, ℐ⟩
36:     for all ⟨i, k⟩ ∈ elemMap do
37:         ℐ[i] ← {k}
38:     end for
```

decomposed itemsets by their inclusion relationship (Line 17): this is possible as the set of all possible subsets is a partially ordered set (poset) where the partial order is the subset-equal relationship [40]. Last, we iterate over all the possible decomposed itemsets I which do not appear as a singleton, as their replacement is trivial (Line 18): if the decomposition I' for a set S_j contains the decomposition I for a set S_i, we can elect I as a new refined subset δ_k (Line 25) which is now a component for I' (Line 27).

Example S2. *Let us continue the former example: the refinement step computes a new refinement $\delta_4 \in \Delta'$ which is defined as $\delta_1 \cup \delta_2$; the characterization of $\mathcal{S}$ has now become $\{\delta_4, \delta_1, \delta_3, \delta_4 \cup \delta_3, \delta_1 \cup \delta_3\}$.*

Example S3. *Let us now suppose to have an array of sets containing $\mathtt{xt}\,\mathrm{LTL}_f$ formulæ $\mathcal{S} = \{\{\psi_1, \psi_2, \psi_3\}, \{\psi_2, \psi_3\}, \{\psi_4, \psi_5\}, \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}, \{\psi_2, \psi_3, \psi_4, \psi_5\}\}$, where we want to express each set in $\mathcal{S}$ as a timed disjunction Or^{τ}_{True} formulæ. While doing so, we want also to minimise as possible the computation of unions that are shared among different sets. Figure S1 shows the desired result, where Δ and Δ' contain the intermediate untimed unions or just one single $\mathtt{xt}\,\mathrm{LTL}_f$ formula, while S will contain the nodes that will perform the final union operations.* FinitarySetOperations($\mathcal{S}$, Or_{True}) *returns the desired $\mathtt{xt}\,\mathrm{LTL}_f$ DAG in Figure S1, rooted in each distinct formula associated to a set in $\mathcal{S}$.*

Time Complexity. For MaximalCommonSubset, if we assume that each subset is at most of size m, then the time complexity of the first loop is in $O(|\mathcal{S}|m)$; the second loop iterates over the whole possible items contained in each set in $\mathcal{S}$, and therefore has a time complexity of $O(|\bigcup \mathcal{S}|)$; the last iteration performs a number of iterations which is

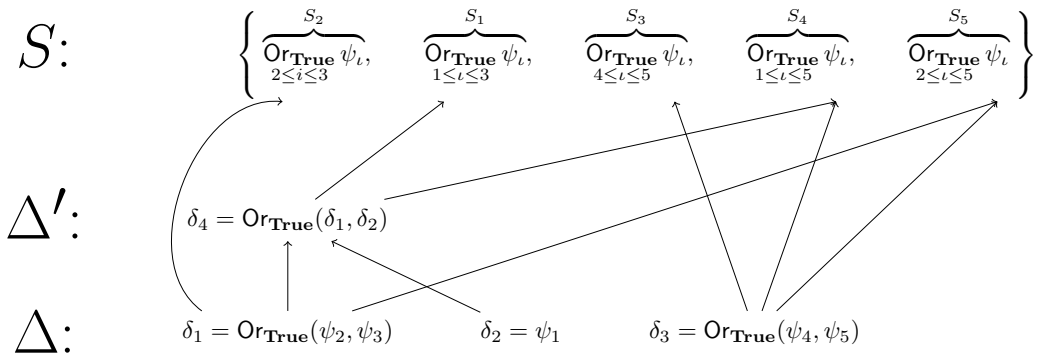

Figure S1. Decomposing untimed unions of $\mathtt{xt}\mathrm{LTL}_\mathrm{f}$ formulæ into maximal shared sub-expressions via an abstract syntax DAG, thus attempting at computing each untimed union shared among different expressions at most once.

Algorithm S4 Finitary set operations for commutative and associative set operations $\oplus$

function FINITARYSETOPERATIONS($\mathcal{S}, \bigoplus$) ▷ $\oplus \in \{\bigcup, \bigcap\}$
  $\Delta, \mathcal{I} \leftarrow$ MAXIMALCOMMONSUBSETS($\mathcal{S}$)
  $\Delta, \Delta', \mathcal{I} \leftarrow$ MCSREFINEMENT($\Delta, \mathcal{I}$)
  for all $s' \in \Delta'$ **(parallel) do**
    $\delta_{s'} \leftarrow \bigoplus_{\delta_\iota \in \Delta} \delta_\iota$
  end for
  for all $\langle i, D\rangle \in \mathcal{I}$ **do** ▷ Decomposition D of S_i through intermediate sets δ_s with $s \in D$
    yield $\langle i, \bigoplus_{s\in D} \delta_s\rangle$ ▷ Result associated to $S_i \in S$
  end for

function FINITARYSETOPERATIONS($\mathcal{S}$, *Results*, $\bigoplus$) ▷ $\oplus \in \{\bigcup, \bigcap\}$
  $\Delta, \mathcal{I} \leftarrow$ MAXIMALCOMMONSUBSETS($\mathcal{S}$)
  $\Delta, \Delta', \mathcal{I} \leftarrow$ MCSREFINEMENT($\Delta, \mathcal{I}$)
  for all $s \in \Delta$ **(parallel) do**
    $\delta_s \leftarrow \bigoplus_{\iota\in s}$ *Results*$[\iota]$
  end for
  for all $s' \in \Delta'$ **(parallel) do**
    $\delta_{s'} \leftarrow \bigoplus_{\iota\in s'} \delta_\iota$
  end for
  for all $\langle i, D\rangle \in \mathcal{I}$ **do** ▷ Decomposition D of S_i through intermediate sets $\delta_s \in \Delta \cup \Delta'$
    yield $\langle i, \bigoplus_{s\in D} \delta_s\rangle$
  end for

comparable to the size of Δ which, in the worst-case scenario, is composed of singletons of size $k = 1$ comprising elements from $\bigcup \mathcal{S}$, and therefore has a time complexity in $O(|\bigcup \mathcal{S}|)$. We also assume that all the insertion operations can be performed in $O(1)$ time, as all the data structures that might be exploited are either hash maps or vectors. Overall, this makes a worst case scenario time complexity of $O(|\mathcal{S}|m + |\bigcup \mathcal{S}|)$.

In MCSREFINEMENT, as checking the subset relationship among itemsets has at most a cost proportional to the size k of each itemset, this makes such sorting $O(|\bigcup \mathcal{S}| \log(|\bigcup \mathcal{S}|))$ when $k = 1$ and $O(|\bigcup S| \log(c))$ when $k = {}^{|\cup \mathcal{S}|}/_{c}$ for c constant; so, even in this scenario, we have the worst case scenario computational complexity for $k = 1$. Last, in the worst case scenario, the double for loop has a computational complexity comparable to $O(|\mathcal{S}|^2)$, as we perform an iteration over all the possible characterizations of the sets in $\mathcal{S}$ while trying to aggregate the remaining shared union operations further.

The overall time complexity of both phases becomes $O(|\mathcal{S}|(m + |\mathcal{S}|) + |\bigcup \mathcal{S}| \log(|\bigcup \mathcal{S}|))$.

Supplement III.1 Use Case: Running $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$ operators

This subsection describes how efficiently computing the results associated to the $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$ operator also requires the computation of the MAXIMALCOMMONSUBSET problem. This is achieved not only by performing all the $\mathsf{Compound}^{\mathcal{L},\tau}_{A/T}$ queries to be run on the same $\mathsf{AttributeTable}^{k}_{\mathcal{L}}$ in one single scan of the former but by also running the final result associated to the $\mathsf{Atom}^{\mathcal{L},\tau}_{A/T}$ by computing the shared intersections among different atoms' definition at most once. This detailed pseudocode is shown in Algorithm S5.

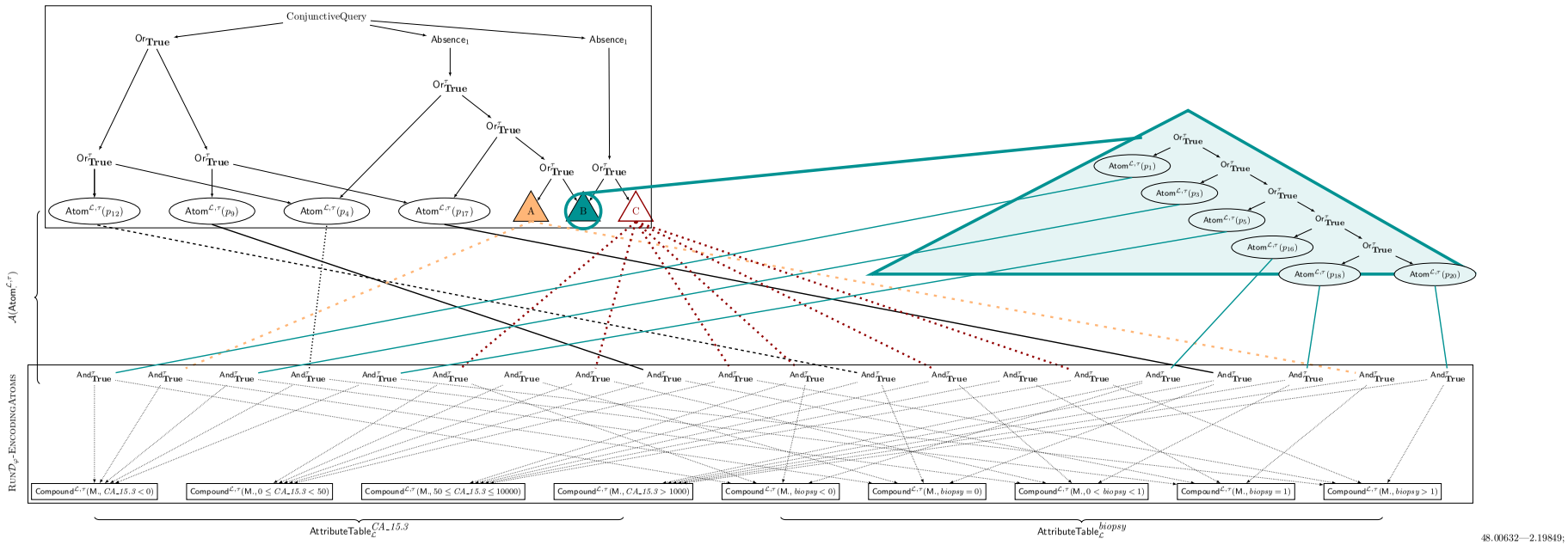

Figure S2. Detailed view of the query plan represented in Figure 5 for Example 15.

Algorithm S5 Query Plan Initialisation and Execution for $\mathcal{D}_\varphi$-encoding p_i atoms.

```
1:  global keyToLabelToSortedIntervals ← {}; S_Σ ← {}; Results ← {}; Cache ← {}

2:  procedure ExecuteRangeQueries(L)
3:      for all ⟨κ, map⟩ ∈ keyToLabelToSortedIntervals do
4:          if table AttributeTable^κ_L not exists then continue
5:          for all ⟨a, L⟩ ∈ map (parallel) do
6:              if ∃ ⟨beg, end⟩ . ⟨beg, end⟩ ∈ AttributeTable^κ_L.primary_index[β(a)] then
7:                  Sort(L)
8:                  for all ⟨low_κ ≤ κ ≤ up_κ, idx⟩ ∈ L do        ▷ Computing Compound^{L,τ}_?(a, κ, [low_κ, up_κ])
9:                      beg ← LowerBound(AttributeTable^κ_L, beg, low_κ, end)
10:                     mid ← UpperBound(AttributeTable^κ_L, beg, up_κ, end)
11:                     while beg<mid and beg<end do
12:                         i, j ← ActivityTable_L[AttributeTable^κ_L[beg](Offset)](Trace, Event)
13:                         Results[idx].put(⟨i, j⟩)
14:                         beg++;
15:                     end while
16:                     beg ← mid; Sort(Results[idx])
17:                     if beg ≥ end then break
18:                 end for
19:             end if
20:         end for
21:     end for

22: procedure Run$\mathcal{D}_\varphi$-encodingAtoms(L)
23:     ExecuteRangeQueries(L)
24:     for all ⟨p_i, Result⟩ ∈FinitarySetOperations(S_Σ, Results, ⋂) do        ▷ Algorithm S4
25:         Cache[p_i] ← Result
26:     end for
27:     S_Σ ← {}; Results ← {}; keyToLabelToSortedIntervals ← {}

28: function A(Atom^{L,τ})(ψ)        ▷ require ψ.atom = {p_ι} with p_ι from the D_φ-encoding pipeline
29:     if ψ.isActivation then                                   ▷ Atom^{L,τ}_A(p_ι)
30:         return { ⟨i, j, {A(j)}⟩ | ⟨i, j⟩ ∈ Cache[p_ι] }
31:     else if ψ.isTarget then                                  ▷ Atom^{L,τ}_T(p_ι)
32:         return { ⟨i, j, {T(j)}⟩ | ⟨i, j⟩ ∈ Cache[p_ι] }
33:     else                                                     ▷ Atom^{L,τ}(p_ι)
34:         return { ⟨i, j, ∅⟩ | ⟨i, j⟩ ∈ Cache[p_ι] }
35:     end if
```

First, we evaluate each compound condition over the AttributeTable$^k_\mathcal{L}$, as each atom is a conjunction of interval queries expressed as compound conditions. As the query compiler grouped such intervals by AttributeTable$^k_\mathcal{L}$ and associated activity labels by calling the RetrieveIntervals method in the `keyToLabelToSortedIntervals` map, we now need to iterate over such a map and access the AttributeTable$^\kappa_\mathcal{L}$ for each key κ appearing in it (Line 4). Each AttributeTable$^k_\mathcal{L}$ is then accessed by activity label a through its primary index (Line 6): if the primary index points to a specific region in the table delimited by a beginning and end table offset, we can access it and query it via the sorted compound condition. For each of those (representing elementary intervals), we get the offset *beg* (and *end*) to the first element within the block equal or greater to the lower bound low_κ (or strictly greater to the upper bound up_κ). Such bounds are returned in Lines 9 and 10. For each record existing in such a range, we reconstruct the trace and event id containing the desired value (Line 12) and then store it into a preliminary result[24] (Line 13). As both the elementary intervals and the table are sorted in ascending order by value and given that each elementary interval is non-overlapping by construction, we can query the next interval in L from the remaining portion of the table, thus steadily reducing the table visiting cost by further reducing the data over which the scan is performed. As these results are not sorted by increasing trace and event id rather than by increasing value in V, we have to sort them accordingly (Line 16) to be compatible with the intermediate results ρ requirements. Please observe that despite each of these intervals could appear in the definition of multiple atoms, this construction guarantees that each of these queries is performed at most once per occurring interval.

[24] Please observe: **not** the intermediate representation ρ!

Second, we can compute the events associated with each p_i atom returned from the $\mathcal{D}_\varphi$-encoding pipeline by intersecting the result associated with each of the intervals. The result of such intersections is then stored in an intermediate cache (Line 25). We can perform these intersections efficiently by guaranteeing the computation of each sub-intersection shared across different atoms at most once through the calculation of maximal common subsets occurring in intersections' sub-expressions.

Third, while evaluating each atom from our scheduled operator in ψ (Line 7 from Algorithm 6), we can retrieve the result from the cache and, dependingly on whether this expression was associated to an activation (Line 29 from Algorithm S5), a target condition (Line 31), or neither of these (Line 34), we can now effectively represent the result as an intermediate result with the appropriate label markers in L.

Example S4. *The lowermost boxes from Figure S2 represent the compound conditions extracted from the atoms appearing on the leaf nodes of the query plan, as well as their associated result stored in "Results" after running the* EXECUTERANGEQUERIES. *Their connection with the corresponding atoms could be tracked by navigating the edges backwards. The result of the intersection between all of the possible compound conditions as per atom definitions is then computed in the* ***for*** *loop in* RUN$\mathcal{D}_\varphi$-ENCODINGATOMS. *At this point, the execution of the leaf nodes on the query plan boils down to accessing the upper nodes in the box above and converting such results according to the activation or target condition associated with them. This mechanism guarantees that the results and the tables are accessed at most once, independently from the activation/target mark associated with the atom nodes.*

Supplement IV Data And Models

We wanted to test our claim that exploiting data access minimisation techniques ~~to~~ can quicken conformance checking as well as model mining time. In addition, we want to prove that even in the worst-case scenario, we still achieve orders of magnitude of gains versus state-of the art solutions. Therefore, we provide two models, Table S2 and Table S1a (also available at [17]). The former was adapted from Burattin et al. [6], where we exploited the constraints they provided, allowing for a direct comparison. These tables are displayed under the assumption from [6], where the trace payloads are injected into every event. In our actual implementation, we inject a `__trace_payload`event at the beginning every trace. Therefore, M_{11} from Table S1a has a KnoBAB representation as Table S1b, where the resulting model size is five (seven) for Burattin (KnoBAB).

By performing conjunctions among these, we were able to provide 11 models, the largest consisting of five (seven, due to trace payload injection) clauses. The queries constructing these larger models are similar, for example, even the model $q_1 \wedge q_2$ hold the clause Response(`A_SUBMITTED`,**true**,`A_ACCEPTED`,**true**) twice. These clauses need not be duplicated in the query plan, and therefore these models should better demonstrate the gains from data access minimisation. Conversely, Table S1a denotes an entirely novel model, that avoids similar queries. With each sub-model, entirely new events are considered, and even within those payload conditions vary. As such, our pipeline can gain much less from DAG optimization techniques. This model, therefore, provides a more appropriate case for analysing these proposed enhancements.

Table S1. Worst-case scenario (SCENARIO 1) model representation for the BPIC_2012 dataset.

(a) Clause constituents of each model, where each sub-model is guaranteed to be a subset of the larger.

Model	*Clauses*
$M_1 =$	$q_1 :=$ Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) $q_2 :=$ Response(A_SUBMITTED, AMOUNT_REQ $\geq 10^3$, A_ACCEPTED, **true**) $q_3 :=$ Response(A_SUBMITTED, AMOUNT_REQ $< 10^3$, A_ACCEPTED, **true**) $q_4 := q_1$ where A_SUBMITTED.org:resource $=$ A_ACCEPTED.org:resource $q_5 := q_1$ where A_SUBMITTED.org:resource $\neq$ A_ACCEPTED.org:resource
$M_2 = M_1+$	$q_6 :=$ Response(W_Completeren aanvraag, **true**, W_Valideren aanvraag, **true**) $q_7 :=$ Response(W_Completeren aanvraag, **true**, O_CANCELLED, **true**) $q_8 := q_6$ where W_Valideren aanvraag.org:resource $\neq$ W_Valideren aanvraag.org:resource $q_9 :=$ Response(W_Valideren aanvraag, AMOUNT_REQ $= 5 \cdot 10^3$, O_CANCELLED, **true**) $q_{10} := q_9$ where W_Valideren aanvraag.org:resource $=$ O_CANCELLED.org:resource
$M_3 = M_2+$	$q_{11} :=$ Response(O_SELECTED, **true**, O_CANCELLED, **true**) $q_{12} := q_{11}$ where O_SELECTED.org:resource $=$ O_CANCELLED.org:resource $q_{13} :=$ Response(O_SELECTED, AMOUNT_REQ $< 8 \cdot 10^3$, O_CANCELLED, **true**) $q_{14} := q_{13}$ where O_SELECTED.org:resource $=$ O_CANCELLED.org:resource $q_{15} :=$ Response(O_SELECTED, AMOUNT_REQ $> 10^3$, O_CANCELLED, **true**) where O_SELECTED.org:resource $\neq$ O_CANCELLED.org:resource
$M_4 = M_3+$	$q_{16} :=$ Response(A_PARTLYSUBMITTED, **true**, A_DECLINED, **true**) $q_{17} := q_{16}$ where A_PARTLYSUBMITTED.org:resource $=$ A_DECLINED.org:resource $q_{18} :=$ Response(A_PARTLYSUBMITTED, AMOUNT_REQ $> 2 \cdot 10^4$, A_DECLINED, **true**) $q_{19} :=$ Response(A_PARTLYSUBMITTED, AMOUNT_REQ $> 2 \cdot 10^4$, A_CANCELLED, **true**) $q_{20} := q_{18}$ where A_PARTLYSUBMITTED.org:resource $=$ A_DECLINED.org:resource

(b) Distinguishing consisting clauses of from Table S2, where [6] must inject the trace payload into each event.

Clause	*Declare Analyzer Representation*	*KnoBAB representation*
q_1	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**)	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**)
q_2	Response(A_SUBMITTED, AMOUNT_REQ $\geq 10^3$, A_ACCEPTED, **true**)	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) Exists(__trace_payload, AMOUNT_REQ $\geq 10^3$, ≥ 1)
q_3	Response(A_SUBMITTED, AMOUNT_REQ $< 10^3$, A_ACCEPTED, **true**)	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) Exists(__trace_payload, AMOUNT_REQ $< 10^3$, ≥ 1)
q_4	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) where A_SUBMITTED.org:resource $=$ A_ACCEPTED.org:resource	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) where A_SUBMITTED.org:resource $=$ A_ACCEPTED.org:resource
q_5	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) where A_SUBMITTED.org:resource $\neq$ A_ACCEPTED.org:resource	Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) where A_SUBMITTED.org:resource $\neq$ A_ACCEPTED.org:resource

Table S2. Declare Models for a data access best-case scenario for the BPIC_2012 dataset. These queries are performed under the assumption the trace payload has been injected into every event of the trace.

Model Name	*Conjunctive Query*
M_1	$q_1 :=$ Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**)
M_2	$q_2 :=$ Response(A_SUBMITTED, AMOUNT_REQ $\geq 10^3$, A_ACCEPTED, **true**)
M_3	$q_3 :=$ Response(A_SUBMITTED, AMOUNT_REQ $< 10^3$, A_ACCEPTED, **true**)
M_4	$q_4 := q_1$ where A_SUBMITTED.org:resource $=$ A_ACCEPTED.org:resource
M_5	$q_5 := q_1$ where A_SUBMITTED.org:resource $\neq$ A_ACCEPTED.org:resource
M_6	$q_1 \wedge q_2$
M_7	$q_1 \wedge q_2 \wedge q_4$
M_8	$q_1 \wedge q_3 \wedge q_4$
M_9	$q_1 \wedge q_2 \wedge q_5$
M_{10}	$q_1 \wedge q_3 \wedge q_5$
M_{11}	$q_1 \wedge q_2 \wedge q_3 \wedge q_4 \wedge q_5$