Article

Not peer-reviewed version

DECLARE*d*: A Polytime LTL_f Fragment

Giacomo Bergami *

Posted Date: 6 March 2024

doi: 10.20944/preprints202403.0286.v1

Keywords: LTLf; Declare; Verified Artificial Intelligence; Equational logic and rewriting

 doi:10.20944/preprints202403.0286.v1

Article

DECLARE*d*: A Polytime LTL_f Fragment

Giacomo Bergami

Newcastle University; Giacomo.Bergami@newcastle.ac.uk

Abstract: This paper considers a specific fragment of Linear Temporal Logic for Finite traces, DECLARE*d*, which, to the best of our knowledge, we prove for the first time to be a polytime fragment of LTL_f. We derive this in terms of the following ancillary results: we propose a set of novel LTL_f equivalence rules that, when applied to LTL_f specifications, lead to an equivalent specification which can be computed faster by any existing verified temporal artificial intelligence task. We also introduce the concept of temporal non-simultaneity, prescribing that two activities shall never satisfy the same atom, and temporal short-circuit, that occurs when a specification interpreted in LTL would accept an infinitely long trace while, on LTL_f, it can be rewritten so to postulate the absence of certain activity labels. We test these considerations over formal synthesis (Lydia), SAT-Solvers (AALTAF) and formal verification (KnoBAB) tools, where formal verification can be also run on top of a relational database and can be therefore expressed in terms of relational query answering. We show that all these benefit from the aforementioned assumptions, as running their tasks over a rewritten equivalent specification will improve their running times.

Keywords: LTL_f; declare; Verified Artificial Intelligence; equational logic and rewriting

---

Verified Artificial Intelligence [1] calls for exact procedures ascertaining whether a model of the system $\mathfrak{S}$ abides by the specifications in Φ through yes or no answers ($\mathfrak{S} \vDash \Phi$) when written in a formalism for efficient computations, either for verifying the compliance of a system to a specification (formal verification [2]) or for producing a system abiding by a given specification (formal synthesis [3]). This can be determined after a specification mining phase used to extract Φ from a system $\mathfrak{S}$ [4]; these considerations bridge temporal reasoning with artificial intelligence, as in both we can extract a specification from the data that can be used to determine decision problems. Under these assumptions, we are then interested in a temporal description of such systems, when different runs are collected as logs $\mathfrak{S}$ and referred to as traces $\sigma \in \mathfrak{S}$. These are temporally ordered records of observed and completed (or aborted) labelled activities. We are then interested in temporal specifications expressible over a fragment of Linear Temporal Logic for Finite traces (LTL_f), where LTL_f assumes that there is only one possible immediately following event to another and that the traces of interest contain a finite number of events. The major difference between LTL [5] and LTL_f is that, while the former might also prescribe the need for traces of infinite length, the latter will discard any temporal behaviour requiring such infinite traces to occur. This is evident from their procedural characterization through automata: while LTL models can be expressed as Büchi automata, LTL_f can be conveniently represented as NFAs only accepting finite traces [6]. To the best of our knowledge, this paper studies these situations for the first time and refers to them as **temporal short-circuits**:

Example 1. *Let us assume to have a temporal specification* $\Phi = \Box(a \Rightarrow \bigcirc c) \wedge \Box(c \Rightarrow \bigcirc a)$*: while the interpretation of the former in LTL accepts any trace either containing neither* a*-s nor* c*-s or accepting either* $(ac)^\omega$ *or* $(ca)^\omega$ *where* ω *is the first infinite ordinal, the latter can never occur in* LTL_f*, thus prescribing the absence of any* a*-s or* c*-s from the patterns. By interpreting* Φ *in* LTL_f*, we can then express it equivalently to* $\Box\neg a \wedge \Box\neg c$ *(§2.2) while preferring the latter representation as it completely removes the need to check whether the constraint leads to infinite behaviour never expressible in finite traces.*

Declarative languages in the context of Business Process Management (BPM) such as Declare [7] ease the practitioners' task to understand complex log patterns of interest in a straightforward way: by restricting the set of all the possible temporal behaviours of interest to the one in Table 1, we can conveniently extract compact specifications in which conformance checking tasks determine

 doi:10.20944/preprints202403.0286.v1

the abidance by the hospitalization procedures [8]. These specifications do not necessarily have to be hard-coded, but can be mined from such logs [4]. For BPM, each event is associated with exactly one single label [9] and, under the occasion that each event is also dataful and therefore associated with some data payload, we can always generate a finite set of mutually exclusive atoms partitioning the data space into non-overlapping intervals [10]. This ensures the theoretical possibility of defining atoms so an event will satisfy at most one of them. This evidence is also corroborated by data as represented in the real world: recent work on time series segmentation showed the possibility of representing a time series as a sequence of dichotomous increase and non-increase events [11] as well as the transitioning of a system into distinct non-overlapping states [12]. Furthermore, different types of malware can be distinguished just from the distinct name of the system calls being invoked at the operative system level [13,14]. As it is a common assumption in specification mining algorithms using Declare to return a finite conjunction of specifications, it is quite common to return inconsistent specifications under the **temporal non-simultaneity axiom** (§2.3) when support metrics below 100% are considered for increasing the algorithmic recall (see Example 2). Declare works under such an axiom, as BPM's practitioners implicitly assume that each trace event corresponds to explicitly one activity event, where all the distinct activity labels are assumed to be mutually exclusive predicates. Detecting this in advance will prevent running any verified temporal artificial intelligence technique on such specifications for the aforementioned practical scenarios, as no trace will ever satisfy an inconsistent specification.

Example 2. *Let us assume to have the following log* $\mathfrak{S} = \{\mathsf{acdefac}, \mathsf{adcfeadad}, \mathsf{acugac}, \mathsf{addadduadd}\}$ *As "*$\mathtt{a}$*" appears in all traces, we return* $\Diamond \mathtt{a}$ *(*$\mathtt{Exists}(\mathtt{a})$ *in Declare) as well as postulating that, when an* $\mathtt{a}$ *activity occurs in the log, this is immediately followed by* $\mathtt{c}$ *50% of the times (*$\Box(\mathtt{a} \Rightarrow \bigcirc \mathtt{c})$ *or* $\mathtt{ChainResponse}(\mathtt{a}, \mathtt{c})$ *in DECLAREd with 50% support) and by the remaining 50% percent by d (*$\Box(\mathtt{a} \Rightarrow \bigcirc \mathtt{d})$ *or* $\mathtt{ChainResponse}(\mathtt{a}, \mathtt{d})$*). Under the assumption of Axiom 1, the occurrence of any "*$\mathtt{a}$*" cannot possibly occur, thus rewriting the two latter statements as* $\Box\neg\mathtt{a}$*. Still, this conflicts with the first occurrence clause, thus generating a globally un-satisfiable specification* $\bot$*.*

Table 1. DECLARE*d*: our Declare's subset of interest, where A (respectively, B) denote activation (resp., target) conditions.

Exemplifying clause (c_l)	$\mathrm{LTL_f}$ Semantics ($[\![c_l]\!]$)
$\mathsf{Exists}(A)$	$\Diamond A$
$\mathsf{Absence}(A)$	$\Box\neg A$
$\mathsf{Choice}(A, A')$	$\Diamond A \vee \Diamond A'$
$\mathsf{NotCoExistence}(A, A')$	$\neg(\Diamond A \wedge \Diamond A')$
$\mathsf{ExlChoice}(A, A')$	$[\![\mathsf{Choice}(\mathtt{A}, \mathtt{A'})]\!] \wedge [\![\mathsf{NotCoExistence}(\mathtt{A}, \mathtt{A'})]\!]$
$\mathsf{RespExistence}(A, B)$	$\Diamond A \Rightarrow \Diamond B$
$\mathsf{CoExistence}(A, B)$	$[\![\mathsf{RespExistence}(\mathtt{A}, \mathtt{B})]\!] \wedge [\![\mathsf{RespExistence}(\mathtt{B}, \mathtt{A})]\!]$
$\mathsf{Precedence}(A, B)$	$\neg B \mathcal{W} A$
$\mathsf{Response}(A, B)$	$\Box(A \Rightarrow \Diamond B)$
$\mathsf{Succession}(A, B)$	$[\![\mathsf{Precedence}(\mathtt{A}, \mathtt{B})]\!] \wedge [\![\mathsf{Response}(\mathtt{A}, \mathtt{B})]\!]$
$\mathsf{NegSuccession}(A, B)$	$\Box(A \Rightarrow \neg\Diamond B)$
$\mathsf{ChainPrecedence}(A, B)$	$\Box(\bigcirc A \Rightarrow B)$
$\mathsf{ChainResponse}(A, B)$	$\Box(A \Rightarrow \bigcirc B)$
$\mathsf{ChainSuccession}(A, B)$	$[\![\mathsf{ChainPrecedence}(\mathtt{B}, \mathtt{A})]\!] \wedge [\![\mathsf{ChainResponse}(\mathtt{A}, \mathtt{B})]\!]$
$\mathsf{AltResponse}(A, B)$	$\Box(A \Rightarrow \bigcirc(\neg A \mathcal{U} B))$
$\mathsf{NegChainSuccession}(A, B)$	$\Box(A \Leftrightarrow \bigcirc\neg B)$
$\mathsf{AltPrecedence}(A, B)$	$[\![\mathsf{Precedence}(\mathtt{A}, \mathtt{B})]\!] \wedge \Box(B \Rightarrow \bigcirc(\neg B \mathcal{W} A))$
$\mathsf{AltSuccession}(A, B)$	$[\![\mathsf{AltPrecedence}(\mathtt{A}, \mathtt{B})]\!] \wedge [\![\mathsf{AltResponse}(\mathtt{A}, \mathtt{B})]\!]$

 doi:10.20944/preprints202403.0286.v1

This paper focuses on a DECLARE fragment, DECLARE*d* (Table 1), which is still affected by the aforementioned problem. These two preliminary definitions, where the second is optional, alongside the determination of a set of equivalence rules for DECLARE*d* (Figure 1) and the definition of an algorithm for DECLARE*d* rewriting Φ_d (§3) lead to our major result, that our proposed algorithm runs in $\textsc{Poly}(n)$ (§4). In fact, such algorithm[1] will return $\bot$ if the module has inconsistencies, $\top$ if this is detected trivially true, and a rewritten set of DECLARE*d* clauses Φ'_d when possible. As a byproduct of the previous result, we show that such running times were not achieved by other tools not running on the same fragment:

- Our rewritten equivalent specification Φ'_d speeds up the time of existing verified temporal artificial intelligence algorithms (§1) if compared with their runtime over the original specification Φ_d, thus proving that such algorithms do not support the notion of *temporal short-circuit* (§5.1).
- Under *temporal non-simultaneity*, the time required for both running a verified temporal artificial intelligence task and computing Φ'_d is also smaller than the running time of running such tasks over Φ (§5.2).

Graph Notation. (See §A) We denote a *(finite) graph* G as a pair (V, E), where V is a set of vertices and $E \subseteq V^2$ is a set of directed edges; to denote a vertex (or edge) set of a graph G, we use the notation V_G (or E_G). $\textsc{Out}_G(u)$ (or $\textsc{In}_G(u)$) is the set of the vertices reachable through the *outgoing* (or *incoming*) edges for $v \in V_G$. *Removing a vertex* $u \in V_G$ from a graph G ($V_G^-(v)$) requires also removing all the incoming and outgoing edges from a graph while *removing an edge* $(u, v) \in E_G$ from such a graph ($E_G^-(u, v)$) also requires removing the nodes u or v if such removal nullifies the degree of such nodes. $\wp(S)$ is the *powerset* of S. We represent a *(hash)multimap* f associating a single activity in Σ to a finite subset of Σ as a finite function $f \colon \Sigma \to \wp(\Sigma)$ where, for each $x \in \Sigma$ not being a key for the multimap, we guarantee $f(x) = \varnothing$ and $f(x) \neq \varnothing$ otherwise; an empty multimap f returns $\varnothing$ for each $x \in \Sigma$. $\textsc{Del}_f(x)$ removes x as a key of the multimap while $\textsc{Put}_f(x, u)$ adds u to the set of values associated to x in f. Given V a set of vertices and $\iota \colon |V| \leftrightarrow \{0, \ldots, |V| - 1\}$ a bijection enumerating each vertex in V to, a *directed circulant graph* $\mathcal{C}_L^{n,\pm}$ on n vertices has each vertex $v \in V$ adjacent to the immediately preceding and following vertices in V in the order expressed within a set of natural numbers $L \in \wp(\mathbb{N})$, i.e. $E_{\mathcal{C}_L^{n,\pm}} := \left\{ (u, v) \in V^2 \mid \exists k \in L.\ \iota(v) = (\iota(u) \pm k) \bmod n \right\}$. Given this, we define $\mathcal{C}_{\{1\}}^{n,+}$ as a cyclic graph representing exactly one *chain* and $\mathcal{C}_{\{0,\ldots,n-1\}}^{n,\pm}$ is a *complete graph*.

1. Brief Related Work

Formal Synthesis.

Lydia[2] [3] generates a DFA for a LTL_f specification Φ such that it will accept a trace σ iff. $\sigma \vDash \Phi$. This works over a finite alphabet Σ inferred from such formula. The authors efficiently do so by exploiting a compositional bottom-up approach after rewriting a LTL_f formula into an equivalent LDL_f one. Automata operations are implemented using MONA for compact representation. Benchmarks show the effectiveness of such an approach if compared to competing ones. By considering the effects of specification rewriting in automata generation, we want to verify whether temporal short-circuit rewriting tasks are already occurring over Lydia while building the automaton instead of pursuing the approach in §2.2 (see Section 2.2 vs. Corollary A1).

Formal Verification.

KnoBAB[3] [2] is a tool implementing the semantics for LTL_f operators into custom relational operators ($\texttt{xt}\text{LTL}_\text{f}$) providing a 1-to-1 mapping with the former. This was achieved by adequately

[1] https://anonymous.4open.science/r/DECLAREd-B1BF
[2] https://github.com/whitemech/lydia
[3] https://github.com/datagram-db/knobab

 doi:10.20944/preprints202403.0286.v1

representing all the traces in a log $\mathfrak{S}$ under a main memory columnar representation. Its architecture stores all the activities associated to traces' events in an `ActivityTable`, which is then sorted by increasing activity id in Σ, trace id, and event id. At loading time, the system also builds up a `CountingTable`, which determines the number of occurrences of each activity label per trace. This architecture supports MAX-Sat queries, declarative confidence and support, and returning the traces satisfying a given clause alongside its associated activated and target conditions ($\sigma \vDash \Phi$ for $\sigma \in \mathfrak{S}$). As KnoBAB outperforms existing tools for formal verification, we take this as a computational model of reference for determining the computational complexity of formal verification tasks over given specifications Φ_d in $\texttt{xt}\text{LTL}_\text{f}$ (see Section 2.2).

SAT-Solvers.

AALTAF[4] [15] is a SAT-checker determining whether an LTL_f formula is satisfiable by generating a corresponding transition system where each state represents subformulæ of the original LTL_f specification while leveraging traditional SAT-solvers. Differently from KnoBAB, which determines whether traces in a log satisfy an LTL_f specification expressed in algebraic terms ($\texttt{xt}\text{LTL}_\text{f}$) or not, AALTAF is more general than this and determines whether no traces will ever satisfy a given specification, thus determining its unsatisfiability, or whether there might exist a finite trace allowing this. This paper will show that DECLARE*d* provides a polytime fragment of LTL_f for which our equational rewriting algorithm (§3) also provides a decidable polyspace decision function for satisfiability (see Theorem 1).

LTL_f Modulo Theories

While LTL_f is generally known as a decidable language, most recent research [16] also consider decidable fragments of LTL_f also involving first-order logic arithmetic properties. Differently from our proposed characterization, events are not labelled as required when actions need to be ascertained. Furthermore, none of the proposed decidable fragments with arithmetic properties involves considerations on polytime complexity, while still referring to polyspace complexity.

2. Preliminaries

We consider temporal specifications Φ_d expressed in DECLARE*d* as a finite set of clauses from Table 1 being instantiated over a finite set of possible activity labels Σ. By interpreting this as conjunctive models [10], we can equivalently represent Φ_d as the finite conjunction of the LTL_f semantics associated to such clauses, i.e. $\Phi \equiv \bigwedge_{c_l \in \Phi_d} [\![c_l]\!]$. We use both notations interchangeably. Proofs are postponed to the Appendix due to space limitations.

2.1. Rewriting Rules

Figure 1 identifies any possible rewriting of the DECLARE*d* clauses into `Absence` or `Exists`, while determining the effects of the latter when interfering with the templates' activation (A) or target (B) conditions. If available, we are also interested in rewriting rules to identify an inconsistency leading to an unsatisfiable specification $\bot$. We now consider rewriting rules not assuming the temporal non-simultaneity axiom, thus remarking on the possibility of encoding these in already-existing tools as an LTL_f pre-processing step without any further assumption. As $\texttt{AltPrecedence}(\texttt{a},\texttt{b})$ can be rewritten as the LTL_f expression associated to $\texttt{Precedence}(\texttt{a},\texttt{b})$ as well as $\Box(\texttt{b} \rightarrow \bigcirc(\neg\texttt{b}\ \mathcal{U}\ \texttt{a}))$, we named `AltPrecedence` rewriting rules only the ones related to this LTL_f expression. Due to the limited space, we omit the proofs, but the reader can easily verify their correctness by checking that both sides of the equivalences generate the same automata[5].

[4] https://github.com/lijwen2748/aaltaf
[5] An on-line tool is available here: http://ltlf2dfa.diag.uniroma1.it/

 doi:10.20944/preprints202403.0286.v1

In all the rules, we assume that $\mathtt{a} \neq \mathtt{b}$ where $\mathtt{a}, \mathtt{b} \in \Sigma$. Other rewriting rules (Lemma A6) are implicitly assumed in forthcoming algorithmic subroutines (Section 3.1) and while loading and indexing specifications (Algorithm 2).

ChainResponse Rules ($G^{\mathtt{cr}}$)

$$\mathtt{ChainResponse(a,a)} \wedge \mathtt{Absence(a)} \equiv \mathtt{Absence(a)}$$
$$\mathtt{ChainResponse(a,a)} \wedge \mathtt{AltResponse(a,a)} \equiv \mathtt{Absence(a)}$$
$$\mathtt{ChainResponse(a,b)} \wedge \mathtt{NegChainSuccession(a,b)} \equiv \mathtt{Absence(a)}$$

Response Rules ($G^{\mathtt{r}}$)

$$\mathtt{Response(a,a)} \wedge \mathtt{Absence(a)} \equiv \mathtt{Absence(a)}$$
$$\mathtt{Response(a,c)} \wedge \mathtt{ChainResponse(a,b)} \equiv \mathtt{ChainResponse(a,b)}$$
$$\mathtt{Response(a,b)} \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(b)} \wedge \mathtt{Absence(a)}$$

NegSuccession Rules ($G^{\mathtt{ns}}$)

$$\mathtt{NegSuccession(a,b)} \wedge \mathtt{Absence(a)} \equiv \mathtt{Absence(a)}$$
$$\mathtt{NegSuccession(a,b)} \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(b)}$$
$$\mathtt{NegSuccession(a,b)} \wedge \mathtt{ChainResponse(a,b)} \equiv \mathtt{Absence(a)}$$
$$\mathtt{NegSuccession(a,b)} \wedge \mathtt{Response(a,b)} \equiv \mathtt{Absence(a)}$$

AltResponse Rules ($G^{\mathtt{ar}}$)

$$\mathtt{AltResponse(a,b)} \wedge \mathtt{Absence(a)} \equiv \mathtt{Absence(a)}$$
$$\mathtt{AltResponse(a,b)} \wedge \mathtt{ChainResponse(a,b)} \equiv \mathtt{ChainResponse(a,b)}$$
$$\mathtt{AltResponse(a,b)} \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(b)} \wedge \mathtt{Absence(a)}$$
$$\mathtt{AltResponse(a,b)} \wedge \mathtt{NegSuccession(a,b)} \equiv \mathtt{Absence(a)}$$

AltPrecedence Rules ($G^{\mathtt{ap}^-}$)

$$\Box(\mathtt{b} \rightarrow \bigcirc(\neg \mathtt{b}\ \mathcal{W}\ a)) \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(b)}$$
$$\Box(\mathtt{b} \rightarrow \bigcirc(\neg \mathtt{b}\ \mathcal{W}\ a)) \wedge \mathtt{ChainResponse(b,a)} \equiv \mathtt{ChainResponse(b,a)}$$

RespExistence Rules ($G^{\mathtt{re}}$)

$$\mathtt{RespExistence(a,b)} \wedge \mathtt{Exists(a)} \wedge \mathtt{Absence(b)} \equiv \bot$$
$$\mathtt{RespExistence(a,b)} \wedge \mathtt{Exists(a)} \equiv \mathtt{Exists(a)} \wedge \mathtt{Exists(b)}$$
$$\mathtt{RespExistence(a,b)} \wedge \mathtt{Exists(b)} \equiv \mathtt{Exists(b)}$$
$$\mathtt{RespExistence(a,b)} \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(a)} \wedge \mathtt{Absence(b)}$$

Choice Rules ($G^{\mathtt{c}}$)

$$\mathtt{Choice(a,b)} \wedge \mathtt{Exists(a)} \wedge \mathtt{Exists(b)} \equiv \mathtt{Exists(a)} \wedge \mathtt{Exists(b)}$$
$$\mathtt{Choice(a,b)} \wedge \mathtt{Absence(a)} \wedge \mathtt{Absence(b)} \equiv \bot$$
$$\mathtt{Choice(a,b)} \wedge \mathtt{Absence(a)} \equiv \mathtt{Absence(a)} \wedge \mathtt{Exists(b)}$$
$$\mathtt{Choice(a,b)} \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(b)} \wedge \mathtt{Exists(a)}$$

NotCoExistence Rules ($G^{\mathtt{nce}}$)

$$\mathtt{NotCoExistence(a,b)} \wedge \mathtt{Absence(a)} \equiv \mathtt{Absence(a)}$$
$$\mathtt{NotCoExistence(a,b)} \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(b)}$$
$$\mathtt{NotCoExistence(a,b)} \wedge \mathtt{Exists(a)} \wedge \mathtt{Exists(b)} \equiv \bot$$
$$\mathtt{NotCoExistence(a,b)} \wedge \mathtt{Exists(a)} \equiv \mathtt{Exists(a)} \wedge \mathtt{Absence(b)}$$
$$\mathtt{NotCoExistence(a,b)} \wedge \mathtt{Exists(b)} \equiv \mathtt{Exists(b)} \wedge \mathtt{Absence(a)}$$
$$\mathtt{NotCoExistence(a,b)} \wedge \mathtt{RespExistence(a,b)} \equiv \mathtt{Absence(a)}$$

Precedence Rules ($G^{\mathtt{p}}$)

$$\mathtt{Precedence(a,b)} \wedge \mathtt{Absence(a)} \equiv \mathtt{Absence(a)} \wedge \mathtt{Absence(b)}$$
$$\mathtt{Precedence(a,b)} \wedge \mathtt{Absence(b)} \equiv \mathtt{Absence(b)}$$

Figure 1. Some rewriting Rules for DECLARE*d* (§3.3)

2.2. Temporal Short-Circuit Rewriting

A finite conjunction of LTL_f *statements* $\varphi := \bigwedge_i \varphi_i$ *leads to a* ***temporal short-circuit*** *if this can be rewritten as a finitary conjunction, either* $\varphi' := \bigwedge_j \Box\neg a_j$ *or* $\varphi' := \bigwedge_j \Diamond a_j$*, for each distinct atom* a_j *freely occurring in* φ *when* φ' *is not syntactically equivalent to* φ*. We apply a* ***(temporal) short-circuit rewriting*** *to a* LTL_f *specification* Φ *if we replace any sub-formula* φ *in* Φ *leading to a temporal short-circuit with* φ'*.*

Short-circuits based on `ChainResponse` boil down to the absence of each of its atoms:

Given $A = \{\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_n\} \subseteq \Sigma$, $\Phi' := \bigwedge_{\mathtt{c}_i \in A} \Box\neg \mathtt{c}_i$ is equivalent to $\Phi := \Box(\mathtt{c}_n \Rightarrow \bigcirc \mathtt{c}_1) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(\mathtt{c}_i \Rightarrow \bigcirc \mathtt{c}_{i+1})$ in LTL_f.

Such a rewriting will streamline formal verification tasks: Given $A = \{\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_n\} \subseteq \Sigma$, computing $\Phi' := \bigwedge_{\mathtt{c}_i \in A} \Box\neg \mathtt{c}_i$ in lieu of $\Phi := \Box(\mathtt{c}_n \Rightarrow \bigcirc \mathtt{c}_1) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(\mathtt{c}_i \Rightarrow \bigcirc \mathtt{c}_{i+1})$ always leads to a positive average speed-up.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 March 2024 doi:10.20944/preprints202403.0286.v1

After representing all the $\texttt{ChainResponse}(\mathsf{a},\mathsf{c})$ in a input specification as a graph $G^{\texttt{cr}}$ with edge $\mathsf{a} \to \mathsf{c} \in E_{G^{\texttt{cr}}}$ and nodes $\mathsf{a},\mathsf{c} \in V_{G^{\texttt{cr}}}$, we can show as a corollary of the first lemma that this boils down to removing all circuits appearing over some nodes $\beta^{\texttt{cr}} \subseteq V_{G^{\texttt{cr}}}$ and rewriting such clauses as $(\bigwedge_{v\in\beta^{\texttt{cr}}} \texttt{Absence}(v)) \wedge (\bigwedge_{\substack{u\to v\in E_{G^{\texttt{cr}}} \\ u,v\notin\beta^{\texttt{cr}}}} \texttt{ChainResponse}(u,v))$ in polytime on the size of Φ (Corollary A1). We can infer similar lemmas for $\texttt{AltResponse}$ in terms of rewriting such resulting temporal short-circuits to absences (Lemma A2) thus resulting in time speed-up (Corollary A3).

2.3. Temporal Non-Simultaneity

Axiom 1 (Temporal Non-Simultaneity)**.** *Given the set of all the possible activity labels* Σ*, we prescribe that no distinct activity could occur simultaneously in the same instant of time. This can be expressed as* $\forall a, b \in \Sigma. a \neq b \Rightarrow \Box(a \wedge b \Rightarrow \bot)$.

As we assume a finite set of activity labels Σ to be fully known from our specification or data, we can represent this axiom as an extension of the properties Φ to be checked as a new property $\Phi_{\downarrow\Sigma} := \Phi \wedge \bigwedge_{\substack{\mathsf{a},\mathsf{b}\in\Sigma \\ \mathsf{a}\neq\mathsf{b}}} \Box(\mathsf{a} \wedge \mathsf{b} \Rightarrow \bot)$. As prior approaches using LTL_f did not consider this assumption, Φ should be directly stated as $\Phi_{\downarrow\Sigma}$ for both Lydia and AALTAF. On the other hand, our solver Reducer only takes Φ_d as it works under such an axiom. KnoBAB automatically assumes that each distinct activity label is different from the rest, thus entailing an implicit semantic difference between different types of events.

Rewriting Rules

We can identify that the following rewriting rule holds for $\mathsf{c} \neq \mathsf{b}$ in Σ, as we can never have an event being labelled with both c and b after the same occurring event:

$$[\texttt{ChainResponse}(\mathsf{a},\mathsf{b}) \wedge \texttt{ChainResponse}(\mathsf{a},\mathsf{c})]_{\downarrow\Sigma} \equiv \texttt{Absence}(\mathsf{a})$$

Temporal Short-Circuit Rewriting

We now consider temporal short-circuit rewriting rules that only hold under temporal non-simultaneity. For $\mathsf{a} \neq \mathsf{b}$ in Σ, as any b shall always occur after the first occurring a for Response, we can express it as:

$$\Box(\mathsf{a} \to \Diamond\mathsf{b})_{\downarrow\Sigma} \equiv \Box(\mathsf{a} \to \bigcirc\Diamond\mathsf{b})$$

Due to this, we need to discard the eventuality that $|A| = 1$, as $\Box(\mathsf{a} \to \Diamond\mathsf{a})$ is, on the other hand, trivially true and leads to no temporal short-circuit.

Given $A = \{\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_n\} \subseteq \Sigma$ with $|A| = n > 2$, $\Phi' := \bigwedge_{\mathsf{c}_i\in A} \Box\neg\mathsf{c}_i$ is equivalent to $\Phi := \left[\Box(\mathsf{c}_n \Rightarrow \Diamond\mathsf{c}_1) \wedge \bigwedge_{\substack{i\in\mathbb{N} \\ 1\leq i<n}} \Box(\mathsf{c}_i \Rightarrow \Diamond\mathsf{c}_{i+1})\right]_{\downarrow\Sigma}$ in LTL_f.

Given $A = \{\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_n\} \subseteq \Sigma$ with $|A| = n > 2$, computing $\Phi' := \bigwedge_{\mathsf{c}_i\in A} \Box\neg\mathsf{c}_i$ in lieu of $\Phi := \left[\Box(\mathsf{c}_n \Rightarrow \Diamond\mathsf{c}_1) \wedge \bigwedge_{\substack{i\in\mathbb{N} \\ 1\leq i<n}} \Box(\mathsf{c}_i \Rightarrow \Diamond\mathsf{c}_{i+1})\right]_{\downarrow\Sigma}$ always leads to a positive average speed-up.

 doi:10.20944/preprints202403.0286.v1

Algorithm 1 Algorithmic Subroutines for our specification rewriter

```
1: function Ex(x):= F ← Put_F(x, true); return |F(x)| = 1
2: function Abs(x):= F ← Put_F(x, false); return |F(x)| = 1
3: function Clear(x):=
4:     V⁻_Gcr(x); V⁻_Gar(x); V⁻_Gacs(x); V⁻_Gcr(x); V⁻_Gas(x); V⁻_Gre(x)
5:     for all b ∈ Out_Gch(x) do if not Ex(b) then return false
6:     for all b ∈ In_Gch(x) do if not Ex(b) then return false
7:     V⁻_Gch(x)
8:     for all b ∈ Out_GP(x) do
9:         if not Abs(x) then return false else E⁻_GP(x, b)
10:    return true
11: function Reduce*(x):=
12:    visited ← ∅
13:    toremove ← {x}
14:    while toremove ≠ ∅ do
15:        x ← Pop(toremove)
16:        if x ∈ visited then continue else visited ← visited ∪ {x}
17:        if not Abs(x) or not Clear(x) then return false
18:        toremove ← toremove ∪ In_G*(x)
19:        if ⋆ = ch then
20:            toremove ← toremove ∪ Out_G*(x)
21:            for all y ∈ toremove do: if not Expand^RE(y) then return false
22:    return true
23: function Reduce^P(x) override :=
24:    visited ← ∅
25:    toremove ← {x}
26:    while toremove ≠ ∅ do
27:        x ← Pop(toremove)
28:        if x ∈ visited then continue else visited ← visited ∪ {x}
29:        if not Abs(x) or not Clear(x) then return false
30:        toremove ← toremove ∪ Out_GP(x)
31:        V⁻_GP(x)
32:    return true
33: function Expand^RE(x):=
34:    visited ← ∅
35:    torexpand ← {x}
36:    while torexpand ≠ ∅ do
37:        x ← Pop(toremove)
38:        if x ∈ visited then continue
39:        visited ← visited ∪ {x}
40:        if not Ex(x) then return false else V⁻_Gch(x)
41:        for all u ∈ In_Gre(x) do: E⁻_Gre(u, x)
42:        for all u ∈ In_Gas(x) do: if not Abs(u) or not Clear(u) return false
43:        for all u ∈ Out_Gre(x) do
44:            if (true ∉ F(u)) then torexpand ← torexpand ∪ {u}
45:            V⁻_Gre(x)
46:
```

3. Reducer: Equational Rewriting

The Reducer algorithm for rewriting Φ_d into Φ'_d proceeds as follows: after showing the subroutines for removing redundant clauses from the specification while propagating the detection of an inconsistency towards the function call chain (§3.1), we outline how a specification Φ_d can be efficiently loaded as a collection of graphs $G^\star$ for each DECLAREd template $\star$ for clause indexing (§3.2). After applying the aforementioned equivalence rules (§3.3), we apply the temporal short-circuit rewriting (§3.4) before returning the rewritten specification Φ'_d from the edges remaining from $G^\star$ and values in an F map storing `Absence` and `Exists` clauses. Upon detecting the joint satisfaction of an `Absence(x)` and `Exists(x)` for an activity label `x`, we immediately detect an inconsistency for which we return $\bot$. If the resulting specification appears to be empty in spite of no inconsistency being detected, we then obtain a trivially true specification $\top$. Otherwise, we return a rewritten specification Φ'_d.

3.1. Algorithmic Subroutines

Algorithm 1 shows crucial algorithmic subroutines ensuring to propagate the detection of an absence/presence of an activity label while dealing with clauses c_l derivable from the input specification Φ_d clauses of the specification.

Let F be a finite multimap associating each activity label $\mathtt{a} \in \Sigma$ to a set of booleans, where **true** (and **false**) denotes that `Exists(a)` (and `Absence(a)`) can be inferred from the specification. If both `Exists(a)` and `Absence(a)` are entailed, we deem the overall specification as inconsistent, for which we will return $\bot$. Ex in L. 1 (and Abs in L. 2) returns **false** whether the addition of `Exists` (or an `Absence`) to a specification makes it explicitly inconsistent.

Clear at L. 3 removes all the clauses in which activation condition x would never occur per `Absence(`x`)`. For `Choice(x, b)`, this triggers the generation of `Exists(b)` which, in turn, might lead to an inconsistent specification (L. 5 and 6). For `Precedence(x,b)`, the absence of the activation requires `Absence(b)`, which is then in turn added while testing for the specification's inconsistency (L. 9). The function returns **true** if the specification is not currently detected as inconsistent (L. 10).

Reduce at L. 11 can be applied to templates $\star$ such as `ChainResponse`, `Response`, and `AltResponse` for implementing a cascade effect upon the specification supporting `Absence(x)` by also requiring that the associated activations should be absent from the specification (L. 18). We return **true** if no inconsistency was detected, and **false** otherwise. This idea was revised so as to be applied to `Precedence` (L.

 doi:10.20944/preprints202403.0286.v1

24): for this, the absence of the activation triggers the necessity of the second argument to be absent as well, thus enforcing to visit the graph towards the outgoing edges (L. 31). We also ensure to remove all the vertices and edges associated with x (L. 32).

Dually, EXPAND$^{\texttt{re}}$ works by recursively applying the head from the tail of the `RespExistence` clauses upon the request that an event x shall exist in the data (L. 45). As this trivially exists, we remove all the clauses having this condition in the head of such rules (L. 42) while, if x appears as a second argument of a `NegSuccession(u,x)`, we still postulate for the absence of u from the specification (L. 43).

Algorithm 2 Loading binary clauses in primary memory

```
1:  function SpecificationLoading(Φ)
2:      match each x ∈ Φ with
3:          case Exists(a): if not Ex(a) then return ⊥
4:          case Absence(a): if not Abs(a) then return ⊥
5:          case ChainPrecedence(a,b): E+_{G^cp}(a,b)
6:          case ChainResponse(a,b): E+_{G^cr}(a,b); E+_{G^r}(a,b)
7:          case Response(a,b) s.t. a ≠ b: E+_{G^re}(a,b)
8:          case Precedence(a,b) s.t. a ≠ b: E+_{G^p}(a,b)
9:          case RespExistence(a,b) s.t. a ≠ b: E+_{G^re}(a,b)
10:         case NegSuccession(a,b):
11:             if a = b then
12:                 if not Abs(a) then return ⊥
13:             else
14:                 E+_{G^ns}(a,b)
15:         case NegChainSuccession(a,b): E+_{G^ncs}(a,b)
16:         case Succession(a,b) s.t. a ≠ b:
17:             E+_{G^s}(a,b); E+_{G^p}(a,b)
18:         case CoExistence(a,b) s.t. a ≠ b:
19:             E+_{G^re}(a,b); E+_{G^re}(b,a)
20:         case ChainSuccession(a,b):
21:             E+_{G^cr}(a,b); E+_{G^cp}(a,b); E+_{G^ap-}(b,a)
22:         case AltPrecedence(a,b) s.t. a ≠ b:
23:             E+_{G^ap-}(b,a); E+_{G^p}(a,b)
24:         case AltResponse(a,b):
25:             if a = b then
26:                 if not Abs(a) then return ⊥
27:             else
28:                 E+_{G^ar}(a,b); E+_{G^cr}(a,b)
29:         case AltSuccession(a,b):
30:             if a = b then
31:                 if not Abs(a) then return ⊥
32:             else
33:                 E+_{G^s}(a,b); E+_{G^ap-}(b,a); E+_{G^ar}(a,b)
34:         case Choice(a,b):
35:             if a = b then
36:                 if not Ex(a) then return ⊥
37:             else E+_{G^ch}(min{a,b}, max{a,b})
38:         case ExclChoice(a,b):
39:             if a = b then return ⊥
40:             E+_{G^ch}(min{a,b}, max{a,b})
41:             E+_{G^coe}(min{a,b}, max{a,b})
42:         case NotCoExistence(a,b):
43:             if a = b then
44:                 if not Abs(a) then return ⊥
45:             else E+_{G^ncs}(min{a,b}, max{a,b})
```

3.2. Specification Loading as Graphs

Algorithm 2 shows the process of loading and indexing the clauses from Φ_d in primary memory.

We add `Absence` and `Exists` in map F; at this stage, the specification is deemed inconsistent (returning $\bot$) if a given activity label a is required to both appear and be absent from each trace.

Binary clauses $\star(\mathtt{a},\mathtt{b})$ are loaded as edges $(\mathtt{a},\mathtt{b})$ of a graph $G^\star$ where $V_{G^\star} \subseteq \Sigma$. Clauses being the conjunction of other clauses are then rewritten into their constituents; `AltPrecedence`$(\mathtt{a},\mathtt{b})$ is rewritten into `Precedence`$(\mathtt{a},\mathtt{b})$, to be stored in an edge $(\mathtt{a},\mathtt{b})$ for $G^{\mathtt{p}}$, and $\Box(\mathtt{b} \rightarrow \bigcirc(\neg \mathtt{b}\ \mathcal{W}\ \mathtt{a}))$, to be stored in an edge $(\mathtt{b},\mathtt{a})$ for $G^{\mathtt{ap}-}$. For binary clauses entailing the universal truth $\top$ when both arguments are associated with the same activity label (e.g., `Response`), we avoid inserting this clause as an edge. For other clauses (e.g., `AltResponse`, L. 26), this same situation might be rewritten as the absence of a specific activity label which, if leading to an inconsistency, also immediately ensures to return an empty specification. Conversely, a `Choice` having both arguments being the same boils down to an `Exists`, which is also added in place of `Choice` (L. 35), while we might never have an `ExclChoice` where both arguments are the same (L. 38). For clauses being symmetric (e.g., `Choice`), we avoid duplicated entries by preferring only one of the two equivalent writings (e.g., `Choice(a,b)` over `Choice(b,a)` for $\mathtt{a} \preceq \mathtt{b}$, L. 36).

3.3. Applying Equational Rewriting

Equational rewriting for rules in Figure 1 is run as follows: we consider each graph in order of appearance in §2.1 and we iterate over its edges. For each of these, we detect their match with one of the cases appearing in the first binary clause $\star(\mathtt{a},\mathtt{b})$ on the left-hand side of the formula, and we look up for the occurrence of any clause appearing in the same hand-side.

If the condition described by the left-hand side is then reflected by the specification represented by edges for graphs $G^\star$ and F, we determine the rewriting strategy depending on the definition of the right-hand side. If the latter is $\bot$, we immediately return it and detect an unsatisfiable model. Also, we remove any edge $(\mathtt{a},\mathtt{b})$ from $G^\star$ if $\star(\mathtt{a},\mathtt{b})$ does not appear on the right-hand side of the formula, and

 doi:10.20944/preprints202403.0286.v1

we add any edge $\dagger(\mathtt{a},\mathtt{b})$ in $G^{\dagger}$ not appearing on the left-hand side. If an $\mathtt{Exists}(\mathtt{a})$ (or $\mathtt{Absence}(\mathtt{a})$) appears only on the right-hand side, we add it by invoking $\textsc{Ex}(\mathtt{a})$ (or $\textsc{Abs}(\mathtt{a})$), while immediately returning an empty specification if an inconsistency is detected while doing so. These methods are also changed according to the interdependencies across templates: e.g., the generation of new $\mathtt{Exists}(x)$ for `RespExistence` rules triggers $\textsc{Expand}^{\mathtt{re}}(x)$ instead, and the $\mathtt{Absence}(x)$ for `NotCoExistence` invokes $\textsc{Reduce}^{\mathtt{ch}}(x)$ which, in turn, will also call for $\textsc{Expand}^{\mathtt{re}}(x)$ as per Algorithm 1; $\mathtt{Absence}(x)$ for `RespExistence` will call for $\textsc{Reduce}^{\mathtt{re}}(x)$. Similar considerations can be provided for other templates and rules. This process does not clear out clauses, as we at least fill in F, from which we are also returning `Exists` or `Absence` for Φ_d'.

After applying the rules on a single graph $G^{\dagger}$, we then iterate over all the activities x required to be absent by the specification ($\mathbf{false} \in F(x)$), for which we run all the $\textsc{Reduce}^{\star}(x)$ methods and $\textsc{Clear}(x)$, through which we propagate the effect of requiring the absence of a specific activity label to all the clauses in the specification. If, while doing so, any inconsistency is detected by returning a **false**, we immediately return $\perp$ (see Lemma A5).

3.4. Applying Short-Circuit Rewriting

This further algorithmic step is run after running the *AltPrecedence Rules* and before running the *RespExistence* ones, thus potentially reducing the number of clauses to be considered due to the absence of a specific activity label.

We prefer to detect the existence of a circuit $v_{\alpha_1} \to \cdots \to v_{\alpha_n} \to v_{\alpha_1}$ of length $n+1$ through a DFS visit of the graph with back-edge detection [17]. Once we detect a circuit, we generate $\mathtt{Absence}(v_{\alpha_i})$ clauses for each node v_{α_i} in it, while removing such nodes from the graph. The latter operation is efficiently computed by creating a view over such a graph through an absence set R which will contain all of the nodes in the circuit being removed. Then, for each graph traversal, we avoid starting the visit from nodes in R and we avoid traversing edges leading to nodes in R. This avoids extremely costly graph restructuring operations. As by construction we cannot have a single non-connected node as each clause is represented by one single edge, if at the end of this reduction process we obtain nodes with zero degrees, such nodes were previously connected to nodes belonging to cycles and that were therefore also part of cycles: those also constitute `Absence` clauses.

For all the novel $\mathtt{Absence}(\mathtt{a})$ clauses being inserted in the specification in lieu of the detected temporal short-circuits, we also run all the available $\textsc{Reduce}^{\star}(x)$ methods as well as $\textsc{Clear}(x)$, thus ensuring a cascading effect removing the remaining clauses that will never be activated while keeping searching for inconsistencies via sub-routine calls.

4. A PolyTime($|\Phi_d|$) SAT-Solver for DECLARE*d*

This section is mainly to prove that satisfiability in DECLARE*d* can be carried out in polytime over the size of the original specification $|\Phi_d|$; this strengthens the previous informal results over conformance checking provided over KnoBAB over such a fragment. We twin this result with the overall algorithmic correctness.

Theorem 1. *The Reducer specification rewriting process is a decidable SAT-Solver for DECLAREd running in* $\textsc{Poly}(|\Phi_d|)$*-time.*

Proof. The main lemmas supporting this proof are reported in the appendix due to the lack of space. First, we need to prove that the previous section describes an always-terminating computation: given Lemmas A6–A8 and Corollary A4, we have that each previous subsection in §3 describes a polytime procedure. The composition of each single non-mutually-recursive sub-routine terminates and leads to a polytime decision function.

Last, we discuss the correctness of the resulting procedure. If Φ_d in input were a tautology, all the clauses in the original specification would have been cancelled out as they would have trivially held, thus providing no further declarative clause to be returned (Lemma A4). If, on the other hand,

 doi:10.20944/preprints202403.0286.v1

any inconsistency was detected, the computation would have stopped before returning the reduced specification, thus ignoring all the remaining rewriting steps (Lemma A5). If neither of the previous cases holds, we have then by exclusion a satisfiable specification Φ_d which is also rewritten into an equivalent specification Φ'_d under the temporal non-simultaneity axiom (Axiom 1). □

5. Empirical Evaluation

We determine a set of activity labels Σ from the Cybersecurity dataset [14] and by creating 8 distinct subsets $A_1, \ldots, A_8 \subseteq \Sigma$ of size $|A_i| = 2^i$ such that $A_i \subset A_j$ for each $1 \leq i < j \leq 8$. We then consider each A_i as a set of vertices for which we instantiate a complete graph, a cyclic graph representing a chain, and a circulant graph $\mathcal{C}^{|A_i|,\pm}_{\{0,1,2,3\}}$. Given each of these graphs g, we then generate a specification $\Phi_i^{c,g}$ for each of these A_i by interpreting each edge $(a, b) \in A_i^2$ in the generated graph as a declarative clause:

- *c=ChainResponse*: $\texttt{ChainResponse}(\texttt{a}, \texttt{b})$
- *c=Precedence*: $\texttt{Precedence}(\texttt{a}, \texttt{b})$
- *c=Response*: $\texttt{Response}(\texttt{a}, \texttt{b})$
- *c=RespExistence+Exists*: $\texttt{RespExistence}(\texttt{a}, \texttt{b})$
- *c=RespExistence+ExclChoice+Exists*: $\texttt{RespExistence}(\texttt{a}, \texttt{b})$, $\texttt{ExclChoice}(\texttt{a}, \texttt{b})$

For the last two cases, we also add a clause $\texttt{Exists}(\texttt{u})$ for $\texttt{u} = \min A_i$. Given the same A_i and u, we also generate two other specifications $\Phi_i^{c,g}$ where clauses are instead generated for each activity label $\texttt{a} \in A_i$:

- *c=(Chain+Alt)Response*: $\texttt{ChainResponse}(\texttt{a}, \texttt{a})$, $\texttt{AltResponse}(\texttt{a}, \texttt{a})$
- *c=ChainResponseAX*: $\texttt{ChainResponse}(\texttt{u}, \texttt{a})$.

We then expect that, if any of the aforementioned verified temporal artificial intelligence tasks provide no LTL_f or Declarative rule rewriting as per this paper, running any verified temporal artificial intelligence task over a $\Phi_i^{g,c}$ being generated from $g = \mathcal{C}^{|A_i|,\pm}_{\{0,\ldots,|\Sigma_i|-1\}}$ will take more time than running it over a $\Phi_i^{g',c}$ where $g' = \mathcal{C}^{|A_i|,\pm}_{\{0,1,2,3\}}$, which in turn will take more time than running a specification $\Phi_i^{g'',c}$ generated over $g'' = \mathcal{C}^{|A_i|,+}_{\{1\}}$. This last consideration also includes the aforementioned rewriting task in its worst-case scenario. As we are expecting that, for each c, each of these $\Phi_i^{c,g}$ for any of such graphs g will be then always rewritten into the same specification $\Phi_i'^{c}$, we are expecting to have similar running times for each rewritten specification, as we expect the running time in the latter to be dependant on the number of atoms/vertices and not on the number of clauses as the former.

Each of these generated specifications is then fed to our rewriting algorithm, which returns the specification in both the LTL_f representation required by Lydia and AALTAF and the declarative specification for KnoBAB. We discard parsing and query plan generation times for each solution: running times for Lydia only considered the time required for generating the DFA from the LTL_f formula as per their internal main-memory representation. For the formal verification task run in KnoBAB as a relational database query, we consider a sample of 9 traces from the original log over which we run the generated specifications. To alleviate potential out-of-memory issues, we test the specifications in batches of 10 at a time, thus sacrificing greater query plan minimization with the certainty of completing the computation without out-of-memory errors. On the other hand, the two aforementioned tools do not require a log to work, for which it is sufficient to have a specification. The dataset for specifications and logs is available online[6].

[6] https://drive.google.com/drive/folders/1EK44vwSBVdz_A17XZintDP1eeEoBh53Q?usp=sharing

 doi:10.20944/preprints202403.0286.v1

Due to the lack of space, we present a condensed version of the benchmarks, while §E gives more extensive plots.

5.1. Rule Rewriting without Temporal Non-Simultaneity

We now consider all the resulting specifications being generated except *Response* and *ChainResponseAX*, which are discussed in the next subsection as they assume the temporal non-simultaneity axiom. The ones here discussed are a mere application of temporal short-circuit rewriting. Therefore, this section aims at remarking on the generality of our result, which can be shown even without assuming 1.

Each plot in Figure 2 is grouped by c and the black solid line refers to the time required for the solver to generate $\Phi_{d_i}^{\prime c}$ from $\Phi_{d_i}^{c,g}$ in milliseconds. Missing data points refer to data points not being collected as the solutions went out of primary memory. For both Lydia and AALTAF, we consider running those over both the non-rewritten specification $\Phi_i^{c,g}$ as well as over the rewritten one $\Phi_i^{\prime c}$ (LYDIA(R) and AALTAF(R) in Figure 2). In its worst-case scenario, Reducer has a running time comparable to AALTAF over the non-reduced specification while, in the best-case scenario, it has a running time inferior or comparable to AALTAF over the rewritten specification. This ensures that running our solver as a pre-processing mechanism can benefit existing verified temporal artificial intelligence algorithms.

Given these experiments, such tools did not support our suggested rewriting rules. Otherwise, the tasks' running time on the original specification $\Phi_i^{c,g}$ would have been comparable to the one for $\Phi_i^{\prime c}$ plus the running time for Reducer. Since experimental evidence suggests that the running time for $\Phi_i^{c,g}$ is always greater than the one for $\Phi_i^{\prime c}$, we have thus empirically demonstrated both the novelty and necessity of these rewriting rules in the aforementioned tools. Figure 3 shows the benchmarks for the formal verification task running on KnoBAB. This plot was separated from the rest to improve the plot's readability. Even in this scenario, the formal verification task over the reduced specification comes, in the worst case scenario, with a running time comparable to the one over the original specification while, in the best case scenario, we have a speed-up of seven orders of magnitude, as all the tasks requiring the access to the ActivityTable are rewritten into clauses that can leverage the sole CountingTable. This also remarks that the query plan minimisation strategy cannot satisfactorily outperform any declarative specification pre-processing strategy, leading to a resulting reduced specification, as its associated running time would have had a comparable running time otherwise.

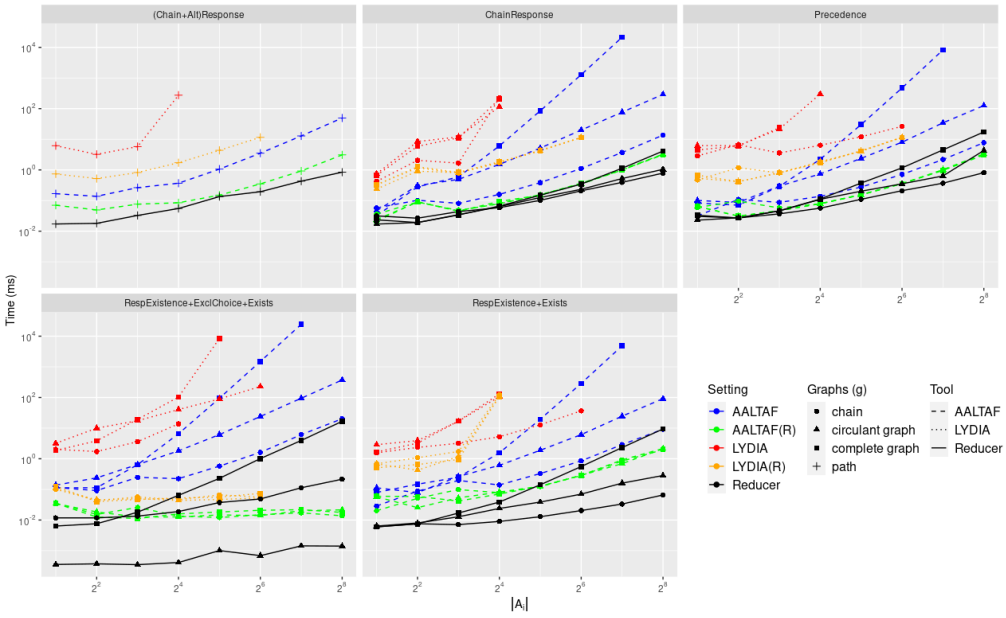

Figure 2. Comparing the specification reducer's running time with the ones of Lydia and AALTAF running over $\Phi_i^{c,g}$ vs. running over $\Phi_i^{\prime c}$ (LYDIA(R) and AALTAF(R) respectively).

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 March 2024 doi:10.20944/preprints202403.0286.v1

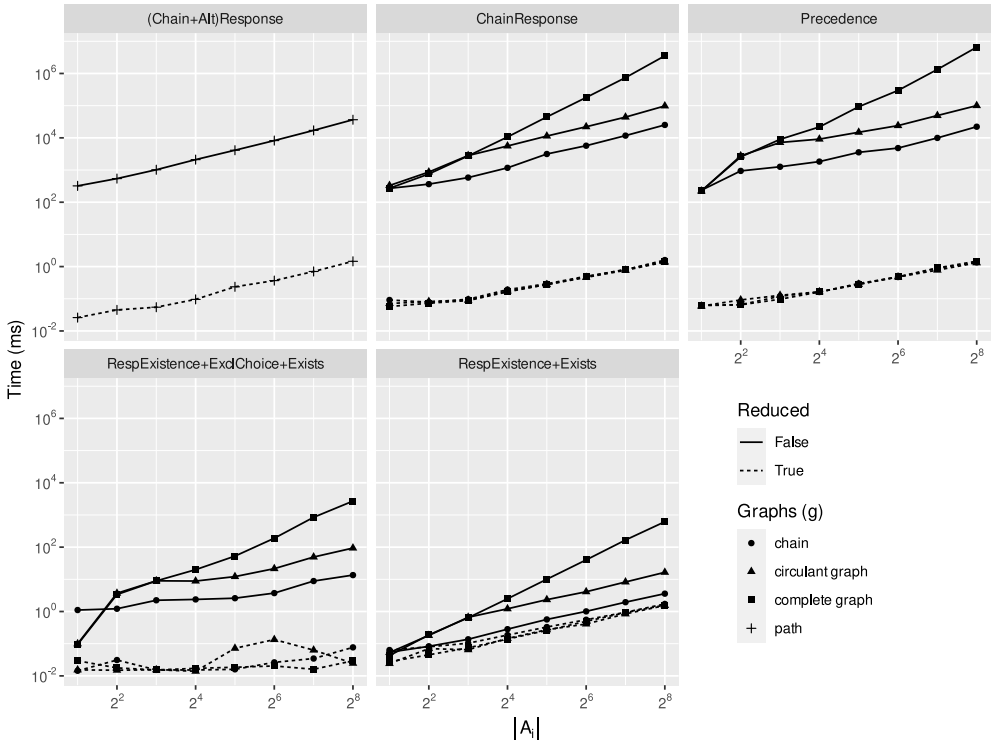

Figure 3. Comparing different running times of KnoBAB over Φ_d (False) vs. Φ'_d (True).

5.2. Rule Rewriting requiring Temporal Non-Simultaneity

We now consider rewriting rules assuming the temporal non-simultaneity axiom. For this, we then need to compare the running time for $(\Phi_i^{c,g})_{\downarrow A_i}$ where the axiom is grounded over the finite set of activity labels A_i to the one resulting from the rewriting process by $\Phi_i^{c\prime}$. In Figure 4, we give running times for $\Phi_i^{c,g}$ for detecting any additional overheads being implied by the instantiation of the axiom over the atoms in A_i. If both Lydia and AALTAF supported LTL_f rewriting rules as per this paper, carrying a task over $(\Phi_i^{c,g})_{\downarrow A_i}$ would have a comparable running time to $\Phi_i^{c\prime}$, while a considerable overhead for computing $(\Phi_i^{c,g})_{\downarrow A_i}$ if compared to $\Phi_i^{c,g}$ denotes that the additional rules coming from the instantiation to the axiom provide a significant computational burden rather than helping in simplifying the specification.

This set of experiments confirms all our previous observations from the previous set-up regarding comparisons between our specification rewriting strategy and the existing verified temporal artificial intelligence tasks. We observe that, in the best-case scenario, such tasks exhibit a running time for $(\Phi_i^{c,g})_{\downarrow A_i}$ comparable to the one for $\Phi_i^{c,g}$ while, in the worst-case scenario, they increase their computational gap proportionally to the increase of the number of clauses. Still, the tasks running over $\Phi_i^{c\prime}$ are consistently outperforming the same tasks running in $(\Phi_i^{c,g})_{\downarrow A_i}$ while also guaranteeing to minimise the out-of-memory exceptions. In the case of AALTAF, these are then completely nullified. Similar considerations can be drawn for running formal specification tasks over the 9 traces sampled from our Cybersecurity scenario, Figure 5 shows that running Φ_{d_i}' gives speed-ups between 3 and 7 orders of magnitude by consistently exploiting the CountingTable instead of the ActivityTable if compared to the running times for the original specification $\Phi_{d_i}^{c,g}$.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 March 2024 doi:10.20944/preprints202403.0286.v1

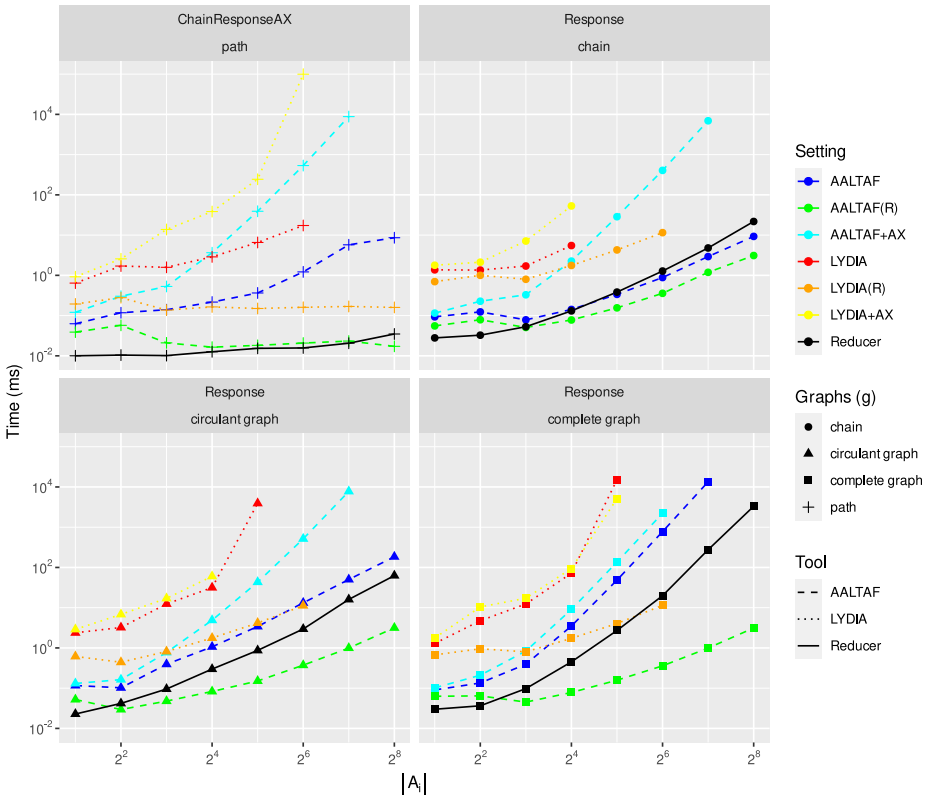

Figure 4. Comparing the specification reducer's running time with the ones of Lydia and AALTAF running over $\Phi_i^{c,g}$ vs. running over $\Phi_i'^{c}$ (LYDIA(R) and AALTAF(R) respectively) and the grounded representation $(\Phi_i^{c,g})_{\downarrow A_i}$ (LYDIA+AX and AALTAF+AX respectively).

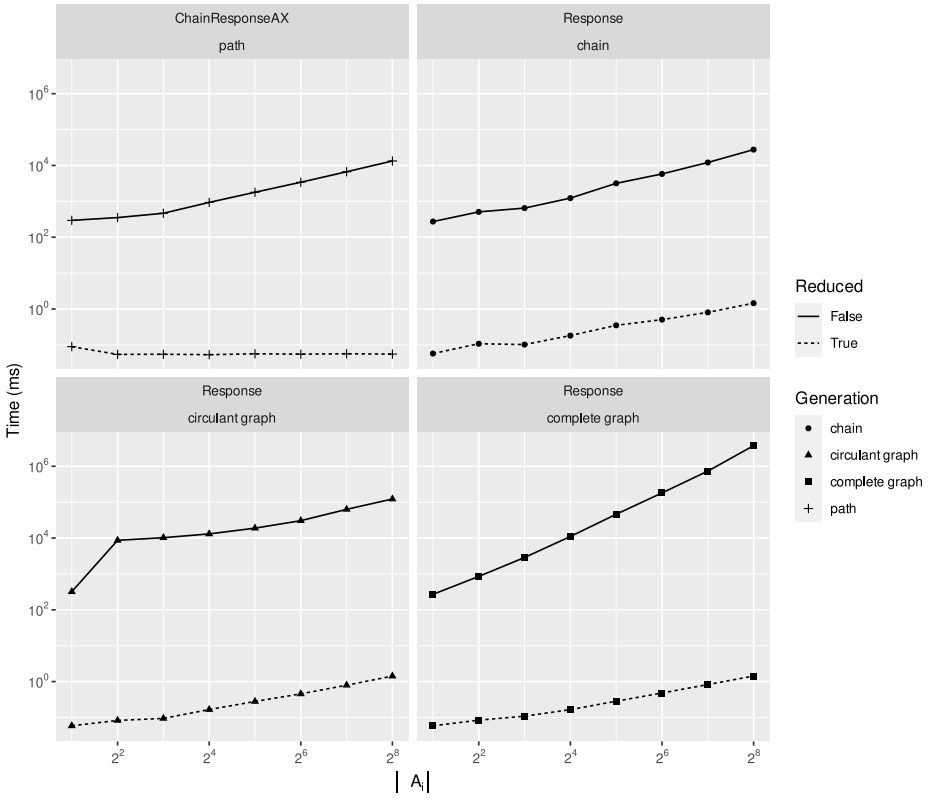

Figure 5. Comparing different running times of KnoBAB over Φ_d (False) vs. Φ_d' (True).

 doi:10.20944/preprints202403.0286.v1

6. Conclusion and Future Works

This paper showed for the first time the existence of a polytime fragment of LTL_f, DECLARE*d* by simply circumscribing the temporal expressiveness of the language. This was possible by observing differences between LTL_f and LTL and, to some extent, assuming mutually exclusive conditions across events. We, therefore, design a scalable SAT-Solver working under equational rewriting, thus rewriting a temporal specification into an equivalent and more tractable rewritten temporal specification Φ'_d. Future works will analyse DECLARE*d*'s time complexity by also considering first-order arithmetic conditions [16]. Experiments on Lydia remarked that the latter does not support adequate rewriting for internal formula minimisation as computing $(\Phi_i^{c,g})_{\downarrow A_i}$ is always slower than $\Phi_i'^{c}$: no algebraic rewriting is considered, as the minimisation steps are only performed while composing the DFA and never at the LTL_f level. Running Lydia on $\Phi_i'^{c}$ significantly improves the running time for temporal formal synthesis. Future works will assess whether the construction of the DFA over the alphabet $\{\{\mathsf{c}\}|\mathsf{c} \in \Sigma\} \cup \{\varnothing\}$ instead of $\wp(\Sigma)$ per Axiom 1, where $\varnothing$ denotes any other atom not in Σ, will boost the algorithm. We will also consider using graph equi-joins in lieu of product construction for conjunction of states, as the former technique already proved to be more efficient than traditional automata composition for DFA generation over Φs [18].

Experiments on AALTAF showed it does not exploit rewriting rules as introduced in this paper: computing $(\Phi_i^{c,g})_{\downarrow A_i}$ is also more costly than $\Phi_i^{c,g}$, and the computation over $\Phi_i'^{c}$ as generated by our solver is always faster than computing either $(\Phi_i^{c,g})_{\downarrow A_i}$ or $\Phi_i^{c,g}$, thus remarking the benefit of our approach in rewriting the formula. Future works will consider generalising the rewiring rules here defined for DECLARE*d*, a fragment of LTL_f, so as to be implemented in any LTL_f tool considering such a rewriting step.

Last, our tool also proved to be beneficial as a specification preprocessing step for optimising formal verification tasks over relational databases, as computing Φ'_d is always faster than computing Φ_d. Future work will consider defining a query plan optimisation strategy not only by computing each shared sub-expression within a given specification once, but also implementing suitable algebraic rewriting rules while supporting Axiom 1.

Funding: This research received no external funding

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

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

Appendix A. Formal Definition

We now provide the formal definition of some operators that were given informally at the end of the Introduction.

Appendix A.1. Graph Operations

$$\textsc{Out}_G(u) := \{\, v \in V_G \mid (u,v) \in E_G \,\}$$

$$\textsc{In}_G(v) := \{\, u \in V_G \mid (u,v) \in E_G \,\}$$

$$\textsc{Deg}_G(v) := |\textsc{Out}_G(v)| + |\textsc{In}_G(v)|$$

$$\text{E}_G^+(\alpha,\beta) := (V_G \cup \{\alpha,\beta\},\ E_G \cup \{(\alpha,\beta)\})$$

$$\text{V}_G^-(\alpha) := (V_G \backslash \{\alpha\}, \{(u,v) \in E | u \neq \alpha \wedge v \neq \alpha\})$$

$$\begin{aligned}&\text{E}_G^-(\alpha,\beta) :=\\ &(V_G \backslash \{u | (\alpha,\beta) \in E_G \wedge (u = \alpha \vee u = \beta) \wedge \textsc{Deg}_G(u) = 1\},\\ &\{(u,v) \in E_G | u \neq \alpha \wedge v \neq \beta\})\end{aligned}$$

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 March 2024 doi:10.20944/preprints202403.0286.v1

Appendix A.2. Multimap Operations

$$\textsc{Del}_f(x) := y \mapsto \begin{cases} \varnothing & y = x \\ f(y) & y \neq x \end{cases}$$

$$\textsc{Put}_f(x, v) := y \mapsto \begin{cases} f(x) \cup \{v\} & y = x \\ f(y) & y \neq x \end{cases}$$

Appendix B. Short-Circuit Rewriting: Time Complexity

The following rewriting lemmas will heavily rely upon the algorithm outlined in [6] for generating an NFA out of a temporal specification Φ. We will use this algorithm to prove that, upon applying this algorithm and minimising the resulting non-deterministic automaton, we will obtain automata equivalently expressing the total absence of the activity labels being represented within the specification. As the size of the resulting graph is exponential over the number of the clauses of the graph, we show that any algorithm using this approach for detecting short-circuit rewriting will take at least ExpTime over the size of the specification Φ_d to detect this. On the other hand, the next section shows a convenient algorithm for detecting those in polytime over the size of the specification. Please also observe that we provide generic proofs for each of the following lemmas, not necessarily requiring that all the activity labels in $A \subseteq \Sigma$ shall be considered mutually exclusive.

Appendix B.1. Automata-Based Strategy

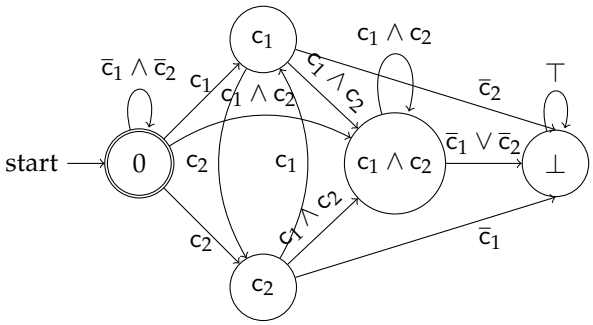

Figure A1. Representation of the NFA associated to $\Phi_d = \{\texttt{ChainResponse}(c_i, c_{i+1 \bmod 2})\}_{i \in \{1,2\}}$ before minimisation for Section 2.2.

Given $A = \{c_1, c_2, \ldots, c_n\} \subseteq \Sigma$, $\Phi' := \bigwedge_{c_i \in A} \Box \neg c_i$ is equivalent to $\Phi := \Box(c_n \Rightarrow \bigcirc c_1) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(c_i \Rightarrow \bigcirc c_{i+1})$ in LTL_f.

Proof. Please observe that the following proof works independently from the non-simultaneity axiom: as we will show, the activation of at least one condition in A will require to verify all the requirements in the specification indefinitely, thus never leading to an acceptance state. So, we now prove this for A not necessarily containing mutually exclusive activity labels.

For this, let us define an atom substitution function ρ as $[c_i \mapsto c_{(i+1 \bmod |\Sigma|)}]_{1 \leq i \leq n}$ indicating the next required atom to follow according to the formula requirements. The generic formula for a ChainResponse temporal short-circuit leads to the NFA that can be constructed as follows: an initial state q_0 being also the sole acceptance state, having a self-loop for $\bigwedge_{a \in A} \neg a$, a sink falsehood state $\bot$ having a self-loop as the only outgoing edge with the universal truth formula $\top$; for each $S \in \wp(A) \backslash \{\varnothing\}$, we generate a new state S associated to the formula $S_\phi = S^+_\phi \wedge S^-_\phi$ with $S^+_\phi = \bigwedge_{a \in S} a$ and $S^-_\phi := \bigwedge_{a \in A \backslash S} \neg a$, describing the actions that were performed to each this system state. For each of these, where we outline the following edges:

 doi:10.20944/preprints202403.0286.v1

- an edge $q_0 \xrightarrow{S_\phi} S$: this requires that, as soon as at least one of the activities in A is run, then we need to follow the requirements associated to the specification;
- an edge $S \xrightarrow{\neg(\rho S^+_\phi)} \bot$: this requires that, as soon as we miss one of the transition conditions requiring that each of the activities being true in S should then immediately move to the immediately following activities ρS, we then violate the specification;
- for each $T \in \wp(A)\backslash\{\emptyset\}$, we define a new edge $S \xrightarrow{T_\phi} T$ if $\rho S \subseteq T$: we connect each of such states not only to the immediately following actions as per ρ, but we also assume that further activation conditions must hold.

Please observe that, as soon as all the activities in A are activated, all are then required to be always true, thus having that the state A will have as outgoing edges its self-loop, prescribing that all the conditions in A must hold, and a transition towards $\bot$ as soon as at least one of these conditions are no more satisfied. After minimising this DFA, we can observe that we obtain q_0, still an initial acceptance state retaining its self-loop, and $\bot$, also retaining its self-loop while having an edge labelled as $\bigvee_{\mathsf{a} \in A} a$ coming from q_0, thus entailing a DFA accepting only the traces where none of the atoms in A. Thus, we proved the correctness of our reduction into a conjunction of DECLARE*d* absences for each activity label in A.

As the number of states is in $O(|\wp(A)|)$, the generation of the automaton will take at least exponential time over the size of the ChainResponse short-circuit, also corresponding to the size of $A \subseteq \Sigma$. □

Lemma A1. *Given* $A = \{ c_1, c_2, \ldots, c_n \} \subseteq \Sigma$ *with* $|A| > 2$, $\Phi' := \bigwedge_{c_i \in A} \Box\neg c_i$ *is equivalent to* $\Phi := \left[\Box(c_n \Rightarrow \bigcirc\Diamond c_1) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(c_i \Rightarrow \bigcirc\Diamond c_{i+1})\right]$ *in* LTL$_f$.

 doi:10.20944/preprints202403.0286.v1

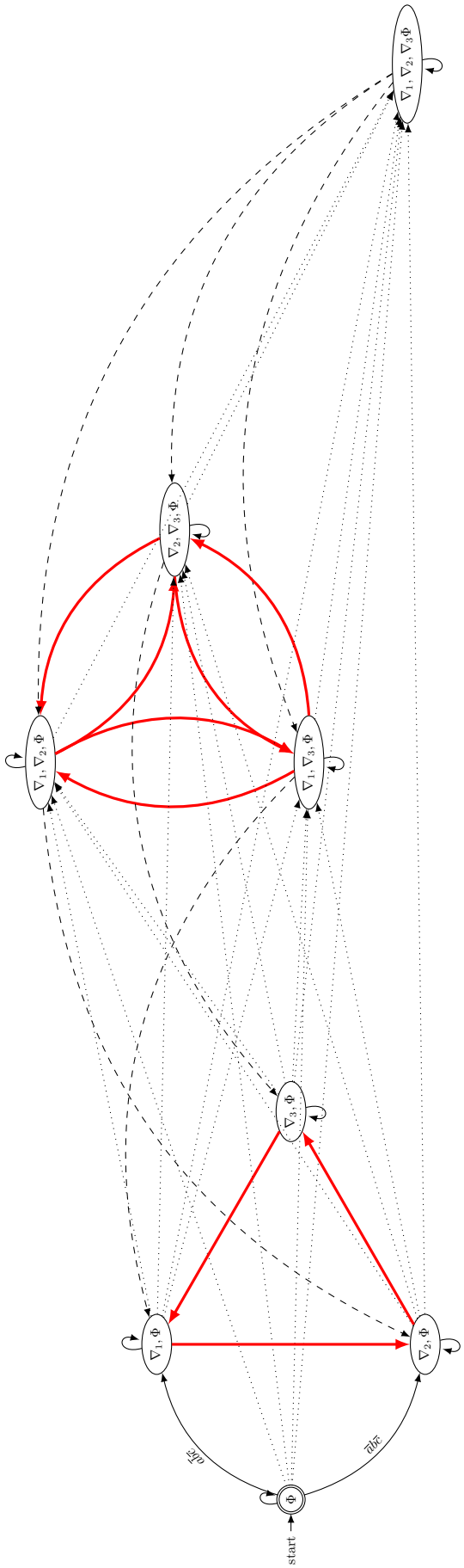

Figure A2. Representation of the NFA associated to $\Phi = \bigwedge_{1 \leq i \leq 3} \Box(c_i \Rightarrow \bigcirc\Diamond c_{i+1 \bmod 3})$ before minimisation for Lemma A1.

 doi:10.20944/preprints202403.0286.v1

Proof. Differently from the previous proof, where the NFA automaton could have been greatly simplified due to the involvement of $\bigcirc$ within the construction, the proof for this lemma needs to be handled with greater care. Before starting, we remind the reader of the special temporal properties holding in LTL_f: $\Box\phi = \phi \wedge \bigcirc\Box\phi$, $\Box(\phi \wedge \phi') = \Box\phi \wedge \Box\phi'$, and $\Diamond\phi = \phi \vee \bigcirc\Diamond\phi$.

The activation of a i-th clause at any state S while constructing the NFA by $c_i \in A$ "generates" the corresponding target condition to be met $\nabla_i := \Diamond(c_{(i+1 \bmod |A|)})$ expressed as $\nabla_i \equiv (c_{(i+1 \bmod |A|)} \vee \bigcirc\nabla_i)$ by the special temporal property of the eventuality operator. By running [6], we also generate $2^{|A|}$ states, where the sole initial and acceptance state associated to Φ, and the other states S are associated to the formulæ in $\mathcal{S} = \{S' \cup \{\Phi\} | S' \in \wp(\{\nabla_i | \mathsf{c}_i \in A\}) \backslash \{\emptyset\}\}$, none of which is an accepting state. No explicit falsehood sink state is present, as the invalidation of one of the ∇_i will actually require never having an event $\mathsf{c}_{i+1 \bmod |A|}$, for which we would still transit across non-accepting states in $\mathcal{S}$ indefinitely. We then identify the following transitions among such states:

1. $\Phi \xrightarrow{\bigwedge_{\mathsf{c}_i \in A} \neg \mathsf{c}_i} \Phi$: if none of the clauses is activated, we trigger no activation condition c_i leading into a ∇_i target requirement to be met; we, therefore, persist on the same initial accepting state.
2. $\Phi \xrightarrow{\bigwedge_{\nabla_i \in S} \mathsf{c}_i \wedge \bigwedge_{\nabla_j \notin S} \neg \mathsf{c}_j} S$ for each $S \in \mathcal{S}$: when at least one activity label in Φ activates one clause, we transit to a state representing the activation of the specific clause.
3. $S \xrightarrow{F} S$ for each $S \in \mathcal{S}$: for remaining in the same state, we require that no other condition c_i not appearing as $\nabla_i \in S$ shall be activated, otherwise, we will be transiting towards another state. Furthermore, we require that the activation of any $c_{i+1 \bmod |A|}$ for which both $\nabla_{i+1 \bmod |A|}$ and ∇_i appear in S should also require the activation of the activity c_i as, otherwise, we will reduce the number of ∇_i, thus ending into a state containing a subset of activated conditions: this is enforced by the construction of Φ. Given this, we obtain the following formulation for F:

$$\bigwedge_{\nabla_{i+1 \bmod |A|}, \nabla_i \in S} (\mathsf{c}_{i+1} \Rightarrow \mathsf{c}_i) \wedge \bigwedge_{\nabla_j \notin S} \neg \mathsf{c}_j$$

   Please observe that we do not require that $\bigwedge_{\nabla_{i+1 \bmod |A|} \notin S, \nabla_i \in S}(\mathsf{c}_i)$ should also hold, as the eventuality of the condition for the target condition does not strictly require that such condition must immediately hold in a subsequent step, so either the possibility of c_i being still activated or its opposite are both considered.
4. $\{\nabla_j\}_{j \in S'} \cup \{\Phi\} \xrightarrow{F} \{\nabla_j\}_{j \in S''} \cup \{\Phi\}$ with $S' \subset S''$: we need to ensure that the conditions appearing only in S'' are newly activated, while the ones being active in S' shall be kept active in S''; given this, we obtain the following F:

$$\bigwedge_{\substack{i \in S'' \backslash S' \cup \\ \{j | (j=1 \wedge |A| \in S') \vee (j \neq 1 \wedge j-1 \in S')\}}} c_i \wedge \left(\bigwedge_{i \in \{1,\ldots,|A|\} \backslash (S \cup S')} \neg c_i \right)$$

5. $\{\nabla_j\}_{j \in S'} \cup \{\Phi\} \xrightarrow{F} \{\nabla_j\}_{j \in S''} \cup \{\Phi\}$ with $S' \supset S''$: this transition can only occur if, by attempting to consume ∇_i with $i \in S'$ by executing an action $\mathsf{c}_{i+1 \bmod |A|}$, this leads to generating $\nabla_{i+1 \bmod |A|}$ appearing in $S'' \cap S'$ or, otherwise, we would have transited towards another state. This then provides a restriction of the states towards which we can transit and the states that can move backwards. F can be then defined as:

$$\bigwedge_{\substack{i \in S', \\ i+1 \bmod |A| \in S' \cap S''}} \mathsf{c}_{i+1 \bmod |A|} \wedge \bigwedge_{i \in \{1,\ldots,|A|\} \backslash (S'' \cup S')} \neg c_i$$

6. $\{\nabla_j\}_{j \in S'} \cup \{\Phi\} \xrightarrow{F} \{\nabla_j\}_{j \in S''} \cup \{\Phi\}$ with $S' \neq S''$: otherwise, we can transit $\{\nabla_j\}_{j \in S'} \cup \{\Phi\} \xrightarrow{F} \{\nabla_j\}_{j \in S''} \cup \{\Phi\}$ with $S' \neq S''$ at the following conditions: for each $j \in S'$, either $(j+1 \bmod$

 doi:10.20944/preprints202403.0286.v1

$|A|) \in S''$ or $j \in S''$. If those conditions had not been met, given the previous considerations, it would not have been possible to transit exactly between these two states.

As we can observe from the former transitions, once one trace event satisfies a condition in A, we will always navigate towards states in $\mathcal{S}$ without ever having the possibility of going back to the initial and sole accepting state Φ, as we must perennially guarantee that the conditions in A shall be eventually satisfied in turns, thus entailing that no finite trace will ever satisfy such conditions. As the number of states required for generating these formulae is therefore exponential in the size of both Φ and A as $|\Phi| = |A|$ by construction, by assuming that the most efficient algorithm for generating such automaton from the LTL_f specification will take at least a time comparable to the size of the graph without any additional computation overhead, we therefore have that such algorithm will take at least an exponential time on the size of the specification, thus in $o\left(2^{|A|}\right)$.

Similarly to the previous lemma, even in this scenario, the minimisation of such automaton will lead to one being equivalent to the one predicate the absence of all the activities in A.

□

 doi:10.20944/preprints202403.0286.v1

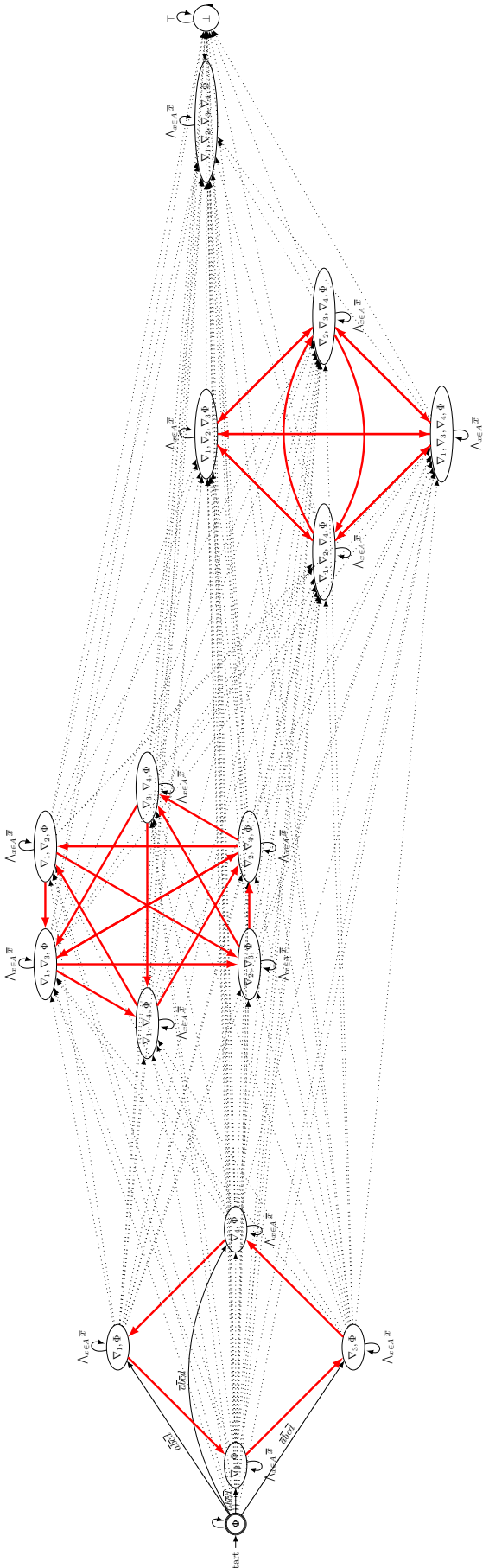

Figure A3. Representation of the NFA associated to $\Phi = \{\texttt{AltResponse}(c_i, c_{i+1 \bmod 4})\}_{1 \leq i \leq 4}$ before minimisation for Lemma A2.

 doi:10.20944/preprints202403.0286.v1

Given $A = \{\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_n\} \subseteq \Sigma$ with $|A| = n > 2$, $\Phi' := \bigwedge_{\mathsf{c}_i \in A} \Box\neg \mathsf{c}_i$ is equivalent to $\Phi := \left[\Box(\mathsf{c}_n \Rightarrow \Diamond \mathsf{c}_1) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(\mathsf{c}_i \Rightarrow \Diamond \mathsf{c}_{i+1})\right]_{\downarrow \Sigma}$ in $\mathrm{LTL_f}$.

Proof. It derives as a corollary from Lemma A1, which holds independently from the non-simultaneity axiom. □

Lemma A2. *Given* $A = \{\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_n\} \subseteq \Sigma$, $\Phi' := \bigwedge_{\mathsf{c}_i \in A} \Box\neg \mathsf{c}_i$ *is equivalent to* $\Phi := \Box(\mathsf{c}_n \Rightarrow \bigcirc(\neg \mathsf{c}_n \mathcal{U} \mathsf{c}_1)) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(\mathsf{c}_i \Rightarrow \bigcirc(\neg \mathsf{c}_i \mathcal{U} \mathsf{c}_{i+1}))$ *in* LTL_f.

Proof. We proceed similarly to the previous lemma where now, due to the adoption of the Until operator, we change the definition of ∇_i per its special property as follows:

$$\nabla_i := \mathsf{c}_{i+1 \bmod |A|} \vee (\neg \mathsf{c}_i \wedge \neg \mathsf{c}_{i+1 \bmod |A|} \wedge \bigcirc \nabla_i)$$

prescribing that in any subsequent step c_i can never occur until the first occurrence of $\mathsf{c}_{i+1 \bmod |A|}$. Furthermore, similarly to the ChainResponse case, we add a falsehood sink state, towards each state will transit upon violation of the conditions prohibited by ∇_i. In fact, this lemma restricts the expected behaviour in the former lemma, as we now prescribe that c_i, once activating a clause thus adding ∇_i to a state, cannot occur before the occurrence of $\mathsf{c}_{i+1 \bmod |A|}$ or, otherwise, we have to transit towards a never-accepting sink falsehood state. Consequently, except for the sink falsehood state, I could loop over any other state only if none of the activities in A are considered. We now stress the main differences from the definition of the transition functions if compared with the previous lemma:

1. $\Phi \xrightarrow{\bigwedge_{\mathsf{c}_i \in A} \neg \mathsf{c}_i} \Phi$: Same as per the previous lemma.
2. $\Phi \xrightarrow{\bigwedge_{\nabla_i \in S} \mathsf{c}_i \wedge \bigwedge_{\nabla_j \notin S} \neg \mathsf{c}_j} S$ for each $S \in \mathcal{S}$: Same as per the previous lemma.
3. $S \xrightarrow{F} S$ for each $S \in \mathcal{S}$: As per previous observation, this is now changed to $F = \bigwedge_{\mathsf{c}_i \in A} \neg \mathsf{c}_i$, as performing none of the states that are recorded in A is the only possible way not to transit into any other state.
4. $\{\nabla_j\}_{j \in S'} \cup \{\Phi\} \xrightarrow{F} \{\nabla_j\}_{j \in S''} \cup \{\Phi\}$ with $S' \subset S''$: Same as per previous lemma.
5. $\{\nabla_j\}_{j \in S'} \cup \{\Phi\} \xrightarrow{F} \{\nabla_j\}_{j \in S''} \cup \{\Phi\}$ with $S' \supset S''$. This type of transition never occurs similarly to Section 2.2. Without any loss of generality, let us assume that $S' = \{i, j, k\}$ and $S'' = \{i, j\}$ as in the previous lemma: allowing such transition would require to have an event abiding by $\mathsf{c}_{k+1 \bmod |A|}$ for which either $i = k + 1 \bmod |A|$ or $j = k + 1 \bmod |A|$. The only possible way to make this admissible is to make also ∇_i (or ∇_j) move with a corresponding $\mathsf{c}_{i+1 \bmod |A|}$ (or $\mathsf{c}_{j+1 \bmod |A|}$) action; still, this would have contradicted the assumption that i and j are not moving, and therefore this action would either violate ∇_i or ∇_j, which is then impossible. Therefore, executing any of the activation conditions c_i for $i \in S'$ being explicitly prohibited by the corresponding ∇_i will just move the current source state of interest towards the sink falsehood state.
6. $\{\nabla_j\}_{j \in S'} \cup \{\Phi\} \xrightarrow{F} \{\nabla_j\}_{j \in S''} \cup \{\Phi\}$ with $S' \neq S''$: otherwise, as observed in the previous point, we can move towards a new state by either consuming a $\mathsf{c}_{i+1 \bmod |A|}$ with $i \in S'$ leading to an $i + 1 \bmod |A| \in S''$, or by ensuring a $\mathsf{c}_{j+1 \bmod |A|}$ with $j + 1 \bmod |A| \in S'' \backslash S'$ and $j \notin S'$ for not violating an already-activated condition. Overall, we can observe that this leads to never transiting from a state containing more activation conditions towards one containing less than those, at any rate.

Under all the remaining circumstances, we transit from S towards the falsehood sink state. Similarly, as in the previous construction, we can observe that any algorithm generating such a graph before minimisation will take an exponential time on the size of both the specification Φ and A. □

 doi:10.20944/preprints202403.0286.v1

Appendix B.2. Proposed Methodology

In this section, we show an efficient algorithm for detecting short-circuit rewritings within the DECLARE*d* fragment in polytime over the size of the specification.

Corollary A1. *We can rewrite Φ_d containing a* `ChainResponse` *short-circuit in polytime on the size of Φ_d.*

Proof. Given the construction sketched in §2.2, the best case scenario for Φ constitutes in Φ containing exactly one single `ChainResponse` circuit, for which we obtain a graph $G^{\mathtt{cr}}$ representing itself a cycle of size $|\Phi|$. By adopting a DFS visit for detecting a cycle, we take $O(V+E)$ to recognize the whole graph $G^{\mathtt{cr}}$ as a cycle.

In the worst-case scenario, Φ contains a conjunction of clauses $\bigwedge_{\substack{\mathsf{a},\mathsf{c}\in\Sigma\\ \mathsf{a}\neq\mathsf{c}}} \mathtt{ChainResponse}(\mathsf{a},\mathsf{c})$ leading to a fully connected graph $G^{\mathtt{cr}}$. Within this scenario, the worst-case scenario for detecting a cycle is detecting a cycle of size 2 after fully visiting $G^{\mathtt{cr}}$. After doing so, we remove the two nodes from $G^{\mathtt{cr}}$ and repeat the visit over such a reduced graph. If we always assume to detect cycles of size 2 for each visit, we will end up running $\frac{|V^{\mathtt{cr}}|}{2}$ visits of the graph, and the overall time complexity becomes $\sum_{i=0}^{|V^{\mathtt{cr}}|/2}(|V^{\mathtt{cr}}|-2i+(|V^{\mathtt{cr}}|-2i)^2) \in O(|V^{\mathtt{cr}}|^3)$.
□

As a further corollary from this, we immediately deduce that our strategy from §3.4 is far way more efficient than generating a DFA associated with a formula for then minimising it as in Section 2.2, as in the best case scenario we still have to generate an exponential number of states in the size of A, while in our proposed approach we do not. This is possible as our current envisioned approach assumes Section 2.2 to hold without needing to go through the aforementioned exponential construction algorithm.

Corollary A2. *We can rewrite Φ_d containing a* `Response` *short-circuit in polytime on the size of Φ_d.*

Proof. This can be considered a further corollary of Corollary A1, as both the graph visit and construction phase are completely independent of the nature of the clause, which is completely neglected and sketched in terms of mutual dependencies across activity labels through a dependency graph. Similar conclusions then hold, also in terms of time complexity for the graph visit. □

Corollary A3. *We can rewrite Φ_d containing an* `AltResponse` *short-circuit in polytime on the size of Φ_d.*

Proof. As per Corollary A2, the goal is closed similarly to Lemma A1 due to the same way the graph is constructed independently from its associated $\mathrm{LTL_f}$ semantics. □

Appendix C. Formal Verification Speedup

While the previous section clarified that, by assuming a templated temporal language, we can rewrite temporal short-circuits in polynomial time, this section remarks the benefits of the aforementioned rewriting within formal verification tasks, as all aforementioned state-of-the-art algorithms in these regard do not contemplate clause rewriting. This will then provide a theoretical validation over the empirical results provided in the main paper.

While considering the computational complexity associated with formal verification tasks, we assume the KnoBAB computational model, where the entire set of traces within a log is considered, and each trace is not necessarily computed one at a time. Therefore, we interpret the $\mathrm{LTL_f}$ computation for each trace in the log regarding the associated $\mathtt{xtLTL_f}$ operators in KnoBAB [2].

Given $A = \{\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_n\} \subseteq \Sigma$, computing $\Phi' := \bigwedge_{\mathsf{c}_i \in A} \Box\neg\mathsf{c}_i$ in lieu of $\Phi := \Box(\mathsf{c}_n \Rightarrow \bigcirc\mathsf{c}_1) \wedge \bigwedge_{\substack{i\in\mathbb{N}\\ 1\leq i<n}} \Box(\mathsf{c}_i \Rightarrow \bigcirc\mathsf{c}_{i+1})$ always leads to a positive average speed-up.

 doi:10.20944/preprints202403.0286.v1

Proof. Given $\|\mathfrak{S}\|$ the number of all the events in the log obtained by summing up all the trace lengths in $\mathfrak{S}$, $\|\mathfrak{S}\|$ denotes the number of all the events in the entire log in all traces. We also denote #a as the number of all the events in $\mathfrak{S}$ having "a" as an activity label.

Using KnoBAB as a computational model for computing LTL_f via $\texttt{xtLTL}_\text{f}$, we can determine $\sigma, t \vDash \bigcirc \mathsf{c}_i$ in $\#\mathsf{c}_i$ time. As the number of all the events not being c_i in $\mathfrak{S}$ is $\|\mathfrak{S}\| - \#\mathsf{c}_i$, computing $\sigma, t \vDash \neg \mathsf{c}_i$ requires $\|\mathfrak{S}\| - \#\mathsf{c}_i$ time. Under the KnoBAB intermediate result representation assumption, all the intermediate results from $\texttt{xtLTL}_\text{f}$ expressions are pre-sorted by trace id and temporal position, we can compute either $\sigma, t \vDash \varphi \wedge \varphi'$ or $\sigma, t \vDash \varphi \wedge \varphi'$ in at most $|\varphi| + |\varphi'|$ time. Per each clause occurring in the specification, we are interested in computing $\phi = \mathsf{c}_i \Rightarrow \bigcirc \mathsf{c}_{i+1 \bmod |A|}$. We can then ϕ as $(\neg \mathsf{c}_i) \vee (\mathsf{c}_i \wedge \bigcirc \mathsf{c}_{i+1 \bmod |A|})$: as we observe that the next $\bigcirc\varphi$ operator provides a linear scan of the input operator, this computation can be carried out in an overall $\|\mathfrak{S}\| - \cancel{\#\mathsf{c}_i + \#\mathsf{c}_i} + 2\#\mathsf{c}_{i+1 \bmod |A|}$ generating, in the worst case scenario, data in the size of $\|\mathfrak{S}\|$. Furthermore, we observe that computing this for each clause in Φ leads to a total time of $|A|\|\mathfrak{S}\| + 2\|\mathfrak{S}\|$. Therefore, computing $\sigma, t \vDash \Box\phi$ as an $\texttt{xtLTL}_\text{f}$ operator over all events satisfying $\sigma, t \vDash \phi$ will take at most $\|\mathfrak{S}\| \log \|\mathfrak{S}\|$ per clause, thus adding up to $|A|\|\mathfrak{S}\| \log \|\mathfrak{S}\|$. Furthermore, the cost of computing the conjunction among the result of all such clauses adds up to $|A|\|\mathfrak{S}\|$ in its worst-case scenario.

On the other hand, computing each $\Box\neg \mathsf{c}_i$ in the resulting specification Φ' requires KnoBAB to check in the counting table for each trace that c_i occurs zero times with a linear scan; thus, Φ' can be computed in $2|A||\mathfrak{S}| = 2le$ time, as we also need to encompass the time required for computing the disjunction between all the data being computed per traces.

For $l := \|\mathfrak{S}\| > 0$ and $e := |A| > 0$, we therefore compute the positive speed-up by expressing it as the ratio between the time complexity for computing a formal verification task composed by ChainRespons-es leading to a temporal short-circuit and the one for computing an equivalent set of absence clauses after short-circuit rewriting. From this ratio, we observe that the computation of the rewritten specification leads to a positive speed-up over the former, as the the resulting value is always greater or equal than zero:

$$\frac{le(1 + \frac{2}{e}) + le \log l + le}{2le} \geq 1 \Leftrightarrow \frac{2}{e} + \log l \geq 0$$

□

Given $A = \{\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_n\} \subseteq \Sigma$ with $|A| = n > 2$, computing $\Phi' := \bigwedge_{\mathsf{c}_i \in A} \Box\neg \mathsf{c}_i$ in lieu of $\Phi := \left[\Box(\mathsf{c}_n \Rightarrow \Diamond \mathsf{c}_1) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(\mathsf{c}_i \Rightarrow \Diamond \mathsf{c}_{i+1})\right]_{\downarrow \Sigma}$ always leads to a positive average speed-up.

Proof. We use the proof for Section 2.2 as a calque for this other speed-up analysis, where we only have to change ϕ to $\mathsf{c}_i \Rightarrow \Diamond \mathsf{c}_{i+1 \bmod |A|}$ thus focussing our analysis on $\sigma, t \vDash \phi$: this can be then equivalently expressed as $\sigma, t \vDash \neg \mathsf{c}_i \vee (\mathsf{c}_i \wedge \Diamond \mathsf{c}_{i+1 \bmod |A|})$, where $\varphi \wedge \Diamond\varphi'$ is computed using a specific derived operator taking $|\varphi||\varphi'| \log |\varphi'|$ time. For each clause, this leads to $\|\mathfrak{S}\| - \#\mathsf{c}_i + \#\mathsf{c}_i \cdot$ $\#\mathsf{c}_{i+1 \bmod |A|} \log \#\mathsf{c}_{i+1 \bmod |A|}$ time per clause computing ϕ returning, in the worst case scenario, $\|\mathfrak{S}\|$ events; all the clauses take at most $\|\mathfrak{S}|(|A| - 1) + \|\mathfrak{S}\| k \log k$ time to compute this expression by assuming $k \simeq \#\mathsf{c}_1 \simeq \cdots \simeq \#\mathsf{c}_{|A|} \simeq {}^{\|\mathfrak{S}\|}/_{|A|}$. As in the previous Lemma, the computation of the associated $\Box$ operator for each $\Box\phi$ per each of the $|A|$ clauses will take at most $|A|\|\mathfrak{S}\| \log \|\mathfrak{S}\|$ time. The time for computing Φ' is also $2le$ as per the previous Lemma.

Similarly to the previous lemma, we then compute the ratio between the time complexity for formal verification over AltResponse-es leading to a temporal short-circuit and the one over the equivaletly rewritten specification. As the ratio between the former and the latter is strictly greater than zero, the rewriting leads to a speed-up at least proportional to the size of the log and traces for $l := \|\mathfrak{S}\| > 0$ and $e := |A| > 0$:

$$\frac{l(e-1) + lk \log k + le \log l + le}{2le} = \frac{1 - \frac{1}{e} + \frac{k \log k}{e} + \log l + 1}{2} \geq 1$$

 doi:10.20944/preprints202403.0286.v1

Thus entailing an always positive speed-up for formal verification tasks for sufficiently large $k, l, e \to \infty$. □

Lemma A3. *Given* $A = \{ c_1, c_2, \ldots, c_n \} \subseteq \Sigma$, *computing* $\Phi' := \bigwedge_{c_i \in A} \Box \neg c_i$ *in lieu of* $\Phi := \Box(c_n \Rightarrow \bigcirc(\neg c_n \mathcal{U} c_1)) \wedge \bigwedge_{\substack{i \in \mathbb{N} \\ 1 \leq i < n}} \Box(c_i \Rightarrow \bigcirc(\neg c_i \mathcal{U} c_{i+1}))$ *always leads to a positive average speed-up.*

Proof. We can exploit a similar formulation as per Section 2.2 and Section 2.3, where we now only need to consider that $\neg \mathsf{c}_i \mathcal{U} \mathsf{c}_{i+1 \bmod |A|}$ will come at the cost of $(\|\mathfrak{S}\| - \#\mathsf{c}_i)^2 \#\mathsf{c}_{i+1 \bmod |A|}$. This computation will generate at most data in the size of $\|\mathfrak{S}\| - \#\mathsf{c}_i$, as the latter data within the first operand of the Until will also contain the events satisfying the condition in the second argument, thus leading to an additional $\|\mathfrak{S}\| - \#\mathsf{c}_i$ cost for computing the associated $\bigcirc$ operator. As the previous clauses, in the worst case scenario each clause will take $(\|\mathfrak{S}\| - \#\mathsf{c}_i)^2 \#\mathsf{c}_{i+1 \bmod |A|} + \|\mathfrak{S}\| - \#\mathsf{c}_i$ to compute and, when considering all the clauses so far, this adds up to $\|\mathfrak{S}\|^3 \left(1 - \frac{\|\mathfrak{S}\|}{|A|}\right)^2 + \|\mathfrak{S}\|(|A| - 1)$ for each clause by considering $\#\mathsf{c}_i \simeq \#\mathsf{c}_{i+1 \bmod |A|} \simeq {}^{\|\mathfrak{S}\|}/_{|A|}$.

So, as this increases the overall time complexity for each clause, we also obtain as per Section 2.3 an always positive speed-up. □

Appendix D. DECLARE*d* SAT

This section remarks that the rule rewriting strategy outlined in this paper can be used as a SAT-solver for DECLARE*d*. After showing the correctness of this procedure (§D.1), we finally show that the underlying time complexity of the overall procedure is in polynomial time (§D.2).

Appendix D.1. Correctness

Lemma A4. *If the specification is a tautology, then the formula is completely rewritten into* $\top$.

Proof. Algorithm 2 is the only part detecting trivially-holding conditions: this occurs all the time that an edge is not added in $G^\star$ for a clause with the template $\star$ while invoking neither Abs nor Ex, as these would otherwise trigger the generation of `Absence` and `Exists` clauses at the end of the computation. In fact, any further clause rewriting resulting from applying the rewriting rules as described in §3.3 always invokes one of the two former functions, thus not necessarily guaranteeing that an empty specification will be returned. Therefore, the aforementioned algorithm is the only point in the code where the non-insertion of clauses jointly with the lack of the invocation of Abs/Ex might lead to the generation of an empty specification. As the clauses that were not inserted in the specification were actually trivially true, if we obtain a specification with empty graphs and an empty F, we infer that the overall specification is also trivially true. Therefore, in this situation, we return $\top$ as a resulting Φ' for Φ'_d. □

Lemma A5. *If the specification is unsatisfiable, then the computation abruptedly terminates while returning* $\bot$

Proof. We observe that we detect the specification as unsatisfiable only under three circumstances, whether *(i)* $\exists x \in \mathrm{dom}(F).|F(x)| = 2$, thus implying by algorithmic construction that the absurd condition $\texttt{Absence}(\mathsf{x}) \wedge \texttt{Exists}(\mathsf{x})$ should hold, *(ii)* whether we trigger a rewriting rule leading to $\bot$, and *(iii)* at loading time. This proof follows from the assumption that no further inconsistency can be detected from the described rules and algorithms.

The first scenario requires checking, each time a new $\texttt{Absence}(\mathsf{x})$ or $\texttt{Exists}(\mathsf{x})$ clause is generated, to always check for *(i)* while ensuring that the detection of *(i)* is propagated through the function call chain. The second condition requires iterating over all the edges and correctly detecting the conditions leading to $\bot$ from Figure 1 while applying the rewriting rules as per §3.3: the return of $\bot$ in this occasion is described in this section. The third scenario is as described in Algorithm 2. We close the two last sub-goals as we covered all the possible cases leading to a direct inconsistency.

 doi:10.20944/preprints202403.0286.v1

This leads to then proving the remaining first sub-goal. First, we can prove that detecting an inconsistent specification is propagated backwards given the function call stack. Let us now focus on the sub-routines in Algorithm 1: we observe that ABS/EX returns **false** when a specification is being detected as inconsistent, while all the other sub-routines in the same Algorithm immediately return **false** upon calling any of the other functions when at least one call detects such an inconsistency. As the generation of $\texttt{Absence}(\texttt{x})$ or $\texttt{Exists}(\texttt{x})$ is also achieved by calling the previous functions, we always ensure that any potential inconsistency is detected. Furthermore, the code guarantees that any call to REDUCE and CLEAR returning **false** immediately returns $\bot$: this in fact holds as the respective functions guarantee that explicit application of the rewriting rules involving each clause that we know per $\texttt{Absence}(x)$ that will be never activated, thus guaranteeing an a posteriori rewriting of the specification even after scanning all of the clauses associated to the same template as per §3.3. For $\texttt{NotCoExistence}$, we also guarnatee that this detection occurs by directly calling $\text{REDUCE}^{\texttt{ch}}$ instead, thus also leading to the generation of $\texttt{Exists}$ clauses. Dually, this also holds for the generation of new $\texttt{Exists}(x)$ rules, which are then leading to the invocation of the $\texttt{Expand}^{\texttt{re}}(x)$ sub-routine which, in turn, is also checking for $\text{EX}(x)$. Thus we can observe that our algorithm guarantees that all of the rules are properly expanded as well as always updating on the current state for the existence/absence of inconsistencies, thus leading to correctly detecting an inconsistency if any. As the rewriting rules provide all the possible combinations for which the absence or the presence of specific activity labels might generate further activation or target conditions, we immediately ensure to return an inconsistent specification upon detection given the rewriting rules completely describing the language. □

Last, we also provided some unit tests for ensuring, to the best of our knowledge, the correctness of the implemented solution: https://anonymous.4open.science/r/DECLAREd-B1BF/tests.cpp.

Appendix D.2. Convergence in PolyTime(Φ_d)

We now prove the lemmas dealing with DECLARE*d*'s decidability and polynomial time complexity for each sub-routine within our equational rewriting algorithm.

Lemma A6. *The sub-routines in Algorithm 1 always terminate in polynomial time.*

Proof. We now analyse each declared sub-routine. Before doing so, we observe that no rule generates activity labels that are not originally considered within the original specification Φ_d, thus ensuring that the computation will always terminate. Given Σ the set of all the activity labels occurring in the original specification Φ_d, we can only have $|\Sigma|$ distinct calls to these functions and, given that no rule in both Figure 1 and temporal short-circuit rewriting generates novel activity labels not occurring in the formula, we are never expecting having $|\text{dom}(F)| > |\Sigma|$, thus ensuring the non-divergence of our computation. This assumption (A1) is then transferred to each call of the following functions:

EX(x): This function takes note that the specification requires, at some point of the rewriting, that x shall exist anytime in a trace; this mainly updates a hashmap F and immediately returns a boolean value determining whether this is also associated to an absence, for which then F will be associated to two distinct values instead of one. Therefore, this trivially terminates in $O(1)$.

ABS(x): This function is the exact dual of the previous one, as it predicates the absence of an event associated with an activity label x. Even in this case, this function always terminates in $O(1)$.

CLEAR(x): This function calls only other functions for removing vertices and edges from a graph associated with a clause template, which functions are non-recursive and trivially terminating. Furthermore, this function only reduces the previously loaded and indexed information, except for the clauses associated with calls to Ex(x) and Abs(x), used to detect inconsistencies within the temporal specification. This is carried out in linear time over the size of the currently-loaded specification.

REDUCE*(x) for $\star \neq \texttt{ch}$: This function mainly describes an iterative backward DFS visit over a $G^{\star}$ graph with $\star \neq \texttt{p}$ using *toremove* as a stack by traversing all the edges in the graph backwards

 doi:10.20944/preprints202403.0286.v1

from x; we avoid in-definitively traversing loops in the graph by remembering which nodes were already visited and popped from the aforementioned stack (*visited*). This call jointly with CLEAR(x) ensures that no clause containing $x \in \Sigma$ as an activity label will be returned in the resulting specification, as this function will remove all the vertices representing the activity label x. Henceforth, even this function does not generate new data jointly with A1. Overall, the algorithm is then guaranteed always to terminate in polynomial time over the size of the specification.

REDUCE$^{\mathtt{p}}$(x): we can draw similar considerations as the previous algorithm, as the main difference is merely in the direction of the graph visit: we are now traversing the edges forward instead than in reverse ($\forall a, b \in \Sigma.\ \mathtt{Precedence}(a,b) \wedge \mathtt{Absence}(a) \equiv \mathtt{Absence}(a) \wedge \mathtt{Absence}(b)$, Line 45 also in Figure 1). Vertices from the $G^{\mathtt{p}}$ are also explicitly covered in this function (L. 32), as this is not considered as part of CLEAR(x); this ensures that this function cannot be called with the same argument x.

EXPAND$^{\mathtt{re}}$(x): Notwithstanding that this function is the dual of the previous, this works similarly: when a new $\mathtt{Exists}(x)$ DECLARE*d* clause is attempted to be generated, we ensure that this will not trigger another rewriting annihilating some RespExistence clauses ($\forall a, b \in \Sigma.\ \mathtt{RespExistence}(a,b) \wedge \mathtt{Exists}(a) \equiv \mathtt{Exists}(a) \wedge \mathtt{Exists}(b)$, Line 45). Similarly to the previous steps, we are not adding information in the graph $G^{\mathtt{re}}$ that we are traversing, rather than removing those, thus ensuring to avoid unnecessary re-computations over the same activity label $x \in \Sigma$. Furthermore, we remove any occurrence of a Choice clause that might be trivialised by the existence of x ($\forall a, b \in \Sigma.\ \mathtt{Choice}(a,b) \wedge (\mathtt{Exists}(a) \vee \mathtt{Exists}(b)) \equiv (\mathtt{Exists}(a) \vee \mathtt{Exists}(b))$, Line 41) as well as checking whether the existence of x might lead to inconsistencies related to the required absence of the label expressed in the target condition at Line 43:

$$\forall a, b \in \Sigma.\ \mathtt{NegSuccession}(a,b) \wedge \mathtt{Exists}(a) \equiv \mathtt{Absence}(b)$$

Similarly to the other sub-routines, this function is gradually reducing the number of the clauses which are potentially rewritten in Exists/Absence: due to (A1), this procedure is also guaranteed to terminate.

REDUCE*(x) for $\star = \mathtt{ch}$: This provides a restriction to the case $\star \neq \mathtt{ch}$ inasmuch as the activity labels in *toremove* are not visited from the stack, rather than being used for calling $\mathtt{Expand}^{\mathtt{re}}$ which, in turn, is also a terminating function. Overall, REDUCE is always terminating independently from $\star$.

Overall, we conclude that each of the sub-routines is guaranteed to terminate in at most polynomial time while also guaranteeing to reduce the information being stored in the graphs associated with each declarative template. □

Lemma A7. *The computation allowing the expansion of some DECLAREd clauses while loading those in the appropriate graphs $G^{\star}$ for each template $\star$ terminates in linear time over the size of the specification (Algorithm 2).*

Proof. First, as the Φ_d set is always finite under the assumption that this algorithm works by loading specification as written in a computer, then this algorithm will always take a finite time to linearly iterate over all the finite set of DECLARE*d* clauses being represented in the specification. Next, each invocation to EX and ABS is guaranteed to terminate in $O(1)$ due to Lemma A6. Furthermore, while graphs are instantiated by adding edges (and therefore the corresponding nodes if missing), we are never traversing those. As per previous considerations, at this stage we also consider expansion rules rewriting some of the given clauses in Φ as other clauses. As this expansion does not trigger any further rewriting rule in Figure 1, we are simply adding new clauses without being stuck in never-ending cycles. This also goes hand in hand with (A1) from the previous lemma, as we generate clauses without inferring new activity labels not in the original specification. Therefore, even this algorithmic step is guaranteed to terminate in most linear time concerning the specification size. □

 doi:10.20944/preprints202403.0286.v1

Corollary A4. *The computation of short-circuit rewriting is guaranteed to terminate in a polytime over the size of the original specification.*

Proof. This can be seen as a further corollary of Corollary A1, Corollary A2, and Corollary A3: as the composition of distinct terminating function calls leads to an overall terminating computation, we guarantee that all the short-circuit rewritings lead to a terminating computation in polytime over the size of the original specification, Φ_d. □

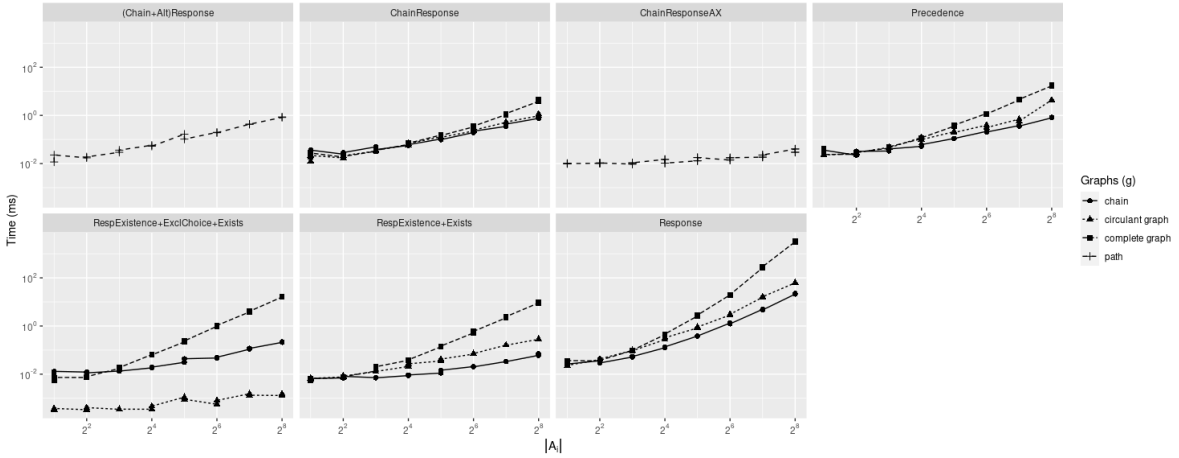

Figure A4. Running times for rewriting Φ_d as Φ'_d

Lemma A8. *The computation of the rewriting rules leads to a terminating procedure.*

Proof. Last, we consider the termination for the procedure sketched in §3.3. In the worst case scenario, we are never generating an inconsistency, thus never abruptedly terminating the procedure by returning an inconsistent $\perp$ specification. If no rewriting rule is ever triggered, we then simply linearly iterate over all the edges of the graphs $G^\star$ for each template $\star$ without triggering any of the past functions. Furthermore, the iteration over the domain of F will be always be the same. Therefore, no additional overhead is introduced and the procedure terminates. On the other hand, we trigger at least one rewriting rule that, per the (A1) assumption, never generates a clause containing an activity label that was not present in Σ: under this, all the previous functions are also guaranteed not to generate more information than the one being available in Σ. Furthermore, this computation never generates new edges to be visited, as the only expansion phase occurs as described in Lemma A7. At most. we trigger the deletion of edges from the graph in $O(1)$ that we are currently generating or the generation of novel `Exists/Absence` clauses, but we never generate other clauses. Furthermore, the graphs are mainly depleted after iterating over the edges, by invoking REDUCE or CLEAR functions for each x s.t. $F(x) =$ **false** after the aforementioned edge iteration. Even in this scenario, the computation is guaranteed to converge in polynomial time: as we mainly boil down the clauses to absences and existentials while ensuring to remove entailing clauses, and given that the number of such templates is finite, we therefore guarantee to converge in at most polynomial time over the size of the declarative specification. □

 doi:10.20944/preprints202403.0286.v1

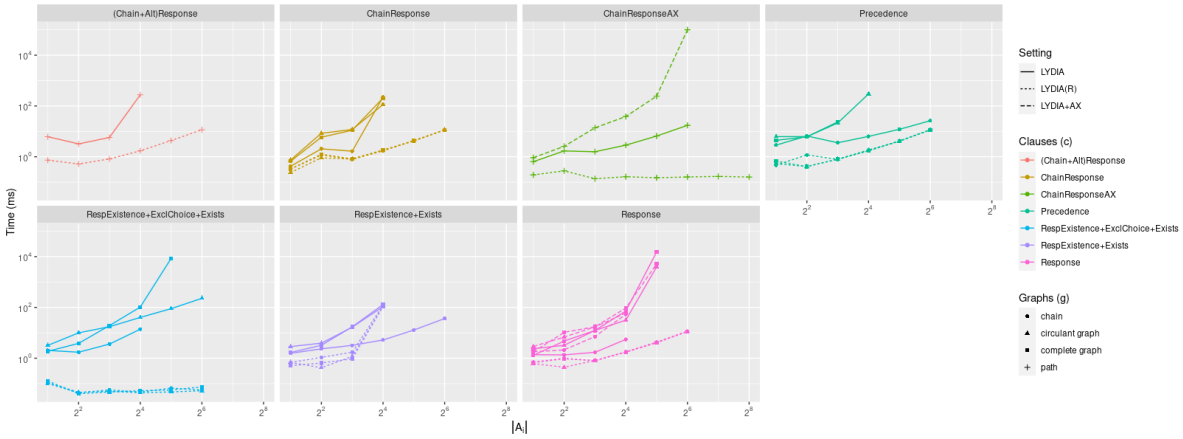

Figure A5. Running times for Lydia for both $\Phi_i^{c,g}$ (LYDIA), $\Phi_i'^{c}$ (LYDIA(R)) and $(\Phi_i^{c,g})_{\downarrow\Sigma}$ (LYDIA+AX)

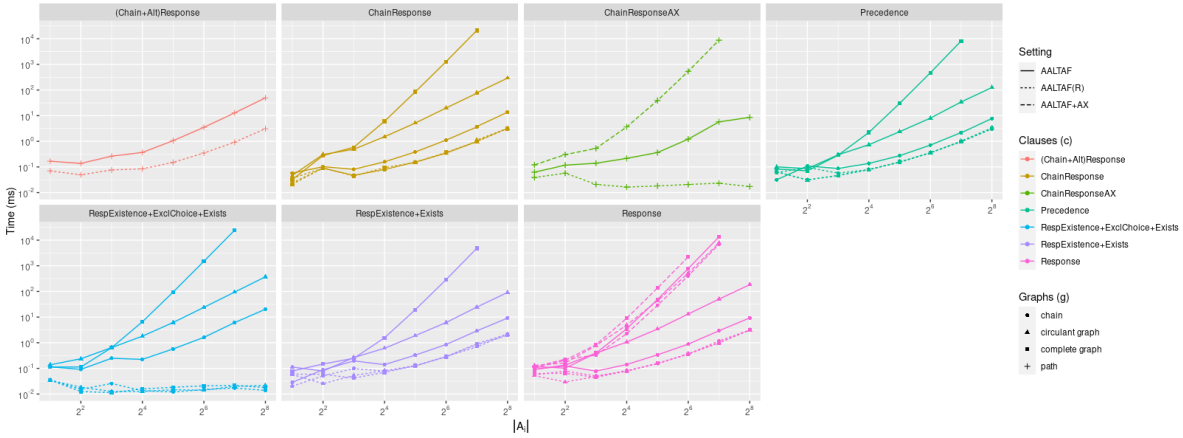

Figure A6. Running times for AALTAF for both $\Phi_i^{c,g}$ (AALTAF), $\Phi_i'^{c}$ (AALTAF(R)) and $(\Phi_i^{c,g})_{\downarrow\Sigma}$ (AALTAF+AX)

Appendix E. Detailed Benchmarks

This section provides the aforementioned benchmarks in a greater size, so as to better remark the running times associated with each single algorithm. As in the main paper, missing data points for specific 2^i values of $|\Sigma|$ refer to missing data points due to out-of-memory issues while dealing with the automaton representation of the LTL_f formula. Figure A4 provides all the running times for the reducer, while Figure A5 andFigure A6 refers to the running time of the formal synthesis (Lydia) and SAT-Checker (AALTAF) tasks over different specifications representations.

References

1. Bergami, G. Streamlining Temporal Formal Verification over Columnar Databases. *Information* **2024**, *15*. https://doi.org/10.3390/info15010034.
2. Bergami, G.; Appleby, S.; Morgan, G. Quickening Data-Aware Conformance Checking through Temporal Algebras. *Inf.* **2023**, *14*, 173.
3. De Giacomo, G.; Favorito, M. Compositional Approach to Translate LTLf/LDLf into Deterministic Finite Automata. *Proceedings of the International Conference on Automated Planning and Scheduling* **2021**, *31*, 122–130.
4. Bergami, G.; Appleby, S.; Morgan, G. Specification Mining over Temporal Data. *Computers* **2023**, *12*.
5. Pnueli, A. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), 1977, pp. 46–57. https://doi.org/10.1109/SFCS.1977.32.
6. Giacomo, G.D.; Masellis, R.D.; Montali, M. Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness. In Proceedings of the Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July

27 -31, 2014, Québec City, Québec, Canada; Brodley, C.E.; Stone, P., Eds. AAAI Press, 2014, pp. 1027–1033. https://doi.org/10.1609/aaai.v28i1.8872.

7. Pesić, M.; Schonenberg, H.; van der Aalst, W.M. DECLARE: Full Support for Loosely-Structured Processes. In Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 2007, pp. 287–287.
8. 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 Medical Informatics and Decision Making* **2020**, *20*, 303. https://doi.org/10.1186/s12911-020-01323-7.
9. Giacomo, G.D.; Maggi, F.M.; Marrella, A.; Patrizi, F. On the Disruptive Effectiveness of Automated Planning for LTL*f*-Based Trace Alignment. In Proceedings of the Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA; Singh, S.; Markovitch, S., Eds. AAAI Press, 2017, pp. 3555–3561. https://doi.org/10.1609/aaai.v31i1.11020.
10. Bergami, G.; Maggi, F.M.; Marrella, A.; Montali, M. Aligning Data-Aware Declarative Process Models and Event Logs. In Proceedings of the Business Process Management; Polyvyanyy, A.; Wynn, M.T.; Van Looy, A.; Reichert, M., Eds., Cham, 2021; pp. 235–251.
11. 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 Systems with Applications* **2022**, *201*, 117176.
12. Wang, C.; Wu, K.; Zhou, T.; Cai, Z. Time2State: An Unsupervised Framework for Inferring the Latent States in Time Series Data. *Proc. ACM Manag. Data* **2023**, *1*.
13. Yazi, A.F.; Çatak, F.Ö.; Gül, E. Classification of Methamorphic Malware with Deep Learning(LSTM). In Proceedings of the 27th Signal Processing and Communications Applications Conference, SIU 2019, Sivas, Turkey, April 24-26, 2019. IEEE, 2019, pp. 1–4.
14. Catak, F.O.; Ahmed, J.; Sahinbas, K.; Khand, Z.H. Data augmentation based malware detection using convolutional neural networks. *PeerJ Computer Science* **2021**, *7*, e346.
15. Li, J.; Pu, G.; Zhang, Y.; Vardi, M.Y.; Rozier, K.Y. SAT-based explicit LTLf satisfiability checking. *Artificial Intelligence* **2020**, *289*, 103369.
16. Geatti, L.; Gianola, A.; Gigante, N.; Winkler, S. Decidable Fragments of LTL_f Modulo Theories. In Proceedings of the ECAI; Gal, K.; et al.., Eds., 2023.
17. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms, 3rd Edition*; MIT Press, 2009.
18. Bergami, G. Fast Synthetic Data-Aware Log Generation for Temporal Declarative Models. In Proceedings of the Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), New York, NY, USA, 2023; GRADES & NDA '23.