Monitoring Compliance Policies over Incomplete and Disagreeing Logs[⋆]

David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu

Institute of Information Security, ETH Zurich

Abstract. When monitoring system behavior to check compliance against a given policy, one is sometimes confronted with incomplete knowledge about system events. In IT systems, such incompleteness may arise from logging infrastructure failures and corrupted log files, or when the logs produced by different system components disagree on whether actions took place. In this paper, we present a policy language with a three-valued semantics that allows one to explicitly reason about incomplete knowledge and handle disagreements. Furthermore, we present a monitoring algorithm for an expressive fragment of our policy language. We illustrate through examples how our approach extends compliance monitoring to systems with logging failures and disagreements.

1 Introduction

Laws, inter-business contracts, security policies, and similar normative regulations define compliance requirements that IT systems need to enforce. For example, IT systems in US hospitals must enforce HIPAA [1], which regulates the dissemination of medical records and the subsequent obligations that medical staff are expected to fulfill. For banks, separation-of-duty constraints should reduce the risk of fraud [2]. Data-usage contracts between different businesses regulate how sensitive documents are exchanged and subsequently disposed. Checking whether implemented IT systems comply with a body of regulations or policies is a problem of growing importance, since non-compliant behavior can lead to serious security breaches, monetary penalties, and the erosion of stakeholder's internal standards and commitments.

Runtime-verification techniques [4,5,19,22–24] offer a promising approach for automated compliance checking of IT systems. These techniques require logging mechanisms for recording policy-relevant system actions (represented as events), a suitable language for expressing policies and unambiguously defining permissible and prohibited system behavior, and a monitoring algorithm for determining and reporting policy violations.

In complex IT systems, which are usually composed of numerous interacting subsystems, the problem of incomplete knowledge about performed actions arises. In particular, logs may contain gaps due to corrupted files, logging-mechanism crashes, network failures, and so forth. Furthermore, when multiple

[⋆] This work was partly supported by Google Inc.

logs are required to verify compliant behavior, they may disagree whether certain actions took place. For example, sharing a sensitive document between two parties may require the recipient to fulfill certain obligations. Thus, when analyzing the recipient's and the sender's logs against this policy, we need to treat all disagreements over the transfer of the document as incomplete knowledge, since favoring one log over the other may result in missed violations or false positives. Most runtime monitors, however, do not distinguish between a gap and a non-occurrence of an event. Thus applying them to incomplete logs can yield *wrong* results. For example, consider a policy like *a subject can access a document if the subject is not blacklisted*. If it is unknown whether a subject is blacklisted, then the subject is incorrectly reported as compliant.

In this paper, we present a policy language and an accompanying monitoring algorithm that accounts for possibly incomplete and disagreeing logs. At the core of our approach is a three-valued truth space [25]. In addition to the classical Boolean values t (true) and f (false), which respectively represent the occurrence and non-occurrence of an event, we represent a knowledge gap about an event's occurrence by the third truth value $\bot$. Furthermore, when evaluating policies, their interpretation is as follows: the Boolean values t and f correspond to policy compliance and policy violation and $\bot$ represents an inconclusive answer, which can be due to knowledge gaps of event occurrences or disagreeing events.

Our policy language is a variant of a first-order temporal logic [7, 17]. First-order temporal logics have been a good fit in various case studies for formally expressing and monitoring compliance policies, see, e.g., [5,23]. Special care must be taken when defining the semantics of a logic with additional truth values besides the classical Boolean values. In particular, a vital requirement for monitoring incomplete and disagreeing logs is to ensure that reported violations cannot be retracted if or when the log is eventually completed, for example, by recovering lost files. Otherwise, these results are of no value. More precisely, formalized policies must be monotonic with respect to the underlying partial ordering on knowledge, i.e., $\bot$ is less than f and t, and f and t are incomparable [9, 10, 20]. Our policy language guarantees this monotonicity requirement. Furthermore, the third truth value $\bot$ is a first-class citizen at the object-level of our policy language: the classical logical connectives are extended to the three-valued truth space and there are specific connectives that guarantee expressive-completeness with respect to the set of knowledge-monotonic operators. Such monotonic operators are needed in our application context to express at the logic's object-level how disagreements between logged events should be resolved.

The monitoring algorithm presented in this paper for this three-valued setting is inspired by the one from [6, 7] for the standard Boolean setting. It iteratively scans the logged actions and soundly reports violations, i.e., whenever a violation is reported, it indeed is a policy violation. It also soundly reports potential violations, i.e., depending on how the knowledge gaps are filled, these might turn out to be real policy violations. However, our monitoring algorithm is not complete in the sense that some policy violations might not be reported. This limitation stems from the expressivity of our policy language over infinite domains. Importantly, however, for an expressive fragment, which retains all the language's connectives but limits the usage of free variables within a formula, we show that our monitoring algorithm guarantees completeness.

In summary, our main contribution is a solution to the problem of checking policy compliance in the presence of logging failures and disagreements between logged events. Our solution comprises a policy language and a monitoring algorithm. The policy language supports reasoning with incomplete knowledge. The monitoring algorithm may be used either off-line (for audit) or on-line (at runtime), and reports all policy violations and potential policy violations for an expressive fragment of our language. Although several features of our solution are present in related work—see Section 6 for a comparison—combining them to solve the stated problem is novel. In particular, our language is the first compliance language to consider three truth values at the object level, and our monitoring algorithm is the first algorithm to guarantee both soundness and completeness in a three-valued first-order setting.

The remainder of the paper is structured as follows. In Section 2, we describe our abstract logging setting. In Section 3, we introduce our policy language. In Section 4, we analyze our policy language with respect to monotonicity and expressiveness. In Section 5, we present our monitoring algorithm. Finally, in Sections 6 and 7, we discuss related work and draw conclusions. Additional technical details are given in the appendix.

2 Logging Knowledge Base

We abstract from a particular *physical* log file structure, and view a logging infrastructure as producing a single logging knowledge base, which is evaluated against a compliance policy. A logging knowledge base uses the three-valued truth space $\mathbf{3} := \{\mathsf{t}, \mathsf{f}, \bot\}$ to explicitly distinguish between what is known and unknown regarding event occurrences.

To formally define a logging knowledge base over $\mathbf{3}$, we introduce a *logging signature* S, which is a tuple (C, R, ι), where C is a finite set of constant symbols, R is a finite set of predicates disjoint from C, and the function $\iota : R \to \mathbb{N}$ assigns each predicate $r \in R$ an arity $\iota(r)$. Each predicate r denotes an action, and its arguments $\bar{a}$ denote the action's parameters, $r(\bar{a})$ denoting an event. A *logging structure* $\mathcal{D}$ over the signature S consists of a domain $|\mathcal{D}| \neq \emptyset$ and interpretations $c^{\mathcal{D}} \in |\mathcal{D}|$, and $r_{\mathsf{t}}^{\mathcal{D}} \subseteq |\mathcal{D}|^{\iota(r)}$ and $r_{\mathsf{f}}^{\mathcal{D}} \subseteq |\mathcal{D}|^{\iota(r)}$, for each $c \in C$ and $r \in R$, such that $r_{\mathsf{t}}^{\mathcal{D}}$ and $r_{\mathsf{f}}^{\mathcal{D}}$ are disjoint. We let $r_{\bot}^{\mathcal{D}} := |\mathcal{D}|^{\iota(r)} \setminus (r_{\mathsf{t}}^{\mathcal{D}} \cup r_{\mathsf{f}}^{\mathcal{D}})$. We define a *logging knowledge base* over the signature S as a sequence $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ of logging structures over S, with the following properties:

1. $\bar{\mathcal{D}}$ has constant domains, that is, $|\mathcal{D}_i| = |\mathcal{D}_{i+1}|$, for all $i \geq 0$. We denote the domain by $|\bar{\mathcal{D}}|$.
2. Each constant symbol $c \in C$ has a rigid interpretation, that is, $c^{\mathcal{D}_i} = c^{\mathcal{D}_{i+1}}$, for all $i \geq 0$. We denote c's interpretation by $c^{\bar{\mathcal{D}}}$.

We call the indices of the elements in the sequence $\bar{\mathcal{D}}$ *time points* and denote them with the Greek letter τ. We interpret a logging knowledge base $\bar{\mathcal{D}}$ as follows:

- If $\bar{a} \in r_{\mathsf{t}}^{\mathcal{D}_\tau}$, then the event $r(\bar{a})$ happened at the time point τ.
- If $\bar{a} \in r_{\mathsf{f}}^{\mathcal{D}_\tau}$, then the event $r(\bar{a})$ did not happen at the time point τ.
- If $\bar{a} \in r_{\perp}^{\mathcal{D}_\tau}$, then $\bar{\mathcal{D}}$ contains a *knowledge gap* at the time point τ with regard to whether the event $r(\bar{a})$ happened at τ. In practice, a gap is determined by additional information about logging failures.

Thus a logging knowledge base states explicitly whether logging information is complete at a time point τ. In case of incomplete knowledge, we have $r_{\perp}^{\mathcal{D}_\tau} \neq \emptyset$.

We extend the classical logging assumption, whereby there are only finitely many events happening at each time point, to a three-valued setting.

Assumption 1. *Let $\bar{\mathcal{D}}$ be a logging knowledge base over the signature (C, R, ι). For each $r \in R$ and $\tau \in \mathbb{N}$, either $r_{\mathsf{t}}^{\mathcal{D}_\tau}$ is finite and $r_{\perp}^{\mathcal{D}_\tau} = \emptyset$, or $r_{\perp}^{\mathcal{D}_\tau} = |\bar{\mathcal{D}}|^{\iota(r)}$.*

This assumption formalizes that as long as a particular logging process is running, it correctly records all events. If the process crashes, then nothing is recorded until the process is restarted. In line with our model of a logging knowledge base, this means that at each time point τ and for each relation r either $r_{\perp}^{\mathcal{D}_\tau} = \emptyset$ or $r_{\perp}^{\mathcal{D}_\tau} = |\bar{\mathcal{D}}|^{\iota(r)}$.

Note that a logging knowledge base does not differentiate between multiple instances of the same event happening at the same time point. To do so, one would have to ensure that either the time points' granularity is sufficient to render this scenario impossible, or to add unique artificial parameters (such as counters) for each such event instance.

3 Compliance Policy Language

In this section, we define our policy language $\mathcal{L}_3$ and illustrate with examples how policies are formalized and evaluated in the presence of incomplete knowledge. We also show how disagreements can be handled with $\mathcal{L}_3$'s operators.

Syntax and semantics. In the following, let $S = (C, R, \iota)$ be a signature and let V be a countably infinite set of variables, where $V \cap (C \cup R) = \emptyset$. Also, let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$. We often write an interval in $\mathbb{I}$ as $[b, b') := \{a \in \mathbb{N} \mid b \leq a < b'\}$, where $b \in \mathbb{N}$, $b' \in \mathbb{N} \cup \{\infty\}$, and $b < b'$.

Definition 2. *The $\mathcal{L}_3$ formulas over the signature S are given by the grammar*

$$\varphi ::= \mathsf{f} \mid r(t_1, \dots, t_{\iota(r)}) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \otimes \varphi \mid \forall x.\, \varphi \mid \varphi \,\mathsf{S}_I\, \varphi \mid \varphi \,\mathsf{U}_I\, \varphi\,,$$

where r ranges over the elements in R, the t_is over the elements in $C \cup V$, x over the elements in V, and I over the elements in $\mathbb{I}$.

Before formally defining the evaluation semantics, Figure 1(a) shows $\mathcal{L}_3$'s interpretation of the logical connectives over **3**. We mildly abuse notation and use same symbols to denote logical connectives and their corresponding three-valued operators. The classical connectives $\neg$ and $\wedge$ retain their interpretation when restricted to the Boolean values t and f. The $\otimes$ connective does not have

$\neg$	
t	f
f	t
$\bot$	$\bot$

$\wedge$	t	f	$\bot$
t	t	f	$\bot$
f	f	f	f
$\bot$	$\bot$	f	$\bot$

$\otimes$	t	f	$\bot$
t	t	$\bot$	$\bot$
f	$\bot$	f	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

(a) primitive operators

$\vee$	t	f	$\bot$
t	t	t	t
f	t	f	$\bot$
$\bot$	t	$\bot$	$\bot$

$\rightarrow$	t	f	$\bot$
t	t	f	$\bot$
f	t	t	t
$\bot$	t	$\bot$	$\bot$

(b) derived operators

Fig. 1. Truth tables for three-valued operators (strong Kleene logic [25]).

a classical counterpart. Intuitively, it represents a *consensus* on how much truth can be agreed upon and is useful for combining different sources of knowledge when neither t nor f should be preferred over the other.

In the following, a *valuation* is a mapping $\theta : V \to |\bar{\mathcal{D}}|$. For a valuation θ, the variable vector $\bar{x} = (x_1, \dots, x_n)$, and $\bar{d} = (d_1, \dots, d_n) \in |\bar{\mathcal{D}}|^n$, $\theta[\bar{x} \mapsto \bar{d}]$ is the valuation mapping x_i to d_i, for $i \in \{1, \dots, n\}$, and the other variables' valuation is unaltered. We abuse notation by applying a valuation θ also to constant symbols $c \in C$, with $\theta(c) := c^{\bar{\mathcal{D}}}$.

Definition 3. *Let $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ be a temporal structure over the signature S, θ a valuation, and $\tau \in \mathbb{N}$ a time stamp. We inductively define the mapping $[\![\cdot]\!]^{\bar{\mathcal{D}},\theta,\tau}$ from formulas over S to values in $\mathbf{3}$ as follows:*

$$
\begin{aligned}
[\![\mathsf{f}]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \mathsf{f}\\
[\![r(t_1, \dots, t_{\iota(r)})]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= v \textit{ if } \big(\theta(t_1), \dots, \theta(t_{\iota(r)})\big) \in r_v^{\mathcal{D}_\tau}, \textit{ where } v \in \mathbf{3}\\
[\![\neg\varphi]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \neg[\![\varphi]\!]^{\bar{\mathcal{D}},\theta,\tau}\\
[\![\varphi_1 \wedge \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau} \wedge [\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau}\\
[\![\varphi_1 \otimes \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau} \otimes [\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau}\\
[\![\forall x.\, \varphi]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \textstyle\bigwedge_{d \in |\bar{\mathcal{D}}|} [\![\varphi]\!]^{\bar{\mathcal{D}},\theta[x \mapsto d],\tau}\\
[\![\varphi_1 \mathsf{S}_I\, \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \textstyle\bigvee_{\tau-\tau' \in I} \big([\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau'} \wedge \bigwedge_{\tau'' \in (\tau',\tau]} [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau''}\big)\\
[\![\varphi_1 \mathsf{U}_I\, \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \textstyle\bigvee_{\tau'-\tau \in I} \big([\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau'} \wedge \bigwedge_{\tau'' \in [\tau,\tau')} [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau''}\big)
\end{aligned}
$$

In this definition, $\bigwedge$ and $\bigvee$ are respectively the (possibly infinitary) meet and join over the ordering $\mathsf{f} \le \bot \le \mathsf{t}$. Note that they match the corresponding operators in Figure 1. The temporal connectives are accompanied by intervals and a formula of the form $\varphi \mathsf{S}_I\, \psi$ or $\varphi \mathsf{U}_I\, \psi$ is only satisfied in $\bar{\mathcal{D}}$ at the time point τ if it is satisfied within the bounds given by the interval I of the respective temporal operator. We may omit the interval I if it is $[0, \infty)$.

We introduce the following additional syntactic sugar. We write t for $\neg$f, $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, $\varphi \to \psi$ for $\neg\varphi \vee \psi$, and $\exists x.\, \varphi$ for $\neg\forall x.\, \neg\varphi$. For a vector of variables $\bar{x} = (x_1, x_2, \dots, x_n)$, with $n \ge 0$, we write $\forall \bar{x}.\, \varphi$ for $\forall x_1.\, \forall x_2 \dots \forall x_n.\, \varphi$. Moreover, we define the temporal connectives $\blacklozenge_I\, \psi$ and $\lozenge_I\, \psi$ as $\mathsf{t}\,\mathsf{S}_I\, \psi$ and $\mathsf{t}\,\mathsf{U}_I\, \psi$, respectively. Intuitively, $\blacklozenge_{[b,b')}\, \psi$ is t at τ, if ψ is t at least at one *past* time point in the time interval $[\max(0, \tau - b' - 1), \tau - b]$. If ψ is f at all these time points, then $\blacklozenge_{[b,b')}\, \psi$ is f at τ. The presence of at least one $\bot$ and no t results in the truth value $\bot$ for $\blacklozenge_{[b,b')}\, \psi$ at τ, since depending on how the incompleteness is resolved either outcome (t or f) is possible. The interpretation of $\lozenge_{[b,b')}\, \psi$ is similar for *future* time points. The dual temporal connectives are $\square_I\, \psi := \neg \lozenge_I \neg\psi$

and $\blacksquare_I\, \psi := \neg \blacklozenge_I \neg\psi$. We use standard conventions concerning the binding strength of connectives to omit parentheses. For instance, temporal connectives bind weaker than the other connectives. Furthermore, $\rightarrow$ binds weaker than $\vee$, which in turn binds weaker than $\wedge$ and $\otimes$.

Finally, we introduce some additional notation. Given a formula φ, we denote by $fv(\varphi)$ and $\bar{fv}(\varphi)$ the set and respectively the vector of free variables of φ. We call a formula φ *closed* if $fv(\varphi) = \emptyset$. For a formula φ with $\bar{fv}(\varphi) = \bar{x} = (x_1, \ldots, x_n)$, we define the set of elements of $|\bar{\mathcal{D}}|^n$ for which φ evaluates to $v \in \mathbf{3}$ at a time point $\tau \in \mathbb{N}$ as

$$\llbracket\varphi\rrbracket_v^{\bar{\mathcal{D}},\tau} := \left\{ \bar{d} \in |\bar{\mathcal{D}}|^n \;\middle|\; \llbracket\varphi\rrbracket^{\bar{\mathcal{D}},\theta[\bar{x}\mapsto\bar{d}],\tau} = v, \text{ for some valuation } \theta \right\}.$$

Compliance policies. Regardless of the policy language, compliance policies are usually given as a set of regulative *normative* statements (norms), which expess what an agent is obliged to do given some actions it has performed, or which conditions need to hold (or to have held) for an agent to be permitted to execute some actions. Norms are meant to be applied at all times within a system, and it has also been argued [11, 12] that deadlines are of essential importance in regulating temporal norms. Following these notions, compliance policies in $\mathcal{L}_3$ are formalized as follows:

Definition 4. *A* compliance policy *represented in $\mathcal{L}_3$ is a closed formula of the form $\square\, \forall\bar{x}.\, \psi$, where each future temporal connective in ψ is bounded.*

The outermost unbounded $\square$ connective specifies that a policy must be fulfilled at each time point. Bounded inner future temporal connectives guarantee that each obligation has a deadline.

We map the truth values onto policy evaluations as follows: t/f denotes that a policy is satisfied/violated, and $\perp$ denotes that it is unknown whether a policy is satisfied or violated. Furthermore, for a compliance policy $\square\, \forall\bar{x}.\, \psi$, it is often useful to report additional information regarding its violations, which is given by the aforementioned sets $\llbracket\psi\rrbracket_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$, $\llbracket\psi\rrbracket_{\perp}^{\bar{\mathcal{D}},\tau}$, and $\llbracket\psi\rrbracket_{\mathsf{t}}^{\bar{\mathcal{D}},\tau}$, for a time point τ. Their interpretation is as follows:

- The elements in $\llbracket\psi\rrbracket_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$ witness a policy violation at time point τ.
- For elements in $\llbracket\psi\rrbracket_{\perp}^{\bar{\mathcal{D}},\tau}$, it is unknown whether they violate the policy at time point τ. They are potential violations.
- The elements in $\llbracket\psi\rrbracket_{\mathsf{t}}^{\bar{\mathcal{D}},\tau}$ satisfy the policy at time point τ.

In Section 4, we show that all reported violations and satisfactions at τ persist regardless of how incompleteness is resolved.

Examples. We begin with the following security policy requiring that *if a request is serviced at a web-server then it must not have been denied by a firewall.* In practice, this policy would be a part of a larger specification. However, this excerpt is enough to illustrate how $\mathcal{L}_3$'s semantics deal with logging failures. We formalize this policy as $\square\, \forall r.\, \psi_1$, where

$$\psi_1 := \mathit{service}(r) \rightarrow \neg \blacklozenge_{[0,4)}\, \mathit{deny}(r)\,.$$

When there are no failures, then any serviced request that has previously been denied violates the policy, and is contained in $[\![\psi_1]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$. If the web-server's logger crashes at a time point τ, i.e. $\mathit{service}_{\bot}^{\mathcal{D}_\tau} = |\bar{\mathcal{D}}|$, then all requests that had been denied at the previous four time points by the firewall potentially violate the policy, i.e. $[\![\psi_1]\!]_{\bot}^{\bar{\mathcal{D}},\tau} = \bigcup_{\tau'} \mathit{deny}_{\mathsf{t}}^{\mathcal{D}_{\tau'}}$, where $\tau - 4 < \tau' \leq \tau$. If, however, there are no denied requests in the designated interval, the set $[\![\psi_1]\!]_{\bot}^{\bar{\mathcal{D}},\tau}$ is empty and the policy is therefore satisfied. This shows that not all logging failures must result in potential violations. We note that if all unknown events are treated as not to have happened, then the policy would be *wrongly* reported as satisfied.

For our second example, we focus on formalizing inter-business contracts. These contracts often specify obligations that the signing parties must enforce regarding the treatment of sensitive documents used during the collaborations. To ensure that each party complies with its obligations, a policy must specify how events are combined from different logs belonging to different stakeholders. For example, when two companies exchange sensitive information, the contract might say that *all received documents must be paid for within 5 days.* A straightforward, but naive, formalization of this policy is $\Box\, \forall d.\, \psi_2$, where

$$\psi_2 := \mathit{receive}(d) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\,.$$

The *receive* event is taken from the receiving stakeholder's log. This specification assumes that the receiving stakeholder is honest, since if its IT system does not log a received document, the stakeholder's behavior is trivially compliant according to the given specification. We can attempt to expand the formalization to include the sender's *send* event (from the sender's log) as follows

$$\psi_2' := \mathit{send}(d) \vee \mathit{receive}(d) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\,.$$

In this case, the receiver is obliged to pay if either it receives a document, or the sender says that it has sent the document. However, this is also unsatisfactory, as the sender can cheat and insert fictitious *send* events causing policy violations. In $\mathcal{L}_3$ we can combine the logs with the $\otimes$ operator and obtain[1]

$$\psi_2'' := \mathit{send}(d) \otimes \mathit{receive}(d) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\,.$$

In this case, all disagreements at some τ about payments are in $[\![\psi_2'']\!]_{\bot}^{\bar{\mathcal{D}},\tau}$, since $\bot \to \mathsf{f}$ is $\bot$. The specification no longer favors one stakeholder over the other. This has the benefit of not requiring additional pre-processing of logs, which would need its own language and semantics. We remark that the given specification cannot be directly expressed in existing compliance policy languages because $\bot$ does not exist at the object level in those languages.

[1] We assume that the time granularity is coarse enough to allow *receive* and *send* happen at the same time point. If a *receive* can happen with a delay of, e.g., at most one time unit after a *send*, a more elaborate formalization is required:

$$\Box\, \forall d.\, \big(\mathit{send}(d) \wedge (\mathit{send}(d) \otimes \Diamond_{[0,2)}\, \mathit{receive}(d)) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\big) \wedge \\ \big(\mathit{receive}(d) \wedge (\mathit{receive}(d) \otimes \blacklozenge_{[0,2)}\, \mathit{send}(d)) \to \Diamond_{[0,5)}\, \mathit{pay}(d)\big)\,.$$

For our third example, we consider a form of separation-of-duty constraint [2]: *a subject s may access an object o if it has not previously accessed some object o', where o''s dataset conflicts with o's.* One possible formalization of this requirement is

$$\Box\, \forall s.\, \forall o.\, \forall d.\, \forall o'.\, \forall d'.\, \mathit{access}(s,o,d) \wedge (\blacklozenge\, \mathit{access}(s,o',d')) \to \neg \mathit{conflict}(d,d')\,.$$

In this example, $\mathit{access}(s,o,d)$ records that s accessed o in a dataset d. The predicate *conflict* does not correspond to an event; it describes a property of a system state. When having the events $\mathit{conflict}_s$ and $\mathit{conflict}_f$ at hand, which mark the start point and the end point of two datasets being conflicting, the formula $\neg \mathit{conflict}_f(d,d')\, \mathsf{S}\, \mathit{conflict}_s(d,d')$ can be used to describe this state property. For the sake of brevity, we assume that an object belongs to at most one dataset. In case s accessed an o, and it is unknown whether s had any other accesses, then if there exists d' in conflict with d, such an access is a potential violation.

Notice that the above formalization only considers whether the data items are in conflict at the time point when o is accessed. This means that even if the datasets are in conflict just before the access, the policy is not violated. With respect to the separation-of-duty requirement, one may say that this behavior is in a compliance *gray* area. In $\mathcal{L}_3$, we define the following temporal connective C_I that treats such *gray* areas as $\perp$, signaling that it is unclear whether the policy is satisfied or violated:

$$\mathsf{C}_I \psi := (\blacklozenge_I\, \psi) \otimes (\blacksquare_I\, \psi)\,.$$

Intuitively, $\mathsf{C}_I\psi$ insists that the truth value of ψ does not change in the given past interval I. Any change results in $\perp$, and otherwise the truth value is not changed. We can define a similar temporal connective using $\Box$ and $\Diamond$ to mark a future gray zone. We make use of C_I by changing the original formalization to

$$\Box\, \forall s.\, \forall o.\, \forall d.\, \forall o'.\, \forall d'.\, \mathit{access}(s,o,d) \wedge (\blacklozenge\, \mathit{access}(s,o',d')) \to \mathsf{C}_{[0,2)} \neg \mathit{conflict}(d,d')\,,$$

where $[0,2)$ is a two-day *gray* zone interval.

4 Monotonicity and Compositional Expressiveness

A logging knowledge base may grow in knowledge by resolving missing information about the occurrences and non-occurrences of events, i.e., moving elements from $r_\perp^{\mathcal{D}_\tau}$ to the relations $r_{\mathsf{t}}^{\mathcal{D}_\tau}$ or $r_{\mathsf{f}}^{\mathcal{D}_\tau}$.

Definition 5. *An* extension *of a logging knowledge base $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ over $S = (C, R, \iota)$ is a logging knowledge base $\bar{\mathcal{D}}^\star = (\mathcal{D}_0^\star, \mathcal{D}_1^\star, \dots)$ over S with $|\bar{\mathcal{D}}^\star| = |\bar{\mathcal{D}}|$, $c^{\bar{\mathcal{D}}} = c^{\bar{\mathcal{D}}^\star}$ for all $c \in C$, and $r_b^{\mathcal{D}_\tau} \subseteq r_b^{\mathcal{D}_\tau^\star}$ for all $b \in \{\mathsf{t}, \mathsf{f}\}$, $\tau \in \mathbb{N}$, and $r \in R$.*

Under Assumption 1, an extension either does not alter a relation $r_\perp^{\mathcal{D}_\tau}$ or empties $r_\perp^{\mathcal{D}_\tau}$ by moving finitely many elements to $r_{\mathsf{t}}^{\mathcal{D}_\tau}$ and the remaining elements to $r_{\mathsf{f}}^{\mathcal{D}_\tau}$.

We say that a policy specification is *monotonic* if the t and f evaluations, over a given logging knowledge base, can never be retracted for any of its extensions. In other words, regardless of how the logging base's incompleteness is resolved, the policy violations and satisfactions persist. Monotonicity is a vital requirement for a compliance policy, because monotonic specifications prevent a non-compliant behavior from being turned into a compliant behavior by holding back information. In the following, we establish that for $\mathcal{L}_3$ all policy specifications are monotonic by construction. To formalize monotonicity, we first order the truth values with a partial ordering $\leq_k$ as follows: $\bot \leq_k \mathsf{f}$, $\bot \leq_k \mathsf{t}$, and f and t are incomparable. In short, f and t contain more knowledge than $\bot$. The following theorem states that the evaluations of $\mathcal{L}_3$'s formulas do not reduce the amount of knowledge, when incompleteness is resolved in a logging knowledge base's extension.

Theorem 6. *Given an $\mathcal{L}_3$ formula ψ, a valuation θ, and a logging knowledge base $\bar{\mathcal{D}}$, then $[\![\psi]\!]^{\bar{\mathcal{D}},\theta,\tau} \leq_k [\![\psi]\!]^{\bar{\mathcal{D}}^\star,\theta,\tau}$, for all extensions $\bar{\mathcal{D}}^\star$ of $\bar{\mathcal{D}}$ and all $\tau \in \mathbb{N}$.*

Proof. From the definition of a logging knowledge base's extension, and by structural induction using the fact that all of $\mathcal{L}_3$ connectives' corresponding operators are $\leq_k$-monotonic, including the infinitary operators for temporal connectives.

As a corollary, given a compliance policy $\square\,\forall\bar{x}.\,\psi$, a logging knowledge base $\bar{\mathcal{D}}$, a valuation θ, and a time point τ, if $[\![\psi]\!]^{\bar{\mathcal{D}},\theta,\tau}$ is t or f, then this evaluation persists at τ, for all extensions $\bar{\mathcal{D}}^\star$. Moreover, we have $[\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}}^\star,\tau} \supseteq [\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$ and $[\![\psi]\!]_{\mathsf{t}}^{\bar{\mathcal{D}}^\star,\tau} \supseteq [\![\psi]\!]_{\mathsf{t}}^{\bar{\mathcal{D}},\tau}$, for all extensions $\bar{\mathcal{D}}^\star$ and $\tau \in \mathbb{N}$. Therefore, even with incomplete knowledge it is sound to report the elements in $[\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$ as policy violations when monitoring $\bar{\mathcal{D}}$.

Given that all $\mathcal{L}_3$ policies are monotonic, an important question is: *Can all monotonic compositional operators for combining events from different logs be defined as syntactic sugar in $\mathcal{L}_3$?* If the answer is positive, then $\mathcal{L}_3$ does not need to be further extended. An n-ary three-valued operator $O : \mathbf{3}^n \to \mathbf{3}$ is *representable* using a set $\mathcal{C}$ of operators if O can be written as the functional composition of operators in $\mathcal{C}$. We utilize the following theorem to show that any monotonic operator can be expressed in $\mathcal{L}_3$.

Theorem 7 (Blamey [10]). *For any $n \in \mathbb{N}$, every $\leq_k$-monotonic n-ary operator over the $\mathbf{3}$ truth space is representable using the set $\{\mathsf{f}, \neg, \wedge, \otimes\}$ of operators.*

Blamey's proof is constructive and yields a function that given a monotonic operator produces an expression showing how to compose the operators f, $\neg$, $\wedge$, and $\otimes$. As $\mathcal{L}_3$ has all the corresponding connectives, such an expression can be seen as a formula in $\mathcal{L}_3$. Hence $\mathcal{L}_3$ can express any n-ary three-valued $\leq_k$-monotonic operator, including those for combining different logs.

5 Monitoring Algorithm

The input of our algorithm consists of a compliance policy $\square\,\forall\bar{x}.\,\varphi$ and a logging knowledge base $\bar{\mathcal{D}}$ over a signature $S = (C, R, \iota)$. The algorithm iteratively processes the logging structures $\mathcal{D}_\tau$, for each $\tau \in \mathbb{N}$. To process a structure $\mathcal{D}_\tau$ for formulas with bounded future operators, the algorithm might need to process structures $\mathcal{D}_{\tau'}$ with $\tau' > \tau$ as well. When run in the *on-line* mode, the algorithm waits until such structures become available. For the rest of this section, we fix the signature S, the logging knowledge base $\bar{\mathcal{D}}$, and the policy $\Box\,\forall\bar{x}.\varphi$. Furthermore, we assume that the domain $|\bar{\mathcal{D}}|$ is infinite.

At each iteration τ, the algorithm outputs a triple $(S^\tau_{\mathsf{t}}, S^\tau_{\mathsf{f}}, S^\tau_{\bot})$, where for each $v \in \mathbf{3}$, the element S^τ_v is either $\mathsf{Fin}\ V$, CoFin, or None, where Fin, CoFin, and None are labels standing respectively for "finite set", "cofinite set", and "inconclusive", and V is a finite set.

Our algorithm is sound, i.e. if $S^\tau_v = \mathsf{Fin}\ V$ then $V = [\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_v$, for all $v \in \mathbf{3}$ and $\tau \in \mathbb{N}$. However, our algorithm is not *complete*, where completeness means that the algorithm always returns a value from which one can deduce all compliant tuples ($[\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{t}}$), all violations ($[\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{f}}$), and all potential violations ($[\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_{\bot}$). Note that when φ has free variables, all these sets cannot be explicitly output, as at least one is infinite. However, if two sets are finite, then the third one is cofinite, and it is thus implicitly determined. Therefore our algorithm is complete when at least two of the elements of the returned triples are of the form $\mathsf{Fin}\ V$. When φ is closed, completeness means that at each iteration a truth value is returned, as the triples $(\mathsf{Fin}\ \{()\},\ \mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \emptyset)$, $(\mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \{()\},\ \mathsf{Fin}\ \emptyset)$, and $(\mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \{()\})$ correspond respectively with the truth values t, f, and $\bot$.

Incompleteness of our algorithm is rooted in the standard issues that arise when dealing with infinite domains [3], which $\mathcal{L}_3$ inherits from first-order queries in the Boolean setting. Consider for instance the formula $\psi = p(x) \vee q(y)$ with $x \neq y$ and assume that $p^{\mathcal{D}_\tau}_{\mathsf{t}}$ and $q^{\mathcal{D}_\tau}_{\mathsf{t}}$ are finite and non-empty, and $p^{\mathcal{D}_\tau}_{\bot} = q^{\mathcal{D}_\tau}_{\bot} = \emptyset$, for some $\tau \in \mathbb{N}$. Then $[\![\psi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{t}}$ and $[\![\psi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{f}}$ are neither finite nor cofinite, hence our algorithm cannot deal with it: at τ, it returns $(\mathsf{None},\ \mathsf{None},\ \mathsf{Fin}\ \emptyset)$. Formulas such as ψ are problematic in the Boolean setting, since their evaluation results are domain-dependent [3]. In the three-valued setting, there are similar issues, even for formulas that are non-problematic in the Boolean setting. Consider the formula $\psi' = p(x) \wedge q(y)$ with $p^{\mathcal{D}_\tau}_{\mathsf{t}}$ finite and non-empty and $q^{\mathcal{D}_\tau}_{\bot} = |\bar{\mathcal{D}}|$, for some $\tau \in \mathbb{N}$. Then both $[\![\psi']\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{t}}$ and $[\![\psi']\!]^{\bar{\mathcal{D}},\tau}_{\bot}$ are infinite and domain-dependent.

Even though the algorithm is incomplete on $\mathcal{L}_3$, we obtain completeness for a fragment of $\mathcal{L}_3$, presented at the end of this section.

Algorithmic overview. We briefly describe the main ideas underlying the algorithm. Due to space constraints, a detailed presentation is deferred to Appendix A.

The algorithm's core is the procedure eval, whose arguments are a formula ψ, a finite set $\Gamma = \{(r, E_r) \mid r \in R\}$ representing the relations of the logging structure $\mathcal{D}_\tau$, and a time point τ. The values E_r, i.e., the second component of elements in Γ, as well as the return value of the eval procedure, are triples of the form $(S_{\mathsf{t}}, S_{\mathsf{f}}, S_{\bot})$, where each S_v with $v \in \mathbf{3}$ is either $\mathsf{Fin}\ V$, CoFin, or None. Such values satisfy (either by Assumption 1 or by construction) the following invariant with regard to some formula γ and time point τ: if $S_v = \mathsf{Fin}\ V$, then $[\![\gamma]\!]^{\bar{\mathcal{D}},\tau}_v$ is a finite subset of $|\bar{\mathcal{D}}|^{|fv(\gamma)|}$ and $V = [\![\gamma]\!]^{\bar{\mathcal{D}},\tau}_v$; if $S_v = \mathsf{CoFin}$, then $[\![\gamma]\!]^{\bar{\mathcal{D}},\tau}_v$

```
proc init($\varphi$)
  for each $\psi \in \mathsf{sf}(\varphi)$ with $\psi = \psi \mathsf{S}_I \psi'$ do
    $L_\psi \leftarrow \langle\rangle$

proc eval($\varphi$, $\Gamma$, $\tau$)
  case $\varphi = \mathsf{f}$
    return (Fin $\emptyset$, Fin $\{()\}$, Fin $\emptyset$)

  case $\varphi = r(\bar{t})$
    $E_r \leftarrow$ get_value($r$, $\Gamma$)
    return eval_predicate($\varphi$, $E_r$)

  case $\varphi = \neg\psi$
    return eval_neg(eval($\psi$, $\Gamma$, $\tau$))

  case $\varphi = \psi \wedge \psi'$
    $E_\psi \leftarrow$ ($\psi$, eval($\psi$, $\Gamma$, $\tau$))
    $E_{\psi'} \leftarrow$ ($\psi'$, eval($\psi'$, $\Gamma$, $\tau$))
    return eval_and($E_\psi$, $E_{\psi'}$)

  case $\varphi = \psi \otimes \psi'$
    $E_\psi \leftarrow$ ($\psi$, eval($\psi$, $\Gamma$, $\tau$))
    $E_{\psi'} \leftarrow$ ($\psi'$, eval($\psi'$, $\Gamma$, $\tau$))
    return eval_times($E_\psi$, $E_{\psi'}$)

  case $\varphi = \forall\bar{x}.\psi$
    $E_\psi \leftarrow$ eval($\psi$, $\Gamma$, $\tau$)
    return eval_forall($\bar{x}$, $\psi$, $E_\psi$)

  case $\varphi = \psi \mathsf{S}_I \psi'$
    $E_\psi \leftarrow$ eval($\psi$, $\Gamma$, $\tau$)
    $E_{\psi'} \leftarrow$ eval($\psi'$, $\Gamma$, $\tau$)
    return eval_since($\varphi$, $\tau$, $E_\psi$, $E_{\psi'}$)
```

Fig. 2. The init and eval procedures.

is a cofinite subset of $|\bar{\mathcal{D}}|^{|fv(\gamma)|}$ and the other two elements of the triple are of the form $S_{v'} = \mathsf{Fin}\ V'$, for $v' \in \mathbf{3} \setminus \{v\}$. This invariant is denoted as $Inv(\gamma, \tau, E)$, where $E = (S_\mathsf{t}, S_\mathsf{f}, S_\perp)$. By Assumption 1, the values E_r from the set Γ satisfy the invariant $Inv(r(\bar{x}), \tau, E_r)$, where $\bar{x}$ is a sequence of distinct variables of length $\iota(r)$. We prove in Theorem 9 that the return value E of $\mathsf{eval}(\varphi,\ \Gamma,\ \tau)$ satisfies the invariant $Inv(\varphi, \tau, E)$, thus establishing the correctness of our algorithm.

The eval procedure, given in Figure 2, is called recursively over ψ's subformulas. The procedure performs a case distinction on all possible top-level connectives. Some of the sub-procedures used by eval are in Figure 3, while the remaining the pseudo-code is given in the Appendix.

Next, we sketch each case of the eval procedure. The simplest case is when ψ is the truth value f. In this case we simply return the triple $(\mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \{()\},\ \mathsf{Fin}\ \emptyset)$. When ψ is of the form $r(\bar{t})$ for some predicate r, we first retrieve the value E_r associated with r from the set Γ of pairs. We then retrieve the sets $[\![r(\bar{t})]\!]_v^{\bar{\mathcal{D}},\tau}$ from $r_v^{\mathcal{D}_\tau}$, for each $v \in \mathbf{3}$, by filtering the relations $r_v^{\mathcal{D}_\tau}$ according to the implicit constraints present in the sequence $\bar{t}$ of constants and variables.

To evaluate formulas ψ whose top-most connective is a non-temporal connective, we first evaluate the direct sub-formulas of ψ and then compute, whenever possible, the sets $[\![\psi]\!]_v^{\bar{\mathcal{D}},\tau}$ for $v \in \mathbf{3}$, using the equalities given in Lemma 8 below. These equalities extend the standard equalities that express the relationship between first-order logic and relational algebra, from the Boolean to the three-valued setting. They use the relational algebra operators *projection* and *join* [3]. We refer to the Appendix A for their formal definitions, and here we proceed with their intuitive description. As the temporal aspect is not relevant in this case of eval, we also fix the time point τ and drop the superscript in $[\![\psi]\!]_v^{\bar{\mathcal{D}},\tau}$, i.e., we just write $[\![\psi]\!]_v$, for $v \in \mathbf{3}$ and a formula ψ.

Given a formula ψ and a truth value $v \in \mathbf{3}$, we can see the set $[\![\psi]\!]_v$ as a *named relation*, where columns in $[\![\psi]\!]_v$ are named by the free variables in $fv(\psi)$. Given a free variable x of ψ, the *projection* of the tuples in the relation $[\![\psi]\!]_v$

```
proc eval_and($H_\psi$, $H_{\psi'}$)
  $R_t \leftarrow$ join($H_\psi$, $H_{\psi'}$, t, t)
  $R_f \leftarrow$ union($H_\psi$, $H_{\psi'}$, f, f)
  $R_\bot \leftarrow$ eval_and$_\bot$($H_\psi$, $H_{\psi'}$)
  return update_cofin($\psi \wedge \psi'$, $R_t$, $R_f$, $R_\bot$)

proc eval_and$_\bot$($H_\psi$, $H_{\psi'}$)
  $R_1 \leftarrow$ join($H_\psi$, $H_{\psi'}$, t, $\bot$)
  $R_2 \leftarrow$ join($H_\psi$, $H_{\psi'}$, $\bot$, t)
  $R_3 \leftarrow$ join($H_\psi$, $H_{\psi'}$, $\bot$, $\bot$)
  case $R_1$, $R_2$, $R_3$ = Fin $V_1$, Fin $V_2$, Fin $V_3$
    return Fin ($V_1 \cup V_2 \cup V_3$)
  otherwise
    return None

proc eval_neg($S_t, S_f, S_\bot$)
  return ($S_f, S_t, S_\bot$)

proc eval_forall($\bar{x}$, $\psi$, ($S_t, S_f, S_\bot$))
  ($R_t, R_f, R_\bot$) $\leftarrow$ (None, None, None)
  case $S_t$ = Fin $T$
    $R_t \leftarrow$ Fin $\emptyset$
    case $S_\bot$ = Fin $U$
      $R_\bot \leftarrow$ Fin $\emptyset$
  case $S_f$ = Fin $F$
    $\bar{s} \leftarrow$ get_positions($\bar{x}, \psi$)
    $R_f \leftarrow$ Fin ($\pi_{\bar{s}}(F)$)
    case $S_\bot$ = Fin $U$
      $R_\bot \leftarrow$ Fin ($\pi_{\bar{s}}(U) \setminus \pi_{\bar{s}}(F)$)
  return update_cofin($\forall \bar{x}.\psi$, $R_t$, $R_f$, $R_\bot$)
```

```
proc eval_times($H_\psi$, $H_{\psi'}$)
  $R_t \leftarrow$ join($H_\psi$, $H_{\psi'}$, t, t)
  $R_f \leftarrow$ join($H_\psi$, $H_{\psi'}$, f, f)
  $R_\bot \leftarrow$ eval_times$_\bot$($H_\psi$, $H_{\psi'}$)
  return update_cofin($\psi \otimes \psi'$, $R_t$, $R_f$, $R_\bot$)

proc eval_times$_\bot$($H_\psi$, $H_{\psi'}$)
  $R_1 \leftarrow$ union($H_\psi$, $H_{\psi'}$, $\bot$, $\bot$)
  $R_2 \leftarrow$ join($H_\psi$, $H_{\psi'}$, t, f)
  $R_3 \leftarrow$ join($H_\psi$, $H_{\psi'}$, f, t)
  case $R_1$, $R_2$, $R_3$ = Fin $V_1$, Fin $V_2$, Fin $V_3$
    return Fin ($V_1 \cup V_2 \cup V_3$)
  otherwise
    return None

proc update_cofin($\psi$, $R_t$, $R_f$, $R_\bot$)
  $R_t \leftarrow$ update($\psi$, $R_t$, $R_f$, $R_\bot$)
  $R_f \leftarrow$ update($\psi$, $R_f$, $R_t$, $R_\bot$)
  $R_\bot \leftarrow$ update($\psi$, $R_\bot$, $R_t$, $R_f$)
  return ($R_t$, $R_f$, $R_\bot$)

proc update($\psi$, $R_1$, $R_2$, $R_3$)
  case $R_2$ = Fin _ and $R_3$ = Fin _
    if $fv(\psi) \neq \emptyset$ then return CoFin
    else if $R_2$ = Fin $\emptyset$ and $R_3$ = Fin $\emptyset$ then
      return Fin {()}
    else
      return Fin $\emptyset$
  otherwise
    return $R_1$
```

Fig. 3. The eval_neg, eval_and, eval_times, and eval_forall procedures.

on the columns corresponding to other free variables is denoted $\pi_x([\![\psi]\!]_v)$. For instance, if $[\![p(x,y)]\!]_{\mathsf{t}} = \{(0,2),(1,2),(1,3)\}$, then $\pi_x([\![p(x,y)]\!]_{\mathsf{t}}) = \{(2),(3)\}$. For $v, v' \in \mathbf{3}$, the *natural join* of the sets $[\![\psi]\!]_v$ and $[\![\psi']\!]_{v'}$, denoted $[\![\psi]\!]_v \bowtie [\![\psi']\!]_{v'}$, is the set of tuples for which the projections on the columns, corresponding to ψ's and ψ''s free variables, are in $[\![\psi]\!]_v$ and respectively in $[\![\psi']\!]_{v'}$, and the fields of which match on the common free variables. For instance, if $[\![q(y,z)]\!]_{\mathsf{t}} = \{(2,4)\}$, then $[\![p(x,y)]\!]_{\mathsf{t}} \bowtie [\![q(y,z)]\!]_{\mathsf{t}} = \{(0,2,4),(1,2,4)\}$. We adopt the convention that $\bowtie$ binds stronger than $\cup$.

Lemma 8. *Let $\bar{\mathcal{D}}$ be a logging knowledge base, τ be a time point, and ψ and ψ' be $\mathcal{L}_3$ formulas. The following equalities hold:*

$$\begin{aligned}
[\![\neg\psi]\!]_v &= [\![\psi]\!]_{\neg v}, \textit{ if } v \in \mathbf{3}\\
[\![\psi \wedge \psi']\!]_{\mathsf{t}} &= [\![\psi]\!]_{\mathsf{t}} \bowtie [\![\psi']\!]_{\mathsf{t}}\\
[\![\psi \wedge \psi']\!]_{\mathsf{f}} &= [\![\psi]\!]_{\mathsf{f}} \cup [\![\psi']\!]_{\mathsf{f}}, \textit{ if } fv(\psi) = fv(\psi')\\
[\![\psi \wedge \psi']\!]_\bot &= [\![\psi]\!]_{\mathsf{t}} \bowtie [\![\psi']\!]_\bot \cup [\![\psi]\!]_\bot \bowtie [\![\psi']\!]_{\mathsf{t}} \cup [\![\psi]\!]_\bot \bowtie [\![\psi']\!]_\bot\\
[\![\psi \otimes \psi']\!]_b &= [\![\psi]\!]_b \bowtie [\![\psi']\!]_b, \textit{ if } b \in \{\mathsf{t},\mathsf{f}\}\\
[\![\psi \otimes \psi']\!]_\bot &= [\![\psi]\!]_\bot \bowtie [\![\psi']\!]_\bot \cup [\![\psi]\!]_{\mathsf{t}} \bowtie [\![\psi']\!]_{\mathsf{f}} \cup [\![\psi]\!]_{\mathsf{f}} \bowtie [\![\psi']\!]_{\mathsf{t}}\\
[\![\forall x.\,\psi]\!]_{\mathsf{t}} &= \emptyset, \textit{ if } [\![\psi]\!]_{\mathsf{t}} \textit{ is finite and } x \in fv(\psi)\\
[\![\forall x.\,\psi]\!]_{\mathsf{f}} &= \pi_x([\![\psi]\!]_{\mathsf{f}}), \textit{ if } x \in fv(\psi)\\
[\![\forall x.\,\psi]\!]_\bot &= \pi_x([\![\psi]\!]_\bot) \setminus \pi_x([\![\psi]\!]_{\mathsf{f}}), \textit{ if } x \in fv(\psi)
\end{aligned}$$

These equalities provide a method to compute, under the stated conditions, the relations $[\![\psi]\!]_v$ from the corresponding relations for ψ's direct sub-formulas.

For instance, if $\psi = \psi_1 \wedge \psi_2$ and $[\![\psi_1]\!]_\mathsf{t}$, $[\![\psi_2]\!]_\mathsf{t}$ are finite relations, then $[\![\psi]\!]_\mathsf{t}$ is a finite relation given by the join of the other two relations. Furthermore, when $[\![\psi_1]\!]_\mathsf{t}$ is finite, $[\![\psi_2]\!]_\mathsf{t}$ is cofinite, and $\mathit{fv}(\psi_2) \subseteq \mathit{fv}(\psi_1)$, then $[\![\psi]\!]_\mathsf{t}$ is a finite relation that we can compute as $[\![\psi_1]\!]_\mathsf{t} \bowtie [\![\psi_2]\!]_\mathsf{t} = [\![\psi_1]\!]_\mathsf{t} \bowtie \big(|\bar{\mathcal{D}}|^{|\mathit{fv}(\psi_2)|} \setminus ([\![\psi_2]\!]_\mathsf{f} \cup [\![\psi_2]\!]_\perp)\big)$. Note that the condition $\mathit{fv}(\psi_2) \subseteq \mathit{fv}(\psi_1)$ is essential, as otherwise $[\![\psi_1]\!]_\mathsf{t} \bowtie [\![\psi_2]\!]_\mathsf{t}$ may be infinite. For example, if $\bar{\mathit{fv}}(\psi_1) = (x)$ and $\bar{\mathit{fv}}(\psi_2) = (x, y)$ with $[\![\psi_1]\!]_\mathsf{t} = \{(1)\}$, $[\![\psi_2]\!]_\mathsf{f} = \{(1,2)\}$, and $[\![\psi_2]\!]_\perp = \{(3,4)\}$, then $[\![\psi]\!]_\mathsf{t} = \{1\} \times (|\bar{\mathcal{D}}| \setminus \{2,4\})$. The same method is applied to each of the other sub-cases of the binary connectives.

The described approach is implemented through the procedures $\mathsf{eval\_neg}$, $\mathsf{eval\_and}$, $\mathsf{eval\_times}$, and $\mathsf{eval\_forall}$, given in Figure 3. Each procedure returns a triple $(R_\mathsf{t}, R_\mathsf{f}, R_\perp)$, where R_v is a value computed based on the identities in Lemma 8 using the procedures join and union, which are given in the Appendix. The join procedure takes as arguments tuples (ψ, E) and (ψ', E'), and truth values v and v'. Provided that the invariants $\mathit{Inv}(\psi, \tau, E)$ and $\mathit{Inv}(\psi', \tau, E')$ are satisfied, the return value is either Fin $([\![\psi]\!]_v \bowtie [\![\psi']\!]_{v'})$ or None, depending on whether a finite relation can be computed. The union procedure has similar arguments and return values. The auxiliary procedures $\mathsf{update\_cofin}$ and update from Figure 3 handle the following corner case: If two elements of the newly formed triple $(R_\mathsf{t}, R_\mathsf{f}, R_\perp)$ are of the form $\mathsf{Fin}\ V$ and the remaining element is None, then $\mathsf{update\_cofin}(\psi, R_\mathsf{t}, R_\mathsf{f}, R_\perp)$ changes None to either CoFin if $\mathit{fv}(\varphi) \neq \emptyset$, or otherwise (when $\mathit{fv}(\varphi) = \emptyset$) to Fin $\{()\}$ or Fin $\emptyset$ depending on the truth value that should be returned. This ensures that the invariant Inv is preserved by the return value of the $\mathsf{eval\_and}$, $\mathsf{eval\_times}$, and $\mathsf{eval\_forall}$ procedures.

Finally, we consider the temporal operators. Let $\psi = \alpha\ \mathsf{S}_I\ \beta$. For efficiency, eval maintains between iterations a sequence L_ψ, which is initialized by the init procedure with the empty sequence. The sequence L_ψ contains values $E_{\tau'}$ that satisfy the invariant $\mathit{Inv}(\alpha\ \mathsf{S}_{[\delta,\delta]}\ \beta, \tau, E_{\tau'})$, where $\delta = \tau - \tau'$ and τ' is such that $0 \leq \tau - \tau' < b$, with $I = [a, b)$. In this way, the sub-formulas α and β are not re-evaluated at previous time points τ'. Instead, the result of their evaluation is stored in L_ψ. The return value is computed by iteratively calling $\mathsf{eval\_or}$ on the elements $E_{\tau'}$ of L_ψ for which $(\tau - \tau') \in I$. This last step reflects the equivalence between $\alpha\,\mathsf{S}_I\,\beta$ and $\bigvee_{\delta \in I} \alpha\,\mathsf{S}_{[\delta,\delta]}\,\beta$. Given two formulas ψ_1 and ψ_2 and two values E_1 and E_2 satisfying respectively the invariants $\mathit{Inv}(\psi_1, \tau, E_1)$ and $\mathit{Inv}(\psi_2, \tau, E_2)$, the procedure $\mathsf{eval\_or}$ returns a value E that satisfies $\mathit{Inv}(\psi_1 \vee \psi_2, \tau, E)$.

The case for Until is analogous to Since. The only significant difference is that the procedure must delay its answer until all relevant events have occurred. Various optimizations, which we mention in Appendix A, can further improve the efficiency of handling temporal operators.

The following theorem establishes termination and soundness of our algorithm. To state it formally, we first explicitly define the relationship between the arguments Γ_τ of the eval procedure, and the logging structures $\mathcal{D}_\tau$ of $\bar{\mathcal{D}}$. We let

$$\mathit{triples}(\mathcal{D}_\tau) := \left\{\left(r, (\mathit{val}(r_\mathsf{t}^{\mathcal{D}_\tau}), \mathit{val}(r_\mathsf{f}^{\mathcal{D}_\tau}), \mathit{val}(r_\perp^{\mathcal{D}_\tau}))\right) \mid r \in R\right\},$$

where $\mathit{val}(V)$ is $\mathsf{Fin}\ V$ if V is finite, and is CoFin otherwise. Thus $\Gamma_\tau = \mathit{triples}(\mathcal{D}_\tau)$.

Theorem 9. *Let $\bar{\mathcal{D}}$ be a logging knowledge base, φ a formula in $\mathcal{L}_3$, and $\tau \in \mathbb{N}$ a time point. The procedure* $\mathsf{eval}(\varphi,\ \Gamma_\tau,\ \tau)$ *returns a value E that satisfies the* *invariant* $Inv(\varphi, \tau, E)$*, whenever* $\mathsf{init}(\varphi)$*,* $\mathsf{eval}(\varphi,\ \Gamma_0,\ 0)$*, ...,* $\mathsf{eval}(\varphi,\ \Gamma_{\tau-1},\ \tau-1)$ *were called previously in this order, where* $\Gamma_{\tau'} = triples(\mathcal{D}_{\tau'})$*, for* $\tau' \leq \tau$*.*

A complete fragment. In general, our algorithm is incomplete. However, by limiting the usage of free variables, we obtain the fragment $\mathcal{L}_3^c$ for which we guarantee completeness.

Definition 10. *The set* $\mathcal{L}_3^c$ *of formulas is inductively defined:*

– $\mathsf{f} \in \mathcal{L}_3^c$ *and* $r(t_1, \ldots, t_{\iota(r)}) \in \mathcal{L}_3^c$*,*
– *if* $\varphi \in \mathcal{L}_3^c$*, then* $\neg\varphi \in \mathcal{L}_3^c$ *and* $\forall x.\, \varphi \in \mathcal{L}_3^c$*,*
– *if* $\varphi, \psi \in \mathcal{L}_3^c$ *and either* $fv(\varphi) = fv(\psi)$*,* $fv(\varphi) = \emptyset$*, or* $fv(\psi) = \emptyset$*, then* $\varphi \wedge \psi \in \mathcal{L}_3^c$*,* $\varphi \otimes \psi \in \mathcal{L}_3^c$*,* $\varphi \mathbin{\mathsf{S}_I} \psi \in \mathcal{L}_3^c$*, and* $\varphi \mathbin{\mathsf{U}_I} \psi \in \mathcal{L}_3^c$*.*

Note that $\mathcal{L}_3^c$ allows universal quantification and, by using $\neg$, also existential quantification of free variables, and both quantifiers can be nested freely. But if an $\mathcal{L}_3^c$ formula contains a sub-formula with no quantifiers and two or more predicates, they must have the same free variables. As all of $\mathcal{L}_3$'s connectives are retained and their application is not restricted, $\mathcal{L}_3^c$ can still express all monotonic finitary operators. However, they cannot be used as liberally as in $\mathcal{L}_3$.

The first and second policy examples in Section 3 fall within $\mathcal{L}_3^c$. However, due to the free-variable restriction, the following formula is not in $\mathcal{L}_3^c$:

$$\Box\, \forall s.\, \forall r.\, \forall m.\, send(s, r, m) \to \Diamond_I\, authorize(m)\,.$$

It says that all messages m, sent by s to r must be subsequently authorized. This is a typical compliance policy from the HIPAA Privacy Rule [1]. By pushing the quantification of s and r inside the antecedent, we obtain a formula in $\mathcal{L}_3^c$:

$$\Box\, \forall m.\, \big(\exists s.\, \exists r.\, send(s, r, m)\big) \to \Diamond_I\, authorize(m)\,.$$

One can check that evaluating $\forall x.\, \varphi \to \psi$ and $(\exists x.\, \varphi) \to \psi$, as well as $\exists x.\, \varphi \wedge \psi$ and $(\exists x.\, \varphi) \wedge \psi$, where $x \notin fv(\psi)$, over an arbitrary logging knowledge base and an arbitrary time point yields the same truth value.

It is not always possible to rewrite a formula such that the result falls into $\mathcal{L}_3^c$. Recall the third example (the separation-of-duty requirement) from Section 3. Clearly, it does not fall within $\mathcal{L}_3^c$. However, if there are finitely many datasets, we can partially ground the formula, obtaining a family of formulas $\varphi_{\mathrm{d},\mathrm{d}'}$, where d and d$'$ range over the datasets. Each is in $\mathcal{L}_3^c$ after similar rewriting as above:

$$\varphi_{\mathrm{d},\mathrm{d}'} := \Box\, \big(\exists s.\, (\exists o.\, access(s, o, \mathrm{d})) \wedge \exists o'.\, \blacklozenge\, access(s, o', \mathrm{d}')\big) \to \mathsf{C}_{[0,2)} \neg conflict(\mathrm{d}, \mathrm{d}')\,.$$

Syntactic rewriting and partial grounding cannot always be applied. Still, $\mathcal{L}_3^c$ is an expressive fragment that captures a wide-range of compliance policies.

Finally, we state our result on the algorithm's completeness on $\mathcal{L}_3^c$ formulas. To do so, we define the stronger invariant $Inv_c(\varphi, \tau, E)$ which, in addition to $Inv(\varphi, \tau, E)$, requires that there are $v', v'' \in \mathbf{3}$ with $v' \neq v''$ such that $S_{v'} = \mathsf{Fin}\ V'$ and $S_{v''} = \mathsf{Fin}\ V''$ for some sets V', V'', where $E = (S_{\mathsf{t}}, S_{\mathsf{f}}, S_{\perp})$.

Theorem 11. *Let* $\bar{\mathcal{D}}$ *be a logging knowledge base,* φ *a formula in* $\mathcal{L}_3^c$*, and* $\tau \in \mathbb{N}$ *a time point. The procedure* $\mathsf{eval}(\varphi,\ \Gamma_\tau,\ \tau)$ *returns a value* E *that satisfies the invariant* $Inv_c(\varphi, \tau, E)$*, whenever* $\mathsf{init}(\varphi)$*,* $\mathsf{eval}(\varphi,\ \Gamma_0,\ 0)$*, ...,* $\mathsf{eval}(\varphi,\ \Gamma_{\tau-1},\ \tau-1)$ *were called previously in this order, where* $\Gamma_{\tau'} = triples(\mathcal{D}_{\tau'})$*, for* $\tau' \leq \tau$*.*

6 Related Work

The only work we are aware of that addresses the problem of compliance checking with incomplete knowledge is Garg et al. [21]. Their policy language is a restricted first-order logic. It has a more liberal usage of free variables compared to $\mathcal{L}_3^c$, but it does not consider $\perp$ at the object-level and cannot express the $\otimes$ operator. They adopt a weaker logging assumption, whereby a finite or an infinite number of event occurrences can be unknown. However, their compliance algorithm is not suitable for on-line monitoring and, more importantly, it is incomplete, even with our logging assumption. Recall our first policy example in Section 3. If the web-server's logger crashes and there are no denials, their algorithm does not report that there are no violations. Instead, it wrongly reports that there may be potential violations, where in fact there are none. Similarly, it may also fail to report violations. For example, given a specification of the form

$$\Box \forall \bar{x}.\, c(\bar{x}) \to \exists \bar{y}.\, c'(\bar{x}, \bar{y}) \land \forall \bar{z}.\, \varphi(\bar{x}, \bar{y}, \bar{z})\,,$$

then all $\bar{x}$ that violate the policy by making c true and φ false, but for which all c' events are missing, are not reported. This is because their algorithm evaluates formulas in a top-down fashion: it first finds all $\bar{x}$ that satisfy c, then it partially grounds[2] the consequent, then it finds all $\bar{y}$ that satisfy c', and then partially grounds φ, and so forth. However, if there are no partial groundings, the algorithm stops further evaluations. In contrast, since our algorithm works in a bottom-up fashion, it does not have this problem.

The problem of incompleteness and disagreements is also present in other fields, and some approaches there are also based on many-valued logics. Some access-control policy languages [15, 18] use multiple truth values to represent different access-control decisions. These languages are propositional and do not support temporal reasoning. Several model-checking approaches [13, 14, 16] also consider a many-valued truth space. However, their many-valued semantics do not guarantee policy-compliance monotonicity. Furthermore, their specification languages only have the classical Boolean and temporal connectives.

Bauer et al. [8] extend the classical LTL semantics by also assigning non-Boolean truth values to finite and complete prefixes of infinite traces. Their semantics differentiate whether all or some extensions of a finite trace satisfy a property. However, the Boolean and temporal operators are not extended over the additional truth values. Furthermore, they do not consider the ordering $\leq_k$ of the truth values in knowledge.

Another approach to dealing with incompleteness is to make quantitative statements, e.g., how certain it is whether a property is violated. Stoller et al. [26] present such an approach for monitoring traces with gaps. Their solution first assigns probabilities to whether events happened during gaps, and then computes the overall probability that a temporal property is violated. This solution is orthogonal to ours. It requires a reliable training set to derive appropriate probability assignments for different event occurrences.

[2] Their logging assumption and language restrictions guarantee that there are always only finitely many satisfying ground instances.

7 Conclusions

In complex IT systems, logging failures happen and knowledge about the occurrence of system actions is incomplete when monitoring the system. Furthermore, system components can disagree on whether actions took place. Approaches for checking system compliance based on the classical Boolean setting are insufficient since they may incorrectly report policy violations. A three-valued truth space allows us to correctly distinguish between violations and potential violations. The solution presented in this paper carefully adopts a three-value truth space so that policy evaluations are correct regardless of how knowledge gaps are resolved. The presented monitoring algorithm shows that policy violations and potential violations can be soundly and completely determined.

As future work we will investigate how to efficiently resolve potential violations as prior knowledge gaps are incrementally resolved. We also plan case studies to evaluate our monitoring algorithm in real-world settings. Finally, we would like to explore different truth spaces to distinguish between different kinds of knowledge gaps and disagreements.

Acknowledgments. We thank Germano Caronni and Matúš Harvan for fruitful discussions on this topic.

References

1. The Health Insurance Portability and Accountability Act of 1996 (HIPAA), 1996. Public Law 104-191.
2. Gramm-Leach-Bliley Act of 1999 (GLBA), 1999. Public Law 106-102.
3. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
4. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *J. Aero. Comput. Inform. Comm.*, 7:365–390, 2010.
5. D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. Monitoring usage-control policies in distributed systems. In *Proceedings of the 18th International Symposium on Temporal Representation and Reasoning (TIME)*, pages 88–95. IEEE Computer Society, 2011.
6. D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. MONPOLY: Monitoring usage-control policies. In *Proceedings of the 2nd International Conference on Runtime Verifivation (RV)*, volume 7186 of *Lect. Notes Comput. Sci.*, pages 360–364. Springer, 2012.
7. D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proceedings of the 28th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–60. Schloss Dagstuhl - Leibniz Center for Informatics, 2008.
8. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Logic Comput.*, 20(3):651–674, 2010.
9. N. D. Belnap, Jr. A useful four-valued logic. In J. M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-Valued Logic*, volume 2 of *Episteme*, pages 7–37. D. Reidel Publishing Company, 1977.

10. S. Blamey. Partial logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 5, pages 261–353. Kluwer Academic Publishers, 2002.
11. G. Boella, J. Broersen, and L. van der Torre. Reasoning about constitutive norms, counts-as conditionals, institutions, deadlines and violations. In *Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA)*, volume 5357 of *Lect. Notes Comput. Sci.*, pages 86–97. Springer, 2008.
12. J. Broersen. On the logic of 'being motivated to achieve ϱ, before δ'. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 3229 of *Lect. Notes Comput. Sci.*, pages 334–346. Springer, 2004.
13. G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *Lect. Notes Comput. Sci.*, pages 274–287. Springer, 1999.
14. G. Bruns and P. Godefroid. Model checking with multi-valued logics. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lect. Notes Comput. Sci.*, pages 281–293. Springer, 2004.
15. G. Bruns and M. Huth. Access control via Belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inform. Syst. Secur.*, 14(1), 2011.
16. M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Meth.*, 12(4):371–408, 2003.
17. J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.
18. J. Crampton and C. Morisset. PTaCL: A language for attribute-based access control in open systems. In *Proceedings of the 1st Conference on Principles of Security and Trust (POST)*, volume 7215 of *Lect. Notes Comput. Sci.*, pages 390–409. Springer, 2012.
19. N. Dinesh, A. K. Joshi, I. Lee, and O. Sokolsky. Checking traces for regulatory conformance. In *Proceedings of the 8th International Workshop on Runtime Verification (RV)*, volume 5289 of *Lect. Notes Comput. Sci.*, pages 86–103. Springer, 2008.
20. M. Fitting. Kleene's logic, generalized. *J. Log. Comput.*, 1(6):797–810, 1991.
21. D. Garg, L. Jia, and A. Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 151–162. ACM Press, 2011.
22. A. Groce, K. Havelund, and M. Smith. From scripts to specification: The evaluation of a flight testing effort. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, volume 2, pages 129–138. ACM Press, 2010.
23. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Trans. Serv. Comput.*, 5(2):192–206, 2012.
24. T. Hvitved, F. Klaedtke, and E. Zălinescu. A trace-based model for multiparty contracts. *J. Log. Algebr. Program.*, 81(2):72–98, 2012.
25. S. C. Kleene. *Introduction to Metamathematics.* D. Van Nostrand, Princeton, 1950.
26. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Proceedings of the 2nd International Conference on Runtime Verification (RV)*, volume 7186 of *Lect. Notes Comput. Sci.*, pages 193–207. Springer, 2012.

A Additional Algorithmic Details and Proof Details

The eval procedure and proof details of Theorem 9. We proceed with a more detailed description of our monitoring algorithm. The procedure eval, given in Figure 2, uses various sub-procedures, given in the Figures 4–6.

Preliminaries. In the pseudo-code we use the following notation.

- Case switching is expressed with the **case** and the (optional) **otherwise** constructions. The body of the case is executed whenever (and independently of the other cases) the case check passes. The body of the **otherwise** case is executed only if all other case checks have failed. A case check performs pattern matching where fresh variables are bound accordingly.
- The symbol _ denotes an arbitrary fresh variable.
- $\lambda\bar{x}.e$ denotes an unnamed procedure with the sequence $\bar{x}$ of arguments and with body e.
- $+\!\!+$ denotes list concatenation.
- The functions map and fold_left have standard interpretation from functional programming languages, such as ML.

For $I \in \mathbb{I}$ and $\tau \in \mathbb{N}$, we define $I - \tau := \{\tau' - \tau \mid \tau' \in I\} \cap \mathbb{N}$ and $\tau - I := \{\tau - \tau' \mid \tau' \in I\} \cap \mathbb{N}$. We denote the left and right margins of an interval I by $\ell(I)$ and $r(I)$ respectively. For instance, if $I = [2, 4)$, then $\ell(I) = 2$ and $r(I) = 3$, and if $I = (0, \infty)$, then $\ell(I) = 1$ and $r(I) = \infty$. We denote the logical equivalence between the formulas φ and ψ by $\varphi \equiv \psi$. Finally, $\mathsf{sf}(\varphi)$ denotes the set of all sub-formulas of φ.

Overview. Along with the presentation of the algorithm, we also explain how eval preserves the invariant *Inv*, thus proving Theorem 9. We fix a formula $\varphi \in \mathcal{L}_3$, a logging knowledge base $\bar{\mathcal{D}}$, and a time point τ, and we reason by induction using the lexicographic ordering of pairs $(\tau, |\varphi|)$, where $\tau \in \mathbb{N}$ and $|\varphi|$ denotes φ's size, defined as expected.

To handle temporal operators efficiently, the eval procedure maintains a sequence L_ψ for each temporal sub-formula[3] ψ of φ. By this, we avoid re-evaluating sub-formulas ψ at already seen time points. The procedure init initializes the sequences L_ψ with the empty sequence. Assume that $\psi = \alpha \,\mathsf{S}_I\, \beta$. At the end of the iteration τ, the sequence L_ψ consists of triples $(\gamma, E_{\tau'}, \tau')$ ordered increasingly by τ', where τ' appears in the sequence if and only if $\tau - \tau' \leq r(I)$, γ is β if $\tau' = \tau$ and ψ otherwise, and $E_{\tau'}$ satisfies the invariant $\mathit{Inv}(\alpha\ \mathsf{S}_{[\delta,\delta]}\ \beta, \tau, E_{\tau'})$, where $\delta = \tau - \tau'$. We consider this property of sequences L_ψ as part of the induction invariant needed to prove Theorem 9.

The eval procedure is called recursively over the sub-formulas of φ. In the following, we present the cases of the eval procedure.

Atomic formulas. The simplest case is when the sub-formula φ is the truth value f. In this case we return the corresponding triple $E :=$ (Fin $\emptyset$, Fin $\{()\}$, Fin $\emptyset$). The invariant $\mathit{Inv}(\mathsf{f}, \tau, E)$ trivially holds.

[3] Without loss of generality, we assume that a temporal sub-formula occurs only once in φ.

```
proc eval_predicate($\psi$, $(S_t, S_f, S_\perp)$)
  case $S_t$ = Fin $T$
    $(\bar{s}, G) \leftarrow$ get_predicate_constraints($\psi$)
    $R_t \leftarrow$ Fin $\pi_{\bar{s}}(\sigma_G(T))$
    return update_cofin($\psi$, $R_t$, $S_f$, $S_\perp$)
```

Fig. 4. The eval_predicate procedure.

When φ is of the form $r(\bar{t})$ for some predicate symbol r, we first retrieve the value $E_r = (S_\mathsf{t}, S_\mathsf{f}, S_\perp)$ associated with r from the set Γ of pairs, using the procedure get_value for which we omit the pseudo-code. Then, by calling eval_predicate(φ, E_r), given in Figure 4, we retrieve the sets $[\![r(\bar{t})]\!]_v^{\bar{\mathcal{D}},\tau}$ from $r_v^{\mathcal{D}_\tau}$, for each $v \in \mathbf{3}$, by filtering the relations $r_v^{\mathcal{D}_\tau}$ according to the implicit constraints present in the sequence $\bar{t}$ of constants and variables. For instance, if $\iota(r) = 3$, $r_\mathsf{t}^{\mathcal{D}_0} = \{(1,1,1),(1,1,2),(1,2,3)\}$, and $r(\bar{t}) = r(x,x,2)$, then $[\![r(\bar{t})]\!]_\mathsf{t}^{\bar{\mathcal{D}},0} = \{(1,1,2)\}$. Here the constraints are that the first and second fields of triples in $r_\mathsf{t}^{\mathcal{D}_\tau}$ are equal, and the third field equals 2. Such constraints are formalized next using the relational algebra operators *projection* and *selection*.

Let $A \subseteq |\bar{\mathcal{D}}|^n$ be an n-ary relation, with $n \geq 0$. An index $i \in \mathbb{N}$ is a *column* in A if $1 \leq i \leq n$. Given a vector $\bar{s} = (i_1, i_2, \ldots, i_k)$ of k columns in A, with $k \geq 0$, the *projection* of A on $\bar{s}$ is the k-ary relation:

$$\pi_{\bar{s}}(A) := \{(d_{i_1}, d_{i_2}, \ldots, d_{i_k}) \in |\bar{\mathcal{D}}|^k \mid (d_1, d_2, \ldots, d_n) \in A\}.$$

Given a set G of pairs of the form (i,j) with i and j columns in A, and of the form (i,d) where i is a column in A and $d \in |\bar{\mathcal{D}}|$, then the *selection* on A with respect to G is the defined as the n-ary relation:

$$\sigma_G(A) := \{(d_1, \ldots, d_n) \in A \mid d_i = d_j \text{ for any } (i,j) \in G,\ d_i = d \text{ for any } (i,d) \in G\}.$$

The pairs in G are called *selection constraints* and are denoted $i = j$ or $i = c$. For convenience, we call the pair $(\bar{s}, G)$ a *constraint*.

Given a formula $\varphi = r(\bar{t})$, the projection columns $\bar{s}$ and the selection constraints G determined by the sequence $\bar{t}$ are computed by the procedure get_predicate_constraints(φ). We omit its straightforward pseudo-code description. Next, we filter the tuples from $p_\mathsf{t}^{\mathcal{D}_\tau}$ according to the constraint $(\bar{s}, G)$ and store the result in R_t. Note that, by Assumption 1, we have $r_v^{\mathcal{D}_\tau} = \emptyset$ and $r_{v'}^{\mathcal{D}_\tau}$ is infinite, for $\{v, v'\} = \{\mathsf{f}, \perp\}$. Hence we have that $S_v = \mathsf{Fin}\ \emptyset$ and $S_{v'} = \mathsf{CoFin}$. Thus, if φ has free variables we simply return the triple $(R_\mathsf{t}, S_\mathsf{f}, S_\perp)$. However, if φ is a closed formula, then we first update $S_{v'}$. To do so, we use the procedure update_cofin, given in Figure 3. This procedure changes $S_{v'}$ to $\mathsf{Fin}\ \{()\}$ or to $\mathsf{Fin}\ \emptyset$, depending on whether $[\![r(\bar{t})]\!]_\mathsf{t}^{\bar{\mathcal{D}},\tau}$ is empty or not. The pseudo-code of the procedure handles a more general case, as it is also used as a sub-procedure by other cases of eval.

The returned triple satisfies the invariant *Inv*, because E_r satisfies $Inv(r(\bar{x}), \tau, E_r)$ by Assumption 1, and because $[\![r(\bar{t})]\!]_\mathsf{t}^{\bar{\mathcal{D}},\tau} = \pi_{\bar{s}}(\sigma_G(r_\mathsf{t}^{\mathcal{D}_\tau}))$, where $\bar{x}$ is a sequence of distinct variables of length $\iota(r)$.

Note that we have hitherto treated the two base cases our inductive proof.

```
proc join(($\psi$, ($S_t, S_f, S_\bot$)), ($\psi'$, ($S'_t, S'_f, S'_\bot$)), $v$, $v'$)
  case $S_v$ = Fin $\emptyset$ or $S'_{v'}$ = Fin $\emptyset$
    return Fin $\emptyset$
  $c \leftarrow$ get_join_constraints ($\psi$, $\psi'$)
  case $S_v$ = Fin $V$ and $S'_{v'}$ = Fin $V'$
    return Fin ($V \bowtie_c V'$)
  otherwise
    $\{a, b\} \leftarrow \mathbf{3} \setminus \{v\}$
    $\{a', b'\} \leftarrow \mathbf{3} \setminus \{v'\}$
    case $S_v$ = Fin $V$ and $S'_{a'}$ = Fin $A'$ and $S'_{b'}$ = Fin $B'$ and $fv(\psi') \subseteq fv(\psi)$
      return Fin ($V \rhd_c (A' \cup B')$)
    case $S'_{v'}$ = Fin $V'$ and $S_a$ = Fin $A$ and $S_b$ = Fin $B$ and $fv(\psi) \subseteq fv(\psi')$
      return Fin ($V' \rhd_c (A \cup B)$)
    otherwise
      return None

proc union(($\psi$, ($S_t, S_f, S_\bot$)), ($\psi'$, ($S'_t, S'_f, S'_\bot$)), $v$, $v'$)
  case $S_v$ = Fin $V$ and $S'_{v'}$ = Fin $V'$ and
      (($fv(\psi) = fv(\psi')$ or ($fv(\psi) \subseteq fv(\psi')$ and $V = \emptyset$) or ($fv(\psi') \subseteq fv(\psi)$ and $V' = \emptyset$))
    return Fin ($V \cup V'$)
  otherwise
    return None
```

Fig. 5. The join and union procedures.

Non-temporal connectives. As this case was explained in the body of the paper, here we just recall the formal definition of the relation algebra operator *join* [3]. For this case, we simplify notation and write $[\![\varphi]\!]_v$ instead of $[\![\varphi]\!]_v^{\bar{\mathcal{D}},\tau}$.

Given two relations $A \subseteq |\bar{\mathcal{D}}|^n$ and $B \subseteq |\bar{\mathcal{D}}|^k$, with $n, k \geq 0$, a vector of columns $\bar{s}$ in $A \times B$, and a set G of selection constraints on $A \times B$, we define the *join* of A and B with respect to $\bar{s}, G$ as $A \bowtie_{\bar{s},G} B := \pi_{\bar{s}}(\sigma_G(A \times B))$ and the *antijoin* as $A \rhd_{\bar{s},G} B := A \setminus (A \bowtie_{\bar{s},G} B)$. Note that tuples in A and in $A \bowtie_{\bar{s},G} B$ have the same arity iff $|\bar{s}| = n$.

Given a formula φ and a truth-value $v \in \mathbf{3}$, we see the set $[\![\varphi]\!]_v$ as a *named relation* where columns in $[\![\varphi]\!]_v$ are named by the free variables in $\bar{fv}(\varphi)$. The *natural join* of $[\![\varphi]\!]_v$ and $[\![\varphi']\!]_{v'}$ is the set $[\![\varphi]\!]_v \bowtie_{\bar{s},G} [\![\varphi']\!]_{v'}$ where the join constraint $(\bar{s}, G)$ is obtained as follows: G consists of the selection constraints $i = i'$ for which the variable at the position i in $\bar{fv}(\varphi)$ equals the variable at the position i' in $\bar{fv}(\varphi')$ and $\bar{s} = (1, 2, \ldots, n, n + j_1, n + j_2, \ldots, n + j_p)$ with $n = |fv(\varphi)|$, $p = |fv(\varphi') \setminus fv(\varphi)|$, and $(j_1, \ldots, j_p)$ is the maximal subsequence of $\bar{fv}(\varphi')$ such that each element does not appear in $fv(\varphi)$. We drop the subscript $(\bar{s}, G)$ and simply write $[\![\varphi]\!]_v \bowtie [\![\varphi']\!]_{v'}$ and $[\![\varphi]\!]_v \rhd [\![\varphi']\!]_{v'}$, because there is no risk of confusion, given that we do not use constraints other than these. In our algorithm, these constraints are determined by the procedure get_join_constraints(φ, φ'), for which we omit the pseudo-code. The join and union procedures, which implement the homonym operators, are given in Figure 5.

Temporal connectives. For brevity, we omit the case dealing with the Until operator, whose treatment is similar to the Since case. The sub-procedure handling the Since operator is eval_since, given in Figure 6. Its arguments are the tem-

```
proc eval_since($\varphi$, $\tau$, $E_\alpha$, $E_\beta$) // $\varphi = \alpha\, \mathsf{S}_I\, \beta$
  case $L_\varphi = (\gamma, E', \tau') :: L'$
    if $(\tau - \tau') \notin I$ then $L_\varphi \leftarrow L'$

  f_aux $\leftarrow \lambda(\gamma, E', \tau').\, (\varphi$, eval_and(($\gamma$, $E'$), ($\alpha$, $E_\alpha$)), $\tau'$)
  $L_\varphi \leftarrow$ map f_aux $L_\varphi$
  $L_\varphi \leftarrow L_\varphi$ ++ $(\beta, E_\beta, \tau)$
  if $0 \in I$ and $|L_\varphi| = 1$ then return $E_\beta$
  else $E_{id} \leftarrow$ update_cofin($\varphi$, Fin $\emptyset$, None, Fin $\emptyset$)
       return fold_left(aux_since, $E_{id}$, $L_\varphi$)

proc aux_since($E$, ($\gamma, E', \tau'$))
  if $(\tau - \tau') \notin I$ then return $E$
  else return eval_or(($\varphi$, $E$), ($\gamma$, $E'$))

proc eval_or(($\psi$, $E$), ($\psi'$, $E'$))
  return eval_neg(eval_and(($\psi$, eval_neg($E$)), ($\psi'$, eval_neg($E'$))))
```

Fig. 6. The eval_since procedure.

poral formula $\varphi = \alpha\, \mathsf{S}_I\, \beta$, the current time point τ, and the values E_α and E_β produced by the recursive calls to eval at τ for the sub-formulas α and respectively β. Intuitively, the evaluation of φ reflects the logical equivalences[4] $\alpha\, \mathsf{S}_I\, \beta \equiv (0 \in I \to \beta) \vee (\alpha \wedge \bullet(\alpha\, \mathsf{S}_{I-1}\, \beta))$ and $\alpha\, \mathsf{S}_I\, \beta \equiv \bigvee_{\delta \in I} \alpha\, \mathsf{S}_{[\delta,\delta]}\, \beta$, which also hold in the three-valued setting.

By the induction hypothesis, when we enter the procedure, the sequence L_ψ consists of triples $(\gamma, E_{\tau'}, \tau')$ ordered increasingly by τ', where τ' appears in the sequence if and only if $0 \leq \tau - 1 - \tau' \leq r(I)$, γ is β if $\tau' = \tau - 1$ and φ otherwise, and $E_{\tau'}$ satisfies the invariant $Inv(\alpha\, \mathsf{S}_{[\delta,\delta]}\, \beta, \tau - 1, E_{\tau'})$, where $\delta = \tau - 1 - \tau'$. The eval_since procedure updates the sequence L_φ in line with the first equivalence, and it uses L_φ to compute the return value in line with the second equivalence.

The eval_since procedure starts by removing the first element $(\gamma, E_{\tau'}, \tau')$ of L_φ if τ' fell out of the relevant time interval, i.e. $\tau - I$. Note that if I is bounded then there is only one such element, while if I is unbounded there is no such element. Next, the elements of the L_φ are updated by calling eval_and on (γ, E') and (α, E_α), for each element (γ, E', τ') of L_φ. This operation corresponds to the conjunction in the right-hand side of the first equivalence, while the next one, that is, appending (β, E_β, τ) to the end of L_φ roughly corresponds to the disjunction. At this point L_φ is updated, and for each triple $(\gamma, E_{\tau'}, \tau')$ in L_φ we have that $E_{\tau'}$ satisfies the invariant $Inv(\alpha\, \mathsf{S}_{[\delta,\delta]}\, \beta, \tau, E_{\tau'})$, where $\delta = \tau - \tau'$.

Finally, we iteratively call eval_or on (φ, E) and (φ, E') for each element (E', τ') of L_φ with $(\tau - \tau') \in I$, where E is the accumulated result of previous

[4] We abused notation here, as the right hand side is not a formula in $\mathcal{L}_3$. However, we note that $0 \in I$ can be written as $\mathsf{f}\, \mathsf{S}_I\, \mathsf{t}$, and $\bullet$ denotes the previous operator, which refers to the previous time-point (if it exists).

calls. This last step reflects the second equivalence. Note that E_{id} is the identity element with respect to the eval_or operation[5]. This completes the proof.

We also mention two optimizations that can be applied. First, note that when $r(I) = \infty$ then no elements are removed from the list L_φ. However, instead of storing in L_φ all elements $(\gamma, E_{\tau'}, \tau')$ with $\tau' \leq \tau$, it is sufficient to store those elements for which $\tau - \ell(I) < \tau' \leq \tau$. The length of the list is thus constant, i.e. $\ell(I)$, instead of constantly growing, i.e. $\tau + 1$. It is straightforward to update the constructions for this case. Second, when computing the return value at τ by iteratively calling aux_since, we could reuse some of the intermediary results computed at the previous iteration $\tau - 1$.

Proof details of Theorem 11. Let $\bar{\mathcal{D}}$ be a logging knowledge base, φ a formula in $\mathcal{L}_3^c$, and $\tau \in \mathbb{N}$ a time point. Let $\Gamma_{\tau'} = \mathit{triples}(\mathcal{D}_{\tau'})$, for $\tau' \leq \tau$. We reason again by induction on the pairs $(\tau, |\varphi|)$ ordered lexicographically. By Theorem 9, the call to $\mathsf{eval}(\varphi,\ \Gamma_\tau,\ \tau)$ returns a value E that satisfies the invariant $\mathit{Inv}(\varphi, \tau, E)$. Thus it suffices to show that there are two elements in the triple E which represent finite relations.

- $\varphi = \mathsf{f}$. The return value E clearly satisfies the invariant $\mathit{Inv}_c(\varphi, \tau, E)$.
- $\varphi = r(\bar{t})$. Then E satisfies the invariant $\mathit{Inv}_c(\varphi, \tau, E)$ by Assumption 1.
- $\varphi = \neg\psi$ and $\psi \in \mathcal{L}_3^c$. By the induction hypothesis, the return value $E' = (S_\mathsf{t}, S_\mathsf{f}, S_\perp)$ of the call to $\mathsf{eval}(\psi,\ \Gamma_\tau,\ \tau)$ satisfies the invariant $\mathit{Inv}_c(\psi, \tau, E')$. Thus two of the three elements of E' represent finite relations. As $E = (S_\mathsf{f}, S_\mathsf{t}, S_\perp)$, clearly the same holds for E.
- $\varphi = \forall\bar{x}.\psi$ and $\psi \in \mathcal{L}_3^c$. Let $E' = (S_\mathsf{t}, S_\mathsf{f}, S_\perp)$ be the return value of the call to $\mathsf{eval}(\psi,\ \Gamma_\tau,\ \tau)$. By the induction hypothesis, two of the three elements of E' represent finite relations. We easily see that the pseudo-code of eval_forall guarantees that $E = (R_\mathsf{t}, R_\mathsf{f}, R_\perp)$ is such that if S_v represents a finite relation, then R_v is a finite relation too, for any $v \in \mathbf{3}$. Thus E satisfies the invariant $\mathit{Inv}_c(\varphi, \tau, E)$.
- $\varphi = \psi_1 \wedge \psi_2$, with $\mathit{fv}(\psi_1) = \mathit{fv}(\psi_2)$ or $\mathit{fv}(\psi_1) = \emptyset$ or $\mathit{fv}(\psi_2) = \emptyset$.
  Let $E_i = (S^i_\mathsf{t}, S^i_\mathsf{f}, S^i_\perp)$ be value returned by the call to $\mathsf{eval}(\psi_i,\ \Gamma_\tau,\ \tau)$, for $i \in \{1, 2\}$. By the induction hypothesis, 4 values S^1_u, $S^1_{u'}$, S^2_v, $S^2_{v'}$ (out of 6) represent finite relations, for some $u, u', v, v' \in \mathbf{3}$ with $u \neq u'$, $v \neq v'$. Let $E = (R_\mathsf{t}, R_\mathsf{f}, R_\perp)$.
  Suppose first that $\mathit{fv}(\psi_1) = \mathit{fv}(\psi_2)$. We distinguish the following cases, up to symmetry:
  - $\{u, u'\} = \{\mathsf{t}, \mathsf{f}\}$, $\{v, v'\} = \{\mathsf{t}, \mathsf{f}\}$. Then R_t and R_f represent finite sets.
  - $\{u, u'\} = \{\mathsf{t}, \mathsf{f}\}$, $\{v, v'\} = \{\mathsf{t}, \perp\}$. Then R_t and $R_\perp$ represent finite sets.
  - $\{u, u'\} = \{\mathsf{t}, \mathsf{f}\}$, $\{v, v'\} = \{\mathsf{f}, \perp\}$. Then R_t and R_f represent finite sets.
  - $\{u, u'\} = \{\mathsf{t}, \perp\}$, $\{v, v'\} = \{\mathsf{t}, \perp\}$. Then R_t and $R_\perp$ represent finite sets.
  - $\{u, u'\} = \{\mathsf{t}, \perp\}$, $\{v, v'\} = \{\mathsf{f}, \perp\}$. Then R_t and $R_\perp$ represent finite sets.

[5] The special handling of the case where $0 \in I$ and $|L_\varphi| = 1$ is necessary, because otherwise the only call to eval_or should be $\mathsf{eval\_or}((\beta,\ E'_{id}),\ (\beta,\ E_\beta))$, instead of $\mathsf{eval\_or}((\varphi,\ E_{id}),\ (\beta,\ E_\beta))$, where $E'_{id} = \mathsf{update\_cofin}(\beta,\ \mathsf{Fin}\ \emptyset,\ \mathsf{None},\ \mathrm{Fin}\ \emptyset)$.

- $\{u, u'\} = \{\mathsf{f}, \bot\}$, $\{v, v'\} = \{\mathsf{f}, \bot\}$. Then R_f and $R_\bot$ represent finite sets.

  Suppose now that $fv(\psi_1) = \emptyset$ (the other case, $fv(\psi_2) = \emptyset$, is symmetric). Then, by the induction hypothesis, two elements of E_1 are finite. As update_cofin is always called, the third element is also finite. We distinguish the following cases:
  - $E_1 = (\mathsf{Fin}\ \{()\},\ \mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \emptyset)$. We easily verify that if S_v represents a finite relation, then R_v also represents a finite relation.
  - $E_1 = (\mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \{()\},\ \mathsf{Fin}\ \emptyset)$. We easily verify that $R_\mathsf{t} = R_\bot = \mathsf{Fin}\ \emptyset$.
  - $E_1 = (\mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \emptyset,\ \mathsf{Fin}\ \{()\})$. We easily verify that $R_\mathsf{t} = \mathsf{Fin}\ \emptyset$ and that if S_v represents a finite relation, then R_v also represents a finite relation, for $v \in \{\mathsf{f}, \bot\}$.

– $\varphi = \psi \otimes \psi'$. This case is similar to the previous one.

– $\varphi = \psi\ \mathsf{S}_I\ \psi'$. First note that, by the induction hypothesis, for elements $(\gamma, E_{\tau'}, \tau')$ of L_φ, the values $E_{\tau'}$ satisfy the invariant $Inv_c(\psi', \tau, E_{\tau'})$. By (an inner) induction on the length of L_φ we can easily prove that the elements of the updated list L_φ obtained after calling map f_aux L_φ still satisfy the invariant. Let K be the subsequence of L_φ such that $(\tau - \tau') \in I$. That is, K consists of those elements of L_φ on which the procedure eval_or is applied. Let E_n be the result after n calls to aux_since. We have $E_0 = E_{id}$ and $E = E_{|K|}$. We prove by (another inner) induction on the length of K that the triple E_n is such that two of its elements represent finite relations. The base case, when $n = 0$, is trivial. In the inductive case, when $n > 0$, the value E_{n+1} is obtained by calling eval_or on E_n and $E_{\tau'}$. As eval_or only calls eval_neg and eval_and, which we have already analyzed, and E_n and $E_{\tau'}$ satisfy the property by the inner and respectively the outer induction hypothesis, it follows that E_{n+1} also satisfies the property. Hence E satisfies the invariant $Inv_c(\varphi, \tau, E)$.

– $\varphi = \psi\ \mathsf{U}_I\ \psi'$. This case is similar to the previous one.