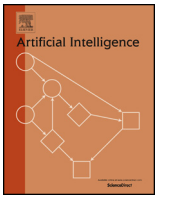
Artificial Intelligence 306 (2022) 103668

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



The delay and window size problems in rule-based stream reasoning ☆

Alessandro Ronca [a,b,*], Mark Kaminski [a], Bernardo Cuenca Grau [a], Ian Horrocks [a]

[a] Department of Computer Science, University of Oxford, UK
[b] DIAG, Sapienza Università di Roma, Italy

ARTICLE INFO

Article history:
Received 21 August 2019
Received in revised form 23 August 2021
Accepted 23 January 2022
Available online 31 January 2022

Keywords:
Knowledge representation and reasoning
Ontologies
Temporal reasoning
Stream reasoning
Stream processing

ABSTRACT

In recent years, there has been an increasing interest in extending stream processing engines with rule-based temporal reasoning capabilities. To ensure correctness, such systems must be able to output results over the partial data received so far as if the entire (infinite) stream had been available; furthermore, these results must be streamed out as soon as the relevant data is received, thus incurring the minimum possible delay; finally, due to memory limitations, systems can only keep a limited history of previous facts in memory to perform further computations. These requirements pose significant theoretical and practical challenges since temporal rules can derive new information and propagate it both towards past and future time points; as a result, streamed answers can depend on data that has not yet been received, as well as on data that arrived far in the past. Towards developing a solid foundation for practical rule-based stream reasoning, we propose and study in this paper a suite of decision problems that can be exploited by stream reasoning algorithms to tackle the aforementioned challenges, and provide tight complexity bounds for a core temporal extension of Datalog. All of the problems we consider can be solved at design time (under reasonable assumptions), prior to the processing of any data. Solving these problems enables the use of reasoning algorithms that process the input streams incrementally using a sliding window, while at the same time supporting an expressive rule-based knowledge representation language and minimising both latency and memory consumption.

1. Introduction

Query processing over data streams is a key aspect of Big Data applications. For instance, algorithmic trading relies on real-time analysis of stock tickers and financial news items [3], oil and gas companies continuously monitor and analyse data coming from their wellsites in order to detect equipment malfunction and predict maintenance needs [4], and network providers perform real-time analysis of network flow data to identify traffic anomalies and DoS attacks [5].

In stream processing, an input data stream is typically seen as an unbounded, append-only, relation of timestamped tuples, where timestamps are either added by the external device that issued the tuple or by the stream management

☆ This paper is based on and extends work presented at the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018) [1] and at the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018) [2].

* Corresponding author at: DIAG, Sapienza Università di Roma, Italy. Most of the work was carried out while the author was at the University of Oxford.
E-mail addresses: ronca@diag.uniroma1.it (A. Ronca), mark.kaminski@cs.ox.ac.uk (M. Kaminski), bernardo.cuenca.grau@cs.ox.ac.uk (B. Cuenca Grau), ian.horrocks@cs.ox.ac.uk (I. Horrocks).

https://doi.org/10.1016/j.artint.2022.103668
0004-3702/

system receiving it [6,7]. Data is only available for processing in a single pass and information stored by the system is thus inherently incomplete. Streaming jobs are long-running: standing queries are deployed once and continue to produce results as a stream until removed. Most applications of stream processing require near real-time analysis using limited resources, which poses significant challenges to stream management systems. On the one hand, to ensure correctness, systems must be able to compute query answers over the partial data received so far as if the entire (infinite) stream had been available; furthermore, they must stream query answers out as soon as the relevant data is received, thus incurring the minimum possible delay. On the other hand, due to memory limitations, systems can only keep a limited *history* of previously received input facts in memory to perform computations. These challenges have been addressed by implementing various extensions of traditional database query languages with window constructs, which declaratively specify the finite part of the input stream relevant to the answers at the current time [8].

A growing body of research has recently focused on extending stream management systems with reasoning capabilities [1,9–16]. Languages well-suited for stream reasoning applications can be formalised as temporal extensions of Datalog [17,18]—a prominent language with a rich tradition in the database and knowledge representation communities, which is frequently being used in advanced applications that mix AI and data management techniques. Datalog programs can be used to represent in a succinct and declarative way domain knowledge as 'if-then' rules where it holds that, if all atoms in the antecedent of a rule are satisfied, then the atom in the consequent part of the rule must also be satisfied. An important feature of Datalog rules is that they can be recursive: the application of a rule can (possibly indirectly) trigger an application of the same rule.

Well-known temporal extensions of Datalog include Datalog_{1S} [19] and *DatalogMTL* [20]. In this paper, we consider a core temporal rule-based language, which we call *Temporal Datalog*. Our language is an extension of negation-free Datalog where predicates are equipped with a special time sort interpreted over the non-negative integers, and allowing for terms of the form $t + k$ with t a time variable and k an integer. Under unary coding of integers, our language can be seen as a syntactic fragment of Datalog_{1S} as defined by Chomicki and Imieliński [19]. In particular, in contrast to Datalog_{1S}, the language we consider imposes a *guardedness* condition on time arguments ensuring that rule application to a stream is localised, in the sense that facts matching a rule's antecedent cannot be arbitrarily far apart in the stream. Our language is expressive enough to capture prominent temporal formalisms [21,22], and the guardedness condition makes it naturally well-suited for incremental stream processing [13,15].[1] The following examples illustrate the use of Temporal Datalog rules in streaming applications.

Example 1.1. Consider a computer network which is being monitored for external threats using an intrusion detection policy (IDP). Bursts (unusually high amounts of data) between any pair of nodes in the network are detected by specialised monitoring devices and streamed to the network's management centre as timestamped facts. A monitoring task in the centre is to identify nodes that may have been hacked according to a specific IDP, and add them to a blacklist of nodes. In this setting, one may want to know the contents of the blacklist at any given point in time in order to decide on further action. This task is captured by a Temporal Datalog program consisting of the rules given next:

$$Burst(x, y, t) \wedge Burst(z, y, t+1) \rightarrow Attack(x, y, t+1) \tag{1}$$

$$Attack(x, y, t) \wedge Attack(x, y, t+1) \wedge Attack(x, y, t+2) \rightarrow Black(x, t+2) \tag{2}$$

$$Black(x, t) \rightarrow Black(x, t+1) \tag{3}$$

$$Attack(x, y, t) \rightarrow Grey(x, l3, t) \tag{4}$$

$$Grey(x, l3, t) \rightarrow Grey(x, l2, t+1) \tag{5}$$

$$Grey(x, l2, t) \rightarrow Grey(x, l1, t+1) \tag{6}$$

$$Grey(x, l, t) \wedge Burst(x, y, t) \rightarrow Black(x, t). \tag{7}$$

Rule (1) identifies two consecutive bursts from nodes x and z to a node y in the network as an attack on y originated by x. Rule (2) implements an IDP where three consecutive attacks from x on y result in x being added to the blacklist, where it remains indefinitely (Rule (3)). Rules (4)–(7) implement a second IDP where an attack from x on any node leads to x being identified as suspicious and added to a "greylist". Such a list comes with a succession of three decreasing warning levels, where the maximum is represented by the constant $l3$. As time goes by, the warning level decreases; however, if at any point during this process node x generates another burst to any other node in the network, then it gets blacklisted.

Example 1.2. Consider the management of a wind farm in the North Sea. Each turbine is equipped with a sensor, which continuously records temperature levels of key devices within the turbine and sends those readings to a data centre monitoring the functioning of the turbines. Temperature levels are streamed by sensors using a ternary predicate *Temperature*, whose

[1] As we will discuss in Section 3, the reasoning problems we propose in this paper can be formulated and studied for other temporal rule-based languages.

arguments identify the device, the temperature level, and the time of the reading. A monitoring task in the data centre is to track the activation of cooling measures in each turbine, record temperature-induced malfunctions and shutdowns, and identify parts at risk of future malfunction. This task is captured by the following set of rules:

$$Temperature(x, high, t) \rightarrow Flag(x, t) \tag{8}$$

$$Flag(x, t) \wedge Flag(x, t+1) \rightarrow Cool(x, t+1) \tag{9}$$

$$Cool(x, t) \wedge Flag(x, t+1) \rightarrow Shutdown(x, t+1) \tag{10}$$

$$Shutdown(x, t) \rightarrow Malfunction(x, t-2) \tag{11}$$

$$Shutdown(x, t) \wedge Near(x, y, t) \rightarrow AtRisk(y, t) \tag{12}$$

$$AtRisk(x, t) \rightarrow AtRisk(x, t+1). \tag{13}$$

Rule (8) 'flags' a device whenever a high temperature reading is received. Rule (9) says that two consecutive flags on a device trigger cooling measures. Rule (10) says that an additional consecutive flag after activating cooling measures triggers a pre-emptive shutdown. By Rule (11), a shutdown is due to a malfunction that occurred when the first flag leading to shutdown was detected. Finally, Rules (12) and (13) identify devices located near a shutdown device as being at risk at the current time point as well as at all future time points.

As already mentioned, stream processing applications require near real-time response using limited resources. This becomes especially challenging in the context of rule-based stream reasoning since, as seen in our examples, rules may derive new information and recursively propagate it both towards past and future time points. As a result, the output of a Temporal Datalog program at a particular time point τ can depend on data that has not yet been received, as well as on data that arrived far back in the past. This can critically handicap the design of a real-time stream reasoning system, due to the following reasons.

- The system may not be able to determine whether all facts for time point τ have already been derived, thus introducing a (potentially unbounded) *delay* on applications that rely on the availability of all such facts.
- The system may be forced to store a (potentially unbounded) input history to ensure correctness, thus precluding the use of efficient stream processing techniques based on a *sliding window*.

Towards developing a solid foundation for practical rule-based stream reasoning, we propose and study in this paper a suite of decision problems that can be exploited by stream reasoning algorithms to tackle the aforementioned challenges.

We take as a starting point a generic stream reasoning algorithm, which we describe in Section 3. The algorithm extends traditional stream processing techniques based on a sliding window to the stream reasoning setting while, at the same time, abstracting away from all implementation details. The algorithm is parametrised with a Temporal Datalog program Π, a delay d, and a window size w. It accepts as input a stream S of timestamped facts and outputs as a stream a subset of the logical consequences of $\Pi \cup S$, which is determined by the values of the parameters d and w. As the algorithm receives and stores in memory the input stream at time point τ, it computes all implicit facts holding at $\tau - d$ using only the facts currently held in memory. The computed facts are first added to the memory and subsequently streamed as part of the output. Finally, the algorithm updates the memory by discarding all facts holding at $\tau - w$.

For the algorithm to be correct, the output stream must include all logical consequences of $\Pi \cup S$, for any input stream S. Such an assurance, however, can only be given for certain values of the delay and window size parameters. Hence, we formally define the notions of a valid delay d and a valid window size w as properties of the program Π, which are independent from the input stream S, and show that the algorithm parametrised with Π, d and w is correct if and only if d and w are valid.

There are, however, programs for which a valid delay does not exist, and hence for which our algorithm is not applicable. We show in Section 3 that, if a program does admit a valid delay, then it must admit a valid window size as well. Thus, the computational task of interest in our setting is to first check whether a program Π admits a valid delay and, if it does, to then compute the corresponding minimal delay and window size values. Reducing the delay and window size is important in practice, as it ensures that a correct algorithm will keep to a minimum the number of facts stored in memory at any point in time, and will minimise latency by returning each output fact as early as possible. We therefore study the computational properties of the following decision problems for a given input program Π.

- *Delay existence*: Decide whether there exists a valid delay for Π.
- *Delay validity*: Decide whether a given $d \geq 0$ is a valid delay for Π.
- *Window size validity*: Given a valid delay d for Π and $w \geq d$, decide whether w is a valid window size for Π and d.

We then define the class of *backward-bounded programs*, which preclude temporal recursion towards past time points. This is a useful fragment of Temporal Datalog, which can be recognised in polynomial time, and where programs are guaranteed to admit a valid delay of size at most linear in the size of the program.

In Section 4, we explore the limitations of our approach and establish a number of undecidability results. For this, we show that the decision problems we consider are closely related to *program containment*—a fundamental problem in static analysis and query optimisation, which is well-known to be undecidable already for non-temporal Datalog programs. To regain decidability, we consider the situation where the set of domain objects that can occur in a stream is fixed in advance. In Example 1.1 this assumption amounts to fixing the nodes in the network, and in Example 1.2 it amounts to fixing the set of sensors generating temperature readings. This is a rather mild assumption in practice; it is appropriate, for instance, in applications where the set of data-generating devices remains fixed during a query session—or more generally, where we can establish upfront an upper bound to the number of such devices. It allows us to ground the object variables of the program to a set of known objects; such grounding is exponential and results in an *object-ground* program where all variables are time variables.

In Section 5 we characterise delay existence, delay validity and program containment in terms of languages over finite words. In Sections 6 and 7 we exploit this characterisation to show, using automata-based techniques, that delay existence, delay validity, and window size validity are solvable in ExpSpace under the aforementioned assumption that the object domain of the input streams is fixed, and regardless of whether numbers in the input program are coded in unary or in binary; furthermore, the complexity of all problems drops to PSpace if we assume that object variables have been grounded and that numbers in rules are coded in unary. Finally, in Section 8 we show that these results are tight by providing matching lower bounds.

We believe that our results constitute a first step towards the development of robust and scalable stream reasoning engines with provable correctness guarantees. Although all of the problems we consider are computationally intractable, they can be solved "offline" at design time prior to the processing of any data. Solving these problems enables the use of reasoning algorithms that process the input streams incrementally using a sliding window while, at the same time, supporting an expressive rule-based knowledge representation language and minimising both latency and memory consumption.

This paper builds on some of the results presented in earlier conference publications [1,2]. In particular, our main technical contribution is to extend the stream reasoning framework and complexity results in [2] to include a much broader class of Temporal Datalog programs, and to consider also the complexity of all relevant problems under both unary and binary coding of numbers. A detailed discussion will be provided in Section 9.3.

2. Temporal Datalog

We recapitulate the syntax and semantics of Datalog with a time argument, which we call *Temporal Datalog*. We assume familiarity with the basics of complexity theory, logic, and rule-based languages for databases and knowledge representation.

As usual in First-order Logic, we assume mutually disjoint and countably-infinite sets of *predicates* (written P, Q, R, etc.), *objects* (written a, b, c, etc.), *object variables* (written x, y, z, etc.), and *time variables* (written t, t_1, t_2, etc.). Each predicate comes with a non-negative arity; as usual, we assume an infinite supply of predicates of each arity.

A *constant* is either an object or a *time point* $\tau \in \mathbb{N}$ (where $\mathbb{N}$ includes zero). An *object term* is an object or an object variable. A *time term* is a time point or an expression of the form $t + k$ where t is a time variable and k is an integer (the *offset*); we write $t - k$ for $t + k'$ with k' the opposite of k, and we abbreviate $t + 0$ as t. A *term* is either an object term or a time term. An *atom* is an expression $P(s_1, \ldots, s_n)$ where P is an n-ary predicate, each s_i for $1 \leq i \leq n-1$ is an object term, and s_n is a time term.

A *fact* is an atom where each term is either an object or a time point. For a (possibly infinite) set of facts F and an interval ρ of $\mathbb{Z}$, we denote with $F\restriction_\rho$ the subset of facts in F holding in ρ, that is, $F\restriction_\rho = \{P(\mathbf{a}, \tau) \in F \mid \tau \in \rho \cap \mathbb{N}\}$; we write $F\restriction_\tau$ for $F\restriction_{[\tau,\tau]}$. A *rule* r is a first-order sentence of the form (14), where the *body* φ of r is a conjunction of atoms and the *head* α is an atom:

$$\forall \mathbf{x}.\varphi \rightarrow \alpha. \tag{14}$$

As usual, we will omit universal quantifiers when writing rules and require that rules are *safe*, i.e., that each variable in a rule occurs in its body. Furthermore, we impose a *temporal guardedness condition* on rules, which requires that all the atoms in a rule mention the same time variable. A *program* is a finite set Π of (safe and temporally guarded) rules.[2]

Intuitively, guardedness ensures that an application of a rule to a stream is localised, in the sense that the time arguments of all facts in the stream matching the antecedent of the rule are close to each other (with the maximum distance depending on the radius of the rule). Temporal guardedness formally captures within our framework a principle underlying all stream query languages based on sliding-window constructs—e.g., CQL [8]—where each query evaluation considers only time points at constant distance from the current evaluation time point.

Example 2.1. The rules in Examples 1.1 and 1.2 satisfy our guardedness condition. In contrast, rule $r = P(t_1) \wedge Q(t_2) \rightarrow R(t_2)$ does not as it contains two different time variables in the body, where only one of them is propagated to the head.

[2] Note that, strictly speaking, Temporal Datalog is not a syntactic extension of Datalog, as it does not allow for atoms without a time argument; it is, however, straightforward to encode any Datalog program into Temporal Datalog so that entailment is preserved in a well-understood way.

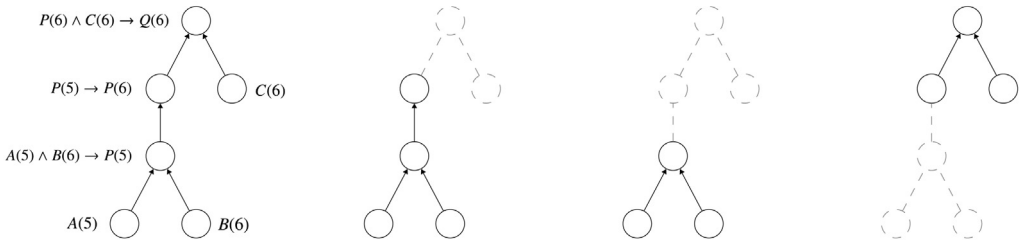


Fig. 1. The first labelled tree from the left is a derivation δ of $Q(6)$ from $\Pi \cup D$ with $\Pi = \{A(t) \wedge B(t+1) \rightarrow P(t), P(t) \rightarrow P(t+1), P(t) \wedge C(t) \rightarrow Q(t)\}$ and $D = \{A(5), B(6), C(6)\}$. Each node has its label right next to it. Considering the same labelling for the other trees, the second tree is a $P(6)$-subderivation of δ, which is also strict and immediate; the third tree is a $P(5)$-subderivation of δ, which is strict but not immediate; finally, the fourth tree is a partial derivation, but not a derivation because it has no $P(5)$-subderivation.

Intuitively, r derives a fact $R(\tau)$ for a time point τ if Q holds at the same time point and P holds *somewhere* in the stream. Thus, fact $R(\tau)$ may be justified by a fact about P that appears arbitrarily far away into the future, and hence a stream reasoning algorithm would need to wait indefinitely in order to be certain as to whether $R(\tau)$ holds or not. It is worth noticing that rule r can be rewritten into rules with a single time variable by introducing fresh predicates and recursion, thus obtaining rules $Q(t) \wedge P'(t) \rightarrow R(t)$, $P(t) \rightarrow P'(t)$, $P'(t) \rightarrow P'(t+1)$, and $P'(t) \rightarrow P'(t-1)$. Such rewriting, however, can easily turn a program admitting a valid delay (as will be defined later on in Section 3) into a program with no valid delay, thus making our stream reasoning algorithms not applicable to the resulting rewriting.

We distinguish between *extensional* (EDB) and *intensional* (IDB) predicates, where only the former can occur in data and only the latter can occur in rule heads. Formally, a predicate is IDB in program Π if it occurs in the head of a rule of Π, and it is EDB otherwise. An atom is IDB (respectively, EDB) in Π if so is its predicate; we will often not mention Π when referring to EDB or IDB predicates or atoms when it is clear from the context. A term, atom, rule, or program is *ground* if it has no variables, *object-ground* if it has no object variables, and *object-free* if it has no object terms.

The *forward radius* of a rule r of the form (14) is the maximum between zero and $k - k'$ for k the offset in α and k' the minimum offset of an atom in φ; similarly, the *backward radius* of r is the absolute value of the minimum between zero and $k - k''$, with k'' the maximum offset of an atom in φ. A rule is *forward-propagating* if its backward radius is zero, and a program is *forward-propagating* if so are all of its rules.

A *substitution* σ is a finite partial mapping of variables to terms of the suitable sort. For α a term, an atom, a rule, or a set thereof, $\alpha\sigma$ denotes the result of replacing each variable x in α (and defined in σ) with $\sigma(x)$. If σ is defined on all variables of α, then $\alpha\sigma$ is an *instance* of α. We write $\downarrow\tau$ for the substitution that maps all time variables to a time point τ.

A *(Herbrand) interpretation* is a (possibly infinite) set of facts. An interpretation $\mathcal{I}$ *satisfies* a ground atom α, written $\mathcal{I} \models \alpha$, if evaluating the numeric term in α by computing the relevant addition yields a fact in $\mathcal{I}$. Interpretation $\mathcal{I}$ satisfies a conjunction of ground atoms if it satisfies each of the conjuncts, and it satisfies a rule of the form (14) if, for each ground instance $\varphi\sigma \rightarrow \alpha\sigma$ of the rule we have that $\mathcal{I} \models \varphi\sigma$ implies $\mathcal{I} \models \alpha\sigma$. Finally, $\mathcal{I}$ satisfies a set of rules and/or facts if it satisfies each element of the set. For E an atom, a conjunction of atoms, a rule, a program, or a set of facts, an interpretation $\mathcal{I}$ is a *model* of E if $\mathcal{I}$ satisfies E; a program Π and a set of facts D *entail* E, written $\Pi \cup D \models E$, if each interpretation satisfying both Π and D satisfies E.

As customary in the treatment of database query languages, we often see a program Π as a transformation that maps each set D of facts to the set of all IDB facts that are entailed by $\Pi \cup D$, which we denote as $\Pi(D)$.

Fact entailment is the problem of checking $\Pi \cup D \models \alpha$ for a given program Π, a finite set of facts D, and a fact α as input. The *data complexity* of fact entailment is the complexity when Π and α are considered fixed and only D is considered as the input. Fact entailment in Datalog_{1S} is PSPACE-complete in data complexity [19]. Under unary coding of numbers, Temporal Datalog can be seen as a fragment of Datalog_{1S} where a term $t + k$ corresponds to k applications of the successor function symbol to t. It is worth noticing, however, that our temporal guardedness does not make standard reasoning easier: using essentially the same complexity lower bound proofs as for Datalog_{1S} it is immediate to show that fact entailment over Temporal Datalog remains PSPACE-hard in data complexity, even if numbers are coded in unary.

Entailment of α from $\Pi \cup D$ can be characterised in terms of existence of a *derivation* of α from $\Pi \cup D$, where such derivation δ is a finite node-labelled tree satisfying the following properties: *(i)* each node is labelled with a ground instance of a rule in $\Pi \cup D$ (where a fact in D is seen as a rule whose body is empty); *(ii)* fact α is the head of the rule labelling the root; and *(iii)* for each node v, the body of the rule labelling v contains an atom β if and only if β is the head of a rule labelling a child of v. A *β-subderivation* of δ is a subtree of δ that is itself a derivation of β. We say that the subderivation is *strict* if it is not δ itself, and it is an *immediate subderivation* of δ if it is rooted in a child of the root of δ. In some of our constructions later on in the paper, it will be useful to consider *partial derivations*, where condition *(iii)* is weakened to only require that each head of a rule labelling a non-root node v must occur in the body of the rule labelling the parent of v. The former notions are illustrated in Fig. 1. We then say that a fact α *depends* on a fact β in a ground program Π (via $n \geq 1$

Algorithm 1: Stream reasoning.

Parameters: program Π, non-negative integers d (the delay) and w (the window size) with $w \geq d$.
Input: EDB stream S for Π.
Output: IDB stream for Π.

Assign $M := \emptyset$ and $\tau := 0$;
loop
  Receive $S\restriction_\tau$ and set $M := M \cup S\restriction_\tau$;
  Compute $\Pi(M)\restriction_{\tau-d}$ and set $M := M \cup \Pi(M)\restriction_{\tau-d}$;
  Stream out all IDB facts in $M\restriction_{\tau-d}$;
  Remove from M all facts in $M\restriction_{\tau-w}$;
  $\tau := \tau + 1$;

steps) if there is a partial derivation of α from $\Pi \cup \{\beta\}$ having height n; and that α has *rank* k in Π if k is the minimum non-negative integer such that α depends on no fact in Π via more than k steps.

Example 2.2. In a ground program $\Pi = \{A(5) \rightarrow P(5),\ B(6) \wedge P(5) \rightarrow Q(5)\}$, fact $P(5)$ depends on $A(5)$ via one step, and fact $Q(5)$ depends on $A(5)$ via two steps and on $B(6)$ via one step; furthermore, the rank of fact $Q(5)$ is two, and the rank of $P(5)$ is one; finally, the rank of $A(5)$ and $B(6)$ is zero because they depend on no fact.

In addition to fact entailment, we will also consider the *program containment problem* for input programs Π_1 and Π_2 with the same set of EDB predicates. We say that Π_1 is *contained* in Π_2, written $\Pi_1 \sqsubseteq \Pi_2$, if $\Pi_1(D) \subseteq \Pi_2(D)$ for each set D of EDB facts; then, *program containment* is the problem of checking $\Pi_1 \sqsubseteq \Pi_2$ given Π_1 and Π_2 as input. Note that our definition of containment considers possibly infinite sets of facts, which is required to capture streams. This does not change the nature of the problem—due to the compactness and monotonicity properties of first-order logic, the well-known undecidability result for the containment of Datalog programs (with respect to finite sets of facts) transfers to our setting [23,24].

3. A generic stream reasoning algorithm

In this section, we describe a generic stream reasoning algorithm that is compatible with a wide range of expressive rule languages. Indeed, the properties of our algorithm rely only on the following assumptions about the underpinning rule language.

1. A *discrete timeline*, such as the non-negative integers; stream reasoning over dense timelines (such as the rationals) is a substantially different problem, and we refer the reader to [25] for first results in this direction.
2. Data represented as *timestamped facts* both in the input and during processing by the algorithm; in particular, it is assumed that the algorithm ingests, stores and streams out timestamped facts only and that all the facts stored by the algorithm are logically entailed by the program and input stream (this latter requirement could potentially be relaxed by allowing the algorithm to store additional information). The notion of a timestamp can be seen as a syntactic artifact ensuring the more fundamental requirement of an order between the pieces of information received and processed by the algorithm.
3. *Monotonicity* of the entailment relation, which ensures that answers proved to hold over the partial data received so far will continue to hold regardless of the data that may be received in the future.

Our algorithm relies on the notion of a sliding window to process the input stream incrementally, while at the same time abstracting away from many details that are not fundamental to the overall approach.

We start by providing a formal definition of a *stream*. Intuitively, we see streams as infinite datasets with a finite projection to each specific time point.

Definition 3.1. A stream is a (possibly infinite) set of facts S such that $S\restriction_\tau$ is a finite set for each time point τ. The *object domain* of S is the set of objects occurring in S. An *EDB stream* (respectively, *IDB stream*) for a program Π is a stream where all facts are EDB (respectively, IDB) with respect to Π; we will often omit the reference to Π where it is clear from the context.

A stream reasoning algorithm receives as input a (possibly unbounded) *stream* S of timestamped facts, where we assume that facts in the stream arrive in non-decreasing order of timestamp; although facts may arrive with a delay in a distributed system, modern stream processors exploit a number of techniques (such as low watermarks [26]) to check whether all facts up to a given time point have been received. The stream reasoning algorithm then implements the transformation $\Pi(S)$ for a fixed program Π, where the output facts are also returned as a stream. In addition to program Π, Algorithm 1 is also parametrised with a non-negative *delay* d and *window size* w. The delay parameter influences the latency of the algorithm,

as it determines when exactly the derived IDB facts are streamed out; in turn, the window size parameter w determines memory consumption. The algorithm is initialised in Line 1, where the memory M is set empty and the current time τ is set to zero. The core of the algorithm is an infinite loop, where each iteration of the loop processes a single time point τ in the input; the current time τ is incremented at the end of each iteration. More precisely, each iteration of the main loop consists of the following steps.

1. The set of all input stream facts holding at τ is received and loaded into memory (Line 3).
2. All (implicit) IDB facts holding at $\tau - d$ are computed and stored in memory (Line 4).
3. All IDB facts holding at $\tau - d$ are read from memory and streamed as part of the output (Line 5).
4. All facts (EDB or IDB) holding at $\tau - w$ are removed from memory (Line 6).

The first important feature of the algorithm is that old facts falling outside the sliding window are discarded in Line 6 and never reconsidered again; this has the key advantage of limiting the number of facts that the algorithm needs to reason about at any point in time, and hence favours fast processing. The use of a sliding window, however, carries the risk of missing IDB facts α entailed by $\Pi \cup S$ if the facts that α depends on are removed from memory too early.

The second feature of the algorithm is that output facts are streamed in increasing order of timestamp; indeed, all output facts for a given timestamp are returned (in Step 3) during the same iteration of the main loop. This has the advantage that applications do not need to wait indefinitely for output facts, but carries the risk of missing IDB facts α with timestamp τ entailed by $\Pi \cup S$ if the facts in S that α depends on have not been received yet by the time the algorithm generates the output for τ.

Finally, the third important feature of the algorithm is that it keeps in memory a complete materialisation of the window—that is, all input EDB facts and entailed IDB facts for the relevant time points. Computing and incrementally maintaining a full materialisation is a common reasoning approach adopted by many rule-based systems [27–31].

Clearly, the delay and window size parameters must be chosen to be as small as possible (thus maximising the algorithm's efficiency) while at the same time ensuring that all facts entailed by $\Pi \cup S$ will be returned as part of the output stream. The following example illustrates how a poor choice of parameters may compromise the algorithm's correctness.

Example 3.2. Consider Algorithm 1 parametrised with program $\Pi = \{P(t) \rightarrow Q(t-5)\}$, delay $d = 0$ and window size $w = 0$. Consider also the input stream S consisting of a fact $P(10)$. We have that $\Pi \cup \{P(10)\} \models Q(5)$, so the algorithm is required to eventually output $Q(5)$. The algorithm, however, will output all consequences for $\tau = 5$ in the sixth iteration of the main loop (and will never attempt to compute them again later on); in this iteration, no input fact will have been received yet and the memory M will be empty as a result. Thus, fact $Q(5)$ will not be returned. By contrast, Algorithm 1 parametrised with Π as above, $d = 5$ and $w = 0$ will output $Q(5)$ in the eleventh iteration of the main loop. Similarly, Algorithm 1 parametrised with $\Pi' = \{P(t) \rightarrow Q(t+5)\}$, $d = 0$ and $w = 0$ will not output $Q(15)$ on S even though $\Pi' \cup \{P(10)\} \models Q(15)$; in order for the algorithm to return $Q(15)$, the window size parameter w needs to be increased to 5.

We are now ready to define validity of a delay d for a program Π. We define delay validity as a property of a program Π, and hence our definition is not tied in any way to Algorithm 1. Intuitively, d is a valid delay if, in order to compute the logical consequences of $\Pi \cup S$ up to time $\tau - d$, one does not need to consider any future facts in S with timestamp exceeding τ. As a result, Algorithm 1 does not need to wait indefinitely before it can determine with certainty that all entailed IDB facts have already been computed for a given time point.

Definition 3.3. A non-negative integer d is a *valid delay* for a program Π if, for each EDB stream S and each time point $\tau \geq d$, we have that $\Pi(S)\restriction_{\tau-d} \subseteq \Pi(S\restriction_{[0,\tau]})$.

Note that, by definition, if d is a valid delay for Π, then so is each $d' > d$; hence, we will be interested in determining whether Π admits a valid delay and, if so, in computing the smallest valid delay.

We next define the notion of a valid window size w as a property of a program Π having valid delay d. Again, we define window size validity exclusively as a property of Π; thus, our definition is not tied to Algorithm 1. Intuitively, the definition ensures that, in order to compute a logical consequence α of $\Pi \cup S$ at time $\tau - d$, one does not need to consider any fact (EDB or IDB) older than $\tau - w$, and hence such old facts can be safely "forgotten" by a procedure such as Algorithm 1. Furthermore, the definition ensures that the IDB facts in the interval $(\tau - d, \tau]$ entailed by $\Pi \cup S$ are not required in order to derive α, which implies that an algorithm only needs to keep in memory a full materialisation in the interval between $\tau - w$ and $\tau - d$.

Definition 3.4. Let Π be a program and d a valid delay for Π. We say that $w \geq d$ is a *valid window size* for Π and d if, for each EDB stream S and each $\tau \geq d$, the following inclusion holds, where $N = \Pi(S\restriction_{[0,\tau]})$, $\tau_d = \tau - d$, and $\tau_w = \tau - w$:

$$N\restriction_{\tau_d} \subseteq \Pi(N\restriction_{[\tau_w, \tau_d - 1]} \cup S\restriction_{[\tau_w, \tau]}).$$

Note that, as in the case of delay, if w is a valid window size for Π and d, then so is each $w' > w$. Hence, in practice we will be interested in computing the minimum such w.

We next show that delay and window size validity as introduced in Definitions 3.3 and 3.4 characterise the correctness of Algorithm 1—that is, the algorithm will correctly output $\Pi(S)$ for each S if and only if d is a valid delay for Π and w is a valid window size for Π and d. Before establishing this result formally, we prove a technical lemma, which describes the contents M_τ of the memory at any point τ during the execution of the algorithm in terms of the logical consequences of $\Pi \cup S$.

Lemma 3.5. *Consider the execution of Algorithm 1 parametrised with Π, d and w on an input stream S. For τ a time point, let M_τ be the value of the memory variable M in the $(\tau+1)$-th iteration of the main loop of the algorithm right after executing Line 4. Then, the following containment holds, where $N = \Pi(S\restriction_{[0,\tau-1]})$, $\tau_d = \tau - d$, and $\tau_w = \tau - w$:*

$$M_\tau \subseteq \Pi(N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]})\restriction_{[\tau_w,\tau_d]} \cup S\restriction_{[\tau_w,\tau]}.$$

Proof. Note that, by definition of the algorithm, M_τ satisfies the following recursive equation for each $\tau \geq 0$ where $M_{-1} = \emptyset$:

$$M_\tau = M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau \cup \Pi(M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau)\restriction_{\tau_d}. \tag{15}$$

We show the claim of the lemma by induction on τ. In the base case, we have $\tau = 0$. Then, $M_0 = \Pi(S\restriction_0)\restriction_{\tau_d} \cup S\restriction_0$, which implies the claim since $S\restriction_0 = S\restriction_{[-w,0]}$.

In the inductive case, we have $\tau > 0$, and we assume that the claim holds for $\tau - 1$. By (15), it suffices to show the following inclusion, for $T = \Pi(N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]})\restriction_{[\tau_w,\tau_d]} \cup S\restriction_{[\tau_w,\tau]}$:

$$M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau \cup \Pi(M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau)\restriction_{\tau_d} \subseteq T.$$

Clearly, $S\restriction_\tau \subseteq T$, and thus it remains to be shown that $M_{\tau-1}\restriction_{[\tau_w,\infty)} \subseteq T$ and $\Pi(M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau)\restriction_{\tau_d} \subseteq T$. For the former inclusion, note that the inductive hypothesis yields the following inclusion, for $N' = \Pi(S\restriction_{[0,\tau-2]})$:

$$\begin{aligned} &M_{\tau-1}\restriction_{[\tau_w,\infty)} \\ &\subseteq (\Pi(N'\restriction_{[\tau_w-1,\tau_d-2]} \cup S\restriction_{[\tau_w-1,\tau-1]})\restriction_{[\tau_w-1,\tau_d-1]} \cup S\restriction_{[\tau_w-1,\tau-1]})\restriction_{[\tau_w,\infty)} \\ &= \Pi(N'\restriction_{[\tau_w-1,\tau_d-2]} \cup S\restriction_{[\tau_w-1,\tau-1]})\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau-1]}. \end{aligned}$$

However, we have $\Pi(N'\restriction_{[\tau_w-1,\tau_d-2]} \cup S\restriction_{[\tau_w-1,\tau-1]}) \subseteq N$, and hence the above implies

$$M_{\tau-1}\restriction_{[\tau_w,\infty)} \subseteq N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau-1]} \subseteq T \tag{16}$$

as required. For the remaining inclusion, note that the first inclusion in (16) immediately yields

$$\Pi(M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau)\restriction_{\tau_d} \subseteq \Pi(N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]})\restriction_{\tau_d} \subseteq T$$

which concludes the proof. □

With Lemma 3.5 at hand, we are ready to characterise correctness of Algorithm 1 in terms of delay and window size validity.

Theorem 3.6. *Algorithm 1 parametrised with Π, d, and w outputs $\Pi(S)$ for each input stream S if and only if d is a valid delay for Π and w is a valid window size for Π and d.*

Proof. ($\Leftarrow$) Assume that d is a valid delay for Π and that $w \geq d$ is a valid window size for Π and d, and let S be an arbitrary input stream. We show that the algorithm outputs $\Pi(S)$. Let τ be a time point with $\tau \geq d$ and let τ_d, τ_w and M_τ be defined as in Lemma 3.5. Note that the algorithm outputs an IDB fact α with time argument τ if and only if $\alpha \in M_{\tau+d}$; thus, it suffices to show that $\Pi(S)\restriction_{\tau_d} \subseteq M_\tau$. Furthermore, since d is assumed to be a valid delay for Π, it suffices to show that

$$S\restriction_{[\tau_w,\tau]} \cup N\restriction_{[\tau_w,\tau_d]} \subseteq M_\tau \text{ where } N = \Pi(S\restriction_{[0,\tau]}).$$

We show this claim by induction on τ. In the base case, we have $\tau = 0$, and hence either $d = 0$ or $\tau_d < 0$. In the latter case, we have $N\restriction_{[\tau_w,\tau_d]} = \emptyset$, while in the former case, we have $N\restriction_{[\tau_w,\tau_d]} = \Pi(S\restriction_0)\restriction_0 \subseteq M_0$ by (15). Furthermore, in both cases we have $S\restriction_{[\tau_w,\tau]} = S\restriction_0 \subseteq M_0$ by (15). In the inductive case, we have $\tau > 0$ and we assume that the claim holds for $\tau - 1$. First, $S\restriction_{[\tau_w,\tau]} \subseteq M_\tau$ by (15) and the inductive hypothesis. Second, note that, for $N' = \Pi(S\restriction_{[0,\tau-1]})$,

$$N\restriction_{[\tau_w,\tau_d-1]} \subseteq N'\restriction_{[\tau_w,\tau_d-1]} \subseteq M_{\tau-1}\restriction_{[\tau_w,\infty)} \subseteq M_\tau$$

where the first inclusion holds since d is a valid delay, the second inclusion holds since $N'\restriction_{[\tau_w-1,\tau_d-1]} \subseteq M_{\tau-1}$ by the inductive hypothesis, and the last inclusion holds by (15). Hence, it remains to be shown that

$$N\restriction_{\tau_d} \subseteq M_\tau.$$

To this end, note that, since w is a valid window size, we have

$$N\restriction_{\tau_d} \subseteq \Pi(N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]})\restriction_{\tau_d} \tag{17}$$

where the latter set can be equivalently written as $\Pi((N\restriction_{[\tau_w-1,\tau_d-1]} \cup S\restriction_{[\tau_w-1,\tau-1]})\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau)\restriction_{\tau_d}$. Furthermore, since d is a valid delay for Π, we have $N\restriction_{[\tau_w-1,\tau_d-1]} = N'\restriction_{[\tau_w-1,\tau_d-1]}$, and hence $N\restriction_{[\tau_w-1,\tau_d-1]} \subseteq M_{\tau-1}$ by the inductive hypothesis. Since also $S\restriction_{[\tau_w-1,\tau-1]} \subseteq M_{\tau-1}$ by the inductive hypothesis, then, by monotonicity of entailment, inclusion (17) implies

$$N\restriction_{\tau_d} \subseteq \Pi(M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau)\restriction_{\tau_d}$$

and the claim follows since $\Pi(M_{\tau-1}\restriction_{[\tau_w,\infty)} \cup S\restriction_\tau)\restriction_{\tau_d} \subseteq M_\tau$ by (15).

($\Rightarrow$) We show that Algorithm 1 does not output $\Pi(S)$ for any input S if either *(i)* d is not a valid delay for Π or *(ii)* d is a valid delay for Π but w is not a valid window size for Π and d. As observed before, in either case it suffices to show that $\Pi(S)\restriction_{\tau_d} \not\subseteq M_\tau$.

In the first case, let S be an input stream, let $\tau \geq d$ be a time point, and let α be a fact in $\Pi(S)\restriction_{\tau_d} \setminus \Pi(S\restriction_{[0,\tau]})$—such S, τ and α exist because d is not a valid delay for Π. We have $(M_\tau \setminus S\restriction_{[0,\tau]}) \subseteq \Pi(S\restriction_{[0,\tau]})$ by Lemma 3.5, and hence $\alpha \notin M_\tau$, as required.

In the second case, let S be a stream, let $\tau \geq d$, let $N = \Pi(S\restriction_{[0,\tau]})$, and let $\alpha \in N\restriction_{\tau_d} \setminus \Pi(N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]})$, which exist since w is not a valid window size for Π and d. By Lemma 3.5, $\alpha \notin M_\tau$, and the claim follows since $N\restriction_{\tau_d} \subseteq \Pi(S)\restriction_{\tau_d}$. □

Unfortunately, as illustrated by the following example, there are programs for which a valid delay does not exist, and hence to which our algorithm is not applicable.

Example 3.7. Consider $\Pi = \{P(t) \to Q(t), Q(t) \to Q(t-1)\}$, which does not admit a valid delay. Indeed, for each non-negative integer d, we can define $S_d = \{P(d+1)\}$, and obtain $\Pi \cup S_d \models Q(0)$, while $\Pi \cup S_d\restriction_{[0,d]} \not\models Q(0)$ since $S_d\restriction_{[0,d]} = \emptyset$. Intuitively, Π lacks a valid delay because the rules in Π can recursively propagate information towards past time points in an unbounded way.

We can show, however, that if a program Π does admit a valid delay d, then it must admit a valid window size w as well. Furthermore, the window size is bounded linearly in d and the forward radius of Π. The proof of this theorem critically relies on our temporal guardedness assumption; indeed, a program with multiple time variables may admit no valid window size even if it admits a valid delay. This is the case for the program consisting of rules $A(t) \to P(t)$, $P(t) \to P(t+1)$, and $P(t) \wedge A(t') \to Q(t)$; indeed, an input fact $A(0)$ can never be deleted. The same is true for rules mentioning time points, such as $A(0) \wedge B(t) \to C(t)$.

Theorem 3.8. *Let Π be a program, let d be a valid delay for Π, and let ρ be the maximum forward radius of a rule in Π. Then, $w = d + \rho$ is a valid window size for Π and d.*

Proof. Let S be an EDB stream, $\tau \geq d$ a time point, $w = d + \rho$, $\tau_d = \tau - d$, $\tau_w = \tau - w$, and $N = \Pi(S\restriction_{[0,\tau]})$. We show that every fact α that has time argument $\tau' \geq \tau_d$ and is entailed by $\Pi \cup S\restriction_{[0,\tau]}$ is also entailed by $\Pi \cup N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]}$ by induction on the height of a shortest derivation δ of α from $\Pi \cup S\restriction_{[0,\tau]}$; the main claim is then immediate. Let α, τ', and δ be as required. In the base case, we have $\alpha \in \Pi \cup S\restriction_{[0,\tau]}$, and hence $\alpha \in S\restriction_{[0,\tau]}$ as Π contains no facts. Consequently, $\alpha \in S\restriction_{[\tau_w,\tau]}$ as $\tau' \geq \tau_d \geq \tau_w$, and the claim follows. In the inductive case, let r be the rule labelling the root of δ and let β be an arbitrary atom in the body of r with time argument τ''. It suffices to show $\Pi \cup N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]} \models \beta$. Note that, since Π is temporally guarded, $\tau'' \geq \tau' - \rho \geq \tau_d - \rho = \tau_w$. Thus, we distinguish two cases: $\tau'' \in [\tau_w, \tau_d - 1]$ and $\tau'' \geq \tau_d$. In the former case, the claim follows since $\Pi \cup S\restriction_{[0,\tau]} \models \beta$, and hence $\beta \in N\restriction_{[\tau_w,\tau_d-1]} \cup S\restriction_{[\tau_w,\tau]}$, while in the latter case, the claim follows from $\Pi \cup S\restriction_{[0,\tau]} \models \beta$ by the inductive hypothesis as β has a derivation that is strictly shorter than δ. □

It is important to notice, however, that the valid window size provided by Theorem 3.8 may not be the minimal one.

Example 3.9. Consider the program consisting of rules $A(t) \to P(t+1)$ and $P(t) \wedge B(t+1) \to Q(t)$. The program admits a minimum valid delay of one, and its maximum forward radius is one as well. As a result, $w = 2$ is a valid window size by Theorem 3.8. We can show, however, that the minimum window size of this program is one.

It follows from our previous discussion that the computational task of interest in our setting is to first check whether a program Π admits a valid delay and, if it does, to subsequently compute the corresponding minimal valid delay d for Π and the minimal valid window size w for Π and d. Reducing delays and window sizes is important in practice, as it ensures that a correct algorithm will keep to a minimum the number of facts stored in memory at any point in time, and will minimise latency by returning each output fact as early as possible. Therefore, in the remainder of this paper, we will focus on the following decision problems.

Definition 3.10. We define the following decision problems for an input program Π.

- *Delay existence*: Decide whether a valid delay for Π exists.
- *Delay validity*: Decide whether a non-negative integer d is a valid delay for Π.
- *Window size validity*: Given a valid delay d for Π, and $w \geq d$, decide whether w is a valid window size for Π and d.

We conclude this section by introducing the class of *backward-bounded programs*. Intuitively, these programs ensure existence of a valid delay by precluding the kind of temporal recursion towards past time points illustrated in Example 3.7.

Definition 3.11. The *weighted dependency graph* of a program Π is an edge-weighted graph having a node for each predicate in Π and an edge $\langle P, R, w\rangle$ whenever P and R are predicates occurring in the body and head of a rule in Π, respectively, and where $w = \min\{k - k' \mid (P(\mathbf{s}', k') \wedge \varphi \rightarrow R(\mathbf{s}, k)) \in \Pi\}$. Program Π has *backward bound* k (for $k \geq 0$) if, for each path in the weighted dependency graph of Π that starts in an EDB predicate, the sum of the edge weights is at least $-k$; Π is *backward-bounded* if it has a backward bound.

Our motivating Examples 1.1 and 1.2 are backward-bounded; moreover, the program in Example 1.1 has backward bound zero as it is forward-propagating. Note that, while forward-propagating programs have backward bound zero, there are programs with backward bound zero that are not forward-propagating, e.g., the program $\{A(t) \rightarrow B(t+1), B(t) \rightarrow C(t-1)\}$. Note also that checking whether a program is backward-bounded and, if so, computing the least backward bound is feasible in polynomial time using standard graph algorithms. We next show that the backward bound of a program Π is guaranteed to be a valid delay for Π.

Theorem 3.12. *Let Π be a program with backward bound k. Each integer $d \geq k$ is a valid delay for Π.*

Proof. Assume that $d \geq k$ is not a valid delay for Π. We show that k is not a backward bound for Π. By the definition of valid delay, there is a stream S, a time point $\tau \geq d$, and a fact β with time argument $\tau - d$ such that $\beta \in \Pi(S) \setminus \Pi(S\restriction_{[0,\tau]})$. As a result, there exists a fact α with time argument $\tau' > \tau$ such that $\beta \notin \Pi(S \setminus \{\alpha\})$. By a simple induction on the height of a derivation of β from $\Pi \cup S$ we can show existence of a path from the predicate P_α of α to the predicate P_β of β in the weighted dependency graph of Π having weight $\tau - d - \tau'$. Note that $\tau - d - \tau' < \tau' - d - \tau' = -d \leq -k$, where the first inequality holds since $\tau' > \tau$ and the second one since $d \geq k$. Thus, the weighted dependency graph of Π has a path of weight strictly smaller than $-k$, and hence k is not a backward bound for Π. □

It follows from Theorem 3.12 that delay existence, as given in Definition 3.10, is trivial for backward-bounded programs. Delay and window size validity checking, however, remain important for these programs since the valid delay established by the theorem may not be minimal.

Example 3.13. The program consisting of rules $A(t) \rightarrow B(t)$ and $A(t) \wedge C(t+5) \rightarrow B(t)$ is backward-bounded with least backward bound 5; however, $d = 0$ is a valid delay. In fact, the second rule is redundant, and hence can be removed to obtain a program with backward bound 0.

4. Undecidability results

In this section, we explore the limitations of our approach and establish undecidability results for all the reasoning problems we consider. Our proofs are obtained by reduction from program containment, which is well-known to be undecidable already for non-temporal Datalog [23,24]. We start by establishing undecidability of delay existence.

Theorem 4.1. *Delay existence is undecidable.*

Proof. We provide a reduction from containment of forward-propagating programs, which is undecidable (containment of non-temporal programs is already undecidable). Our reduction maps a pair of forward-propagating programs $\langle \Pi_1, \Pi_2\rangle$ to a program Π such that $\Pi_1 \sqsubseteq \Pi_2$ if and only if Π admits a valid delay. We construct Π as follows. For each predicate P, let P^1, P^2, A_P and B_P be fresh predicates of the same arity as (and uniquely associated to) P. Let Π_1' (respectively Π_2') be the

program obtained by replacing each IDB predicate P in Π_1 (respectively in Π_2) with P^1 (respectively with P^2). Then, Π is the program extending $\Pi_1' \cup \Pi_2'$ with the following rules, where A and B are fresh unary (EDB) predicates:

- Rules (18)–(21) for each IDB predicate P of Π_1:

$$P^1(\mathbf{x}, t) \wedge A(t) \rightarrow A_P(\mathbf{x}, t) \tag{18}$$

$$A_P(\mathbf{x}, t) \rightarrow A_P(\mathbf{x}, t+1) \tag{19}$$

$$A_P(\mathbf{x}, t) \wedge B(t) \rightarrow B_P(\mathbf{x}, t) \tag{20}$$

$$B_P(\mathbf{x}, t+1) \wedge A_P(\mathbf{x}, t) \rightarrow B_P(\mathbf{x}, t). \tag{21}$$

- Rules (22) and (23) for each IDB predicate P occurring in Π_2:

$$P^2(\mathbf{x}, t) \wedge A(t) \rightarrow B_P(\mathbf{x}, t) \tag{22}$$

$$B_P(\mathbf{x}, t) \rightarrow B_P(\mathbf{x}, t+1). \tag{23}$$

Let $\Pi_1 \sqsubseteq \Pi_2$. We argue that zero is a valid delay for Π. For this, observe that $\Pi_1 \sqsubseteq \Pi_2$ implies that, for each set of EDB facts and each IDB P, the extension of P^1 will be contained in that of P^2 by our construction. Thus, Rules (20) and (21) become subsumed by (22) and (23) and hence can be removed from Π without altering fact entailment; in doing so, we can observe that Π would become forward-propagating since so are Π_1 and Π_2, and hence it would admit zero as a valid delay by Theorem 3.12.

Assume now that $\Pi_1 \not\sqsubseteq \Pi_2$ and fix any d; we argue that d cannot be a valid delay for Π. By our assumption, there must be a set D of EDB facts and facts $P^1(\mathbf{o}, \tau)$ and $P^2(\mathbf{o}, \tau)$ such that $\Pi_1' \cup D \models P^1(\mathbf{o}, \tau)$ but $\Pi_2' \cup D \not\models P^2(\mathbf{o}, \tau)$. Let D' be the set extending D with facts $A(\tau)$ and $B(\tau + d + 1)$. But then, we have that $B_P(\mathbf{o}, \tau)$ follows from $\Pi \cup D'$ but $B_P(\mathbf{o}, \tau)$ does not follow from $\Pi \cup D'\restriction_{[0,\tau+d]}$, because $B(\tau + d + 1) \notin D'\restriction_{[0,\tau+d]}$. Therefore, d cannot be a valid delay. □

We next show undecidability of the delay validity problem. Our undecidability result holds even for backward-bounded programs, which, by Theorem 3.12, always admit a valid delay of linear size.

Theorem 4.2. *Delay validity is undecidable, even if restricted to backward-bounded programs.*

Proof. We provide a reduction from containment of backward-bounded programs. The reduction maps a pair of backward-bounded programs $\langle \Pi_1, \Pi_2 \rangle$ to a program Π and a number k such that $\Pi_1 \sqsubseteq \Pi_2$ if and only if k is a valid delay for Π.

We assign k to the maximum between the least backward bounds of Π_1 and Π_2. To define Π, we proceed by defining Π_1' and Π_2' as in the proof of Theorem 4.1 and by introducing also a fresh unary (EDB) predicate A. Then, Π extends $\Pi_1' \cup \Pi_2'$ with two rules of the form (24) and (25), respectively, for each IDB predicate P in $\Pi_1 \cup \Pi_2$:

$$P^1(\mathbf{x}, t) \wedge A(t + k + 1) \rightarrow A_P(\mathbf{x}, t) \tag{24}$$

$$P^2(\mathbf{x}, t) \rightarrow A_P(\mathbf{x}, t). \tag{25}$$

Clearly, Π is backward-bounded since so are Π_1 and Π_2 and predicate A_P is fresh.

We next argue that the reduction is correct. Let us assume $\Pi_1 \sqsubseteq \Pi_2$. We argue that k is a valid delay for Π. For this, observe that $\Pi_1 \sqsubseteq \Pi_2$ implies that, for each finite set of EDB facts (and hence for each infinite one) and each IDB predicate P, the extension of P^1 is contained in the extension of P^2 by our construction. As a result, Rule (24) becomes subsumed by Rule (25) and hence can be removed from Π without altering fact entailment; in doing so, we can observe that Π would admit k as a backward bound, and hence as a valid delay by Theorem 3.12.

Assume now that k is a valid delay for Π. Let D be a finite set of EDB facts, let α be an IDB fact of the form $P(\mathbf{o}, \tau)$, and assume that $\Pi_1 \cup D \models \alpha$. We argue that $\Pi_2 \cup D \models \alpha$, which implies that $\Pi_1 \sqsubseteq \Pi_2$. Since $\Pi_1 \cup D \models \alpha$ we have that $\Pi_1' \cup D \models P^1(\mathbf{o}, \tau)$. Let $D' = D \cup \{A(\tau + k + 1)\}$; clearly, $\Pi \cup D' \models A_P(\mathbf{o}, \tau)$ since Rule (24) becomes applicable. It follows that $\Pi \cup D'\restriction_{[\tau+k]} \models A_P(\mathbf{o}, \tau)$ since k is a valid delay for Π. It follows that $\Pi \cup D'\restriction_{[\tau+k]} \models P^2(\mathbf{o}, \tau)$ since A_P occurs only in Rules (24) and (25), and Rule (24) requires $A(\tau + k + 1)$ to derive $A_P(\mathbf{o}, \tau)$, but $\Pi \cup D'\restriction_{[\tau+k]} \not\models A(\tau + k + 1)$. It follows that $\Pi_2 \cup D'\restriction_{[\tau+k]} \models P(\mathbf{o}, \tau)$, and hence $\Pi_2 \cup D\restriction_{[\tau+k]} \models P(\mathbf{o}, \tau)$ as required, since $D' \setminus D = \{A(\tau + k + 1)\}$ and predicate A does not occur in Π_2. □

We conclude by showing undecidability of window size validity checking. Our undecidability result applies even to forward-propagating programs, which admit zero as a valid delay (see Theorem 3.12) and a linear valid window size (see Theorem 3.8).

Theorem 4.3. *Window size validity is undecidable, even if restricted to forward-propagating programs.*

Proof. We provide a reduction from containment of forward-propagating programs. The reduction maps a pair of forward-propagation programs $\langle \Pi_1, \Pi_2 \rangle$ to a program Π with valid delay zero and a number w such that $\Pi_1 \sqsubseteq \Pi_2$ if and only if w is a valid window size for program Π and delay zero.

In our reduction, we assign w to the maximum forward radius of a rule in $\Pi_1 \cup \Pi_2$. To define Π, we proceed by constructing Π_1' and Π_2' as in the proofs of Theorems 4.1 and 4.2, and by introducing also a fresh unary (EDB) predicate A. Then, Π extends $\Pi_1' \cup \Pi_2'$ with two rules of the form (26) and (27), respectively, for each IDB predicate P occurring in $\Pi_1 \cup \Pi_2$:

$$P^1(\mathbf{x}, t) \wedge A(t - w - 1) \rightarrow A_P(\mathbf{x}, t) \tag{26}$$

$$P^2(\mathbf{x}, t) \rightarrow A_P(\mathbf{x}, t). \tag{27}$$

Clearly, Π is forward-propagating if so are Π_1 and Π_2 and hence admits zero as a valid delay.

We next argue that the reduction is correct. Assume that $\Pi_1 \sqsubseteq \Pi_2$, let S be an EDB stream for Π_1, let α be a fact with time point τ, and assume that $\Pi \cup S\restriction_{[0,\tau]} \models \alpha$. We show that $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup S\restriction_{[\tau_w,\tau]} \models \alpha$, with $N = \Pi(S\restriction_{[0,\tau]})$ and $\tau_w = \tau - w$, which is sufficient to establish that w is a valid window size for program Π and delay zero. The claim clearly holds if α is a fact of the form $P^i(\mathbf{o}, \tau)$ as a direct consequence of Theorem 3.8 and our definition of w. So, assume that α is of the form $A_P(\mathbf{o}, \tau)$. Since A_P occurs in Π only in the head of rules (26) and (27), we have that either $\Pi \cup S\restriction_{[0,\tau]} \models P^2(\mathbf{o}, \tau)$ or $\Pi \cup S\restriction_{[0,\tau]} \models \{P^1(\mathbf{o}, \tau), A(\tau_w - 1)\}$. In the first case, we have $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup S\restriction_{[\tau_w,\tau]} \models P^2(\mathbf{o}, \tau)$ again as a consequence of Theorem 3.8 and hence $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup S\restriction_{[\tau_w,\tau]} \models \alpha$ by rule (27), as required. In the second case, we have $\Pi \cup S\restriction_{[0,\tau]} \models P^1(\mathbf{o}, \tau)$, which implies $\Pi_1 \cup S\restriction_{[0,\tau]} \models P(\mathbf{o}, \tau)$. Since $\Pi_1 \sqsubseteq \Pi_2$ by assumption, we then have $\Pi_2 \cup S\restriction_{[0,\tau]} \models P(\mathbf{o}, \tau)$, and $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup S\restriction_{[\tau_w,\tau]} \models P^2(\mathbf{o}, \tau)$ by Theorem 3.8; thus, $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup S\restriction_{[\tau_w,\tau]} \models \alpha$ by rule (27), as required.

Assume now that w is a valid window size for program Π and delay zero. Let D be a finite set of EDB facts, let α be an IDB fact of the form $P(\mathbf{o}, \tau)$, and assume that $\Pi_1 \cup D \models \alpha$. We argue that $\Pi_2 \cup D \models \alpha$, which suffices to show that $\Pi_1 \sqsubseteq \Pi_2$. Since $\Pi_1 \cup D \models \alpha$, we have that $\Pi \cup D \models P^1(\mathbf{o}, \tau)$, and hence $\Pi \cup D \models A_P(\mathbf{o}, \tau)$ by rule (26). Let $N = \Pi(D\restriction_{[0,\tau]})$. It follows that $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup D\restriction_{[\tau_w,\tau]} \models A_P(\mathbf{o}, \tau)$ since w is a valid window size for Π and zero. Hence, $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup D\restriction_{[\tau_w,\tau]} \models P^2(\mathbf{o}, \tau)$ since A_P occurs in Π only in rules (26) and (27) and $\Pi \cup N\restriction_{[\tau_w,\tau-1]} \cup D\restriction_{[\tau_w,\tau]} \not\models A(\tau_w - 1)$. As a result, we have $\Pi \cup D \models P^2(\mathbf{o}, \tau)$ by the definition of N and the monotonicity and transitivity of entailment. We then have $\Pi_2 \cup D \models P(\mathbf{o}, \tau)$, as required. □

In the following sections, we show that decidability of all our reasoning problems can be regained by making rather mild assumptions on the input streams. In particular, we consider the situation where the set of domain objects that can occur in a stream is fixed in advance. This is a reasonable assumption in many applications; for instance, in Examples 1.1 and 1.2, it amounts to assuming that the nodes in the network and the sensors generating temperature readings remain the same throughout a given query session.

From a technical point of view, fixing the object domain allows us to rewrite any program Π as an object-free program $\zeta(\Pi)$ by means of an exponential grounding transformation. Formally, we associate a unary predicate $P_{\mathbf{o}}$ to each n-ary predicate P and n-tuple of objects $\mathbf{o}$ over the fixed object domain. Then, function ζ maps each object-ground atom $P(\mathbf{o}, s)$ to $P_{\mathbf{o}}(s)$; each set D of facts to $\bigcup_{\alpha \in D} \zeta(\alpha)$; and each program Π to the program $\zeta(\Pi)$ obtained by first grounding Π over the object domain and then replacing each (object-ground) atom α with $\zeta(\alpha)$. For notational convenience, we also define $\zeta(\mathbf{P})$ for a set of predicates $\mathbf{P}$ as the set of all unary predicates $P_{\mathbf{o}}$ in the range of ζ where predicate P is in $\mathbf{P}$. Clearly, transformation ζ preserves entailment of facts—that is, $\Pi \cup D \models \alpha$ if and only if $\zeta(\Pi) \cup \zeta(D) \models \zeta(\alpha)$.

We proceed as follows.

- In Section 5 we characterise delay existence and delay validity for a program Π in terms of languages over finite words defined for the object-free program $\zeta(\Pi)$; similarly, we characterise program containment in terms of finite word languages defined over their corresponding object-free programs.
- In Section 6, we show that the aforementioned languages relative to object-free programs expressed in a suitable normal form can be recognised by finite automata, and we also analyse the complexity of constructing such automata.
- In Section 7, we exploit these automata to establish upper bounds for delay existence, delay validity, and program containment for both arbitrary and object-free programs. In each case, we provide upper bounds for both unary and binary coding of numbers in the input program.
- Finally, in Section 8 we establish the corresponding matching lower bounds.

5. Language-theoretic characterisation of delay existence, delay validity, and program containment

In this section, we describe word languages that can be used to characterise delay existence, delay validity, and program containment. These languages are defined for an object-free program Ω and a subset $\mathbf{J}$ of its IDB predicates. As illustrated in Fig. 2, their alphabet is constructed so that each word represents an instance of the fact entailment problem with respect to Ω, given as a pair $\langle S, \alpha \rangle$ consisting of a set S of input EDB facts (representing a stream prefix) and a fact α with predicate in $\mathbf{J}$.

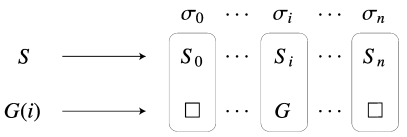


Fig. 2. Encoding of fact-entailment instances as words. An instance consisting of a finite set of facts S and a fact $G(i)$ is encoded as the word $\sigma_0 \dots \sigma_n$ where n is the maximum between i and the largest time point occurring in S. Each symbol σ_j is a pair where the first component is the set of predicates S_j occurring in $S\restriction_j$, and the second component is either the predicate G or the blank symbol □.

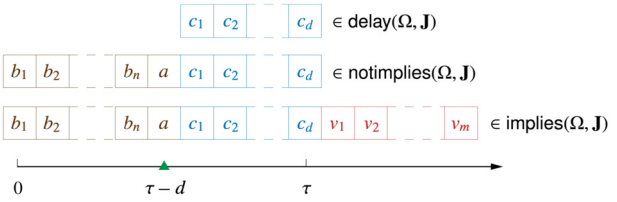


Fig. 3. The delay language. A word $w_1 = c_1 \dots c_d$ is in the language delay$(\Omega, \mathbf{J})$ if it can be extended to a word $w_2 = b_1 \dots b_n a w_1$ of notimplies$(\Omega, \mathbf{J})$ for a encoding an answer fact over a predicate in $\mathbf{J}$, and in addition w_2 can be extended to a word $w_3 = w_2 v_1 \dots v_m$ of implies$(\Omega, \mathbf{J})$. This means that w_3 encodes a stream S and a fact α holding at $\tau - d$ such that $\Omega \cup S \models \alpha$, and w_2 encodes that $\Omega \cup S\restriction_{[0,\tau]} \not\models \alpha$. Thus, the length d of w_1 is not a valid delay for Ω, and hence words in delay$(\Omega, \mathbf{J})$ witness the non-validity of delays via their length.

Intuitively, the language implies$(\Omega, \mathbf{J})$ will contain all words representing an instance $\langle S, \alpha \rangle$ such that $\Omega \cup S \models \alpha$; in this way, the language captures the logical $\mathbf{J}$-consequences of Ω when applied to an arbitrary stream. Dually, the language notimplies$(\Omega, \mathbf{J})$ will contain all words representing $\langle S, \alpha \rangle$ such that $\Omega \cup S \not\models \alpha$. Finally, as illustrated in Fig. 3, language delay$(\Omega, \mathbf{J})$ is defined in terms of languages implies$(\Omega, \mathbf{J})$ and notimplies$(\Omega, \mathbf{J})$ so that it contains a word of length d if and only if d is not a valid delay for Ω with respect to $\mathbf{J}$.

Definition 5.1. Let Ω be an object-free program, let $\mathbf{E}'$ be the set of all EDB predicates in Ω, let $\mathbf{J}$ be a set of IDB predicates in Ω, and let □ be a fresh symbol. We write $\Sigma(\Omega, \mathbf{J})$ for the set $2^{\Omega} \times (\mathbf{J} \cup \{\square\})$. Elements of $\Sigma(\Omega, \mathbf{J})$ are called *letters*. A letter $\langle E, b \rangle$ is an *answer letter* if $b \neq \square$. For $E_0, \dots, E_n \in 2^{\mathbf{E}'}$, $i \in [0, n]$, and $P \in \mathbf{J}$, we write $\langle i, P, E_0, \dots, E_n \rangle$ for the word $\langle E_0, b_0 \rangle \dots \langle E_n, b_n \rangle$ over $\Sigma(\Omega, \mathbf{J})$ where $b_i = P$ and $b_j = \square$ for each $j \neq i$.

We define the following languages over the aforementioned alphabet:

- implies$(\Omega, \mathbf{J})$ consists of each word $\langle i, P, E_0, \dots, E_n \rangle$ such that $\Omega \cup D \models P(i)$, where $D = \{A(j) \mid A \in E_j, 0 \leq j \leq n\}$.
- notimplies$(\Omega, \mathbf{J})$ consists of each word $\langle i, P, E_0, \dots, E_n \rangle$ such that $\Omega \cup D \not\models P(i)$, where $D = \{A(j) \mid A \in E_j, 0 \leq j \leq n\}$.
- delay$(\Omega, \mathbf{J})$ consists of each word u_1 for which there is a word u_2 ending with an answer letter such that $u_2 u_1$ is in prefimplies$(\Omega, \mathbf{J}) \cap$ notimplies$(\Omega, \mathbf{J})$, for prefimplies$(\Omega, \mathbf{J})$ the language containing all prefixes of words in implies$(\Omega, \mathbf{J})$.

The following theorem characterises program containment, delay validity, and delay existence for arbitrary programs (i.e., not necessarily object-free) in terms of the languages in Definition 5.1.

Theorem 5.2. *Let Π_1 and Π_2 be programs with the same set of EDB predicates, and let $\mathbf{I}$ be the set of IDB predicates in Π_1. Furthermore, let $\Omega_1 = \zeta(\Pi_1)$, let $\Omega_2 = \zeta(\Pi_2)$, and let $\mathbf{J} = \zeta(\mathbf{I})$. Then, the following statements hold:*

1. *$\Pi_1 \sqsubseteq \Pi_2$ if and only if* implies$(\Omega_1, \mathbf{J}) \cap$ notimplies$(\Omega_2, \mathbf{J}) = \emptyset$.
2. *A non-negative integer d is a valid delay for Π_1 if and only if each word in* delay$(\Omega_1, \mathbf{J})$ *is of length at most $d - 1$.*
3. *Π_1 has a valid delay if and only if* delay$(\Omega_1, \mathbf{J})$ *is a finite language.*

Proof. We show each of the statements in the theorem.

Let $\Pi_1 \sqsubseteq \Pi_2$ and let u be a word in implies$(\Omega_1, \mathbf{J})$ of the form $\langle i, P_{\mathbf{a}}, E_0, \dots, E_n \rangle$. By the definition of implies$(\Omega_1, \mathbf{J})$, we have $\Omega_1 \cup \zeta(D) \models P_{\mathbf{a}}(i)$ for $D = \{A(\mathbf{o}, j) \mid A_{\mathbf{o}} \in E_j, 0 \leq j \leq n\}$, and hence $\Pi_1 \cup D \models P(\mathbf{a}, i)$. Since $\Pi_1 \sqsubseteq \Pi_2$, we have $\Pi_2 \cup D \models P(\mathbf{a}, i)$, hence $\Omega_2 \cup \zeta(D) \models P_{\mathbf{a}}(i)$ and $u \notin$ notimplies$(\Omega_2, \mathbf{J})$. For the converse direction, assume implies$(\Omega_1, \mathbf{J}) \cap$ notimplies$(\Omega_2, \mathbf{J}) = \emptyset$ and let $\Pi_1 \cup D \models P(\mathbf{a}, i)$ for i a time point, D a finite set of EDB predicates and P an IDB predicate. Let n be the largest time point in D, and let $E_0, \dots, E_n$ be sets of predicates such that $\zeta(D) = \{A_{\mathbf{o}}(j) \mid A_{\mathbf{o}} \in E_j, 0 \leq j \leq n\}$. Consider the word $u = \langle i, P_{\mathbf{a}}, E_0, \dots, E_n \rangle$, which belongs to implies$(\Omega_1, \mathbf{J})$, since $\Omega_1 \cup \zeta(D) \models P_{\mathbf{a}}(i)$. By our assumption, we have that $u \notin$ notimplies$(\Omega_2, \mathbf{J})$, hence $\Omega_2 \cup \zeta(D) \models P_{\mathbf{a}}(i)$ by the definition of notimplies$(\Omega_2, \mathbf{J})$, and hence $\Pi_2 \cup D \models P(\mathbf{a}, i)$, as required.

For the second statement, assume that $u_1 \in$ delay$(\Omega_1, \mathbf{J})$ and $|u_1| \geq d$; we show that d is not a valid delay for Π_1. Since $u_1 \in$ delay$(\Omega_1, \mathbf{J})$, there is u_2 ending with an answer letter and satisfying $u_2 u_1 \in$ prefimplies$(\Omega_1, \mathbf{J}) \cap$ notimplies$(\Omega_1, \mathbf{J})$. Since $u_2 u_1 \in$ notimplies$(\Omega_1, \mathbf{J})$, there are sets $E_1, \dots, E_n$ of predicates, an integer $p \in [1, n]$, and a predicate $P_{\mathbf{a}}$ such that

$u_2u_1 = \langle i, P_{\mathbf{a}}, E_0, \ldots, E_n\rangle$. Since u_2 ends with an answer letter and u_2u_1 has exactly one answer letter, we have that u_1 is of the form $\langle E_{i+1}, \square\rangle, \ldots, \langle E_n, \square\rangle$. Furthermore, since $u_2u_1 \in \mathsf{prefimplies}(\Omega_1, \mathbf{J})$, there must exist u_3 such that $u_2u_1u_3 \in \mathsf{implies}(\Omega_1, \mathbf{J})$; furthermore, u_3 will be of the form $\langle E_{n+1}, \square\rangle, \ldots, \langle E_m, \square\rangle$ for some $m \geq n$. Let $S = \{A(\mathbf{o}, j) \mid A_{\mathbf{o}} \in E_j,\ 0 \leq j \leq m\}$. It follows that $\Omega_1 \cup \zeta(S) \models P_{\mathbf{a}}(i)$ since $u_2u_1u_3 \in \mathsf{implies}(\Omega_1, \mathbf{J})$, and hence $\Pi_1 \cup S \models P(\mathbf{a}, i)$; furthermore, $\Omega_1 \cup \zeta(S)\restriction_{[0,n]} \not\models P_{\mathbf{a}}(i)$ since $u_2u_1 \in \mathsf{notimplies}(\Omega_1, \mathbf{J})$, and hence $\Pi_1 \cup S\restriction_{[0,n]} \not\models P(\mathbf{a}, i)$. Since $S\restriction_{[0,i+d]} \subseteq S\restriction_{[0,n]}$ (note that $n - i \geq d$ is the length of u_1), it follows that $\Pi_1 \cup S\restriction_{[0,i+d]} \not\models P(\mathbf{a}, i)$ by the monotonicity of first-order logic, and hence d is not a valid delay for Π_1.

For the other direction, assume that d is not a valid delay for Π_1; we show that $\mathsf{delay}(\Omega_1, \mathbf{J})$ contains a word of length at least d. By the definition of valid delay, there exists a stream S, a predicate P, a tuple of objects $\mathbf{a}$, and a time point i with $i \geq d$ such that $\Pi_1 \cup S \models P(\mathbf{a}, i - d)$ and $\Pi_1 \cup S\restriction_{[0,i]} \not\models P(\mathbf{a}, i - d)$. By compactness of first-order logic, we can assume w.l.o.g. that S is finite. Let n be the maximum time point occurring in S and let $E_0, \ldots, E_n$ be sets of predicates such that $\zeta(S) = \{A_{\mathbf{o}}(i) \mid A_{\mathbf{o}} \in E_i,\ 0 \leq i \leq n\}$. Let $u_1 = \langle i - d, P_{\mathbf{a}}, E_0, \ldots, E_n\rangle$, and let $u_2 = \langle i - d, P_{\mathbf{a}}, E_0, \ldots, E_i\rangle$. We have that $\Pi_1 \cup S\restriction_{[0,i]} \not\models P(\mathbf{a}, i - d)$ implies $\Omega_1 \cup \zeta(S)\restriction_{[0,i]} \not\models P_{\mathbf{a}}(i - d)$, and hence $u_2 \in \mathsf{notimplies}(\Omega_1, \mathbf{J})$. Furthermore, $\Pi_1 \cup S \models P(\mathbf{a}, i - d)$ implies $\Omega_1 \cup \zeta(S) \models P_{\mathbf{a}}(i - d)$, and hence $u_1 \in \mathsf{implies}(\Omega_1, \mathbf{J})$. Thus, $u_2 \in \mathsf{prefimplies}(\Omega_1, \mathbf{J})$ since u_2 is a prefix of u_1. Let $\sigma_0, \ldots, \sigma_n$ be such that $u_1 = \sigma_0, \ldots, \sigma_n$. Let $u_3 = \sigma_{i-d+1}, \ldots, \sigma_i$, and let $u_4 = \sigma_0, \ldots, \sigma_{i-d}$. It suffices to show $u_3 \in \mathsf{delay}(\Omega_1, \mathbf{J})$, for which it remains to show that the following two conditions hold: *(i)* the last letter of u_4 is an answer letter, and *(ii)* $u_4u_3 \in \mathsf{prefimplies}(\Omega_1, \mathbf{J}) \cap \mathsf{notimplies}(\Omega_1, \mathbf{J})$. The last letter of u_4 is σ_{i-d}, which is an answer letter by its definition and the definition of u_1. We have $u_4u_3 \in \mathsf{prefimplies}(\Omega_1, \mathbf{J}) \cap \mathsf{notimplies}(\Omega_1, \mathbf{J})$ since $u_4u_3 = u_2$ and we have already argued both $u_2 \in \mathsf{prefimplies}(\Omega_1, \mathbf{J})$ and $u_2 \in \mathsf{notimplies}(\Omega_1, \mathbf{J})$.

Finally, note that *Statement 3* is an immediate corollary of the second statement. □

In the following we will focus on constructing automata that accept the languages we just described. For this, it will be helpful to restrict ourselves to programs in a normal form where the magnitude time offsets occurring in rules is at most one.

Definition 5.3. A program is *normal* if it contains only rules of the form $\bigwedge_i P_i(\mathbf{s}_i, t + k_i) \to P(\mathbf{s}, t)$ with each $k_i \in \{-1, 0, +1\}$.

The following two theorems show that our restriction to normal programs is without loss of generality. Theorem 5.4 is for arbitrary programs, and Theorem 5.5 is a refinement for object-free programs. In fact, Theorem 5.4 assumes an object domain with at least two objects, and it considers a normalisation that always introduces object variables. These two aspects of the theorem clash with the object-free case, thus motivating a separate theorem for object-free programs.

Theorem 5.4. *Assume that the object domain contains at least two objects. Let Π_1 be a program and let $\mathbf{I}_1$ be the set of IDB predicates in Π_1. Then, there exists a normal program Π_2 such that the following conditions hold, where $\Omega_1 = \zeta(\Pi_1)$, $\Omega_2 = \zeta(\Pi_2)$ and $\mathbf{J}_1 = \zeta(\mathbf{I}_1)$:*

1. $\mathsf{implies}(\Omega_1, \mathbf{J}_1) = \mathsf{implies}(\Omega_2, \mathbf{J}_1)$;
2. $\mathsf{notimplies}(\Omega_1, \mathbf{J}_1) = \mathsf{notimplies}(\Omega_2, \mathbf{J}_1)$;
3. $\mathsf{delay}(\Omega_1, \mathbf{J}_1) = \mathsf{delay}(\Omega_2, \mathbf{J}_1)$.

Furthermore, Π_2 can be computed from Π_1 in polynomial time.

Proof. We show how to construct Π_2 from Π_1 and then prove that it satisfies the relevant properties. We first introduce a program Π_{succ} that defines the successor relation over (binary encodings of) numbers in the interval $[0, \max(\rho_f, \rho_b)]$, where ρ_f and ρ_b are the forward and backward radius of Π_1, respectively. Let m be the number of bits required to encode $\max(\rho_f, \rho_b)$. Program Π_{succ} consists of rules (28)–(31) and a rule of the form (32) for each $i \in [0, m - 1]$ where A is a fresh unary predicate, *Bit* is a fresh binary predicate, *succ* is a fresh $(2m + 1)$-ary predicate, $\bar{0}$ and $\bar{1}$ are two distinct objects—intuitively standing for zero and one, respectively—each x_j is a fresh object variable, each $\mathbf{x}_j$ is the list of variables $x_1, \ldots, x_j$, each $\bar{\mathbf{1}}_j$ and each $\bar{\mathbf{0}}_j$ is the list of $\bar{1}$'s and $\bar{0}$'s, respectively, having length $m - j - 1$:

$$A(t - 1) \to A(t) \tag{28}$$

$$A(t + 1) \to A(t) \tag{29}$$

$$A(t) \to \mathit{Bit}(\bar{0}, t) \tag{30}$$

$$A(t) \to \mathit{Bit}(\bar{1}, t) \tag{31}$$

$$\textstyle\bigwedge_{j=1}^{i} \mathit{Bit}(x_j, t) \to \mathit{succ}(\mathbf{x}_i, \bar{0}, \bar{\mathbf{1}}_i, \mathbf{x}_i, \bar{1}, \bar{\mathbf{0}}_i, t). \tag{32}$$

The key property of Π_{succ} is that, for each time point τ, and each time point τ', $\Pi_{\mathrm{succ}} \cup \{A(\tau)\} \models \mathit{succ}(\mathbf{i}, \mathbf{j}, \tau')$ if and only if $\mathbf{i}$ and $\mathbf{j}$ are m-tuples over $\{\bar{0}, \bar{1}\}$ encoding two numbers $i, j \in [0, 2^m]$ such that and $i + 1 = j$.

Then, the normal program Π_2 is obtained by extending Π_{succ} with the following rules for each predicate P occurring in Π_1, where P^+ and P^- are fresh predicates uniquely associated with P and having the arity of P increased by m:

$$P(\mathbf{x}, t) \rightarrow A(t) \tag{33}$$

$$P(\mathbf{x}, t) \rightarrow P^+(\mathbf{x}, \bar{\mathbf{0}}, t) \tag{34}$$

$$P(\mathbf{x}, t) \rightarrow P^-(\mathbf{x}, \bar{\mathbf{0}}, t) \tag{35}$$

$$P^+(\mathbf{x}, \mathbf{y}, t-1) \wedge succ(\mathbf{y}, \mathbf{z}, t) \rightarrow P^+(\mathbf{x}, \mathbf{z}, t) \tag{36}$$

$$P^-(\mathbf{x}, \mathbf{y}, t+1) \wedge succ(\mathbf{z}, \mathbf{y}, t) \rightarrow P^-(\mathbf{x}, \mathbf{z}, t), \tag{37}$$

and the following rule for each rule in Π_1 of the form $\bigwedge_i P_i(\mathbf{u}_i, t + k_i) \rightarrow P(\mathbf{u}, t + k)$, where $\mathbf{b}_i$ is the binary encoding of $|k_i - k|$ and $s_i \in \{+, -\}$ is the sign of $k_i - k$ (plus if zero):

$$\bigwedge_i P_i^{s_i}(\mathbf{u}_i, \mathbf{b}_i, t) \rightarrow P(\mathbf{u}, t). \tag{38}$$

We claim that $\Pi_1 \cup D \models \alpha$ if and only if $\Pi_2 \cup D \models \alpha$, for each set D of facts and each fact α over predicates in Π_1. The first two statements in the theorem follow as a straightforward consequence of this claim, the properties of ξ, and the definition of the relevant languages; in turn, the third statement follows by the definition of the delay language and the previous two statements.

It is straightforward to show, by induction on i from 0 to $\max(\rho_f, \rho_b)$, that

$$\Pi_2 \cup D \models P(\mathbf{o}, \tau) \text{ if and only if } \Pi_2 \cup D \models \{P^+(\mathbf{o}, \mathbf{i}, \tau - i), P^-(\mathbf{o}, \mathbf{i}, \tau + i)\} \tag{39}$$

for each P in Π_1, each object tuple $\mathbf{o}$, and each set D of facts over predicates in Π_1, with $\mathbf{i}$ the binary encoding of i.

Assume now that $\Pi_1 \cup D \models \alpha$. Let δ be a derivation of α from $\Pi_1 \cup D$. We show $\Pi_2 \cup D \models \alpha$ by induction on the height h of δ. In the base case, we have $h = 0$ and hence $\alpha \in \Pi_1 \cup D$. But then, since Π_1 does not mention time points, $\alpha \in D$, and hence $\Pi_2 \cup D \models \alpha$. In the inductive case, we have $h > 0$, and we assume that the claim holds for $h - 1$. Let r be the rule labelling the root of δ, and let r' be a rule in Π_1 such that r is an instance of r'. Then r' has the form $\bigwedge_{i=1}^m P_i(\mathbf{u}_i, t + k_i) \rightarrow P(\mathbf{u}, t + k)$ and r is of the form $\bigwedge_{i=1}^m P_i(\mathbf{o}_i, \tau - k + k_i) \rightarrow P(\mathbf{o}, \tau)$ (where τ and $\tau - k + k_i$ are time points and $P(\mathbf{o}, \tau) = \alpha$). By construction, Π_2 contains a rule of the form $\bigwedge_{i=1}^m P_i^{s_i}(\mathbf{u}_i, \mathbf{b}_i, t) \rightarrow P(\mathbf{u}, t)$ where s_i is the sign of $k_i - k$ and $\mathbf{b}_i$ is the binary encoding of $|k_i - k|$. Thus, it suffices to show $\Pi_2 \cup D \models P_i^{s_i}(\mathbf{o}, \mathbf{b}_i, \tau)$ for each $i \in [1, m]$. To this end, note that, by assumption, for each $i \in [1, m]$, we have $\Pi_1 \cup D \models P_i(\mathbf{o}_i, \tau - k + k_i)$, hence $\Pi_2 \cup D \models P_i(\mathbf{o}_i, \tau - k + k_i)$ by the inductive hypothesis, and hence $\Pi_2 \cup D \models P_i^{s_i}(\mathbf{o}_i, \mathbf{b}_i, \tau)$ by (39).

Assume $\Pi_2 \cup D \models \alpha$. Let δ be a derivation of α from $\Pi_2 \cup D$, and let $\sharp\delta$ be the maximum number of nodes on a root-to-leaf path in δ that are labelled with an instance of a rule of the form (38). We show $\Pi_1 \cup D \models \alpha$ by induction on $\sharp\delta$. In the base case, we have $\sharp\delta = 0$, and hence $\alpha \in D$. Consequently, $\Pi_1 \cup D \models \alpha$, as required. In the inductive case, let r be the rule labelling the root of δ, and let r' be a rule in Π_2 such that r is an instance of r'. By assumption, r' has the form $\bigwedge_{i=1}^m P_i^{s_i}(\mathbf{u}_i, \mathbf{b}_i, t) \rightarrow P(\mathbf{u}, t)$, and thus r has the form $\bigwedge_{i=1}^m P_i^{s_i}(\mathbf{o}_i, \mathbf{b}_i, \tau) \rightarrow P(\mathbf{o}, \tau)$. Moreover, Π contains a rule of the form $\bigwedge_{i=1}^m P_i(\mathbf{u}_i, t + k_i) \rightarrow P(\mathbf{u}, t + k)$ such that s_i the sign of $k_i - k$ and $\mathbf{b}_i$ the binary encoding of $|k_i - k|$. Therefore, it suffices to show $\Pi_1 \cup D \models P_i(\mathbf{o}_i, \tau + k_i - k)$ for each $i \in [1, m]$. To this end, note that, by assumption, for each $i \in [1, m]$, we have $\Pi_2 \cup D \models P_i^{s_i}(\mathbf{o}_i, \mathbf{b}_i, \tau)$, and hence $\Pi_2 \cup D \models P_i(\mathbf{o}_i, \tau + k_i - k)$ by (39). Moreover, by the definition of Π_2, the subderivation of δ deriving $P_i^{s_i}(\mathbf{o}_i, \mathbf{b}_i, \tau)$ must include a derivation δ' of $P_i(\mathbf{o}_i, \tau + k_i - k)$, and consequently $\sharp\delta' < \sharp\delta$. Hence, the inductive hypothesis applies to δ' yielding $\Pi_1 \cup D \models P_i(\mathbf{o}_i, \tau + k_i - k)$, as required. □

Theorem 5.5. *Let Ω_1 be an object-free program, and let $\mathbf{J}_1$ be the set of IDB predicates occurring in Ω_1. There exists an object-free normal program Ω_2 such that:*

1. $\mathsf{implies}(\Omega_1, \mathbf{J}_1) = \mathsf{implies}(\Omega_2, \mathbf{J}_1)$;
2. $\mathsf{notimplies}(\Omega_1, \mathbf{J}_1) = \mathsf{notimplies}(\Omega_2, \mathbf{J}_1)$;
3. $\mathsf{delay}(\Omega_1, \mathbf{J}_1) = \mathsf{delay}(\Omega_2, \mathbf{J}_1)$.

Furthermore, program Ω_2 can be computed from Ω_1 in time polynomial in the size of Ω_1 and the value of $\rho_f + \rho_b$, where ρ_f and ρ_b are the forward and backward radius of Ω_1, respectively.

Proof. The construction of Ω_2 from Ω_1 is similar to the one of Π_2 from Π_1 given in the proof of Theorem 5.4, with the difference that the one here does not involve objects. Let P^i be a fresh unary predicate uniquely associated with P and $i \in [-\rho_f, \rho_b]$. Program Ω_2 consists of the following rules for each predicate P occurring in Ω_1:

$$P(t) \to P^0(t) \tag{40}$$

$$P^i(t+1) \to P^{i+1}(t) \qquad \forall i \in [0, \rho_b - 1] \tag{41}$$

$$P^i(t-1) \to P^{i-1}(t) \qquad \forall i \in [-\rho_f + 1, 0] \tag{42}$$

and the following rule for each rule in Ω_1 of the form $\bigwedge_i P_i(t' + k_i) \to P(t' + k)$:

$$\bigwedge_i P_i^{k_i - k}(t') \to P(t'). \tag{43}$$

As in the proof of Theorem 5.4, Statements 1–3 follow from the claim that $\Omega_1 \cup D \models \alpha$ if and only if $\Omega_2 \cup D \models \alpha$, for each set D of facts and each fact α over predicates $\mathbf{J}_1$. To see that the claim holds, it suffices to draw a comparison with the construction of Theorem 5.4. Namely, rules (40)–(42) are analogous to rules (34)–(37) (which also require additional rules to count), and rules (43) correspond to rules (38). □

6. Automata construction

In this section, we show that all the languages given in Definition 5.1 can be recognised by finite automata. For convenience we assume that programs have already been grounded and normalised, so let us consider an *object-free normal program* Ω and a subset $\mathbf{J}$ of its IDB predicates.

Our first step will be to define a non-deterministic automaton $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ recognising $\mathsf{implies}(\Omega, \mathbf{J})$. Intuitively, the automaton checks whether an input word corresponds to a finite stream prefix S and fact α such that $\Omega \cup S \models \alpha$ by guessing a finite set of facts M such that $\Omega \cup S \models M$ and $\alpha \in M$. The key challenge in the construction of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ is making sure that each fact in M can be derived from $\Omega \cup S$; indeed, a derivation of a fact from $\Omega \cup S$ may involve facts at many different time points, whereas $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ can only check whether a fact holding at τ is entailed by facts holding at $[\tau - 1, \tau + 1]$. Thus, the transition relation of the automaton will ensure that M satisfies $\Omega \cup M\restriction_{\tau-1} \cup S\restriction_{\tau} \cup M\restriction_{\tau+1} \models M\restriction_{\tau}$ for each such τ. This alone, however, is not enough to ensure $\Omega \cup S \models M$; for instance, for $\Omega = \{A(t-1) \to B(t), B(t+1) \to A(t)\}$ and $S = \emptyset$, set $M = \{A(0), B(1)\}$ satisfies the aforementioned property yet is clearly not entailed by $\Omega \cup S$. It is easily seen that all such spurious sets contain a fact that can only be derived using itself as an assumption; to exclude these sets, we will enrich the states of the automaton with dependency information and require that no fact depends on itself in a minimal derivation. The automaton encodes time by exploiting the natural order of states in a run; for instance, the automaton encodes $M\restriction_{\tau}$ using the set M_τ of predicates occurring in $M\restriction_{\tau}$. Due to such encoding of sets of facts as sets of predicates, it is convenient to introduce the notation $X(\tau) = \{P(\tau) \mid P \in X\}$ for X a set of unary predicates and τ a time point; for instance, this notation allows us to easily write the relationship $M\restriction_{\tau} = M_\tau(\tau)$ between $M\restriction_{\tau}$ and M_τ. Fig. 4 illustrates these concepts.

Definition 6.1. Let Ω be an object-free normal program, let $\mathbf{P}$ be the set of all predicates in Ω, and let $\mathbf{J} \subseteq \mathbf{P}$ be a set of IDB predicates. Then, $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ is the non-deterministic finite automaton over alphabet $\Sigma(\Omega, \mathbf{J})$ defined as follows.

- A state is a 5-tuple $\langle a, X, Y, \Omega', \Gamma \rangle$ where $a \in \{1, 2, 3\}$, $X, Y \subseteq \mathbf{P}$, $\Omega' \subseteq \Omega \downarrow 1$ and $\Gamma \subseteq \mathbf{P} \times \mathbf{P}$.[3]
- The set of initial states consists of each state of the form $\langle 1, \emptyset, X, \emptyset, \emptyset \rangle$.
- The set of final states consists of each state of the form $\langle a, X, \emptyset, \Omega', \Gamma \rangle$ with $a \in \{2, 3\}$.
- The transition relation $\Delta \cup \Delta_\varepsilon$ is defined as follows. Δ consists of each transition $\langle a_1, X, Y, \Omega_1, \Gamma_1 \rangle \xrightarrow{\langle U, b \rangle} \langle a_2, Y, Z, \Omega_2, \Gamma_2 \rangle$ satisfying the following conditions:
  - (2.1) if $b \in \mathbf{I}$ then $b \in Y$, $a_1 = 1$, and $a_2 = 2$;
  - (2.2) if $b = \square$ then $a_1 = a_2 = 1$ or $a_1 = a_2 = 2$;
  - (2.3) $X(0) \cup U(1) \cup Z(2) \cup \Omega_2 \models Y(1)$;
  - (2.4) Γ_2 contains the transitive closure of the binary relation consisting of each pair $\langle P, R \rangle$ such that either
    - (2.4.1) $P(1)$ depends on $R(1)$ in Ω_2, or
    - (2.4.2) there exists a predicate T such that $P(1)$ depends on $T(0)$ in Ω_2 and $T(1)$ depends on $R(2)$ in Ω_1, or
    - (2.4.3) there exists $\langle T_1, T_2 \rangle \in \Gamma_1$ such that $P(1)$ depends on $T_1(0)$ in Ω_2 and $T_2(1)$ depends on $R(2)$ in Ω_1.
  - (2.5) there exists no predicate P such that $\langle P, P \rangle \in \Gamma_2$;

  In turn, Δ_ε consists of each transition $\langle a, X, Y, \Omega_1, \Gamma_1 \rangle \xrightarrow{\varepsilon} \langle 3, Y, Z, \Omega_2, \Gamma_2 \rangle$ satisfying the following conditions:
  - (3.1) $a \in \{2, 3\}$;
  - (3.2) $X(0) \cup Z(2) \cup \Omega_2 \models Y(1)$;
  - (3.3) Conditions (2.4) and (2.5) hold.

Automaton $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ computes in three stages. In the first stage, it expects to read a (non-ε) input letter and eventually an answer letter. In the second stage, it has read an answer letter, it expects to see no other answer letter, and

[3] We remind the reader that $\downarrow \tau$ is the substitution that maps all time variables to the specified time point τ.

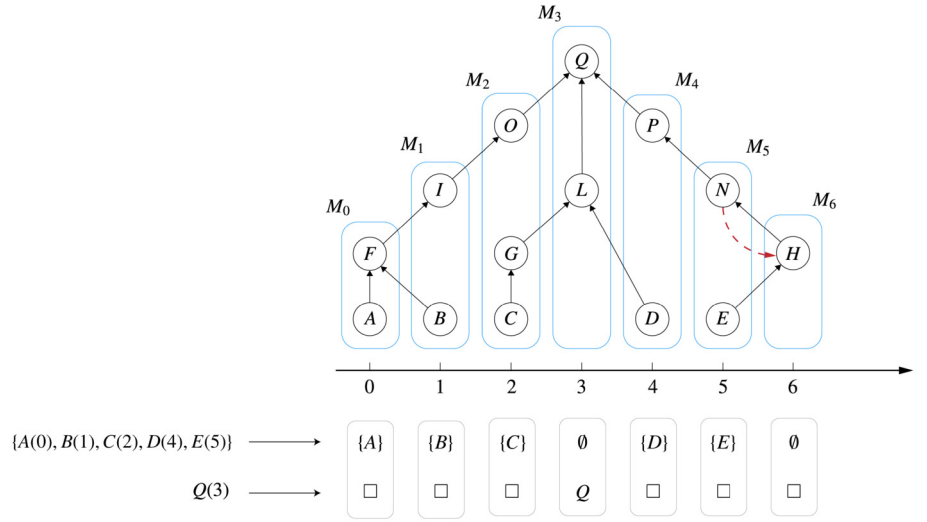


Fig. 4. An execution of the automaton $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ for some program Ω and some subset $\mathbf{J}$ of its IDB predicates. Below the timeline, we have an input word encoding an EDB stream $S = \{A(0), B(1), C(2), D(4), E(5)\}$ and the answer fact $Q(3)$. Above the timeline, we have a set of facts $M_0 \cup \cdots \cup M_6$ which encodes a derivation of $Q(3)$ from $S \cup \Omega$. Each predicate X in the figure aligned with a point τ of the timeline encodes the fact $X(\tau)$; for instance, A encodes $A(0)$. A box labelled with M_τ encloses the content of M_τ; for instance, the left-most box encodes the content $\{A, F\}$ of M_0. Arrows denote how IDB facts are derived from other facts, where multiple incoming arrows are to be interpreted as 'logical and'; for instance, $F(0)$ is derived by the rule $A(0) \wedge B(1) \rightarrow F(0)$. The automaton keeps track of the transitive closure of the graph induced by the derivation, and prevents cycles from appearing in the closure, so to have a well-formed derivation. For instance, when the automaton processes time 6, it may consider deriving $H(6)$ from $N(5)$, but it avoids to do so because this would yield a cycle as indicated by the red dashed arrow in the figure.

it has not performed any ε-transition yet. In the third stage, it has performed an ε-transition and it does not expect to see an input letter anymore.

The components of an automaton state $\langle a, X, Y, \Omega', \Gamma \rangle$ have the following meaning: the integer a describes in which of the three stages the automaton currently is; the two sets X and Y represent two consecutive subsets $M\restriction_\tau$ and $M\restriction_{\tau+1}$ of the set M described above; Ω' is an instance of Ω over the time interval $[0, 2]$ which encodes labels of the derivation the automaton (implicitly) builds; the pairs in Γ describe how the derived facts depend on each other, i.e., a pair $\langle P, R \rangle$ specifies that $P(\tau)$ depends on $R(\tau)$ for τ the (implicit) current time point. Time in X and Y is implicit in the position they occur in a run, whereas time in Ω' is relative and can be mapped to absolute time through the position Ω' occurs in a run; specifically, time 1 in Ω' stands for the current time point, time 0 for the previous time point, and 2 for the next time point. Each Ω' is subset of $\Omega \downarrow 1$, which denotes the instance of Ω obtained by substituting the time variable with 1.

The transitions of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ serve three main purposes. First, they progress through the different stages. Second, they enforce the aforementioned property $\Omega \cup M\restriction_{\tau-1} \cup S\restriction_\tau \cup M\restriction_{\tau+1} \models M\restriction_\tau$; in particular, the transitions in Δ read $S\restriction_\tau$ from the input word, and the ε-transitions in Δ_ε allow to reason beyond the time points of the input, implicitly taking $S\restriction_\tau = \emptyset$. Third, they incrementally maintain the dependency information in Γ and avoid the presence of cycles. Considering a transition $\langle \_, X, Y, \_, \_ \rangle \longrightarrow \langle \_, Y, Z, \_, \_ \rangle$, we can see that X represents the facts already checked to hold at the previous time point, Y represents the facts to be checked now, and Z represents a set of facts that we assume to hold for the next time point and we promise to check at the next transition. Following this line of reasoning, a state $\langle a, X, Y, \Omega', \Gamma \rangle$ is initial if no facts have been derived in the previous time points, and hence if it has $X = \Omega' = \Gamma = \emptyset$; and it is final if an answer fact has been read and all assumptions have been proved, and hence if it has $a \in \{2, 3\}$ and $Y = \emptyset$.

Theorem 6.2. *Automaton* $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ *recognises the language* $\mathsf{implies}(\Omega, \mathbf{J})$.

Proof. The proof is given in Appendix A. □

The automaton $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ can be computed in polynomial space because each of its elements (input letter, state, or transition) is of polynomial size, and we can guess a candidate element at a time and check whether it belongs to the automaton.

Theorem 6.3. *Automaton* $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ *can be computed in polynomial space with respect to the size of* Ω *for each* $\mathbf{J}$*; furthermore, the size of each input letter, each state, and each transition of* $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ *is polynomial in the size of* Ω.

Proof. Clearly, each input letter, state, and transition of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ is of polynomial size in Ω. We next show how to compute the transitions in Δ (see Definition 6.1) in polynomial space using a 'generate and filter' algorithm—the remain-

ing transitions, the input alphabet, the states, and the final states of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ can be computed similarly. Checking whether a triple $\langle a_1, X_1, Y_1, \Omega_1, \Gamma_1\rangle \xrightarrow{\langle U,b\rangle} \langle a_2, X_2, Y_2, \Omega_2, \Gamma_2\rangle$ is a transition of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ is feasible in polynomial time with respect to the size of the triple since checking Condition (2.3) amounts to checking fact entailment for propositional Horn clauses. We check each of these triples one by one while reusing space in each check and we output those triples corresponding to valid transitions. Clearly, this computation takes polynomial space since each considered triple is of polynomial size. □

We next define the automaton $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$, which recognises $\mathsf{notimplies}(\Omega, \mathbf{J})$. Similarly to the algorithm for fact entailment by Chomicki and Imieliński [19], the automaton checks whether an input word corresponds to a finite stream prefix S and fact α such that $\Omega \cup S \not\models \alpha$ by determining whether S can be extended to an infinite model M of $\Omega \cup S$ not containing α. To this end, we first define a family of automata $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J}), n]$, for $n \geq 0$, which accept words representing a stream prefix S and a fact α such that S can be extended to a partial model M_n that contains every fact in S, does not contain α, and is closed under the rules of Ω on the interval $[0, \tau_{\max} + n]$, for $\tau_{\max}$ the maximum time point in S. It is then argued that, when n exceeds a certain exponential bound in the number of predicates in Ω, any such partial model M_n exhibits a periodic pattern and can thus be extended to an infinite model of $\Omega \cup S$. We therefore define $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ as $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J}), n]$ for a value of n corresponding to the aforementioned exponential bound.

Definition 6.4. Let Ω be an object-free normal program over predicates $\mathbf{P}$, let $\mathbf{J} \subseteq \mathbf{P}$ be a set of IDB predicates, and let $n \geq 0$. Then $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J}), n]$ is the non-deterministic finite automaton over alphabet $\Sigma(\Omega, \mathbf{J})$ defined as follows.

- A state is a 4-tuple $\langle a, X, Y, i\rangle$ where $a \in \{1, 2, 3\}$, $X, Y \subseteq \mathbf{P}$, and $i \in [0, n]$.
- The set of initial states consists of each state in S of the form $\langle 1, \emptyset, X, 0\rangle$.
- The set of final states consists of each state in S of the form $\langle 3, X, Y, n\rangle$.
- The transition relation $\Delta \cup \Delta_\varepsilon$ is defined as follows. Δ consists of each transition $\langle a_1, X, Y, 0\rangle \xrightarrow{\langle U,b\rangle} \langle a_2, Y, Z, 0\rangle$ satisfying the following conditions:
  (2.1) if $b \in \mathbf{I}$ then $b \notin Y$, $a_1 = 1$, and $a_2 = 2$;
  (2.2) if $b = \square$ then $a_1 = a_2 = 1$ or $a_1 = a_2 = 2$;
  (2.3) $X(0) \cup Y(1) \cup Z(2) \models \Omega \downarrow 1 \cup U(1)$;
  In turn, Δ_ε consists of each transition $\langle a, X, Y, i\rangle \xrightarrow{\varepsilon} \langle 3, Y, Z, i+1\rangle$ satisfying the following conditions:
  (3.1) $a \in \{2, 3\}$;
  (3.2) $X(0) \cup Y(1) \cup Z(2) \models \Omega \downarrow 1$.

We define $N = 2^{2\cdot|\mathbf{P}|}$ and $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ as $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J}), N]$.

Although $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ shares many elements with $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$, it operates rather differently. Each transition of $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ checks, for increasing time points τ, whether the portion $M\restriction_{[\tau-1,\tau+1]}$ of the guessed facts M satisfies the program and input facts, while making sure that the answer fact α is not in M. In order to ensure that a sufficient number of time points have been checked after reading the last letter of the input, automaton $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ maintains a counter in its state which has value zero as long as the automaton performs non-ε-transitions and, then, it is incremented at every ε-transition.

Theorem 6.5. *Automaton* $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ *recognises the language* $\mathsf{notimplies}(\Omega, \mathbf{J})$.

Proof. The proof is given in Appendix A. □

We are now ready to show that automaton $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ can be computed in polynomial space.

Theorem 6.6. *Automaton* $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ *can be computed in polynomial space with respect to the size of* Ω *for each* $\mathbf{J}$*; furthermore, the size of each input letter, each state, and each transition of* $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ *is polynomial in the size of* Ω.

Proof. Clearly, the size of each input letter, each state, and each transition of $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ is polynomial in the size of Ω. Automaton $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ can be computed in polynomial space with the respect to the size of Ω through a 'generate and filter' algorithm similar to the one described in the proof of Theorem 6.3. Note that checking Conditions 2.3 and 3.2 amounts to model checking in propositional logic, which is feasible in polynomial time. □

Finally, we define an automaton $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ recognising the language $\mathsf{delay}(\Omega, \mathbf{J})$. To this end, we first define an automaton $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ for the language $\mathsf{prefimplies}(\Omega, \mathbf{J})$ and show that it can be computed in polynomial space with respect to the size of Ω. We then use $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ to define $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$, and show that $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ can also be computed in polynomial space.

Definition 6.7. Automaton $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$, for Ω an object-free normal program and $\mathbf{J}$ a set of IDB predicates in Ω, is obtained from $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ by redefining the final states as those occurring in some accepting run of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$.

Automaton $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ captures the prefix language of $\mathsf{implies}(\Omega, \mathbf{J})$ because, by construction, its accepting runs can be completed into accepting runs of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$.

Proposition 6.8. *Automaton* $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ *recognises* $\mathsf{prefimplies}(\Omega, \mathbf{J})$.

The construction of $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ requires checking reachability in the graph induced by the automaton. The following lemma shows that this is feasible in polynomial space using an adaptation of the well-known NLogSpace algorithm for graph reachability [32] to the case where the input graph is concisely given as a polynomial space computable function.

Lemma 6.9. *Let S be a set and g a total function from S to labelled directed graphs computable in polynomial space and such that, for each $x \in S$, the size of each node and each edge of $g(x)$ is polynomial in the size of x. The problem of checking, given some $x \in S$ and two nodes u, v of $g(x)$, whether v is reachable from u in $g(x)$ can be solved in polynomial space.*

Proof. Given $x \in S$ and nodes u, v in $g(x)$, we first establish the number n of nodes in $g(x)$ by enumerating and counting them. We then initialise a counter i to 0 and store u in a variable p. While $i < n$, we repeat the following loop. We check if $p = v$, in which case we accept. Otherwise, we guess an integer $j \in [1, m]$ and enumerate the edges of $g(x)$, picking the j-th edge $\langle a, \sigma, b \rangle$. If $a \neq p$, we reject. Otherwise we increment i by one, set p to b, and repeat the loop. If the loop terminates with $i = n$, we reject.

The correctness of the algorithm is immediate since v is reachable from u in $g(x)$ if and only if it is reachable by a path of length at most $n - 1$. Finally, we argue that the algorithm runs in polynomial space with respect to the size of x. By assumption, we can enumerate each node and each edge of $g(x)$ in polynomial space, and each such node and edge is of polynomial size. It follows that polynomial memory suffices to represent p, a, σ, and b. It also follows that the number of nodes and edges is at most exponential, and hence polynomial memory suffices for the counters i and j. □

The result in Lemma 6.9 makes it easy to compute $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ from $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$.

Lemma 6.10. *Automaton* $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ *can be computed in polynomial space with respect to the size of Ω for each $\mathbf{J}$; furthermore, the size of each input letter, each state, and each transition of* $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ *is polynomial in the size of Ω.*

Proof. The second claim follows by Theorem 6.3 as the input alphabet, the states, and the transitions of $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ coincide with those of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$. To compute the final states of $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$, we enumerate all triples $\langle s, s', s'' \rangle$ of states of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ where s is initial and s'' is final; for each such triple, we check whether s' is reachable from s and whether s'' is reachable from s' in the labelled directed graph induced by the transitions of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$; if both checks succeed, we add s' to the set of final states of $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$, and otherwise we ignore the triple. This can be done in polynomial space with respect to the size of Ω by Lemma 6.9 and Theorem 6.3. □

The automaton for the delay language can now be defined starting from the automata we have defined above.

Definition 6.11. Automaton $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ is obtained from the product automaton $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})] \times \mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$. by introducing a fresh initial state s_I and a transition $s_\mathrm{I} \xrightarrow{\varepsilon} s$ whenever the product automaton admits a run ending with a transition into s labelled with an answer letter.

The construction of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$—apart from capturing the intersection of two languages by means of the well-known cross-product construction—captures string suffixes similarly to the way automaton $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})]$ captures prefixes. One difference is that here we are interested in specific suffixes—the ones starting right after an answer letter.

Theorem 6.12. *Automaton* $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ *recognises* $\mathsf{delay}(\Omega, \mathbf{J})$ *and can be computed in polynomial space with respect to the size of Ω for each $\mathbf{J}$; furthermore, the size of each input letter, state, and transition is polynomial in the size of Ω.*

Proof. The only statement in the theorem that doesn't follow directly from previous results and the construction of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ is that the newly introduced transitions can be computed in polynomial space. To do so, we enumerate the transitions $s \xrightarrow{\sigma} s'$ in the product automaton $\mathcal{A}[\mathsf{prefimplies}(\Omega, \mathbf{J})] \times \mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ and output the transition $s_\mathrm{I} \xrightarrow{\varepsilon} s'$ if σ is an answer letter and s is reachable from an initial state of the product automaton; this check is feasible in polynomial space by Lemma 6.9. □

7. Complexity upper bounds

In this section, we provide complexity upper bounds for our reasoning problems assuming that all the EDB streams (and sets of EDB facts) are over the same finite object domain. In the next section, we will provide matching lower bounds.

We start by establishing upper bounds for delay existence and validity. Intuitively, since the length n of a word in a delay language witnesses the non-validity of n as a delay, we can check: *(i) delay existence* by checking whether words in the language are of bounded length (i.e., whether the language is finite), and hence whether the corresponding automaton has no cycle reachable from the initial state, involving the final state, and containing a non-ε-transition; *(ii) validity of a delay d* by checking whether words in the language have length at most $d-1$, and hence checking for absence of a path in the corresponding automaton from the initial state to a final state involving at least $d-1$ non-ε-transitions. The proof also involves normalisation and grounding over the fixed object domain, since our automata are applicable to object-free normal programs only.

Theorem 7.1. *Delay existence and delay validity are in* ExpSpace *if the object domain of streams is considered fixed. The two problems are in* PSpace *when restricted to object-free programs with offsets coded in unary.*

Proof. We prove upper bounds for delay existence and delay validity separately. For each problem, we first show a PSpace upper bound for the case of a normal object-free program Ω having IDB predicates $\mathbf{J}$. Let us call it the *core* case. Then, we show the claimed bounds by reduction to the core case.

Delay existence. By Theorem 5.2, program Ω admits a valid delay if and only if $\mathsf{delay}(\Omega, \mathbf{J})$ is finite. Furthermore, by Theorem 6.12, the automaton $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ recognises $\mathsf{delay}(\Omega, \mathbf{J})$, and $\mathsf{delay}(\Omega, \mathbf{J})$ is finite if and only if each cycle of each accepting run of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ is over ε-transitions only. Therefore, to show a PSpace upper bound, it suffices to provide a non-deterministic polynomial space algorithm that takes Ω as input and accepts if and only if there is an accepting run of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ having a cycle that contains a non-ε transition. The algorithm computes the initial state s_0 of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$, after which it guesses a pair of states $\langle s, s' \rangle$, and accepts if s and s' are states in $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ such that s' is a final state, s is reachable from s_0, s' is reachable from s, and s is reachable from s through a path containing at least one edge whose label is different from ε. The algorithm can be realised in polynomial space. Indeed, by Theorem 6.12, we can check in polynomial space whether a state belongs to the automaton and whether it is a final state; furthermore, each of the reachability checks can be performed in polynomial space in the combined size of Ω, s_0, s, and s' by Lemma 6.9. Now we show the other bounds. The problem restricted to object-free programs with offsets in unary (resp., binary) is in PSpace (ExpSpace) since it reduces to the core case in polynomial (exponential) time by Theorem 5.5. If the object domain is fixed and contains at most one object, we can ground in polynomial time and hence the problem is in ExpSpace since it reduces to the previous case; otherwise, the problem is in ExpSpace since it reduces to the core case via normalisation and grounding in exponential time by Theorem 5.4.

Delay validity. Let d be a non-negative integer. By Theorem 5.2, d is a valid delay for Ω if and only if each word in $\mathsf{delay}(\Omega, \mathbf{J})$ is of length at most $d-1$. Since automaton $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ recognises $\mathsf{delay}(\Omega, \mathbf{J})$ by Theorem 6.12, d is a valid delay for Ω if and only if each accepting run of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ consists of at most $d-1$ non-ε transitions. To show the PSpace upper bound, it suffices to provide a non-deterministic polynomial space algorithm for the complement of the problem, and hence an algorithm that takes Ω and d as input and accepts iff there is an accepting run of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ consisting of at least d non-ε transitions. The algorithm computes Ω and the initial state s_0 of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$; then, it guesses a state s and accepts if s is a final state of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ and it is reachable from s_0 via a path containing at least d non-ε edges. The aforementioned reachability check is feasible in polynomial space in the combined size of Ω, s_0, s, and d by a straightforward generalisation of Lemma 6.9, and hence in space polynomial in the combined size of Ω and d, since s_0 and s are of size polynomial in the size of Ω by Theorem 6.12. The other bounds are easily obtained via normalisation and grounding using the same arguments as in the case of delay existence. □

Theorem 7.2. *Program containment is in* ExpSpace *if the object domain of streams is considered fixed. It is in* PSpace *when restricted to object-free programs with offsets coded in unary.*

Proof. We prove the PSpace upper bound for containment of normal object-free programs, and then the claimed bounds are easily obtained via normalisation and grounding using the same arguments as in Theorem 7.1. Let Ω_1 and Ω_2 be normal object-free programs with the same set of EDB predicates. By Theorem 5.2, $\Omega_1 \sqsubseteq \Omega_2$ if and only if $\mathsf{implies}(\Omega_1, \mathbf{J}) \cap \mathsf{notimplies}(\Omega_2, \mathbf{J}) = \emptyset$ with $\mathbf{J}$ the set of IDB predicates in Ω_1. Furthermore automaton $\mathcal{A}[\mathsf{implies}(\Omega_1, \mathbf{J})]$ recognises $\mathsf{implies}(\Omega_1, \mathbf{J})$ by Theorem 6.2, and automaton $\mathcal{A}[\mathsf{notimplies}(\Omega_2, \mathbf{J})]$ recognises $\mathsf{notimplies}(\Omega_2, \mathbf{J})$ by Theorem 6.5. Let $\mathcal{A}$ be the product automaton $\mathcal{A}[\mathsf{implies}(\Omega_1, \mathbf{J})] \times \mathcal{A}[\mathsf{notimplies}(\Omega_2, \mathbf{J})]$. We have that $\mathcal{A}$ recognises $\mathsf{implies}(\Omega_1, \mathbf{J}) \cap \mathsf{notimplies}(\Omega_2, \mathbf{J})$, and hence $\mathsf{implies}(\Omega_1, \mathbf{J}) \cap \mathsf{notimplies}(\Omega_2, \mathbf{J}) = \emptyset$ if and only if $\mathcal{A}$ has no accepting run. Summing up, we have that $\Omega_1 \sqsubseteq \Omega_2$ if and only if $\mathcal{A}$ has no accepting run. Clearly, checking this is feasible in polynomial space since reachability between an initial state and a final state can be checked in space polynomial in the combined size of Ω_1 and Ω_2. □

The following upper bound proof for window size validity amounts to modelling the definition of valid window size as the containment $\Pi_1 \sqsubseteq \Pi_2$ where Π_1 and Π_2 capture the left-hand and right-hand side, respectively, of the inclusion in Definition 3.4. The construction is as follows. We first define a program Π_{count} that, given a fact $\mathit{Now}(\tau)$, derives facts spanning the interval $[\tau - m, \tau]$. Specifically, it first derives the binary encoding of zero at τ; then, if it has derived the binary encoding of an integer n at time τ', either $n = m$ and it stops, or it derives the binary encoding of $n+1$ at time $\tau' - 1$. As a result, the program derives the encoding of every $i \in [0, m]$ at time $\tau - i$, and no fact at time $\tau' < \tau - m$. In essence, Π_{count} labels time points with their distance from the time point τ where $\mathit{Now}(\tau)$ holds—note that we guarantee τ to be unique via the flooding technique. We use the labelling above to define a program $\Pi_{\text{intervals}}$, which further labels time points according to whether they belong to each of the intervals mentioned in the definition of window size validity. Finally, we include in the construction two copies of the original programs, with each copy restricted so as to consider facts over different intervals, following the definition of window size validity.

Theorem 7.3. *Window size validity is in* ExpSpace *if the object domain of streams is considered fixed. It is in* PSpace *when restricted to object-free programs with offsets coded in unary.*

Proof. We provide a logarithmic space reduction from window size validity to program containment. The reduction preserves object-freeness, and hence the bounds follow by Theorem 7.2. Our reduction φ maps an instance $\langle \Pi, d, w \rangle$ of window size validity to a pair $\langle \Pi_1, \Pi_2 \rangle$ of programs (defined below) if $w < d + \rho$, with ρ be the maximum forward radius of Π, and to $\langle \emptyset, \emptyset \rangle$ otherwise. We next show how programs Π_1 and Π_2 in our reduction φ are constructed from Π, d and w. Let *Now* and *Continue* be fresh unary predicates. Let k be the number of bits required to encode w. For each $i \in [1, k]$, let Zero_i, One_i, Lower_i, Middle_i and Upper_i be fresh unary predicates. Furthermore, for each $i \in [1, k]$, let B_i be Zero_i if the i-th bit of the binary encoding of w is one and let it be One_i otherwise. We define Π_{count} as the program consisting of the following rules.

$$\mathit{Now}(t) \to \mathit{Zero}_i(t) \qquad \text{for each } i \in [1, k] \tag{44}$$
$$\mathit{One}_1(t) \to \mathit{Lower}_1(t) \tag{45}$$
$$\mathit{Lower}_{i-1}(t) \wedge \mathit{One}_i(t) \to \mathit{Lower}_i(t) \qquad \text{for each } i \in [2, k] \tag{46}$$
$$\mathit{Zero}_1(t) \to \mathit{Middle}_1(t) \tag{47}$$
$$\mathit{Lower}_{i-1}(t) \wedge \mathit{Zero}_i(t) \to \mathit{Middle}_i(t) \qquad \text{for each } i \in [2, k] \tag{48}$$
$$\mathit{Middle}_{i-1}(t) \to \mathit{Upper}_i(t) \qquad \text{for each } i \in [2, k] \tag{49}$$
$$\mathit{Upper}_{i-1}(t) \to \mathit{Upper}_i(t) \qquad \text{for each } i \in [2, k] \tag{50}$$
$$B_i(t) \to \mathit{Continue}(t) \qquad \text{for each } i \in [1, k] \tag{51}$$
$$\mathit{Continue}(t) \wedge \mathit{Lower}_i(t) \to \mathit{Zero}_i(t-1) \qquad \text{for each } i \in [1, k] \tag{52}$$
$$\mathit{Continue}(t) \wedge \mathit{Middle}_i(t) \to \mathit{One}_i(t-1) \qquad \text{for each } i \in [1, k] \tag{53}$$
$$\mathit{Continue}(t) \wedge \mathit{Upper}_i(t) \wedge \mathit{Zero}_i(t) \to \mathit{Zero}_i(t-1) \qquad \text{for each } i \in [1, k] \tag{54}$$
$$\mathit{Continue}(t) \wedge \mathit{Upper}_i(t) \wedge \mathit{One}_i(t) \to \mathit{One}_i(t-1) \qquad \text{for each } i \in [1, k] \tag{55}$$

The construction of Π_{count} ensures that, for each EDB stream S containing at most one fact about *Now*, each time point τ, each $m \in [0, w]$, and for predicates $B_1, \ldots, B_k$ where B_i is Zero_i if the i-th bit in the binary encoding of m is zero and One_i otherwise,

$$\Pi_{\text{count}} \cup S \models \{B_1(\tau), \ldots, B_k(\tau)\} \text{ if and only if } \mathit{Now}(\tau + m) \in S.$$

We next define $\Pi_{\text{intervals}}$ as the extension of Π_{count} with rules (56)–(64), where each predicate of the form $[\![T]\!]$ is a fresh unary predicate that intuitively stands for the time interval (or single time point) denoted by T; furthermore, for each $i \in [1, k]$, Bit_i is a fresh unary predicate, and C_i is Zero_i if the i-th bit of the binary encoding of d is zero and One_i otherwise—i.e., the C_i's encode d.

$$C_1(t) \wedge \cdots \wedge C_k(t) \to [\![\mathit{now} - d]\!](t) \tag{56}$$
$$\mathit{Zero}_i(t) \to \mathit{Bit}_i(t) \qquad \text{for each } i \in [1, k] \tag{57}$$
$$\mathit{One}_i(t) \to \mathit{Bit}_i(t) \qquad \text{for each } i \in [1, k] \tag{58}$$
$$\mathit{Bit}_1(t) \wedge \cdots \wedge \mathit{Bit}_k(t) \to [\![\mathit{now} - w, \mathit{now}]\!](t) \tag{59}$$
$$[\![\mathit{now} - d]\!](t) \to [\![0, \mathit{now} - d - 1]\!](t-1) \tag{60}$$

$$[\![0, now - d - 1]\!](t) \rightarrow [\![0, now - d - 1]\!](t - 1) \tag{61}$$

$$[\![0, now - d - 1]\!](t) \wedge [\![now - w, now]\!](t) \rightarrow [\![now - w, now - d - 1]\!](t) \tag{62}$$

$$Now(t) \rightarrow [\![0, now]\!](t) \tag{63}$$

$$[\![0, now]\!](t) \rightarrow [\![0, now]\!](t - 1) \tag{64}$$

Noting that $w \geq d$ by definition of the problem, it is easy to see that, by the construction of $\Pi_{\text{intervals}}$, for each EDB stream S containing at most one fact about *Now* and each time point τ, we have $Now(\tau) \in S$ if and only if all the following points hold:

1. $\Pi_{\text{intervals}} \cup S \models [\![now - d]\!](\tau - d)$;
2. $\Pi_{\text{intervals}} \cup S \models [\![now - w, now]\!](\tau')$ for each $\tau' \in [\tau - w, \tau]$;
3. $\Pi_{\text{intervals}} \cup S \models [\![0, now - d - 1]\!](\tau')$ for each $\tau' \in [0, \tau - d - 1]$;
4. $\Pi_{\text{intervals}} \cup S \models [\![now - w, now - d - 1]\!](\tau')$ for each $\tau' \in [\tau - w, \tau - d - 1]$;
5. $\Pi_{\text{intervals}} \cup S \models [\![0, now]\!](\tau')$ for each $\tau' \in [0, \tau]$.

Let Π' and Π'' be the programs obtained from Π by renaming each predicate P to fresh predicates P' and P'', respectively. Let Π_{aux} be $\Pi' \cup \Pi'' \cup \Pi_{\text{intervals}}$ extended with a rule of the form (65) for each EDB predicate P occurring in Π, a rule of the form (66) for each EDB predicate P occurring in Π, and a rule of the form (67) for each predicate P occurring in Π.

$$[\![0, now]\!](t) \wedge P(t) \rightarrow P'(t) \tag{65}$$

$$[\![now - w, now]\!](t) \wedge P(t) \rightarrow P''(t) \tag{66}$$

$$[\![now - w, now - d - 1]\!](t) \wedge P'(t) \rightarrow P''(t) \tag{67}$$

By construction, Π_{aux} satisfies the following properties for each EDB stream S containing at most one fact about *Now*, each pair of time points τ and τ', and each predicate P:

1. $\Pi \cup S\restriction_{[0,\tau]} \models P(\tau')$ and $Now(\tau) \in S$ if and only if $\Pi_{\text{aux}} \cup S \models P'(\tau')$;
2. $\Pi \cup N\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]} \models P(\tau')$ and $Now(\tau) \in S$ if and only if $\Pi_{\text{aux}} \cup S \models P''(\tau')$, where $N = \Pi(S\restriction_{[0,\tau]})$.

Let Π_{flood} be the program consisting of rules (68)–(72) together with a rule of the form (73) for each IDB predicate P occurring in Π, where $[\![now + 1, \infty]\!]$ and *Flood* are fresh unary predicates.

$$Now(t) \rightarrow [\![now + 1, \infty]\!](t + 1) \tag{68}$$

$$[\![now + 1, \infty]\!](t) \rightarrow [\![now + 1, \infty]\!](t + 1) \tag{69}$$

$$[\![now + 1, \infty]\!](t) \wedge Now(t) \rightarrow Flood(t) \tag{70}$$

$$Flood(t) \rightarrow Flood(t + 1) \tag{71}$$

$$Flood(t) \rightarrow Flood(t - 1) \tag{72}$$

$$Flood(t) \rightarrow P''(t) \tag{73}$$

Clearly, for each stream S that contains two facts about *Now*, $\Pi_{\text{flood}} \cup S$ entails each fact about P'' for P a predicate in Π. Finally, program Π_1 (respectively, Π_2) is defined as the extension of $\Pi_{\text{aux}} \cup \Pi_{\text{flood}}$ with a rule of the form (74) (respectively, of the form (75)) for each predicate P occurring in Π, where $Goal_P$ is a fresh unary predicate.

$$[\![now - d]\!](t) \wedge P'(t) \rightarrow Goal_P(t) \tag{74}$$

$$[\![now - d]\!](t) \wedge P''(t) \rightarrow Goal_P(t) \tag{75}$$

The construction of these programs ensures that the following statements hold for each EDB stream S containing at most one fact about *Now*, each time point τ, and each predicate P:

1. $\Pi \cup S\restriction_{[0,\tau]} \models P(\tau - d)$ and $Now(\tau) \in S$ if and only if $\Pi_1 \cup S \models Goal_P(\tau - d)$;
2. $\Pi \cup N\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]} \models P(\tau - d)$ and $Now(\tau) \in S$ if and only if $\Pi_2 \cup S \models Goal_P(\tau - d)$, where $N = \Pi(S\restriction_{[0,\tau]})$.

This completes the description of our reduction φ. Clearly, φ can be computed in logarithmic space: k is linear in the size of w, and Π_1 and Π_2 have size polynomial in k and the size of Π.

We conclude by arguing correctness of φ. If $w \geq d + \rho$ then, by Theorem 3.8, w is a valid window size for Π and d and, as required, the containment $\emptyset \sqsubseteq \emptyset$ holds trivially. Let us now consider the case where $w < d + \rho$.

Assume $\Pi_1 \sqsubseteq \Pi_2$; we show that w is a valid window size for Π and d. Let S be an EDB stream, let τ be a time point with $\tau \geq d$, let S' be $S \cup \{Now(\tau)\}$, let α be a fact with time argument $\tau - d$, let $N = \Pi(S\restriction_{[0,\tau]})$, and let $N' = \Pi(S'\restriction_{[0,\tau]})$. We assume w.l.o.g. that S contains no fact about *Now*, and hence S' contains one fact about *Now*. Assume that $\Pi \cup S\restriction_{[0,\tau]} \models \alpha$, where α has the form $P(\tau - d)$. It suffices to show $\Pi \cup N\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]} \models \alpha$. By monotonicity, we have $\Pi \cup S'\restriction_{[0,\tau]} \models P(\tau - d)$, and hence $\Pi_1 \cup S' \models Goal_P(\tau - d)$ by the aforementioned property of Π_1. It follows that $\Pi_2 \cup S' \models Goal_P(\tau - d)$ since $\Pi_1 \sqsubseteq \Pi_2$. But then, $\Pi \cup N'\restriction_{[\tau-w,\tau-d-1]} \cup S'\restriction_{[\tau-w,\tau]} \models P(\tau - d)$ by the properties of Π_2, and hence $\Pi \cup N\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]} \models P(\tau - d)$, as required, since no fact about *Now* occurs in Π.

Finally, assume that w is a valid window size for Π and d, and that $\Pi_1 \cup S \models \alpha$ for S an EDB stream and α a fact. It suffices to show $\Pi_2 \cup S \models \alpha$. If α is not a fact about any $Goal_P$, then $\Pi_{\text{aux}} \cup S \models \alpha$ and hence $\Pi_2 \cup S \models \alpha$ since $\Pi_{\text{aux}} \subseteq \Pi_2$. Now assume that α is of the form $Goal_P(\tau)$. If S contains two facts about *Now*, then $\Pi_2 \cup S \models P''(\tau)$ by the properties of Π_{flood}, and hence $\Pi_2 \cup S \models Goal_P(\tau)$ by rule (75), so suppose that S contains at most one fact about *Now*. Then, $\Pi \cup S\restriction_{[0,\tau+d]} \models P(\tau)$ and $Now(\tau + d) \in S$ by the properties of our construction stated above. Furthermore, for $N = \Pi(S\restriction_{[0,\tau+d]})$, it holds that $\Pi \cup N\restriction_{[\tau+d-w,\tau-1]} \cup S\restriction_{[\tau+d-w,\tau+d]} \models P(\tau)$, since w is a valid window size for Π and d. As a result, $\Pi_2 \cup S \models Goal_P(\tau)$ by the properties of Π_2, as required. □

We conclude by giving upper bounds for the problem of computing a minimum valid delay and window size. The proof of the theorem combines the previous results in a straightforward manner.

Theorem 7.4. *There exists an algorithm running in exponential space that, for a fixed object domain for streams, takes as input a program Π and computes the smallest valid delay d for Π and the smallest valid window size for Π and d whenever Π admits a valid delay, or rejects Π if it does not admit a valid delay. Furthermore, the algorithm runs in polynomial space if Π is object-free with offsets coded in unary.*

Proof. On input Π, the algorithm performs the following steps:

1. It checks whether Π admits a valid delay, and it rejects the input if it does not.
2. If Π is object-free with offsets in unary, then it normalises program Π into a program Ω following Theorem 5.5, and it computes the set $\mathbf{J}$ of the IDB predicates of Π. Otherwise, it normalises and grounds program Π into a program Ω following Theorem 5.4, and it computes the grounding $\mathbf{J}$ of the IDB predicates of Π.
3. It computes the number k of states in the automaton $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ where Ω and $\mathbf{J}$ are as follows. If Π is object-free with offsets in unary, then Ω is the normal program for Π according to Theorem 5.5, and $\mathbf{J}$ is the set of IDB predicates of Π. Otherwise, Ω is the grounding of the normal program for Π according to Theorem 5.4, and $\mathbf{J} = \zeta(\mathbf{I})$ for $\mathbf{I}$ the set of IDB predicates of Π.
4. It computes the smallest valid delay d by iteratively incrementing d from 0 to k, and stopping as soon as the resulting value is a valid delay for Π.
5. It computes the smallest valid window size w by iteratively incrementing w from 0 to $d + \rho$, with ρ the maximum forward radius of a rule in Π, and stopping as soon as the resulting value is a valid window size for Π and d.
6. It outputs d and w.

Correctness of the algorithm follows directly from the following observations. Recall that if Π admits a valid delay then accepting runs of $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ cannot involve a cycle over a non-ε transition, as argued in the proof of Theorem 7.1. As a result, if Π admits a valid delay d, then d cannot be larger than the number k of states of the automaton $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$, which justifies Step 2. Furthermore, by Theorem 3.8, if d is a valid delay then $d + \rho$ is a valid window size, which justifies Step 3.

We now argue that the algorithm runs in exponential space when Π is not object-free or offsets are coded in binary; it is then immediate to see that all bounds are reduced by one exponential in the complementary case of Π object-free and with offsets in unary, where Theorem 5.5 applies instead of Theorem 5.4. The first step is clearly feasible in exponential space by Theorem 7.1, and the second step by Theorem 5.4. For Step 3, note that the number of states in $\mathcal{A}[\mathsf{delay}(\Omega, \mathbf{J})]$ is exponentially bounded in the maximum size of a state, which is exponential. As a result, we can do no more than doubly-exponentially many delay validity calls in Step 3, each of which feasible in exponential space in Π and the tested value d (which can be stored in binary using only exponential space in the size of Π). Finally, Step 4 involves no more than $d + \rho$ window size validity tests, each of which also takes exponential space (again, note that each tested w requires only exponential space in Π). □

8. Complexity lower bounds

We next show that the upper bounds established in the previous section are tight. For this, we first establish lower bounds for containment of forward-propagating programs over a fixed object domain. In particular, we show that the problem is ExpSpace-hard under both unary and binary coding of time offsets; furthermore, if we consider forward-propagating object-free programs, then the problem is ExpSpace-hard under binary coding and PSpace-hard under unary coding. We

then combine these results with the reductions from program containment to delay existence, delay validity, and window size validity developed in Section 4, concluding that all three problems are ExpSpace-hard over a fixed object domain, PSpace-hard for object-free programs under unary coding, and ExpSpace-hard for object-free programs under binary coding.

The ExpSpace lower bound for containment over a fixed object domain follows from the fact that temporal Datalog programs capture *succinct regular expressions* (SREs)—see, e.g., [33]. Specifically, for a given SRE, we construct a program that encodes a one-pass scan over an input word, deriving a special fact if the word is in the language of the corresponding SRE. The program is built following the inductive structure of SREs. The *concatenation* and *alternation* operators are easily captured due to their correspondence to logical *and* and *or*, respectively. The *Kleene plus* operator is captured via recursion. Finally, the *exponentiation* operator of SREs is captured via recursion and binary counting to keep track of the number of recursive steps—where binary counting can be expressed by rules. Binary counting makes use of objects, but it is not required if there is no exponentiation. Thus, object-free programs capture regular expressions, and this implies the PSpace lower bound for object-free programs.

Theorem 8.1. *Containment of forward-propagating programs with respect to a fixed object domain is* ExpSpace*-hard. Furthermore, containment of object-free forward-propagating programs is* PSpace*-hard. In both cases, the mentioned hardness holds already when the only time offsets allowed in rules are* 0 *and* 1.

Proof. We show the first claim by providing a reduction from containment of succinct regular expressions (SREs) to containment of forward-propagating programs. The language of SREs extends regular expressions with the exponentiation operator, which allows one to write expressions of the form R^k where R is an SRE and k is a non-negative integer; the expression R^k matches any concatenation of k words matched by R. The containment problem for SREs, which assumes exponents to be coded in binary, is known to be ExpSpace-complete [33]. Formally, we consider SREs over a given alphabet Σ generated by the grammar

$$R ::= \emptyset \mid \varepsilon \mid \sigma \mid R \cup R \mid R \circ R \mid R^+ \mid R^k$$

where σ ranges over Σ and $k \geq 1$. Our reduction maps a pair $\langle R_1, R_2 \rangle$ of SREs to a pair of programs $\langle \Pi_1, \Pi_2 \rangle$ such that $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$ if and only if $\Pi_1 \sqsubseteq \Pi_2$, for $\mathcal{L}(R_i)$ the language of R_i. Programs Π_1 and Π_2 consist of several components, which we describe next.

- For each $m \geq 1$, program Π^m_{succ} implements a binary counter with m bits. It consists of rules (76)–(79) and a rule of the form (80) for each $i \in [0, m-1]$ where F and A are unary predicates, B is a binary predicate, $succ^m$ is a $(2m+1)$-ary predicate, $\bar{0}$ and $\bar{1}$ are fresh objects, intuitively standing for zero and one, respectively, $|\mathbf{x}| = i$, and $|\bar{\mathbf{1}}| = |\bar{\mathbf{0}}| = m - i - 1$:

$$F(t) \rightarrow A(t) \tag{76}$$
$$A(t) \rightarrow A(t+1) \tag{77}$$
$$A(t) \rightarrow B(\bar{0}, t) \tag{78}$$
$$A(t) \rightarrow B(\bar{1}, t) \tag{79}$$
$$\textstyle\bigwedge_{j=1}^{i} B(x_j, t) \rightarrow succ^m(\mathbf{x}, \bar{0}, \bar{\mathbf{1}}, \mathbf{x}, \bar{1}, \bar{\mathbf{0}}, t) \tag{80}$$

  Formally, for each $m \geq 1$, each time point τ, and each $\tau' \geq \tau$, it holds that $\Pi^m_{\text{succ}} \cup \{F(\tau)\} \models succ^m(\mathbf{i}, \mathbf{j}, \tau')$ if and only if $\mathbf{i}$ and $\mathbf{j}$ are m-tuples over $\{\bar{0}, \bar{1}\}$, and $i + 1 = j$ for i and j the numbers encoded by $\mathbf{i}$ and $\mathbf{j}$, respectively.
- For a given SRE R over alphabet Σ, program Π_R is constructed inductively as follows, where G is a fresh unary predicate, A_σ is a fresh unary predicate for each $\sigma \in \Sigma$, and $\phi(\Pi)$ and $\psi(\Pi)$ are the programs obtained from an arbitrary program Π by renaming each predicate P to fresh predicates P^ϕ and P^ψ, respectively, unless P is of the form A_σ or $succ^m$.
  - If $R = \emptyset$, then $\Pi_R = \emptyset$.
  - If $R = \varepsilon$, then Π_R consists of the rule

$$F(t) \rightarrow G(t). \tag{81}$$

  - If $R = \sigma$ for $\sigma \in \Sigma$, then Π_R consists of the rule

$$F(t) \wedge A_\sigma(t) \rightarrow G(t+1). \tag{82}$$

  - If $R = S \cup T$, then Π_R extends $\phi(\Pi_S) \cup \psi(\Pi_T)$ with the rules

$$F(t) \rightarrow F^\phi(t) \tag{83}$$
$$F(t) \rightarrow F^\psi(t) \tag{84}$$

$$G^{\phi}(t) \to G(t) \tag{85}$$
$$G^{\psi}(t) \to G(t). \tag{86}$$

– If $R = S \circ T$, then Π_R extends $\phi(\Pi_S) \cup \psi(\Pi_T)$ with the rules

$$F(t) \to F^{\phi}(t) \tag{87}$$
$$G^{\phi}(t) \to F^{\psi}(t) \tag{88}$$
$$G^{\psi}(t) \to G(t). \tag{89}$$

– If $R = S^+$, then Π_R extends $\phi(\Pi_S)$ with the rules

$$F(t) \to F^{\phi}(t) \tag{90}$$
$$G^{\phi}(t) \to F^{\phi}(t) \tag{91}$$
$$G^{\phi}(t) \to G(t). \tag{92}$$

– If $R = S^k$ with $k \geq 1$, then Π_R is constructed from Π_S as follows. First, we replace each atom $P(\mathbf{p}, s)$ with $P'(\mathbf{p}, \mathbf{x}, s)$, with P' fresh and $|\mathbf{x}| = m$ for m the number of bits required to encode k, unless P is of the form A_σ or $succ^m$. Then, we extend the resulting program with the following rules where $\mathbf{a}$ is the encoding of $k-1$ as a binary string over $\bar{0}$ and $\bar{1}$ using m bits:

$$F(t) \to F'(\bar{\mathbf{0}}, t) \tag{93}$$
$$G'(\mathbf{a}, t) \to G(t) \tag{94}$$
$$G'(\mathbf{x}, t) \wedge succ^m(\mathbf{x}, \mathbf{y}, t) \to F'(\mathbf{y}, t) \tag{95}$$

Then, Π_1 and Π_2 are defined as follows, where Π_{succ} is the union of all Π^m_{succ} such that $succ^m$ occurs in $\Pi_{R_1} \cup \Pi_{R_2}$, program Π'_{R_2} is obtained from Π_{R_2} by renaming each IDB predicate P to a fresh predicate P' (in particular, G is renamed to G'), and G^* is a fresh unary predicate:

- $\Pi_1 = \Pi_{R_1} \cup \Pi'_{R_2} \cup \Pi_{\text{succ}} \cup \{G(t) \to G^*(t)\}$
- $\Pi_2 = \Pi_{R_1} \cup \Pi'_{R_2} \cup \Pi_{\text{succ}} \cup \{G'(t) \to G^*(t)\}$

We next argue correctness of the reduction. For this, we first show that our construction captures the language of the relevant SREs in the sense of the following two claims.

Claim 1. *Let $s = \sigma_1 \dots \sigma_n$ be a word in $\mathcal{L}(R)$ and let D be a finite set of EDB facts. If Π_{succ} includes each Π^m_{succ} such that $succ^m$ occurs in Π_R, $F(\tau) \in D$, and $A_{\sigma_i}(\tau + i - 1) \in D$ for each $i \in [1, n]$, then $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + n)$.*

We show the claim by induction on R. Consider the base cases. Clearly, $R \neq \emptyset$ as $s \in \mathcal{L}(R)$. If $R = \sigma$, we have $s = \sigma$; but then, $A_\sigma(\tau) \in D$ implies $\Pi_R \cup D \models G(\tau + 1)$ by rule (82). If $R = \varepsilon$, then $s = \varepsilon$ (and hence $n = 0$); but then, $\Pi_R \cup D \models G(\tau)$ by rule (81). Next, we consider the inductive cases.

- $R = S \cup T$ and $s \in \mathcal{L}(S) \cup \mathcal{L}(T)$. If $s \in \mathcal{L}(S)$, then $\Pi_S \cup \Pi_{\text{succ}} \cup D \models G(\tau + n)$ by the inductive hypothesis; in this case, $\phi(\Pi_S) \cup \Pi_{\text{succ}} \cup D \cup \{F^{\phi}(\tau)\} \models G^{\phi}(\tau + n)$ by the definition of $\phi(\Pi_S)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + n)$ by rules (83) and (85). The case $s \in \mathcal{L}(T)$ proceeds analogously using rules (84) and (86).
- $R = S \circ T$ and $s = s_1 s_2$ with $s_1 = \sigma^1_1 \dots \sigma^1_{n_1} \in \mathcal{L}(S)$ and $s_2 = \sigma^2_1 \dots \sigma^2_{n_2} \in \mathcal{L}(T)$. By the inductive hypothesis, $\Pi_S \cup \Pi_{\text{succ}} \cup D \models G(\tau + n_1)$, hence $\phi(\Pi_S) \cup \Pi_{\text{succ}} \cup D \cup \{F^{\phi}(\tau)\} \models G^{\phi}(\tau + n_1)$ by the construction of $\phi(\Pi_S)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F^{\psi}(\tau + n_1)$ by rules (87) and (88). Furthermore, again by the inductive hypothesis, we have $\Pi_T \cup \Pi_{\text{succ}} \cup D \cup \{F(\tau + n_1)\} \models G(\tau + n_1 + n_2)$, hence $\psi(\Pi_T) \cup \Pi_{\text{succ}} \cup D \cup \{F^{\psi}(\tau + n_1)\} \models G^{\psi}(\tau + n_1 + n_2)$ by the construction of $\psi(\Pi_T)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + n_1 + n_2)$ by rule (89).
- $R = S^+$ and $s = s_1 s_2 \dots s_k$ with $s_i = \sigma^i_1 \dots \sigma^i_{n_i} \in \mathcal{L}(S)$ for each $i \in [1, k]$. By a straightforward induction on i, which uses the outer inductive hypothesis applied to S together with rules (90) and (91) applied in the base and inductive case respectively, we can show $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G^{\phi}(\tau + \sum_{j=1}^{i} n_j)$ for each $i \in [1, k]$. This implies $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G^{\phi}(\tau + \sum_{i=1}^{k} n_i)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + \sum_{i=1}^{k} n_i)$ by rule (92).
- $R = S^k$ and $s = s_1 s_2 \dots s_k$ with $s_i = \sigma^i_1 \dots \sigma^i_{n_i} \in \mathcal{L}(S)$ for each $i \in [1, k]$. We show by induction on $i \geq 1$ that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau + \sum_{j=1}^{i} n_j)$ for $\mathbf{b}$ the binary encoding of $i - 1$; this implies $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau + \sum_{i=1}^{k} n_i)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + \sum_{i=1}^{k} n_i)$ by rule (94). In the base case ($i = 1$), we have $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\bar{\mathbf{0}}, \tau)$ by rule (93), and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\bar{\mathbf{0}}, \tau + n_1)$, because $\Pi_S \cup \Pi_{\text{succ}} \cup D \models G(\tau + n_1)$ by the outer inductive hypothesis for

S, and by the construction of Π_R. For $i > 1$, the inner inductive hypothesis yields $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{c}, \tau + \sum_{j=1}^{i-1} n_j)$ where $\mathbf{c}$ is the binary encoding of $i-2$. Hence, $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau + \sum_{j=1}^{i-1} n_j)$ where $\mathbf{b}$ encodes $i-1$, by rule (95) and by the construction of Π_{succ}. But then, $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau + \sum_{j=1}^{i} n_j)$ follows by the construction of Π_R from $\Pi_S \cup D \cup \{F(\tau + \sum_{j=1}^{i-1} n_j)\} \models G(\tau + \sum_{j=1}^{i} n_j)$, which holds by the outer inductive hypothesis for S.

Claim 2. *Let D be a finite set of EDB facts and let τ be a time point. If $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau)$, then there exists a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(R)$ such that $F(\tau - n) \in D$ and $A_{\sigma_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$.*

We show the claim by induction on R. For the base cases, note first that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau)$ implies that $R \neq \emptyset$. If $R = \sigma$, we take $s = \sigma$ since $\{F(\tau - 1), A_\sigma(\tau - 1)\} \subseteq D$ by the construction of Π_R. Finally, if $R = \varepsilon$ we take $s = \varepsilon$ since $F(\tau) \in D$ by the construction of Π_R. We now consider the inductive cases.

- If $R = S \cup T$, we have $\Pi_R \cup D \models G^\phi(\tau)$ or $\Pi_R \cup D \models G^\psi(\tau)$. By the construction of Π_R and the inductive hypothesis, it follows that there is a word $s = a_1 \dots a_n \in \mathcal{L}(S) \cup \mathcal{L}(T)$ such that $A_{a_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$ and either $\Pi_R \cup D \models F^\phi(\tau - n)$ or $\Pi_R \cup D \models F^\psi(\tau - n)$, which both imply $F(\tau - n) \in D$ as required.
- If $R = S \circ T$, we have $\Pi_R \cup D \models G^\psi(\tau)$. By the construction of Π_R and the inductive hypothesis, there is a word $s_1 = a_1 \dots a_n \in \mathcal{L}(T)$ such that $A_{a_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$ and $\Pi_R \cup D \models F^\psi(\tau - n)$. Then, $\Pi_R \cup D \models F^\psi(\tau - n)$ implies $\Pi_R \cup D \models G^\phi(\tau - n)$, and hence, again by the construction of Π_R and the inductive hypothesis, we have that there is a word $s_2 = b_1 \dots b_{n'} \in \mathcal{L}(S)$ such that $A_{b_i}(\tau - n - n' + i - 1) \in D$ for each $i \in [1, n']$ and $\Pi_R \cup D \models F^\phi(\tau - n - n')$, which implies $F(\tau - n - n') \in D$. Finally, note that $s = s_1 s_2 \in \mathcal{L}(R)$, and hence s and D are as required.
- $R = S^+$. By the construction of Π_R and the inductive hypothesis, $\Pi_R \cup D \models G^\phi(\tau')$ with $\tau' \leq \tau$ implies that there is a word $\sigma_1 \dots \sigma_n \in \mathcal{L}(S)$ such that $\Pi_R \cup D \models F^\phi(\tau' - n)$ and $A_{\sigma_i}(\tau' - n + i - 1) \in D$ for each $i \in [1, n]$. By generalising the argument for $R = S \circ T$, we can then deduce the existence of words $s_1, \dots, s_k$ (for $k \geq 1$) such that, for each $i \in [1, k]$:
  - $s_i = \sigma_1^i \dots \sigma_{n_i}^i \in \mathcal{L}(S)$,
  - $\Pi_R \cup D \models F^\phi(\tau - \sum_{j=i}^{k} n_j)$,
  - $A_{\sigma_j^i}(\tau + j - 1 - \sum_{\ell=i}^{k} n_\ell) \in D$ for each $j \in [1, n_i]$,
  - $F(\tau - \sum_{i=1}^{k} n_i) \in D$.

  Then, $s = s_1 \cdots s_k$ and D are as required.
- $R = S^k$ for $k \geq 1$. By the construction of Π_R and the inductive hypothesis, the assertion $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau')$ implies the existence of a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(S)$ such that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau' - n)$ and $A_{\sigma_i}(\tau' - n + i - 1) \in D$ for each $i \in [1, n]$; furthermore, by the construction of Π_{succ}, if $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau')$ with $\mathbf{b}$ encoding $k \geq 0$ then $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{c}, \tau')$ with $\mathbf{c}$ encoding $k - 1$ due to rule (95). Considering that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{a}, \tau)$ with $\mathbf{a}$ the binary encoding of $k - 1$, due to rule (94), the two properties above imply (as it can be shown by a straightforward induction on i from k to 1) the existence of words $s_1, \dots, s_k$ such that, for each $i \in [1, k]$:
  - $s_i = \sigma_1^i \dots \sigma_{n_i}^i \in \mathcal{L}(S)$;
  - $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau - \sum_{j=i}^{k} n_j)$ where $\mathbf{b}$ is the binary encoding of $i - 1$; and
  - $A_{\sigma_j^i}(\tau + j - 1 - \sum_{\ell=i}^{k} n_\ell) \in D$ for each $j \in [1, n_i]$.

  Then, $s = s_1 \cdots s_k$ and D are as required. This concludes the proof of the claim.

We now show correctness. Assume first that $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$. Pick an arbitrary finite set D of EDB facts and a fact α such that $\Pi_1 \cup D \models \alpha$. We show $\Pi_2 \cup D \models \alpha$, which is sufficient to establish $\Pi_1 \sqsubseteq \Pi_2$. It is clear that $\Pi_2 \cup D \models \alpha$ if α is not a fact about G^*, so assume that α has the form $G^*(\tau)$. It follows that $\Pi_1 \cup D \models G(\tau)$, and hence $\Pi_{R_1} \cup \Pi_{\text{succ}} \cup D \models G(\tau)$. By Claim 2, there is a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(R_1)$ such that $F(\tau - n) \in D$ and $A_{\sigma_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$. Then, by assumption, $s \in \mathcal{L}(R_2)$, and hence $\Pi_{R_2} \cup \Pi_{\text{succ}} \cup D \models G(\tau)$ by Claim 1. Consequently $\Pi'_{R_2} \cup \Pi_{\text{succ}} \cup D \models G'(\tau)$, and thus $\Pi_2 \cup D \models G^*(\tau)$.

Assume now that $\Pi_1 \sqsubseteq \Pi_2$. We show $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$. Let $s = a_1 \dots a_n$ be a word in $\mathcal{L}(R_1)$ and let D be the set consisting of the fact $F(0)$ and a fact $A_{a_i}(i-1)$ for each $i \in [1, n]$. It follows that $\Pi_{R_1} \cup \Pi_{\text{succ}} \cup D \models G(n)$ by Claim 1, and hence $\Pi_1 \cup D \models G^*(n)$. By assumption, this implies $\Pi_2 \cup D \models G^*(n)$, and hence, by construction, $\Pi_{R_2} \cup \Pi_{\text{succ}} \cup D \models G(n)$. Then, by Claim 2, there is some $s' = b_1 \dots b_{n'} \in \mathcal{L}(R_2)$ such that $F(n - n') \in D$ and $A_{b_i}(n - n' + i - 1) \in D$ for each $1 \leq i \leq n'$. Furthermore, $n' = n$ since $F(0)$ is the only fact about F in D by the definition of D, and, for each $i \in [1, n]$, we have $b_i = a_i$ since $A_{a_i}(i-1)$ is the only fact in D of the form $A_\sigma(i-1)$. Therefore, $s' = s$, and hence $s \in \mathcal{L}(R_2)$, as required. This completes our ExpSpace-hardness proof.

We conclude by arguing that containment of object-free forward-propagating programs is PSpace-hard. In this case, we reduce from containment of standard regular expressions (without exponentiation), which is a well-known PSpace-hard problem. The reduction is a special case of our reduction from SRE containment: it suffices to observe that we no longer need the program component Π_{succ}, without which the program becomes object-free. The correctness arguments transfer verbatim. □

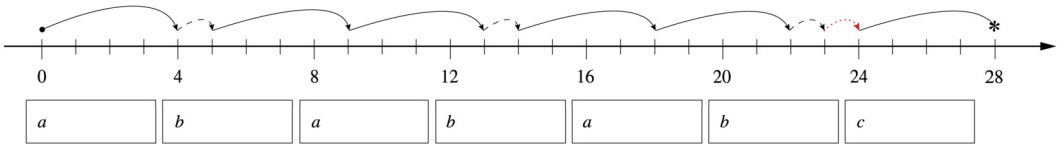


Fig. 5. Pointer's moves for the expression $(ab)^3c$ on input word *abababc*. Each tape cell spans 4 time points, since the exponent requires 2 bits. Solid arrows denote pointer's moves to the next cell with no change in its relative position within a cell (and hence no change in counter values). Dashed arrows denote within-cell moves, to increase the counter by one when subexpression *ab* has just been matched. The dotted red arrow denotes a pointer's move to the next cell with a counter reset, hence a move to the beginning of the cell. The right-most arrow, ending with an asterisk, denotes that the given SRE has been matched.

Our next goal is to show that, also for *forward-propagating object-free* programs, the containment problem is ExpSpace-hard if time offsets are coded in *binary*. Again, we would like to find a reduction from containment of SREs. To do so, we have to encode exponentiation taking advantage of binary time offsets, instead of objects as done above. The idea is still to encode a pointer that scans an input word and derives a special fact if the word is in the language of the corresponding SRE. But now, time should encode both the position of the pointer over the word and the value of the several counters that keep track of how many times an exponentiated expression has been matched. The first issue is that, if input words are encoded as in the previous reduction, moving the pointer to change the value of one of the counters will also make it point to a different letter. To deal with this, we consider input letters to appear every $2^{c\cdot k}$ time points, with c the number of occurrences of exponents in the SRE and k the maximum number of bits required to encode any of the exponents; then we propagate a letter appearing at time τ to the time points $[\tau, \tau + 2^{c\cdot k} - 1]$, i.e., to all time points preceding the next letter. These propagated letters are marked so that they can be distinguished from any spurious letter that may appear in the gap between two valid letters. This way, with respect to reading a letter, it matters in which interval the pointer is, and not exactly at which time point within the interval. We see each of these intervals as a tape *cell* containing one input letter. Each cell, however, consists of $2^{c\cdot k} - 1$ time points. We use the *relative position* of the pointer within the cell to encode counter values. The relative position uses up to $c \cdot k$ bits, which can be used to encode counter values. In particular, the i-th counter is represented by the bits at positions $[i \cdot k, (i + 1) \cdot k - 1]$. All of the above makes use of a subprogram that—via standard techniques for binary counting in logic—starting from a time point where a *start* fact holds, it *labels* every following time point with the distance from the start fact modulo $2^{c\cdot k}$. This way we can always check the pointer's relative position within a cell, in order to check, e.g., counter values, and whether the pointer is at the beginning of a cell and hence it has to read an input letter or it is in the middle of a cell and hence an already propagated letter has to be further propagated from the previous time point. Then, to move the pointer to the next cell without changing the value of any counter, its position is increased by $2^{c\cdot k}$. To increase the value of the i-th counter by one, the pointer is increased by $2^{i\cdot k}$. If the value of the i-th counter was less than $2^k - 1$ before the increase, then it does not overflow, and hence the other counters are not affected nor the pointer gets moved to a different cell. To check the value of the i-th counter, we check the value of bits $[i \cdot k, (i + 1) \cdot k - 1]$ of the label at the current pointer's position. Furthermore, we need a reset mechanism for counters, since we may need to reuse a counter. Since the maximum value that is relevant for a counter is the value of the exponent the counter was allocated for, a simple thing to do would be to decrease the value of a counter by the exponent value when such a value is reached. However, this means moving the pointer backwards, and it would introduce backward-propagating rules, whereas our goal is to find a forward-propagating program; thus, we need a slightly more sophisticated solution. We include an automatic reset in the mechanism that moves the pointer from one cell to the next one. Instead of increasing the pointer by $2^{c\cdot k}$ as discussed above, we increase the pointer by $2^{c\cdot k} - e$, where e is the sum of all exponents whose counters have reached the exponent value. This solution simultaneously moves the pointer to the next cell and resets counters. As a final note, the mentioned pointer is not represented explicitly in the program, but it is the result of the interplay of two predicates, F and G, under their various renamings. The pointer's moves for a simple case are depicted in Fig. 5.

To clarify how a single number a can be used to encode all counter values $a_0, \ldots, a_{c-1}$ while being able to operate on a certain a_i without affecting any other a_j, we state the following properties. An integer $a \in [0, 2^{c\cdot k} - 1]$ can be expressed as $a = \sum_{i=0}^{c-1} a_i \cdot 2^{i\cdot k}$ for some integers $a_0, \ldots, a_{c-1} \in [0, 2^k - 1]$, and hence it admits a $(c \cdot k)$-bits binary encoding obtained as the concatenation of the k-bit binary encodings of $a_0, \ldots, a_{c-1}$. Furthermore, for each $i \in [0, c - 1]$ and each $b \in [0, 2^k - 1 - a_i]$, the binary encoding of $a + b \cdot 2^{i\cdot k}$ differs from the one of a only in the bits at positions $[i \cdot k, (i + 1) \cdot k - 1]$, which now encode $a_i + b$.

Theorem 8.2. *Containment of object-free forward-propagating programs is* ExpSpace*-hard if time offsets are coded in binary.*

Proof. The proof is given in Appendix A. □

Theorem 8.3. *Delay existence, delay validity, and window size validity with respect to a fixed object domain are* ExpSpace*-hard regardless of the coding of time offsets. Furthermore, when restricted to object-free programs, the problems are* ExpSpace*-hard if time*

offsets are coded in binary, and PSPACE*-hard if time offsets are coded in unary. The bounds for delay validity hold already for backward-bounded programs, whereas the bounds for window size validity hold already for forward-propagating programs.*

Proof. The proof of Theorem 4.1 defines a reduction f_1 from program containment to delay existence. The proof of Theorem 4.2 defines a reduction f_2 from program containment to delay validity for backward-bounded programs. Finally, the proof of Theorem 4.3 establishes a reduction f_3 from program containment to window size validity for forward-propagating programs. The properties of these reductions do not rely on whether the object domain is considered fixed, and they do not rely on the number coding either. Furthermore, when given object-free programs as input, the reductions produce object-free programs.

The statement of the theorem then trivially follows from Theorems 8.1 and 8.2, since the former establishes EXPSPACE-hardness of containment for forward-propagating programs with respect to a fixed object domain, as well as PSPACE-hardness of containment for object-free forward-propagating programs, and the latter establishes EXPSPACE-hardness of containment for object-free forward-propagating programs assuming binary coding of time offsets. □

9. Related work

In this section, we review related work in stream query processing, stream reasoning, and temporal deductive databases.

9.1. Languages for stream query processing in databases and semantic web

The formal underpinnings of stream query processing in databases were established in [7,8]. Arasu et al. [8] proposed the Continuous Query Language (CQL) as an extension of SQL with window constructs, which specify the input data relevant for query processing at any point in time. CQL has since become the core of many other stream query languages, including languages for the Semantic Web, such as Streaming-SPARQL [34], C-SPARQL [35], CQELS [12], RSP-QL [36], EP-SPARQL [11], SPARQL_{stream} [10], and STARQL [14].

Database views expressed in CQL always admit zero as a valid delay, and hence delay existence and delay validity checking are trivial problems for CQL. At the same time, CQL is expressive enough to capture forward-propagating non-recursive programs; thus, it follows from our previous work [1] that window size validity checking for a set of CQL views is coNEXP-hard.

9.2. Temporal extensions of datalog

There have been many proposals for extending Datalog with temporal features. In this line of work, the focus is typically not on stream reasoning, but rather on standard database research problems such as determining data complexity of fact entailment [19,37,38] and establishing the expressive power of the language [22,38]. In addition, many of these works also consider problems specific to temporal deductive databases such as computing finite specifications of infinite query answers [19,39].

Under the assumption that numbers are coded in unary, the language considered in this paper is a notational variant of Chomicki and Imieliński's Datalog_{1S} [19] with the additional temporal guardedness condition. Chomicki and Imieliński assume in their technical results that Datalog_{1S} rules mention at most one time variable; hence, our guardedness condition only imposes the additional requirement that rules do not mention explicit time points. Datalog_{1S} was proposed as a language for temporal deductive databases, which have been surveyed in [22]. The language proposed by Toman and Chomicki [40] extends Datalog with integer periodicity constraints, which can be used to encode and store information about periodic activities in temporal databases.

Templog is an extension of Datalog with temporal modal operators [21]. As shown by Baudinet et al. [22], Templog and Datalog_{1S} are inter-reducible, and hence both languages are equivalent in terms of expressive power. Baudinet [41] studied the expressive power of Templog and showed that it expresses exactly the finitely regular ω-languages—that is, those languages recognised by finite-acceptance automata on infinite words, where a finite-acceptance automaton is a classical finite automaton that is applied to prefixes of infinite words. This coincides with the expressive power of the μTL^+ fragment of Vardi's [42] fixpoint calculus μTL without negation or greatest fixpoints. Furthermore, Templog with stratified negation expresses exactly the ω-regular languages, and hence its expressiveness coincides with the one of full μTL.

Under unary coding of numbers, our language can also be seen as a fragment of *bidirectional ASP programs*, which extend Datalog with disjunction, non-monotonic negation under the stable model semantics, and function symbols [43,44] while ensuring decidability of standard reasoning tasks. A bidirectional program restricts the syntax of ASP programs with function symbols to ensure tree-shaped stable models. In particular, each rule r must contain a variable x such that the first position of each atom in r is either a constant, variable x, or a unary function term $f(x)$, and every other position is occupied by either a constant or a variable different from x; our Temporal Datalog programs are clearly bidirectional since we can choose variable x in each rule to be the unique temporal variable t. It was shown in [43] using automata-based techniques that reasoning over disjunctive (resp. non-disjunctive) bidirectional programs is 2-EXPTIME-complete (resp. EXP-complete) under bounded predicate arity; furthermore, if we additionally allow for a single unary function symbol (which suffices to model time), reasoning over disjunctive (resp. non-disjunctive) programs becomes EXPSPACE-complete (resp. PSPACE-complete).

The PSPACE upper bound for inconsistency of core bidirectional programs with one function symbol can be used to establish our PSPACE upper bound for containment of object-free Temporal Datalog programs under unary coding of numbers, and thus also our upper bounds to window size validity and delay validity (which are both reducible to containment) also under unary coding.[4] To see how containment reduces to inconsistency checking over bidirectional programs, consider arbitrary object-free programs Π_1 and Π_2 in normal form (cf. Definition 5.3), and let s be a unary function symbol. We transform each rule of the form $P(t+1) \rightarrow P'(t)$ to $P(s(t)) \rightarrow P'(t)$, each rule of the form $P(t-1) \rightarrow P'(t)$ to $P(t) \rightarrow P'(s(t))$, and we leave all other rules unchanged. The programs we obtain are core bidirectional programs with a single function symbol. Furthermore, let us rename each intensional predicate P in Π_1 into P^1, and each intensional predicate P in Π_2 into P^2. Consider then program Π consisting of all the rules obtained as above, a rule of the form (96) for each intensional predicate P from Π_1, and rules (97) and (98), where G is a fresh predicate and 0 a fresh constant.

$$G(t) \leftarrow P^1(t), \mathit{not}\ P^2(t), \tag{96}$$

$$G(t) \leftarrow G(s(t)), \tag{97}$$

$$G(0) \leftarrow \mathit{not}\ G(0). \tag{98}$$

Each stable model of Π must contain $G(0)$ because of rule (98), and the latter fact is derived exactly when one can derive a fact $P^1(\tau)$ without deriving $P^2(\tau)$. Thus, every stable model of Π encodes a counter-example for the containment $\Pi_1 \sqsubseteq \Pi_2$.

In contrast to containment, delay validity and window size validity, it does not seem possible to easily reduce delay existence to reasoning over bidirectional programs. Furthermore, the automata-based techniques in [43] do not seem suitable for checking delay existence either. In particular, their automaton operates on a language of infinite words (with each word encoding an interpretation and an instance of the program), while in our approach to checking delay existence it seems essential to operate on languages of finite words with each word encoding a finite set of extensional facts and an answer fact, and to encode any additional information in the states of the automata.

DatalogMTL [20,38,45] is an extension of Datalog where atoms in rules can mention operators from Metric Temporal Logic (MTL) interpreted over the rational timeline. DatalogMTL has been studied in the contexts of querying temporal data [20,38] and stream reasoning [25]. DatalogMTL has also been recently studied over the *integer timeline* [46]; in this setting, DatalogMTL remains a powerful KR language, which subsumes our Temporal Datalog language with binary coding of numbers, as well as certain Horn fragments of Linear Temporal Logic [47]. It follows from the complexity results in [46] that reasoning in Temporal Datalog under binary coding of numbers remains in PSPACE for data complexity, and hence the representation of numbers does not influence the complexity of standard reasoning. This result, however, does not immediately provide insights into whether delay existence and window size validity remain in PSPACE if numbers in the input program are coded in binary.

9.3. Rule-based stream reasoning

Barbieri et al. [9] consider stream reasoning over non-temporal Datalog programs that are temporally interpreted as if each atom holds at the current time; this can be easily encoded in our framework by replacing each (non-temporal) atom $A(\mathbf{s})$ with the atom $A(\mathbf{s}, t)$ for t a fixed time variable. In this setting, both the delay and the window size problems are trivial since rules refer only to the present time point and hence zero is always a valid delay and window size. In contrast to our work, where data facts involve single time points, Barbieri et al. consider facts annotated with a validity interval $[\tau_1, \tau_2]$, where τ_1 and τ_2 represent insertion and expiration time, respectively.

Wałęga et al. [25] propose a stream reasoning algorithm for a forward-propagating fragment of DatalogMTL, which admits valid delay zero. Wałęga et al. do not consider the window size problem in their work, and their stream reasoning algorithm relies on a window size derived through a syntactic condition similar to ours based on the maximum rule radius. Wałęga et al. address in their algorithm a number of additional difficulties stemming from the fact that DatalogMTL is a richer language than Temporal Datalog, and that it is interpreted over the non-negative rational numbers rather than the natural numbers.

Zaniolo [13] proposes Streamlog: a language for representing standing queries that underlies the stream reasoning system ASTRO [48]. Streamlog extends Temporal Datalog with non-monotonic negation while at the same time restricting the syntax so that only facts over time points mentioned in the data can be derived. Each atom in a Streamlog rule has a single time variable, and time variables can occur in inequality conditions involving also other variables and arithmetic operations. The paper focuses on *sequential programs*, which are locally stratified by restricting the use of inequality conditions so that the time argument of a positive body literal is at most the time argument of the head, and the time argument of a negative body literal is strictly smaller than the time argument of the head. As a result, sequential programs admit a unique stable model, which can be computed by increasing values of time. Such programs thus have delay zero; furthermore, if parametrised with a valid window size, they can be evaluated by a variant of our stream reasoning algorithm that considers each past fact that has not been derived as false—in line with Zaniolo's *progressive closing world assumption* (PCWA).

[4] We thank an anonymous referee for pointing out the potential relationship between our work and reasoning over bidirectional ASP programs.

LARS [15] is a temporal rule-based stream reasoning language featuring built-in window constructs. LARS formulas extend propositional logic with temporal and window operators, and LARS programs are defined as ASP programs allowing for LARS formulas in place of atoms. A stream is seen as function from a finite interval of the natural numbers to sets of propositions, thus streams in LARS are intrinsically bounded. The issues stemming from reasoning over unbounded streams in LARS have been addressed only in [49,50]. Beck et al. [49] describe a LARS-based system where reasoning over streams is reduced to repeated reasoning over datasets, with the latter task solved by a logic-programming reasoner. Laser [50] is a stream reasoner supporting a fragment of LARS, called *plain LARS*, first introduced in [51]. In contrast to our work, the semantics implemented by Laser requires only that the system derives the consequences that do not depend on future facts; specifically, a consequence holding at time τ will be output if and only if it can be derived by reasoning over the interval $[0, \tau]$. By the properties of plain LARS—that can propagate into the past, but cannot distinguish the past from the present—this semantics can be implemented by an algorithm that does not propagate information backwards in time. As a consequence, the delay validity problem is not relevant in this setting. Other works on LARS address one-off [15] and incremental [51,52] reasoning over (finite) datasets, program equivalence [53], and translation of other languages into LARS [16].

Our results build on our prior conference publications [1,2]. Our stream reasoning algorithm is a variant of the "offline" algorithm in [1], where the main difference is that the algorithm in [1] only keeps in memory EDB facts from the input stream (and hence does not retain derived IDB facts from one iteration to the next). As a result, the delay and window size validity notions are defined differently. Despite these differences, however, the reductions from the containment problem to delay and window size validity problems presented in this paper (see Theorems 4.2 and 4.3) are very similar to those in [1]. Finally, our focus in [1] was on non-recursive programs, which are much weaker than backward-bounded programs. The stream reasoning algorithm we present in this paper was first introduced in [2] for forward-propagating programs, which always admit delay zero. Therefore, in [2] we focused on window size validity and showed that window size validity and containment are inter-reducible for forward-propagating programs. Furthermore, we established tight complexity bounds for containment of forward-propagating programs; in particular, the lower bounds for the containment problem presented here in Theorem 8.1 were proved in [2]. This paper extends [2] to programs that are not necessarily forward-propagating by generalising the stream reasoning algorithm, studying the window size validity problem in the extended setting, studying for the first time the problems associated with the notion of program delay, and considering the complexity of all relevant problems under both unary and binary coding of numbers.

9.4. Related problems and techniques

A question closely related to stream reasoning is that of checking *dynamic integrity constraints* for relational databases [54]. In contrast to standard database constraints, dynamic constraints can refer to different states of a database, which are determined by the relevant sequence of transactions. Chomicki [54] considers First-Order Temporal Logic (FOTL), which extends First-Order Logic with past-only temporal operators, as a language to express such dynamic constraints. Chomicki also proposes an incremental update algorithm for checking dynamic FOTL constraints. Chomicki's algorithm builds on the observation that FOTL admits an inductive definition over time, where the truth of sub-formulas at the current time point can be derived from the truth of sub-formulas at the previous time point. As a consequence, the algorithm only needs to store a polynomially-bounded part of the update history (considering the constraints fixed). When applied to forward-propagating programs, our stream reasoning algorithm behaves similarly to Chomicki's: it computes and stores all (EDB and IDB) consequences of the input program for increasing time points while keeping in memory only a polynomially bounded set of facts (considering the program fixed).

There is a connection between delay existence and checking *boundedness* of a Datalog program, that is, checking whether a program is (semantically) recursive. Intuitively, a Temporal Datalog program admits a valid delay if and only if it is non-recursive with respect to temporal backward propagation, and hence delay existence can be seen conceptually as a boundedness check with respect to temporal recursion towards past time points; note, however, that it is hard to establish formal reductions between delay existence and boundedness, as it is unclear how to check recursion towards the past while ignoring recursion towards the future. Cosmadakis et al. [55] established complexity bounds for monadic Datalog programs using techniques similar in spirit to those in Section 6, where automata are used to recognise languages related to the fact entailment problem and its complement. The automata defined in Section 6 are also related to the algorithm in [19] used by Chomicki and Imieliński to establish the data complexity of Datalog_{1S}, where the algorithm searches for counter-models over an exponentially-sized prefix of the timeline using a sliding window of polynomial size.

Stream reasoning is also related to *runtime verification* [56–59]—an area in software verification that deals with the problem of checking whether the current execution of a program violates a given correctness property expressed as a temporal formula φ. In contrast to standard verification, where the main goal is to check whether all possible runs of a system satisfy a correctness property, the focus in runtime verification is on concrete executions, which naturally correspond to finite traces (or equivalently to finite prefixes of a run). Similarly to the soundness and completeness requirements of our stream reasoning algorithm, a *monitor* in runtime verification must check whether φ will hold or not based only on a finite trace of the system. On the one hand, if the monitor reports a verdict on φ based on a finite trace, then every possible continuation of that trace must lead to the same verdict; on the other hand, if φ holds (or doesn't hold) on the complete run, then the monitor must report the same result by examining only a finite execution. Additionally, the

problem of *monitorability* of a temporal property in runtime verification—to check whether a verdict about the property can be safely determined looking only at a finite trace—is conceptually related to our notion of delay. There are, however, important differences between stream reasoning and runtime verification. First, the properties that we study are checked at design time—that is, when given a valid delay and window size our generic stream reasoning algorithm is sound and complete *for every possible input stream*; thus, we are not concerned with concrete executions of the algorithm and no checks are performed at runtime. Second, we are also concerned about forgetting parts of the history that are no longer relevant to future query answers whereas, to the best of our knowledge, this is not a main concern in the context of runtime verification. Finally, in our setting we are interested in reasoning over a set of temporal rules, whereas in runtime verification the focus is on checking whether a temporal LTL or first-order temporal logic formula holds. Although rules have also been considered in the context of runtime verification [60], the head of these rules consists of an action that updates the database by either adding or deleting; thus, such rules are much closer to database triggers than to Datalog.

10. Conclusions and future work

In this paper, we have proposed and studied a suite of decision problems which enable the use of incremental stream reasoning algorithms based on a sliding window, while ensuring correctness and minimising both latency and memory consumption. Although these problems are undecidable for Temporal Datalog, we have shown decidability and established tight complexity bounds under the assumption that the set of objects that may occur in an input stream is fixed. We believe that our results constitute a first step towards the development of robust and efficient stream reasoning engines with provable correctness guarantees.

We see many interesting avenues for future work. First, it is unclear how to extend our results to deal with programs that do not satisfy our temporal guardedness condition. A key aspect of such an extension would be to deal effectively with multiple time variables in rules. As we already mentioned, such rules can be rewritten into rules with a single time variable by introducing fresh predicates and recursion; such rewriting, however, can easily turn a program admitting a valid delay into a program with no valid delay. Furthermore, a program with multiple time variables may admit no valid window size even if it admits a valid delay. We conjecture that such challenges could be tackled by extending the stream reasoning algorithm to also store witnesses for existentially quantified time variables, as well as all facts for the time points occurring explicitly in the input program.

Second, in some scenarios it may not be reasonable to assume a fixed object domain. An interesting direction would be to explore liftings of our fixed object domain assumption, and the work on bounded theories in the situation calculus seems especially relevant to this effect [61]. In particular, the assumption in [61] can be cast in our setting by requiring that the domain of facts that our algorithm keeps in memory is bounded by a constant, which is a strictly less stringent requirement than our assumption that the entire stream mentions a bounded number of objects. Another possibility would be to investigate recursive fragments (other than object-free Temporal Datalog) for which the delay and window size problems become decidable. A natural choice is to restrict programs so that IDB predicates have at most one object argument; such programs extend monadic Datalog [55], for which containment is known to be decidable.

Third, it would be interesting to extend our framework from Section 3 to cover programs featuring non-monotonic negation. In such extended framework, our generic stream reasoning algorithm would remain largely unmodified; however, the associated notions of valid delay and valid window size (as well as the claim of Lemma 3.5) would need to be strengthened by replacing inclusions with equalities in order to take into account that the algorithm may now output unsound results if parametrised with an invalid delay or window size. Additionally, the updated notions of delay validity and window size would yield new decision problems that are at least as hard as those we consider here; the work on bidirectional ASP [43] could provide an ideal starting point to investigate the complexity of delay and window validity in this setting. Another possible way forward in this direction would be to consider an extension of the language in line with Zaniolo's sequential programs [13].

Finally, our framework can be adapted to other temporal rule languages, yielding corresponding delay and window size problems that can then be studied. In particular, it would be interesting to consider temporal languages with a dense model of time such as DatalogMTL [20,25,38].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research was supported by the SIRIUS Centre for Scalable Data Access in the Oil and Gas Domain, the EPSRC projects OASIS (EP/S032347/1) and AnaLOG (EP/P025943/1), the ERC Advanced Grant WhiteMech (No. 834228), and the EU ICT-48 2020 project TAILOR (No. 952215).

Appendix A. Proofs

Theorem 6.2. *Automaton* $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ *recognises the language* $\mathsf{implies}(\Omega, \mathbf{J})$.

Proof. We first prove that $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ accepts each word in $\mathsf{implies}(\Omega, \mathbf{J})$. Our proof relies on properties of derivations in Temporal Datalog, as well as on the connection between certain types of derivations and runs in the automaton. These properties are formalised and proved in Claims 3 and 4, respectively. We say that a derivation δ is *uniform* if, for each (ground) atom α occurring in δ, all the α-subderivations of δ coincide, and it is straightforward to show that any derivation of a fact can be transformed into a uniform derivation of the same fact (whenever a derivation has two distinct subderivations of the same fact, one can safely replace one with the other and still obtain a derivation).

Claim 3. *For* Π *a program,* D *a set of EDB facts, and* α *a fact, let* δ *be a uniform derivation of* α *from* $\Pi \cup D$*, and let* Π_δ *be the set of labels of* δ*. Then no fact depends on itself in* Π_δ*.*

It suffices to show that, if β and γ are facts such that β depends on γ in Π_δ, then the (unique) β-subderivation of δ has a strict γ-subderivation. This is shown by induction on the height $k \geq 1$ of the partial derivation justifying the dependency relation. In the base case, $k = 1$ and there is a rule $r \in \Pi_\delta$ such that β is the head of r and γ is in the body of r. Consider a subderivation δ_1 of δ whose root has label r. Clearly, δ_1 is a β-subderivation of δ and, moreover, has an immediate γ-subderivation, as required. In the inductive case, $k > 1$, and hence there is a fact ψ and a rule $r \in \Pi_\delta$ such that β is the head of r, ψ is in the body of r, and ψ depends on γ in Π_δ via $k-1$ steps. Consider a subderivation δ_1 of δ whose root has label r. Then δ_1 is a β-subderivation of δ and has an immediate ψ-subderivation δ_2. Furthermore, by the inductive hypothesis, δ_2 (which is the unique ψ-subderivation of δ since δ is uniform) has a strict γ-subderivation δ_3. Thus, δ_3 is a strict subderivation of δ_1, which concludes the proof of the claim.

Claim 4. *For each finite set of EDB facts* D*, each uniform derivation* δ *of a fact* $P(\tau)$ *with* $P \in \mathbf{J}$ *from* $\Omega \cup D$*, and each* $m \geq -1$*, the sequence* $s_{-1}\sigma_0 s_0 \dots s_{m-1}\sigma_m s_m$ *is a run of* $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ *where:*

- *for each* $i \in [-1, m]$*,* $s_i = \langle a_i, X_i, X_{i+1}, \Omega_i, \Gamma_i \rangle$ *where, for* Ω'_i *the set of all rules labelling* δ *that are not facts and have time point* i *in the head,*
  - $a_i = 1$ *if* $i < \tau$*,* $a_i = 2$ *if* $\tau \leq i \leq \tau_{\max}$*, and* $a_i = 3$ *if* $i > \tau_{\max}$ *for* $\tau_{\max}$ *the maximum time point in* $D \cup \{P(\tau)\}$*;*
  - $X_i = \{Q \mid Q(i) \in F\}$ *for* F *the union of* D *with the set of all rule heads occurring in* δ *(note that* $X_{-1} = \emptyset$*);*
  - Ω_i *is obtained from* Ω'_i *by replacing each time point* τ' *with* $\tau' - i + 1$ *(note that* $\Omega_{-1} = \emptyset$ *as* $\Omega'_{-1} = \emptyset$*);*
  - $\Gamma_i = \{\langle Q, R \rangle \mid Q(i)$ *depends on* $R(i)$ *in* $\bigcup_{j=0}^{i} \Omega'_j\}$ *(note that* $\Gamma_{-1} = \emptyset$*);*
- *for each* $i \in [0, \tau_{\max}]$*,* $\sigma_i = \langle U_i, b_i \rangle$ *where* $U_i = \{Q \mid Q(i) \in D\}$*,* $b_i = \Box$ *if* $i \neq \tau$*, and* $b_\tau = P$*;*
- *for each* $i \in [\tau_{\max} + 1, m]$*,* $\sigma_i = \varepsilon$*.*

If $m = -1$, the claim holds since $s_{-1} = \langle 1, \emptyset, X_0, \emptyset, \emptyset \rangle$, and hence s_{-1} is an initial state of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$. Hence, without loss of generality, let $0 \leq n \leq m$ be arbitrary. It suffices to show that the transition $s_{n-1} \xrightarrow{\sigma_n} s_n$ is in Δ if $n \leq \tau_{\max}$, and in Δ_ε otherwise. We consider the two cases separately.

Suppose $n \leq \tau_{\max}$. Then $\sigma_n = \langle U_n, b_n \rangle$. It suffices to show satisfaction of Conditions (2.1)–(2.5) in the definition of Δ.

Condition (2.1). Assume $b_n \in \mathbf{I}$. Then $b_n = P$ and $n = \tau$, and hence it suffices to show $P \in X_\tau$, $a_{\tau-1} = 1$, and $a_\tau = 2$. We have $P \in X_\tau$ because δ contains a rule with head $P(\tau)$, while $a_{\tau-1} = 1$ and $a_\tau = 2$ is immediate by definition.

Condition (2.2). Assume $b_n = \Box$. Then $n \neq \tau$. If $n < \tau$, we have $a_{n-1} = a_n = 1$ by definition, while $n > \tau$ implies $n - 1 \geq \tau$, and hence $a_{n-1} = a_n = 2$, as required.

Condition (2.3). We show that $X_{n-1}(0) \cup U_n(1) \cup X_{n+1}(2) \cup \Omega_n \models X_n(1)$. To this end, by construction, it suffices to show that $X_{n-1}(n-1) \cup U_n(n) \cup X_{n+1}(n+1) \cup \Omega'_n \models X_n(n)$, which can be equivalently restated as $F\restriction_{n-1} \cup D\restriction_n \cup F\restriction_{n+1} \cup \Omega'_n \models F\restriction_n$. In turn, $F\restriction_{n-1} \cup D\restriction_n \cup F\restriction_{n+1} \cup \Omega'_n \models F\restriction_n$ can be shown by a straightforward induction on derivations.

Condition (2.4). By the definition of Γ_n, it suffices to show that, for each pair of predicates Q and R, $Q(n)$ depends on $R(n)$ in $\bigcup_{j=0}^{n} \Omega'_j$ if any of the following conditions holds:

(a) $Q(1)$ depends on $R(1)$ in Ω_n,
(b) there exists a predicate T such that $Q(1)$ depends on $T(0)$ in Ω_n and $T(1)$ depends on $R(2)$ in Ω_{n-1}, or
(c) there exists $\langle T_1, T_2 \rangle \in \Gamma_{n-1}$ such that $Q(1)$ depends on $T_1(0)$ in Ω_n and $T_2(1)$ depends on $R(2)$ in Ω_{n-1}.

We consider the three cases separately. In Case (a), $Q(n)$ depends on $R(n)$ in Ω'_n, and hence also in $\bigcup_{j=0}^{n} \Omega'_j$. In Case (b), $Q(n)$ depends on $T(n-1)$ in Ω'_n and $T(n-1)$ depends on $R(n)$ in Ω'_{n-1}, and hence $Q(n)$ depends on $R(n)$ in $\bigcup_{j=0}^{n} \Omega'_j$. In

Case (c), $Q(n)$ depends on $T_1(n-1)$ in Ω'_n and $T_2(n-1)$ depends on $R(n)$ in Ω'_{n-1}. Furthermore, since $\langle T_1, T_2\rangle \in \Gamma_{n-1}$, we also have that $T_1(n-1)$ depends on $T_2(n-1)$ in $\bigcup_{j=0}^{n-1}\Omega'_j$, and hence $Q(n)$ depends on $R(n)$ in $\bigcup_{j=0}^{n}\Omega'_j$.

Condition (2.5). The condition holds by Claim 3 since δ is uniform.

Next, suppose $n > \tau_{\max}$. Then $\sigma_n = \varepsilon$. It suffices to show satisfaction of Conditions (3.1)–(3.3) in the definition of Δ_ε.

Condition (3.1). By definition, we have $a_n = 3$, $a_{n-1} = 2$ if $n = \tau_{\max}+1$ and $a_{n-1} = 3$ if $n > \tau_{\max}+1$, as required.

Condition (3.2). Similarly to the case of Condition (2.3), it suffices to show $F\restriction_{n-1} \cup F\restriction_{n+1} \cup \Omega'_n \models F\restriction_n$. This follows from the corresponding claim in Condition (2.3)—$F\restriction_{n-1} \cup D\restriction_n \cup F\restriction_{n+1} \cup \Omega'_n \models F\restriction_n$—since the claim also holds for $n > \tau_{\max}$ and since $D\restriction_n$ is empty (as $n > \tau_{\max}$).

Condition (3.3). The arguments for Conditions (2.4) and (2.5) do not depend on $n \le \tau_{\max}$, so both conditions hold for $n > \tau_{\max}$.

This concludes the proof of Claim 4.

Let now $w = \langle \tau, P, U_0, \dots, U_n\rangle$ be a word in $\mathsf{implies}(\Omega, \mathbf{J})$. We show that $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ accepts w. By definition, we have $\Omega \cup D \models P(\tau)$ for $D = \{A(j) \mid A \in U_j, 0 \le j \le n\}$. Let δ be a uniform derivation of $P(\tau)$ from $\Omega \cup D$, and let m be the maximum between n and the maximum time point occurring in δ. By Claim 4, $\rho = s_{-1}\sigma_0 s_0 \dots s_{m-1}\sigma_m s_m$ is a run of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ where *(i)* $w = \sigma_0, \dots, \sigma_n$, *(ii)* $\sigma_i = \varepsilon$ for each $i \in [n+1, m]$, and *(iii)* $s_m = \langle a, X, Y, \Omega, \Gamma\rangle$ where $a \in \{2, 3\}$ and $Y(m+1)$ is the set of atoms with time argument $m+1$ occurring in either D or δ. Note that $Y = \emptyset$ since D and δ contain no fact with time argument $m+1$ by the choice of m. Thus, s_m is a final state, meaning that ρ is an accepting run of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ on w.

We finally show that $\mathsf{implies}(\Omega, \mathbf{J})$ contains each word accepted by $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$. Let $w = \sigma_0 \dots \sigma_n$ be a word accepted by $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$. Then there exists a run $s_{-1}\sigma_0 s_0 \dots s_{m-1}\sigma_m s_m$ of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$ where $m \ge n$ and s_m is final. Since s_m is final, by the definition of $\mathcal{A}[\mathsf{implies}(\Omega, \mathbf{J})]$, there exist an integer $k \in [0, n]$, a predicate $P \in \mathbf{J}$, and sets $U_0, \dots, U_n$ of EDB predicates such that $\sigma_k = \langle U_k, P\rangle$, $\sigma_j = \langle U_j, \Box\rangle$ for each $j \in [0, n]$ with $j \ne k$, and $\sigma_j = \varepsilon$ for each $j \in [n+1, m]$. It follows that $w = \langle k, P, U_0, \dots, U_n\rangle$. Thus, it remains to show that $\Omega \cup D \models P(k)$, for $D = \bigcup_{j=0}^{n} U_j(j)$. Let, for each $i \in [-1, m]$, $s_i = \langle a_i, X_i, Y_i, \Omega_i, \Gamma_i\rangle$. Since $\sigma_k = \langle U_k, P\rangle$, we have $P \in X_k$, and hence it suffices to show that, for each $i \in [0, m]$, $\Omega \cup D \models X_i(i)$. To this end, we first show the following claims, where, for each $i \in [0, m]$, $\Omega'_i = \bigcup_{j=0}^{i} \Omega \downarrow j$.

Claim 5. *For each $i \in [0, m]$, Γ_i contains each pair $\langle P, R\rangle$ such that $P(i)$ depends on $R(i)$ in Ω'_i.*

We show the claim by induction on i. In the base case ($i = 0$), whenever $P(0)$ depends on $R(0)$ in Ω'_0, we have that $P(1)$ depends on $R(1)$ in Ω_0, and hence $\langle P, R\rangle \in \Gamma_0$ by Condition (2.4.1). In the inductive case, we have $0 < i \le m$ and we assume that the claim holds for $i-1$. Suppose that $P(i)$ depends on $R(i)$ in Ω'_i via k steps. We show $\langle P, R\rangle \in \Gamma_i$ by induction on k. In the base case ($k = 1$), the claim again holds by Condition (2.4.1) since $P(1)$ depends on $R(1)$ in Ω_i. In the inductive case, we have $k > 1$ and we assume that the claim holds for $k-1$. Then there is a rule $r \in \Omega'_i$ with head $P(i)$ and a body atom $T(j)$ that depends on $R(i)$ in Ω'_i via $k-1$ steps. We have $j \in \{i-1, i\}$, since Ω is normal and no rule of Ω'_i has head time argument $i+1$. If $i = j$, then $\langle P, T\rangle \in \Gamma_i$ by Condition (2.4.1) since $P(1)$ depends on $T(1)$ in Ω_i, and $\langle T, R\rangle \in \Gamma_i$ by the inner inductive hypothesis; $\langle P, R\rangle \in \Gamma_i$ then follows by the transitivity of Γ_i. Next, assume $j = i-1$. Since Ω is normal, either $T(i-1)$ depends on a fact $R'(i)$ via one step in Ω'_{i-1} or $T(i-1)$ depends (via possibly more then one step) on a fact $T'(i-1)$, which in turn depends on a fact $R'(i)$. We consider the two cases separately. In the first case, we have $\langle P, R'\rangle \in \Gamma_i$ by Condition (2.4.2) since $P(1)$ depends on $T(0)$ in Ω_i and $T(1)$ depends on $R'(2)$ in Ω_{i-1}. Furthermore, either $R' = R$ (which immediately implies the claim) or $R'(i)$ depends on $R(i)$ in Ω'_i via at most $k-1$ steps, which implies $\langle R', R\rangle \in \Gamma_i$ by the inner inductive hypothesis. Thus, $\langle P, R\rangle \in \Gamma_i$ holds by the transitivity of Γ_i. In the second case, we have $\langle P, R'\rangle \in \Gamma_i$ by Condition (2.4.3) since $P(1)$ depends on $T(0)$ in Ω_i, $\langle T, T'\rangle \in \Gamma_{i-1}$ by the outer inductive hypothesis, and $T'(1)$ depends on $R'(2)$ in Ω_{i-1}. Then, as before, either $R' = R$ or $R'(i)$ depends on $R(i)$ in Ω'_i via at most $k-1$ steps, and hence, once again, $\langle P, R\rangle \in \Gamma_i$. This concludes the proof of Claim 5.

Claim 6. *For each $i \in [0, m]$, we have $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Omega'_i \models X_i(i)$.*

We show the claim by induction on i. The base case ($i = 0$) holds by Condition (2.3) as $X_{-1} = \emptyset$ because s_{-1} is an initial state. In the inductive case, we have $0 < i \le m$ and we assume $D\restriction_{[0,i-1]} \cup Y_{i-1}(i) \cup \Omega'_{i-1} \models X_{i-1}(i-1)$, or, equivalently, $D\restriction_{[0,i-1]} \cup X_i(i) \cup \Omega'_{i-1} \models X_{i-1}(i-1)$ since $Y_{i-1} = X_i$. By Conditions (2.3) and (3.2), we have $X_{i-1}(i-1) \cup U(i) \cup Y_i(i+1) \cup \Omega \downarrow i \models X_i(i)$ for $U = U_i$ if $i \le n$ and $U = \emptyset$ otherwise. Thus, it suffices to show $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Omega'_i \models \alpha$ for α a fact with time argument i that is entailed by $X_{i-1}(i-1) \cup U(i) \cup Y_i(i+1) \cup \Omega \downarrow i$. We show this by induction on the rank k of α in Ω'_i, which exists because no fact depends on itself in Ω'_i by Claim 5 and Condition (2.5). In the base case, we have $k = 0$, hence α does not depend on any fact in Ω'_i, and hence $\alpha \in U(i) \subseteq D\restriction_{[0,i]}$. In the inductive case, we have $k \ge 1$ and we assume that the claim holds for $k-1$. Let δ be a derivation of α from $X_{i-1}(i-1) \cup U(i) \cup Y_i(i+1) \cup \Omega \downarrow i$, r be the rule labelling the root of δ, and β be a body atom of r. It suffices to show $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Omega'_i \models \beta$. Let j be the time argument of β. We have $j \in \{i-1, i, i+1\}$ since Ω is normal, and we consider the three cases separately. In the first case, we have $j = i$, and hence the claim holds by the inner inductive hypothesis since β has rank at most $k-1$ in Ω'_i. In the

second case, we have $j = i+1$, and hence $\beta \in Y_i(i+1)$ since β is not an instance of a head in $\Omega \downarrow i$. In the third case, we have $j = i-1$, and hence $\beta \in X_{i-1}(i-1)$ since β is not an instance of any head in $\Omega \downarrow i$. The outer inductive hypothesis yields $D\restriction_{[0,i-1]} \cup X_i(i) \cup \Omega'_{i-1} \models \beta$, and hence $D\restriction_{[0,i-1]} \cup A \cup \Omega'_{i-1} \models \beta$ for A the subset of $X_i(i)$ where each fact has rank at most $k-1$ in Ω'_i. Then, $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Omega'_i \models A$ holds by the inner inductive hypothesis, and hence the claim follows by transitivity of entailment. This concludes the proof of Claim 6.

Finally, we show $\Omega \cup D \models X_i(i)$ by induction on $m-i$. In the base case, we have $i = m$, thus $\Omega \cup D \cup Y_m(m+1) \models X_m(m)$ by Claim 6, and hence $\Omega \cup D \models X_m(m)$ since $Y_m = \emptyset$ (as s_m is final). In the inductive case, we have $i < m$, and we assume $\Omega \cup D \models X_{i+1}(i+1)$. By Claim 6, we have $\Omega \cup D \cup Y_i(i+1) \models X_i(i)$. But then, since $Y_i = X_{i+1}$, the inductive hypothesis immediately yields $\Omega \cup D \models X_i(i)$, as required. □

Theorem 6.5. *Automaton* $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ *recognises the language* $\mathsf{notimplies}(\Omega, \mathbf{J})$.

Proof. Let $w = \sigma_0 \dots \sigma_n$ be a word in $\mathsf{notimplies}(\Omega, \mathbf{J})$. We show that $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ accepts w. By definition, there exist $i \in [0, n]$, $P \in \mathbf{I}$, and sets $E_0, \dots, E_n$ of EDB predicates in Ω such that w has the form $\langle i, P, E_0, \dots, E_n \rangle$, where $\Omega \cup D \not\models P(i)$ for $D = \{A(j) \mid A \in E_j,\ 0 \leq j \leq n\}$. Thus, there is a model M of $\Omega \cup D$ that does not contain $P(i)$. Let $\rho = s_{-1}\sigma_0 s_0 \dots \sigma_n s_n \varepsilon s_{n+1} \dots \varepsilon s_{n+N}$, where each state $s_i = \langle a_i, X_i, X_{i+1}, c_i \rangle$ satisfies the following for $\tau_{\max}$ the maximum time point in D:

- $a_i = 1$ if $i < \tau$, $a_i = 2$ if $\tau \leq i \leq \tau_{\max}$, and $a_i = 3$ if $i > \tau_{\max}$;
- $X_i = \{Q \mid Q(i) \in M\}$; and
- $c_i = 0$ if $i \leq \tau_{\max}$ and $c_i = i - \tau_{\max}$ otherwise.

We argue that ϱ is an accepting run of $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$. First note that s_{n+N} is a final state as it has the form $\langle 3, X, Y, N \rangle$. It now suffices to see that the transition $s_{n-1} \xrightarrow{\sigma_n} s_n$ is in Δ if $n \leq \tau_{\max}$ and in Δ_ε otherwise. If $n \leq \tau_{\max}$, then we can check that Conditions (2.1)–(2.3) in the definition of Δ hold, where the interesting case is *Condition (2.3)*. Since M is a model for $\Omega \cup D$, we have $M \models \Omega \downarrow n \cup D\restriction_n$, and hence $M\restriction_{n-1} \cup M\restriction_n \cup M\restriction_{n+1} \models \Omega \downarrow n \cup D\restriction_n$ as Ω is normal and hence each time point occurring in $\Omega \downarrow n \cup D\restriction_n$ is in $[n-1, n+1]$; equivalently, this can be written as $X_{n-1}(n-1) \cup X_n(n) \cup X_{n+1}(n+1) \models \Omega \downarrow n \cup E_n(n)$, as required. If $n > \tau_{\max}$, then $\sigma_n = \varepsilon$, $a_{n-1} \in \{2, 3\}$, $a_n = 3$, $c_n = c_{n-1} + 1$, and $c_n \in [1, N]$, as required; furthermore, the justification of Condition (2.3) above also proves Condition (3.2) since $D\restriction_n = \emptyset$. As a result, ρ is an accepting run.

We next show that $\mathsf{notimplies}(\Omega, \mathbf{J})$ contains each word accepted by $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$. Let $w = \sigma_0, \dots, \sigma_n$ be a word accepted by $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$. Then there exists a run $\rho = s_{-1}\sigma_0 s_0 \dots \sigma_{n+N} s_{n+N}$ of $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ where s_{n+N} is final, and $s_j = \langle a_j, Y_j, Z_j, c_j \rangle$ for $j \in [0, n+N]$. Since s_{n+N} is final, by the definition of $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$ there exists $k \in [0, n]$ such that $\sigma_k = \langle U_k, P \rangle$ with $P \in \mathbf{J}$, $\sigma_j = \langle U_j, \square \rangle$ for each $j \in [0, n]$ with $j \neq k$, and $\sigma_j = \varepsilon$ for each $j \in [n+1, n+N]$. It follows that $w = \langle k, P, U_0, \dots, U_n \rangle$. Thus, it remains to show that $\Omega \cup \bigcup_{j=0}^{n} U_j(j) \not\models P(k)$, which we do next by constructing a model M of $\Omega \cup \bigcup_{j=0}^{n} U_j(j)$ such that $P(k) \notin M$. To this end, we first show the following claim.

Claim 7. *There exist integers* $\phi \in [0, N]$ *and* $T \in [1, N-\phi]$ *such that, for each* $i \geq 1$, $\rho_i = s_{-1}\sigma_0 \dots \sigma_{n+\phi} s_{n+\phi} \varepsilon s'_1 \dots \varepsilon s'_i$ *is an accepting run of* $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J}), \phi + i]$, *where* $s'_j = \langle 3, Y_{n+\phi+(j \bmod T)}, Y_{n+\phi+(j+1 \bmod T)}, \phi + j \rangle$ *for each* $j \in [1, i]$.

Since ρ is a run of $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$, there exist integers j_1 and j_2 in $[n, n+N]$ with $j_1 < j_2$ such that $Y_{j_1} = Y_{j_2}$ and $Z_{j_1} = Z_{j_2}$; indeed, $|[n, n+N]| = N+1$ while there are at most N distinct pairs $\langle Y, Z \rangle$ such that, for $a \in \{2, 3\}$ and some integer c, $\langle a, Y, Z, c \rangle$ is a state of $\mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J})]$. Let $\phi = j_1 - n$ and $T = j_2 - j_1$. Clearly, $\phi \in [0, N]$ and $T \in [1, N-\phi]$, as required. For the rest, let $\mathcal{A}_i = \mathcal{A}[\mathsf{notimplies}(\Omega, \mathbf{J}), \phi + i]$ for each $i \geq 0$. For $1 \leq i \leq T$, ρ_i is an accepting run of $\mathcal{A}_i$ as it is a prefix of ρ and as $s'_i = s_{n+\phi+i}$ is final in $\mathcal{A}_i$. So, without loss of generality, let $i > T$. Let us refer to $s_{n+\phi}$ as s'_0. Clearly, s'_i is again final in $\mathcal{A}_i$ so it suffices to show that, for each $j \in [T, i]$, $s'_{j-1} \xrightarrow{\varepsilon} s'_j$ is a transition of $\mathcal{A}_i$. This follows as $s'_{j-1 \bmod T} \xrightarrow{\varepsilon} s'_{j \bmod T}$ is a transition of $\mathcal{A}_i$, s'_{j-1} coincides with $s'_{j-1 \bmod T}$ in its second and third component, s'_j coincides with $s'_{j \bmod T}$ in its second and third component, and the first and last components of s'_{j-1} and s'_j are as required by ε-transitions of $\mathcal{A}_i$. This concludes the proof of Claim 7.

Let now ϕ and T be as asserted by the claim. We define

$$M = \bigcup_{j=0}^{n+\phi} Y_j(j) \cup \bigcup_{j \geq 1} Y_{n+\phi+(j \bmod T)}(n+\phi+j)$$

We conclude the proof by showing that M is as required. First, we have $P(k) \notin M$ by Condition (2.1). Second, we have $\bigcup_{j=0}^{n} U_j(j) \subseteq \bigcup_{j=0}^{n} Y_j(j) \subseteq M$, where the first containment holds by Condition (2.3). Third, we show $M \models r$ for an arbitrary rule $r \in \Omega$. Since Ω is normal, it suffices to show, for each $i \geq 0$, $M\restriction_{[i-1,i+1]} \models r \downarrow i$. Let $k_j = j$ for $j \leq n+\phi$, and $k_j = n+\phi+(j-n-\phi \bmod T)$ for $j \geq n+\phi+1$. Then, $M\restriction_{[i-1,i+1]} \models r \downarrow i$ can be restated as $Y_{k_{i-1}}(i-1) \cup Y_{k_i}(i) \cup Y_{k_{i+1}}(i+1) \models r \downarrow i$, which holds by Claim 7 together with Conditions (2.3) and (3.2). □

Theorem 8.2. *Containment of object-free forward-propagating programs is* ExpSpace*-hard if time offsets are coded in binary.*

Proof. We provide a reduction from containment of succinct regular expressions (SREs). Our reduction maps a pair $\langle R_1, R_2 \rangle$ of SREs over a finite alphabet Σ to a pair $\langle \Pi_1, \Pi_2 \rangle$ of object-free forward-propagating programs such that $\Pi_1 \sqsubseteq \Pi_2$ if and only if $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$, with $\mathcal{L}(R_i)$ the language of R_i. Programs Π_1 and Π_2 consist of the following components, where $e_0, e_1, \ldots, e_{c-1}$ are the exponents (with repetitions) occurring in R_1 or R_2, and k is the maximum number of bits required to encode any of them.

- Program Π_{count} implements a binary counter, and it consists of the following rules, where $Start$, $Flip_i$, $NoFlip_i$, $Zero_i$, and One_i are fresh unary predicates:

$$Start(t) \rightarrow Zero_i(t) \qquad \forall i \in [1, c \cdot k] \tag{A.1}$$

$$\left(\bigwedge_{\ell=1}^{i-1} One_\ell(t) \right) \wedge Zero_i(t) \rightarrow Flip_j(t) \qquad \forall i \in [1, c \cdot k], \forall j \in [1, i] \tag{A.2}$$

$$\left(\bigwedge_{\ell=1}^{i-1} One_\ell(t) \right) \wedge Zero_i(t) \rightarrow NoFlip_j(t) \qquad \forall i \in [1, c \cdot k], \forall j \in (i, c \cdot k] \tag{A.3}$$

$$Zero_i(t) \wedge Flip_i(t) \rightarrow One_i(t+1) \qquad \forall i \in [1, c \cdot k] \tag{A.4}$$

$$One_i(t) \wedge Flip_i(t) \rightarrow Zero_i(t+1) \qquad \forall i \in [1, c \cdot k] \tag{A.5}$$

$$Zero_i(t) \wedge NoFlip_i(t) \rightarrow Zero_i(t+1) \qquad \forall i \in [1, c \cdot k] \tag{A.6}$$

$$One_i(t) \wedge NoFlip_i(t) \rightarrow One_i(t+1) \qquad \forall i \in [1, c \cdot k] \tag{A.7}$$

  Let us say, for a given set D of EDB facts, that a time point τ encodes an integer $m \in [0, 2^{c \cdot k} - 1]$ if $\Pi_{\text{count}} \cup D \models \{B_1(\tau), \ldots, B_{c \cdot k}(\tau)\}$ where B_i is $Zero_i$ if the i-th bit in the binary coding of m is zero and One_i otherwise. Then, the construction of Π_{count} ensures that, for each set D of EDB facts, each time point τ, and each integer $m \geq 0$, if $Start(\tau) \in D$ then $\tau + m$ encodes $m \bmod 2^{c \cdot k}$. Conversely, for each D, each time point τ such that there is at most one time point $\tau' \leq \tau$ with $Start(\tau') \in D$, and each $m \in [0, 2^{c \cdot k} - 1]$, if τ encodes m then $Start(\tau - m - a \cdot 2^{c \cdot k}) \in D$ for some $a \geq 0$.
- Program Π_{input} consists of the following rules, where A_σ and A'_σ are fresh unary predicates for each $\sigma \in \Sigma$[5]:

$$\left(\bigwedge_{i=1}^{c \cdot k} Zero_i(t) \right) \wedge A_\sigma(t) \rightarrow A'_\sigma(t) \quad \forall \sigma \in \Sigma \tag{A.8}$$

$$\left(\bigvee_{i=1}^{c \cdot k} One_i(t) \right) \wedge A'_\sigma(t-1) \rightarrow A'_\sigma(t) \quad \forall \sigma \in \Sigma \tag{A.9}$$

  The construction of Π_{input} ensures that, for each set D of EDB facts, each time point τ, and each letter $\sigma \in \Sigma$, $\Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \models A'_\sigma(\tau)$ if and only if $A_\sigma(\tau') \in D$ with τ' the largest time point smaller than or equal to τ that encodes zero.
- Program Π_{initial} consists of the following rule, where F is a fresh unary predicate:

$$Start(t) \rightarrow F(t). \tag{A.10}$$

- For a given SRE R over alphabet Σ, program Π_R is constructed inductively as follows, where G is a fresh unary predicate, and $\phi(\Pi)$ and $\psi(\Pi)$ are the programs obtained from an arbitrary program Π by renaming each predicate P to fresh predicates P^ϕ and P^ψ, respectively, unless P is of the form A'_σ, $Zero_i$, or One_i.
  - If $R = \emptyset$, then $\Pi_R = \emptyset$.
  - If $R = \varepsilon$, then Π_R consists of the rule

$$F(t) \rightarrow G(t). \tag{A.11}$$

  - If $R = \sigma$ for $\sigma \in \Sigma$, then Π_R consists of the rule, where each F_i is a fresh unary predicate, each B^i_ℓ is $Zero_\ell$ if the ℓ-th bit in the binary coding of e_i is zero and One_i otherwise, and, dually, each predicate $\bar{B}^i_\ell$ is One_ℓ if the ℓ-th bit in the binary coding of e_i is zero and $Zero_i$ otherwise.[5]

[5] We use disjunction in the body of rules (A.9), (A.14), and (A.27), for the sake of clarity. Each rule of this kind can be easily rewritten into a set of disjunction-free rules, with each rule having one of the disjuncts in place of the disjunction.

$$F(t) \wedge A'_\sigma(t) \to F_0(t) \tag{A.12}$$

$$\left(\bigwedge\nolimits_{\ell=i\cdot k}^{((i+1)\cdot k)-1} B^i_\ell(t)\right) \wedge F_i(t) \to F_{i+1}(t + (2^k-1)\cdot 2^{i\cdot k} - e_i \cdot 2^{i\cdot k}) \quad \forall i \in [0, c-1] \tag{A.13}$$

$$\left(\bigvee\nolimits_{\ell=i\cdot k}^{((i+1)\cdot k)-1} \bar{B}^i_\ell(t)\right) \wedge F_i(t) \to F_{i+1}(t + (2^k-1)\cdot 2^{i\cdot k}) \quad \forall i \in [0, c-1] \tag{A.14}$$

$$F_c(t) \to G(t+1) \tag{A.15}$$

Intuitively, Π_σ mimics reading a letter σ at the current pointer's position and then moving the pointer to the next cell. Furthermore, it resets those counters that have reached the value of the corresponding exponent, according to the integer coded by the time point where rule (A.12) is fired. Rules (A.13) and (A.14) for the same i are complementary, in the sense that just one of them fires; also, the two rules check and set the value of the i-th counter independently from the others. If no counter is to be reset, no rule of the form (A.13) is fired, and the pointer is moved by $1 + \sum_{i=0}^{c-1}(2^k-1)\cdot 2^{i\cdot k} = 2^{c\cdot k}$ time points; this leaves the value of each counter unchanged. If instead a rule of the form (A.13) is fired the pointer is increased by $e_i \cdot 2^{i\cdot k}$ less than if the corresponding rule (A.14) was fired; this ensures that the pointer is moved to a time point where the i-th counter has value zero.

– If $R = S \cup T$, then Π_R extends $\phi(\Pi_S) \cup \psi(\Pi_T)$ with the rules

$$F(t) \to F^\phi(t) \tag{A.16}$$

$$F(t) \to F^\psi(t) \tag{A.17}$$

$$G^\phi(t) \to G(t) \tag{A.18}$$

$$G^\psi(t) \to G(t). \tag{A.19}$$

– If $R = S \circ T$, then Π_R extends $\phi(\Pi_S) \cup \psi(\Pi_T)$ with the rules

$$F(t) \to F^\phi(t) \tag{A.20}$$

$$G^\phi(t) \to F^\psi(t) \tag{A.21}$$

$$G^\psi(t) \to G(t). \tag{A.22}$$

– If $R = S^+$, then Π_R extends $\phi(\Pi_S)$ with the rules

$$F(t) \to F^\phi(t) \tag{A.23}$$

$$G^\phi(t) \to F^\phi(t) \tag{A.24}$$

$$G^\phi(t) \to G(t). \tag{A.25}$$

– If $R = S^{e_i}$ with $e_i \geq 1$, then Π_R is constructed from Π_S as follows. First, we rename each predicate P in Π_S to a fresh predicate P_{e_i}, unless P is of the form A'_σ, $Zero_i$, or One_i. Then, we extend the resulting program with the following rules:[5]

$$F(t) \to F_{e_i}(t + 2^{i\cdot k}) \tag{A.26}$$

$$\left(\bigvee\nolimits_{\ell=i\cdot k}^{((i+1)\cdot k)-1} One_\ell(t)\right) \wedge G_{e_i}(t) \to F_{e_i}(t + 2^{i\cdot k}) \tag{A.27}$$

$$\left(\bigwedge\nolimits_{\ell=i\cdot k}^{((i+1)\cdot k)-1} Zero_\ell(t)\right) \wedge G_{e_i}(t) \to G(t). \tag{A.28}$$

Intuitively, rules (A.26) and (A.27) increase the i-th counter to keep track of the number of times the subexpression S has been matched, and rule (A.28) fires when the i-th counter has just been reset by rules (A.12)–(A.15) because it had reached value e_i.

- Program Π_{flood} consists of the following rules, where *Flood* and *FloodAux* are fresh unary predicates.

$$Start(t) \to FloodAux(t+1) \tag{A.29}$$

$$FloodAux(t) \wedge Start(t) \to Flood(t) \tag{A.30}$$

$$Flood(t) \to Flood(t+1) \tag{A.31}$$

$$Flood(t) \to G(t). \tag{A.32}$$

The construction of Π_{flood} ensures that, for each EDB stream S and each time point τ, $\Pi_{\text{flood}} \cup S \models G(\tau)$ if there are two time points τ_1 and τ_2 such that $\tau_1 < \tau_2 \leq \tau$ and $\{Start(\tau_1), Start(\tau_2)\} \subseteq S$.

Then, Π_1 and Π_2 are defined as follows, where program Π'_{R_2} is obtained from Π_{R_2} by renaming each IDB predicate P to a fresh predicate P' (in particular, G is renamed to G'), and G^* is a fresh unary predicate:

- $\Pi_1 = \Pi_{R_1} \cup \Pi'_{R_2} \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup \Pi_{\text{initial}} \cup \Pi_{\text{flood}} \cup \{G(t) \rightarrow G^*(t)\}$,
- $\Pi_2 = \Pi_{R_1} \cup \Pi'_{R_2} \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup \Pi_{\text{initial}} \cup \Pi_{\text{flood}} \cup \{G'(t) \rightarrow G^*(t)\}$.

We next argue correctness of the reduction. For this, we first show that our construction captures the language of the relevant SREs in the sense of the following two claims.

Claim 8. *Let $s = \sigma_1 \dots \sigma_n$ be a word in $\mathcal{L}(R)$, let D be a finite set of EDB facts, let τ be a time point. If $Start(\tau) \in D$, and $A_{\sigma_i}(\tau + (i-1) \cdot 2^{c \cdot k}) \in D$ for each $i \in [1, n]$, then $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup \Pi_{\text{initial}} \cup D \models G(\tau + n \cdot 2^{c \cdot k})$.*

We prove a more general claim: if $Start(\tau) \in D$, $A_{\sigma_i}(\tau + (i-1) \cdot 2^{c \cdot k}) \in D$ for each $i \in [1, n]$, and $m_0, \dots, m_{c-1}$ are integers with $m_i = 0$ if e_i occurs in R and $m_i \in [0, e_i]$ otherwise, then $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau + \sum_\ell m_\ell \cdot 2^{\ell \cdot k})\} \models G(\tau + n \cdot 2^{c \cdot k} + \sum_{\ell : m_\ell < e_\ell} m_\ell \cdot 2^{\ell \cdot k})$. To see that the new claim implies the original one, it suffices to take $m_0 = \dots = m_{c-1} = 0$ and to note that $\Pi_{\text{initial}} \cup \{Start(\tau)\} \models F(\tau)$. Next we show the claim by induction on R. To be concise, let $\Pi = \Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau + \sum_\ell m_\ell \cdot 2^{\ell \cdot k})\}$. Consider the base cases. Clearly, $R \neq \emptyset$ as $s \in \mathcal{L}(R)$. If $R = \varepsilon$, then $s = \varepsilon$ (and hence $n = 0$); but then, $\Pi_R \cup D \models G(\tau)$ by rule (A.11). Consider now the case $R = \sigma$. We have $s = \sigma$. Then, $\{Start(\tau), A_\sigma(\tau)\} \subseteq D$ implies $\Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \models A'_\sigma(\tau + \sum_\ell m_\ell \cdot 2^{\ell \cdot k})$, and hence $\Pi \models G(\tau + (\sum_\ell m_\ell \cdot 2^{\ell \cdot k}) + 2^{c \cdot k} - (\sum_{\ell : m_\ell = e_\ell} m_\ell \cdot 2^{\ell \cdot k}))$ by rules (A.12)–(A.15), noting that $1 + \sum_{i=0}^{c-1} (2^k - 1) \cdot 2^{i \cdot k} = 2^{c \cdot k}$. The former entailment can be simplified into the claimed one. Next we consider the inductive case for $R = S^{e_i}$. The other inductive cases are analogous to the ones of Claim 1, and hence we omit them. When $R = S^{e_i}$, we have $s = s_1 s_2 \dots s_{e_i}$ with $s_j = \sigma_1^j \dots \sigma_{n_j}^j \in \mathcal{L}(S)$ for each $j \in [1, e_i]$. We first show by induction on $j \in [1, e_i - 1]$ that $\Pi \models G_{e_i}(\tau + j \cdot 2^{i \cdot k} + (\sum_{u=1}^{j} n_u \cdot 2^{c \cdot k}) + p)$, where p is an abbreviation for $\sum_{\ell : m_\ell < e_\ell} m_\ell \cdot 2^{\ell \cdot k}$. In the base case, $j = 1$, we have $\Pi \models F_{e_i}(\tau + 2^{i \cdot k} + \sum_\ell m_\ell \cdot 2^{\ell \cdot k})$ by rule (A.26), and hence $\Pi \models G_{e_i}(\tau + 2^{i \cdot k} + n_1 \cdot 2^{c \cdot k} + p)$ by the construction of Π_R, and because $\Pi \models G(\tau + n_1 \cdot 2^{c \cdot k} + p)$ by the outer inductive hypothesis for S. For $j > 2$, the inner inductive hypothesis yields $\Pi \models G_{e_i}(\tau + (j-1) \cdot 2^{i \cdot k} + (\sum_{u=1}^{j-1} n_u \cdot 2^{c \cdot k}) + p)$. Hence, $\Pi \models F_{e_i}(\tau + j \cdot 2^{i \cdot k} + (\sum_{u=1}^{j-1} n_u \cdot 2^{c \cdot k}) + p)$ by rule (A.27), since the binary encoding of $(j-1) \cdot 2^{i \cdot k} + (\sum_{u=1}^{j-1} n_u \cdot 2^{c \cdot k}) + p)$ contains a one in some of the positions $[i \cdot k, (i+1) \cdot k) - 1]$. But then, $\Pi \models G_{e_i}(\tau + j \cdot 2^{i \cdot k} + (\sum_{u=1}^{j} n_u \cdot 2^{c \cdot k}) + p)$ follows by the construction of Π_R from $\Pi_S \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau + j \cdot 2^{i \cdot k} + (\sum_{u=1}^{j-1} n_u \cdot 2^{c \cdot k}) + p)\} \models G(\tau + j \cdot 2^{i \cdot k} + (\sum_{u=1}^{j} n_u \cdot 2^{c \cdot k}) + p)$, which holds by the outer inductive hypothesis for S. Then, the case $j = e_i = 1$ follows similarly to the base case above, and the case $j = e_i > 1$ follows similarly to the inductive case above, using the claim above in place of the inductive hypothesis; the difference with the case above is that the term $j \cdot 2^{i \cdot k}$ does not appear in the offset of the entailed fact. Thus, $\Pi \models G_{e_i}(\tau + (\sum_{u=1}^{j} n_u \cdot 2^{c \cdot k}) + p)$, which implies $\Pi \models G(\tau + (\sum_{u=1}^{j} n_u \cdot 2^{c \cdot k}) + p)$ by rule (A.28), since the binary encoding of $(\sum_{u=1}^{j} n_u \cdot 2^{c \cdot k}) + p$ has all zeroes in positions $[i \cdot k, (i+1) \cdot k - 1]$. The former entailment can be easily rewritten into the claimed one.

Claim 9. *Let D be a finite set of EDB facts and let τ be a time point. If $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup \Pi_{\text{initial}} \cup D \models G(\tau)$, and there is at most one time point $\tau' \leq \tau$ such that $Start(\tau') \in D$, then there exists a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(R)$ such that $Start(\tau - n \cdot 2^{c \cdot k}) \in D$ and $A_{\sigma_i}(\tau - (n - i + 1) \cdot 2^{c \cdot k}) \in D$ for each $i \in [1, n]$.*

We prove a more general claim: for each time point τ_F, if $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau_F)\} \models G(\tau)$, D does not mention F, and there is at most one time point $\tau' \leq \tau$ such that $Start(\tau') \in D$, then there exist a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(R)$, an integer $m \geq n \cdot 2^{c \cdot k}$, and integers $m_0, \dots, m_{c-1}$ such that:

- $Start(\tau - m) \in D$,
- $A_{\sigma_i}(\tau - (n - i + 1) \cdot 2^{c \cdot k} - (m \bmod 2^{c \cdot k})) \in D$ for each $i \in [1, n]$,
- $\tau_F = \tau - n \cdot 2^{c \cdot k} + (\sum_\ell m_\ell \cdot 2^{\ell \cdot k})$ where $m_\ell = 0$ if e_ℓ occurs in R or the binary encoding of m has a one in some position $[\ell \cdot k, (\ell + 1) \cdot k - 1]$, and $m_\ell \in \{0, e_\ell\}$ otherwise.

To see that the new claim implies the original one, it suffices to observe that necessarily $m = n \cdot 2^{c \cdot k}$ and $m_0 = \dots = m_{c-1} = 0$, since F is IDB with respect to Π_{initial} and $F(\tau_F)$ is entailed only if $Start(\tau_F) \in D$. Next we show the claim by induction on R. For the base cases, note first that $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau_F)\} \models G(\tau)$ implies that $R \neq \emptyset$. If $R = \varepsilon$ we take $s = \varepsilon$ since $Start(\tau - m) \in D$ for some $m \geq 0$ by the construction of $\Pi_R \cup \Pi_{\text{count}}$, and $\tau_F = \tau$ taking each $m_\ell = 0$. If $R = \sigma_1$ we take $s = \sigma$ since $\{Start(\tau - m), A_\sigma(\tau - 2^{c \cdot k} - (m \bmod 2^{c \cdot k}))\} \subseteq D$ for some $m \geq 2^{c \cdot k}$ by the construction of $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}}$, and each integer $m_\ell \in \{0, e_\ell\}$ exists because the necessary $F_{\ell+1}$-fact is entailed either by rule (A.13) or by rule (A.14). Next we consider the inductive case for $R = S^{e_i}$. The other inductive cases are analogous to the ones of Claim 2, and hence we omit them. When $R = S^{e_i}$, by the construction of Π_R and the inductive hypothesis, the assertion $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F_{e_i}(\tau'_F)\} \models G_{e_i}(\tau')$ implies the existence of a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(S)$ and integers m and

$m_0, \dots, m_{c-1}$ with the properties stated in the claim. Furthermore, if $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau_F)\} \models F_{e_i}(\tau')$, then $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau_F)\} \models G_{e_i}(\tau' - 2^{i \cdot k})$ due to rule (A.27), if the binary encoding of m has a one in some position $[i \cdot k, (i+1) \cdot k - 1]$. Considering that $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau_F)\} \models G_{e_i}(\tau)$ and m has all zeroes in positions $[i \cdot k, (i+1) \cdot k - 1]$ due to rule (A.28), the two properties above imply (as it can be shown by a straightforward induction on j from e_i to 1) the existence of words $s_1, \dots, s_{e_i}$ and a partition $L_1, \dots, L_{e_i}$ of the indices $[0, c-1] \setminus \{i\}$ such that, for each $j \in [1, e_i]$:

- $s_j = \sigma_1^j \dots \sigma_{n_j}^j \in \mathcal{L}(S)$;
- $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau_F)\} \models F_{e_i}(\tau - (\sum_{\ell=j}^{e_i} n_\ell) \cdot 2^{c \cdot k} + (\sum_{\ell \in L_j} m_\ell \cdot 2^{\ell \cdot k}) + j \cdot 2^{i \cdot k})$;
- $A_{\sigma_\ell^j}(\tau - (\sum_{u=j}^{e_i} n_u - \ell + 1) \cdot 2^{c \cdot k} - (m \bmod 2^{c \cdot k})) \in D$ for each $\ell \in [1, n_j]$.

Furthermore, $\Pi_R \cup \Pi_{\text{count}} \cup \Pi_{\text{input}} \cup D \cup \{F(\tau_F)\} \models F_{e_i}(\tau - (\sum_{\ell=1}^{e_i} n_\ell) \cdot 2^{c \cdot k} + (\sum_{\ell \in L_1} m_\ell \cdot 2^{\ell \cdot k}) + 2^{i \cdot k})$ implies $\tau_F = \tau - (\sum_{\ell=1}^{e_i} n_\ell) \cdot 2^{c \cdot k} + (\sum_{\ell \in L_1} m_\ell \cdot 2^{\ell \cdot k})$ due to rule (A.26). Then, the word $s = s_1 \cdots s_{e_i}$, and integers m and $m_0, \dots, m_{c-1}$ are as required. This concludes the proof of the claim.

Given the claims above, the correctness proof proceeds as the one of Theorem 8.1, once we observe that the two programs trivially entail the same facts for each time point τ such that there are two time points $\tau_1, \tau_2 \leq \tau$ for which a *Start*-fact holds, by the properties of Π_{flood}. □

References

[1] A. Ronca, M. Kaminski, B. Cuenca Grau, B. Motik, I. Horrocks, Stream reasoning in Temporal Datalog, in: Proc. 32nd AAAI Conference on Artificial Intelligence, AAAI 2018, AAAI Press, 2018, pp. 1941–1948.
[2] A. Ronca, M. Kaminski, B. Cuenca Grau, I. Horrocks, The window validity problem in rule-based stream reasoning, in: Proc. 16th International Conference on Principles of Knowledge Representation and Reasoning, KR 2018, AAAI Press, 2018, pp. 571–581.
[3] G. Nuti, M. Mirghaemi, P.C. Treleaven, C. Yingsaeree, Algorithmic trading, Computer 44 (2011) 61–69.
[4] C. Cosad, K. Dufrene, K. Heidenreich, M. McMillon, A. Jermieson, M. O'Keefe, L. Simpson, Wellsite support from afar, Oilfield Rev. 21 (2009) 48–58.
[5] G. Münz, G. Carle, Real-time analysis of flow data for network attack detection, in: Proc. 10th IFIP/IEEE International Symposium on Integrated Network Management, IM 2007, IEEE, 2007, pp. 100–108.
[6] S. Babu, J. Widom, Continuous queries over data streams, SIGMOD Rec. 30 (2001) 109–120.
[7] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2002, ACM, 2002, pp. 1–16.
[8] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution, VLDB J. 15 (2006) 121–142.
[9] D.F. Barbieri, D. Braga, S. Ceri, E. Della Valle, M. Grossniklaus, Incremental reasoning on streams and rich background knowledge, in: Proc. 7th Extended Semantic Web Conference, ESWC 2010, in: LNCS, vol. 6088, Springer, 2010, pp. 1–15.
[10] J. Calbimonte, Ó. Corcho, A.J.G. Gray, Enabling ontology-based access to streaming data sources, in: Proc. 9th International Semantic Web Conference, ISWC 2010, in: LNCS, vol. 6496, Springer, 2010, pp. 96–111.
[11] D. Anicic, P. Fodor, S. Rudolph, N. Stojanovic, EP-SPARQL: a unified language for event processing and stream reasoning, in: Proc. 20th International Conference on World Wide Web, WWW 2011, ACM, 2011, pp. 635–644.
[12] D.L. Phuoc, M. Dao-Tran, J.X. Parreira, M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: Proc. 10th International Semantic Web Conference, ISWC 2011, in: LNCS, vol. 7031, Springer, 2011, pp. 370–388.
[13] C. Zaniolo, Logical foundations of continuous query languages for data streams, in: Proc. 2nd International Workshop on the Resurgence of Datalog in Academia and Industry, Datalog 2.0 2012, in: LNCS, vol. 7494, Springer, 2012, pp. 177–189.
[14] Ö.L. Özçep, R. Möller, C. Neuenstadt, A stream-temporal query language for ontology based data access, in: Proc. 37th Annual German Conference on AI, KI 2014, in: LNCS, vol. 8736, Springer, 2014, pp. 183–194.
[15] H. Beck, M. Dao-Tran, T. Eiter, LARS: a logic-based framework for analytic reasoning over streams, Artif. Intell. 261 (2018) 16–70.
[16] M. Dao-Tran, H. Beck, T. Eiter, Towards comparing RDF stream processing semantics, in: Proc. 1st Workshop on High-Level Declarative Stream Processing, HiDeSt 2015, in: CEUR Workshop Proceedings, CEUR-WS.org, vol. 1447, 2015, pp. 15–27.
[17] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (2001) 374–425.
[18] S. Abiteboul, R. Hull, V. Vianu (Eds.), Foundations of Databases, Addison-Wesley, 1995.
[19] J. Chomicki, T. Imieliński, Temporal deductive databases and infinite objects, in: Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1988, ACM, 1988, pp. 61–73.
[20] S. Brandt, E.G. Kalayci, V. Ryzhikov, G. Xiao, M. Zakharyaschev, Querying log data with Metric Temporal Logic, J. Artif. Intell. Res. 62 (2018) 829–877.
[21] M. Abadi, Z. Manna, Temporal logic programming, J. Symb. Comput. 8 (1989) 277–295.
[22] M. Baudinet, J. Chomicki, P. Wolper, Temporal deductive databases, in: A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass (Eds.), Temporal Databases, Benjamin Cummings, 1993, pp. 294–320.
[23] T. Eiter, M. Fink, H. Tompits, S. Woltran, Strong and uniform equivalence in answer-set programming: characterizations and complexity results for the non-ground case, in: Proc. 20th National Conference on Artificial Intelligence, AAAI 2005, AAAI Press/the MIT Press, 2005, pp. 695–700.
[24] O. Shmueli, Equivalence of Datalog queries is undecidable, J. Log. Program. 15 (1993) 231–241.
[25] P.A. Wałęga, M. Kaminski, B. Cuenca Grau, Reasoning over streaming data in Metric Temporal Datalog, in: Proc. 33rd AAAI Conference on Artificial Intelligence, AAAI 2019, AAAI Press, 2019, pp. 3092–3099.
[26] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, MillWheel: fault-tolerant stream processing at Internet scale, Proc. VLDB Endow. 6 (2013) 1033–1044.
[27] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Combining rewriting and incremental materialisation maintenance for Datalog programs with equality, in: Proc. 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, AAAI Press, 2015, pp. 3127–3133.
[28] B. Motik, Y. Nenov, R. Piro, I. Horrocks, D. Olteanu, Parallel materialisation of Datalog programs in centralised, main-memory RDF systems, in: Proc. 28th AAAI Conference on Artificial Intelligence, AAAI 2014, AAAI Press, 2014, pp. 129–137.
[29] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Maintenance of Datalog materialisations revisited, Artif. Intell. 269 (2019) 76–136.
[30] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (2006) 499–562.

[31] J. Baget, M. Leclère, M. Mugnier, S. Rocher, C. Sipieter, Graal: a toolkit for query answering with existential rules, in: Proc. 9th International RuleML Symposium, RuleML 2015, in: LNCS, vol. 9202, Springer, 2015, pp. 328–344.
[32] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.
[33] M. Sipser, Introduction to the Theory of Computation, 2nd ed., Thomson Course Technology, 2006.
[34] A. Bolles, M. Grawunder, J. Jacobi, Streaming SPARQL: extending SPARQL to process data streams, in: Proc. 5th European Semantic Web Conference, ESWC 2008, in: LNCS, vol. 5021, Springer, 2008, pp. 448–462.
[35] D.F. Barbieri, D. Braga, S. Ceri, E.D. Valle, M. Grossniklaus, C-SPARQL: a continuous query language for RDF data streams, Int. J. Semant. Comput. 4 (2010) 3–25.
[36] D. Dell'Aglio, E.D. Valle, J. Calbimonte, Ó. Corcho, RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems, Int. J. Semantic Web Inf. Syst. 10 (2014) 17–44.
[37] J. Chomicki, Polynomial time query processing in temporal deductive databases, in: Proc. 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1990, ACM, 1990, pp. 379–391.
[38] P.A. Wałęga, B. Cuenca Grau, M. Kaminski, E. Kostylev, DatalogMTL: computational complexity and expressive power, in: Proc. 28th International Joint Conference on Artificial Intelligence, IJCAI 2019, IJCAI Organization, 2019, pp. 1886–1892.
[39] J. Chomicki, T. Imieliński, Relational specifications of infinite query answers, in: Proc. 1989 ACM SIGMOD International Conference on Management of Data, SIGMOD 1989, ACM, 1989, pp. 174–183.
[40] D. Toman, J. Chomicki, Datalog with integer periodicity constraints, J. Log. Program. 35 (1998) 263–290.
[41] M. Baudinet, On the expressiveness of temporal logic programming, Inf. Comput. 117 (1995) 157–180.
[42] M.Y. Vardi, A temporal fixpoint calculus, in: Proc. 15th Annual ACM Symposium on Principles of Programming Languages, POPL 1988, ACM, 1988, pp. 250–259.
[43] T. Eiter, M. Simkus, Bidirectional answer set programs with function symbols, in: Proc. 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, IJCAI Organization, 2009, pp. 765–771.
[44] T. Eiter, M. Simkus, FDNC: decidable nonmonotonic disjunctive logic programs with function symbols, ACM Trans. Comput. Log. 11 (2010) 14:1–14:50.
[45] P.A. Wałęga, B. Cuenca Grau, M. Kaminski, E. Kostylev, Tractable fragments of Datalog with metric temporal operators, in: Proc. Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, 2020, pp. 1919–1925.
[46] P.A. Wałęga, B. Cuenca Grau, M. Kaminski, E. Kostylev, DatalogMTL over the integer timeline, in: Proc. 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, IJCAI Organization, 2020, pp. 768–777.
[47] A. Artale, R. Kontchakov, A. Kovtunova, V. Ryzhikov, F. Wolter, M. Zakharyaschev, Ontology-mediated query answering over temporal data: a survey (invited talk), in: Proc. of the 24th International Symposium on Temporal Representation and Reasoning, TIME 2017, in: LIPIcs, vol. 90, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2017, pp. 1:1–1:37.
[48] A. Das, S.M. Gandhi, C. Zaniolo, ASTRO: a Datalog system for advanced stream reasoning, in: Proc. 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, ACM, 2018, pp. 1863–1866.
[49] H. Beck, B. Bierbaumer, M. Dao-Tran, T. Eiter, H. Hellwagner, K. Schekotihin, Rule-based stream reasoning for intelligent administration of content-centric networks, in: Proc. 15th European Conference on Logics in Artificial Intelligence, JELIA 2016, in: LNCS, vol. 10021, Springer, 2016, pp. 522–528.
[50] H.R. Bazoobandi, H. Beck, J. Urbani, Expressive stream reasoning with Laser, in: Proc. 16th International Semantic Web Conference, ISWC 2017, Part I, in: LNCS, vol. 10587, Springer, 2017, pp. 87–103.
[51] H. Beck, T. Eiter, C. Folie, Ticker: a system for incremental ASP-based stream reasoning, Theory Pract. Log. Program. 17 (2017) 744–763.
[52] H. Beck, M. Dao-Tran, T. Eiter, Answer update for rule-based stream reasoning, in: Proc. 24th International Joint Conference on Artificial Intelligence, IJCAI 2015, AAAI Press, 2015, pp. 2741–2747.
[53] H. Beck, M. Dao-Tran, T. Eiter, Equivalent stream reasoning programs, in: Proc. 25th International Joint Conference on Artificial Intelligence, IJCAI 2016, IJCAI/AAAI Press, 2016, pp. 929–935.
[54] J. Chomicki, Efficient checking of temporal integrity constraints using bounded history encoding, ACM Trans. Database Syst. 20 (1995) 149–186.
[55] S.S. Cosmadakis, H. Gaifman, P.C. Kanellakis, M.Y. Vardi, Decidable optimization problems for database logic programs (Preliminary report), in: Proc. 20th Annual ACM Symposium on Theory of Computing, STOC 1988, ACM, 1988, pp. 477–490.
[56] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Program. 78 (2009) 293–303.
[57] Y. Falcone, J. Fernandez, L. Mounier, What can you verify and enforce at runtime?, Int. J. Softw. Tools Technol. Transf. 14 (2012) 349–382.
[58] Y. Falcone, J.-C. Fernandez, L. Mounier, Runtime verification of safety-progress properties, in: Runtime Verification, in: LNCS, vol. 5779, Springer, 2009, pp. 40–59.
[59] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to Runtime Verification, Springer International Publishing, Cham, 2018, pp. 1–33.
[60] K. Havelund, Rule-based runtime verification revisited, Int. J. Softw. Tools Technol. Transf. 17 (2015) 143–170.
[61] G.D. Giacomo, Y. Lespérance, F. Patrizi, Bounded situation calculus action theories, Artif. Intell. 237 (2016) 172–203.