FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Expressive Rule-based Stream Reasoning

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Harald Beck
Matrikelnummer 00303187

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistant Prof. Dr. Ezio Bartocci

Diese Dissertation haben begutachtet:

Fredrik Heintz

Sebastian Rudolph

Wien, 1. Oktober 2018

Harald Beck

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Harald Beck
Preßgasse 1-3/10
1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Oktober 2018

Harald Beck

Meinem Vater gewidmet

Danksagung

Kurz nach Abschluss der Diplomarbeit bei Thomas Eiter hatte ich das Glück, von ihm eine Dissertantenstelle im eben bewilligten FWF-Projekt zum Thema „Distributed Heterogeneous Stream Reasoning“ angeboten zu bekommen. Nach der intensiven, lehrreichen Betreuung, die ich schon als Diplomand genießen durfte, gab es für mich keinen Grund, dieses Angebot nicht anzunehmen. Die Zusammenarbeit mit Thomas in diesem Projekt war für mich sehr bereichernd. Neben seiner beeindruckenden fachlichen Expertise konnte ich viel von seiner Arbeitsweise und seiner professionellen Einstellung lernen; besonders seine persönlichen Qualitäten, wie Einsatzbereitschaft, Strukturiertheit, Zuverlässigkeit und nicht zuletzt seine Hilfsbereitschaft, waren vorbildhaft. Es war ein Privileg, diese Arbeit unter seiner Betreuung verfassen zu dürfen. Ich bedanke mich auch herzlich bei meinen Co-Betreuern Stefan Woltran und Ezio Bartocci, die jederzeit verfügbar waren, sowie bei meinen externen Gutachtern Fredrik Heintz und Sebastian Rudolph. Sie alle haben mir für den Feinschliff der Arbeit noch wertvolle Anregungen gegeben.

Diese Dissertation gäbe es auch nicht ohne Minh Dao-Tran, der das zugrundeliegende Forschungsprojekt initiiert hat. Als Post-Doc hat er mit viel Fleiß und Energie für den kontinuierlichen Fortschritt des Projekts gesorgt. Was mich an unserer Zusammenarbeit am meisten fasziniert hat, war die Selbstverständlichkeit, mit der wir uns jederzeit auf die Sicht des anderen eingelassen haben, auch wenn es oft einfacher gewesen wäre, gedanklich entlang der Bahnen der eigenen Ideen zu gleiten. Ein Durchsetzen-Wollen oder Gewinnen-Wollen gab es für uns nicht, stattdessen die ständige Aufmerksamkeit für die Überlegungen des anderen. So konnten wir durch geduldiges Drehen und Wenden unserer unterschiedlichen Ansätze gemeinsame gedankliche Modelle entwickeln. Das war lehrreich, produktiv und eine Freude zugleich.

Im ersten Jahr des Projekts hat auch Michael Fink das Team verstärkt. Besonders eindrucksvoll waren für mich die gemeinsamen Besprechungen zu viert, in denen Michaels ruhige und achtsame Kommunikation zu einer sehr entspannten Arbeitsatmosphäre sowie zu einer geordneten gemeinsamen Herangehensweise beigetragen hat. Dies war von großem Wert für die Entwicklung von LARS, dem zentralen Formalismus dieser Arbeit. Michael hat durch seine beständige, umsichtige Arbeit viel dazu beigetragen, dass das Projekt gut in die Gänge gekommen ist, und dass ich einen guten Start im Forschungsumfeld hatte.

Mein weiterer Dank gilt Juliane Auerböck, Beatrix Buhl und Eva Nedoma für ihre stets freundliche und verlässliche Hilfe in der Organisation. Besonders möchte ich mich bei Eva bedanken, die mich über viele Jahre in allen möglichen und unmöglichen administrativen Belangen perfekt unterstützt hat; zunächst als ich Tutor war, dann Praktikant, Diplomand,

und zuletzt Dissertant. Für ihre Zuverlässigkeit und Hilfsbereitschaft möchte ich mich ebenso herzlich bei den IT-Administratoren Matthias Schlögel und Toni Pisjak bedanken.

Es war mir eine besondere Freude, dass unsere Arbeit auch an anderen Orten Anklang gefunden hat. Die erste Zusammenarbeit hat sich schon Ende des ersten Jahres ergeben, nämlich nach der Vorstellung des ersten Konferenz-Artikels im Jänner 2015, als Konstantin Schekotihin die Nützlichkeit von LARS für ein in Klagenfurt laufendes Forschungsprojekt erkannte. Aus seiner Initiative heraus ist eine produktive Kollaboration entstanden, die wir gemeinsam mit Hermann Hellwagner und Bruno Bierbaumer durchgeführt haben. Hervorheben möchte ich besonders Brunos enormen Einsatz für die Entwicklung der Simulationssoftware, ohne die die Experimente und damit unsere gemeinsamen Publikationen nicht möglich gewesen wären.

Die zweite Kollaboration hat sich auf Initiative von Hamid Bazoobandi ergeben, der im Zuge seines Doktoratsstudiums in Amsterdam zur Entwicklung eines Programms für Datenstromanalysen (*stream reasoning*) mit mir in Kontakt getreten ist. Dank der Einladung zu einem Forschungsaufenthalt durch Jacopo Urbani, seinen Co-Betreuer, konnten wir die ersten Ideen in kurzer Zeit konkretisieren und während Hamids Besuch bei uns in Wien sowie in regelmäßigen Videokonferenzen weiterentwickeln und schließlich auch publizieren. Ich habe den Austausch mit Hamid und Jacopo als besonders wertschätzend erlebt. Darüber hinaus war es für mich sehr erfreulich, dass mit der von Hamid implementierten Software *Laser* noch während der Projektlaufzeit eine zweite Anwendung auf Basis von LARS erarbeitet werden konnte.

Große Freude hat mir auch die Co-Betreuung unterschiedlicher studentischer Arbeiten bereitet, vor allem weil ich das Glück hatte, mit sehr talentierten Studenten arbeiten zu können, die neben dem fachlichen Interesse auch die notwendige Geduld mitbrachten. Dies waren zunächst – branchenüblich nach Nachnamen gereiht – Andreas Humenberger, Andreas Moßburger und Edward Toth. Weiters hat sich eine für mich sehr motivierende Kollaboration mit Christian Folie ergeben, der im Rahmen seines zweiten Masterstudiums auf unser Projekt aufmerksam geworden war. Gemeinsam haben wir *Ticker* entwickelt, eine weitere Software zur Datenstromanalyse, die in der vorliegenden Arbeit genauer präsentiert wird und die es ohne Christian in dieser Form nicht gäbe. In vielen Besprechungen konnten wir die im Vorfeld nur freischwebenden algorithmischen Ideen zur Reife und zur Umsetzung bringen.

Auch außerhalb der eigentlichen Projekttätigkeit war es wichtig, den eigenen Blickwinkel immer wieder zu hinterfragen. Forschung impliziert, dass man sich ständig an Grenzen bewegt, nicht zuletzt an jenen der eigenen Leistungsfähigkeit. Gerade in der theoretischen Arbeit, in der Ergebnisse ausschließlich im Kopf und oft nur über Umwege entstehen, ist wiederkehrender Selbstzweifel gewissermaßen systemimmanent. Ich schätze mich glücklich, dass mir gute Freunde immer wieder dabei geholfen haben, eine neue Perspektive einzunehmen. Insbesondere möchte ich Friedrich Slivovsky danken, der bei unzähligen Mittags- und Kaffeepausen immer ein offenes Ohr für mich hatte, und nicht zuletzt einen wirksamen Humor.

Meinen Eltern, Stephanie und Franz Beck, danke ich für den Rückhalt und das Vertrauen, das sie mir immer geschenkt haben. Besonders dankbar bin ich ihnen dafür,

dass sie mir die ganze Kindheit und Jugend hindurch die Möglichkeit gegeben haben, verschiedensten *eigenen* Interessen nachzugehen, und so meine Selbstbestimmtheit gefördert haben. Ohne die von ihnen vermittelte Sicherheit, mich an meinen eigenen Sichtweisen orientieren zu können, wäre es mir wahrscheinlich weder möglich gewesen, den Wunsch einer wissenschaftlichen Vertiefung zu entwickeln noch diesem nachzugehen; zu laut wären sonst die Störgeräusche gesellschaftlicher Trampelpfade gewesen.

Was die spezifische fachliche Ausrichtung betrifft, sehe ich den Ausgangspunkt auch in der Kindheit; nämlich in der Prägung durch meinen Vater, dem ich diese Dissertation widme. Ich erinnere mich – wenn auch nur skizzenhaft – daran, dass er viele meiner Fragen nicht direkt beantwortet hat, sondern stattdessen oft etwas Allgemeineres erklärt hat, so dass ich mir die Antwort selbst geben konnte. Zusätzlich habe ich dadurch auch Freude am Verstehen selbst entwickelt und gelernt, gedanklich zurückzusteigen, Erklärungen zu suchen, zu analysieren, zu abstrahieren, Analogien zu erkennen und letztlich die *Form* geistiger Inhalte zum Gegenstand der Betrachtung zu machen; insbesondere die Form folgerichtigen Denkens, also die Logik.

Abschließend gilt mein größter Dank Mirjam Moser. Gerade für die ereignisreichen letzten Jahre, in denen diese Arbeit entstanden ist, würden ein paar Zeilen nicht ausreichen. So sehr ich auch Freude am Ringen um treffende Formulierungen entwickelt habe; hier muss ich mir ein weiteres Mal meine Grenzen eingestehen.

Kurzfassung

Die zunehmende Verfügbarkeit und Bedeutung von Datenströmen hat zu zahlreichen Entwicklungen in der Informationsverarbeitung geführt, die über klassiche Datenbanksysteme hinausgehen. Moderne Werkzeuge zur Datenstromverarbeitung bieten verschiedenartige Lösungen zur Evaluierung kontinuierlich strömender Informationen an, üblicherweise mit einem Fokus auf fehlertolerante, verteilte Architekturen für hochperformante Auswertungen. Diese Systeme stellen typischerweise Abfragesprachen oder Programmierschnittstellen bereit, die jene für statische Daten erweitern. Allerdings fehlt es weitgehend an expliziten, Modell-basierten Semantiken, was den Vergleich verschiedener Ansätze bzw. deren formale Analyse erschwert. Aufgrund mangelnder theoretischer Grundlagen kommen diese auch weniger als Problemelösetechniken in der Künstlichen Intelligenz infrage.

Logisches Schließen auf Basis deklarativer Spezifikationen ist ein zentrales Forschungsgebiet innerhalb der Wissensrepräsentation. Dem Umgang mit strömenden Daten widmen sich in diesem Bereich bisher aber noch relativ wenige Arbeiten. Insbesondere wurden Datenströme in regelbasierten Systemen wie der Antwortmengenprogrammierung (*answer set programming*, ASP) erst kürzlich in Betracht gezogen. Im speziellen fehlt es hier noch an expliziten Steuerungselementen wie etwa *window* Mechanismen, die in der Datenstromverarbeitung eine zentrale Rolle spielen.

Anknüpfend an den Mangel formaler Grundlagen für das logische Schließen in der Datenstromverarbeitung wird das LARS Framework vorgestellt. LARS ist ein Akronym für *Logic-based Framework for Analytic Reasoning over Streams*, also ein logik-basiertes Framework für analytisches (logisches) Schließen über Datenströme. LARS *Formeln* erweitern aussagenlogische Formeln mit generischen *window* Operatoren zur Auswahl aktueller Daten, sowie mit Modalitäten um die zeitliche Dimension in Schlussfolgerungen zu steuern. Darauf aufbauend werden LARS *Programme* definiert, die als Erweiterung von ASP für Datenströme betrachtet werden können. Daraus ergibt sich eine theoretische Grundlage für *stream reasoning*, also das logische Schließen über Datenströme, die sich zur formalen Analyse eignet. Weiters ist LARS selbst eine ausdrucksstarke, regelbasierte Sprache für stream reasoning. Der neue Formalismus wird untersucht und mit ausgewählten Methoden unterschiedlicher Forschungszweige verglichen. Im Hinblick auf die Optimierung von LARS Programmen werden dann Äquivalenzbegriffe vorgestellt und charakterisiert, die sich an jenen der ASP Forschung anlehnen.

Danach widmet sich die Arbeit dem Zielkonflikt zwischen Datendurchsatz und Ausdrucksstärke durch die Entwicklung von Techniken zur inkrementellen Auswertung für das praktische *plain LARS* Fragment. Dazu wird werden *Truth Maintenance Systeme*

formalisiert und erweitert, mittels derer Modelle logischer Programme aktualisiert werden können. Es wird eine Kodierung von plain LARS in ASP entwickelt, die inkrementell aktualisiert wird wenn neue Daten einströmen, bzw. nach Ablauf einzelner Zeitpunkte. Dadurch ergibt sich eine inkrementelle Auswertungsmethode für plain LARS (mit *sliding windows*). Die statische wie die inkrementelle Kodierung ist jeweils in einem Evaluierungsmodus in *Ticker* implementiert, einer prototypischen Software für stream reasoning, die hier samt einer empirischen Evaluierung vorgestellt wird.

Zusammenfassend präsentiert diese Arbeit Beiträge zur Theorie und Praxis logikorientierter Verarbeitung von Datenströmen mittels einer ausdrucksstarken, regelbasierten Sprache. Das vorgestellte LARS Framework wird untersucht und verwendet um neue Methoden und Systeme bereitzustellen, die sich für Problemlösetechniken in der Künstlichen Intelligenz eignen.

Abstract

The increasing availability and importance of streaming data has led to many technical advancements in information processing beyond classical databases. Modern stream processing tools offer various solutions for evaluating continuously streaming data with a focus on fault-tolerant, distributed architectures for high-performance computing. These systems typically come with query languages or programming interfaces that extend those for static data. However, model-based semantics are rarely given, which complicates the comparison and formal analysis of different approaches. The lack of theoretical underpinning makes them also less suitable for problem solving in Artificial Intelligence.

Logical reasoning based on declarative specifications is a core research theme in Knowledge Representation and Reasoning, yet streams have attracted little attention so far. In particular, streams have been considered in rule-based approaches like Answer Set Programming (ASP) only recently. So far, they do not provide explicit controls for streams such as *window* mechanisms, which play a central role in stream processing.

To address the lack of formal grounds for *reasoning* in stream processing, we develop LARS, a Logic-based Framework for Analytic Reasoning over Streams. LARS *formulas* extend propositional logic with generic window operators to select recent data, and with modalities to control the temporal dimension of reasoning. On top of this, we define LARS *programs* which can be seen as an extension of ASP for streams. We thus obtain a theoretical stream reasoning framework suitable for formal analysis, as well as an expressive rule-based stream reasoning language by LARS itself. We study this formalism and relate it to selected methods from different lines of research. Towards optimizations of LARS programs, we introduce and characterize notions of equivalence that are in line with previous research for ASP. Furthermore, we tackle the trade-off between data throughput and expressiveness by developing incremental reasoning techniques for the practical *plain LARS* fragment. We formalize and extend Truth Maintenance Systems for updating models of logic programs and give an encoding from plain LARS to ASP that can be incrementally adjusted in response to changing data or passage of time. In this way, we obtain an incremental model update procedure for plain LARS with sliding windows. The static and incremental encodings are implemented in two reasoning modes of the *Ticker* engine, a prototypical stream reasoning system that we present along with an empirical evaluation.

In summary, this work contributes to the theory and practice of expressive rule-based stream reasoning. The proposed LARS framework is studied and utilized as basis for developing techniques and systems geared for problem solving in Artificial Intelligence.

Contents

CHAPTER 1

Introduction

The increasing amount and availability of data streams from sensors, networks, mobile devices, etc., has led to a shift in information processing. Querying a traditional database can be seen as manually pulling information from records of data that do not change, unless explicit update operations are performed. By contrast, in many applications with streaming data, that emerged during the last two decades, information is continuously changing and often pushed automatically to the user. Smart phones, for instance, immediately indicate when a new text message was received, when a system update becomes available for installation, or when the battery level becomes too low. As further examples, consider server logs that document the interactions of users with a web service, temperature signals from weather stations, or transactions of financial institutions.

Processing continuously streaming information usually requires techniques in addition to preexisting tools for databases. If storing data is necessary at all, the question arises which data to keep in the first place, and for how long. Simply recording data fast enough can be challenging when it is arriving at very high frequency. On the semantic side, it is not always clear what the desired result of a query should be if data is changing while being processed. Next, due to the inherent association of data tuples with time, new kinds of queries emerge conceptually, and query languages and processing tools need to take this into account. Many computational tasks that involve streams need evaluation combined with static data, which further complicates software architectures and processing pipelines. Furthermore, data streams impose new demands on algorithms, in particular for query evaluation. If computation takes too long, the obtained results may be already outdated once delivered, and thus irrelevant or even wrong in the light of new, unprocessed data. Moreover, in case of large volumes of data and high frequency updates, low-level processing techniques must ensure sufficient throughput to ensure that further computation is not delayed or distorted by input bottlenecks. In particular, immediate notifications are important in many real-time monitoring use cases.

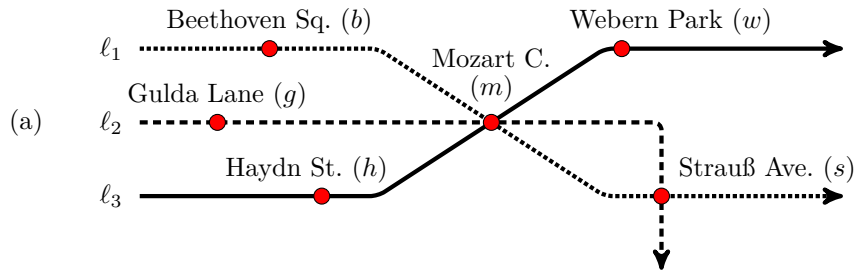


Figure 1.1: Transportation map

Example 1 Consider a public transport domain, where the current location of trams and buses shall be visualized to a user, together with a list of expected arrival times at any selected station. Figure 1.1 illustrates a small map with stops (red dots) along three transportation lines (routes) ℓ_1, ℓ_2 and ℓ_3. Consider Karl travelling with his baby on line ℓ_3. He is currently at Haydn Street (h) and wants to go to Strauß Avenue (s), so he has different options to continue at Mozart Circus (m). Depending on which tram comes first, he might take either line ℓ_1 or ℓ_2. Assuming that trams from different lines depart from physically separated locations, he needs to make a decision where to go, so knowing expected arrivals upfront is advantageous. ∎

During the last three decades, a lot of effort has been put into the development of techniques for dealing with continuously changing information. We now give an overview; a more detailled account of the state of the art will be given in Section 2.2.

1.1 Background

In data management, increasing demand for dealing with streaming data during the 1980s has lead to extensions of classical databases, leading to *active databases* [Day88], *continuous queries* [TGNO92, BW01] and then a research focus on *stream processing* [Ste97, CDTW00, BW01, CÇC+02, CCD+03, ABB+03, AAB+05, ABW06]. Two common themes can be identified in this area. One is the often seen approach of extending SQL or other query languages with streaming capabilities. Towards a uniform treatment of static and dynamic data, many systems employ so-called *window* mechanisms that select only recent portions of the data stream, drop the rest, and view the obtained data as relations. Different selection criteria have been proposed, usually based on time or counting tuples. These windows can move in different ways, e.g., in a *sliding* fashion, or *tumbling*, partitioning the timeline into intervals of equal lengths.

Example 2 (cont'd) Consider a monitoring system for vehicle appearances in the scenario of Example 1. When a tram or a bus reaches a station, the system is notified by a continuous stream of events. Using a discrete, linear timeline, we may assume, for instance, a tram with identifier a_1 arriving at Beethoven Square (b) at time (e.g. minute) 36. The tram sends a notification in the form of predicate $tram(a_1, b)$. Assume

now that we observe at time 44 the event $tram(a_1, m)$. Then, for planning purposes the former event may be outdated and irrelevant. One way to select recent events is to use a time-based window, e.g., a sliding window covering a fixed amount of previous time points, e.g., the last 5 minutes. More suitable in this scenario is to select always the last appearance of each vehicle, which amounts to selecting tuples by counting. ∎

A second research theme in stream processing concerns evaluation techniques. When data is continuously changing, computing query answers or database views repeatedly from scratch is not practical. Accordingly, algorithms and systems have been developed that incrementally update previous results, e.g. [Doy79, dK86, CW91, GMS93, GHM+07, VSM05, GUW09, BBC+10b, MNPH15, HMH18]. Since the early works in stream processing, which typically extended databases with capabilities for handling streams, more and more special purpose streaming tools have been developed, often as extensions of batch-processing systems. Modern stream processing systems such as those maintained by the Apache Software Foundation (e.g. *Apache Spark* [ZCF+10, ZXW+16] and *Apache Flink* [CKE+15]) provide robust, distributed, high-performance architectures for flexible processing of huge volumes of data; they also provide window mechanisms.

Another approach to tackle streaming data is offered by *Complex Event Processing* (CEP) systems [Luc01], which are event-based by design. The general task of a CEP is to efficiently derive complex (or composite) events based on input events by means of pattern matching. In contrast to developments in the database area, CEP systems directly target the continuous processing rather than extending querying techniques for stored and streaming information. Nevertheless, also extensions for SQL have been proposed; e.g. in [BDG+07, DGP+07]. Since a duration is the natural extent of a complex event, expressions of temporal intervals are often provided [BGAH07, CM10]. A notable such system is ETALIS [AFR+10, ARFS12], in which the composition of intervals can be declaratively specified in form of rules.

In the Semantic Web area, multiple works extend the SPARQL query language [HSP13] for streaming in a similar way as CQL, the continuous query language [ABW03], extends SQL. C-SPARQL [BBC+09, BBC+10a], CQELS [PDPH11] and SPARQLStream [CCG10] all provide access to streaming data with window operators. Although their syntax is similar, the semantics of these languages differ, as well as their architectures and modes of operation. A different solution is offered by EP-SPARQL, which adds interval expressions from ETALIS to SPARQL querying. Some works consider reasoning based on ontological entailments, such as STARQL [ÖMN15] and the truth-maintenance techniques in [RP11] for ontology-based data access (OBDA). Many Semantic Web approaches seem to be driven by ad-hoc extensions and prototypes. While the expected behaviour of available systems and the intended meaning of query languages is often intuitively clear, their exact semantics and differences often are not. Accordingly, attempts to unify different SPARQL extensions on formal grounds have been presented in [DVCC14, DDC+16].

Incorporating the temporal dimension in logical reasoning is considered in the study of temporal logics such as *Linear Time Logic* (LTL) [Pnu77], *Computational Tree Logic* (CTL) [CE81], *Metric Temporal Logic* (MTL) [Koy90, AH93], or *Signal Temporal Logic* (STL) [MN04]. Since Pnueli's seminal work [Pnu77], which introduced LTL for verification,

a multitude of temporal logics and techniques have been developed, in particular for model checking [CGP99] and more recently, runtime verification [LS09, BFFR18]. Common to most of these logics is a focus on the evaluation of infinite sequences of states that need to satisfy a property expressed as formula. A variant of MTL based on the *Temporal Action Logic* (TAL) [DK08] was introduced for execution monitoring in the *DyKnow* architecture [HD04, HKD10a] that deals with abstracting heterogeneous low-level sensor input towards high-level conceptual entities for reasoning over streaming data in real-time.

Stream Reasoning

In the cross-section of the developments sketched above, *stream reasoning* emerged as research field to address high-level reasoning over streaming data [Hei09, HKD10c, DCvF09, MDEF17, DDvB17]. The variety of perspectives and approaches yields a landscape of different aspects of logic-oriented stream processing and various systems that usually have an informal or an operational semantics; model-based semantics, as e.g. in most temporal logics, are typically not provided. Standard temporal logics, on the other hand, do not provide explicit window mechanisms, which play a central role in stream processing. In other words, a formal underpinning for the study and comparison of different stream reasoning approaches seems to be missing, and a language to express their declarative semantics. In fact, a lack of theoretical foundations for stream reasoning has been observed earlier [DCvF09, Zan12].

Reasoning on formal grounds is the core research theme in the area of Knowledge Representation and Reasoning (KR&R); yet only few works so far have considered streaming data. Besides work in OBDA [RP11, BKK$^+$17, AKK$^+$17], rule-based systems have increasingly been attracting attention. *Streamlog* [Zan12] considers an explicit time dimension in a fragment of Datalog with stratified negation, and the recent *Temporal Datalog* [RKG$^+$18] is defined as a formal language for the study of reasoning tasks on streaming data. Both languages are explicitly proposed as formal foundations for stream reasoning; however, window mechanisms are not included. Multiple works study the use of Answer Set Programming (ASP) [EIK09, BET11] for streaming data. Besides Datalog, ASP is arguably the most studied rule-based language in KR&R. It provides a natural specification language that is easily readable and equipped with a model-based semantics [GL88]. Thus, it is a useful tool for both theoretical and practical work that deals with tasks involving logical reasoning [BET11]. We will review ASP in Section 2.1.2.

Example 3 (cont'd) Assume for the public transport domain that streaming events are reflected as predicates with a timestamp as last argument. Appearance of tram a_1 at station b at time 36 can then be stated by the predicate $\mathit{tram\text{-}ev}(a_1, b, 36)$. Assume further that we have background knowledge for linking a vehicle's Id to its transportation line L in form of predicates $\mathit{line}(\mathit{Id}, L)$, and relations of the form $\mathit{plan}(L, X, Y, D)$ that reflect planned travel durations D between consecutive stations X and Y. We can then infer expected arrival times at later stations using the following rule:

$$\mathit{exp}(\mathit{Id}, Y, T') \leftarrow \mathit{tram\text{-}ev}(\mathit{Id}, X, T),\ \mathit{line}(\mathit{Id}, L),\ \mathit{plan}(L, X, Y, D),\ T' = T + D \quad (1.1)$$

This rule expresses the following: if there is a tram event (*tram-ev*) for station X at time T such that the tram identifier Id is associated with line L where D is the planned travel time from X to the next station Y, then we expect this tram at station Y at time $T' = T + D$. Assuming database entries $line(a_1, \ell_1)$ and $plan(\ell_1, b, m, 8)$, we derive $exp(a_1, m, 44)$, i.e., that tram a_1 can be expected at station m at time point 44. Given additional expected arrival times for line ℓ_2 before arriving at m via line ℓ_3, Karl can make the decision how to continue his journey. Of course, his decision can be automated by further rules. ∎

Towards using ASP on streaming data, a prototype solution was presented in [DLL11] that made used of the *dlvhex* solver [EIST06]. Next, *StreamRule* [MAPH13] offers a hybrid architecture using CQELS for stream processing and ASP for reasoning. In recent years, the state-of-the-art ASP solver Clingo [GKKS14] has been gradually extended towards stream reasoning capabilities by means of language extensions and controls for the solving process [GKK+08, GGKS11, GGK+12]. In particular, to avoid unnecessary re-evaluations, Clingo offers multi-shot solving capabilities [GKOS15, GKKS17]. In contrast to most works in stream processing and based on SQL or SPARQL, which typically offer window operators but lack model-based semantics, emerging KR&R methods are formally defined but lack explicit window mechanisms.

Example 4 (cont'd) In Example 3, we considered tram events in form of predicates $tram\text{-}ev(Id, X, T)$, where T is the arrival time of tram Id at station X. If we do not delete historic events, Rule (1.1) will continue to infer expected arrival times for all past tram appearances. The naive rule formalization implicitly assumes that only currently relevant events are fed as input to the engine carrying out the rule-based reasoning process. This selection amounts to a window obtained in a pre-processing step. ∎

Example 4 indicates some problems that arise when no explicit streaming controls are available in the reasoning language. While it is technically possible to express most relevant window operators with rules, the resulting encoding is less readable; yet the ability to write concise, understandable specifications is a major reason for using rule-based systems in practice. Even assuming a semantically sufficient encoding, an implementation must be able to drop irrelevant historic data due to storage limitations, respectively for efficiency reasons. Not being able to directly express which data is relevant undermines not only the declarative nature of rule-based systems but also the possibility to automatically handle expiration of historic events. Furthermore, a strict separation of processing and reasoning by first obtaining data snapshots from streams and then reasoning over selected data limits expressiveness. In particular, it excludes windows over inferred information and prohibits the reasoning process itself from controlling window application.

In light of the above considerations it seems natural to extend established KR&R techniques with explicit controls for streams. We use ASP as the formalism of choice due to its rich semantic features and its suitability for both theoretical and practical use [BET11]. This gives us a concrete entry point to add reasoning features to stream processing on formal grounds.

1.2 Problem Statement

The developments sketched above yield a landscape of different approaches to stream reasoning. While they often share conceptual ideas, their exact commonalities and differences remain unclear without a common theoretical underpinning, in which their declarative semantics may be expressed, analyzed, and compared. The *lack of theory* for stream reasoning has been observed already in [DCvF09]. In particular, the authors propose that a theoretical framework for stream reasoning must combine two aspects: first, it has to serve as a basis for explicit formal semantics, and second, it must account for high throughput, i.e., frequency and volume of data.

From a theoretical perspective, the *trade-off between expressiveness and scalability* is evident. In particular, some portions of the data might have a higher frequency and volume than others, and the potential difficulty of reasoning does not imply that all operations are highly complex. A theoretical foundation for stream reasoning should thus aim at covering the entire spectrum, i.e., provide *flexible* means to express both semantically trivial real-time computations as well as complex reasoning tasks that are necessarily slower; possibly within the same query or program. This can be achieved by a *modular* system, where the mechanisms to handle streams (like window operators for deliberate information loss) can be used in a *generic and compositional* way.

Regarding the expressiveness, a rich framework should encompass *advanced reasoning features* as available in KR&R. In particular, this includes intensional data definitions and thus the ability to abstract from (extensional) input data. To this end, *rule-based* approaches as Datalog [CGT90] are a natural approach for *fully declarative*, logic-oriented and logic-based data access. Moreover, *nonmonotonicity* is of special relevance in stream reasoning, i.e., the property that previous conclusions might have to be withdrawn due to later arrival of previously missing information (e.g., in case of defaults) or contrary evidence (in case of contradictions). Next, *model generation* as in SAT solving and Answer Set Programming (ASP) are useful when tackling domains which permit multiple solutions. Such features and according techniques have been studied rarely for streams.

Towards practical applicability of stream reasoning systems, another important issue arises. While the continuous development of query results may be viewed as sequence of models over consecutive time points, repeatedly computing models from scratch at every time point is often impractical. Instead, there is an obvious demand to investigate *incremental reasoning*, i.e., incremental adaptation of previous results in response to changing information. In particular, information changes when windows contain new data, but also when historic data expires. Notably, incremental reasoning is non-trivial when dealing with nonmonotonicity, which may not only come from expressive rule-based semantics, but also from window operators, as we will investigate.

The final yardstick of envisaged studies is a practical realization that proves the usefulness of the developed methods. To this end, central concepts and techniques should result in a *prototypical software* for expressive rule-based stream reasoning.

	Querying	AI Problem Solving
Database	SQL	ASP
Streams	CQL	**?**

Table 1.1: Gap the thesis aims to fill

Objectives of the Thesis

Table 1.1 summarizes the aim of this thesis at the highest level: to fill a gap in stream reasoning in the spirit of ASP. Many modern stream processing systems follow the conceptual approach of CQL [ABW03] to extend existing tools for static data (like SQL) by means of additional access to recent data from streams. This influential idea has proven to be useful not only as a means for coping with data volumes but also for expressing practical use cases. However, streams and window mechanisms have been considered less in declarative reasoning systems, which are a suitable choice to express formal semantics. In turn, fully declarative semantics are often lacking in state-of-the-art stream processing/reasoning systems and languages.

Due to these observations, the thesis aims at providing:

(i) a theoretical framework for expressive stream reasoning with window mechanisms;

(ii) techniques for incremental reasoning within that framework; and based on that

(iii) a prototypical rule-based stream reasoning engine.

The central challenge lies in the inherent trade-off between throughput, i.e., data volume and frequency, and semantic expressiveness. That is to say, a conflict arises when rich reasoning features are needed on large quantities of data per time unit. Traditionally, stream processing focuses on high throughput, targetting low-level processing such as filtering and aggregation. On the other hand, reasoning techniques on static data in KR&R tend to emphasize high-level, complex semantics which typically come at a higher computational cost. When data is continuously changing, new means are needed to address the issue of scalability of complex reasoning.

1.3 Contributions and Thesis Structure

We now summarize the main contributions, addressing the research objectives above.

(i) Towards a theoretical foundation for expressive stream reasoning, we present LARS, a Logic-based framework for Analytical Reasoning over Streams. It builds on a simple stream formalization and specific controls for dealing with streams, i.e., generic window operators to obtain substreams, and modalities for handling the temporal dimension of data. Reasoning in LARS is relative to a stream at a time point, and either based on formulas or programs: in addition to propositional connectives, LARS formulas provide the mentioned stream controls, and LARS programs employ these formulas to extend ASP for streams.

We examine semantic properties and the complexity of reasoning in LARS, as well as its relation to LTL, CQL, and ETALIS. We also explore the use of LARS as specification language, respectively as analytical framework. Furthermore, we present characterizations of different notions of program equivalence.

(ii) Towards incremental evaluation of programs, we present an encoding from LARS to ASP and then show how this encoding can be updated incrementally with the development of an input stream. We extend Truth Maintenance Systems as specific procedure to obtain an updated ASP model due to an updated ASP program and thus obtain an incremental reasoning procedure for LARS with sliding windows; more specifically for the practical *plain LARS* fragment. We also summarize two further incremental reasoning techniques developed in the course of this work.

(iii) Furthermore, we present *Ticker*, a prototypical rule-based stream reasoning engine for evaluating plain LARS programs with sliding windows. Ticker provides two reasoning modes: one repeatedly calls the ASP solver Clingo on the static ASP encoding, and the other one is based on the developed incremental evaluation technique. We assess Ticker empirically, showing in particular under which circumstances incremental reasoning is beneficial.

We are now going to give a more detailled overview of these contributions in the course of presenting the structure of the thesis. We also briefly mention the publications on which the following chapters are based; references to relevant publications will also be stated explicitly at the beginnings of Chapters 3-7.

Chapter 2 - State of the Art. Section 1.1 already sketched the state of the art in stream processing and reasoning. We will expand on this and also give a short introduction to rule-based programming, followed by a review of ASP, which serves as a building block for this work.

Chapter 3 - LARS: A Logic-based Framework for Analytic Reasoning over Streams. In Section 3.1, we start by formalizing *streams* and generic *windows functions* to obtain substreams, i.e., typically recent portions of data. As specific instances, we then introduce generalizations of prominent time-based and tuple-based window functions, as well as partition-based windows and filter windows. The LARS framework, presented in Section 3.2, then extends propositional *formulas* which are evaluated on streams at time points. To handle streams explicitly and in a compositional way, window operators $\boxplus^w$ are the central syntactic ingredient, which may employ any window function w to limit subsequent reasoning on returned substreams. Dually, a reset operator $\triangleright$ serves to re-access the original input stream. Within any considered stream, modalities are available to control the temporal dimension of atoms and formulas, i.e., whether something holds at some time point ($\Diamond$), always ($\Box$), or at a specific time point t ($@_t$). On top of this, LARS *programs* present an extension of Answer Set Programming (ASP) for streams, where in the usual rule syntax $\alpha \leftarrow \beta_1, \ldots, \beta_n$ arbitrary LARS formulas (α and all β_i) are allowed. By extending ASP, we obtain the desired semantic features. These are, in particular, availability of intensional data, nonmonotonic inference, and generation of supported,

minimal models. The use of LARS as specification language is then exemplified in a case study that deals with research on future internet architectures, which we summarize. In Section 3.3 we then analyze the complexity of model checking and satisfiability in LARS, obtaining PSpace-completeness in general. We then show how practically relevant syntactic and semantic restrictions, i.e., bounded window nesting and so-called sparse windows, yield fragments that are not harder than ASP w.r.t. the worst-case complexity.

LARS was developed incrementally and presented in two workshop papers [BDTEF14b, BDTEF14a], followed by a conference paper [BDEF15], a technical report [BDTE17] and a journal version [BDTE18]. The case study in Section 3.2.4 is a summary of [BBD$^+$16] and [BBD$^+$17].

Chapter 4 - Relating LARS to other Formalisms. While Chapter 2 sets up the wider context of related work, this chapter selects representative formalisms from different research fields and studies their relation to LARS in more detail. After having shown in Chapter 3 that LARS programs extend ASP, we introduce in Section 4.1 a practical fragment called *plain LARS*, which will be considered in subsequent chapters. Plain LARS can be seen as a lightweight extension of normal logic programs for streams using the novel language elements. Section 4.2 investigates the relationship between stream reasoning (in terms of LARS) and temporal reasoning in terms of LTL. We show that LARS with sliding time-based windows can be encoded into LTL, which has no explicit window mechanisms. Next, we show in Section 4.3 how the core of CQL can be captured in LARS; in particular, the abstraction step from streams to relations. Section 4.4 investigates the possibility to express intervals in LARS, as used in the complex event processing language ETALIS, which is also rule-based. Next, Section 4.5 notes that LARS, used as analytic framework, may also serve to capture the semantic difference that arises from pull- vs. push-based querying, as exemplified by Semantic Web reasoning with C-SPARQL and CQELS, respectively. Finally, the discussion in Section 4.6 highlights specific semantic features of LARS relevant for practical applications.

Plain LARS was first introduced in [BDE16]; the results on LTL and the final discussion are from [BDTE18]. The results on ETALIS were stated in [BDEF15], which also investigated CQL, albeit in less detail than in [BDTE18]. The comparison of C-SPARQL and CQELS is a summary of work published in [DBE15b] and [DBE15a].

Chapter 5 - Semantic Characterizations of Equivalent LARS Programs. Towards optimizations of LARS programs, we study notions of equivalence between LARS programs. In Section 5.1 we extend notions of strong equivalence (SE) [LPV01], uniform equivalence (UE) [EF03] and relativized uniform equivalence (RUE) [Wol04] from research in ASP. We then define in Section 5.2 a logic called bi-LARS that will serve to characterize answer streams (which define the LARS semantics) for a large fragment of LARS in Section 5.3, and based on that the considered equivalence notions in Section 5.4. We do so by lifting the model-theoretic characterizations of SE/UE/RUE from the ASP literature, which, however, cannot be done in a straightforward way due to expressivity provided by the generic window operators. Restricting them allows us to obtain another characterization in Section 5.5 based on a variant of bi-LARS that extends the logic of

Here-and-There [Hey30], thus establishing a link to equilibrium logic [Pea06, LPV01]. We then give in Section 5.6 the complexity for the considered equivalence relations that are typically not worse than for ordinary ASP. Finally, we discuss the results in Section 5.7.

The work of this chapter is based on the publication [BDE16].

Chapter 6 - Incremental Reasoning for Plain LARS Programs. In this chapter we develop incremental reasoning techniques within the LARS framework. After presenting the high-level idea in Section 6.1, we review Justification-based Truth Maintenance Systems (JTMS) [Doy79] in Section 6.2. In contrast to previous accounts, we formalize the core of JTMS in terms of logic programs and thus obtain a procedure for updating an answer set of a logic program after the addition of a new rule; we extend this procedure for rule removal. We then present in Section 6.3 an encoding from plain LARS to ASP that allows us to compute the answer streams at a given time point by the answer sets of the encoding. In Section 6.4, we show how a slight adaptation of this translation leads to an incremental encoding that can be updated when new data is streaming in or when time passes by. With the extended JTMS in place, this yields an incremental update procedure for plain LARS, where we consider sliding time-based and sliding tuple-based windows. In Section 6.5 we then review further incremental reasoning techniques based on extending JTMS directly for plain LARS (in Section 6.5.1), respectively by incorporating the temporal dimension in semi-naive evaluation as in Datalog for fragments of plain LARS that ensure unique models (in Section 6.5.2). In Section 6.6 we discuss the presented techniques and mention related work.

Section 6.2 is for the largest part based on the technical report [Bec17], which was released as supplementary material to the publication [BEF17]. The latter is the basis of Sections 6.3-6.4. Section 6.5.1 summarizes the work in [BDE15] and Section 6.5.2 presents the key ideas of [BBU17], which introduces the *Laser* engine.

Chapter 7 - The Ticker Engine. The encodings of Sections 6.3-6.4 form the basis for algorithms to evaluate plain LARS programs with sliding windows. We provide a prototypical stream reasoning engine called *Ticker* that implements these algorithms in form of two reasoning modes. In Section 7.1 we introduce Ticker programs, which assign time units to LARS time points, and runtime options of the engine. We then adapt and improve in Section 7.2 the incremental encoding of Section 6.4 towards a feasible realization. Implementation details are then given in Section 7.3, which centers around the architecture of the two reasoning modes for solving based on repeated one-shot solving with Clingo, and the developed incremental technique, respectively. Section 7.4 then presents a detailled empirical evaluation, where we see in particular when incremental reasoning is beneficial. Finally, Section 7.5 discusses some technical details of the presented implementation and points towards future research issues.

This chapter is a significant extension of the practical part in [BEF17]. Ticker was also highlighted in the brief note in [BDEF18] (as well as Laser).

Chapter 8 - Conclusion. To conclude, we give in Section 8.1 a summary of this thesis from the perspective of the above problem statement. Finally, we mention in Section 8.2 some ideas for future research.

CHAPTER 2

State of the Art

In this chapter, we review the technical background available prior to this work, and some recent developments. Parts of this literature review have appeared in publications that we mention explicitly at the beginnings of forthcoming chapters. In particular, Section 2.2 extends the discussion of related work in [BDTE18].

2.1 Rule-based Programming

We give a brief, informal introduction to rule-based programming for readers that are familiar with stream processing but not with rule languages. Then, we formally introduce Answer Set Programming (ASP), a central formalism for this work.

2.1.1 Declarative Programming with Rules

One way of distinguishing programming languages is in the way they direct computation. Algorithms, for instance, are typically specified in an *imperative* way, i.e., by a step-by-step instruction on *how* to manipulate data structures, using variables, assignments, etc. By contrast, *declarative* approaches seek to abstract away from such instructions how computation shall be carried out in detail and instead provide terms for higher-level specifications. For instance, SQL, arguably the most prominent query language for (relational) databases, provides constructs such as `SELECT`, `FROM`, `WHERE`, `GROUP BY`, etc., that name predefined data transformations. The query language then allows one to combine these transformations at a more conceptual level to specify *what* shall be computed, but not the details of *how* the evaluation is carried out to obtain the results. The operational aspect is delegated to the database itself, not the user writing the query. Similarly, *functional* programming aims at providing higher-level constructs to transform (typically immutable) data structures from one to another and moreover allows one to introduce hierarchies of abstractions.

Rule-based languages such as Prolog [CM94] or Datalog [CGT89] offer approaches to declarative programming that focus particularly on concise descriptions of logical relations. Datalog, for instance, is not even concerned with transforming data structures, it only provides means for describing the logical dependencies between relations. One way to think about a Datalog program is as a logic-based view definition of database relations, where a table p with n columns is schematically represented by a predicate $p(X_1, \ldots, X_n)$; we use upper case letters to denote variables. A join is then expressed by a conjunction of two (or more) predicates, written with a comma. This conjunction forms the *body* B of a rule of form $H \leftarrow B$, which can then express a selection in its *head* H.

For instance, consider in a public transport domain the task of selecting line (route) identifiers that are associated with trams. A corresponding SQL statement could be the following, where `line.L` is the identifier of the line and `line.ID` an identifier of an associated tram.

```
SELECT line.L
FROM tram, line
WHERE tram.ID=line.ID
```

Assuming tables `tram` and `line` both have two columns, where the respective first holds the tram identifier, we can establish the same query with the following Datalog rule r:

$$q(L) \leftarrow \mathit{tram}(\mathit{Id}, X), \mathit{line}(\mathit{Id}, L) \tag{2.1}$$

The above rule can be read as follows: if for some values i, x, ℓ, replacing variables Id, X and L, respectively, there are facts of the form $\mathit{tram}(i, x)$ and $\mathit{line}(i, \ell)$, then we conclude $q(\ell)$. A rule without variables is called *ground*. For instance, let us assume the facts $\{\mathit{tram}(t_1, x_1), \mathit{tram}(t_2, x_2), \mathit{line}(t_2, \ell_2), \mathit{line}(b_1, \ell_3)\}$. The join over variable Id is only possible for the pair of facts $\mathit{tram}(t_2, x_2)$ and $\mathit{line}(t_2, \ell_2)$. This way, we get the ground rule

$$q(\ell_2) \leftarrow \mathit{tram}(t_2, x_2), \mathit{line}(t_2, \ell_2). \tag{2.2}$$

The body of this rule holds due to the assumed data, and we derive $q(\ell_2)$, reflecting that line ℓ_2 is associated with some tram. Note that we could alternatively consider a naive grounding by replacing all variables in Rule (2.1) with all combinations of available values. This grounding would include, for instance, also rule $q(\ell_3) \leftarrow \mathit{tram}(t_1, x_1), \mathit{line}(t_1, \ell_3)$. However, among the rules obtained this way, only Rule (2.2) would fire, and we obtain the same result, i.e., $q(\ell_2)$.

Next, we are interested in lines that are associated with buses but not with trams. This can established by adding following rule:

$$q'(L) \leftarrow \mathit{bus}(\mathit{Id}, Y), \mathit{line}(\mathit{Id}, L), \text{not } q(L) \tag{2.3}$$

Here, "not $q(L)$" is a condition that holds if $q(L)$ cannot be derived. Using in addition to the facts above predicate $\mathit{bus}(b_1, y)$ (corresponding to a row in a table `bus`), we obtain the following additional ground instance:

$$q'(\ell_3) \leftarrow \mathit{bus}(b_1, y), \mathit{line}(b_1, \ell_3), \text{not } q(\ell_3) \tag{2.4}$$

Note that the instantiation of variable L in predicate q under negation has to be carried out due to the positive occurrence in predicate *line*. We reason that, having facts $bus(b_1, y)$ and $line(b_1, \ell_3)$, but no way of deriving $q(\ell_3)$, the body of the latter ground rule holds, so we derive $q'(\ell_3)$. That is to say, line ℓ_3 is associated with some bus, but with no tram according to our data.

The examples so far illustrated already two semantic features of rule-based programming in the spirit of Datalog. First, there is no notion of processing order; in particular, the order of rules does not matter. Second, we can easily compose a chain of derivations by introducing names for intermediate results, such as q in the context of our second example. We note that this is also possible in SQL in principle by means of nesting; however, this way queries quickly become hard to read.

Now consider the following rules, expressing a choice between the mode of transport at a station s; the third rule has an empty body and thus expresses a fact.

$$takeTram(s) \leftarrow station(s), \text{not } takeBus(s) \tag{2.5}$$
$$takeBus(s) \leftarrow station(s), \text{not } takeTram(s) \tag{2.6}$$
$$station(s) \leftarrow \tag{2.7}$$

What is the result of this set of rules? Looking at the first rule, we observe that $station(s)$ holds due to Fact (2.7) but $takeBus(s)$ does not hold, so we infer $takeTram(s)$. Now the body of the second rule does not hold, since "not $takeTram(s)$" is false, so we do not conclude $takeBus(s)$. That is to say, we infer (only) $takeTram(s)$. On the other hand, we could have started with the second rule, and thus obtained instead the derivation $takeBus(s)$. Apparently, there is no obvious solution to a query expressed in this form.

The difficulty arises from the cyclic dependency between the predicates *takeTram* and *takeBus*, which involves negation. There are different ways of dealing with the ambiguity that arises. A naive approach would be to produce a random result as we did in this discussion. Another strategy is to impose an order of processing, e.g., from top to bottom. Next, one could argue that neither $takeTram(s)$ nor $takeBus(s)$ should be inferred because these conclusions are not *well-founded* [VGRS91]. Yet another approach is to consider multiple solutions at the same time, i.e., reflecting that $takeTram(s)$ and $takeBus(s)$ are equally reasonable solutions.

The above observation motivates the formal definition of the syntax, and in particular the semantics of a rule-based language. Considering multiple solutions is the most expressive approach, since we can always explicitly exclude solutions by adding further constraints, and a random choice to naively select a random option can always be established by post-processing. We argue that a logic-oriented specification should be fully declarative, hence a definition of models that takes into account an implicit order of rules is not desirable. We thus select logic programming under the answer set semantics, which we formally review next.

Name	Property
extended	$\neg$ allowed
disjunctive	$k \geq 0$
positive	$n = m$
normal	$k \leq 1$
Horn	$k \leq 1, n = m$
fact	$k = 1, n = m = 0$
constraint	$k = 0$

Table 2.1: Different classes of rules/programs

2.1.2 Answer Set Programming

We now present a formalization of Answer Set Programming (ASP), slightly adapting the one given in [Bec13]. As usual, we first present the syntax, and then the semantics of the language.

Syntax

We distinguish three disjoint sets of symbols: predicates $\mathcal{P}$, constants $\mathcal{C}$, and variables $\mathcal{V}$. For our purposes, we may assume they are all finite. An *atom* with *arity* k is an expression of form $p(t_1, \ldots, t_k)$, where p is a predicate and $t_1, \ldots, t_k$ are *terms*, given by $\mathcal{C} \cup \mathcal{V}$. Atoms are called *propositions* if $k = 0$ and *ground* if they do not contain variables. A *literal* ℓ is an atom a or a (classically) *negated* atom $\neg a$, and a *negation as failure (NAF) literal* is either a literal ℓ, or a *default negated* literal "not ℓ," which evaluates to true if the truth of ℓ cannot be derived.

An *rule* r is an expression of the form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\,, \tag{2.8}$$

where $k, m, n \geq 0$, and all a_i and b_j are literals. By $H(r) = \{a_1, \ldots, a_k\}$ we denote the *head* of r, and the *body* $B(r) = B^+(r) \cup B^-(r)$ is given by the *positive body* $B^+(r) = \{b_1, \ldots, b_m\}$ and the *negative body* $B^-(r) = \{b_{m+1}, \ldots, b_n\}$. Using this notation, we can write a rule of form (2.8) abstractly as

$$H(r) \leftarrow B^+(r), \text{not } B^-(r). \tag{2.9}$$

An *(extended disjunctive) program* P is a finite set of rules. By restricting the rule form, as shown in Table 2.1, we obtain different classes of rules. If all rules of the same program fall into one class, we define that also the program belongs to this class. Most notably, *extended (normal) rules* disallow the use of disjunctive rule heads, and in *normal rules* moreover prohibit classical negation. Furthermore, *facts* are non-disjunctive ground rules with empty bodies. We sometimes omit the symbol $\leftarrow$ in this case. A rule is *ground* if all literals in $H(r) \cup B(r)$ are ground, and a program is *ground* if all its rules are ground.

Example 5 Rules (2.1)-(2.7) are all normal rules. Rule (2.7) is a fact. Moreover, Rule r from (2.5) has the positive body $B^+(r) = \{station(s)\}$, the negative body $B^-(r) = \{takeBus(s)\}$ and the head $H(r) = \{takeTram(s)\}$. The only term appearing in r is constant s, i.e., r is ground. ∎

Semantics

We define the answer set semantics of a logic program P by first considering its *ground instantiation*. We use the set HU_P of constant symbols appearing in P, called the *Herbrand universe*; in case this set is empty we use an arbitrary symbol from $\mathcal{C}$. Next, the set of all ground literals constructible from predicate symbols in P and constant symbols from HU_P defines the *Herbrand base* HB_P. The set $ground(r)$ of *ground instances* of a rule $r \in P$ is obtained by replacing all variables of r with constants from HU_P. The *grounding* of P is then defined by $ground(P) = \bigcup_{r \in P} ground(P)$.

Let P be a ground program. An *interpretation* $I \subseteq HB_P$ is *consistent* if for every atom $a \in HB_P$ it holds that $\{a, \neg a\} \not\subseteq I$. An interpretation I *satisfies* a rule $r \in P$, denoted

$$I \models r, \tag{2.10}$$

if $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$ implies $H(r) \subseteq I$. That is, if all literals of the positive body are in I, and no default negated literals is in I, then a literal of the head as to be in I. Furthermore, an interpretation is a *model* of P, denoted

$$I \models P, \tag{2.11}$$

if $I \models r$ for all $r \in P$. We call model I *minimal*, if every smaller interpretation $J \subset I$ is not a model of P. Notably, every positive normal program has a unique minimal model.

The *Gelfond-Lifschitz reduct* [GL88] of P *relative to* I, denoted P^I, is the positive program obtained when each rule $H(r) \leftarrow B^+(r)$, not $B^-(r)$ in P

(i) is deleted, if $B^-(r) \cap I \neq \emptyset$, and

(ii) replaced by $H(r) \leftarrow B^+(r)$, otherwise.

The first step discards those rules where I contradicts a default negated literal, and the second one removes the negative body from the remaining rules. With this, we define that I is an *answer set* of P, if it is a minimal model of the reduct P^I. We point out that the unique minimal model of a positive normal program P is also its (unique) answer set.

Example 6 (cont'd) We recall the program P given by the rules (2.5)-(2.7), which is ground. To satisfy the Fact (2.7), every model must include $station(s)$. We first consider the reduct P^I for the interpretation $I = \{station(s), takeTram(s)\}$, which looks as follows:

$$takeTram(s) \leftarrow station(s) \tag{2.12}$$

$$station(s) \leftarrow \tag{2.13}$$

The first rule is obtained by removing "not *takeBus*(s)" from Rule (2.5), i.e., step (ii). Rule (2.6) was deleted due to step (i). Clearly, I satisfies both rules and thus is a model of P^I. It is also apparent that it is a minimal model of P^I, i.e., an answer set. The second answer set $I' = \{station(s), takeBus(s)\}$ is similarly obtained.

We now show that P has no further answer sets. The empty interpretation can be excluded since it does not satisfy the fact $station(s) \leftarrow$. Thus, consider the interpretation $S = \{station(s)\}$. This yields the reduct P^S with the following rules:

$$takeTram(s) \leftarrow station(s) \tag{2.14}$$

$$takeBus(s) \leftarrow station(s) \tag{2.15}$$

$$station(s) \leftarrow \tag{2.16}$$

We observe that S is only satisfies the last rule and thus cannot be an answer set. Finally, we consider $M = \{station(s), takeTram(s), takeBus(s)\}$, which is a model of the reduct $P^M = \{station(s) \leftarrow\}$, but it is not a minimal model: also $\{station(s)\} \models P^M$ holds. ∎

The answer set semantics (or stable model semantics) is related to other KR-formalisms. In particular, it can be seen as fragment of Reiter's default logic [Rei80], respectively of a disjunctive extension thereof [GPLT91]. While the Gelfond-Lifschitz reduct is arguably the standard definition for stable models, others have been proposed in the literature [Lif08].

An elegant alternative was presented in [FLP04], based on a different definition of the reduct. Consider first for a rule r of form (2.8) the conjunction $\beta(r) = b_1 \wedge \cdots \wedge b_m \wedge \neg b_{m+1} \wedge \cdots \wedge \neg b_n$. Given an interpretation I for a program P, the *FLP-reduct* is then defined as the program

$$P^I = \{r \in P \mid I \models \beta(r)\}, \tag{2.17}$$

i.e., the subset of rules whose body is satisfied classically by I. Again, an interpretation is called an answer set, if it is a minimal model of the (FLP-)reduct. Importantly, for the class of extended disjunctive programs, the answer sets according to both definitions coincide.

Stratified Negation

We now introduce an important class of programs that permits efficient evaluation, i.e., *stratified programs*. Intuitively, they restrict the use of negation such that no cyclic dependencies through negation occur. We focus here on normal programs. The underlying concept for their definition is the *dependency graph* $D(P) = (V, E^+ \cup E^-)$ of a program P that is constructed as follows. The nodes V are the predicates occurring in P. For every predicate p occurring in the head $H(r)$ of a rule r, we employ an directed edge

(i) $(p, q) \in E^+$, if $q \in B^+(r)$, and

(ii) $(p, q) \in E^-$, if $q \in B^-(r)$.

Based on this, we define a *stratification* Σ of P as a set partitioning $\Sigma_1, \ldots, \Sigma_k$ of predicates in P, such that for each $p \in \Sigma_i$ and $q \in \Sigma_j$,

(i) $(p, q) \in E^+$ implies $i \geq j$, and

(ii) $(p, q) \in E^-$ implies $i > j$.

If such a stratification Σ exists, we call P a *stratified program*. By the subsets Σ_i, we obtain the *strata* P_i of P by those with head atoms in Σ_i. Given a stratification, one can evaluate the program bottom-up, using an iterative minimal model computation along strata, where at each stratum the intermediate result is obtained by a fixed-point computation as for positive programs. In the absence of constraints, stratified disjunctive programs always have an answer set, and stratified normal programs have a unique answer set.

Computational Complexity

Finally, we recall central complexity results for ground Answer Set Programming.

Theorem 1 ([MT91]) *Let P be a ground (extended) normal program. Then, deciding whether P has an answer set is* NP*-complete.*

The intuition of this result is that we can first guess a model M and then verify in polynomial time that it is an answer set: the reduct P^M is computed in polynomial time; it is a positive program and we need to check that M is the least model of P^M. The least model is efficiently obtained by the least fixed-point of an *immediate consequence operator* that will collect at each evaluation step the head h of any currently firing rule $h \leftarrow B^+(r)$, i.e., where $B^+(r)$ already holds (cf. [EIK09]).

No guess is needed when there are no loops through negation. We argued above that stratified programs can be evaluated efficiently.

Theorem 2 *Let P be a ground normal program. If P is stratified, then answer set existence can be decided in linear time.*

On the other hand, allowing disjunctive rule heads increases the complexity.

Theorem 3 ([EG95]) *Let P be a ground (extended) disjunctive program. Then, deciding whether P has an answer set is Σ_2^P-complete.*

For the membership in Σ_2^p, one observes that after guessing a model M for the reduct P^M, verification that M is minimal is in co-NP (cf. [Cad92]). To show the hardness part, one can reduce the validity of a quantified Boolean formula (QBF) of the form $\exists \mathbf{X} \forall \mathbf{Y} E$ to answer set existence of a program P, where E is formula over variables from lists $\mathbf{X}$ and $\mathbf{Y}$. For details, we refer to [EG95].

This concludes our brief review of ASP. A longer introduction can be found in [EIK09], which incrementally develops ASP bottom-up by considering fragments of increasing

complexity; it provides many examples and also investigates the practical aspect of ASP as implementation language, including a discussion of solving techniques. A more informal survey on the ASP problem solving paradigm is given in [BET11]. Most recently, a special issue on ASP was presented in [SW18].

2.2 Stream Processing and Reasoning

In this section, we will review the state of the art in stream processing and reasoning approaches, i.e., theoretical and practical contributions from various communities and research areas. Due to the wide spectrum of related approaches relevant to this work, the following overview is necessarily selective.

2.2.1 Temporal Reasoning and Verification

Starting with Pnueli's seminal work [Pnu77], which introduced *Linear Temporal Logic* (LTL), temporal logics have been studied extensively for formal verification of software and hardware. *Model checking* [CGP99] considers whether a desired property φ is implied by a system description M; formally this is the decision problem $M \models \varphi$. A system M is typically considered to be a finite state machine, which can be formalized as a *Kripke structure*, i.e., a triple $M = \langle S, R, L \rangle$, where S is a (finite) set of states, $R \subseteq S \times S$ a transition relation, and $L : S \to 2^{\mathcal{A}}$ an evaluation function that assigns each state $s \in S$ a set $L(s) \subseteq \mathcal{A}$ of propositional atoms. A temporal logic formula then specifies potential paths $\sigma = s_0, s_1, \ldots$ in M, where at each state $s_i \in S$ a property φ can be evaluated. These paths correspond to runs of the system, and $M \models \varphi$ can be decided by an exhaustive search in the specified state space. Thus, naive model checking is computationally expensive, since the transition system grows exponentially with the size of the system description. This so-called *state explosion problem* is the main disadvantage of this verification technique and has been tackled by a wide range of research efforts [CHVB18].

During the last decade, *runtime verification* [LS09, BFFR18] has been gaining considerable popularity as a more efficient verification technique where the aim is not to verify an entire system description with respect to a property, but only the *execution* (*trace*) of a single run, i.e., a finite subsequence (prefix) of a (possibly infinite) sequence of states. Importantly, verification shall take place *during* the execution in order to react when violations are detected. Accordingly, a *monitor* assesses an execution with respect to a correctness property φ, i.e., whether it is a valid trace. Typically, a monitor is generated automatically from φ.

Noting their importance for formal verification, we now turn our attention to specific temporal logics. Many of the logic-based verification techniques are based on a variant of LTL, which was specifically introduced as a means for program verification and thus targeted the formalization of two concepts: *invariance* and *eventuality* [Pnu77]. Invariance describes the continuity of a property throughout the execution of a program as described by a formula. Eventuality, also called *temporal implication*, describes the

temporal dependence between two properties, i.e., that a property φ is followed by some other property ψ. The latter concept particularly addresses, in contrast to earlier work in verification, non-terminating cyclic programs like operating systems that do not naturally end in a halting state. LTL extends the syntax of propositional formulas by the unary operators X (next), F (finally), G (globally) and binary operators U (until), R (release), W (weak until), and M (strong release). Evaluated at a given position in a path, i.e., a sequence of positions, they intuitively express the following conditions, given formulas φ and ψ:

- $\mathsf{X}\,\varphi$ holds if φ holds in the next position,
- $\mathsf{F}\,\varphi$ holds if φ holds eventually, i.e., in some future position (including the current position),
- $\mathsf{G}\,\varphi$ holds if φ holds always, i.e., now and in all future positions,
- $\varphi\,\mathsf{U}\,\psi$ holds if φ holds at least until ψ holds, and ψ has to hold now or in a future position,
- $\varphi\,\mathsf{R}\,\psi$ holds if φ holds exactly until the position where ψ holds (or infinitely if ψ will never hold),
- $\varphi\,\mathsf{W}\,\psi$ holds if φ holds at least until ψ, or forever in case that ψ never becomes true, and
- $\varphi\,\mathsf{M}\,\psi$ holds if φ holds exactly until ψ holds, which has to hold in a future position.

The semantics is defined over infinite sequences $\sigma = \sigma(0), \sigma(1), \ldots$ of positions, relative to a given position i. Given an evaluation function ν that assigns sets of atoms to positions,

$$\sigma, i \models a \;:\Leftrightarrow\; a \in \nu(\sigma(i)), \text{ where } a \text{ is an atom.} \tag{2.18}$$

The definition of propositional connectives is then inductively defined as usual, and moreover,

$$\sigma, i \models \mathsf{X}\,\varphi \;:\Leftrightarrow\; \sigma, i+1 \models \varphi \tag{2.19}$$

$$\sigma, i \models \varphi\,\mathsf{U}\,\psi \;:\Leftrightarrow\; \sigma, j \models \psi \text{ for some } j \geq i \text{ s.t. } \sigma, k \models \varphi \text{ for all } i \leq k < j. \tag{2.20}$$

The other operators can then be defined based on *until*: $\varphi\,\mathsf{R}\,\psi :\Leftrightarrow \neg(\neg\varphi\,\mathsf{U}\,\neg\psi)$, $\mathsf{F}\,\varphi :\Leftrightarrow \top\,\mathsf{U}\,\varphi$, $\mathsf{G}\,\varphi :\Leftrightarrow \neg\,\mathsf{F}\,\neg\varphi$, $\varphi\,\mathsf{W}\,\psi :\Leftrightarrow \psi\,\mathsf{R}\,(\psi \vee \varphi)$, and $\varphi\,\mathsf{M}\,\psi :\Leftrightarrow \neg(\neg\varphi\,\mathsf{W}\,\neg\psi)$; in some cases other characterizations are possible. These operators allow one to express invariances by formulas of the form $\varphi \rightarrow \mathsf{G}\,\psi$, including system properties such as safety, partial correctness, mutual exclusion, and deadlock freedom; dually, eventualities such as liveness and total correctness correspond to the form $\varphi \rightarrow \mathsf{F}\,\psi$ [GPSS80]. Notable variants of LTL include PLTL [LPZ85, LS95], which includes past versions of LTL operators (to look back), and NLTL [LMS02] which further adds unary operator N; it intuitively reads "now" or "from now on." Its effect is that past moments are forgotten, allowing one limit the

horizon of past events prior to specified properties. While sentences in these languages can be rewritten to LTL, NLTL can be exponentially more succinct than PLTL, which in turn can be exponentially more succinct than LTL.

In contrast to LTL and the mentioned variants, *Computation Tree Logic* (CTL) [CE81] evaluates formulas on a tree-structure of evaluation paths. Accordingly, one explicitly quantifies existentially (E) or universally (A) with respect to these paths, followed by an LTL operator. For instance, $\mathsf{A}\,\mathsf{X}\,\varphi$ requires that φ holds in the next step on all branching possibilities, and $\mathsf{E}\,\mathsf{G}\,\varphi$ holds if φ holds globally in some path. LTL and CTL are incomparable with respect to expressive power: although some formulas can be expressed in both languages, each logic has formulas not expressible by the other. The different version of a branching temporal logic is CTL* [EH86], which subsumes both LTL and CTL; it combines quantifiers from LTL in a more flexible way with path quantifiers as used in CTL. Notably, model checking for LTL, CTL and also CTL* is PSpace-complete; the complexities of satisfiability testing are PSpace-complete, ExpTime-complete and 2ExpTime-complete, respectively.

An extension of LTL is *timed propositional temporal logic* (TPTL) [AH89] that is evaluated over *timed state sequences* $\rho = (\sigma, \tau)$, i.e., where successive states σ_i in σ are associated with monotonically increasing time values $\tau(\sigma_i)$. TPTL considers time variables x with so-called *freeze quantifiers*, denoted by $x.$, i.e., an appended dot. Given a formula $\varphi(x)$ in which x occurs free, satisfaction is defined by $\rho, i \models x.\varphi(x) :\Leftrightarrow \rho, i \models \varphi(\tau(i))$, where $\varphi(\tau(i))$ is obtained by replacing free occurrences of x in φ by the time constant $\tau(i)$ associated with the i-th state [AH93]. The freeze quantifier allows one to explicitly refer to time, and thus also express constraints on timing. Using the natural numbers as time domain yields ExpSpace-completeness for both model checking and satisfiability, whereas a dense time ontology (the real numbers respectively the rational numbers) or allowing addition over time results in undecidability. Unlike for LTL, considering past operators in TPTL increases the complexity, making the validity problem non-elementary [AH93].

Metric Temporal Logic (MTL) [Koy90, AH93] considers expressions with time bounds. The notation $\Diamond_{\leq c}\,\varphi$ (corresponding to $\mathsf{F}_{\leq c}\,\varphi$ in our notation) intuitively states that φ holds "eventually within time c," which is observed to be a notational variant of (a subset) of TPTL [AH93]. Unlike the latter, MTL can be augmented with past temporal operators without an increase in complexity: deciding validity in both MTL and the resulting logic MTL$_P$ is ExpSpace-complete.

A fragment of MTL was proposed in [AFH96] that considers a finite but arbitrary precision in the temporal bound to achieve decidability. Their *Metric Interval Temporal Logic* (MITL) also uses a dense time ontology and allows one to express truth of statements over a non-singular time interval I with integer end-points; single time points are disregarded. The definitions are based on a refined (bounded) *until* operator: $\varphi\,\mathsf{U}_I\,\psi$ holds if for some time $t \in I$, ψ holds at t and φ holds at all times $t' < t$. (A next-time operator X is not considered due to the dense time domain.) Based on this, adopting our notation, $\mathsf{F}_I\,\varphi$ for *time-constraint eventuality* and $\mathsf{G}_I\,\varphi$ for *time-constrained always* are defined as in LTL, i.e., by $\top\mathsf{U}_I\varphi$ and $\neg\,\mathsf{F}_I\,\neg\varphi$, respectively. As in MTL with dense time ontology, satisfiability and model checking in MITL are undecidable in general, but

ExpSpace-complete if point intervals excluded.

Recently, a Horn fragment of MITL called *Metric Time Datalog* (datalogMTL) was considered in [BKK+17]. The datalogMTL programs in [BKK+17] contain rules of the form $A^+ \leftarrow A_1 \wedge \cdots \wedge A_k$ where each A_i is either (i) an inequality or (ii) a formula $\mathsf{Op}^1_{I_1} \cdots \mathsf{Op}^n_{I_n} B_i$, $n \geq 0$, where each Op^i is either the always operator (denoted by $\boxplus$ instead of G), the eventual operator ($\oplus$ instead of F), or their past versions ($\boxminus$ and $\ominus$, respectively); furthermore, each I_i is an interval over rational numbers, and B_i is an atom, and A^+ is a formula in without $\oplus$ and $\ominus$. A canonical model property for query answering is worked out, which is akin to the least model property of Horn logic programs. Based on it, query answering is shown to be ExpSpace-complete already in the propositional case. Notably, allowing $\oplus$ and $\ominus$ in rule head leads to undecidability. The canonical model property is then exploited to develop a first-order rewriting for non-recursive (acyclic) Horn rules, referred to as datalog$_{\text{nr}}$MTL. For this fragment, query answering is PSpace-complete. We further note the recent P-MTL [TH16], a probabilistic extension of MTL, developed for stream reasoning in uncertain environments, that considers stochastic states at/between time points and stochastic predictions of future and past states.

Another extension of LTL is *Signal Temporal Logic* (STL) [MN04], which targets applications that require monitoring of continuous signals. A signal s is formalized as partial function from the non-negative real numbers to a specified domain; a value $s[t]$ (at time t) in this domain is based on an interval $I = [0, r)$, where r is a non-negative rational number. The definition of STL builds on MITL$_{[a,b]}$, a restriction of MITL where all intervals in temporal modalities have the form $[a, b]$ such that $0 \leq a < b$; a Boolean domain is considered. STL then uses real-valued signals, i.e., the domain $\mathbb{R}^m$, $m \geq 1$. To obtain a logic, static abstractions of the form $\mu : \mathbb{R}^m \to \{0, 1\}$ are employed that partition the continuous state-space based on inequalities on real variables. That is, only the constant $\mu(s[t])$ is of interest in the logic, not the exact value $s[t]$. Viewing state abstractions $\mu_1, \ldots, \mu_n$ as predicates, STL formulas are obtained by MITL$_{[a,b]}$ formulas over atomic propositions $\mu_1(x), \ldots, \mu_n(x)$. The evaluation of real-valued signals is then reduced to MITL$_{[a,b]}$ using the functions μ_i, i.e., STL amounts to a real-valued fragment of MITL$_{[a,b]}$. It is a particularly useful logic for the description of Cyber-Physical Systems (CPS) [BDD+18] which connect changes in the physical world (represented as real values) with discrete events in digital systems. This hybrid nature of CPS is well reflected by STL. Accordingly, multiple works consider the monitoring problem directly for STL.

Offline monitoring assumes that an entire execution is available for analysis. Then, the monitoring algorithm should produce a qualitative satisfaction value (Boolean) or a quantitative satisfaction value (robust satisfaction). The *robustness degree* is a measure indicating how far a signal is from satisfaction or violation of a formula. For instance, given a predicate $\mathbf{x} < c$, where c is a constant and $\mathbf{x}$ variable for a real number, the robustness degree of a concrete value x of $\mathbf{x}$ is the relative distance of x to c. An efficient (linear) algorithm to compute the robustness degree was presented in [DFM13]. It utilizes a streaming algorithm which computes the minimum and maximum of a sequence of numbers in a sliding window [Lem06] and reduces the bounded until operator to multiple simpler operations. A monitoring tool for boolean evaluation was given in [NM07].

In practice, offline monitoring is not always useful or feasible. In particular, when the behaviour of a real-time system shall be corrected while it is running, *online monitoring* is required, which should provide satisfaction estimates based on partial signals and be highly efficient with respect to memory consumption and evaluation time. Often, a three-valued semantics is used to cope with situations where the available finite trace is inconclusive [BLS06]. There are two main kinds of qualitative online monitoring techniques, i.e., algorithms for deciding satisfaction/violation of a property based on the available prefix of an execution. The first method is *incremental marking* [MN13], which computes satisfaction of a formula bottom-up in the subformula-tree, separately maintaining signal values that have been propagated and those whose super-formula evaluation still depends on them. The second method [HOW14] is based on MTL formulas which are rewritten into LTL and evaluated with automata. Quantitative online monitoring algorithms return a quantitative measure for robust satisfaction. Three different approaches based on different fragments of STL are summarized in [BDD$^+$18], which also presents further details on the works mentioned above. Furthermore, it surveys techniques for extensions of STL and a discussion of recent applications of monitoring in CPS.

Execution monitoring by means of temporal logic is also considered in [KHD08, DKH09] to detect system violations based on declarative specifications. To this end, the *Temporal Action Logic* (TAL) [DK08] is employed which builds on a macro language $\mathcal{L}$(ND). A *monitor formula* in $\mathcal{L}$(ND) builds on MTL and additionally considers features with specific value domains. Accordingly, *elementary fluent formulas*, i.e., expressions of the form $f \hat{=} \omega$, state that a *feature* or *fluent* f has value ω. This atemporal formula can be prefixed with (open or closed) intervals to state its temporal validity. Furthermore, the syntactic form $\omega = \omega'$ is available to express equality of values, and also and (in)equalities between temporal terms are provided. The semantics of $\mathcal{L}$(ND) formulas is then defined by translations into first- and second-order logical theories in a logical language $\mathcal{L}$(FL) that defines fluents, their values, actions and the time domain but does not contain the tense operators from MTL. Towards online monitoring, a progression algorithm pr as in [BK98] is used such that a formula φ holds in a sequence $s_0, s_1, \ldots, s_n$ of states iff $pr(\varphi, s_0)$ holds in the sequence $s_1, \ldots, s_n$. The algorithm ensures that violation at state s_0 can be detected, and otherwise returns a new formula to be checked at the next state, which enables incremental monitoring. Notably, the same logical formalism is used both for planning and monitoring, simplifying the generation of monitor formulas from execution plans. These temporal reasoning features have been extended towards spatio-temporal reasoning in [HdL14] with a focus on incomplete spatial information. Spatial reasoning there is based on the Region Connection Calculus RCC-8 [RN01] that considers spatial relations such as *externally connected*, *equals*, *disconnected*, etc. More recently, MTL and RCC-8 have been combined in the *Metric Spatio-Temporal Logic* (MSTL) [dLH16], which directly equips MTL with spatial relations from RCC-8.

These and other methods have been implemented in *DyKnow* [HD04, HKD10c, HKD10a, HKD10b], an architecture for stream reasoning developed in the context of cognitive robotics, and in particular, autonomous unmanned aerial vehicle (UAV) development. DyKnow offers a systematic middleware solution for combining low-level

sensing with high-level cognitive functions such as planning and goal-directed acting in an uncertain environment, specifically targeting real-time scenarios. Of particular importance is the need to filter data and abstract higher level abstractions from heterogeneous sensors (such as a GPS location and visual input from a camera) efficiently in order to reason over relevant conceptual entities in real-time. A traffic monitoring use case is given (e.g. in [HKD10a]), dealing with the automated navigation of a helicopter for surveillance tasks such as tracking specific cars or detecting accidents. Such events are declaratively specified, e.g., by spatial relations such as `close(car1,car2)`, and are recognized due to hierarchical descriptions that link lower-level signals to high-level symbols. In DyKnow, incremental processing of streams, identified with labels, is carried out by various *knowledge processes* (at different levels of granularity) that publish information by means of stream generators for further processes to subscribe. Knowledge processes can also access static data such as road systems. A declarative *knowledge processing language* is proposed, making it possible to compose a processing network based on knowledge processes and streams. A stream has to satisfy certain requirements as described by declarative *policies*, which can state conditions and constraints (e.g. regarding order or delay, or specific values). A policy can also influence the stream generation and thus serves to hierarchically abstract low-level sensor information to high-level descriptions such as representations of objects or their geographical locations. Notably, the satisfaction of policies is defined based on a formal entailment relation.

To the best of our knowledge, DyKnow is the first work explicitly considering continuous logical reasoning on streaming data and can thus be seen as pioneering work in stream reasoning.[1] With the idea to compose low-level input features hierarchically into more complex features, DyKnow was also shown to be suitable for chronicle recognition [Gha96] (cf. [HD04, HKD10b]), i.e., it includes capabilities of complex event processing (cf. Section 2.2.3). Towards automatic configuration and on-demand semantic processing, DyKnow has been extended in [dLH14] with event processing functionalities of the C-SPARQL stream processing language (cf. Section 2.2.4). We further note that multiple developments for DyKnow have been integrated in the Robot Operating System (ROS) [QCG+09], including the works on semantic integration [HD12, HdL13].

2.2.2 Stream Processing and Data Management

Prior to the late 1980s, databases have predominantly been passive tools for managing data, in the sense that queries or update commands were typically carried out explicitly by a program or a user. Emerging real-time and monitoring use cases then led to the development of *active databases* [Day88] that use *event-condition-action* rules to automatically modify the database: an event triggers the test of a condition, and when it holds, a certain action is performed, e.g., inserts of additional rows, enforcing integrity constraints, view updates, or information collection for optimizing queries. An overview of the rapid initial development of active databases can be found in [WC96, PD99]; trigger mechanisms can now be found in most modern databases. Among the early works

[1] A different path towards the term *stream reasoning* is found in the Semantic Web area, cf. Sec. 2.2.4.

in this area was the Alert system [SPAM91] which introduced *active queries* on append-only tables, i.e., expressions of triggers in SQL that could be mixed with static queries. Similarly, *continuous queries* over append-only databases were proposed in [TGNO92], which introduced the *Tapestry* system; it considered incremental evaluation of new tuples in a database. This system was designed for filtering new text messages such as mails, but was developed in general terms. In particular, the authors defined a *continuous semantics* whose results correspond to the returned data when a query is issued at every instant in time.

An explicit shift towards streaming data has been proposed in [CÇC+02, ACÇ+03], which analyzed that some assumptions of then state-of-the-art database management systems (DBMS) were not suitable for monitoring streams. The resulting *Aurora* system sought to improve previous efforts regarding triggering mechanisms, the handling of incomplete or filtered data, and the need for real-time responses. It provided means to create a network of streaming operators, including *sliding* and *tumbling windows*, i.e., recent selections of data that progresses gradually, respectively without overlaps. Furthermore, a *Latch* window could maintain state of multiple window operations, thereby allowed one to potentially evaluate the entire history of the stream. Its query model combined these operators to a global, procedural query execution plan. *Borealis* [AAB+05], the distributed successor of Aurora, tackled technical challenges regarding the dynamic behaviour of the system, including the revision of query results or query modification, e.g., as response to system overloads. Also *TelegraphCQ* [CCD+03] addressed the need for adaptive query processing, building on the PostgreSQL database. Among the first stream systems to particularly deal with scaling issues for the web was *NiagaraCQ* [CDTW00] which proposed an incremental grouping method to speed up queries based on the similarity of underlying data.

Another notable contribution in the early development of stream processing systems from the database area was the *Stanford Stream Data Manager* (STREAM) [ABB+03], which employed the influential *Continuous Query Language* (CQL) [BW01, ABW06]. The latter is an extension of SQL that provides streams as data sources in addition to tables. In contrast to other SQL-based approaches, CQL provides an explicit operational semantics based on three operators: stream-to-relation (S2R), relation-to-relation (R2R), and relation-to-stream (R2S). The first, S2R, creates a common view on streams and tables in order to make SQL features uniformly accessible in the R2R-part; and R2S then creates an output stream for a continuous query. Importantly, S2R serves as abstraction for various window operators with the central idea to view *snapshots* of recent data essentially as relations. CQL considered three kinds of concrete window operators. *Time-based* windows select recent data due to a fixed temporal range, e.g., the last 10 seconds, or the last 60 seconds, updated every 15 seconds. Dually, *tuple-based windows* select a fixed number of the most recent tuples, regardless of when they appeared. *Partitioned* windows extend the latter by allowing the user to define virtual substreams that are separately counted. The approach to tackle streaming data by repeated or incremental evaluations of recent snapshots has been adopted by many follow-up works, in particular in the Semantic Web area (cf. Section 2.2.4). A language similar to CQL was also used in

[LMT+05] which studied efficient evaluation techniques for different window mechanisms based on explicit semantic definitions that are independent from implementations. Later, the *Spade* system [GAW+08] provided a declarative language for the composition of data-flow graphs in the distributed stream processing middleware *System S* [WYG+07]. As such, it was designed as intermediate language for rapid prototyping in System S, not as stream processing query language. Another line of research within the database area that is connected with stream processing and reasoning concerns incremental update of database views. Work from this area will be reviewed in Section 6.6.

In recent years, a number of new tools have been developed for distributed stream processing with a focus on high-throughput, scalability, fault-tolerance and programming APIs in different languages to create or connect stream processing pipelines. The Apache Software Foundation maintains a notable collection of open source tools that are now widely used. *Apache Kafka*[2] is a platform for managing publish/subscribe models for streams similarly as for message queues; it can store streaming records itself. *Apache Spark*,[3] a batch processing framework [ZCF+10, ZXW+16], quickly gained momentum in the area of so-called Big Data analytics, outperforming the popular *Apache Hadoop*[4] framework for cluster computing. Among its core libraries is *Spark Streaming* [ZDL+13], a module in which complex algorithms on streams can be expressed (in Scala, Java or Python) with declarative constructs similarly as in the *MapReduce* approach [DG08]. In addition, a time-based window can select recent data chunks. *Apache Flink*,[5] a system explicitly designed for highly efficient stream processing [CKE+15], offers APIs to access data at different levels of abstraction. In addition to *time driven* windows (similarly as in Spark), Flink provides *count driven* windows that select a recent number of tuples. Moreover, windows progress can be defined as sliding or tumbling, or in form of *session windows* which group data tuples by periods of activity due to the static or dynamic definition of gaps. Flink distinguishes *unbounded streams* that need to be continuously and immediately processed, and *bounded streams* that can first be consumed as a whole before evaluation needs to start; the latter kind amounts to batch processing. *Apache Apex*[6] also unifies batch processing and stream processing and provides different window mechanisms. By contrast, *Apache Storm*[7] and *Apache Samza*[8] focus exclusively on stream processing. Compared to Spark and Flink, Storm focuses on the topology of streaming architectures and low-level processing. Storm, Flink and Spark Streaming have been compared in [CDE+16], which provides a benchmark for real-world production scenarios; among the results was that Spark could handle higher throughput, while Storm and Flink processed streams with lower latencies.

In summary, these frameworks provide similar solutions for distributed, fault-tolerant, robust, secure processing of streams in heterogeneous environments. Building on general

[2] `https://kafka.apache.org`
[3] `https://spark.apache.org`
[4] `https://hadoop.apache.org`
[5] `https://flink.apache.org`
[6] `https://apex.apache.org`
[7] `https://storm.apache.org`
[8] `https://samza.apache.org`

purpose languages, they offer means to structure streaming pipelines in larger software architectures; some of them include high-level constructs for data transformations. While these works do not build on model-theoretic query semantics, some of them emphasize the importance of declarative APIs. In particular, the *structured streaming* [ADT+18] approach in Spark provides a high-level, functional API that extends the previous Spark SQL interface [AXL+15]. Kafka and Flink also provide implementations for querying streams with SQL. Various tools now refer to SQL-based extensions for SQL by the umbrella terms *StreamSQL* or *Streaming SQL* [JMS+08].

In addition to the above tools, stream processing systems are nowadays also available as managed cloud services; among them are *Google Cloud Dataflow*[9] [ABC+15], *Microsoft Azure Stream Analytics*,[10] and *Amazon Kinesis*;[11] they integrate with batch processing and other forms of large-scale data processing.

2.2.3 Complex Event Processing

Complex Event Processing (CEP) [Luc01] can be seen as special form of stream processing where data signals are explicitly viewed as notifications of events. The generic task is to derive *composite events* (or *situations*) due to complex patterns in the event stream. In contrast to evolved DBMS (cf. Section 2.2.2), which center around querying stored information, CEP system architectures are directly designed for the streaming behaviour: a CEP engine observes external low-level events via sources, processes them immediately to compute high-level events, and sends them to output sinks, i.e., event consumers. For instance, consider a composite event marking the outbreak of fire, which is recognized by multiple rapid increases of sensed temperatures in a given area. In contrast to most stream processing approaches, CEP systems typically offer declarative languages at higher abstraction levels than SQL. In particular, expressions of temporal intervals to capture and combine the duration of events, or sequences thereof, are common.

Among the first CEP developments was *Rapide* [LV95, Luc96], an event-language for concurrent and distributed system architecture simulations. The execution of a Rapide model makes the timings between events and their causal relation explicit, respectively their independence. The *CEDR* system [BGAH07] addressed multiple technical challenges of CEP with the aim of providing an expressive declarative query language. First, its temporal stream model for interval-based events refined the notions *application time* and *system time* [SW04]. More specifically, the application time was separated into a *valid time*, that can be altered, and an *occurrence time*, which stores which valid time was used when. This distinction paved the way for different *consistency guarantees*; in particular, CEDR showed how *retractions* of previous output can be used to tackle out-of-order events without blocking. Notably, the query language of CEDR was given a formal (operational) semantics, and included event sequences and negation to express conditions based on the non-occurrence of events.

[9] `https://cloud.google.com/dataflow`
[10] `https://azure.microsoft.com/en-us/services/stream-analytics`
[11] `https://aws.amazon.com/kinesis`

The event monitoring system *Cayuga* [BDG+07, DGP+07] provides a query language close to SQL, where the `FROM`-clause defines a stream expression, i.e., an event pattern, and the keyword `PUBLISH` precedes the definition of an output stream. The semantics of the Cayuga query language is formally defined based on the Cayuga algebra [DGH+06] but in contrast to CQL it offers no window mechanisms. The resulting system was designed with an emphasis on efficiency and large-scale scenarios, but distributed computation was not supported.

Also the *Sase* system [WDR06] targeted efficient solutions for large-scale event processing, focusing on RFID readings. Queries in Sase are composed of (at most) three constructs: an `EVENT` pattern that has to be matched, a `WHERE`-clause that specifies additional conditions, and a `WITHIN`-clause that expresses a sliding time-based window. The Sase language comes with a formal semantics, based on which optimizations were presented. The implementation of the Sase system creates a query plan based on a proposed dataflow paradigm that composes a pipeline of operators which transforms the input stream to an output stream in six steps. These steps include the creation of sequences and handling negation. A performance analysis showed superior performance compared to TelegraphCQ [CCD+03]. Sase was later extended towards the *Sase+* [GADI08] event language which was able to detect Kleene closure patterns, i.e., an unbounded number of events with the same property. Building in particular on Sase+, efficient pattern matching was the subject of study in [ADGI08], where previous work on automata-based evaluation [WDR06] was further developed and analyzed. In particular, sharing techniques led to significant performance gains.

A notable rule-based event processing system is *ETALIS* [AFR+10, ARFS12], which has a model-based semantics and is thus well-suited for stream reasoning. ETALIS combines CEP techniques, i.e., event pattern matching and derivation of complex events, with logic-programming, and is thus of particular interest in this work. Thus, we will review its capabilities in more detail in Section 4.4.

A complex event processing language with high expressiveness is *Tesla* [CM10], whose semantics is formally defined in a metric temporal logic. A Tesla rule is composed of four parts. The `define` part creates a complex event due to attributes of simpler events from the `from` part. Dependencies on these attributes can be specified with functions in the `where` part. Optionally, a `consuming` part can invalidate events for repeated rule firing. The simplest form of event pattern is the occurrence of single event; complex events can then be composed based on operators that relate them temporally. Furthermore, *timers* in the `from` part can be used to explicitly trigger rule firing periodically, and a *not* operator allows one to reason over events that did not occur. Furthermore, Tesla supports aggregations (such as computing averages), hierarchies of events, and iterations of patterns (Kleene closures). The authors also show how event detection with this language can be realized using automata.

A concrete CEP system implementing the Tesla language is *T-Rex* [CM12], which has been shown to outperform the commercial CEP system *Esper*[12] (Version 4.0.0) in a detailed comparison. The latter provides a query language that extends SQL, e.g.,

[12] `http://www.espertech.com/esper`

by event pattern matching constructs and window mechanisms, tailored for streams and events in combination with static data. Other commercial systems include *Oracle CEP*,[13] which builds on CQL, and *Streaming Analytics* from *Tibco*,[14] formerly known as *StreamBase*. As the websites of many commercial vendors suggest, the term *complex event processing* is nowadays less used; extensions of SQL for streams and other event-based systems are typically found under keywords such as *event (stream) processing*, *Big Data/real-time/stream analytics*, etc. Naturally, the lines between information systems that process events and other streaming data are inherently blurry and evolve over time. Many modern systems, in particular cloud services, blend or combine different large-scale data processing methods for static and streaming data, and CEP is often subsumed in their functionality. For instance, Apache Flink supports CEP as library,[15] and Oracle CEP has now evolved to *Oracle Stream Analytics*[16] which delegates the event stream handling to Apache Spark.

For further details on the historic development of stream processing and complex event processing, we refer the interested reader to the survey in [MC11]. Furthermore, we note that a proposal for a common foundation of various complex event processing approaches based on denotational semantics was recently presented in [GRU17].

2.2.4 Semantic Web

Data management in the Semantic Web area builds on the Resource Description Framework (RDF),[17] a graph-based data model that stores data in form of triples (S, P, O), comprising a subject S, a predicate P, and an object O. The standard query language is SPARQL [HSP13, PAG09],[18] which is similar to SQL, using constructs like `SELECT`, `FROM`, `WHERE`, etc. Following the extension of CQL for SQL, multiple works have considered a similar extension of SPARQL to include streaming data as additional data sources. This has led to research efforts on *RDF Stream Processing (RSP)*;[19] we will briefly review some highlights of this line of research.

C-SPARQL [BBC+09, BBC+10a] extends the SPARQL sytax with time-based and tuple-based (triple-based) windows similarly as CQL. It assumes that streaming RDF triples are annotated with a timestamp which can be accessed via a *timestamp function* in the `WHERE`-clauses of queries. The execution environment of C-SPARQL, discussed in [BBCG10], makes use of existing tools for data stream processing. This is in contrast to CQELS [PDPH11], which similarly extends the syntax of SPARQL for streams, but provides a native implementation towards higher throughput. To this end, CQELS is equipped with an adaptive query processing technique which is able to react to changes in the input data, i.e., internal operators in the execution pipeline can be reordered

[13] https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/toc.htm
[14] https://www.tibco.com/streaming-analytics
[15] https://flink.apache.org/news/2016/04/06/cep-monitoring.html
[16] http://www.oracle.com/technetwork/middleware/complex-event-processing/documentation/
[17] https://www.w3.org/RDF
[18] https://www.w3.org/TR/rdf-sparql-query/
[19] https://www.w3.org/community/rsp/

automatically based on heuristics to speed up query evaluation. Second, caching, indexing and data encoding techniques serve to achieve a significantly better performance than C-SPARQL and ETALIS in an empirical comparison on five selected queries. However, despite the similar syntax of C-SPARQL and CQELS and their conceptual similarities, their effective semantics (in terms of their system outputs) differ due to different execution modes. We will discuss this further in Section 4.5. Another SPARQL extension that conceptually draws from CQL is SPARQLStream [CCG10], which allows one to query streams with time-based windows only; triple-windows and access to static data are not available in their prototype, hence it was not included in the evaluation of CQELS.

EP-SPARQL [AFRS11] builds conceptually on ETALIS [ARFS12] (cf. Section 4.4). It adds to the SPARQL syntax binary operators SEQ, EQUALS, OPTIONALSEQ and EQUALSOPTIONAL to combine query expressions (i.e., so-called graph patterns) similarly as UNION and OPTIONAL in SPARQL. These constructs introduce joins that depend on temporal information. For instance, given patterns P_1 and P_2, P_1 SEQ P_2 results in a join of P_1 and P_2 if they occur in a SEQ relation as in ETALIS, i.e., the instantiation of P_1 must occur strictly before that of P_2. Moreover, functions are provided for expressing conditions on the timestamps of the start time and the end time, and the duration of triples in the `FILTER` clause. The execution model exploits event-driven backward chaining rules as in ETALIS, realized in a Prolog implementation.

As observed in [DTM+13] (for stream prcessing), conceptually identical queries may produce different results on different engines. This may be due to differences that either arise from potential flaws in implementations, but also due to (correctly implemented) different semantics. Without formal semantics, different approaches are confined to experimental analysis [PDP+12] or informal examination on specific examples. For the user it is important to know the exact capabilities and semantic behaviors of given methods for systematic analysis and comparison. An attempt to unify different RSP approaches has been presented with RSP-QL [DVCC14], a formal model to provide a common evaluation semantics. It builds on the formal semantics of SPARQL [PAG09], CQL [ABW06], and SECRET [DTM+13]. Based on this model, the authors explained in particular the differences between C-SPARQL, CQELS and SPARQLStream. More recently, RSP-QL was extended with Complex Event Processing features, resulting in the query language RSEP-QL [DDC+16]. In addition to RSP-QL, the extension also covers the sequence operator (SEQ) from EP-SPARQL, respectively the sequences constructible in C-SPARQL due to its timestamp function. Furthermore, RSEP-QL formalizes event consumption policies that define the invalidation of events for later evaluation steps. A different approach to extend SPARQL was considered for the STARQL language [ÖMN15] for ontology-based data access on sensor data. On top of queries over streaming and static data as in other approaches, the semantics of STARQL query answering includes ontological reasoning. To this end, queries are compiled to relational data in order to use exisiting tools for evaluation. STARQL provides sliding time-based windows and event pattern matching features, and was implemented in the *ExaStream* system [KBJ+16].

Event-driven processing principles, that influenced works like Apache Spark and Apache Storm (cf. Section 2.2.2) were explicitly adressed in [CA15], which presented a

reactive model of computation for RSP engines. It argued for asynchronous message passing based on the Actor model [Agh90] as an alternative to current tightly coupled designs of RSP engines to better address runtime and maintenance issues such as responsiveness, scalability and fault tolerance.

The recent *Strider*^{R} system [RCN+17] offers a solution for scalable stream reasoning on RDF graphs. In addition to processing queries in a SPARQL extension similar to those of C-SPARQL, reasoning in form of RDFS entailments (plus owl:sameAs) is offered. The architecture of *Strider*^{R} builds on top of Apache Kafka and Apache Spark (cf. Section 2.2.2).

Furthermore, [MCCD18] presents a system to combine ontological reasoning and temporal reasoning for streaming data, using Datalog for the former and Tesla [CM10] for latter. Using time-annotated triples as most Semantic Web approaches, it applies ontological reasoning at every time point, taking into account background knowledge. Temporal patterns are then used in a separate reasoning step. The presented DOTR system - abbreviating *Decoupled Ontological and Temporal Reasoning* is empirically evaluated, it clearly outperforms C-SPARQL and CQELS, and may also be faster than Esper in high throughput scenarios. Notably, it provides more flexibility regarding expressive reasoning than these tools; yet explicit window mechanisms are lacking. The authors explicitly exclude them (in particular sliding windows) to avoid three technical issues. The first concerns multiple matching of the same interval that is covered by consecutive windows and then may lead to duplicate results. This may be considered as problematic from an operational view point when focusing on temporal pattern detection. Second, intervals may never cover intervals, either because they are too small or partially selecting them in a tumbling movement. In this case, windows may lead to information loss. Third, using windows may lead to inconcistency during the reasoning process, for instance when a person is inferred to be at multiple positions concurrently. However, the authors explain that such inference problems stem due to the lack of explicit temporal operators, and that no efficient state-of-the-art system provides them as realization of a formal framework.

We observe that issues like flexible data models, high throughput and simultaneous querying of static and streaming data have been adressed by Semantic Web systems. However, a combination of *generic* window mechanisms with temporal control features seem to be lacking, as well as more expressive reasoning features as typically studied in Knowledge Representation and Reasoning like nonmonotonicity, default reasoning, or multiple possible solutions. Such features are important to deal with missing or incomplete data, respectively to enumerate alternative solutions and choices.

2.2.5 Knowledge Representation and Reasoning

Methods in Knowledge Representation and Reasoning (KR&R) typically provide reasoning methods for declaratively encoded knowledge using logic-based formalisms. Most literature in KR&R deals with static knowledge bases, e.g., databases, ontological schemas and assertions, rule bases, or theories in different logics, ranging from propositional logic, to modal logics and nonmonotonic logics. However, methods that deal with continu-

ously changing information, event-based or reactive systems have been considered less. In particular, formal foundations for stream reasoning that offer window operators as used in stream processing tools (cf. Section 2.2.2) seemed to be missing. Nevertheless, first attempts towards expressive stream reasoning have been recently carried out, and we mention here some highlights, with a focus on rule-based systems and Answer Set Programming.

An early proposal for ASP for stream reasoning was given in [DLL11], which presents an architecture, a formal model of streams and a prototype implemetation using the *dlvhex* solver [EIST06]. Streams are viewed as sensor data items that are totally ordered by their (unique) timestamps. Two kinds of sliding windows are considered, based on counting time or tuples. A *data window stream* is then the sequence of windows obtained at consecutive time points, and forms the basis for periodic evaluation. Each such window is then fed to an ASP-based system as a set of facts by an interface to external data via so-called external atoms in dlvhex. Consequently, this approach amounts to repeated ASP solving based on the proposed formalization of streams. However, no specific streaming semantics or operators for streaming are provided.

The lack of logic-based foundations for Data Stream Management Systems (DSMS), similar to those for relational databases, was observed in [Zan12]. This led to *Streamlog*, a system for streaming data based on Datalog. Every predicate has a timestamp as first argument; based on this *sequential programs* are defined that allow one to obtain a stratification and efficient computation of a (unique) model. Notably, Streamlog presents the notion of a *progressive* closed-world assumption (CWA) which essentially equates a predicate's negation with non-derivability *until the current time point*. This paves the way for evaluation of queries that otherwise would be *blocking*, i.e., depending on the availability of all data. By simply imposing restrictions on Datalog, Streamlog thus carries over an elegant formalism for static data to streaming scenarios. However, explicit treatment of streaming information beyond the timestamp arguments are not provided; in particular, no window mechanisms.

Reasoning over streams has also been considered in ontology-based data access (OBDA) [XCK+18]. Ontology Stream Management Systems (OSMS), as introduced in [RP11], consider the use of Truth Maintenance Systems to deal with large volumes of data in $\mathcal{EL}^{++}$ reasoning. Truth Maintenance Systems deal with the incremental update of a model due to changing justifications of possible entailments; we will review in particular Justification-based Truth Maintenance Systems [Doy79] in detail in Section 6.2. Neither Streamlog nor OSMS employ window mechanism. Streams of ontologies are also considered in the query language STARQL [ÖMN15] for query answering over streams; its features include time-based windows (cf. Section 2.2.4). A survey of recent developments in OBDA, more specifically on ontology-mediated query answer over temporal data, is given in [AKK+17].

Multiple works on the ASP solver Clingo have addressed the issue of data or program change. Incremental ASP [GKK+08] introduced new techniques for incremental grounding and solving based on the module theory in [OJ06] which allows for composing programs with explicit (distinct) input and output atoms. An incremental program is a triple

(B, P, Q) consisting of three program parts: B describes static knowledge; P and Q are slices that depend on a parameter t. At each step t, the program grows by a new set $P[t]$, while $Q[t]$ is considered only temporarily at t. Relying on according composition of modules, model computation can then be carried out incrementally. The work resulted in the solver iClingo, which uses declarations `#base`, `#cumulative t`, and `#volatile t` to delineate program parts B, P, and Q from above, respectively. In a step k, the parameter (variable) `t` in a rule of program part P or Q is then instantiated with k. All other variables can be grounded only once, i.e., these instantiations must be derivable from static knowledge. While incremental ASP focuses on stepwise computation of models, reactive ASP [GGKS11] targets real-time systems by providing additional means to add new data online. On top of incremental programs and its update mechanism, reactive programs support an asynchronous control via so-called *online progressions* of external events and inquiries which themselves are programs. In essence, at each step, external information can be incorporated in order to ground new rules dynamically. The resulting solver oClingo uses the additional declaration `#external` to introduce atoms that can be fed into the system in a streaming fashion. However, while reactive ASP provides incremental solving features for streaming data, it lacks a window mechanism. More precisely, the dynamic program parts P (which is cumulative) and Q (which concerns only the current step) do not fit the conceptual approach of windows which express the relevance of information relative to an interval of steps.

With the aim of providing such a window mechanism for stream reasoning, *time-decaying logic programs* [GGK+12] were defined as triples $(B, P, \{Q_1, \ldots, Q_m\})$, where the instantiation of each program part Q_i expires after a specified life span of n_i steps ($n_i \in \mathbb{N} \cup \{\infty\}$). Thus, each program part Q_i resembles functionality of a sliding window of length n_i. To incorporate this facility, oClingo's additional declaration of form `#volatile t:n` states that subsequent rules that are parameterized with variable `t` are discarded after `n` steps.

Ideas from incremental ASP, reactive ASP and time-decaying logic programs have been improved continuously and are now subsumed in the current version 5 of Clingo [GKKS17]. Its multi-shot solving capabilities to evaluate changing programs were presented earlier in [GKOS15] which gave an introduction to multi-shot solving by modeling the board game Ricochet Robots. These works all provide additional control for grounding and solving via additional parameters that can be accessed by an external script. Such mechanisms can be used, e.g., to simulate the progress of time and to encode certain window operators. While Clingo's multi-shot features target the incremental and reactive control of the ASP solving process, no explicit streaming semantics or operators are offered.

Another proposal for nonmonotonic stream reasoning is *StreamRule* [MAPH13], which emphasizes the potential of ASP-based reasoning for the Semantic Web. The proposed architecture combines the Linked Sensor Middleware (LSM) [PNPH12], CQELS [PDPH11] for query processing and pattern matching, and oClingo for subsequent rule-based reasoning. In a similar way, the *PrASP* system [NM15] for probabilistic Answer Set Programming is used as component of a probabilistic stream reasoning system architecture

in [NM14]. StreamRule was recently extended in [PMA17] to support parallelism by partitioning the input based on a syntactic analysis of the program.

Stream reasoning in *Temporal Datalog* has recently been studied in [RKG+18]. It distinguishes *rigid* predicates and *temporal* predicates; the latter have a time point as last argument. Programs are sets of rules as usual; they do not employ negation. The language is used to define four decision problems that capture different aspects in stream reasoning arising from possibility to refer to historic and future data and inferences. The *definitive time point* (DTP) problem addresses the question whether an output at a given time point might be invalidated by later input, and the *forgetting* problem is to determine which part of historic data is certainly irrelevant for future evaluations. The *delay* problem and the *window size* problem are data-independent variants thereof, respectively. The computational complexity of these problems is then studied, resulting in PSpace-completeness in data complexity for DTP and undecidability for the remaining tasks. Restricting to a class of non-recursive queries yields tractability for the first two tasks and co-NExpTime-completeness for the two data-independent problems. Temporal Datalog was explicitly proposed as foundational work for rule-based stream reasoning, and similarly as Streamlog [Zan12] did not provide explicit window mechanisms.

CHAPTER 3

LARS: A Logic-based Framework for Analytic Reasoning over Streams

This chapter introduces LARS, a logic-based framework for analytic reasoning over streams; it is a formalism for expressing the model-based semantics of stream reasoning systems and shall provide a theoretical underpinning for the study, development and analysis of stream reasoning methods. All subsequent chapters of this thesis build on top of this framework.

Outline

We start in Section 3.1 by presenting our formalization of streams and windows, followed by specific window functions to obtain them. We define in respective Sections 3.1.3-3.1.5 time-based, tuple-based and partition-based windows in our setting; i.e., the usual window functions selecting recent parts of historic data. After introducing filter windows in Section 3.1.6, we present in Section 3.1.7 generalizations of the standard windows that also can access future time points. After these building blocks, we introduce the LARS framework in Section 3.2. Based on LARS formulas (Section 3.2.1), that are evaluated on streams, LARS programs (Section 3.2.2) present a rule language for more expressive reasoning. We discuss the semantic properties of the proposed language in Section 3.2.3, followed by a complexity analysis in Section 3.3, where we study model checking and satisfiability for formulas and programs. In particular, we show how the generic framework yields more practical fragments under relevant restrictions, i.e., bounded window nesting (Section 3.3.3) and sparse window functions (Section 3.3.4).

Publications

This chapter presents a part of the journal paper [BDTE18]; further parts will be presented in Chapter 4. A draft of this publication was first released as technical report [BDTE17] that extended the earlier conference paper [BDEF15]. The latter was developed incrementally, incorporating feedback from two workshops [BDTEF14b, BDTEF14a].

3.1 Streams and Windows

This section introduces two central notions: streams and windows. From a conceptual perspective, we can view streams in contrast to relational databases, which are by and large time-agnostic, compound entities; they are typically large and change only due to explicit updates. Streams, on the other hand, can be seen as unbounded sequences of small, individual entities with an inherent notion of temporal appearance. Records in usual databases can be deleted, whereas streaming data can only be ignored or forgotten. Recent portions of a stream's entire history that are retained in memory are called windows, which can be selected in different ways due to window functions. Notably, the result of applying a window function to a stream is another (sub)stream; this allows for nesting of windows and thus to express complex data selections.

In Section 3.1.1 we give the intuitive idea of streams and our formalization, followed by the general notions of windows and window functions in Section 3.1.2. We then introduce in Sections 3.1.3-3.1.7 specific instances as used in practice.

3.1.1 Streaming Data

Streaming data can be viewed from different perspectives. Practically, streams can be seen as data objects that incrementally become available to be read and processed. We are interested here in a domain independent approach, i.e., we do not differentiate whether data signals are chunks of video streams, social media posts, server log entries, values from sensors, etc. A common conceptual abstraction is provided by the notion of an *event*, which is typically formalized as a tuple with a timestamp. In line with the setting of many practical systems for stream processing, we adopt a discrete, linear time ontology; streaming signals are reflected as atoms which are associated with time points.

Example 7 Fig. 3.1 shows the development of a stream. In addition to the data up to time point 43, the later record, shown in Fig. 3.1b, places $tram(a_1, m)$ and $bus(b_2, m)$ at time points 44 and 45, respectively. ∎

Due to our focus on the declarative semantics of stream reasoning, we will not elaborate on system aspects, such as update frequency of the stream, memory bottlenecks, system outages or out-of-order events [ABW06, DTM+13]. Instead, the aim here is to provide a starting point for formalizing an ideal semantics, from which a running system may have to diverge in practice, e.g., under heavy data load. We thus focus on the model-based aspect of stream reasoning, not architectures of potential systems.

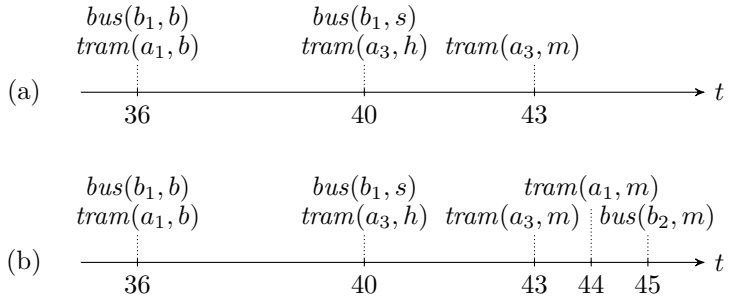


Figure 3.1: Two snapshots of a stream

Formalizing Streams

We use mutually disjoint finite sets of *predicates* $\mathcal{P}$, *constants* $\mathcal{C}$, *variables* $\mathcal{V}$ and *time variables* $\mathcal{U}$. The set $\mathcal{T}$ of *terms* is given by $\mathcal{C} \cup \mathcal{V}$ and the set $\mathcal{A}$ of *atoms* is defined as $\{p(t_1, \ldots, t_n) \mid p \in \mathcal{P}, t_1, \ldots, t_n \in \mathcal{T}\}$. The set $\mathcal{G}$ of *ground atoms* contains all atoms $p(t_1, \ldots, t_n) \in \mathcal{A}$ such that $\{t_1, \ldots, t_n\} \subseteq \mathcal{C}$. We also say a term is ground if it is a constant.

We divide $\mathcal{P}$ into two disjoint subsets, namely the *extensional predicates* $\mathcal{P}^{\mathcal{E}}$ and the *intensional predicates* $\mathcal{P}^{\mathcal{I}}$. Accordingly, we distinguish *extensional atoms* $\mathcal{A}^{\mathcal{E}}$ and *intensional atoms* $\mathcal{A}^{\mathcal{I}}$. Intensional predicates/atoms are used to express inferred information. On the other hand, extensional predicates (respectively atoms) are further partitioned into $\mathcal{P}^{\mathcal{E}}_B$ (respectively $\mathcal{A}^{\mathcal{E}}_B$) for *background data* and $\mathcal{P}^{\mathcal{E}}_S$ (respectively $\mathcal{A}^{\mathcal{E}}_S$) for *data streams*. The mentioned partitions are analogously defined for ground atoms $\mathcal{G}^{\mathcal{I}}$, $\mathcal{G}^{\mathcal{E}}_B$ and $\mathcal{G}^{\mathcal{E}}_S$. Unless stated otherwise, we use (with slight abuse of notation) the symbol $\mathcal{G}$ to refer to $\mathcal{G}^{\mathcal{I}} \cup \mathcal{G}^{\mathcal{E}}_S$, i.e., ground atoms that are not reserved for background data.

Additionally, we assume basic arithmetic operations ($+$, $-$, $\times$, $\div$) and comparisons ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$) are predefined by designated atoms $\mathcal{B} \subseteq \mathcal{A}^{\mathcal{E}}_B$, and written in infix notation. For instance, "$+$" can be realized by a set of atoms of form $p^+(X, Y, Z)$, where p^+ is a designated predicate symbol for "$+$" and X, Y and Z are integers such that $X + Y = Z$; the latter expression is then used in rules but viewed as syntactic sugar for $p^+(X, Y, Z)$. Whether an arithmetic expression holds is then determined by the existence of the according atom in the background data. Such built-in atoms are also used by practical ASP solvers such as dlv [LPF+06, ACD+17].

We now present our formalization of streams, which we base on the linear time ontology $\langle \mathbb{N}, \leq \rangle$; informally, the increase by 1 is the passing of time in terms of a tick by a global system clock.

Definition 1 (Stream) *Let T be a closed nonempty interval in $\mathbb{N}$ and $\upsilon \colon \mathbb{N} \to 2^{\mathcal{G}}$ an* evaluation function *such that $\upsilon(t) = \emptyset$ for all $t \in \mathbb{N} \setminus T$. Then, the pair $S = (T, \upsilon)$ is called a* stream*, T is the* timeline *of S, and the elements of T are* time points*.*

We also write evaluation functions as sets of nonempty mappings. For instance, $\{17 \mapsto \{a, b\}\}$ assigns $\{a, b\}$ to time point 17, and $\emptyset$ else.

Note that our stream definition remains agnostic about the specific meaning of time points. For instance, [SW04] and [DTM+13] distinguish the inherent time of a signal (that might be provided by the data source), called the *application time*, from the instant where a signal becomes available for processing, called the *system time*. DyKnow [HD04] and the recent Apache Flink framework [CKE+15] distinguish three notions of time: the *event time* refers to the creation of the signal at its producer, the *ingestion time* is the timestamp that Flink creates when it receives the signal, and the *processing time* is the local time at an internal operator when the signal is actually processed.

These distinctions are useful for the study and improvement of execution models in stream processing, which are the concern of those works. Our focus, however, shifts to the pure declarative semantics of stream reasoning, which we view in full separation from different possibilities of their computation. In that regard, issues about wrong output due to lags between event time and processing time, delays during computation, etc., are out of our scope. We regard the result of undesired semantic effects of system specifics as deviations from an ideal semantics; we aim here to formalize only the latter.

In formal analysis, time points might refer to all conceptual notions of time mentioned above. In an implemented system, time points will naturally reflect system time, respectively ingestion time (in the granularity of the system). Nevertheless, an explicit distinction between application time and system time can be obtained by means of explicit timestamps as usual.

Example 8 Consider the stream from Fig. 3.1 with the data up to time point 40. We can model the input as the data stream $D = (T, \upsilon)$, with a timeline T that covers the interval $[36, 40]$; we take $T = [0, 50]$, and the evaluation

$$\begin{aligned} \upsilon(36) &= \{tram(a_1, b), bus(b_1, b)\}, \\ \upsilon(40) &= \{tram(a_3, h)\}, \text{ and} \\ \upsilon(t) &= \emptyset \text{ for all } t \in T \setminus \{36, 40\}. \end{aligned}$$

With the mapping notation, we simply write $\{36 \mapsto \{tram(a_1, b), bus(b_1, b)\},\ 40 \mapsto \{tram(a_3, h)\}\}$. In our use case, $40 \mapsto \{tram(a_3, h)\}$ can mean that the tram with identifier a_3 actually appeared at station h at time point 40, or that the processing system received the information about this appearance at time point 40. Assuming the latter, we can still distinguish the two, if the application time is sent explicitly. For instance, the case that the tram appeared one minute (time point) earlier than recognized in the system would be reflected by the mapping $40 \mapsto tram(a_3, h, 39)$. ∎

Keeping the entire history of a stream is rarely practical, and often not useful: hardware restrictions and storage capabilities aside, many use cases on streaming data are intrinsically only concerned about recent snapshots of data. This leads to the notion of windows.

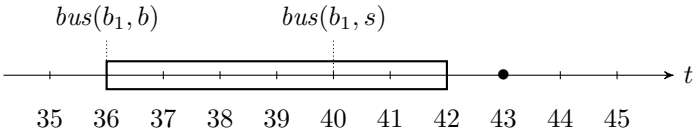


Figure 3.2: A window of the stream in Fig. 3.1. It is obtained at $t = 43$ (bullet), covers time points 36 to 42 (rectangle) and contains the bus signals at 36 and 40.

3.1.2 Windows

Central to the view on stream reasoning adopted in this work is the notion of a *window*, which typically is a recent, small selection of continuously streaming data. Restricting access to streams in form of windows may be a practical necessity due to limitations in computational resources, i.e., storage capacity or evaluation times of posed queries. Windows define which data has to be retained, respectively what can be deleted or forgotten. In that regard, a window separates data into a relevant part used as input for reasoning, and an irrelevant part that is ignored. Many stream processing use cases need this distinction already at the semantic level, in particular when only recent information is of interest. For instance, for the purpose of route planning only current (and predictable) traffic jams are of relevance, not those in the past.

Since our formalization of streams uses finite timelines, we can view windows simply as substreams., i.e., a window is a stream that is contained in another one.

Definition 2 (Window) *Let $S = (T, \upsilon)$ and $S' = (T', \upsilon')$ be two streams such that $T' \subseteq T$ and $\upsilon'(t') \subseteq \upsilon(t')$ for all $t' \in T'$. We then say that S' is a* substream *or* window *of S, denoted by $S' \subseteq S$.*

Furthermore, the *count* $|S|$ of stream $S = (T, \upsilon)$ is defined by $\#\{(a, t) \mid t \in T, a \in \upsilon(t)\}$ (where $\#$ stands for set cardinality), i.e., the total number of atom occurrences. In addition, we define the *size* of S by $|S| + \#\{t \in T \mid \upsilon(t) = \emptyset\}$, which also accounts for empty time points. The *restriction $S|_{T'}$ of S to $T' \subseteq T$* is the stream $(T', \upsilon|_{T'})$, where $\upsilon|_{T'}$ restricts the domain of υ to T', i.e., $\upsilon|_{T'}(t) = \upsilon(t)$ for all $t \in T'$, else $\upsilon|_{T'}(t) = \emptyset$. A *data stream* contains only extensional atoms, which we refer to as *signals*, i.e., the set $\mathcal{G}^{\mathcal{E}}_S$.

Example 9 (cont'd) Fig. 3.2 shows a window of stream D which selects at time $t = 43$ (indicated by the bullet) the timeline $T' = [36, 42]$ (rectangle) and only the bus appearances in this interval. Consequently, the window is given by $D' = (T', \upsilon')$, where $\upsilon' = \{36 \mapsto \{bus(b_1, b)\}, 40 \mapsto \{bus(b_1, s)\}\}$. Note further that the stream Fig. 3.1a is a window of Fig. 3.1b, assuming a shared timeline (e.g. $[0, 50]$). ∎

Towards a generic framework we need an abstract mechanism for the selection of windows, given a stream and a time point.

Definition 3 (Window function) *Any (computable) function w that returns, given a stream $S = (T, \upsilon)$, and a time point $t \in \mathbb{N}$, a window S' of S, is called a* window function.

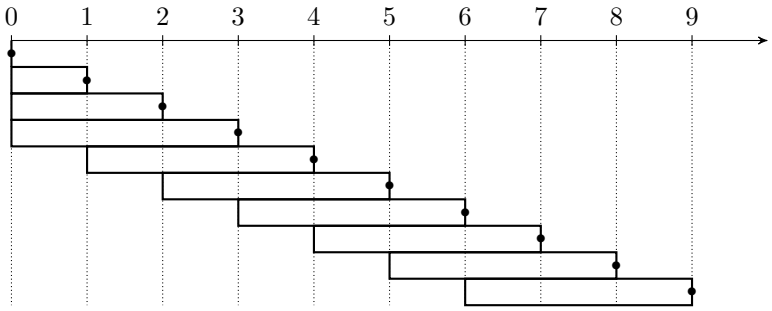


Figure 3.3: Progression for τ^3: *sliding* time-based window of size 3. Bullets indicate the reference time points at which the function is applied to obtain the window (rectangle).

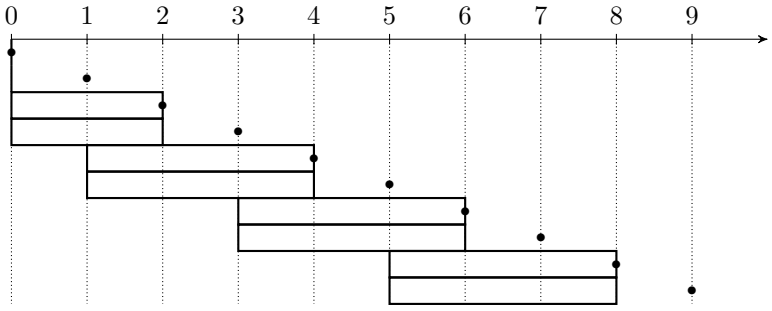


Figure 3.4: Progression for $\tau^{3(2)}$: *hopping* time-based window of size 3 and hop size 2.

We call the time point t at which a window function is applied the *reference time point*. The most common types of windows in practice include time-, tuple-, and partition-based windows, cf. [SW04, ABW06, DTM+13]; they will be presented in Sections 3.1.3-3.1.5. We associate them with three window functions symbols τ, #, and p, respectively. Traditionally [ABW06], these window functions take a fixed size ranging back in time from a reference time point t; we will consider in Section 3.1.7 generalized variants which allow for looking back and forth from t. Moreover, we introduce in Section 3.1.6 a *filter* window function f which only drops data but retains the current timeline.

3.1.3 Time-based Window

Window functions which select recent data based on a fixed amount of time are presumably the most widely used in practice. We formalize such *time-based windows* as follows.

Definition 4 (Time-based window) *Let $S = (T, \upsilon)$ be a stream, $T = [t_{min}, t_{max}]$ and $t \in \mathbb{N}$. Furthermore, let $n \in \mathbb{N} \cup \{\infty\}$ and $d \in \mathbb{N}$ such that $1 \leq d \leq n$. If $t \in T$, the* time-based window function *of* size n *and* hop size d *of S at time t is defined by*

$$\tau^{n(d)}(S, t) = (T', \upsilon|_{T'}),$$

where $T' = [t_\ell, t_u]$, $t_\ell = \max\{t_{min}, t_u - n\}$ and $t_u = \lfloor \frac{t}{d} \rfloor \cdot d$. If $t \notin T$, we define $\tau^{n(d)}(S, t) = S$.

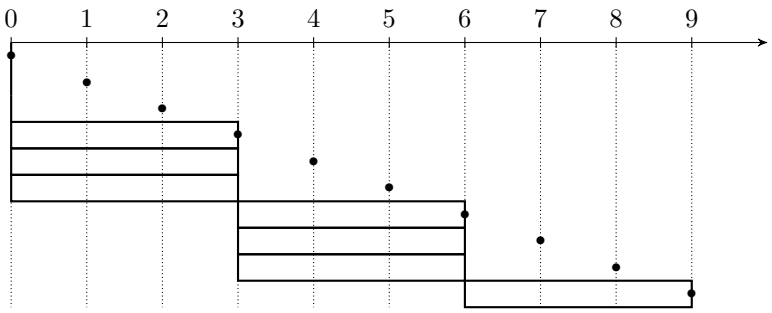


Figure 3.5: Progression for $\tau^{3(3)}$: *tumbling* time-based window of size 3.

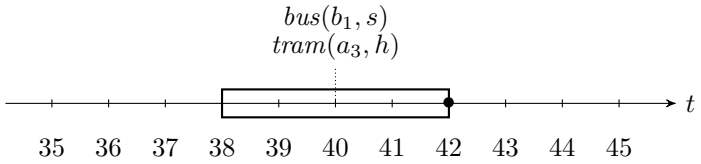


Figure 3.6: Sliding time-based window of size 4 at $t = 42$

Intuitively, a time-based window function selects, at time point t, a *pivot point* $t_u \leq t$ that is the closest time point to t such that the distance to the left end of the timeline is a multiple of d. From this time point t_u, the window reaches back (at most) n time points and selects all atoms in that interval. Note that the case for $t \notin T$ is given only for formal reasons, i.e., compliance with Definition 3. Conceptually, the time-based window is only applicable if $t \in T$. The general approach is useful for a compositional approach as discussed later.

We call a time-based window function $\tau^{n(d)}$ *sliding*, if $d = 1$, and *tumbling* if $d > 1$ and $d = n$, else *hopping*. The sliding window function $\tau^{n(1)}$ is abbreviated by τ^n. Figures 3.3-3.5 illustrate these different modes of window movement. First, Fig. 3.3 depicts a sliding window of size $n = 3$, where from the reference time t (indicated by a bullet), always window $[t - n, t]$ is selected, unless the left end t_ℓ has to be cut when it reaches the global limit $t_\ell = 0$. For other progressions, the right end of the window does not necessarily equal the reference time. If the window shifts in bigger intervals ($1 < d < n$), we get a hopping window as shown in Fig. 3.4. Here, the hop size is 2, i.e., the window shifts to the right every 2 time points. If the hop size equals the window size, we get a tumbling window as shown in Fig. 3.5. Sliding windows can be seen as a means for a fully continuous evaluation of recent events. Hopping windows additionally allow one to specify a certain interval after which a re-evaluation is needed. This is of interest when we want to control the time when a result shall be refreshed, e.g., a condition over the last 60 seconds which shall be recomputed only every 15 seconds. Finally, the tumbling window is suitable when we want to partition the timeline, e.g., to evaluate something in fixed intervals of 60 seconds.

Example 10 (cont'd) On the data stream D of Example 8, consider a monitoring use case where we want to know only bus and tram appearances reported within the last 4 minutes, at every minute. To this end, we can use a sliding time-based window function τ^4. Applying it on D at $t = 42$ gives $\tau^4(D, 42) = ([38, 42], \upsilon')$, where $\upsilon' = \{40 \mapsto \{tram(a_3, h), bus(b_1, s)\}\}$, as shown in Fig. 3.6. Using a hop size of $d = 2$ or $d = 3$ would result in the same window (at $t = 42$); the tumbling window function ($n = d = 4$) would select timeline $[36, 40]$. Note that the time-based window does not drop any data in the selected interval. ∎

Arguably, time-based windows are the most important ones in practice. The second most important class is dual: as one can limit resources typically by time or space, counting tuples (atoms) is the second natural approach for selecting recent chunks of data. We shall look at this tuple-based window next.

3.1.4 Tuple-based Window

Complementary to time-based windows, *tuple-based windows* define recency of data not in terms of time, but in terms of tuple count. Accordingly, they are also called *count windows* [GÖ10, CKE+15]. A tuple-based window of size n simply returns the most recent n atoms, regardless of their temporal appearance. Practically, one can view it as a buffer that always contains the n most recent atoms.

There are different reasons for using tuple-based windows. First, when reasoning about as much of historic data as possible, tuple-based windows may serve to define a threshold to address memory limits. Secondly, many use cases are intrinsically counting-based, where tuple-based windows can then select the relevant number of atoms. Typical examples in stream processing would be aggregations of recent signals that are received aperiodically from some sensor; for instance to decide the likelihood of a traffic jam by computing the average velocity of the last 50 cars. Another frequent use of tuple-based windows concerns selecting only the most recent atom of a stream, e.g., the current value of a car's temperature sensor to control the air conditioning. We now formally define tuple-based windows.

Definition 5 (Tuple-based window) *Let $S = (T, \upsilon)$ be a stream, $T = [t_{min}, t_{max}]$, $n \in \mathbb{N} \cup \{\infty\}$ and $t \in \mathbb{N}$. Furthermore, let $T' = [t_\ell, t]$, where*

$$t_\ell = \max(\{t_{min}\} \cup \{t' \mid t_{min} \leq t' \leq t \wedge |S|_{[t',t]}| \geq n\})$$

is called the left point*. If $t \in T$, a multi-valued* tuple-based window *of* size *n of S at time t is defined by*

$$\#^n(S, t) = (T', \upsilon'|_{T'}),$$

where

$$\upsilon'(t') = \begin{cases} \upsilon(t') & \text{if } t' \in T' \setminus \{t_\ell\} \\ X & \text{if } t' = t_\ell, \end{cases}$$

and $X \subseteq \upsilon(t_\ell)$ such that $|(T', \upsilon'|_{T'})| = n$. If $t \notin T$, we define $\#^n(S, t) = S$.

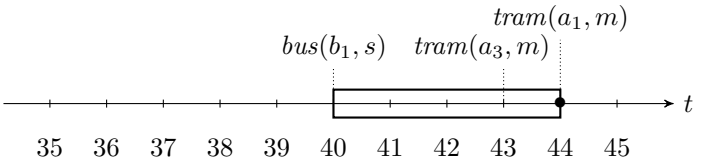


Figure 3.7: Tuple-based window of size 3 at $t = 44$ (one of two possibilities)

Intuitively, a tuple-based window of size n is obtained by searching back from current time point t until a time point t_ℓ is reached such that the total number of atoms counted in the interval $[t_\ell, t]$ is at least n. Then, in general, only a portion $X \subseteq \upsilon(t_\ell)$ of the atoms at the left point t_ℓ can be retained to achieve the exact tuple count n. Note that, in case the selection $X = \upsilon(t_\ell)$ can be guaranteed (as e.g. for streams with at most one atom per time point), the window definition simplifies to $(T', \upsilon|_{T'})$ as for time-based windows; then only the left point t_ℓ needs to be computed to determine $T' = [t_\ell, t]$. In general, however, multiple options exist for set X, making the tuple-based window function multi-valued. Practically, tuple-based windows can be assumed to be deterministic due to the specific implementation. In the sequel, when using a tuple-based window function, we always assume some single-valued refinement.

Example 11 To illustrate tuple-based windows, we use the stream from Fig. 3.1b, formalized by $D = (T, \upsilon)$ where $T = [0, 50]$ and

$$\upsilon = \left\{ \begin{array}{l} 36 \mapsto \{tram(a_1, b), bus(b_1, b)\}, \\ 40 \mapsto \{tram(a_3, h), bus(b_1, s)\}, \\ 43 \mapsto \{tram(a_3, m)\}, \\ 44 \mapsto \{tram(a_1, m)\}, \\ 45 \mapsto \{bus(b_2, m)\} \end{array} \right\}.$$

To get the last three vehicle appearances w.r.t. reference time point $t = 44$, we can use the tuple-based window function $\#^3$, which formally gives two possible windows at t: $S_1 = (T_1, \upsilon_1)$ and $S_2 = (T_2, \upsilon_2)$, where for both $j \in \{1, 2\}$, $T_j = [40, 44]$,

$$\upsilon_j(43) = \{tram(a_3, m)\},$$
$$\upsilon_j(44) = \{tram(a_1, m)\};$$

$\upsilon_1(40) = \{bus(b_1, s)\}$ and $\upsilon_2(40) = \{tram(a_3, h)\}$. That is to say, the two windows differ in the evaluation at time point 40, where a nondeterministic choice is made to pick exactly three atoms from the input stream from time point 44 back to 40. Window S_1 is shown in Fig. 3.7. ∎

There are two natural possibilities to enforce the uniqueness of a tuple-based window in practice. First, if there is a total order over all atoms, one can give a deterministic definition of the set X in Definition 5. Second, one may omit the requirement that *exactly* n atoms are contained in the window, but instead demand the substream obtained by the smallest interval $[t_\ell, t]$ containing *at least* n atoms.

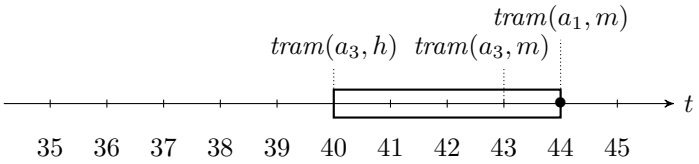


Figure 3.8: Partition-based window for the last 3 trams at $t = 44$

Note that our definition of tuple-based windows leads to a sliding progression in the sense that the right end of the resulting window's timeline always is the reference time point. An extension with a hop size (analogous to that of the time-based window) is possible, yet less used in practice. Another refinement is more important; it concerns more fine-grained control of the counting mechanism, as we will introduce next.

3.1.5 Partition-based Window

The previous section presented tuple-based windows which select a given number of the most recent atoms. We now introduce a refinement which allows one to select different numbers of different atoms at the same time. The *partitioned-based window function* first separates virtual substreams and then obtains tuple-based windows (with different sizes) on them. Finally, their union is the resulting window. The prototypical application uses only two partitions: relevant atoms to be counted, and irrelevant atoms that shall be ignored.

Example 12 (cont'd) Example 11 used a tuple-based window for the last three vehicle signals. Assume now that we are only interested in trams. This would amount to partitioning the data stream D into a virtual substream D_1 that contains only tram appearances (i.e., all atoms with predicate *tram*) and another one, D_2, for bus appearances. Assuming access to these substreams, we can apply a tuple-based window of size 3 on D_1, another one of size 0 on D_2,[1] and then merge the result. At $t = 44$ this would result in the window as depicted in Fig. 3.8, i.e., the alternate tuple-based window S_2 from before. ∎

Partition-based windows also provide a useful link between physical data streams and our formalization in Definition 1. In contrast to the latter, streaming applications usually combine multiple input sources. For instance, it is likely that tram signals and bus signals stem from different streams, or even that each vehicle sends its position in its own stream. Our focus, however, is not on processing the interplay of different streams but logic-oriented reasoning on top of their cumulative information. We therefore employ a single stream S which abstracts away different physical origins of data. Nevertheless, we can study different input streams in the formalism by selecting virtual substreams of S; partition-based windows are one possibility to obtain them.

[1] Note that for $t \in T$ and $n = 0$ the tuple-based window is $([t, t], \emptyset)$.

To define partition-based windows, we make use of a finite *index set* $I \subset \mathbb{N}$ to tag atoms: we use an *index function* $\mathrm{idx} : \mathcal{G} \to I$ *for* I which associates each ground atom $a \in \mathcal{G}$ with an *index* $\mathrm{idx}(a)$. Thus, given a stream $S = (T, \upsilon)$, each index $i \in I$ *induces* a substream $S_i = (T, \upsilon_i)$, where $\upsilon_i(t) = \{a \in \upsilon(t) \mid \mathrm{idx}(a) = i\}$. Moreover, we utilize a *tuple size function* $n : I \to \mathbb{N}$ *for* I that defines the size $n(i)$ for the tuple-based window on substream S_i. Naturally, we define the *union* $\upsilon = \upsilon_1 \cup \upsilon_2$ of two evaluation functions υ_1 and υ_2 by $a \in \upsilon(t)$ iff $a \in \upsilon_1(t) \cup \upsilon_2(t)$ for all $a \in \mathcal{G}$ and $t \in \mathbb{N}$; the *union* of two streams $S_1 = ([\ell_1, u_1], \upsilon_1)$ and $S_2 = ([\ell_2, u_2], \upsilon_2)$ is then defined by $S_1 \cup S_2 = ([\min\{\ell_1, \ell_2\}, \max\{u_1, u_2\}], \upsilon_1 \cup \upsilon_2)$.

Definition 6 (Partition-based window) *Let $S = (T, \upsilon)$ be a stream and $t \in \mathbb{N}$. Moreover, let* idx *be an index function and n be a tuple size function for an index set I. If $t \in T$, the* partition-based window *of S at time t (relative to* idx, n*) is defined by*

$$\mathrm{p}^{\mathrm{idx},n}(S, t) = \bigcup_{i \in I} \#^{n(i)}(S_i, t).$$

If $t \notin T$, we define $\mathrm{p}^{\mathrm{idx},n}(S, t) = S$.

With this definition in place, we continue with the details for the previous example.

Example 13 (cont'd) To define a window that selects the last three trams, we use an index set $I = \{1, 2\}$, where index 1 is used for trams and 2 for buses. Accordingly, we let $X = \{tram(c_1, c_2) \mid c_1, c_2 \in \mathcal{C}\}$ be the set of possible ground atoms with predicate *tram*, and define for all atoms $a \in \mathcal{G}$

$$\mathrm{idx}(a) = \begin{cases} 1 & \text{if } a \in X, \\ 2 & \text{else.} \end{cases}$$

This induces the substreams $D_1 = (T, \upsilon_1)$ and $D_2 = (T, \upsilon_2)$, where

$\upsilon_1 = \{36 \mapsto \{tram(a_1, b)\}, 40 \mapsto \{tram(a_3, h)\}, 43 \mapsto \{tram(a_3, m)\}, 44 \mapsto \{tram(a_1, m)\}\}$;
$\upsilon_2 = \{36 \mapsto \{bus(b_1, b)\}, 40 \mapsto \{bus(b_1, s)\}, 45 \mapsto \{bus(b_2, m)\}\}$.

We employ a tuple size function n such that $n(1) = 3$ and $n(2) = 0$, i.e., we will select three atoms from D_1 and none from D_2. At $t = 44$, this yields the windows $D' := \#^{n(1)}(D, t) = ([40, 44], \upsilon_1')$, where

$$\upsilon_1' = \{40 \mapsto \{tram(a_3, h)\}, 43 \mapsto \{tram(a_3, m)\}, 44 \mapsto \{tram(a_1, m)\}\},$$

and $\#^{n(2)}(D, t) = ([43, 43], \emptyset)$, respectively. Their union equals D', i.e., $\mathrm{p}^{\mathrm{idx},n}(D, t) = D'$. ∎

We present another example, which concerns the tracking of each individual vehicle.

Example 14 (cont'd) We are now interested in selecting the last appearance for each individual tram and bus. Consequently, we need a substream for each vehicle, then we can select the last atom on each of these substreams. We assume that all potential tram

instances can have identifiers a_1 to a_{10}, likewise buses b_1 to b_{50}, and there might be signals with predicates other than *tram* and *bus*. For the latter, we reserve index 0 by defining simply

$$\mathrm{idx}(g) = 0 \text{ for all } g \in \mathcal{G} \setminus \{tram(c_1, c_2), bus(c_1, c_2) \mid c_1, c_2 \in \mathcal{C}\}.$$

Next we assign $i = 1, \ldots, 10$ to tram identifiers and $i = 101, \ldots, 150$ to buses:

$$\begin{aligned} &\mathrm{idx}(tram(a_i, c)) = i \quad \text{for all } c \in \mathcal{C} \\ &\mathrm{idx}(bus(b_i, c)) = i + 100 \quad \text{for all } c \in \mathcal{C} \end{aligned}$$

This gives us four non-empty induced substreams for D at 45:

$$\begin{aligned} D_1 &= ([36, 45], \{36 \mapsto \{tram(a_1, b)\}, 44 \mapsto \{tram(a_1, m)\}\}) \\ D_3 &= ([40, 45], \{40 \mapsto \{tram(a_3, h)\}, 43 \mapsto \{tram(a_3, m)\}\}) \\ D_{101} &= ([36, 45], \{36 \mapsto \{bus(b_1, b)\}, 40 \mapsto \{bus(b_1, s)\}\}) \\ D_{102} &= ([45, 45], \{45 \mapsto \{bus(b_2, m)\}\}) \end{aligned}$$

We use the tuple size function n such that $n(0) = 0$ and $n(i) = 1$ for all $i \in I \setminus \{0\}$ and thus obtain the following respective tuple-based windows (of size 1):

$$\begin{aligned} D'_1 &= ([44, 45], \{44 \mapsto \{tram(a_1, m)\}\}) \\ D'_3 &= ([43, 45], \{43 \mapsto \{tram(a_3, m)\}\}) \\ D'_{101} &= ([40, 45], \{40 \mapsto \{bus(b_1, s)\}\}) \\ D'_{102} &= ([45, 45], \{45 \mapsto \{bus(b_2, m)\}\}) \end{aligned}$$

The union of these windows gives us the result of the partition-based window:

$$\mathrm{p}^{\mathrm{idx},n}(D, 45) = ([40, 45], \upsilon'),$$

where

$$\upsilon' = \{40 \mapsto \{bus(b_1, s)\}, 43 \mapsto \{tram(a_3, m)\}, 44 \mapsto \{tram(a_1, m)\}, 45 \mapsto \{bus(b_2, m)\}\}. \quad \blacksquare$$

The previous example indicates the power of partitioned-based windows that arises from its flexible selection mechanism. Applications may define more narrow variants which can then be uniformly represented by index functions. For formal analysis, however, a more compositional approach is desirable that does not hide multiple computation steps in a single function. For the use case illustrated in Examples 12 and 13 we suggest an alternative approach, where the tuple-based window follows an initial projection step. To this end, we now introduce filter windows.

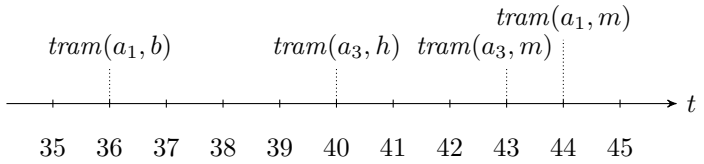


Figure 3.9: Filter window for the set A of tram occurrences

3.1.6 Filter Window

Time-based windows and tuple-based windows shrink the timeline and thus drop data outside their defined scope. Dually, *filter* windows leave the timeline as it is and drop all but a specified set of atoms. Accordingly, given a stream $S = (T, \upsilon)$, we define for a set $A \subseteq \mathcal{G}$ of ground atoms, the *projection* of υ to A by $\upsilon|_A(t) = \upsilon(t) \cap A$ for all $t \in \mathbb{N}$, and the *projection* of S to A by $S|_A = (T, \upsilon|_A)$. This already yields the filter window.

Definition 7 (Filter window) *Let $S = (T, \upsilon)$ be a stream, $t \in \mathbb{N}$, and $A \subseteq \mathcal{G}$ be a set of atoms. The* filter window function *for A (at time t) is defined by*

$$\mathrm{f}^A(S, t) = S|_A.$$

Note that the filter window function works in a time-independent manner, i.e., it always retains the timeline and returns the same result for all $t \in \mathbb{N}$, in particular, for $t \notin T$. This is dual to the time-based window, where all atoms in the selected timeline can remain, and this timeline is selected independently from atoms.

Filter windows are particularly useful as preprocessing step before applying tuple-based windows which then only count a defined subset of atoms.

Example 15 (cont'd) In Examples 12 and 13 we selected the last three trams by means of a partition-based window. Alternatively, we can use the set A of atoms with predicate *tram*, where we obtain the stream D_1 above, shown in Fig. 3.9, by $\mathrm{f}^A(D, t)$ (for any $t \in \mathbb{N}$). Consequently, this specific partition-based window is equally obtained by consecutive applications of f^A and $\#^3$; i.e., using idx, n from Example 13 we get $\mathrm{p}^{\mathrm{idx},n}(D, t) = \#^3(\mathrm{f}^A(D, t'), t)$ (for all $t, t' \in \mathbb{N}$).

Finally, we revisit Fig. 3.1. Using a set B of atoms with predicate *bus*, we obtain the depicted window by $\tau^{6(2)}(\mathrm{f}^B(D, t), 43)$, or equivalently, by $\mathrm{f}^B(\tau^{6(2)}(D, 43), t)$ (for any $t \in \mathbb{N}$); the hopping time-based window selects, at $t = 43$, the timeline $[36, 42]$, and the filter window retains only bus signals. Due to the logical independence between these functions, we can apply them in arbitrary order. ∎

In contrast to the windows studied above filtering also affects atoms appearing *after* the time point t where function f^A is applied. We will now introduce generalizations of the other window functions that may also access future time points.

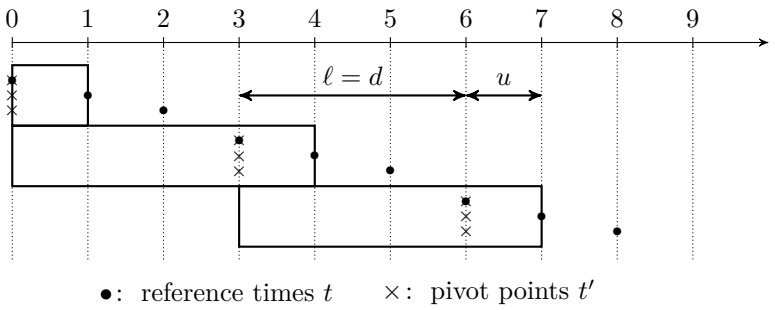


Figure 3.10: Progression for $\tau^{3,1(3)}$: generalized (hopping) time-based window

3.1.7 Windows with Access to the Future

Standard window functions select substreams that contain recent data from the past. Time-based windows select an interval $[t_1, t_2]$, where t_2 is close or equal to reference time t, tuple-based windows and partition-based windows determine an interval $[t_1, t]$ by counting atoms. We will now consider generalizations where the timeline can also extend to time points after t. One motivation for such future windows concerns predictions, i.e., inferred information about future events. Windows accessing later time points can then restrict access to intensional atoms assigned to them.

Example 16 (cont'd) When Bob is approaching Mozart Circus (station m) on line ℓ_3 at time $t = 42$, he wants to know whether there will be a connecting tram. For this, he is only interested in expected arrival times of trams within the next 5 minutes. These predictions could be reflected by intensional atoms of form $exp(Id, St)$, where Id is the tram identifier and St is the station where the tram is expected. We thus would like to have a time-based window that drops all such atoms relating to time points *after* $t + 5$. ■

Having access to time points after the reference time point is also useful when we navigate within a timeline. Our forthcoming framework will introduce temporal modalities, which open up the possibility to apply windows at multiple time points within the same evaluation. This shifts the conceptual perspective from "past" and "future" to "before" and "after" relative to a reference time point that can be controlled. In that regard, it is natural to lift the time-based window accordingly.

Definition 8 (Generalized time-based window) *Let $S = (T, \upsilon)$ be a stream, $T = [t_{min}, t_{max}]$ and let $t \in \mathbb{N}$. Furthermore, let $\ell, u \in \mathbb{N} \cup \{\infty\}$ and $d \in \mathbb{N}$ such that $d \leq \ell + u$. If $t \in T$, the generalized* time-based window function *with* range (ℓ, u) *and* hop size d *of S at time t is defined by*

$$\tau^{\ell,u(d)}(S, t) = (T', \upsilon|_{T'}) \,,$$

where $T' = [t_\ell, t_u]$, $t_\ell = \max\{t_{min}, t' - \ell\}$ with pivot point $t' = \lfloor \frac{t}{d} \rfloor \cdot d$, $t_u = \min\{t' + u, t_{max}\}$. *If $t \notin T$, we define $\tau^{\ell,u(d)}(S, t) = S$.*

The generalized time-based window replaces in Definition 4 the size n by a range (ℓ, u): in addition to selecting the ℓ time points left of the pivot point, it also selects u time points right of it. Accordingly, the *size* of the generalized time-based window (function) $\tau^{\ell,u(d)}$ is given by $\ell + u$. When we use time-based windows with future time points (after t) in the sequel, we will implicitly refer to the one of Definition 8.

As before, a time-based window function $\tau^{\ell,u(d)}$ is called *sliding*, if $d = 1$, and *tumbling* if $d > 1$ and $d = \ell + u$, else *hopping*. We abbreviate $\tau^{\ell,u(1)}$ by $\tau^{\ell,u}$, $\tau^{\ell,0(1)}$ by τ^{ℓ} and $\tau^{0,u(1)}$ by τ^{+u}.

Example 17 Fig. 3.10 shows the selection mechanism of a hopping (generalized) time-based window with range $(3, 1)$ and hop size 3. Consider for instance reference time point $t = 7$. Function $\tau^{3,1(3)}$ selects pivot time $t' = \lfloor \frac{7}{3} \rfloor \cdot 3 = 6$, from which it reaches by $t' - \ell$ time point 3 for the lower end, and by $t' + u$ time point 7 for the upper end. Consequently, the resulting window has the timeline $[3, 7]$. ∎

The case for tuple-based windows is analogous: we want to be able to select a certain number of tuples before and after a reference time point, where the window is applied. Accordingly, we generalize the tuple-based window as follows.

Definition 9 (Generalized tuple-based window) *Let $S = (T, \upsilon)$ be a stream, $T = [t_{min}, t_{max}]$ and $t, \ell, u \in \mathbb{N}$. We define the* left point t_ℓ and *right point t_u* by

$$t_\ell = \max(\{t_{min}\} \cup \{t' \mid t_{min} \leq t' \leq t \wedge |S|_{[t',t]}| \geq \ell\}), \text{ and}$$
$$t_u = \min(\{t_{max}\} \cup \{t' \mid t \leq t' \leq t_{max} \wedge |S|_{[t+1,t']}| \geq u\}),$$

respectively. Let $T_\ell = [t_\ell, t]$ and $T_u = [t + 1, t_u]$. If $t \in T$, a multi-valued generalized tuple-based window *with* range (ℓ, u) *of S at time t is defined by*

$$\#^{\ell,u}(S, t) = (T', \upsilon'|_{T'}),$$

where $T' = [t_\ell, t_u]$,

$$\upsilon'(t') = \begin{cases} \upsilon(t') & \text{if } t' \in T' \setminus \{t_\ell, t_u\} \\ X_\ell & \text{if } t' = t_\ell \\ X_u & \text{if } t' = t_u, \end{cases}$$

and $X_q \subseteq \upsilon(t_q)$, $q \in \{\ell, u\}$, such that $|(T_q, \upsilon'|_{T_q})| = q$. If $t \notin T$, we define $\#^{\ell,u}(S, t) = S$.

This generalization of Definition 5 is straightforward: it adds to the selection of ℓ (n in Definition 5) atoms left of reference time t the selection of further u atoms right of t. We thus define the *size* of the generalized tuple-based window (function) $\#^{\ell,u}$ by $\ell + u$. As before, we abbreviate $\#^{0,u}$ by $\#^{+u}$.

We immediately get the definition of *generalized partition-based windows*, by allowing extended tuple-based windows in Definition 6 (using tuple size functions that range over $\mathbb{N} \times \mathbb{N}$).

Time-based:	$\boxplus^{\ell,u(d)} := \boxplus^{\tau^{\ell,u(d)}}$	$\boxplus^{\ell} := \boxplus^{\tau^{\ell}}$	$\boxplus^{+u} := \boxplus^{\tau^{+u}}$
Tuple-based:	$\boxplus^{\#\ell,u} := \boxplus^{\#^{\ell,u}}$	$\boxplus^{\#\ell} := \boxplus^{\#^{\ell}}$	
Partition-based:	$\boxplus^{\text{idx},n} := \boxplus^{\text{p}^{\text{idx},n}}$		
Filter:	$\boxplus^{A} := \boxplus^{\text{f}^{A}}$		

Figure 3.11: Overview of window operator shortcuts

Example 18 (cont'd) We now can determine the best connection at Mozart Circus. To this end, we use an index function idx that assigns index 1 to expected arrivals of trams at station m, i.e., the set A of atoms of form $exp(Id, m)$, and index 0 for all other atoms. Next, we define the tuple size function n such that $n(1) = (0, 1)$ (i.e., the parameters $\ell = 0$ and $u = 1$ for the generalized tuple-based window on substream S_1) and $n(0) = (0, 0)$. This results in a window that contains only the next tram at m. Alternatively, we get the same result by employing f^A followed by $\#^{+1}$. ∎

After having introduced streams, windows and different window functions, we are now prepared to propose a logic-based framework for stream reasoning.

3.2 The LARS Framework

We now present an improved version of LARS [BDEF15], a Logic for Analytic Reasoning over Streams. LARS extends propositional logic for streaming data by employing any window function w (Definition 3) in a window operator $\boxplus^w$. Within the resulting substream, one can then control the temporal modality of formulas, respectively access temporal information. Based on such formulas, LARS then provides a rule-based language with a model-based, nonmonotonic semantics, which can be seen as an extension of Answer Set Programming for streaming data.

Before defining syntax and semantics of LARS below, we first present the central concepts informally.

Window Operators & Stream Reset

If w is a window function, we call $\boxplus^w$ a *window operator*. Given a formula α, the expression $\boxplus^w\alpha$ has the effect that α will be evaluated on the window obtained by applying w in the current stream S at the current time t. Dually, the *reset operator* $\triangleright$ serves to re-access the original stream. For specific window operators we use syntactic shortcuts as listed in Fig. 3.11. We use the letter W to denote a set of window functions.

Temporal Modalities

Regardless if formula evaluation is on the entire input stream or a window thereof, we provide explicit means to deal with the temporal information. Let $S = (T, \upsilon)$ be a stream,

i.e., the input stream or a window, and $t \in T$ be a time point. There are different ways to evaluate a formula α in S at t. First, we express by $@_{t'}\alpha$, where $t' \in \mathbb{N} \cup \mathcal{U}$, that α has to hold when changing the evaluation time to t'. We call t' in $@_{t'}\alpha$ a *time pin*; which is *ground* if $t' \in \mathbb{N}$, else *non-ground*. Next, time might be abstracted away. That is to require that α holds at *some* time point $t' \in T$, denoted by $\Diamond\alpha$. Dually, $\Box\alpha$ shall hold iff α holds at *all* time points in T. Based on these modalities, we define our language.

3.2.1 LARS Formulas

Definition 10 (Formulas) *Let $a \in \mathcal{A}$ be an atom, $t \in \mathbb{N} \cup \mathcal{U}$ and w be a window function. The set $\mathcal{F}$ of* formulas *is defined by the following grammar:*

$$\alpha ::= a \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \alpha \rightarrow \alpha \mid \Diamond\alpha \mid \Box\alpha \mid @_t\alpha \mid \boxplus^w\alpha \mid \triangleright\alpha \tag{3.1}$$

The set $\mathcal{F}_\mathcal{G}$ of *ground formulas* contains all formulas where each term and each time pin is ground. In addition to streams, we consider background knowledge in form of static data, i.e., a set $B \subseteq \mathcal{G}_B^\mathcal{E}$ of ground atoms which does not change over time. From a semantic perspective, the difference to streams is that static data is always available, regardless of window applications.

The following definitions concern the semantics of ground formulas.

Definition 11 (Structure) *Let $S = (T, \upsilon)$ be a stream, W a set of window functions and $B \subseteq \mathcal{G}_B^\mathcal{E}$ a set of facts. Then, we call $M = \langle S, W, B\rangle$ a* structure*, S the* interpretation stream *and B the* background data *of M.*

We now define when a ground formula holds in a structure.

Definition 12 (Entailment) *Let $M = \langle S^\star, W, B\rangle$ be a structure, $S^\star = (T^\star, \upsilon^\star)$ and let $S = (T, \upsilon)$ be a substream of $S^\star$. Moreover, let $t \in T^\star$. The* entailment *relation $\Vdash$ between (M, S, t) and formulas is defined as follows. Let $a \in \mathcal{G}$ be an atom, let $\alpha, \beta \in \mathcal{F}_\mathcal{G}$ be ground formulas and $w \in W$. Then,*

$$\begin{array}{lcl}
M,S,t \Vdash a & :\Leftrightarrow & a \in \upsilon(t) \text{ or } a \in B,\\
M,S,t \Vdash \neg\alpha & :\Leftrightarrow & M,S,t \nVdash \alpha,\\
M,S,t \Vdash \alpha \wedge \beta & :\Leftrightarrow & M,S,t \Vdash \alpha \text{ and } M,S,t \Vdash \beta,\\
M,S,t \Vdash \alpha \vee \beta & :\Leftrightarrow & M,S,t \Vdash \alpha \text{ or } M,S,t \Vdash \beta,\\
M,S,t \Vdash \alpha \rightarrow \beta & :\Leftrightarrow & M,S,t \nVdash \alpha \text{ or } M,S,t \Vdash \beta,\\
M,S,t \Vdash \Diamond\alpha & :\Leftrightarrow & M,S,t' \Vdash \alpha \text{ for some } t' \in T,\\
M,S,t \Vdash \Box\alpha & :\Leftrightarrow & M,S,t' \Vdash \alpha \text{ for all } t' \in T,\\
M,S,t \Vdash @_{t'}\alpha & :\Leftrightarrow & M,S,t' \Vdash \alpha \text{ and } t' \in T,\\
M,S,t \Vdash \boxplus^w\alpha & :\Leftrightarrow & M,S',t \Vdash \alpha, \text{where } S' = w(S,t),\\
M,S,t \Vdash \triangleright\alpha & :\Leftrightarrow & M,S^\star,t \Vdash \alpha.
\end{array}$$

If $M, S, t \Vdash \alpha$ holds, we say that (M, S, t) *entails* α. Moreover, we say that M *satisfies* α *at time* t, if $(M, S^\star, t)$ entails α. In this case we write $M, t \models \alpha$ and call M a *model* of α

at time t. Satisfaction and the notion of a model are extended to sets of formulas as usual. Moreover, we define that for the special atoms $\top$ and $\bot$, $M, S, t \Vdash \top$ always and $M, S, t \Vdash \bot$ never holds.

Example 19 (cont'd) Let $D = (T, \upsilon)$ be the data stream of Example 10 and $S^\star = (T^\star, \upsilon^\star) \supseteq D$ be a stream such that $T^\star = T$ and

$$\upsilon^\star = \left\{ \begin{array}{ll} 36 \mapsto \{tram(a_1, b)\}, & 40 \mapsto \{tram(a_3, h),\} \\ 43 \mapsto \{exp(a_3, m)\}, & 44 \mapsto \{exp(a_1, m)\} \end{array} \right\}.$$

Let $M = \langle S^\star, W, B \rangle$, where $W = \{\tau^{+5}\}$, and B is an empty set of facts. Then $M, S^\star, 42 \Vdash \boxplus^{+5} \Diamond exp(a_3, m)$ holds: the window operator $\boxplus^{+5}$ selects $S' = (T', \upsilon')$, with timeline $T' = [42, 47]$ and $\upsilon' = \{43 \mapsto \{exp(a_3, m)\}, 44 \mapsto \{exp(a_1, m)\}\}$, i.e., there is some $t' \in T'$ ($t' = 43$) such that $M, S', t' \Vdash exp(a_3, m)$. ∎

We note that the original presentation of LARS [BDEF15] employed a so-called *stream choice* in window operators that allowed to direct the window function to be applied on the original stream $S^\star$ (stream choice 1) or the current stream S (stream choice 2). Definition 12 presents a cleaner approach, where a window operator is always applied on the current stream. In case the original stream needs to be re-accessed in nested windows, this can be done by an explicit reset step $\triangleright$, followed by a window operator $\boxplus^w$. We define this combination as *input window operator* $\boxtimes^w := \triangleright \boxplus^w$.

Example 20 Consider a monitoring use case where a signal s must always appear within 5 minutes. Testing whether this condition holds for the last hour amounts to the formula $\boxplus^{60} \Box \boxtimes^5 \Diamond s$: we first select by a sliding time-based window the last 60 minutes. At every time point in this window, it must hold that if we consider the last 5 minutes there, signal s holds at some time point. Notably, by using $\boxtimes^5$ instead of $\boxplus^5$ we ensure that this inner window reaches beyond the limits of the first. For instance, consider a stream $S = ([0, 500], \upsilon)$. First, at $t = 500$, $\boxplus^{60}$ selects $S' = (T', \upsilon|_{T'})$, where $T' = [440, 500]$. During evaluation of $\Box$ at $t' = 440$, $\boxtimes^5$ now selects $S|_{[435,440]}$, while $\boxplus^5$ would select $S'|_{[440,440]}$, since the timeline in S' only starts at 440. ∎

Another subtle improvement over the previous version [BDEF15] concerns the fact that a window operator $\boxplus^w$ may return a substream that does not contain the evaluation time point, i.e., $w(S, t) = (T', \upsilon')$ does not imply $t \in T'$. However, given a (sub)formula $\boxplus^w \varphi$, one typically wants to evaluate φ in the obtained window regardless of the specific evaluation time and its position relative to the window. While this could be technically handled for relevant cases, we now consider time points $t \in T^\star$, and not $t \in T$. That is to say, the evaluation time point t needs to be contained only in the global timeline $T^\star$, not in the timeline T of the current substream S.

Example 21 Consider again Fig. 3.5, which illustrates the progress of a tumbling time-based window of size 3, i.e., the function $\tau^{3(3)}$. Assume further we are interested whether an atom x occurs in this window when evaluated at time 8. Accordingly, we evaluate

$M, S, 8 \Vdash \boxplus^{3(3)} \Diamond x$ and the substream returned by $\tau^{3(3)}$ has timeline $[3, 6]$. We still expect that the entailment holds iff x appears within $[3, 6]$, regardless of the fact that $8 \notin [3, 6]$. ■

Notably, allowing any $t \in T^\star$ in the formula evaluation not only serves the applicability of windows such as hopping or tumbling windows. It also allows one to inspect whether the evaluation time is contained in the current timeline T. This possibility stems from the requirement that t' is contained in T for $@_{t'}\alpha$ to hold: the standard tautology $\top := a \vee \neg a$ holds (in all structures) at every time point $t \in T^\star$, where entailment is defined; however, $@_{t'}\top$ holds if and only if $t' \in T$. Since also time points $t \in T^\star \setminus T$ can be evaluated, formulas can express conditions based on T. For instance, $M, S, t \Vdash \boxplus^w(@_t\top \wedge \varphi)$ holds only if t is contained in the timeline of $w(S, t)$.

Queries & Non-ground Formulas

We now consider the use of variables, leading to open formulas and queries.

Definition 13 (Query) *Let $M = \langle S, W, B\rangle$ be a structure, $\alpha \in \mathcal{F}$ be a formula and let $u \in \mathbb{N} \cup \mathcal{U}$. Then, the tuple $Q = \langle M, u, \alpha\rangle$ is called a* query*. We say Q is* ground *if α and u are ground, else* non-ground*.*

Given a ground query $Q = \langle M, t, \alpha\rangle$, where $M = \langle S, W, B\rangle$, we define the *answer* $?Q$ to Q as *yes*, if $M, S, t \Vdash \alpha$ holds, else *no*.

To define the semantics of non-ground queries, we need the notions of a *substitution* σ, defined as mapping $\mathcal{V} \cup \mathcal{U} \to \mathcal{C} \cup \mathbb{N}$ that assigns (i) each variable $V \in \mathcal{V}$ a constant $\sigma(V) \in \mathcal{C}$, and (ii) each time variable $U \in \mathcal{U}$ a natural number $\sigma(U) \in \mathbb{N}$. The *grounding* $\sigma(\alpha)$ (respectively $\sigma(u)$) of formula α (respectively time pin u) due to σ is obtained by applying the substitutions on variables/time variables as usual. Given a timeline T, we say a substitution σ is *over* $(\mathcal{C}, T)$, if the image of σ is contained in $\mathcal{C} \cup T$; we denote by $\sigma(\mathcal{C}, T)$ the set of all such substitutions. With this, we define the following.

Definition 14 (Answer) *The* answer $?Q$ *to a non-ground query $Q = \langle M, u, \alpha\rangle$ is defined by*

$$?Q = \{\sigma \in \sigma(\mathcal{C}, T) \mid M, S, \sigma(u) \Vdash \sigma(\alpha)\}. \tag{3.2}$$

This definition gives a general semantics to two important subclasses of non-ground queries $Q = \langle M, u, \alpha\rangle$. First, if α is ground and $u \in \mathcal{U}$ is a time variable, then the answer to Q amounts to the time points when α holds. Dually, if $u \in \mathbb{N}$ and α is non-ground, we obtain a semantics for non-ground formula evaluation at a fixed time point.

For queries, the set of window functions W in a stated structure $M = \langle S, W, B\rangle$ is implicitly given by the window operators used in α. In case we make use of background knowledge (beyond implicit auxiliary atoms for arithmetic) the set B is defined explicitly.

Example 22 Consider again the stream from Fig. 3.1b, which we now formalize by $S = (T, \upsilon)$, where $T = [30, 50]$. We ask:

Q_1 At $t = 45$, which trams arrived at which stations in the last 5 minutes?

Q_2 At $t = 45$, at which times and which stations did tram a_3 arrive in the last 5 minutes?

Q_3 At which times did we record a tram arrival at a station, where a bus arrived within the next 3 minutes?

We formalize these queries as $Q_1 = \langle M, 45, \alpha_1 \rangle$, $Q_2 = \langle M, 45, \alpha_2 \rangle$, and $Q_3 = \langle M, U, \alpha_3 \rangle$, where

$$\alpha_1 = \boxplus^5 \Diamond tram(A, St),$$
$$\alpha_2 = \boxplus^5 @_U tram(a_3, St), \text{ and}$$
$$\alpha_3 = tram(A, St) \wedge \boxplus^{+3} \Diamond bus(B, St).$$

We obtain the following answers:

$$?Q_1 = \{\, \{A \mapsto a_3, St \mapsto h\}, \{A \mapsto a_3, St \mapsto m\}, \{A \mapsto a_1, St \mapsto m\} \,\}$$
$$?Q_2 = \{\, \{St \mapsto h, U \mapsto 40\}, \{St \mapsto m, U \mapsto 43\} \,\}$$
$$?Q_3 = \{\, \{A \mapsto a_1, St \mapsto b, U \mapsto 36, B \mapsto b_1\}, \{A \mapsto a_3, St \mapsto m, U \mapsto 43, B \mapsto b_2\},$$
$$\{A \mapsto a_1, St \mapsto m, U \mapsto 44, B \mapsto b_2\} \,\}$$ ∎

We observe that the operator @ allows for reconsidering a historic query. At any time $t' > t$, we can ask $\langle M, t', @_t \alpha \rangle$ to simulate a previous query $\langle M, t, \alpha \rangle$. In fact, this applies for any $t' \in \mathbb{N}$.

Example 23 (cont'd) Consider again Q_1 from Example 22, where α_1 is evaluated at $t = 45$, and let $\sigma \in ?Q_1$ be an answer. Then, for any time point $t' \in \mathbb{N}$, σ is also an answer of $\langle M, t', @_{45} \alpha_1 \rangle$, since by definition, $M, S, 45 \Vdash \sigma(\alpha_1)$ iff $M, S, t' \Vdash @_{45} \sigma(\alpha_1)$ and $45 \in T$. ∎

Nested Windows

Typically, window functions are used exclusively to restrict the processing of streams to a recent subset of the input. In our view, window functions provide a flexible means to discard data.

Example 24 (cont'd) Recall that we can select the last appearance of tram a_1 by first using a filter window function f^A for tram atoms $A = \{tram(a_1, c) \mid c \in \mathcal{C}\}$ followed by a tuple-based window. We now also ask at which time U the tram was last recorded. The according LARS query $Q = \langle M, 45, \alpha \rangle$ is given by $M = \langle D, \{\mathrm{f}^A, \#^1\}, \emptyset \rangle$ and $\alpha = \boxplus^A \boxplus^{\#1} @_U tram(a_1, St)$, which has the single answer $\sigma = \{St \mapsto m, U \mapsto 44\}$. That is, tram a_1 last appeared at station m at time 44. ∎

LARS formulas provide a powerful, flexible language to query streaming data. However, the formalism presented so far has no means of expressing auxiliary information, i.e., intensional atoms, and thus comes with limitations.

Example 25 Dual to query Q_3 in Example 22, we now want to ask for which tram appearances *no* bus arrived within 3 minutes at the same station. The intended answer to this query should only contain tram a_3 at station h at time 40. A naive translation simply adds negation to α_3, i.e., $\alpha_3' = \mathit{tram}(A, St) \wedge \neg \boxplus^{+3} \Diamond \mathit{bus}(B, St)$. However, the resulting query $Q_3' = \langle M, U, \alpha_3' \rangle$ expresses the following: at which time points U did some tram appear at station St, where within the next 3 minutes there was no bus arriving at St for any B, i.e., for any constant that can be substituted for B. Thus, whenever a tram a_i is at station st at time t, we hypothetically consider any atom $\mathit{bus}(x, St)$, where $x \in \mathcal{C}$, at time points $n = t,\ t+1,\ t+2,\ t+3$, and get an answer of form $\{A \mapsto a_i, St \mapsto st, B \mapsto x, U \mapsto t\}$, whenever $\mathit{bus}(x, st)$ is not in the evaluations from $v(t)$ to $v(t+3)$ in D. That results in answers like the following:

$$\begin{gathered}
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto a_1,\ U \mapsto 36\} \\
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto a_3,\ U \mapsto 36\} \\
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto b_2,\ U \mapsto 36\} \\
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto s,\ U \mapsto 36\} \\
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto h,\ U \mapsto 36\} \\
\vdots \\
\{A \mapsto a_3,\ St \mapsto h,\ B \mapsto b_1,\ U \mapsto 40\} \\
\vdots
\end{gathered}$$

Clearly, we can limit the scope of B by considering only constants that have been observed as bus identifiers so far, using $\alpha_3'' = \mathit{tram}(A, St) \wedge \neg \boxplus^{+3} \Diamond \mathit{bus}(B, St) \wedge \Diamond \mathit{bus}(B, St')$. The additional subformula $\Diamond \mathit{bus}(B, St')$ now matches all bus appearances throughout the stream and will thus be joined with every tram appearance $\mathit{tram}(A, St)$ (at time U). Semantically, this cross product yields potential answers of form $\{A \mapsto a_i, St \mapsto st, B \mapsto x, St' = st', U \mapsto t\}$, which are reduced by those entries for which $\mathit{bus}(x, st)$ appeared between t and $t+3$. For instance, $\{A \mapsto a_1, St \mapsto b, B \mapsto b_1, St \mapsto y, U \mapsto 36\}$, where $y \in \{b, s, m\}$, is not returned since $\mathit{bus}(b_1, b)$ appeared at 36. We get, among others, the following answers:

$$\begin{gathered}
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto b_2,\ St' \mapsto b,\ U \mapsto 36\} \\
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto b_2,\ St' \mapsto s,\ U \mapsto 36\} \\
\{A \mapsto a_1,\ St \mapsto b,\ B \mapsto b_2,\ St' \mapsto m,\ U \mapsto 36\} \\
\{A \mapsto a_3,\ St \mapsto h,\ B \mapsto b_1,\ St' \mapsto b,\ U \mapsto 40\} \\
\{A \mapsto a_3,\ St \mapsto h,\ B \mapsto b_1,\ St' \mapsto s,\ U \mapsto 40\} \\
\{A \mapsto a_3,\ St \mapsto h,\ B \mapsto b_1,\ St' \mapsto m,\ U \mapsto 40\} \\
\{A \mapsto a_3,\ St \mapsto h,\ B \mapsto b_2,\ St' \mapsto b,\ U \mapsto 40\} \\
\{A \mapsto a_3,\ St \mapsto h,\ B \mapsto b_2,\ St' \mapsto s,\ U \mapsto 40\} \\
\{A \mapsto a_3,\ St \mapsto h,\ B \mapsto b_2,\ St' \mapsto m,\ U \mapsto 40\} \\
\vdots
\end{gathered}$$

The first question is how the result shall be interpreted. The non-essential St' does not capture anything of the conceptual query, which does not talk about stations of arbitrary

bus stations. To remedy this, a simple post-processing may filter out such bindings and reduce resulting duplicates accordingly. However, this still leaves wrong results. For instance, the first three answers would reduce to $\{A \mapsto a_1, St \mapsto b, B \mapsto b_2, U \mapsto 36\}$. What this answer says is that bus b_2 did not arrive at station b within 3 minutes, but we intended to query for tram appearances after which *no* bus arrived. ∎

The fundamental problem in Example 25 is that we query for substitutions of a bus identifier B when we are interested in cases where none exists. That is, we have to abstract away from specific bus existences which can only be expressed by auxiliary atoms, i.e., intensional atoms. Thus, towards more expressive reasoning over data streams, we now introduce LARS programs.

3.2.2 LARS Programs

We now define a rule language for stream reasoning with semantics similar to ASP.

Definition 15 (Rule, Program) *A* program P *is a set of* rules*, i.e., given formulas $\alpha, \beta_1, \ldots, \beta_n \in \mathcal{F}$, expressions of the form*

$$\alpha \leftarrow \beta_1, \ldots, \beta_n. \tag{3.3}$$

Given a rule r of form (3.3), $H(r)$ denotes the *head* α, and $\beta(r) = \beta_1 \wedge \ldots \wedge \beta_n$ the *body* of r; the commas in (3.3) are a syntactic variant of $\wedge$ as usual. We can thus write a rule alternatively as material implication

$$\beta(r) \rightarrow H(r). \tag{3.4}$$

To evaluate a program on a data stream, we first need interpretations.

Definition 16 (Interpretation) *Let $D = (T, \upsilon_D)$ be a data stream and $I = (T, \upsilon)$ be a stream such that $D \subseteq I$. If for all $t \in T$ it holds that $\upsilon(t) \setminus \upsilon_D(t) \subseteq \mathcal{A}^{\mathcal{I}}$, then I is called an* interpretation stream (for D)*, and a structure $M = \langle I, W, B\rangle$ is called an* interpretation (for D)*.*

Intuitively, interpretations are structures whose streams extend the data stream exlusively with intensional atoms.

Definition 17 (Model) *Let $M = \langle I, W, B\rangle$, where $I = (T, \upsilon)$, be an interpretation for data stream $D = (T, \upsilon_D)$. Furthermore, let P be a ground program. We then say*

- *M is a* model of rule $r \in P$ for D at time t*, denoted $M, t \models r$, if $M, t \models \beta(r) \rightarrow H(r)$;*
- *M is a* model of P for D at time t*, denoted $M, t \models P$, if $M, t \models r$ for all rules $r \in P$;*
- *M is a* minimal model*, if no model $M' = \langle S', W, B\rangle$ of P for D at time t exists such that $S' = (T, \upsilon')$ and $S' \subset S$.*

Note that minimality is defined with respect to the same timeline T. We often omit "for D" and/or "at t" if this is clear from the context. The *reduct* of a program P with respect to M at time t is defined by

$$P^{M,t} = \{r \in P \mid M, t \models \beta(r)\}, \tag{3.5}$$

i.e., the subset of rules whose bodies are satisfied.

Definition 18 (Answer Stream) *Let $M = \langle I, W, B\rangle$ be a structure, where $I = (T, \upsilon)$ is an interpretation stream for a data stream D, let P be a program and $t \in T$. Then, I is called an* answer stream *of P for D at time t (relative to W and B), if M is a $\subseteq$-minimal model of the reduct $P^{M,t}$ for D at time t.*

Intuitively, the reduct $P^{M,t}$ serves to disregard irrelevant rules with respect to an interpretation, i.e., those that do not fire. The remaining ones need to be satisfied in order for M to be a model. By demanding in addition that M is minimal, we ensure that all conclusions are supported, i.e., each intensional atom in an answer stream can be justified by derivations from (chains of) firing rules based on extensional atoms, i.e., atoms from the stream or background data. Anything not derivable this way is assumed to be false, i.e., we adopt default negation.

For ASP fragments of LARS, answer streams correspond to answer sets as defined by the FLP-reduct [FLP04], which we formulated for LARS programs above. More precisely, consider an interpretation stream $I = (\{t\}, \upsilon')$ for a data stream $D = (\{t\}, \upsilon)$ and let P_{ASP} be a program where in each rule of form (3.3) all body formulas β_i are literals, i.e., atoms or negated atoms, and the head α is a disjunction of atoms. Then, we have:

Proposition 1 *For I and P_{ASP} as described, I is an answer stream of P for D at t relative to arbitrary W and B iff $\upsilon'(t)$ is an answer set of $P_{ASP} \cup \upsilon(t) \cup B$.*

That is, ordinary answer set programs are subsumed by LARS programs. In other words, the rich semantic properties of ASP carry over to LARS. We thus obtain a nonmonotonic semantics that comes with possibly multiple, supported minimal models. Section 3.2.3 below further explores the semantic properties of LARS.

Non-ground programs

As for formulas, we consider non-ground programs as schematic versions of ground programs with variables of two sorts, namely constant variables $\mathcal{V}$ and time variables $\mathcal{U}$. The semantics of these *non-ground programs* is given by the answer streams of according groundings, obtained by replacing variables with constants from $\mathcal{C}$, respectively time points from T, in all possible ways.

Example 26 (cont'd) We now solve the problem of Example 25 as follows: the intention of formula α_3' is formalized as rule r_1 which uses the intensional atom *aBus*, derived by

rule r_2:

$$\begin{aligned} r_1: \quad & q(U,A,St) \leftarrow @_U\, tram(A,St), @_U \neg \boxplus^{+3} \Diamond aBus(St); \\ r_2: \quad & @_U\, aBus(St) \leftarrow @_U\, bus(B,St). \end{aligned}$$

Here, q is the output relation which may be used for post processing and *aBus* is the intended abstraction for the appearance of any bus at the given station St. Rule r_2 assigns to any time point U an (intensional) atom $aBus(St)$ whenever there is an atom $bus(B,St)$ at U.

Apart from the expressiveness issue, intensional atoms may also be used to enhance readability. For instance, the complex formula $@_U \neg \boxplus^{+3} \Diamond aBus(St)$ in r_1 may be simplified, giving the subformula $\boxplus^{+3} \Diamond aBus(St)$ a name on its own. This leads to the following approach:

$$\begin{aligned} r_1': \quad & q(U,A,St) \leftarrow @_U\, tram(A,St), @_U \neg busSoon(St), \\ r_2': \quad & @_U\, busSoon(St) \leftarrow @_U\, \boxplus^{+3}\, \Diamond bus(B,St). \end{aligned}$$

Note further that rule r_2' may be written without the use of a window as

$$@_U\, busSoon(St) \leftarrow @_{U'}\, bus(B,St),\ U' > U,\ U' - U \leq 3.$$

Using the program $P = \{r_1', r_2'\}$, we now get the intended result in the single answer stream $I = (T, v^I)$ at $t = 45$: the evaluation function v^I assigns intensional atoms as follows (grouping shared time points):

$$\begin{aligned} 33,34,35,36 &\mapsto \{busSoon(b)\} \\ 37,38,39,40 &\mapsto \{busSoon(s)\} \\ 42,43,44,45 &\mapsto \{busSoon(m)\} \\ 45 &\mapsto \{q(40,a_3,h)\} \end{aligned}$$

That is, $busSoon(b)$ is assigned to time points 33, 34, 35, and 36, $busSoon(s)$ holds throughout the interval from 37 to 40, $busSoon(m)$ from 42 until 45; and the evaluation of time point 45 additionally contains $q(40,a_3,h)$. The latter derivation correctly reflects that only at station h no bus appeared within 5 minutes after a tram appearance (tram a_3 at minute 40). ∎

Particular semantic assets for LARS programs are inherited from Answer Set Programming, i.e., a multiple-model semantics permitting nonmonotonic reasoning.

Example 27 (cont'd) Consider now a scenario where Karl wants to travel only with modern trams, since old trams are inconvenient with baby strollers. We calculate expected arrival times based on tram appearences at stations where we have no recent report of a traffic jam. Then, we consider the change from tram Id_1 to a modern tram Id_2 as a

good connection, if it is expected at the same station X within the next 5 minutes. This is expressed by the following two rules.

$$@_T\, exp(Id, Y) \leftarrow \boxplus^{\mathrm{idx},n} @_{T_1}\, tram(Id, X), line(Id, L), \neg\, \boxplus^{20} \Diamond jam(X), \\ plan(L, X, Y, D), T = T_1 + D. \tag{3.6}$$

$$gc(Id_1, Id_2, X) \leftarrow @_T\, exp(Id_1, X), @_T\, \boxplus^{+5} \Diamond exp(Id_2, X), \\ Id_1 \neq Id_2, \neg old(Id_2). \tag{3.7}$$

Rule (3.6) encodes when a tram is expected at later stops. Similarly as in Example 14, we use for the partition-based window operator $\boxplus^{\mathrm{idx},n}$ the index function idx such that $\mathrm{idx}(g) = i$ for an atom $g \in \mathcal{G}$ of form $tram(a_i, X)$ and $\mathrm{idx}(g) = 0$ else. By the tuple-based windows of sizes $n(i) = 1$ for $i > 0$ and $n(0) = 0$ we get for each tram a_i only its most recent appearance at some stop X. Usually, the expected arrival time on the next stop can be computed by the travelling duration according to the table *plan*. For the case of traffic jams within the last 20 minutes, we block such conclusions by means of default negation.

Next, Rule (3.7) builds on the expected arrival times of Rule (3.6) to identify good connections where the targeted tram is not an old make and the expected waiting time is at most 5 minutes. It uses a time-based window that looks 5 minutes ahead from the time when $exp(Id_1, X)$ is concluded and checks the existence (operator $\Diamond$) of an expected (different) tram Id_2.

Assuming background data B including the facts

$$plan(\ell_1, b, m, 8), plan(\ell_2, g, m, 7), plan(\ell_3, h, m, 3), \ldots,$$
$$line(a_1, \ell_1), line(a_2, \ell_2), line(a_3, \ell_3), \ldots,$$
$$old(a_1), \ldots,$$

we observe that the interpretation stream of the structure M of Example 19 is an answer stream of P for D at time t. Note that $gc(a_3, a_1, m)$ is not derived. Tram a_1 appears one minute after a_3 at Mozart Circus, but it is old. ∎

The next example demonstrates another advantage of our rule-based approach, namely the possibility to obtain different models for nondeterministic choices.

Example 28 (cont'd) Consider an extended scenario where a tram with identifier a_2 of line ℓ_2 is reported at Gulda Lane (g) at time point 38. This updates the data stream $D = (T, \upsilon)$ in Example 8 to $D' = (T, \upsilon')$, where υ' is the evaluation $\upsilon \cup \{38 \mapsto \{tram(a_2, g)\}\}$. By the entries $line(a_2, \ell_2)$ and $plan(\ell_2, g, m, 7)$ in B, Rule (3.6) derives that tram a_2 is expected to arrive at Mozart Circus at $t = 45$. Furthermore, we now assume that tram a_1 is not old, i.e., $old(a_1) \notin B$. This gives Karl three good connections at stop m, when leaving tram a_3 at time 43:

$$G = \{gc(a_3, a_1, m), gc(a_1, a_2, m), gc(a_3, a_2, m)\}$$

Karl is not interested in the connection from a_1 to a_2, since he is currently travelling with a_3. His smart phone streams an according tuple $on(a_3)$ at query time. This leaves him two options: he can either change to line ℓ_1 (and take tram a_1 after 1 minute at time point 44), or to line ℓ_2 (and take tram a_2 after 2 minutes at 45). The following two rules formalize the possibility to either change trams or skip a good connection:

$$change(Id_1, Id_2, X) \leftarrow on(Id_1), gc(Id_1, Id_2, X), \neg skip(Id_1, Id_2, X). \tag{3.8}$$
$$skip(Id_1, Id_2, X) \leftarrow gc(Id_1, Id_2, X), change(Id_1, Id_3, X), Id_2 \neq Id_3. \tag{3.9}$$

Consider the program P consisting of rules (3.6)-(3.9). Moreover, let $D'' = (T, \upsilon'')$ be the data stream obtained from D' by adding $\{42 \mapsto \{on(a_3)\}\}$ to the evaluation and let $I_0 = (T, \upsilon_0)$, $I_1 = (T, \upsilon_1)$ and $I_2 = (T, \upsilon_2)$ be the following interpretation streams for D'': we take

$$\upsilon_0 = \upsilon \cup \left\{ \begin{array}{ll} 42 \mapsto G, & 43 \mapsto \{exp(a_3, m)\} \\ 44 \mapsto \{exp(a_1, m)\}, & 45 \mapsto \{exp(a_2, m)\} \end{array} \right\},$$

and for $i \in \{1, 2\}$, let $\upsilon_i = \upsilon_0 \cup \{42 \mapsto choice_i\}$, where

$$choice_1 = \{change(a_3, a_1, m), skip(a_3, a_2, m)\}, \text{ and}$$
$$choice_2 = \{change(a_3, a_2, m), skip(a_3, a_1, m)\}.$$

Then, I_1 and I_2 are (the only) two answer streams for P at time 42 relative to $W = \{\tau, \mathrm{p}\}$ and B, i.e., we get the user choices as separate models. ∎

Note that in this example we did not constrain good connections by the actual destination Karl wants to reach. By means of the presented formalism, such reachability relations can be expressed elegantly through recursion as in Datalog.

Another benefit of our approach for advanced stream reasoning is the possibility to *retract* previous conclusions due to new input data. Combined with (minimal) model generation, i.e., alternatives that may be enumerated, compared under preference etc., such nonmonotonic reasoning allows for sophisticated AI applications in data stream settings.

Example 29 (cont'd) If the lines ℓ_1 and ℓ_2 have the same travelling time from Mozart Circus to Strauß Avenue, Karl will pick $choice_1$ (answer stream I_1), since at $t = 42$ tram a_1 is expected to arrive one minute earlier than tram a_2.

Suppose a few seconds later (still at $t = 42$) a traffic jam is reported for Beethoven Square. Thus, we now consider the data stream $D_j = (T, \upsilon_j)$, where $\upsilon_j = \upsilon \cup \{42 \mapsto \{on(a_3), jam(b)\}\}$. Thus, we have no expectation anymore when tram a_1 will arrive at Mozart Circus. Now $exp(a_1, m)$ cannot be concluded for $t = 44$, and as a consequence, $gc(a_3, a_1, m)$ will not hold anymore. Thus, the previous two answer streams are discarded and only $change(a_3, a_2, m)$ remains recommended in the resulting unique answer stream. ∎

3.2.3 Semantic Properties of LARS Programs

In this subsection, we show that some basic properties of the answer semantics of logic programs carry over to the notion of answer stream defined above. These are minimality of answer streams, supportedness by rules and consistency, i.e., existence of an answer stream in the absence of negation provided that the windows functions involved are monotonic, i.e., return growing substreams if the stream data increases.

Let P be a program, D be a data stream and $t \in \mathbb{N}$. By $\mathcal{AS}(P, D, t)$ we denote the set of answer streams of P for D at time t. The letter M always stands for the structure $M = \langle I, W, B\rangle$, where I is the considered answer stream, and W and B are implicit and fixed. By Definition 18, the structure M (due to answer stream I) is a minimal model of the reduct $P^{M,t}$ for D at time t. Importantly, this implies that M is a model of the original program P, and in fact a minimal model.

Theorem 4 (Minimality of answer streams) *Let P be a LARS program, D be a data stream, t be a time point and $I \in \mathcal{AS}(P, D, t)$. Then, $M = \langle I, W, B\rangle$ is a minimal model of P for D at time t.*

Proof. Consider the structure $M = \langle I, W, B\rangle$, where $I \in \mathcal{AS}(P, D, t)$. We first show that M is a model of P at time t, i.e., that $M, t \models r$ for all rules $r \in P$. There are two cases. If $r \in P^{M,t}$, satisfaction at t holds by definition. Else, let $r = \alpha \leftarrow \beta_1, \ldots, \beta_n$. We have $M, t \not\models \beta(r)$ and thus $M, t \models \beta(r) \rightarrow \alpha$.

As for minimality, suppose that $M' = \langle I, W, B\rangle$ where $M' \subset M$ is a model of P for D at time t. Then, $M', t \models r$ for each rule $r \in P$, and hence $M', t \models r$ for each rule $r \in P^{M,t} \subseteq P$. This means M' is a model of $P^{M,t}$ at time t; hence M is not a minimal model of $P^{M,t}$ at time t, which contradicts $I \in \mathcal{AS}(P, D, t)$. □

Thus, answer streams warrant the property of minimality that answer sets enjoy, in the spirit of logic programming semantics. A simple consequence of minimality of models is the following.

Corollary 1 (Incomparability) *Answer streams are incomparable w.r.t. $\subseteq$. That is, if $I, I' \in \mathcal{AS}(P, D, t)$, then $I \neq I'$ implies $I \not\subseteq I'$ and $I' \not\subseteq I$.*

Our definition of answer streams follows the approach in [FLP04], which requires a supporting rule for every derived atom. In other words, dropping any atom from an answer set would invalidate some rule. In our case, dropping an intensional atom a from an answer stream would lead to an unsatisfied rule that supports its derivation for some time point t'. To simplify notation, we consider $I = (T, \upsilon)$ also as set $\{t' \mapsto a \mid a \in \upsilon(t'), t' \in T\}$. Accordingly, $I \setminus \{t' \mapsto a\}$ amounts to removing in I atom a from $\upsilon(t')$, etc.

Theorem 5 (Supportedness) *Let $I \in \mathcal{AS}(P, D, t)$. Then, for every $t' \mapsto a \in I \setminus D$ there exists a rule $r \in P$ such that*

(i) $M, t \models \beta(r)$, and

(ii) $M', t \not\models r$, *where* $M' = \langle I \setminus \{t' \mapsto a\}, W, B\rangle$.

Proof. Let $I \in \mathcal{AS}(P, D, t)$ and $t' \mapsto a \in I \setminus D$. By Definition 18, $M = \langle I, W, B\rangle$ is a minimal model of $P^{M,t}$ for D at t. Towards a contradiction, assume that for all $r \in P$ it holds that (i) $M, t \not\models \beta(r)$ or (ii) $M', t \models r$, where $M' = \langle I \setminus \{t' \mapsto a\}, W, B\rangle$. We first observe that item (i) cannot hold for all rules, as this would imply that $P^{M,t} = \emptyset$, which has the single minimal model $\langle D, W, B\rangle$ at t; this would imply $I = D$ and thus $t' \mapsto a \in I$ would be impossible, contradiction.

We now only consider those rules $r \in P$ where $M, t \models \beta(r)$, i.e., the reduct $P^{M,t} \neq \emptyset$. Since for all $r \in P^{M,t}$ we have that $M', t \models r$, where $M' = \langle I \setminus \{t' \mapsto a\}, W, B\rangle$, we conclude that M is not a minimal model of $P^{M,t}$. This yields the contradiction. □

Note that the conditions (i) and (ii) in Theorem 5 amount for ordinary logic programs to the usual notion of supportedness of answer sets; if the rule head $\alpha = a_1 \vee \cdots \vee a_k$ in (ii) is a disjunction of atoms, then M must satisfy a single atom a_i in α, and $a_i = a$.

Finally, let us consider LARS programs in which α and each formula β_i are positive, i.e., each atom occurs in the formula tree only under an even number of negations; we call such programs *positive*. As for windows, we naturally call a window function w *monotonic*, if for any streams S and S' such that $S \subseteq S'$ and for any time t' it holds that $w(S, t') \subseteq w(S', t')$. Then we obtain

Theorem 6 (Consistency) *Let P be a positive LARS program such that all heads α of rules in P are satisfiable and all window operator $\boxplus^w$ occurring in P have monotonic window functions w. Then for any D and t, (i) $\mathcal{AS}(P, D, t) \neq \emptyset$ and (ii) any $M = \langle I, W, B\rangle$ is a minimal model of $P^{M,t}$ at t iff $I \in \mathcal{AS}(P, D, t)$.*

Proof. Under the asserted properties, clearly some model $M = \langle I, W, B\rangle$ of P exists for D at t; simply let in I all intensional atoms be true. Furthermore, any model $M' \subset M$ of $P^{M,t}$ for D at t satisfies $M', t \models P$, as under the monotonicity assertions $M', t \not\models \beta(r)$ for each (grounded) rule r in $P \setminus P^{M,t}$. By repeating this argument for M' etc., we can build a maximal, strictly decreasing chain of models $M_0 = M, M_1, M_2, \ldots$ of P for D at t. The intersection N of all these models is another model of P for D at t, and hence no model $N' \subset N$ of P for D at t can exist. Consequently, N is also a minimal model of $P^{N,t}$ for D at t; in other words, N is an answer stream of P for D at t. This proves part (i). As for part (ii), by Theorem 4 each $I \in \mathcal{AS}(P, D, t)$ is such that $M = \langle I, W, B\rangle$ is a minimal model of $P^{M,t}$ at t; conversely, the chain construction in part (i) starting with any minimal model $M = \langle I, W, B\rangle$ of P for D at t yields $N = M$, and thus $I \in \mathcal{AS}(P, D, t)$ holds. □

For example, sliding time-based windows are monotonic and likewise the other time-based windows considered above; furthermore, also filter windows are monotonic. Tuple-based windows (thus also partition-based windows) are not monotonic, and the statement in the theorem does not hold, even for very restricted rule syntax.

Example 30 Consider the program P consisting of the rules

$$\begin{aligned} r_0: &\quad c. \\ r_1: &\quad d \leftarrow \boxplus^{\#1}\Diamond c. \\ r_2: &\quad a \wedge b \leftarrow d. \\ r_3: &\quad b \leftarrow \boxplus^{\#1}\Diamond a. \\ r_4: &\quad a \leftarrow \boxplus^{\#1}\Diamond b. \end{aligned}$$

and assume that the tie-break in the tuple selection is by lexicographic ordering, i.e., a before b before c before d. Informally, $\boxplus^{\#1}\Diamond x$ expresses that the single selected tuple is x. Then $M = \langle I, W, B\rangle$ where (in abuse of notation) $I = \{a, b, c\}$ is a model of P for the data stream $D = ([0, 0], \emptyset)$ at $t = 0$. Moreover, it is the single minimal model for D at t: for any other model $M' = \langle I', W, B\rangle$, we have that $d \in I'$ implies $I' = \{a, b, c, d\}$ (by r_0 and r_2), which is not minimal. Furthermore, $\{a, b\} \cap I' = \emptyset$ would lead by r_1 and r_2 to $\{a, b\} \subseteq I'$, which is contradictory; similarly $I' = \{c, a\}$ (respectively $I' = \{c, b\}$) would lead by r_3 to $b \in I'$ (respectively by r_4 to $a \in I'$), which is again a contradiction. Thus, M is the only answer stream candidate. However, the reduct $P^{M,t} = \{r_0;\ r_3\}$ has a model $M' = \langle I', W, B\rangle$ for D at t where $I' = \{c, b\}$. Thus, M is not an answer stream of P for D at $t = 0$ and $\mathcal{AS}(P, D, t) = \emptyset$ follows. (The same holds if we replace r_2 with rules $a \leftarrow d$ and $b \leftarrow d$; the resulting program is in the *plain* LARS fragment; cf. Section 4.1.) ∎

However, refined versions of tuple-based windows, which e.g. count only extensional data (as occurs often in practice), are monotonic and thus admissible in Theorem 6; furthermore, monotone windows can be nested as monotonicity is preserved. We remark that the consistency result (part (i) of the theorem) can be extended to classes of programs with layered (stratified) negation and recursion through non-monotonic windows.

We note that Definition 15 is liberal in the sense that it permits extensional atoms also in rule heads. This is convenient in some scenarios with complex rule heads. Notably, any answer stream for a data stream D may only add intensional atoms to D. Thus, satisfaction of extensional atoms in rule heads anyway hinges on the input D; it is not possible to infer input data. If desired, one may rewrite a program in order to exclude extensional atoms from rule heads. To this end, one replaces every extensional predicate p (that is mentioned in a rule head) by a fresh intensional predicate p' and adds the rules $@_T\, p' \leftarrow @_T\, p$, $@_T\, \neg p' \leftarrow @_T\, \neg p$ (or for the latter rule, alternatively the constraint $\bot \leftarrow \Diamond(p' \wedge \neg p)$). In case a ground program is required that works for all inputs, one may alternatively use the following set of rules, where p'' is another fresh predicate:

$$\begin{aligned} \Box(p' \vee p'') &\leftarrow . \\ \bot &\leftarrow \Diamond(p \wedge p''). \\ \bot &\leftarrow \Diamond(p' \wedge p''). \end{aligned}$$

Clearly, using these encodings, the answer streams of the original program P and of the rewritten program P' are in one-to-one correspondence. Moreover, we note that

for programs that do not use extensional predicates in rule heads, data streams can be reduced to programs without extensional data, by replacing any input atom $p \in \upsilon(t)$ in a data stream $D = (T, \upsilon)$ by the fact $@_t\, p \leftarrow$.

In conclusion, we find that LARS programs have the basic semantic properties comparable to those of ordinary answer set programs under the FLP-reduct. They can thus be seen as an extension of ASP for use cases in streaming with flexible window functions.

3.2.4 Case Study: LARS as Specification Language

LARS was used as specification language in a research project on internet architectures, which we now briefly review.

Modern internet usage is no longer characterized by sending and retrieving text messages as in its beginning, but by rapid distribution of media content, in particular videos. The underlying architecture was suitable for the initial research network of a few nodes, but does no longer fit today's demands of fast global delivery of large data volumes. Various so-called Future Internet research activities address this problem. In particular, Content-Centric Networking (CCN) [JST+09] proposes a replacement of location-based addressing with a name/content-based one, i.e., centering around *what* the user requests, not from *where* data packets can be retrieved. To this end, a CCN router can *cache* recently transmitted content such as chunks of video data. This way, user requests can be satisfied more quickly and at lower costs than by repeated retrieval from other sources. Clearly, routers can only store a limited portion of all requested content. Consequently, they need to implement a *caching strategy* that decides which content is cached and for how long it is maintained, before memory is used for other content items.

Following up on previous studies on the influence of caching strategies on the performance of CCN routers, we presented in [BBD+16] and [BBD+17] a dynamic approach that changes the caching strategy based on the usage pattern. Our *intelligent caching agent* (ICA) adds to a typical CCN architecture a control unit for each router that switches strategies in response to a parameter α (of a Zipf distribution) that indicates the level of distribution of content interest. Intuitively, a high value indicates a strong concentration towards a smaller number of items. This may be observed, e.g., shortly after a new episode of a highly popular show becomes available online.

We ran simulations to measure the influence of four different caching strategies, and we could show in particular that performance gains can be expected when strategies are adapted dynamically based on the request pattern, as abstracted in the α-parameter. The decision unit of the developed simulation architecture was based on LARS: the knowledge base, when to use which strategy, was written in a customized rule language with LARS semantics. More specifically, the employed fragment was *plain LARS* (cf. Section 4.1).

Problem	formula class		program class	
	α	α^-	P	P^-
Model Checking (MC)	PSpace	P	PSpace	co-NP
Satisfiability (SAT)	PSpace	NP	PSpace	Σ_2^p

Table 3.1: Complexity of reasoning in ground LARS (completeness results)

Example 31 Consider the following two rules from such a knowledge base:

$$@_T\, high \leftarrow \boxplus^{30} @_T\, \alpha(V), V \geq 18 \tag{3.10}$$

$$lfu \leftarrow \boxplus^{30} \Box high \tag{3.11}$$

The first rules states that any time point T within the last 30 time points (seconds) will be associated with atom *high* if the α-value at T was at least 1.8 (written as integer 18). Then, if the α-value has always been high in this interval, caching strategy *lfu* (i.e., *least frequently used*) shall be adopted; it dismisses the content item with the least number of cache hits (accesses) when space for new content is required. ∎

The reasoning module, materializing this logic, was implemented using DLVHEX 2.5 [EMRS15]. We made use of a single window operator that selects all tuples in a given timeline; this function could be conveniently implemented using *external atoms*, i.e., interfaces to data outside the rule set. The reasoning system was periodically triggered, and the simulated router then continued with the caching strategy derived for the current evaluation data.

The simulation architecture for this study was the first show case for reasoning with LARS, where the possibility to write declarative rules as such was particularly useful. Moreover, a specific advantage in the CCN context is the possibility to change rules without taking the router offline. Notably, the proposed method is also applicable for explorative or experimental scenarios, since different decision criteria can be immediately tested while the overall system keeps running.

In Chapter 7, that introduces the standalone stream reasoning system Ticker, we will return to this use case. In particular, we will use two benchmark programs from this context for the empirical evaluation of Ticker.

3.3 Computational Complexity of Reasoning in LARS

In this section, we analyze the computational complexity of LARS, where we consider model checking and the satisfiability problem, for both LARS formulas and programs. In our analysis, we concentrate on the general case but pay attention to the effect of nested windows and particular classes of windows; a comprehensive study of the computational complexity for a rich taxonomy of classes of LARS formulas and LARS programs remains however for further study.

3.3.1 Problem Statements and Overview of Results

We say that a stream $S = (T, v)$ *is over* a subset $\mathcal{A}' \subseteq \mathcal{A}$ of atoms $\mathcal{A}$, if $v(t) \setminus \mathcal{A}' = \emptyset$ for all $t \in T$. We study the complexity of the following reasoning tasks, where in the following W is a set of window functions that are evaluable in polynomial time, $B \subseteq \mathcal{A}$ is a set of background atoms, and where α is a ground LARS formula and P a ground LARS program.

(1) *Model Checking* (MC). Given $M = \langle S^\star, W, B\rangle$, $S^\star = (T, v)$, and $t \in T$, check whether

- for a given stream $S \subseteq S^\star$ and formula α it holds that $M, S, t \Vdash \alpha$; respectively
- $I = (T, v)$ is an answer stream of a given program P for a data stream $D \subseteq I$ at t.

(2) *Satisfiability* (SAT). For decidability, we assume that relevant atoms are confined to a subset $\mathcal{A}' \subseteq \mathcal{A}$ of polynomial size in the input. The reasoning tasks are:

- Given W, B, a timeline T, a time point $t \in T$, and a formula α, does some stream $S = (T, v)$ over $\mathcal{A}'$ exist such that $M, S, t \Vdash \alpha$, where $M = \langle S, W, B\rangle$?
- Given W, B, a data stream D with timeline T, a time point $t \in T$, and a program P, does P have some answer stream for D at t that is over $\mathcal{A}'$?

Table 3.1 shows the computational complexity of reasoning in ground LARS, where α^- and P^- denotes the class of formulas respectively programs where the nesting depth of window operators in formulas and rules is bounded by a constant.

As we can see from Table 3.1, in the general case model checking and satisfiability checking are both PSpace-complete, and thus beyond the Polynomial Hierarchy. Informally, the recursive evaluation of a formula creates an exponential size tree, but at each point in time, only a polynomial size portion of this tree needs to be in memory. The PSpace-hardness arises from the temporal operators $\Box$ and $\Diamond$ in combination with window operators. This allows for encoding quantified Boolean formulas (QBFs), whose evaluation is a canonical PSpace-complete problem.

The picture changes if we bound the window nesting depth in formulas and programs. Under a constant bound, the evaluation tree that is built has only polynomially many nodes (i.e., substreams). This allows us to solve the model checking problem for ground α^- formulas in polynomial time by using labeling techniques. The remaining results for satisfiability testing and for ground LARS programs are obtained from guess and check algorithms. The lower bounds (the hardness results) are in essence inherited from the complexity of answer set programs, except for model checking of LARS formulas. The P-hardness of the latter problem is due to the generic form of windows whose associated functions can be P-hard.

Note that from the results on Model Checking in Table 3.1, we immediately obtain complexity results for answering ground queries $Q = \langle M, t, \alpha\rangle$: the problem is PSpace-complete in general, but polynomial for bounded window-nesting; and as the discussion in Section 3.3.4 shows, this generalizes to a richer class of queries.

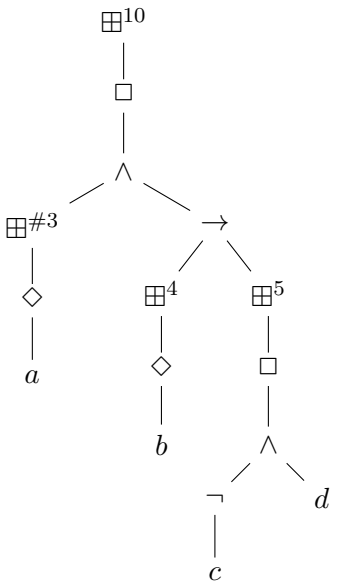


Figure 3.12: Tree representation of formula $\boxplus^{10}\Box(\boxplus^{\#3}\Diamond a \wedge (\boxplus^{4}\Diamond b \rightarrow \boxplus^{5}\Box(\neg c \wedge d)))$

3.3.2 Derivation of the Complexity Results

LARS Formulas

The complexity results for LARS formulas in the general case are based on the following result for model checking.

Theorem 7 *Given a structure $M = \langle S^\star, W, B\rangle$, a stream $S = (T, \upsilon)$ such that $S \subseteq S^\star$, a time point t, and an arbitrary ground formula α, deciding $M, S, t \Vdash \alpha$ is* PSpace*-complete, where the* PSpace*-hardness holds for $S = S^\star$.*

Intuitively, PSpace-membership is shown by a depth-first-search evaluation of the input formula α along its tree representation. An example for the formula tree is shown in Fig. 3.12.

At each node of the tree, we need to store the content according to the window operators that are applied as in the path from the root. This requires only polynomial space for that node and all nodes on the path to it as well.

To show PSpace-hardness, we reduce the evaluation of QBFs $\exists x_1 \forall x_2 \cdots Q_n x_n \phi(\mathbf{x})$ to model checking. A LARS formula $\alpha = \Diamond \boxplus^{\text{set}:x_1} \Box \boxplus^{\text{set}:x_2} \cdots \phi(\mathbf{x})$ on the timeline $T = [0, 1]$ is constructed where the window operator $\boxplus^{\text{set}:x_i}$ effects the possible truth assignments to x_i at the time points 0 or 1. To this end, the initial stream $S^\star$ has all atoms $x_1, \ldots, x_n$ at both 0 and 1. When $\boxplus^{\text{set}:x_i}$ is evaluated at time point 0 (respectively 1), it removes x_i from (respectively keeps x_i in) the stream. By branching to 0 or 1, all truth assignments to $x_1, \ldots, x_n$ are generated in an evaluation tree. On top, $\Diamond$, $\Box$ naturally encode the quantifiers $\exists$ and $\forall$. Fig. 3.13 shows an example evaluation tree:

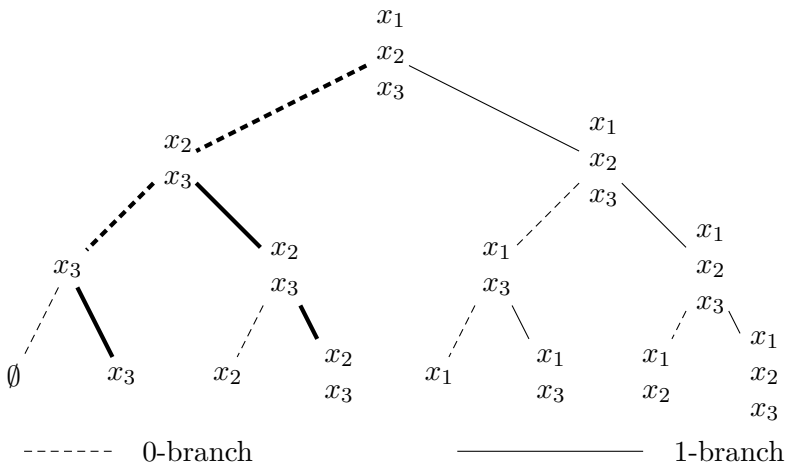


Figure 3.13: Evaluation tree for the formula $\Diamond \boxplus^{\text{set}:x_1} \Box \boxplus^{\text{set}:x_2} \Diamond \boxplus^{\text{set}:x_3} (\neg x_1 \wedge (x_2 \vee x_3))$, encoding the QBF $\exists x_1 \forall x_2 \exists x_3 (\neg x_1 \wedge (x_2 \vee x_3))$. Nodes $x_1\, x_2\, x_3$, $x_2\, x_3$, etc. represent the streams with timeline $[0, 1]$ and interpretation $\upsilon(0) = \upsilon(1) = \{x_1, x_2, x_3\}$, $\upsilon(0) = \upsilon(1) = \{x_2, x_3\}$, etc., as obtained by window operators $\boxplus^{\text{set}:x_i}$.

the bold lines mark subtrees for which, given the assignment to $x_1, \ldots, x_i$ by the path, the subformula $Q_{i+1} x_{i+1} \ldots Q_n x_n \phi(\mathbf{x})$ evaluates to true. (More details are given in the Appendix.)

For the satisfiability problem of LARS formulas, we obtain a similar result.

Theorem 8 *Problem SAT for LARS formulas, i.e., given W, B, T, and t, is there a stream $S = (T, \upsilon)$ is over $\mathcal{A}'$ such that $M, S, t \Vdash \alpha$, where $M = \langle S, W, B \rangle$, is* PSpace*-complete.*

Informally, a suitable evaluation function can be guessed and checked in polynomial space, and from NPSpace = PSpace we obtain membership in PSpace. On the other hand, LARS model checking can be easily reduced to satisfiability testing (see Appendix).

LARS Programs

Based on Theorem 7, we show that model checking for ground LARS programs has the same complexity as for LARS formulas in general.

Theorem 9 *Problem MC for LARS programs, i.e., given a structure $M = \langle I, W, B \rangle$, a data stream D, a program P, and a time point t, deciding whether $I = (T, \upsilon)$ is an answer stream of P for D at time t, is* PSpace*-complete.*

Proof. To decide the problem, we can (a) check that I is an interpretation stream for D, (b) compute $P^{M,t}$, and (c) check that M is a minimal model of $P^{M,t}$, i.e., that (c.1) $M, t \models P^{M,t}$ and (c.2) no $M' = \langle I', W, B \rangle$, with $I' = (T, \upsilon') \subset (T, \upsilon)$ exists such that $M', t \models P^{M,t}$. Now,

1. step (a) is trivially polynomial;

2. steps (b) and (c.1) are feasible in polynomial time using a PSpace oracle; and

3. step (c.2) is feasible in nondeterministic polynomial time using a PSpace oracle (guess (T', v') and check $M', t \models P^{M,t}$).

Overall, the computation is feasible in NPSpace, thus in PSpace (as NPSpace = PSpace).

The PSpace-hardness of the problem is easily obtained from Theorem 7: for given ground formula α and $M = \langle S, W, B\rangle$, let $P = \{\alpha \leftarrow \top\}$, where $\top$ is an arbitrary tautology. Note that no intensional data occur in α, and thus no interpretation M' that is smaller than M is possible, and thus $M = \langle S, W, B\rangle$ is an answer stream for P for D at t iff $M, S, t \Vdash P$ holds. □

For checking satisfiability of LARS programs, we obtain based on the previous theorem also PSpace-completeness.

Theorem 10 *Deciding SAT for LARS programs, i.e., given W, B, D and some LARS program P, does P have some answer stream I over $\mathcal{A}'$ for D at t, is* PSpace*-complete.*

Proof. To show satisfiability of a ground LARS program P, we can guess a stream $I = (T, v)$ and check that I is an answer stream of P for D at t; the guess is polynomial in the size of $\mathcal{A}'$ and the check feasible in PSpace by Theorem 9; overall, the computation is feasible in NPSpace, thus in PSpace.

The PSpace-hardness of SAT for LARS programs P follows from the reduction of MC for LARS formulas to MC for LARS programs in the proof of Theorem 9. □

3.3.3 Bounded Window Nesting

Revisiting Fig. 3.13, we see that an exponential size evaluation tree results from the evaluation of nested window operators $\boxplus^{\text{set}:x_i}$, where each of them is evaluated at both time points 0 and 1. In this way, exponentially many different substreams are produced in the evaluation. Such an exponential explosion is avoided, if we bound the nesting of window operators in LARS formulas.

Definition 19 (Window nesting depth $wnd(\alpha)$) *The* window nesting depth *of a LARS formula α, denoted $wnd(\alpha)$, is the maximal number of window operators encountered on any path from the root to a leaf in the formula tree of α.*[2] *Formally, $wnd(a) = 0$ for every atom a and inductively $wnd(\neg\alpha) = wnd(\Box\alpha) = wnd(\Diamond\alpha) = wnd(\triangleright\alpha) = wnd(\alpha)$; $wnd(\alpha \wedge \beta) = wnd(\alpha \vee \beta) = wnd(\alpha \rightarrow \beta) = \max(wnd(\alpha), wnd(\beta))$; and $wnd(\boxplus\alpha) = 1 + wnd(\alpha)$.*

[2]For simplicity, we omit a more fine-grained definition of *wnd* that respects $\triangleright$, for which "encountered" is replaced by "encountered subsequently ... with no $\triangleright$ in between." The tractability result carries over to the larger class of formulas.

Note in particular that $wnd(\alpha) = 0$ means no window operators occur in α, and that $wnd(\alpha) = 1$ means that window operators occur but unnested.

If $\#w(\alpha)$ is the number of window operators occurring in a LARS formula α, then at most $(\#w(\alpha) \cdot |T|)^{wnd(\alpha)}$ many substreams of a stream $S = (T, \upsilon)$ (respectively $S^\star = (T^\star, \upsilon^\star)$) are created in a recursive evaluation of $M, S, t \Vdash \alpha$. If $wnd(\alpha)$ is bounded by a constant, then this number is polynomial in the size of S and α. We can thus use a labeling technique to evaluate formulas bottom up (from subformulas) over the possible substreams in polynomial time.

Theorem 11 *Problem MC for LARS formulas α is in* P*, if $wnd(\alpha)$ is bounded by some constant $k \geq 0$, and is* P*-complete for arbitrary window operators.*

The P membership part follows from a more general result in the next subsection (Theorem 13). We also have matching P-hardness (and thus P-completeness) in general due to the fact that evaluating window functions can be P-complete in general.

As a consequence of Theorem 11, also satisfiability of LARS formulas becomes easier to decide when the nesting depth is bounded.

Corollary 2 *Problem SAT for LARS formulas α is* NP*-complete, if $wnd(\alpha)$ is bounded by some constant $k \geq 0$.*

The membership is via a simple guess and check argument. Since LARS subsumes propositional logic, the problem is clearly also NP-hard.

Turning to LARS programs, let us define the window nesting depth for a program P naturally as follows.

Definition 20 (Window nesting depth $wnd(P)$) *Given a LARS program P, its* window nesting depth *is defined as $wnd(P) = \max\{wnd(\beta(r) \rightarrow \alpha) \mid \alpha \leftarrow \beta(r) \in P\}$.*

Our focus is here on finite LARS programs P, for which the nesting depth is always well-defined and finite. For model checking such programs, we obtain the following result.

Theorem 12 *Problem MC for LARS programs P is co-*NP*-complete, if $wnd(P)$ is bounded by some constant $k \geq 0$.*

Proof. Membership in co-NP can be seen as follows: the PSpace oracle in the algorithm considered in the proof of Theorem 9 can by Theorem 11 be replaced by a polynomial-time computation. It follows that we can refute nondeterministically in polynomial time that I is an answer stream of P for D at t. Consequently, problem MC is for LARS programs P^- in co-NP. On the other hand, co-NP-hardness is inherited from the co-NP-completeness of answer set checking for (disjunctive) propositional logic programs, cf. [EG95], which is subsumed by model checking for LARS programs. □

From Theorem 12, the following corollary is not difficult to obtain.

Corollary 3 *Problem SAT for LARS programs P is Σ_2^p-complete, if $wnd(P)$ is bounded by some constant $k \geq 0$.*

The membership in Σ_2^p follows from Theorem 12, as a candidate answer stream for P w.r.t. a data stream D and time point t can be guessed and checked in polynomial time with an NP oracle. The Σ_2^p-hardness is inherited from propositional disjunctive logic programs, for which deciding answer set existence is Σ_2^p-complete [EG95].

3.3.4 Semantic Restriction: Sparse Window Functions

Bounding the nesting depth of windows serves as a restriction that allows us to obtain tractability of model checking for LARS formulas. In addition to this syntactic criterion, we can obtain other important cases for which solving this problem is feasible in polynomial time due to semantic properties of the window operators that occur in a LARS formula.

An important such property is that a window operator $\boxplus^w$ and a nested window operator $\boxplus^{w_1}\boxplus^{w_2}$, applied to a stream S from a small (polynomial size) set of streams, will always return a stream from that set. By sharing nodes in the substream evaluation tree, the resulting evaluation graph has polynomial size and the relevant subformula labeling for deciding satisfiability can be produced in polynomial time. Following the intuition that such window operators/functions can produce only few substreams in total, we call them *sparse*.[3]

A prototypical example of sparse window functions/operators (or sparse windows, for short) are (generalized) sliding time-based windows $\boxplus^{\ell,u}$ (short for $\boxplus^{\ell,u(1)}$, i.e., hop size $d = 1$) that cover the previous ℓ and the next u time points.[4] Applied on a stream $S = (T, \upsilon)$, the resulting window at time point t is the substream $S|_{T'}$, which restricts the timeline to $T' = T \cap [t - \ell, t + u]$. Notably, the result of evaluating nested sliding time-based windows $\boxplus^{\ell_1,u_1} \cdots \boxplus^{\ell_k,u_k}$ also is a substream S' obtained by simply restricting the timeline; overall, there are $O(|T|^2)$ many such S'.

We next describe evaluation graphs and results for sparse windows in more detail, and then discuss concrete classes of window operators that ensure the sparse window property. All time-based, tuple-based and filter windows considered in Section 3.1.3, 3.1.4, and 3.1.6, respectively, are among them, as well as a large class of partition-based windows in Section 3.1.5; furthermore, windows from these classes can be mixed arbitrarily.

For uniformity, we regard in the sequel the reset operator $\triangleright$ in abuse of the notion as a window operator that yields, in the context of a structure $M = \langle S^\star, W, B\rangle$, the original stream; i.e., $\triangleright$ is viewed as $\boxplus^{w_\triangleright^{S^\star}}$ where $w^{S^\star_\triangleright}(S, t) = S^\star$ (for all $t \in \mathbb{N}$).

Window graph

For any formula φ, we refer to the sequences of window operators $\boxplus^{w_1} \to \boxplus^{w_2} \to \cdots \to \boxplus^{w_k}$ of φ along the branches of the formula tree of φ, where $k = 1, 2, \ldots$, as the *window-paths* of φ. Consider now a structure $M = \langle S^\star, W, B\rangle$ and a substream $S \subseteq S^\star$. We call a set $\mathcal{S}$ of streams an *evaluation base* of (M, S, φ) and write $\mathcal{S}_B(M, S, \varphi)$ or simply $\mathcal{S}_B$, if it contains S and each stream S_{k+1} that results if we apply the window operators

[3] Note that the condition is about large gaps in the space of potential substreams produced by a window operator; the property does not concern the number of atoms in the input stream.

[4] Subsequent results carry over for windows $\boxplus^{\ell,u(d)}$ where $d > 1$, we confine here to $d = 1$ for simplicity.

$\boxplus^{w_k}$, starting from $S_0 = S$, at each time point of the current stream S_k recursively along a window-path of φ. An evaluation base includes all streams that can be encountered in the recursive evaluation of $M, S, t \Vdash \varphi$ according to Definition 12, but it in general it includes further streams as well (we shall discuss this aspect later in this section). Given such a base $\mathcal{S}_B$, the *window graph* for (M, S, φ), denoted $WG_{\mathcal{S}_B}(M, S, \varphi)$ or simply $WG_{\mathcal{S}_B}$ is the graph $WG_{\mathcal{S}_B} = (N, E)$ with nodes $N = \mathcal{S}_B$ and edges E that are obtained inductively along window-paths as follows: add from the node S_{k-1} for each time point t in S_{k-1} an edge labeled $(\boxplus^{w_k}, t)$ to $S_k = w_k(S_{k-1}, t)$, where $S_0 = S$. Informally, paths in $WG_{\mathcal{S}_B}$ starting at S allow us to navigate between substreams of S as obtained by window operators as they occur in φ.

Example 32 Consider a structure $M = \langle S^\star, W, B\rangle$, where $S^\star = ([0, 8], \upsilon)$ and $\upsilon = \{5 \mapsto \{a\}, 7 \mapsto \{b\}, 8 \mapsto \{c\}\}$, and the formula $\varphi = \boxplus^{\#2}(\boxplus^3 \Diamond b \wedge \boxplus^{\#1} \Diamond c)$. We take $S = S^\star$. The window-paths of maximal length are $p_1 = \boxplus^{\#2} \to \boxplus^3$ and $p_2 = \boxplus^{\#2} \to \boxplus^{\#1}$, i.e., on the initial stream S we can apply $\boxplus^{\#2}$ (at potentially every time point), and in the resulting streams one can apply $\boxplus^3$, respectively $\boxplus^{\#1}$. We now establish an evaluation base $\mathcal{S}_B$ for (M, S, φ). The initial window operator $\boxplus^{\#2}$ yields potential windows $\#^2(S, 0), \ldots, \#^2(S, 8)$, i.e., by abbreviating the restriction $S|_{T'}$ to timeline $T' = [a, b]$ by S_{ab}, the streams S_{00}, S_{01}, S_{02}, …, S_{06}, S_{57}, S_{78}. For path p_1, we may now apply on any of these streams at their respective time points the window function τ^3, similarly for p_2 function $\#^1$. This results in an evaluation base $\mathcal{S}_B$. Note that some of these additional applications return their input stream, e.g., $\tau^3(S_{78}, 8) = S_{78}$, since the timeline $[7, 8]$ has size 1 and is not shrunk further by a time-based window of size 3.

As for the edges of the window graph $WG_{\mathcal{S}_B}$, we add in the first step an edge $S \to S_{00}$ with label $(\boxplus^{\#2}, 0)$, an edge $S \to S_{01}$ with label $(\boxplus^{\#2}, 1)$, …, and an edge $S \to S_{78}$ with label $(\boxplus^{\#2}, 8)$. Then, in the second step for p_1, we add an edge $(S_{00} \to S_{00}, 0)$ with label $(\boxplus^3, 0)$, $S_{01} \to S_{00}$ with $(\boxplus^3, 0)$, $S_{01} \to S_{01}$ with $(\boxplus^3, 1)$, …, $S_{06} \to S_{52}$ with $(\boxplus^3, 5)$, $S_{06} \to S_{36}$ with $(\boxplus^3, 6)$, …, $S_{78} \to S_{77}$ with $(\boxplus^3, 7)$, and $S_{78} \to S_{78}$ with $(\boxplus^3, 8)$; and similarly with $\boxplus^{\#1}$ for p_2. ∎

For our purposes, the following lemma is useful.

Lemma 1 *Given a structure $M = \langle S^\star, W, B\rangle$, a substream $S \subseteq S^\star$, a formula φ and an evaluation base $\mathcal{S}_B$ for (M, S, φ), the window graph $WG_{\mathcal{S}_B}$ for (M, S, φ) is computable in polynomial time.*

Proof. Indeed, by traversing the recursive Definition 12, we can add the edges of $WG_{\mathcal{S}_B} = (N, E)$ as described, by calculating each $w_k(S_{k-1}, t)$, which takes polynomial time; there are $|T^\star|$ many such calculations to make, and in total thus at most $\#w(\varphi) * |T^\star|$ many, where $\#w(\varphi)$ is the total number of window occurrences in φ. In order to find $w_k(S_{k-1}, t)$ in $\mathcal{S}_B$, i.e., the stream $S' \in \mathcal{S}_B$ such that $S' = w_k(S_{k-1}, t)$ one can use hashing or, if the stream is to large, use a trie structure which makes this feasible in $O(\|S'\|)$ time, where $\|S'\|$ is the size of S' (which is $O(|\mathcal{A}| * |T|)$).

Thus in total the time to compute $WG_{\mathcal{S}_B}$ for (M, S, φ) is

$$O(\#w(\varphi) * |\mathcal{S}_B| * |T^\star| * (C_w + \|S^\star\|) + |\varphi|) \;=\; O(|\varphi| * |\mathcal{S}_B| * |T^\star|^{k+1} * |\mathcal{A}|^k), \quad (3.12)$$

where $C_w = O(\|S^\star\|^k) = O(|T^\star|^k * |\mathcal{A}|^k)$ is a polynomial in $\|S^\star\|$ that bounds the evaluation time of any window. Indeed, there are at most $\#w(\varphi) * |\mathcal{S}_B| * |T^\star|$ many edges to consider, and computing plus matching a window S' against $\mathcal{S}_B$ takes $O(C_w + \|S^\star\|)$ time. Overall, this is polynomial in the size of M, $\mathcal{S}_B$, and φ. □

Stream Labeling

Given a structure $M = \langle S^\star, W, B\rangle$, a substream $S \subseteq S^\star$ and a window graph $WG_{\mathcal{S}_B} = (N, E)$ for (M, S, φ), we label each pair (S, t) such that $S \in N$ and $t \in T^\star$, where $S = (T, \upsilon)$, with relevant formulas that hold in stream S at time t due to the evaluation base $\mathcal{S}_B$. We define the label set $L_{\mathcal{S}_B}(S, t)$ by the following steps:

1. take a subformula $\boxplus^{w_k}\alpha_k$ in φ such that $\boxplus^{w_1} \to \boxplus^{w_2} \to \cdots \to \boxplus^{w_k}$ is a maximal window-path in the formula tree of φ that has not yet been considered. We label each pair (S, t), such that $S = S_k$, with all subformulas of α_k that evaluate in the stream S at time t to true.

   More precisely, we add a subformula α' of α_k to $L_{\mathcal{S}_B}(S, t)$, where $S = (T, \upsilon)$, by a case distinction on the form of α' due to Definition 12 as follows. We add:

   - atom a, if $a \in \upsilon_k(t)$;
   - $\neg\alpha$, if α is not in $L_{\mathcal{S}_B}(S, t)$;
   - $\alpha \wedge \beta$, if α and β are in $L_{\mathcal{S}_B}(S, t)$; similarly for $\vee$ and $\to$;
   - $\Diamond\alpha/\Box\alpha$, if $\alpha \in L_{\mathcal{S}_B}(S, u)$ for some/all $u \in T$;
   - $@_u\alpha$, if $\alpha \in L_{\mathcal{S}_B}(S, u)$ and $u \in T$.

   This labeling can be carried out bottom up along the formula tree. Note that in the first application of Step 1 α_k does not contain any window operator.

2. We label (S, t), where $S = S_{k-1}$, with $\boxplus^{w_k}\alpha_k$ if (S_k, t) was labeled with α_k in Step 1.

3. Inductively, the window path $\boxplus^{w_1} \to \boxplus^{w_2} \to \cdots \to \boxplus^{w_i}$ is considered in Step 1 for $i < k$, i.e., one considers subformula $\boxplus^{w_i}\alpha_i$, after all window operators that occur in α_i have been considered. Any subformula $\boxplus^{w_j}\alpha$ of α_i starting with a window operator is like an atom, and presence of $\boxplus^{w_j}\alpha$ in $L_{\mathcal{S}_B}(S_i, t)$ reflects the entailment result for this subformula in S_i at t.

Example 33 (cont'd) Consider the window graph $WG_{\mathcal{S}_B}$ of Example 32. We are interested whether $M, S, 8 \Vdash \varphi$ holds ($S = S^\star$) and start illustrating the bottom up evaluation by considering window-path p_2 at $t = 8$, i.e., edges $S \to S_{78}$ and $S_{78} \to S_{88}$ with window graph labels $(\boxplus^{\#2}, 8)$ and $(\boxplus^{\#1}, 8)$, respectively. The (maximal) window-path p_2 ends before subformula $\Diamond c$ which, in Step 1, is evaluated in $S_k = S_{88} = ([8, 8], \{8 \mapsto \{c\}\})$. Thus, we obtain formula labels $L_{\mathcal{S}_B}(S_k, 8) = \{c, \Diamond c\}$. In Step 2, we thus get $L_{\mathcal{S}_B}(S_{k-1}, 8) = \{\boxplus^{\#1}\Diamond c\}$, where $S_{k-1} = S_{78} = ([7, 8], \{7 \mapsto \{b\}, 8 \mapsto \{c\}\})$, i.e., the previous stream in the considered path.

Likewise, we evaluate subformula $\Diamond b$ for path p_1 at $t = 8$, i.e., the edges $S \to S_{78}$ and $S_{78} \to S_{78}$ with window graph labels $(\boxplus^{\#2}, 8)$ and $(\boxplus^3, 8)$, respectively. In Step 1, we add label $\Diamond b$ to $L_{\mathcal{S}_B}(S_{78}, 8)$. Note that b does not hold at time 8 in S_{78} but $b \in L_{\mathcal{S}_B}(S_{78}, 7)$ (and $7 \in [7, 8]$) from similar evaluation, e.g., along path $S \to S_{78} \to S_{77}$ with window graph labels $(\boxplus^{\#2}, 8)$ and $(\boxplus^3, 7)$, respectively. Thus, we add in Step 2 to the formula $\boxplus^3 \Diamond b$ to $L_{\mathcal{S}_B}(S_{78}, 8)$ (note that in this path $S_k = S_{k-1}$).

Step 3 recognizes that all window operators of the conjunction $\varphi' = \boxplus^3 \Diamond b \wedge \boxplus^{\#1} \Diamond c$ have been considered. Hence, we go to Step 1 and find that for the formula $\alpha_k = \varphi'$ both conjuncts $\boxplus^3 \Diamond b$ and $\boxplus^{\#1} \Diamond c$ are in $L_{\mathcal{S}_B}(S_{78}, 8)$, i.e., φ' holds and is added. In the next Step 2, we consider $S_{k-1} = S$, i.e., the original stream before evaluating $\boxplus^{\#2}$. (That is, we navigate back the first edge $S \to S_{78}$ with label $(\boxplus^{\#2}, 8)$ for either path.) Since φ' has been added to $(S_{78}, 8)$ in Step 1, we now assign $L_{\mathcal{S}_B}(S, 8) = \{\boxplus^{\#2} \varphi'\}$. Finally, we recognize in Step 3 that no window operator remains to be considered along paths p_1 and p_2 at $t = 8$. We skip the stream labeling of further pairs (S, t), as we already obtained that $\varphi \in L_{\mathcal{S}_B}(S, 8)$ which means that $M, S, 8 \Vdash \varphi$ holds. ∎

Proposition 2 *Let $\mathcal{S}_B$ be an evaluation base for (M, S, φ), where $M = \langle S^\star, W, B \rangle$ and $S \subseteq S^\star$, and let $t \in T^\star$. Then, it holds that $M, S, t \Vdash \varphi$ iff $\varphi \in L_{\mathcal{S}_B}(S, t)$.*

Proof. The statement is proved by induction on the structure of the formula φ. In the base case, φ is a single atom a, and by construction the windows graph $WG_{\mathcal{S}_B}$ has no edges. The node (S, t), $t \in T^\star$ is labeled with a iff $a \in \upsilon(t)$, where $S = (T, \upsilon)$, and thus $M, S, t \Vdash \varphi$ iff $\varphi \in L_{\mathcal{S}_B}(S, t)$ holds. In the induction step, assume that the statement holds for all subformulas of φ, and consider the different cases of the root connective *op* of φ. From the induction hypothesis that or each M, S, t and subformula α of φ, the entailment $M, S, t \Vdash \alpha$ is correctly reflected by $\alpha \in L_{\mathcal{S}_B}(S, t)$, it is not hard to verify that in each case, φ is added to the label of (S, t) if and only if $M, S, t \Vdash \varphi$ holds. Note in this context that for $op = \boxplus^w$ and $op = \triangleright$, the window graph $WG_{\mathcal{S}_B}$ for φ contains the one for $M, S', t \Vdash \alpha$ respectively $M, S, t' \Vdash \alpha$ or $M, S^\star, t \Vdash \alpha$ in the recursive Definition 12 as a subgraph. □

We thus obtain an algorithm to decide $M, S, t \Vdash \varphi$ as follows:

1. given an evaluation base $\mathcal{S}_B$ for (M, S, φ), compute the window graph $WG_{\mathcal{S}_B}$;
2. compute the labeling $L_{\mathcal{S}_B}$ for φ;
3. return "yes" iff $\varphi \in L_{\mathcal{S}_B}(S, t)$.

The correctness of this algorithm follows from Proposition 2. Regarding its time complexity, it is not hard to see that the algorithm runs in time polynomial in the size of $\mathcal{S}_B$, M and φ (see Appendix). In particular, if the evaluation base $\mathcal{S}_B$ is small, we obtain tractability.

Theorem 13 *Let $M = \langle S^\star, W, B \rangle$ be a structure, $S \subseteq S^\star$ and let φ be a formula. Suppose that (M, S, φ) has some evaluation base $\mathcal{S}_B$ of size polynomial in the size of M and φ. Then $M, S, t \Vdash \varphi$ is decidable in polynomial time.*

Proof. From a window graph $WG_{\mathcal{S}_B} = (N, E)$ for (M, S, φ), where $M = \langle S^\star, W, B\rangle$, we can drop each node $S' \neq S$ from $\mathcal{S}_B$ that does not occur in E; the remaining graph $WG_{\mathcal{S}^\star_B} = (N', E)$ is the smallest window graph possible, i.e., $N' \subseteq N$ holds for each window graph $WG_{\mathcal{S}'_B} = (N, E)$ for (M, S, φ). Notably we can build $\mathcal{S}^\star_B$ on the fly by initially setting $\mathcal{S}^\star_B = \{S\}$ and by then adding any $S' = w_k(S_{k-1}, t)$ along the window path that is not yet member of $\mathcal{S}^\star_B$. Following the analysis in Lemma 1, as inserting a stream S into $\mathcal{S}_B$ is like searching feasible in $O(\|S\|)$ time, building $WG_{\mathcal{S}^\star_B}$ take also time bounded by (3.12), i.e., by $O(\#w(\varphi) * |\mathcal{S}_B| * |T^\star| * (C_w + \|S^\star\|) + |\varphi|) = O(|\varphi| * |\mathcal{S}_B| * |T^\star|^{k+1} * |\mathcal{A}|^k)$.

The time to construct the bottom labeling is bounded by $O(|T^\star| * |\mathcal{S}^\star_B| * |\varphi|)$: for each subformula φ' of φ and pair (S', t'), where $S \in \mathcal{S}^\star_B$ and $t' \in T^*$ we have to decide whether φ' is put in $L_{\mathcal{S}_B}(S', t')$. This can be decided by constantly many lookups of already constructed labels for subformulas of φ'; for $\Diamond\alpha$, we can use a flag that is set true if (S', t') is labeled with α, where $t' \in T$. For $\Box\alpha$, we can proceed similarly.

The total runtime of the algorithm is thus bounded by

$$O(|\varphi| * |\mathcal{S}_B| * |T^\star|^{k+1} * |\mathcal{A}|^k + |T^\star| * |\mathcal{S}^\star_B| * |\varphi|) \;=\; O(|\varphi| * |\mathcal{S}_B| * |T^\star|^{k+1} * |\mathcal{A}|^k),$$

Given that the size of $\mathcal{S}_B$ is polynomial in the size of M and φ, it follows that the runtime is polynomial in the size of M and φ. □

As already mentioned, the presented stream labeling $L_{\mathcal{S}_B}$ for φ will in general contain more streams than necessary for the evaluation of $M, S, t \Vdash \varphi$. We considered in Example 33 the entailment relation $M, S, 8 \Vdash \varphi$, where S has the timeline $[0, 8]$. All pairs (S', t') with a proper substream S' of S that has a timeline overlapping with $[0, 6]$ are irrelevant, as the first window operator $\boxplus^{\#2}$ already restricts the relevant timeline to $[7, 8]$ and any further consideration affects only substreams of S_{78}. This would be different if a temporal modality $\circ \in \{\Diamond, \Box, @_{t'}\}$ was in front of φ. More generally, if we consider a formula $\circ\psi$ such that ψ does not start with a temporal modality at some time point t, we observe that $\circ$ changes which time points have to be considered (i.e., all or some t'), while only a window operator in ψ will change the timeline. Moreover, given a sequence $\circ_1 \cdots \circ_n$ of modalities, we observe that the first $n - 1$ are irrelevant. Accordingly, we can restrict both the evaluation base $\mathcal{S}_B$ and the window graph $WG_{\mathcal{S}_B}$ by considering the last modality $\circ$ for the current stream S_{k-1} to first determine the relevant time points t' based on which we step to $S_k = (S_{k-1}, t')$. Following this intuition, we in fact skipped the discussion of most pairs (S', t') in Example 33, by directly starting with the paths for $t = 8$ and focusing on the relevant streams in $\mathcal{S}_B$ and the relevant edges of the stream graph $WG_{\mathcal{S}_B}$. It is feasible to obtain these relevant subsets in polynomial time as well. While one can expect a significant speedup in a practical realization of this improvement, the worst case polynomial complexity of the stream labeling procedure does not change. In a more fine-grained view of the formula φ that looks besides window operators also at the occurrence of temporal operators, further tractable fragments of LARS formulas could be identified; we leave this for future work.

On the other hand, from the theoretical perspective, we note that an even more abstract approach is possible: we may alternatively define an evaluation base independently of a formula, i.e., purely based on a structure $M = \langle S^\star, W, B\rangle$: any (potentially infinite)

sequence of window operators $\boxplus^w$ with $w \in W$ will eventually not produce new streams. A given formula φ that only uses window functions from W only represents a subset of these sequences. Sparse windows ensure that, even in this high-level approach, the size of the evaluation base remains polynomial in the size of of the structure.

Classes of Sparse Window Functions

From Theorem 13 we immediately obtain the P-membership of model checking for LARS formulas with bounded window nesting in Theorem 11. Furthermore, we can conclude that model checking for LARS formulas with unbounded nesting is tractable for a broad class of window operators.

Concerning time-based window operators $\boxplus^{\ell,u(d)}$, as already observed above, evaluating the window function $\tau^{\ell,u(d)}(S,t)$ on the stream S at time point t always yields a substream $S' = (T', \upsilon') = (T', \upsilon|_{T'})$, i.e., S' restricts S to the timeline T'. If we apply a further time-based window on S', we obtain another stream of this form. Overall, there are $O(|T|^2)$ many such S'; if we take possible occurrence of the reset operator $\triangleright$ into account, there are $O(|T^\star|^2)$ many such streams. Thus, Theorem 13 holds for all LARS formulas that use only time-based window operators.

A similar consideration establishes the same result for tuple-based windows $\boxplus^{\#\ell,u}$: evaluating a tuple-based window function $\#^{\ell,u}(S,t)$ on the stream S at time point t yields a substream $S' = (T', \upsilon')$ of $S = (T, \upsilon)$ that diverges from $S = (T', \upsilon|_{T'})$ at most on the stream boundaries t_ℓ and t_u, where $T' = [t_\ell, t_u]$. In any case, S' is uniquely identified by the triple (ℓ, u, t). If we apply a further tuple-based window on S', we again obtain a substream of S of this form; overall, there are $O(|T^\star| \cdot A^2)$ many such streams, where $A = \sum_{t \in T^\star} |\upsilon^\star(t)|$ is the total number of atoms in the stream $S^\star$, thus polynomially many.

Each tuple-based window $\boxplus^{\#\ell,u}$ trivially amounts to a partition-based window $\boxplus^{\mathrm{idx}_\mathcal{A}, n_\mathcal{A}}$ where all atoms are in one partition. The question is thus whether also partition-based windows are sparse. Unfortunately, the answer is negative.

Theorem 14 *Problem MC for LARS formulas in which only partition-based windows occur is* PSpace*-complete.*

Proof. By Theorem 7, it remains to show PSpace-hardness. For this, we reconsider the reduction from evaluating a QBF $\Phi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, x_2, \ldots, x_n)$ as in (A.1) to model checking for a LARS formula (A.2), and adapt the reduction as follows.

For each atom x_i, we introduce a fresh atom w_i, and we change the content of stream $S^\star = ([0,1], \upsilon^\star)$ to $\upsilon^\star(0) = \{x_1, \ldots, x_n\}$, $\upsilon^\star(1) = \{w_1, x_1, \ldots, w_n, x_n\}$. Furthermore, we replace the window $\boxplus^{\mathrm{set}:x_1}$ with the partition-based window $\boxplus^{\mathrm{idx}^{(i)}, n^{(i)}}$, where $\mathrm{idx}^{(i)}$ creates two partitions $\mathrm{idx}^{-1}(1) = \{w_i, x_i\}$ and $\mathrm{idx}^{-1}(2) = \mathcal{A} \setminus \{w_i, x_i\}$, with the counts $n^{(i)}(1) = (1,2)$ and $n^{(i)}(2) = (\infty, \infty)$, where ∞ can be replaced by any number $\geq 2 * (n-1)$. For making the selection deterministic, we assume any total order $\leq$ (e.g., lexicographic order) such that $w_i \leq x_i$ for all $i = 1, \ldots n$.

Informally, these changes have the following effects:

- evaluated at time point 0, the partition based window function $\mathrm{p}^{\mathrm{idx}^{(i)},n^{(i)}}$ will for the partition $\{w_i, x_i\}$ remove at time0 one atom ($\ell^{(i)} = 1$); as w_i is not in $\upsilon^\star(0)$, it will remove x_1; at 1, it will remove two atoms ($u^{(i)} = 2$), and thus both w_i and x_i.

- evaluated at time point 1, $\mathrm{p}^{\mathrm{idx}^{(i)},n^{(i)}}$ will for the partition $\{w_i, x_i\}$ remove at time 1 one atom (as $\ell^{(i)} = 1$), and as $w_i < x_i$, it will remove w_i; the count $u^{(i)} = 2$ has no effect as 1 is the maximal time point in $T^\star = [0, 1]$.

Thus, the stream $\mathrm{p}^{\mathrm{idx}^{(i)},n^{(i)}}(S^\star, 0)$ has neither x_i nor w_i at $t = 0, 1$, and $\mathrm{p}^{\mathrm{idx}^{(i)},n^{(i)}}(S^\star, 0)$ has x_i at $t = 0, 1$ and w_i not at $t = 0, 1$. Thus, the evaluation of the formula

$$\alpha' = W_1 \boxplus^{\mathrm{idx}^{(1)},n^{(1)}} W_2 \boxplus^{\mathrm{idx}^{(2)},n^{(2)}} \cdots W_n \boxplus^{\mathrm{idx}^{(n)},n^{(n)}} \phi(x_1, x_2, \ldots, x_n),$$

on the modified $S^\star$ generates at the innermost evaluation level the same streams

$$\mathrm{p}^{\mathrm{idx}^{(n)},n^{(n)}}(\mathrm{p}^{\mathrm{idx}^{(n-1)},n^{(n-1)}}(\cdots \mathrm{p}^{\mathrm{idx}^{(1)},n^{(1)}}(S^\star, t_1), \cdots), t_{n-1}), t_n),$$

where $t_1, \ldots, t_n \in \{0, 1\}$, as the evaluation of the formula α in (A.2) on the original $S^\star$ at the innermost level, which are given by $\mathrm{set}^{x_n}(\mathrm{set}^{x_{n-1}}(\cdots \mathrm{set}^{x_1}(S^\star, t_1), \cdots), t_{n-1}), t_n)$.

Consequently, α' is entailed at an arbitrary $t \in \{0, 1\}$ on the modified $S^\star$ iff α is entailed at an arbitrary $t \in \{0, 1\}$ on the original $S^\star$. This proves PSpace-hardness.

We note that the reduction proves the result where each partition-based window creates only two (individual) partitions, but all such windows use the same tuple counts $n^{(i)} = (1, 2)$. The reduction above can be easily adjusted to partition-based windows that use all the *same partitioning* but *different tuple counts*: just let $\mathrm{idx}^{(i')} = \mathrm{idx}$, where $\mathrm{idx}(j) = \{w_j, x_j\}$ and $n^{(i')}(j) = (\ell^i_j, u^i_j)$, for $i, j = 1, \ldots, n$, such that $\ell^i_j = u^i_j = \infty$ if $j \neq i$, and $\ell^i_j = 1$ and $u^i_j = 2$ if $j = i$; then each associated partition-based window function $\mathrm{p}^{\mathrm{idx}^{(i')},n^{(i')}}$ clearly coincides with $\mathrm{p}^{\mathrm{idx}^{(i)},n^{(i)}}$, which implies PSpace-hardness for this setting. □

An analysis of the proof shows that the result even holds if each partition-based window creates only two partitions (which is the minimum in order not to collapse with a tuple-based window); it is recursive nesting and the use of either changing partitions, or of changing tuple counts (or both) which leads to intractability.

The result for tuple-based windows generalizes to partition-based windows, provided that the index functions idx of the window operators $\boxplus^{\mathrm{idx},n}$ that occur in the formula partition the ground atoms $\mathcal{G}$ into groups that are formed from constantly many base groups B_i. That is, each group $\mathrm{idx}^{-1}(i)$ is of the form $\mathrm{idx}^{-1}(i) = \bigcup \mathcal{B}$, where $\mathcal{B} \subseteq \{B_1, \ldots, B_k\}$ and $\mathcal{G} = \bigcup_{i=1}^{k} B_i$, where k is constant. Let us call such partitions *meager*. In this case, each (nested) result of evaluating a window function can be uniquely identified by a tuple $(\ell_1, u_1, \ldots, \ell_k, u_k, t)$, and there are polynomially many such tuples.

Finally, let us consider filter windows as introduced in Section 3.1.6. Recall that the function f^A associated with $\boxplus^A$ projects the input stream to the atoms in A. We can extend the description for partition-based windows results above by adding a concrete filter

A that is applied prior to the partition-based selection. While in general, semantically an exponential number of filters A are possible, for the concrete evaluation of a LARS formula φ only filters A that syntactically result from the formula matter, and there are only linearly many of them. Thus, the number of relevant substream descriptions $(\ell_1, u_1, \ldots, \ell_k, u_k, t, A)$ is still polynomially bounded.

Clearly, the meager-partition representations include all tuple-based representations, which in turn include all time-based representations. Thus, we obtain the following result.

Theorem 15 *For LARS formulas α (respectively LARS programs P), problem MC is in* P *(respectively co-*NP*-complete), if only time-based, tuple-based, meager partition-based, and filter windows occur in α (respectively in P).*

Proof. To begin with, each reset operator $\triangleright$ that occurs in the formula α returns the stream $S^\star$, and thus we need to deal with substreams of S and $S^\star$ in the evaluation of α. It is sufficient to consider just S (and substreams of it), as $S = S^\star$ is covered and the result then clearly follows.

As argued in the discussion, each result of a time-based window function $\tau^{\ell,u(d)}(S,t)$ can be expressed as the result of a tuple-based window function $\#^{\ell',u'}(S,t')$, where the counts ℓ' and u' are set such that exactly the atoms in $\tau^{\ell,u(d)}(S,t')$ will be selected. Furthermore, each result of a tuple-based window function $\#^{\ell,u}(S,t)$ can be viewed as the result of a dummy partition-based window function $\mathrm{p}^{\mathrm{idx}_\mathcal{A},n_\mathcal{A}}(S,t)$ where $\mathrm{idx}_\mathcal{A} : \mathcal{A} \to \{1\}$ and $n_\mathcal{A}(1) = (\ell, u)$, i.e., only one partition exists that contains all tuples; clearly, this trivial partition is always the union $B_1 \cup \cdots \cup B_k$ of all base groups B_i for meager partitionings. Thus, it is sufficient to consider the latter case.

Consider a stream $S = (T, \upsilon)$, $T = [t_\ell, t_u]$, and some time point $t \in T$. We can describe with a tuple $d_i = (\ell_1, u_1, \ldots, \ell_k, u_k, t)$ stream S, where ℓ_i states how many atoms in S from $[t_\ell, t]$ that are in partition B_i should be included, and u_i similarly how many atoms in S from $[t+1, t_u]$ are in B_i. Now if $\mathrm{p}^{\mathrm{idx},n}(S,t')$ is applied on S, we can single out, for each partition $\mathrm{idx}^{-1}(j) = \bigcup\{B_i \mid B_i \subseteq \mathrm{idx}^{-1}(j)\}$, how many atoms back (respectively forward) from t' have to be included, depending on $n(j) = (\ell_j^{\mathrm{idx}}, u_j^{\mathrm{idx}})$ and we can break this down to counts ℓ_i', u_i' for all base groups B_i, $i = 1, \ldots, k$. Thus we obtain a description $d_i' = (\ell_1', u_1', \ldots, \ell_k', u_k', t')$ that describes $\mathrm{p}^{\mathrm{idx},n}(S,t')$. Overall, there are polynomially many such descriptions in the size of the input.

Finally, we extend the description d_i for a substream with a filter A to $sd = (\ell_1, u_1, \ldots, \ell_k, u_k, t, A)$, where for describing the initial stream S we can use $A = \mathcal{A}$ (i.e., no atom is filtered). Clearly, if we apply a meager partition-based window function on S at t', the description $sd' = (\ell_1, u_1, \ldots, \ell_k, u_k, t, A)$ of S can be adjusted to represent $\mathrm{p}^{\mathrm{idx},n}(S,t')$ by a description $sd = (\ell_1', u_1', \ldots, \ell_k', u_k', t', A)$;[5] in case of a filter window application, we can represent $\mathrm{f}^{A'}(S,t')$ by $sd' = (\ell_1', u_1', \ldots, \ell_k', u_k', t', A \cap A')$. Consequently, the possible descriptions that result are of the form $sd = (\ell_1, u_1, \ldots, \ell_k, u_k, t, A_1 \cap A_1 \cap \cdots \cap A_i)$ where $\boxplus^{A_1}, \boxplus^{A_2}, \ldots, \boxplus^{A_i}$ are the filter windows encountered on some path from the root of the formula tree of φ; overall, the number of paths (and thus such sequences) is bounded

[5] We tacitly assume here that the order of atoms that is used to resolve non-deterministic selection is static and does not depend on the time point t.

by the size of φ. Hence, overall the number of extended descriptions is polynomially bounded in the size of the input. □

This result can be further generalized by allowing in addition restricted occurrence of arbitrary windows in formulas. In particular, this holds if the nesting depth of such additional window operators in a formula is bounded by a constant. This is because if on a root-path in the formula tree the encountered such windows are $\boxplus^{w_1}, \ldots, \boxplus^{w_\ell}$, then the resulting substream S can be described by a sequence

$$sd_0, \boxplus^{w_1}, sd_1, \ldots, sd_{\ell-1}, \boxplus^{w_\ell}, sd_\ell,$$

where each $sd_i = (\ell_1^{(i)}, u_1^{(i)}, \ldots, \ell_k^{(i)}, u_k^{(i)}, t^{(i)}, A^{(i)})$ is an extended partition-based window description. In total, only polynomially many such descriptions will matter.

Thus in conclusion, for a wide range of formulas that occur in practice the Model Checking problem for LARS formulas is solvable in polynomial time. Furthermore, in frequent use cases with time-based and tuple-based windows it will have low complexity inside P. Finally, based on the tractability of model checking, for satisfiability the same results as for α^- and P^- in Table 3.1 can be established using analog arguments.

3.4 Summary

In this chapter we presented and studied the LARS framework, which aims at filling a previous gap in theoretical stream reasoning research. It provides the possibility to express model-based semantics for queries or programs over streams using generic window operators and different temporal modalities. We formalized streams, windows, window functions and then introduced a framework that extends ASP for streams on top of a monotonic core, i.e., LARS formulas. The framework is generic and very expressive; various concrete logics can be obtained by fixing (i) a suitable fragment for the task at hand, and (ii) the set of employed window functions.

The remainder of this thesis makes use of LARS in several ways. First, we will compare LARS with other formalisms in Chapter 4 and also use it to clarify other informally specified semantics there. The possibility to formally define the output of a query or program is also a prerequisite for optimizations of processing tools. We examine in Chapter 5 notions of equivalence between LARS programs and how they can be characterized in model-based terms. We will then investigate in Chapter 6 approaches for incremental reasoning, addressing the issue of frequently updating results. There, LARS serves as a means to specify precisely a method for updating a model due to an incremental program change that reflects the changes of the stream. Moreover, it is also employed as stream reasoning language itself. We mention in Section 6.5.2 the new streaming tool *Laser* that aims at efficiently updating the unique answer stream in stratified fragments of LARS. *Ticker*, another stream reasoning engine, will be presented in more detail in Chapter 7. It implements the developed incremental reasoning techniques for *plain LARS* programs, i.e., the core fragment of the framework that has emerged in the course of this thesis (cf. Section 4.1).

CHAPTER 4

Relating LARS to other Formalisms

After having introduced the LARS framework in Chapter 3, we are now going to relate it to the larger context of stream processing and reasoning, as well as temporal reasoning. We will study the relation to selected approaches from different research areas and also indicate its use as tool for formal analysis.

Outline

Section 4.1 introduces *plain LARS*, the central fragment that will be used in subsequent chapters; it naturally extends of normal logic programs with the new operators. Section 4.2 provides a translation from LARS to LTL with past time operators. Section 4.3 then studies the Continuous Query Language (CQL) and how LARS can provide a model-based semantics in addition to its operational one. In Section 4.4 we investigate the possibility to express temporal intervals in LARS, comparing it with the rule-based language ETALIS for complex event processing. We then compare in Section 4.5 two Semantic Web approaches for stream processing, i.e., C-SPARQL and CQELS, where LARS serves to clarify a central semantic difference on formal grounds. Finally, we discuss Section 4.6 distinctive semantic features of LARS.

Publications

As in Chapter 3, we build on the publication [BDTE18] and its predecessors [BDTE17, BDEF15, BDTEF14b, BDTEF14a]. In particular, Sections 4.3 (CQL), 4.2 (LTL) and 4.5 (C-SPARQL and CQELS) are from there; the latter is a summary of two further papers, i.e., the workshop paper [DBE15b] and the conference version [DBE15a]. The discussion from Section 4.6 is also from [BDTE18]. Section 4.4 on ETALIS presents results

from [BDEF15]; technical details are given in the Appendix. Plain LARS was first defined in [BDE16].

4.1 Answer Set Programming (ASP): Plain LARS

We already observed in Proposition 1 (page 57) that LARS programs extend logic programs under the answer set semantics. In addition to this, we define here a practical fragment of LARS, called *plain LARS*, as lightweight extension of normal logic programs for streams as follows.

Given an atom $a \in \mathcal{A}$ and a time point $t \in \mathbb{N}$, a formula of form

- $@_t a$ is called an @-*atom*, and
- $\boxplus^w \star a$, where $\star \in \{\Diamond, \Box, @_t\}$, a *window atom*.

Finally, a formula α is an *extended atom*, if it is an atom, an @-atom, or a window atom, i.e., from the grammar

$$a \mid @_t a \mid \boxplus @_t a \mid \boxplus \Diamond a \mid \boxplus \Box a\,. \tag{4.1}$$

We use $\mathcal{A}^+(P)$ to denote the set of extended atoms occurring in a program P. Plain LARS is then defined by allowing in normal logic programs @-atoms in rule heads, and extended atoms in rule bodies.

Definition 21 (Plain LARS) *A* plain LARS rule *is of form*

$$\alpha \leftarrow \beta_1, \ldots, \beta_n, \text{not}\, \beta_{n+1}, \ldots, \text{not}\, \beta_m\,; \tag{4.2}$$

where α is an atom or an @-atom, and each β_i ($1 \leq i \leq m$) is an extended atom. A plain LARS program *is a set of Plain LARS rules.*

As for ASP, the body of a rule r of the above form can now be separated into the *positive body* $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and the *negative body* $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. We use the negation "not" in (4.2) as a notational variant of $\neg$ (as defined in LARS). In that respect, the *body* $B(r)$ (as defined earlier) equals $\{\beta_1, \ldots, \beta_n, \neg\beta_{n+1}, \ldots, \neg\beta_m\}$.

Plain LARS emerged as guiding fragment for many studies in the course of this thesis. In particular, it is the targeted formalism for the work on incremental reasoning (Chapter 6). We will give in Section 6.3 an encoding of plain LARS to ASP. Elaborating on how this encoding can be updated dynamically then leads to the incremental reasoning algorithm in Section 6.4. This algorithm is implemented in the Ticker engine (Chapter 7), which is also based on plain LARS.

Plain LARS is also the employed fragment in further work on incremental reasoning, as mentioned in Section 6.5.1, summarizing [BDE15], where it was first considered, and in Section 6.5.2; the latter uses a slight variation where negation is considered only directly in front of atoms. Plain LARS was also used in the work on Content-Centric Networking that we mentioned in Section 3.2.4. Apart from that, we give in Section 5.5 additional results for this fragment, when considering semantic characterizations of different notions of equivalence.

4.2 Linear Temporal Logic (LTL)

In this section, we compare LARS to temporal logic, where we naturally focus on Linear Temporal Logic (LTL) [Pnu77] extended with past time operators (PLTL) [Mar03]. Syntactically, these logics extend propositional logic with temporal operators, according to the following syntax

$$\phi ::= \bot \mid a \mid \neg\alpha \mid \alpha \wedge \beta \mid \alpha \vee \beta \mid \alpha \rightarrow \beta \mid \mathsf{X}\,\alpha \mid \alpha\,\mathsf{U}\,\beta \mid \mathsf{X}^{-1}\,\alpha \mid \alpha\,\mathsf{U}^{-1}\,\beta;$$

where $a \in \mathcal{G}$. The informal meaning of $\mathsf{X}\,\alpha$ is that α is true at the next point in time, and $\alpha\,\mathsf{U}\,\beta$ means that α is from now on true until β is true at some point; $\mathsf{X}^{-1}\,\alpha$ and $\alpha\,\mathsf{U}^{-1}\,\beta$ are the counterparts for the past (not available in LTL),[1] i.e., that α is true at the previous point of time respectively that α has always been true after some time point at which β was true. Important derived operators are $\mathsf{F}\,\alpha$ and $\mathsf{G}\,\alpha$ which are shorthand for $\top\,\mathsf{U}\,\alpha$, where $\top = \neg\bot$, and $\neg(\top\,\mathsf{U}\,\neg\alpha)$ and state that α is true now or at some respectively now and at every time point in the future; $\mathsf{F}^{-1}\,\alpha = \top\,\mathsf{U}^{-1}\,\alpha$ and $\mathsf{G}^{-1}\,\alpha = \neg(\top\,\mathsf{U}^{-1}\,\neg\alpha)$ express the counterparts for the past.

Semantically, PLTL-formulas are evaluated over paths, which are infinite sequences $\pi = \pi(0), \pi(1), \pi(2), \ldots$ of positions with an associated interpretation $\nu(\pi(i))$ of propositional atoms $\mathcal{A}$, for each $i \geq 0$; the latter is often tacitly omitted. The satisfaction relation $\pi, i \models \alpha$ is inductively defined as follows:

$$\begin{array}{lcl}
\pi, i \models a & :\Leftrightarrow & a \in \nu(\pi(i)), \quad \text{for } a \in \mathcal{A}, \\
\pi, i \models \neg\alpha & :\Leftrightarrow & \pi, i \not\models \alpha, \\
\pi, i \models \alpha \wedge \beta & :\Leftrightarrow & \pi, i \models \alpha \text{ and } \pi, i \models \beta, \\
\pi, i \models \alpha \vee \beta & :\Leftrightarrow & \pi, i \models \alpha \text{ or } \pi, i \models \beta, \\
\pi, i \models \alpha \rightarrow \beta & :\Leftrightarrow & \pi, i \not\models \alpha \text{ or } \pi, i \models \beta, \\
\pi, i \models \mathsf{X}\,\alpha & :\Leftrightarrow & \pi, i+1 \models \alpha, \\
\pi, i \models \alpha\,\mathsf{U}\,\beta & :\Leftrightarrow & \pi, j \models \beta \text{ for some } j \geq i \text{ such that } \pi, k \models \alpha \text{ for all } i \leq k < j, \\
\pi, i \models \mathsf{X}^{-1}\,\alpha & :\Leftrightarrow & i > 0 \text{ and } \pi, i-1 \models \alpha, \\
\pi, i \models \alpha\,\mathsf{U}^{-1}\,\beta & :\Leftrightarrow & \pi, j \models \beta \text{ for some } j \leq i \text{ such that } \pi, k \models \alpha \text{ for all } j < k \leq j.
\end{array}$$

Note in particular that $\neg\,\mathsf{X}^{-1}\,\top$ allows us to recognize that we are at the beginning of the path, as $\pi, i \models \neg\,\mathsf{X}^{-1}\,\top$ holds iff $i = 0$. Two PLTL formulas α and β are *equivalent*, if for every path π and integer $i \geq 0$, it holds that $\pi, i \models \alpha$ iff $\pi, i \models \beta$, and *initially equivalent*, if for every path π it holds that $\pi, 0 \models \alpha$ iff $\pi, 0 \models \beta$.

It is well-known that PLTL is not more expressive than LTL in the sense that every PLTL formula is initially equivalent to some LTL-formula [Gab87], but the smallest such formula can be exponentially larger [Mar03].

Comparison to LARS. Comparing LARS to LTL, we see that the temporal operators are clearly different. The temporal operators $\Box$ and $\Diamond$ in LARS have as counterparts the pairs $\mathsf{G}, \mathsf{G}^{-1}$ and $\mathsf{F}, \mathsf{F}^{-1}$ respectively, which allow one to address all positions in a path; the past time operators are indispensable for evaluation inside the path. The window

[1] To stress symmetry, we write U^{-1} instead of the usual S.

operators in LARS have no counterpart in LTL and PLTL, and similarly the $@_{t'}$ operator which is known as nominal in hybrid logic and can be traced back to Prior's work [Pri67]. On the other hand, LARS has no next-time X nor until U or any of their past time counterparts.

The presence of temporal operators in LTL formulas affects the computational complexity of model checking and satisfiability testing in general (cf. [DS02]). In the general case, for both LTL and PLTL these problems are PSpace-complete, cf. [SC85], where satisfiability of a formula α means existence of a path π such that $\pi, 0 \models \alpha$, and model checking that $\pi, 0 \models \alpha$ for every path π in a given Kripke structure. Thus, at the surface LARS and LTL have the same computational complexity.

However, in LARS we consider only *single path Kripke structures* of a given length, for both satisfiability and model checking at some given time point t. For both LTL and PLTL, model checking single paths is feasible in polynomial time,[2] and satisfiability is thus easily seen to be NP-complete in this setting. Thus, there is a considerable complexity gap between LARS and LTL.

Nonetheless, it is possible to express (propositional) LARS in LTL, if we confine to particular window operators. We show in the next section that this is possible for sliding time-based windows; that is, we can view this instance of LARS as a fragment of LTL.

Translation of LARS to Linear Temporal Logic

Formally, we represent any LARS structure $M = \langle S, W, \emptyset \rangle$, where $S = (T, \upsilon)$, as a PLTL interpretation $\pi(M) = \pi(0), \pi(1), \ldots$ where for each integer $i \geq 0$, $\nu(\pi(i)) = \{\mathbf{u}\}$ if $i \notin T$, and $\nu(\pi(i)) = \upsilon(i)$ otherwise, where $\mathbf{u}$ is a special atom which expresses that the position i is not in the stream.

Our translation of LARS formulas to PLTL-formulas for evaluation at a time point t in a stream $S = (T, \upsilon)$, where $T = [\ell', u']$, is shown in Algorithm 4.1. The parameters ℓ, u mark the interval $[\ell, u]$ of the substream that is currently considered, while ℓ', u' marks the original interval. The translation proceeds recursively, where the temporal modalities $\Box$, $\Diamond$ and $@_{t'}$ are effected using the X operator, where we use here $\mathsf{X}^k\,\alpha$ as a shorthand for the k-fold iteration of X on α; that is, $\mathsf{X}^0\,\alpha = \alpha$, for $k > 0$ we have $\mathsf{X}^k\,\alpha = \mathsf{X}\,\mathsf{X}^{k-1}\,\alpha$, and for $k < 0$ we have $\mathsf{X}^k\,\alpha = \mathsf{X}^{-1}\,\mathsf{X}^{k+1}\,\alpha$. For window operators $\boxplus^{i,j}$ the current interval has to be adjusted to at most i steps before resp. j steps after t, while for the reset operator $\triangleright$ the original interval is selected.

Example 34 Consider the formula $\varphi = @_1 q \vee p \wedge \boxplus^{1,3} \Diamond r$, and let $[\ell', u'] = [\ell, u] = [2, 4]$ and $t = 3$. Then we have

$$\textsc{pltl}(2, 2, 4, 4, 3, \varphi) = \bot \vee p \wedge \mathsf{X}^{-1}\, r \wedge r \wedge \mathsf{X}\, r.$$

The $\bot$ disjunct is due to the fact that position 1 is not in the timeline $T = [2, 4]$. The conjunction $\mathsf{X}^{-1}\, r \wedge r \wedge \mathsf{X}\, r$ stems from the translation of the window $\boxplus^{1,3}$. Note that $\mathsf{X}^2\, r$ and $\mathsf{X}^3\, r$ are missing since $t = 3$ is in distance 1 to the end of the bound $u = 4$. ∎

[2] It is known that the problem is NLogSpace-hard but unknown whether it is in NLogSpace or P-hard in this setting, for both LTL and PLTL [LMS02].

Algorithm 4.1: LARS to PLTL translation

Input: integers ℓ', ℓ, u, u' such that $0 \leq \ell' \leq \ell \leq u \leq u'$, $t \in [\ell', u']$, ground LARS formula φ

Output: PLTL formula

function $\textsc{pltl}(\ell', \ell, u, u', t, \varphi)$

match φ

case	atom a	$\Longrightarrow$	a
case	$\neg\alpha$	$\Longrightarrow$	$\neg\textsc{pltl}(\ell', \ell, u, u', t, \alpha)$
case	$\alpha \wedge \beta$	$\Longrightarrow$	$\textsc{pltl}(\ell', \ell, u, u', t, \alpha) \wedge \textsc{pltl}(\ell', \ell, u, u', t, \beta)$
case	$\alpha \vee \beta$	$\Longrightarrow$	$\textsc{pltl}(\ell', \ell, u, u', t, \alpha) \vee \textsc{pltl}(\ell', \ell, u, u', t, \beta)$
case	$\alpha \rightarrow \beta$	$\Longrightarrow$	$\textsc{pltl}(\ell', \ell, u, u', t, \alpha) \rightarrow \textsc{pltl}(\ell', \ell, u, u', t, \beta)$
case	$\Box\alpha$	$\Longrightarrow$	$\bigwedge_{k=\ell-t}^{u-t} \mathsf{X}^k \textsc{pltl}(\ell', \ell, u, u', t+k, \alpha)$
case	$\Diamond\alpha$	$\Longrightarrow$	$\bigvee_{k=\ell-t}^{u-t} \mathsf{X}^k \textsc{pltl}(\ell', \ell, u, u', t+k, \alpha)$
case	$@_{t'}\,\alpha$	$\Longrightarrow$	**if** $\ell \leq t' \leq u$ **then** $\mathsf{X}^{t'-t} \textsc{pltl}(\ell', \ell, u, u', t', \alpha)$ **else** $\bot$
case	$\boxplus^{i,j}\alpha$	$\Longrightarrow$	$\textsc{pltl}(\ell', \max(\ell, t-i), \min(t+j, u), u', t, \alpha)$
case	$\triangleright\,\alpha$	$\Longrightarrow$	$\textsc{pltl}(\ell', \ell', u', u', t, \alpha)$

end function

We then can show that the transformation in Algorithm 4.1 works properly.

Theorem 16 *Let $M = \langle S, W, \emptyset \rangle$, where $S = (T, \upsilon)$ and $T = [t_\ell, t_u]$, and let $t \in T$ and φ be a LARS formula. Then*

$$M, S, t \Vdash \varphi \text{ iff } \pi(M), t \models \textsc{pltl}(t_\ell, t_\ell, t_u, t_u, t, \varphi) \text{ iff } \pi(M), 0 \models \mathsf{X}^t \textsc{pltl}(t_\ell, t_\ell, t_u, t_u, t, \varphi).$$

Proof. The first equivalence is shown by induction on the structure of the formula; the second follows trivially. The base case of an atom is trivial; the other cases follow easily from the induction hypothesis. Indeed, the cases where φ is a Boolean combination are immediate; likewise, for $\Box\alpha$ and $\Diamond\alpha$ simple quantifier elimination works, and for $@_{t'}\alpha$ moving to the respective position. For the case of $\boxplus^{i,j}\alpha$, the window around t is properly calculated, where $T' = [\ell', u']$ and $\ell' = \max(\ell, t-i)$ and $u' = \min(t+j, u)$; note that $t \in T'$ holds. Finally, for $\triangleright\alpha$, as time-based windows do not remove content, all what needs to be done is to reset the bounds of the interval considered. □

By this theorem, we can reduce model checking of a LARS formula φ on an input stream $S = (T, \upsilon)$, to model checking an ad-hoc PLTL formula constructed from φ, M and the specific time point $t \in T$, on a single path $\pi(M)$. In particular, we can do this for $T = [0, t]$, i.e., at the end of the input stream. Furthermore, we can transform the PLTL formula easily to an LTL formula that is initially equivalent.

In [BDTE18], two more results are given for LARS with sliding time-based windows. First, LARS formulas yield a strict fragment of PLTL, and thus of regular languages. This is shown by a more involved encoding that does not depend on the input stream. Furthermore, using intensional atoms and LARS programs, all regular languages can be expressed.

4.3 Continuous Query Language (CQL)

A particularly influential work in stream processing has been the Stanford Stream Data Manager (STREAM) [ABB+03] and its Continuous Query Language (CQL) [ABW03, ABW06]. The central idea is to reuse existing features from SQL and extend it with streams as additional data sources. To this end, different window operators are used to obtain recent snapshots of data, which are then essentially viewed as database relations.

Example 35 Following up on our running example from Chapter 3, we state a CQL query for expected arrival times of trams where no traffic jam has been reported at their last station within the last 20 minutes. Recall that the relation $plan(L, X, Y, D)$ records for line L the scheduled travel time D between station X and Y.

```
SELECT TRAM.ID, PLAN.Y, TY
FROM  TRAM[PARTITION BY ID ROWS 1], LINE, PLAN
WHERE TRAM.ID=LINE.ID AND LINE.L=PLAN.L AND
      TRAM.ST=PLAN.X AND TY=TRAM.T+PLAN.D AND
      NOT EXISTS
        (SELECT * FROM JAM[RANGE 20]
         WHERE JAM.ST=TRAM.ST)
```

Note that streams `TRAM` and `JAM` have designated timestamp fields "`T`", i.e., explicit attributes that state the time when the tuple occurred in the stream. ∎

In CQL, a *stream* is viewed as bag of *elements* of the form $\langle \mathbf{c}, t \rangle$, where $\mathbf{c}$ is a tuple (which we can view as vector of constants) and t a timestamp; a *relation* maps timestamps to bags of tuples. To translate between these concepts, the operational semantics of CQL builds on three operators:

- *Stream-to-relation* (S2R) operators apply window functions to the input stream to create a relation for recent tuples, i.e., those in the selected window.
- *Relation-to-relation* (R2R) operators can manipulate relations similarly as in relational algebra, respectively SQL.
- *Relation-to-stream* (R2S) translates back a relation into a stream for the output of continuous queries.

Our focus here is on the first two operators, the R2S operator only concerns how output is generated but does not influence the query semantics as such. The S2R operator allows us to consider streaming tuples as sets of atoms. The semantics of CQL thus essentially reduces to the R2R operator, once recent snapshots of streaming data have been selected by S2R. Due to this, we show that LARS programs capture CQL by exploiting two well-known translations: from SQL to relational algebra [DS90] and from relational algebra to Datalog [GUW09]. Let us call the former translation *RelAlg* and the latter *Dat*.

The idea is to have a 3-step process to obtain a Datalog program for a CQL query q:

(1) replace in `FROM` clauses the input sources (i.e., streams with window expressions) by virtual table names due to the renaming function *rel* as defined in Table 4.1. By replacing in CQL query q each occurrence of an input stream s by a relation $rel(s)$, we obtain a SQL query $rel(q)$.

(2) Apply *RelAlg* on this query to obtain a relational algebra expression.

(3) Apply *Dat* on the expression to obtain a Datalog program with a designated predicate $\hat{q}$ that reflects the resulting tuples.

More formally, we get for a CQL query q a Datalog program $\Delta_D(q) = Dat(RelAlg(rel(q)))$. Any static relation (table) `B` can be naturally encoded as

$$\Delta(\texttt{B}) := \{b(\mathbf{c}) \mid \mathbf{c} \text{ is a tuple in } \texttt{B}\}, \tag{4.3}$$

where the lower case b version of relation name `B` serves as predicate name for atoms; tuples $\mathbf{c}$ can be seen as vectors of constants.

We observe that LARS allows us to model the S2R operator. A snapshot of a stream `S` amounts to (i) applying an according window operator and then (ii) abstracting away time. The second step amounts to existential quantification over time, i.e., formulas of form $\boxplus^w \Diamond \varphi$. Table 4.1 lists the LARS window functions corresponding to those in CQL. We thus can derive each snapshot relation $rel(s)$ for a CQL input source s (as listed in the table) using a *snapshot rule* of form

$$\Delta_L(s) := rel(s)(\mathbf{V}) \leftarrow \boxplus^{w(s)} \Diamond s(\mathbf{V}), \tag{4.4}$$

where the lower case s version of stream name `S` serves as predicate name, and $\mathbf{V}$ is the list of variables corresponding to the attributes of tuples in `S`. We refer to static relations and input streams (with window expressions) uniformly as *input sources*. We thus obtain a LARS program

$$\Delta_L(q) = \Delta_D(q) \cup \{\Delta_L(s) \mid s \text{ is an input source in } q\}.$$

For a set Q of queries, we simply take respective unions, i.e., $\Delta_x(Q) = \bigcup_{q \in Q} \Delta_x(q)$, $x \in \{L, D\}$.[3]

Example 36 We give a Datalog translation $\Delta_D(q)$ of the CQL query q in Example 35. (Note that due to the exact translation from SQL and potential optimizations the intermediate relational algebra representation might vary and thus the specific set of derived Datalog rules. The employed translation is detailed in the Appendix.) Let $\mathbf{T} = Id_1, S_{\mathrm{T}1}, T_1$; $\mathbf{L} = Id_2, L_2$; $\mathbf{P} = L_3, X_3, Y_3, D_3$; $\mathbf{J} = S_{\mathrm{T}4}, T_4$ (subscripts for variables

[3]Note that LARS formulas of form $\boxplus^w \Diamond \varphi$ could by themselves be viewed as relation names and interpreted as Datalog programs.

Input source s in `FROM` clause	Relation $rel(s)$	$w(s)$
`S[RANGE L]`	$s\_range\_\boldsymbol{L}$	τ^{L}
`S[RANGE L SLIDE D]`	$s\_range\_\boldsymbol{L}\_slide\_\boldsymbol{D}$	$\tau^{L(D)}$
`S[RANGE UNBOUNDED]`	s_range_unb	τ^{∞}
`S[NOW]`	s_range_0	τ^{0}
`S[ROWS N]`	$s\_rows\_\boldsymbol{N}$	$\#^{N}$
`S[PARTITION BY X1,...,Xk ROWS N]`	$s\_part\_\boldsymbol{X1}\_\ldots\_\boldsymbol{Xk}\_rows\_\boldsymbol{N}$	$\mathrm{p}^{\mathrm{idx},n}$

Table 4.1: Translation function rel and LARS window function $w(s)$

serve to reflect their origin in the same schema in a readable way).

$$
\begin{aligned}
q_0(\mathbf{T},\mathbf{L},\mathbf{P}) &\leftarrow tram\_part\_\boldsymbol{ID}\_rows\_\boldsymbol{1}(\mathbf{T}), line(\mathbf{L}), plan(\mathbf{P}).\\
q_1(\mathbf{T},\mathbf{L},\mathbf{P}) &\leftarrow q_0(\mathbf{T},\mathbf{L},\mathbf{P}), ST_1 = X_3, Id_1 = Id_2, L_2 = L_3.\\
q_2(\mathbf{T},\mathbf{J}) &\leftarrow tram\_part\_\boldsymbol{ID}\_rows\_\boldsymbol{1}(\mathbf{T}), jam\_range\_\boldsymbol{20}(\mathbf{J}).\\
q_{12}(\mathbf{T},\mathbf{L},\mathbf{P},\mathbf{J}) &\leftarrow q_1(\mathbf{T},\mathbf{L},\mathbf{P}), q_2(\mathbf{T},\mathbf{J}).\\
q'_{12}(\mathbf{T},\mathbf{L},\mathbf{P}) &\leftarrow q_{12}(\mathbf{T},\mathbf{L},\mathbf{P},\mathbf{J}).\\
q(Id_1, Y_3, TY) &\leftarrow q_1(\mathbf{T},\mathbf{L},\mathbf{P}), \neg q'_{12}(\mathbf{T},\mathbf{L},\mathbf{P}), TY = T_1 + D_3.
\end{aligned}
$$

Informally, q_0 captures the cross product of relations `LINE` and `PLAN` as given in the `FROM`-clause, and the relation corresponding to the window on stream `TRAM`. The selection based on the statement `TRAM.ID=LINE.ID AND LINE.L=PLAN.L` in the `WHERE`-clause is captured in predicate q_1. The cross product of recent tram appearances at stations and traffic jams is then reflected in q_2 and the join with q_1 yields q_{12}, which thus captures tram appearances that shall not be considered. In order to remove these, jam information is projected away to obtain predicate q'_{12}. Finally, those variable groundings for q_1 are reported that are not groundings for q'_{12}, and in addition the calculated arrival time TY which adds the planned travel time D_3 to occurrence time T_1 of the last station. (Note that we explicitly model arrival times in tuples. Thus, they remain accessible after S2R, resp. in the Datalog and LARS encodings.)

Using snapshot rules of form (4.4), we obtain a LARS program $\Delta_L(q)$ by adding the following rules (idx, n are from Example 27):

$$
\begin{aligned}
tram\_part\_\boldsymbol{ID}\_rows\_\boldsymbol{1}(\mathbf{T}) &\leftarrow \boxplus^{\mathrm{idx},n}\Diamond tram(\mathbf{T}).\\
jam\_range\_\boldsymbol{20}(\mathbf{J}) &\leftarrow \boxplus^{20}\Diamond jam(\mathbf{J}).
\end{aligned}
$$

■

To establish the correspondence between the result of a set Q of CQL queries and its LARS translation $\Delta_L(Q)$, we first build a conversion of the input streams in Q to a LARS data stream. (Recall that LARS considers only a single stream which can be

virtually split, e.g., by partition-based windows.) Without loss of generality, we assume that Q is evaluated on static relations $\mathtt{B}_1, \ldots, \mathtt{B}_m$ and input streams $\mathtt{S}_1, \ldots, \mathtt{S}_n$, and that any stream is only used in one place in the `FROM` clause in a single query (we can always duplicate streams and rename them). We consider the union of these input streams, given by

$$\mathtt{S} = \{\langle \mathbf{c}_{ij}, t_{ij} \rangle \mid 1 \leq i \leq n, 1 \leq j \leq m_i\},$$

where the element $\langle \mathbf{c}_{ij}, t_{ij} \rangle$ represents the tuple $\mathbf{c}_{ij}$ that occurs at the j^{th} position at time t_{ij} in stream $\mathtt{S}_i$ (with m_i elements). We use the (lower case) name s_i of CQL stream $\mathtt{S}_i$ as predicate symbol of according atoms and thus obtain the LARS data stream by

$$\begin{aligned} \Delta(\mathtt{S}) &= (T, \upsilon) \text{ such that} \\ T &= [\min\{t_{ij}\}, \max\{t_{ij}\}] \text{ and} \\ \upsilon(t) &= \{s_i(\mathbf{c}_{ij}) \mid t_{ij} = t\} \text{ for all } t \in T; \end{aligned}$$

where $1 \leq i \leq n$ and $1 \leq j \leq m_i$. Similarly, we define for $\mathtt{B} = \mathtt{B}_1, \ldots, \mathtt{B}_m$ the atom set $\Delta(\mathtt{B}) = \bigcup_{i=1}^{m} \{b_i(\mathbf{c}) \mid \mathbf{c} \in \mathtt{B}_i\}$. Let $res(q, t)$ denote the set of resulting tuples of CQL query q at time t and let $res(Q, t) = \bigcup_{q \in Q} res(q, t)$. The following theorem shows that the translation Δ_L faithfully captures CQL. For a set A of atoms and a set Q of CQL queries let $A|_Q$ denote the set of all tuples $\mathbf{c}$ such that $\hat{q}(\mathbf{c}) \in A$ for some query $q \in Q$ (recall that $\hat{q}$ is the "output" predicate of the Datalog transformation of q).

Theorem 17 *Let Q be a set of CQL queries to be evaluated on input streams $\mathtt{S} = \mathtt{S}_1, \ldots, \mathtt{S}_n$ and background relations $\mathtt{B} = \mathtt{B}_1, \ldots, \mathtt{B}_m$, $P = \Delta_L(Q)$, and t a time point. Moreover, let $M = \langle I, W, \Delta(\mathtt{B}) \rangle$ such that W is implicit by Table 4.1 and windows mentioned in Q. Then,*

(i) If $I = (T, \upsilon)$ is an answer stream of P for $\Delta(\mathtt{S})$ at t, then $\upsilon(t)|_Q = res(Q, t)$.

(ii) There exists an answer stream $I = (T, \upsilon)$ of P for $\Delta(\mathtt{S})$ at t s.t. $\upsilon(t)|_Q = res(Q, t)$.

Intuitively, (i) establishes the soundness and (ii) the completeness of the translation Δ_L.

The idea of the proof is as follows. Consider a set Q of CQL queries and its translations $\Delta_D(Q)$ to Datalog and $\Delta_L(Q)$ to LARS, and moreover the data stream $\Delta(\mathtt{S})$ corresponding to CQL input stream $\mathtt{S}$. First, we observe that the Datalog program $\Delta_D(Q)$ is an acyclic program and thus has as well-known a single answer set. Using the snapshot of the streaming data (i.e., the result of the S2R operator) as input, we thus get by the correctness from *RelAlg* and *Dat* that the result of Q is captured by the answer set of $\Delta_D(Q)$. As noted above, the result of the S2R operator on an input source s amounts to abstracting away the temporal information. This step is carried out in LARS by existential temporal quantification with $\Diamond$ in the according window as listed in Table 4.1. We observe that snapshot rules, i.e. $\Delta_L(Q) \setminus \Delta_D(Q)$, add a stratified layer to $\Delta_D(Q)$ and faithfully derive the relations $rel(s)$ as follows. Provided encoding $\Delta(\mathtt{S})$ (and background data $\Delta(\mathtt{B})$ for static relations $\mathtt{B}$), $rel(s)(\mathbf{c})$ will be derived iff $\mathbf{c}$ is a tuple in the snapshot of input source s in Q: any ground snapshot rule of form $rel(s)(\mathbf{c}) \leftarrow \boxplus^{w(s)} \Diamond s(\mathbf{c})$ must be satisfied when the formula $\boxplus^{w(s)} \Diamond s(\mathbf{c})$ is satisfied, thus

the rule head $rel(s)(\mathbf{c})$ is concluded if the tuple $\mathbf{c}$ is contained in the snapshot; the only-if part is ensured by minimality of answer streams and the fact that relation names do not occur elsewhere as rule heads in the translation. From this also follows that the interpretation of these predicates must coincide in $\Delta_D(Q)$ and $\Delta_L(Q)$; the latter contains no further rules not contained in $\Delta_D(Q)$. It thus follows that the answer set of $\Delta_D(Q)$ corresponds to the answer stream of $\Delta_L(Q)$. That is to say, given the answer stream $I = (T, \upsilon)$ of $\Delta_L(Q)$ for $\Delta(\mathtt{S})$ at time t, $\hat{q}(\mathbf{c}) \in \upsilon(t)$ iff $\mathbf{c} \in res(Q, t)$. More details can be found in the Appendix. We conclude this section by contrasting CQL and LARS.

Differences between CQL and LARS

The continuous query language extends SQL in a similar way as LARS extends ASP. As a consequence, the semantic differences between SQL and ASP naturally carry over to CQL and LARS. The first difference in this regard is a conceptual one in their respective definitions: CQL has an operational semantics, while the LARS semantics is model-based; CQL specifies how streaming tuples are evaluated in stepwise processing of a query, while LARS merely formalizes what a program entails, given a stream and a time point. CQL works on multisets (bags) of tuples per time point, while LARS uses sets. Expressivity features of ASP carry over to LARS, in particular those that SQL (respectively CQL) has not. LARS is a paradigm for expressive reasoning, while CQL targets query processing. One way to think about this distinction is that SQL-like querying usually centers around filtering, joining and aggregating data in a deterministic way (in the sense that the set of returned tuples is unique) while reasoning typically involves abstractions, constraints, complex negation (potentially in a cyclic way) and non-determinism.

More specifically, core reasoning features of ASP (respectively LARS) that are not directly available in SQL (respectively CQL) include supported minimal model by means of the stable model semantics, the possibility to enumerate multiple solutions for use cases with choices, nonmonotonic negation with defaults, loops through negation, and constraints. The answer set semantics (respectively answer stream semantics) is fully declarative in the sense that the order of rules does not influence the output, and the possibility to express combinatorial problems makes it applicable for constraint satisfaction problems. For further discussion on the semantic features of ASP we refer the interested reader to [EIK09, BET11, BET16].

On top of the properties that LARS inherits from ASP, generic window functions are the central mechanism for handling streams. The stream-to-relation mechanism of CQL serves as an elegant abstraction over specific window functions. However, as explained above, viewing streams as relations implicitly comes with existential quantification with respect to time. By dropping temporal information, any window function w in CQL is captured in LARS by a window operator $\boxplus^w$, followed by $\Diamond$.

Further differences between CQL and LARS concern the additional temporal modalities ($\Box$ and @), and in particular, their combination with generic windows. We will explore this further in Section 4.6, when we highlight LARS-specific features from an application oriented perspective.

4.4 Complex Event Processing: ETALIS

Related to stream processing is *complex event processing* (CEP), where one deals with the derivation of complex events, that typically span over temporal intervals, based on (atomic) events, that occur at time points. We briefly study the relation between LARS and the CEP language ETALIS [AFR+10, ARFS12]. This allows us to draw a line between stream reasoning and CEP by means of LARS. We present here the main ideas and the results from a conceptual perspective; formal details can be found in the Appendix.

In ETALIS, an *event stream* ϵ associates *atomic events* (represented as ground atoms) with time points, i.e., non-negative rational numbers. For comparability with LARS, we consider here only natural numbers. *Complex events* can be described by rules due to so-called *event patterns*, which resemble interval relations of Allen's interval algebra [All83]. For instance the pattern x SEQ y matches interval $\langle t_1, t_4\rangle$, if there are intervals $\langle t_1, t_2\rangle$ and $\langle t_3, t_4\rangle$ assigned to events x and y, respectively, such that $t_2 < t_3$. The pattern x AND y selects the temporal overlaps, x EQUALS y selects intervals assigned to both x and y, etc. These interval expressions can then be composed by rules of the form $a \leftarrow pt$, where a is an atom and pt is an event pattern. Intuitively, every interval that matches the rule body (pt) must be assigned to the head (a).

To formalize this intuition, an ETALIS interpretation $\mathcal{I}$ is a function that maps atoms (events) to sets of pairs $\langle t_1, t_2\rangle \in \mathbb{N} \times \mathbb{N}$ representing intervals $[t_1, t_2]$. An intepretation $\mathcal{I}$ *satisfies* a rule $r = a \leftarrow pt$, if $\mathcal{I}(pt) \subseteq \mathcal{I}(a)$. Furthermore, given an event stream ϵ and a rule base $\mathcal{R}$, an interpretation $\mathcal{I}$ is a *model* for $\epsilon, \mathcal{R}$, if it

(i) maps each atomic event a to the interval $\langle t, t\rangle$ if a occurs in ϵ at time point t, and

(ii) satisfies each rule $r \in \mathcal{R}$.

The semantics of ETALIS is then defined in terms of minimal models, which require that only those intervals are assigned to atoms that can be derived by rules based on the event stream. The defined minimal model is unique and can be efficiently constructed bottom up from the event stream, repeatedly adding the intervals matching a rule's body to the assignment of its head until a fixed point is reached.

Intervals in LARS. In contrast to ETALIS, the semantics of LARS is based on time points. Nevertheless, we can represent intervals in LARS and thus capture the ETALIS event patterns under some restrictions. Consider a window function $w_{[\ell,u]}$ that selects the (maximal) substream of the interval $[\ell, u]$. Given a formula α, we define the abbreviations

$$\begin{aligned} [\![\ell, u]\!]\alpha &:= \boxplus^{w_{[\ell,u]}} \Box\alpha\,, \\ \langle\!\langle \ell, u \rangle\!\rangle\alpha &:= [\![\ell, u]\!]\alpha \ \wedge\ @_{\ell-1}\neg\alpha \ \wedge\ @_{u+1}\neg\alpha\,. \end{aligned}$$

That is to say, formula $[\![\ell, u]\!]\alpha$ holds iff α holds at every time point in the interval $[\ell, u]$, regardless of the query time. Similarly, $\langle\!\langle \ell, u \rangle\!\rangle\alpha$ holds iff $[\ell, u]$ is a *maximal* interval in which α always holds.

Example 37 Consider two events, x and y, which hold in the intervals $\langle t_1, t_2\rangle$ and $\langle t_3, t_4\rangle$, respectively, and assume $t_2 < t_3$. An ETALIS rule $r = z \leftarrow x$ SEQ y thus assigns the interval $\langle t_1, t_4\rangle$ to z. With the above syntactic abbreviations, we may express r in LARS as

$$[\![t_1, t_4]\!]z \leftarrow \langle\!\langle t_1, t_2\rangle\!\rangle x,\ \langle\!\langle t_3, t_4\rangle\!\rangle y,\ t_2 < t_3.$$

That is, if $[t_1, t_2]$ is a maximal interval in which x holds, and $[t_3, t_4]$, where $t_2 < t_3$, is a maximal interval in which y holds, then z must hold at every time point in $[t_1, t_4]$. ∎

However, this straightforward encoding does not suffice to express ETALIS in LARS. The essential problem arises from overlapping intervals (for the same event/formula). LARS assigns atoms to a single timeline by an evaluation function $\upsilon : T \to 2^{\mathcal{A}}$. Unless an encoding makes use of time points in atoms, we can encode intervals only by assigning atoms to consecutive time points. Adjacent or overlapping intervals for the same atom cannot be distinguished, they all amount to a merged view of them. For instance, consider an event e that is assigned to $\langle 1, 4\rangle$ and $\langle 3, 5\rangle$ in ETALIS. By naively constructing a LARS interpretation υ, we would assign $\upsilon(t) = \{e\}$, for $t = 1, \ldots, 4$ and $t = 3, \ldots, 5$, and then only be able to read off the merged interval $[1, 5]$ for e. Using the temporal controls of the LARS framework, one could not distinguish the intervals $[1, 4]$ and $[3, 5]$ for the same atom (or formula) e.

For the sake of illustrating the capabilities of LARS regarding intervals, we thus consider *separable* ETALIS interpretations, i.e., where such overlaps do not occur. If the minimal model of an event stream ϵ and a rule base $\mathcal{R}$ is separable, we also call the pair $(\epsilon, \mathcal{R})$ *separable*. We point out that the notion of minimality in LARS is based on set inclusion, whereas ETALIS defines minimality in terms of minimal length and the supportedness of inferred intervals. Notably, ETALIS defines a form of negation (i.e., the NOT-pattern) which is compatible with a bottom-up fixed-point evaluation. A direct translation to LARS would in this case give rise to multiple LARS models in general. When confining to positive ETALIS programs with separable minimal models, a straightforward translation as indicated in Example 37 captures the ETALIS semantics.

Theorem 18 *Let ϵ be an event stream, let $\mathcal{R}$ be a positive rule base (i.e., without negation) and let $\mathcal{I}$ be a separable interpretation for $\epsilon, \mathcal{R}$. Then one can construct a LARS input stream Δ^{ϵ}, a program $\Delta^{\mathcal{R}}$, and an interpretation stream $\Delta^{\mathcal{I}} = (T, \upsilon)$ such that for each $t \in T$, $\mathcal{I}$ is the minimal model for $\epsilon, \mathcal{R}$ iff $\Delta^{\mathcal{I}}$ is the (unique) answer stream of $\Delta^{\mathcal{R}}$ for Δ^{ϵ} at time t relative to $W = \{w_{[\ell,u]} \mid \ell, u \in \mathbb{N}, \ell \leq u\}$ and $B = \emptyset$.*

Taking LARS and ETALIS as reference languages, we can view separability as the dividing line between stream reasoning and complex event processing. If we view ETALIS from a natural logic-oriented point of view, where a formula can only hold or not hold at a given time point (given an underlying set semantics), we obtain correspondence.

Corollary 4 *Let $\epsilon, \mathcal{R}$ be an event stream, $\mathcal{R}$ be a positive rule base such that the minimal model $\mathcal{I}$ of $\epsilon, \mathcal{R}$ is separable and let $\Delta^{\mathcal{I}} = (T, \upsilon)$, i.e., the answer stream of $\Delta^{\mathcal{R}}$. Then,*

for all atoms $a \in \mathcal{A}$ and for all $t \in T$, $a \in \upsilon(t)$ iff there exists an interval $\langle t_1, t_2 \rangle \in \mathcal{I}(a)$ such that $t \in [t_1, t_2]$.

We point out that these correspondence results are of more theoretical interest, since separability is a very limiting condition in practice, and it is also a data dependent one. The presented investigation on ETALIS allowed us to study the formal capabilities of LARS regarding intervals by means of straightforward interval windows. The suggested intuitive translation is, regarding intervals, less expressive than ETALIS, where an atom can be assigned to overlapping intervals. Conceptually, this would amount to parallel timelines in LARS.

Apart from the presented natural encoding, another way to reflect intervals is due to additional arguments in atoms. For instance, we can reflect that an atom a holds during the interval $[7, 19]$ by using a fresh atom $a'(7, 19)$. Clearly, this allows one to access also overlapping intervals for the same (original) atom. We only need a rule of the form $a'(T, T) \leftarrow @_T\, a$ for every atom in a program, and then translate the ETALIS pattern definitions into normal logic programs. This way, ETALIS is subsumed already by ASP, and thus LARS: the unique model of ETALIS is obtained by a fixed-point computation, and a standard bottom-up evaluation can be emulated in ASP, including the NOT pattern.

4.5 Semantic Web: C-SPARQL, CQELS

Among research initiatives for the Semantic Web, RDF Stream Processing (RSP) emerged to address the question of querying heterogeneous streams. The RSP community is interested in extending SPARQL for streams in a similar way as CQL builds on SQL. In particular, C-SPARQL [BBC$^+$09, BBC$^+$10a, BBCG10] and CQELS [PDPH11] employ an operational semantics that is based on the CQL approach of reducing stream reasoning to relational processing between a pre-processing of input streams and a post-processing towards output streams.

In [DBE15a, DBE15b] we investigated these two query languages for RDF data. We studied their differences which arises mainly due to the different execution modes. While C-SPARQL is *pull-based*, i.e., repeatedly returns a query result after a fixed temporal interval, CQELS is *push-based*, i.e., reports results after every new input. We presented the comparative analysis by first formalizing these execution modes semantically for LARS programs; we then gave translations for the two RSP languages to LARS in a similar way as for CQL.

One difference between CQL and the SPARQL extensions are due to the fact that RDF graphs (i.e., sets of triples) do not come with a schema. While for CQL the integration of multiple input sources (tables and streams) is clear, there are different ways to integrate different input streams that arrive in RDF format. While C-SPARQL merges the relational snapshots into one graph (by stating them in the FROM-part of the query), CQELS provides explicit access to each input stream (in the WHERE clause of the query).

The central idea of capturing the push- and pull-based execution modes in LARS is to introduce an auxiliary atom c to rule bodies in order to control potential rule firing. The push-based mode will infer c whenever any atom holds in window $\boxplus^0$ that contains only the current time point. On the other hand, the encoding of pull-based execution amounts to testing for $\boxplus^0@_T\top$ whether current time T is a multiple of the query interval; only in this case, c shall hold.

Furthermore, window expressions of C-SPARQL and CQELS can easily be encoded as window operators in LARS. Note that LARS allows for using any kind of computable window function that does not need further information than the input stream and the query time point. In fact, the time-based and tuple-based window functions as presented in Sections 3.1.3 and 3.1.4 correspond to those used in considered RSP languages.

Finally, the translation from RSP queries to LARS is based on an existing reduction from SPARQL to Datalog rules [Pol07]. For C-SPARQL, it suffices to use a uniform representation of triples $t(s, p, o)$ that are available as input $i(s, p, o, x)$ in stream source x at some time point in the considered snapshot. Thus, the encoding essentially takes the form $t(s, p, o) \leftarrow \boxplus^w \Diamond i(s, p, o, x)$, where $\boxplus^w$ is the according window translation. For CQELS, we additionally have to disambiguate the stream source due to the so-called stream graph pattern as stated in the original WHERE clause.

Based on these translations, questions on the semantics of C-SPARQL and CQELS can then be made precise; in particular, when syntactically similar queries indeed produce the same results. In [DBE15a] we formalized a notion of agreement and gave sufficient conditions for agreement for sliding time-based windows (under some restrictions). Our results formally reflect the drastic effect of execution modes on the query results from a semantic perspective.

4.6 Discussion

We conclude this chapter by discussing implications for using LARS in practice, and in particular, its distinguishing semantic features.

Sections 4.2-4.4 explored the formal relationship between LARS and related formalisms, and we discussed further related work in the previous section. We now elaborate on practical implications of such differences, in particular, which kind of features in applications can be targeted with LARS but not with other approaches, or where LARS is easier to apply. In doing so, we leave aside the theoretical framework side of LARS to formalize the semantics of other languages and concentrate on its potential use as genuine reasoning language.

Regarding temporal reasoning in the spirit of LTL or MTL, the biggest difference regarding applications is that LARS formulas and programs center around the idea of dropping data, whereas temporal reasoning is usually concerned with specifications of guarantees over infinite timelines. That is to say, languages for temporal reasoning are more geared towards monitoring and verification rather than computations that involve abstractions or auxiliary information (as in ASP/LARS rule heads). Disregarding discussed encoding potentials (cf. Section 4.2), we imagine that from an application

perspective the choice between temporal reasoning and stream reasoning with windows is determined by these aspects. On the other hand, combinations of their respective features point at intriguing future research issues, i.e., augmenting temporal reasoning languages like LTL or MTL with generic window functions, respectively considering infinite timelines in variants of LARS.

Example 27 (page 58) illustrated some of the benefits that come with LARS-based reasoning. We compute expected arrival times of trams in rule (3.6) and then reason in rule (3.7) which pairs of expected arrivals yield good connections. First to mention here is the abstraction which takes place: by introducing rule (3.6) the condition of the body gets a name that can be reused in other rules. Clearly, standard stream query languages such as those building on CQL can similarly introduce abstraction by means of nested queries. However, this way, according translations of more involved programs (i.e., with many rules) quickly become unreadable. On the technical side, it is possible to chain such query nesting but it is not possible to impose mutual conditions on their respective evaluations, i.e., cycles. This is related to a commitment to single models in most stream processing approaches, as we shall further discuss below.

The ability to compute transitive closures is among the semantic assets inherited from ASP. We may assume that the background facts *plan* for travelling times are specified only for consecutive stations. Nevertheless, we can then compute expected arrival times at further stations by adding the following rule:

$$@_{T_Y}\, exp(Id, Y) \leftarrow @_{T_X}\, exp(Id, X), line(Id, L), plan(L, X, Y, D), T_Y = T_X + D. \quad (4.5)$$

The rule expresses the following: We expect a tram Id at station Y at time T_Y if it is already expected D minutes before at the previous station X at time T_X, where D is the planned time it takes the tram to get from X to Y. Importantly, the same rule then applies for station Y and its subsequent station Z, and so forth. In other words, we can compute expected arrival times for all remaining stations of that line. Given in addition similar rules for connecting lines, all potential future trajectories can be reasoned about in principle. Such recursive reasoning is typical for ASP but rarely available in stream query languages; LARS makes it available in flexible combination with streaming data. Note in particular that predicate *exp* depends (in rule (3.6)) on the current inexistence of traffic jams. Thus, any path of pairwise reachable stations that depends on station X (in terms of the predicate *exp*) will be dismissed as soon as a traffic jam is reported at X. This nonmonotonic effect of negation that additional information may lead to the retraction of previous conclusions is an important feature if one wants to reason with defaults such as the normal flow of traffic.

The possibility to express loops through predicates also in the presence of negation makes LARS suitable for combinatorial problems and dynamic constraint satisfaction problems. We illustrated this in Example 28, where we used the information on pre-selected good connections (due to rules (3.6) and (3.7)) to specify a choice between changing trams or skipping a good connection. This was established by the predicate *change* which is in a cyclic dependency with predicate *skip*. Using query languages in the spirit of CQL, we could specify some conditions for changing trams in one query and

for skipping connections in another, but it would not be feasible to write a query that outputs both options as a result of the same query due to the mutual dependence and the inherent determinism. Apart from the limitations regarding cyclic dependencies, the only way to express multiple options in such deterministic languages would be to encode their enumeration as such in the query result. This however may lead to large outputs, if an exponential number of options exists. In contrast, a nondeterministic language allows one to produce the options declaratively one by one, in an enumerative fashion, where one may stop at any time; this would in a deterministic language require operational elements, as e.g. in Prolog.

Limitation to single solutions is also a shortcoming of ETALIS. It can express combinations of intervals that go beyond the temporal capabilities of LARS but it has a rudimentary form of negation and no other means that could serve to express multiple solutions. When using time points as additional arguments in atoms, LARS can emulate ETALIS. On the other hand, it is not feasible to encode the rich semantic properties of LARS in ETALIS.

Languages like CQL that adopt the snapshot semantics (which drop the temporal information of obtained windows) cannot express temporal reasoning (beyond what can be encoded with explicit timestamps in tuples). By contrast, LARS provides fine-grained control. This not only allows one to reason about the contents of windows as such (as in CQL) but also about temporal occurrence. In other words, LARS can not only express *whether* something holds but also that something *always* holds or *when* something is true. Consider, for instance, the modality $@_T$ in the head of rule (3.6). This places the inference explicitly to a time point that is defined in the rule body. In that regard the @-operator not only serves to match or select time points from streaming data, but to reason also about the temporal information of inferred data. An example of such reasoning is realized by the formula

$$@_T \boxplus^{+5} \Diamond exp(Id_2, X) \tag{4.6}$$

in rule (3.7): based on the specific time T of the expected arrival of the other tram Id_1 in that rule, the formula is satisfied if there is an expected tram at a station within the *next* 5 minutes *relative to* T. That is to say, we control at which time T the window $\boxplus^{+5}$ is selected, where T itself is inferred information. Note further that this window is non-standard in that it selects future time points. LARS itself is agnostic about which window functions to employ; everything that computes substreams is applicable. We emphasize that such windows may include also inferred data, as do for instance our time-based windows. As a consequence, the full power of LARS operator combinations extends to inferred information. Beyond semantic properties inherited from ASP, it is exactly this compositionality of generic windows and temporal modalities ($\Diamond$, $\Box$ and @) that sets LARS apart from existing streaming languages.

In the works on Content-Centric Networking mentioned above (cf. Section 3.2.4) we made use of the compositional nature of LARS operators, staying even within the plain LARS fragment. In combination with nested window operators (and reset), the possibility to navigate along the timeline with temporal modalities leads to highly

expressive reasoning capabilities, as examined in the complexity analysis. Example 20 (page 52) illustrated the use of window nesting in combination with temporal modalities, where we formalized

$$\varphi = \boxplus^{60}\Box \triangleright \boxplus^{5}\Diamond s \tag{4.7}$$

to check whether a signal s appeared always within 5 minutes during the last 60 minutes. While nested queries are often available in other approaches, specifications of temporal conditions on their evaluation are not. Note that φ can be seen as reasoning over a *sequence* of nested reasoning steps, thus yielding a formal counterpart to evaluating re-runs ($\boxplus^{60}\Box\psi$) of the same query ($\psi = \boxplus^{5}\Diamond s$, after stepping out with reset $\triangleright$). This way LARS can express evaluations of evaluations (in arbitrary depth) and in this way serve for hierarchical reasoning.

CHAPTER 5

Semantic Characterizations of Equivalent LARS Programs

In Chapter 1, we stressed the trade-off between throughput and expressiveness as a central challenge in stream reasoning. In practice, given a fixed reasoning task, one may exploit domain specific properties to gain performance by means of specific algorithms. To some degree, optimizing or tailoring a program for a specific use case is also possible in rule-based or other declarative approaches.

However, highly tailored queries, programs or encodings come with costs. First, the development effort increases for the programmer who has to invest time in understanding the structure of the problem and the data. Second, maintaining optimized code bases is difficult and error-prone since they are typically less readable and less generic. Third, the idea of a low-level management towards controlling the solving process for better performance undermines the conceptual idea of declarative languages, which should simplify the query task by providing a more intuitive, high-level representation. After all, the essence of declarative programming lies in abstracting away the specifics of how the evaluation is carried out. It is the task of solvers or preprocessing tools to translate human readable representations (that may be computationally expensive if processed exactly as stated) into equivalent ones that can be computed more efficiently, i.e., by means of equivalence preserving transformations. However, this presupposes the ability for checking when two programs are equivalent in the first place.

Example 38 Consider the rule $r = a \leftarrow \boxplus^{\#10}\Diamond b \wedge \boxplus^{\#10}\Diamond c$, which will conclude a if among the last 10 atoms both b and c occur. Clearly, it is equivalent to rule $r' = a \leftarrow \boxplus^{\#10}(\Diamond b \wedge \Diamond c)$. Depending on how the specific formula structure is operationally handled, one of the two rules might be preferable. For instance, conjunctions are a natural point to parallelize evaluation into two nodes. In r, the conjunction is outside windows, requiring window evaluation at both nodes, while in r' the parallelization can be carried out within the same physical representation of a single window. We further

note that rule $r'' = a \leftarrow \boxplus^{\#10}\Diamond(b \wedge c)$ is not equivalent, since it enforces b and c to appear at the same time point. ∎

As a first step towards optimizing LARS programs for improving the efficiency of evaluation, we are thus interested in suitable notions of equivalence, i.e., to obtain criteria when two programs will produce the same results. In particular, the aim is to provide semantic characterizations of different notions of equivalence similar as for Answer Set Programming [LPV01, EF03, EFW07]. Characterizing equivalence between LARS programs in purely logical terms is challenging due to a non-structural definition of the FLP-semantics [FLP04] defined for them, which imposes some limitations. Yet another difficulty arises from the generic definition of window operators.

Outline

This chapter is structured as follows. Section 5.1 introduces practically relevant notions of equivalence for LARS programs that extend well-known equivalence relations for logic programs, i.e., ordinary, strong and uniform equivalence. Moreover, we introduce *data equivalence* for streams. In Section 5.2 we define a novel logic called bi-*LARS* which we employ in Section 5.3 to capture the FLP-based semantics of a large fragment of LARS programs, including plain LARS (cf. Section 4.1). In Section 5.4 we lift model-theoretic characterizations of strong/uniform/relativized uniform equivalence from the ASP literature to the stream setting to characterize the defined equivalence relations. Data equivalence is then obtained as a special case of relativized uniform equivalence. In Section 5.5 we show how a monotone variant of bi-LARS leads to an extension of the logic of Here-and-There [Hey30] for our setting. We thus get a link to equilibrium logic [LPV01] for a class of programs. Section 5.6 summarizes complexity results for the developed equivalence notions. The results give a first entry point towards optimizing LARS programs at the semantic level.

Publications

This chapter is based on [BDE16], which did not include proofs. In addition, we formally present the characterization result for relativized uniform equivalence. Complexity results from [BDE16], established by co-authors, are summarized in Section 5.6.

5.1 Equivalence Notions

We now introduce equivalence notions for stream reasoning within the LARS framework. Given a timeline T, we say a set A of @-*atoms*, i.e., formulas of form $@_t a$ where $a \in \mathcal{A}$, has *time references in* T, if $\{t \mid @_t a \in A\} \subseteq T$. This notion carries over naturally for programs, i.e., a program P has *time references in* T if $\{t \mid @_t a \text{ occurs in } P\} \subseteq T$. Recall that by $\mathcal{AS}(P, D, t)$ we denote the set of answer streams of P for D at time t.

Inspired by the rich literature on equivalence notions for Answer Set Programming, we define the following for LARS programs.

Definition 22 (Equivalence Notions) *Let T be a timeline, $D = (T, \upsilon)$ be a data stream, and let $t \in T$ be a time point. We say two LARS programs P and Q are*

(i) (ordinary) equivalent *(for D at t), denoted by $P \equiv Q$, if*

$$\mathcal{AS}(P, D, t) = \mathcal{AS}(Q, D, t)\,; \tag{5.1}$$

(ii) strongly equivalent *(for D at t), denoted by $P \equiv_s Q$, if*

$$\mathcal{AS}(P \cup R, D, t) = \mathcal{AS}(Q \cup R, D, t) \tag{5.2}$$

for all LARS programs R with time references in T;

(iii) uniformly equivalent *(for D at t), denoted by $P \equiv_u Q$, if*

$$\mathcal{AS}(P \cup F, D, t) = \mathcal{AS}(Q \cup F, D, t) \tag{5.3}$$

for sets F of @-atoms with time references in T;

(iv) data equivalent *(for D at t), denoted by $P \equiv_d Q$, if*

$$\mathcal{AS}(P, D \cup S, t) = \mathcal{AS}(Q, D \cup S, t) \tag{5.4}$$

for all data streams S with timeline T.

Intuitively, (i)-(iii) can be seen as extensions of corresponding notions in ASP [LPV01, EFW07] to the LARS setting: two ordinary logic programs P and Q are said to be (ordinary) *equivalent* if they possess the same answer sets, and *strongly* (respectively *uniformly*) *equivalent*, if the answer sets of $P \cup R$ and $Q \cup R$ coincide for every program (respectively set of facts) R. Thus, when considering LARS programs, these notions emerge when we restrict to the fragment of ordinary logic programs (i.e., using neither windows nor temporal operators), and considering a void data stream $D = ([0, 0], \emptyset)$ at time point 0. On the other hand, data equivalence is well-known in the database area and plays an important role in stream reasoning, as the possibility to drop data is crucial to gain performance. The consideration of all rules respectively facts for addition accounts for the nonmonotonic nature of answer streams, since replacing ordinary equivalent programs P and Q within the context of other rules R might lead to different answer streams.

5.2 Bi-Structural LARS Evaluation

We now define an extended variant of LARS semantics, where formulas (respectively programs) are evaluated on a pair of streams $(S_\mathrm{L}, S_\mathrm{R})$ at the same time. We will later consider a substream relation $S_\mathrm{L} \subseteq S_\mathrm{R}$ on according models similar to the logic of Here-and-There [Hey30] which was extensively studied in relation to equivalence notions for Answer Set Programming.

In the sequel, we use the following notation. Given streams $S_{\mathrm{L}} = (T, \upsilon_{\mathrm{L}})$ and $S_{\mathrm{R}} = (T, \upsilon_{\mathrm{R}})$, we call $\mathbf{S} = (S_{\mathrm{L}}, S_{\mathrm{R}})$ a *bi-stream* and $\mathbf{M} = \langle \mathbf{S}, W, B \rangle$ a *bi-structure*, where W are window functions and B is background data as in LARS. We call S_{L} the *left-stream* and S_{R} the *right-stream*. Moreover, $M_{\mathrm{L}} = \langle S_{\mathrm{L}}, W, B \rangle$ and $M_{\mathrm{R}} = \langle S_{\mathrm{R}}, W, B \rangle$ denote the underlying *LARS structures of* $\mathbf{M}$; the *left-structure* and the *right-structure*, respectively.

In this chapter, we confine to ground LARS, hence $\mathcal{A}$ denotes ground atoms and $\mathcal{F}$ ground formulas.

Definition 23 (bi-LARS Entailment) *Let* $\mathbf{M} = \langle \mathbf{S}, W, B \rangle$ *be as above and let* $t \in \mathbb{N}$. *The* bi-LARS-entailment *relation* $\Vdash$ *between* $(\mathbf{M}, t, \mathrm{w})$ *for worlds* $\mathrm{w} \in \{\mathrm{L}, \mathrm{R}\}$ *and formulas is defined as follows (where* $\alpha, \beta \in \mathcal{F}$ *are formulas):*

$$
\begin{array}{lcl}
\mathbf{M}, t, \mathrm{w} \Vdash a & :\Leftrightarrow & a \in \upsilon_{\mathrm{w}}(t) \text{ or } a \in B, \text{ for } a \in \mathcal{A}, \\
\mathbf{M}, t, \mathrm{w} \Vdash \alpha \wedge \beta & :\Leftrightarrow & \mathbf{M}, t, \mathrm{w} \Vdash \alpha \text{ and } \mathbf{M}, t, \mathrm{w} \Vdash \beta, \\
\mathbf{M}, t, \mathrm{w} \Vdash \Diamond \alpha & :\Leftrightarrow & \mathbf{M}, t', \mathrm{w} \Vdash \alpha \text{ for some } t' \in T, \\
\mathbf{M}, t, \mathrm{w} \Vdash \Box \alpha & :\Leftrightarrow & \mathbf{M}, t', \mathrm{w} \Vdash \alpha \text{ for all } t' \in T, \\
\mathbf{M}, t, \mathrm{w} \Vdash @_{t'} \alpha & :\Leftrightarrow & \mathbf{M}, t', \mathrm{w} \Vdash \alpha \text{ and } t' \in T, \\
\mathbf{M}, t, \mathrm{L} \Vdash \alpha \to \beta & :\Leftrightarrow & \mathbf{M}, t, \mathrm{L} \nVdash \alpha \text{ or } \mathbf{M}, t, \mathrm{L} \Vdash \beta, \text{ and } \mathbf{M}, t, \mathrm{R} \Vdash \alpha \to \beta \\
\mathbf{M}, t, \mathrm{R} \Vdash \alpha \to \beta & :\Leftrightarrow & \mathbf{M}, t, \mathrm{R} \nVdash \alpha \text{ or } \mathbf{M}, t, \mathrm{R} \Vdash \beta, \\
\text{For } \bullet \in \{\neg, \boxplus^w\}\text{:} & & \\
\mathbf{M}, t, \mathrm{L} \Vdash \bullet \alpha & :\Leftrightarrow & M_{\mathrm{L}}, t \Vdash \bullet \alpha \text{ and } M_{\mathrm{R}}, t \Vdash \bullet \alpha\,, \\
\mathbf{M}, t, \mathrm{R} \Vdash \bullet \alpha & :\Leftrightarrow & M_{\mathrm{R}}, t \Vdash \bullet \alpha.
\end{array}
$$

Moreover, we define $\mathbf{M}, t \Vdash \alpha \;:\Leftrightarrow\; \mathbf{M}, t, \mathrm{L} \Vdash \alpha$.

Similarly as for LARS, we define for the special atoms $\top$ and $\bot$ that $\mathbf{M}, t \Vdash \top/\bot$ always/never holds. If $\mathbf{M}, t \Vdash \alpha$ holds, we say that $\mathbf{M}$ *entails* α *at time* t and we then call $\mathbf{M}$ a *bi-model of* α *at time* t. Entailment and the notion of a model are extended to sets of formulas as usual.

We point out that the lack of a reset operator $\triangleright$ definition in bi-LARS entailment is for ease of presentation. Reset plays a minor role and introducing it leads to a more involved notation (requiring to additionally retain a current bi-stream $\mathbf{S}$ left of $\Vdash$). However, the results in the sequel carry over for a straightforward definition of $\triangleright$. We discuss this further in Section 5.7.

We draw here on the ideas from [Pea06], which introduced Equilibrium Logic as a characterization of the stable model semantics in form of a minimal model reasoning for the logic Here-and-There [Hey30]. The latter evaluates formulas in two worlds of a Kripke structure: everything that holds in the Here-world H (left) has to hold in the There-world T (right). Viewed as interpretations, i.e., sets of atoms, these worlds have the property $H \subseteq T$, and implication ($\to$) connects the two: In order for $\alpha \to \beta$ to be true Here, it also has to be true There, by definition. The same principle is reflected in the definition of connective $\to$ of bi-LARS entailment (Definition 23). Since Here-and-There is a monotone logic, and LARS in general is not, we need further adjustments. In contrast to LARS, Here-and-There defines negation as $\neg\alpha := \alpha \to \bot$. Using the negation of

LARS also for bi-LARS in general breaks the connection from left to right, as well as some window operators: when employing straightforward recursive definitions (as for $\wedge$, $\Diamond$, $\Box$, $@_{t'}$ and $\rightarrow$) for $\neg$ and $\boxplus^w$, an initial initial inclusion of the left-stream in the right-stream, as will be considered later, may not be retained. Hence, we branch into separate evaluation in the underlying LARS structures. We will examine later when a recursive definition is possible also for $\boxplus^w$.

Example 39 Let $S = ([0,3], \{0 \mapsto \{a\}, 1 \mapsto \{a\}, 3 \mapsto \{b,x\}\})$ be a stream and let $\varphi = \boxplus^2\Diamond a \wedge \neg y \rightarrow x$ be a LARS formula. Consider the structure $M = \langle S, \{\tau^2\}, \emptyset\rangle$. We have that $M,3 \Vdash \boxplus^2\Diamond a \wedge \neg y$. Indeed, since $y \notin \upsilon(3)$, it follows that $M,3 \Vdash \neg y$. Furthermore, $M,3 \Vdash \boxplus^2\Diamond a$ since $M',1 \Vdash a$, where M' is obtained by replacing S with $S' = ([1,3], \{1 \mapsto \{a\}, 3 \mapsto \{b,x\}\})$, i.e., the result of applying the time-based window of size 2 on S at time point 3. Furthermore, $M,3 \Vdash x$ as $x \in \upsilon(3)$; thus $M,3 \Vdash \varphi$ holds.

Next, let

$$S_{\mathrm{L}} = ([0,3], \{0 \mapsto \{a\}, 1 \mapsto \{a\}, 3 \mapsto \{b,x\}\}), \text{ and}$$
$$S_{\mathrm{R}} = ([0,3], \{0 \mapsto \{a\}, 1 \mapsto \{a\}, 3 \mapsto \{b,y\}\}).$$

Formula φ holds in both LARS-structures M_{L} and M_{R}, and in the bi-LARS-structure $\mathbf{M} = \langle \mathbf{S}, W, B\rangle$ at time 3: We observe that (i) $M_{\mathrm{L}},3 \Vdash \varphi$, since $M_{\mathrm{L}},3 \Vdash x$, from which also ($\star$) $\mathbf{M},3,\mathrm{L} \Vdash x$ follows. Further, we get (ii) $M_{\mathrm{R}},3 \Vdash \varphi$, since ($\star\star$) $M_{\mathrm{R}},3 \nVdash \neg y$ and thus $M_{\mathrm{R}},3 \nVdash \boxplus^2\Diamond a \wedge \neg y$. By definition, $\mathbf{M},3,\mathrm{R} \Vdash \boxplus^2\Diamond a \wedge \neg y \rightarrow x$ iff $\mathbf{M},3,\mathrm{R} \Vdash x$ or $\mathbf{M},3,\mathrm{R} \nVdash \boxplus^2\Diamond a \wedge \neg y$, and the latter iff $\mathbf{M},3,\mathrm{R} \nVdash \boxplus^2\Diamond a$ or $\mathbf{M},3,\mathrm{R} \nVdash \neg y$. We have the last condition by ($\star\star$), so we obtain ($\star\star\star$) $\mathbf{M},3,\mathrm{R} \Vdash \varphi$. Finally, we get (iii) $\mathbf{M},3 \Vdash \varphi$ by ($\star$) and ($\star\star\star$) which is defined as ($\mathbf{M},3,\mathrm{L} \nVdash \boxplus^2\Diamond a \wedge \neg y$ or $\mathbf{M},3,\mathrm{L} \Vdash x$) and $\mathbf{M},3,\mathrm{R} \Vdash \varphi$. ∎

We note that in general, entailment in both LARS structures does not imply entailment in the bi-structure.

Example 40 Consider the bi-stream $\mathbf{S} = (S_{\mathrm{L}}, S_{\mathrm{R}})$, where $S_{\mathrm{L}} = ([0,0], \{0 \mapsto \{a\}\})$ and $S_{\mathrm{R}} = ([0,0], \{0 \mapsto \emptyset\})$, and take $\alpha = a \rightarrow \beta$, where $\beta = (a \rightarrow a) \rightarrow a$. We have $M_{\mathrm{L}},0 \Vdash \alpha$ and $M_{\mathrm{R}},0 \Vdash \alpha$. For $\mathbf{M},0 \Vdash \alpha$, we need $\mathbf{M},0,\mathrm{L} \nVdash a$ or $\mathbf{M},0,\mathrm{L} \Vdash \beta$. However, $\mathbf{M},0,\mathrm{L} \Vdash a$, as $a \in \upsilon_{\mathrm{L}}(0)$ and $\mathbf{M},0,\mathrm{L} \nVdash \beta$ as $\mathbf{M},0,\mathrm{L} \Vdash (a \rightarrow a) \rightarrow a$ only if $\mathbf{M},0,\mathrm{R} \Vdash (a \rightarrow a) \rightarrow a$, but $\mathbf{M},0,\mathrm{R} \Vdash (a \rightarrow a)$ and $\mathbf{M},0,\mathrm{R} \nVdash a$. We thus have $M_{\mathrm{L}},0 \Vdash \alpha$ and $M_{\mathrm{R}},0 \Vdash \alpha$, but $\mathbf{M},0 \nVdash \alpha$. ∎

The following lemma, which is easily shown by induction on the structure of α, intuitively states that evaluation for the right-stream is independent of the left-stream.

Lemma 2 $\mathbf{M},t,\mathrm{R} \Vdash \alpha$ *iff* $M_{\mathrm{R}},t \Vdash \alpha$.

We call a bi-stream $(S_{\mathrm{L}}, S_{\mathrm{R}})$ *total*, if $S_{\mathrm{L}} = S_{\mathrm{R}}$. Restricting the study to total interpretations, bi-LARS-satisfaction collapses to LARS-satisfaction.

Proposition 3 *Let* $M = \langle S, W, B\rangle$ *be a structure, where* $S = (T, \upsilon)$ *and* $\mathbf{M} = \langle \mathbf{S}, W, B\rangle$, *with* $\mathbf{S} = (S, S)$, *and let* $t \in \mathbb{N}$ *and* α *be a formula. Then,* $\mathbf{M},t \Vdash \alpha$ *iff* $M,t \Vdash \alpha$.

Proof. The equivalence clearly holds if in every step in the evaluation of the recursive definitions, the left-structure M_{L} remains identical to the right-structure M_{R}. This may change only by the application of a window operator. However, in these cases, the evaluation branches into evaluation in the underlying LARS structures, and the equivalence follows. □

bi-LARS Semantics for LARS Programs

Entailment in bi-LARS is extended from formulas to programs analogously as for LARS. Recall the definition $\beta(r) = \beta_1 \wedge \ldots \wedge \beta_n$ for a rule r with body formulas $\beta_1, \ldots, \beta_n$.

Definition 24 (bi-LARS Program Semantics) *Let $D = (T, \upsilon)$ be a data stream, $t \in T$ and let P be a program. We say a* bi*-structure* $\mathbf{M}$ satisfies *a rule r of form $\alpha \leftarrow \beta_1, \ldots, \beta_n$ at t, denoted by*

$$\mathbf{M}, t \models r\,,$$

if $\mathbf{M}, t \Vdash \beta(r) \rightarrow \alpha$. In this case, $\mathbf{M}$ is a (bi-)model *of r (for D at t). Furthermore, we define*

$$\mathbf{M}, t \models P,$$

if $\mathbf{M}, t \models r$ for all $r \in P$. We then call $\mathbf{M}$ a (bi-)model *of P (for D at t).*

Example 41 (cont'd) Consider the bi-stream $\mathbf{S} = (S_{\mathrm{L}}, S_{\mathrm{R}})$ from Example 39 and the program P containing only the rule $r = x \leftarrow \boxplus^2 \Diamond a, \neg y$. We have $\beta(r) = \boxplus^2 \Diamond a \wedge \neg y$ and $H(r) = x$, i.e., $\beta(r) \rightarrow H(r)$ is formula φ from above. Since $\mathbf{M}, 3 \Vdash \boxplus^2 \Diamond a \wedge \neg y \rightarrow x$ holds, we have $\mathbf{M}, 3 \models r$ and $\mathbf{M}, 3 \models P$. ∎

We will consider the following fragment of LARS. For a rule r (of form $\alpha \leftarrow \beta_1, \ldots, \beta_n$) we define $\mathcal{F}(r) = \{\alpha, \beta_1, \ldots, \beta_n\}$, i.e., the set of its formulas, and for a program P naturally $\mathcal{F}(P) = \bigcup_{r \in P} \mathcal{F}(r)$.

Definition 25 ($\mathcal{F}_{\mathrm{bi}}$, $\mathrm{LARS}_{\mathrm{bi}}$) *By $\mathcal{F}_{\mathrm{bi}}$ we denote the class of LARS formulas without $\rightarrow$, where $\Diamond$ only occurs in the scope of $\neg$ or $\boxplus^w$. Moreover, $\mathrm{LARS}_{\mathrm{bi}}$ is the class of LARS programs P where all formulas in $\mathcal{F}(P)$ are in $\mathcal{F}_{\mathrm{bi}}$.*

Class $\mathcal{F}_{\mathrm{bi}}$ does not cover formulas of form $\Diamond\alpha$, which requires an evaluation of formula α over the entire history of the stream. However, existential quantification is typically of interest within the bounds of window. This corresponds to the form $\boxplus^w \Diamond \alpha$, which is a formula in $\mathcal{F}_{\mathrm{bi}}$ (for any window function w). Moreover, $\mathcal{F}_{\mathrm{bi}}$ permits arbitrary nesting of all connectives and operators, and in particular, window operators. Thus, it not only includes window atoms (from plain LARS), i.e., formulas of the form $\boxplus^w \circ \alpha$, where $\circ \in \{@_t, \Diamond, \Box\}$ and α is an atom, but also their immediate generalization $\boxplus^{w_1} \cdots \boxplus^{w_n} \circ \alpha$, where α is any LARS formula; moreover $\neg$ can be inserted at any position. Using formulas from $\mathcal{F}_{\mathrm{bi}}$ as building blocks yields a class of LARS programs. In particular, our guiding fragment plain LARS is subsumed by $\mathrm{LARS}_{\mathrm{bi}}$.

Unless stated otherwise, programs are now tacitly assumed to be in $\mathrm{LARS}_{\mathrm{bi}}$.

5.3 Characterizing Answer Streams

We now study properties which arise when considering bi-structures $\mathbf{M} = \langle \mathbf{S}, W, B \rangle$, where the left-stream $S_{\mathrm{L}} = (T, \upsilon_{\mathrm{L}})$ is a substream of the right-stream $S_{\mathrm{R}} = (T, \upsilon_{\mathrm{R}})$, i.e., given $\mathbf{S} = (S_{\mathrm{L}}, S_{\mathrm{R}})$, the condition

$$S_{\mathrm{L}} \subseteq S_{\mathrm{R}} \,.$$

The first observation is that under this assumption bi-entailment implies LARS entailment for the right-structure.

Lemma 3 (Persistence) *Let* $\mathbf{M} = \langle \mathbf{S}, W, B \rangle$ *s.t.* $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ *and let* $\alpha \in \mathcal{F}_{\mathrm{bi}}$. *If* $\mathbf{M}, t \Vdash \alpha$, *then* $M_{\mathrm{R}}, t \Vdash \alpha$.

Proof. Assume $\mathbf{M}, t \Vdash \alpha$ holds, i.e., $\mathbf{M}, t, \mathrm{L} \Vdash \alpha$. We show that $M_{\mathrm{R}}, t \Vdash \alpha$ holds by a case distinction of the structure of α. For atoms a, we have $a \in B$ or $a \in \upsilon_{\mathrm{L}}(t)$. This is clear if $a \in B$, as B is also the background data of M_{R}. If $a \in \upsilon_{\mathrm{L}}(t)$, then $a \in \upsilon_{\mathrm{R}}(t)$, as $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$. Thus, $M_{\mathrm{R}}, t \Vdash a$. The cases for $\neg$, $\boxplus^w$ and $\rightarrow$ follow by definition and Lemma 2. Consider α of form $\varphi \wedge \psi$. We have by definition $\mathbf{M}, t \Vdash \varphi$ and $\mathbf{M}, t \Vdash \psi$. By induction follows $M_{\mathrm{R}}, t \Vdash \varphi$ and $M_{\mathrm{R}}, t \Vdash \psi$ and thus $M_{\mathrm{R}}, t \Vdash \varphi \wedge \psi$. Next, suppose α is of form $\Box\varphi$, i.e., $\mathbf{M}, t', \mathrm{L} \Vdash \varphi$ for all $t' \in T$. By induction, we get $M_{\mathrm{R}}, t' \Vdash \alpha$ for all $t' \in T$ and thus $M_{\mathrm{R}}, t \Vdash \Box\varphi$. For α of form $@_{t'}\varphi$ we observe $\mathbf{M}, t', \mathrm{L} \Vdash \varphi$, and thus again $M_{\mathrm{R}}, t \Vdash \varphi$ inductively. Finally, consider α of form $\Diamond\varphi$. By definition, there exists a time point $t' \in T$ s.t. $\mathbf{M}, t', \mathrm{L} \Vdash \varphi$. By induction, $M_{\mathrm{R}}, t' \Vdash \varphi$, and thus $M_{\mathrm{R}}, t \Vdash \Diamond\varphi$. □

The analogous case for the left-structure does not hold, as the following example shows.

Example 42 To see that $\mathbf{M}, t \Vdash \alpha$ and $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ does not imply $M_{\mathrm{L}}, t \Vdash \alpha$, consider the streams $S_{\mathrm{L}} = ([0,0], \{0 \mapsto \emptyset\})$ and $S_{\mathrm{R}} = ([0,0], \{0 \mapsto \{a\}\})$. Let $\alpha = \neg a \rightarrow b$. We have

(i) $M_{\mathrm{L}}, 0 \Vdash \neg a$ and $M_{\mathrm{L}}, 0 \nVdash b$, thus
(ii) $M_{\mathrm{L}}, 0 \nVdash \neg a \rightarrow b$;
(iii) $M_{\mathrm{R}}, 0 \nVdash \neg a$, thus
(iv) $M_{\mathrm{R}}, 0 \Vdash \neg a \rightarrow b$; however
(v) $\mathbf{M}, 0 \nVdash \neg a$ by (iii) and
(vi) $\mathbf{M}, 0 \Vdash \neg a \rightarrow b$ by (iv) and (v). ∎

Let $\mathcal{F}_{\not\rightarrow}$ be the class of LARS formulas without implication. For such formulas, entailment in the bi-structure carries over to the left-structure as well.

Lemma 4 *Let* $\mathbf{M} = \langle \mathbf{S}, W, B \rangle$ *s.t.* $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ *let* $\alpha \in \mathcal{F}_{\not\rightarrow}$. *If* $\mathbf{M}, t \Vdash \alpha$, *then* $M_{\mathrm{L}}, t \Vdash \alpha$.

The lemma is shown by induction on the structure of α.

Proof. Assume $\alpha \in \mathcal{F}_{\not\to}$ and $\mathbf{M}, t \Vdash \alpha$ holds, i.e., $\mathbf{M}, t, \mathrm{L} \Vdash \alpha$. We show that $M_\mathrm{L}, t \Vdash \alpha$ holds by a case distinction of the structure of α. The case is clear for atoms a, where have $a \in B$ (and $M_\mathrm{L} = \langle S_\mathrm{L}, W, B\rangle$, where $S_\mathrm{L} = (T, \upsilon_\mathrm{L})$) or $a \in \upsilon_\mathrm{L}(t)$. The cases for $\neg$ and $\boxplus^w$ follow by definition. Consider α of form $\varphi \wedge \psi$. We have by definition $\mathbf{M}, t \Vdash \varphi$ and $\mathbf{M}, t \Vdash \psi$. By induction follows $M_\mathrm{L}, t \Vdash \varphi$ and $M_\mathrm{L}, t \Vdash \psi$ and thus $M_\mathrm{L}, t \Vdash \varphi \wedge \psi$. Next, suppose α is of form $\Box\varphi$, i.e., $\mathbf{M}, t', \mathrm{L} \Vdash \varphi$ for all $t' \in T$. By induction, we get $M_\mathrm{L}, t' \Vdash \alpha$ for all $t' \in T$ and thus $M_\mathrm{L}, t \Vdash \Box\varphi$. For α of form $@_{t'}\varphi$, we observe $\mathbf{M}, t', \mathrm{L} \Vdash \varphi$, and thus again $M_\mathrm{L}, t \Vdash \varphi$ inductively. Finally, consider α of form $\Diamond\varphi$. By definition, there exists a time point $t' \in T$ s.t. $\mathbf{M}, t', \mathrm{L} \Vdash \varphi$. By induction, $M_\mathrm{L}, t' \Vdash \varphi$, and thus $M_\mathrm{L}, t \Vdash \Diamond\varphi$. □

Lemmas 3 and 4 together yield that $\mathbf{M}, t \Vdash \alpha$ implies $M_\mathrm{L}, t \Vdash \alpha$ and $M_\mathrm{R}, t \Vdash \alpha$, given that $S_\mathrm{L} \subseteq S_\mathrm{R}$ and $\alpha \in \mathcal{F}_{\not\to}$. The next example shows that the opposite direction needs further restrictions.

Example 43 Assume at evaluation time point t, the current stream contains the set A of atoms. We consider a window function w_ε (and a corresponding window operator $\boxplus^\varepsilon$) that returns the substream $([t,t], t \mapsto \emptyset)$, if $\varepsilon \in A$, and $([t,t], t \mapsto A)$ else. Let $S_\mathrm{L} = ([0,1], \upsilon_\mathrm{L})$ and $S_\mathrm{R} = ([0,1], \upsilon_\mathrm{R})$ be two streams with respective LARS structures $M_\mathrm{J} \in \{M_\mathrm{L}, M_\mathrm{R}\}$, as follows, where $M'_\mathrm{J} \in \{M'_\mathrm{L}, M'_\mathrm{R}\}$ denote the resulting structures after applying the window was applied at time points 0 and 1, respectively:

M_J	$\upsilon_\mathrm{J}(0)$	$\upsilon_\mathrm{J}(1)$	M'_J	$\upsilon'_\mathrm{J}(0)$	$\upsilon'_\mathrm{J}(1)$
M_L	$\{a\}$	$\emptyset$	M'_L	$\{a\}$	$\emptyset$
M_R	$\{a, \varepsilon\}$	$\{a\}$	M'_R	$\emptyset$	$\{a\}$

Consider the LARS formula $\varphi = \Diamond\boxplus^\varepsilon\Diamond a$ and the bi-structure $\mathbf{M} = \langle (S_\mathrm{L}, S_\mathrm{R}), \{w_\varepsilon\}, \emptyset\rangle$. We observe:

(i) $M_\mathrm{L}, 1 \Vdash \varphi$, as there exists a time point $t' = 0$ s.t. $M_\mathrm{L}, t' \Vdash \boxplus^\varepsilon\Diamond a$: after the window operator, we evaluate in $M'_\mathrm{L} = \langle S'_\mathrm{L}, \{w_\varepsilon\}, \emptyset\rangle$, where $S'_\mathrm{L} = ([0,0], 0 \mapsto \{a\})$, and have $M'_\mathrm{L}, 0 \Vdash \Diamond a$ because $M'_\mathrm{L}, 0 \Vdash a$.

(ii) $M_\mathrm{R}, 1 \Vdash \varphi$: Similarly for $t' = 1$ we have $M_\mathrm{R}, 1 \Vdash \boxplus^\varepsilon\Diamond a$, since $M'_\mathrm{R}, 1 \Vdash \Diamond a$.

(iii) However, $\mathbf{M}, 1 \nVdash \varphi$, as intuitively, the first $\Diamond$ breaks the connection between left and right. There is no time point t' such that $\mathbf{M}, t' \Vdash \boxplus^\varepsilon\Diamond a$: the formula evaluates positively on the left only at $t' = 0$, and on the right only at $t' = 1$.

Note that the same problem occurs for $\varphi' = \Diamond\boxplus^\varepsilon a$. ∎

For formula class $\mathcal{F}_\mathrm{bi}$ we can show the desired relationship, i.e., that bi-LARS entailment follows from entailment in both LARS structures.

Lemma 5 *Let* $\mathbf{M} = \langle \mathbf{S}, W, B\rangle$ *s.t.* $S_\mathrm{L} \subseteq S_\mathrm{R}$ *and let* $\alpha \in \mathcal{F}_\mathrm{bi}$. *If* $M_\mathrm{L}, t \Vdash \alpha$ *and* $M_\mathrm{R}, t \Vdash \alpha$, *then* $\mathbf{M}, t \Vdash \alpha$.

Proof. We again prove this by a case distinction on the structure of α. In case α is an atom, we get the result by definition, and similarly for formulas of form $\neg\varphi$ and $\boxplus\varphi$. Consider α of form $\varphi \wedge \psi$. For both $M_{\mathrm{J}} \in \{M_{\mathrm{L}}, M_{\mathrm{R}}\}$ and $\gamma \in \{\varphi, \psi\}$, we have $M_{\mathrm{J}}, t \Vdash \gamma$. We get $\mathbf{M}, t \Vdash \varphi$ inductively from $M_{\mathrm{L}}, t \Vdash \varphi$ and $M_{\mathrm{R}}, t \Vdash \varphi$, and likewise $\mathbf{M}, t \Vdash \psi$. Consequently, $\mathbf{M}, t \Vdash \varphi \wedge \psi$. Next, consider α of form $\Box\varphi$. We have for both $M_{\mathrm{J}} \in \{M_{\mathrm{L}}, M_{\mathrm{R}}\}$, $M_{\mathrm{J}}, t' \Vdash \varphi$ for all $t' \in T$. That is, for each $t' \in T$, we have $\mathbf{M}, t' \Vdash \varphi$ by induction, and thus $\mathbf{M}, t \Vdash \Box\varphi$ by definition. For α of form $@_{t'}\varphi$ we similarly observe that having $M_{\mathrm{J}}, t' \Vdash \alpha$ for both $M_{\mathrm{J}} \in \{M_{\mathrm{L}}, M_{\mathrm{R}}\}$ inductively gives $\mathbf{M}, t' \Vdash \varphi$ and hence $\mathbf{M}, t \Vdash @'_t\varphi$. Finally, since $\alpha \in \mathcal{F}_{\mathrm{bi}}$ it cannot be of form $\Diamond\varphi$, but for the inductive argument, consider a subformula α' of this form. By assumption, it appears in the scope of some $\bullet \in \{\neg, \boxplus\}$. We thus consider the subformula $\bullet\,\alpha'$, which is evaluated in both the left-structure and the right-structure. Consequently, if for both $M_{\mathrm{J}} \in \{M_{\mathrm{L}}, M_{\mathrm{R}}\}$ we have $M_{\mathrm{J}}, t \Vdash \bullet\,\alpha'$, we get $\mathbf{M}, t \Vdash \bullet\,\alpha'$ by definition. □

Remark. As illustrated in Example 43, restricting the use of $\Diamond$ is crucial, for which the inductive argument (outside the scope of $\neg$ or $\boxplus^w$) does not work: suppose we have for both $M_{\mathrm{J}} \in \{M_{\mathrm{L}}, M_{\mathrm{R}}\}$ that $M_{\mathrm{J}}, t \Vdash \Diamond\varphi$. This implies that $M_{\mathrm{L}}, t_{\mathrm{L}} \Vdash \varphi$ and $M_{\mathrm{R}}, t_{\mathrm{R}} \Vdash \varphi$ for some time points t_{L} and t_{R}, but not that $t_{\mathrm{L}} = t_{\mathrm{R}}$. Consequently, the induction step to $\mathbf{M}, t' \Vdash \varphi$ does not work, because in general, a common time point t' where φ holds does not exist. The problem does not occur for $\Box$, which ensures entailment at *all* time points, and $@_{t'}$, since it directly specifies the same time point for both LARS structures.

Note that the FLP-semantics of answer streams (Definition 18) is defined non-recursively. Still, we can capture it by branching in bi-LARS evaluation of $\neg$ and $\boxplus^w$ into separate LARS evaluations for left and right, due to the following central property, which follows from Lemmas 3-5 (recall that $\mathcal{F}_{\mathrm{bi}} \subseteq \mathcal{F}_{\not\Rightarrow}$).

Proposition 4 *Let $\mathbf{M} = \langle \mathbf{S}, W, B\rangle$ be a* bi*-structure such that $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ and let $\alpha \in \mathcal{F}_{\mathrm{bi}}$. Then, $\mathbf{M}, t \Vdash \alpha$ iff $M_{\mathrm{L}}, t \Vdash \alpha$ and $M_{\mathrm{R}}, t \Vdash \alpha$.*

The stipulated relation $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ naturally arises with minimality checking of models, where intuitively S_{R} is a model M of a program P at time t and S_{L} is a candidate model of the reduct $P^{M,t}$. It establishes a semantic connection between left and right, which can be exploited to conclude that $\mathbf{M}, t \Vdash \alpha$ implies $S_{\mathrm{L}}, t \Vdash \alpha$.

The following result now captures the essence of the reduct-based semantics: the left-structure must satisfy each rule whose body is true in the model given by the right-structure. The proof is based on [LPV01].

Theorem 19 *For any $\mathbf{M} = \langle \mathbf{S}, W, B\rangle$ such that $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$, time t and program P, we have $\mathbf{M}, t \models P$ iff $M_{\mathrm{R}}, t \models P$ and $M_{\mathrm{L}}, t \models P^{M_{\mathrm{R}},t}$.*

Proof. ($\Rightarrow$) Let $\mathbf{M}, t \models P$, i.e., $\mathbf{M}, t \models r$ for all rules $r \in P$. By Definition 23, we have that, if $\mathbf{M}, t \models r$, then $\mathbf{M}, t, \mathrm{R} \Vdash \beta(r) \rightarrow \alpha$, where α is the head of r. By Lemma 2, we get $M_{\mathrm{R}}, t \Vdash r$, and thus $M_{\mathrm{R}}, t \models P$.

It remains to show $M_{\mathrm{L}}, t \models P^{M_{\mathrm{R}},t}$. Towards a contradiction, assume this is not the case, i.e., $M_{\mathrm{L}}, t \nVdash \beta(r) \rightarrow \alpha$ for some $r \in P^{M_{\mathrm{R}},t}$, i.e., some $r \in P$ where $M_{\mathrm{R}}, t \Vdash \beta(r)$. We

have $M_{\mathrm{L}}, t \Vdash \beta(r)$ and $M_{\mathrm{L}}, t \nVdash \alpha$. Since $\alpha \in \mathcal{F}_{\not\to}$ ($\mathcal{F}_{\mathrm{bi}} \subseteq \mathcal{F}_{\not\to}$), we get by Lemma 4 that $\mathbf{M}, t \nVdash \alpha$ holds. Since for both $M_{\mathrm{J}} \in \{M_{\mathrm{L}}, M_{\mathrm{R}}\}$ we have $M_{\mathrm{J}}, t \Vdash \beta_i$ for all body formulas β_i we get by Lemma 5 that $\mathbf{M}, t \Vdash \beta_i$ (for all β_i). Hence $\mathbf{M}, t \Vdash \beta(r)$ and $\mathbf{M}, t \nVdash \alpha$, i.e., $\mathbf{M}, t \nVdash \beta(r) \to \alpha$, which contradicts that $\mathbf{M}$ entails all rules in P at t.

($\Leftarrow$) Indirectly, suppose $\mathbf{M}, t \not\models P$, i.e., that there exists some rule $r \in P$ such that $\mathbf{M}, t \nVdash \beta(r) \to \alpha$. That is, either

(i) $\mathbf{M}, t, \mathrm{L} \Vdash \beta(r)$ and $\mathbf{M}, t, \mathrm{L} \nVdash \alpha$, or

(ii) $\mathbf{M}, t, \mathrm{R} \nVdash \beta(r) \to \alpha$.

Case (ii) is by definition equal to $\mathbf{M}, t, \mathrm{R} \Vdash \beta(r)$ and $\mathbf{M}, t, \mathrm{R} \nVdash \alpha$, which gives $M_{\mathrm{R}}, t \Vdash \beta(r)$ and $M_{\mathrm{R}}, t \nVdash \alpha$ by Lemma 2 and thus $M_{\mathrm{R}}, t \nVdash \beta(r) \to \alpha$. Hence, we get $M_{\mathrm{R}}, t \nVdash P$.

Thus, we assume $\mathbf{M}, t, \mathrm{R} \Vdash \beta(r) \to \alpha$ (resp. $M_{\mathrm{R}}, t \Vdash r$) and analyze Case (i), where we get from $\mathbf{M}, t \Vdash \beta(r)$ (a) by Lemma 3 that $M_{\mathrm{R}}, t \Vdash \beta(r)$, and (b) by Lemma 4 that $M_{\mathrm{L}}, t \Vdash \beta(r)$. From the assumption and Lemma 2 follows $M_{\mathrm{R}}, t \Vdash \beta(r) \to \alpha$, which, together with (a) yields $M_{\mathrm{R}}, t \Vdash \alpha$. This, given $\mathbf{M}, t \nVdash \alpha$ (by the assumption in Case (i)) allows us to conclude by Lemma 5 that $M_{\mathrm{L}}, t \nVdash \alpha$, and hence $M_{\mathrm{L}}, t \nVdash \beta(r) \to \alpha$. It also follows from (a) that $r \in P^{M_{\mathrm{R}},t}$, and since $M_{\mathrm{L}}, t \nVdash r$, we conclude $M_{\mathrm{L}}, t \not\models P^{M_{\mathrm{R}},t}$. □

We are now going to characterize answer streams similarly as in [LPV01] and [Tur01], by capturing the minimality condition in terms of bi-equilibrium models. Intuitively, a bi-equilibrium model is a bi-model with a total bi-stream $(S_{\mathrm{R}}, S_{\mathrm{R}})$ such that no smaller bi-stream $(S_{\mathrm{L}}, S_{\mathrm{R}})$ where $S_{\mathrm{L}} \subset S_{\mathrm{R}}$ is a bi-model.

Definition 26 (bi-Equilibrium Model) *Let $M = \langle I, W, B\rangle$ be a structure. We say $\mathbf{M} = \langle (I, I), W, B\rangle$ is a* bi-equilibrium model *of a program P for data stream D at time t, if*

(i) $\mathbf{M}, t \models P$, *and*

(ii) $\mathbf{M}', t \not\models P$, *for each* $\mathbf{M}' = \langle (I', I), W, B\rangle$ *such that* $D \subseteq I' \subset I$ *and* $I' = (T, \upsilon')$.

We obtain the next theorem from Definition 18, Proposition 3 and Theorem 19.

Theorem 20 *Let $M = \langle I, W, B\rangle$ be a structure such that I is an interpretation stream for D at t. Then, $I \in \mathcal{AS}(P, D, t)$ iff $\mathbf{M} = \langle (I, I), W, B\rangle$ is a* bi-*equilibrium model.*

Proof. ($\Rightarrow$) Let $I \in \mathcal{AS}(P, D, t)$, i.e., $M = \langle I, W, B\rangle$ is a minimal model of $P^{M,t}$ for D at t. Since, $M, t \models P$, we have $\mathbf{M}, t \models P$ by Proposition 3. For Condition (ii), assume towards a contradiction, that some interpretation stream $I' = (T, \upsilon') \subset I$ exists s.t. $\mathbf{M}', t \models P$, where $\mathbf{M}' = \langle (I', I), W, B\rangle$. By Theorem 19, we then have $M', t \models P^{M,t}$, where $M' = \langle I', W, B\rangle$ (i.e., the left-structure). This means M' is a smaller model for the reduct $P^{M,t}$ than M, i.e., I cannot be an answer stream.

($\Leftarrow$) Indirectly, suppose $I \notin \mathcal{AS}(P, D, t)$ and let $M = \langle I, W, B\rangle$. If $M, t \not\models P$, then by Proposition 3, $\mathbf{M}, t \not\models P$, violating Condition (i) for $\mathbf{M}$ to be a bi-equilibrium model. If $M, t \models P$, as $I \notin \mathcal{AS}(P, D, t)$, there must exist a proper substream $I' \subset I$ such that

$M' = \langle I', W, B\rangle$ is a smaller model of $P^{M,t}$, i.e., $M', t \models P^{M,t}$. Then, by Theorem 19, $\mathbf{M}', t \models P$, violating Condition (ii). □

As explained above, the obtained results do not generalize to arbitrary LARS programs. Example 43 illustrated why we consider $\Diamond$ only in the scope of $\neg$ or $\boxplus^w$, and why we branch into LARS evaluation for these operators. This allows us to characterize program equivalences which include non-trivial window operators.

5.4 Characterizing Equivalence Notions

We now characterize equivalence notions by means of bi-models. To this end, from now we tacitly restrict to bi-structures $\mathbf{M} = \langle \mathbf{S}, W, B\rangle$ such that $S_\mathrm{L} \subseteq S_\mathrm{R}$. Unless stated otherwise, programs are from class $\mathrm{LARS}_\mathrm{bi}$. Moreover, we let $\mathbf{M}_\mathrm{LR} = \langle (S_\mathrm{L}, S_\mathrm{R}), W, B\rangle$ and $\mathbf{M}_\mathrm{RR} = \langle (S_\mathrm{R}, S_\mathrm{R}), W, B\rangle$. By $\mathrm{bi}(P)$, we denote the set of all respective bi-models of a program P (where data stream D and time point t are implicit).

The following lemma is immediate from the definition of bi-models.

Lemma 6 *Let P and X be two programs. Then,* $\mathrm{bi}(P \cup X) = \mathrm{bi}(P) \cap \mathrm{bi}(X)$.

Proof. First, we observe that for a structure M and a time point t, ($M, t \models P$ and $M, t \models X$) iff $M, t \models P \cup X$. Furthermore, $P^{M,t} \cup X^{M,t} = (P \cup X)^{M,t}$. Now, consider a bi-structure $\mathbf{M}$ with underlying LARS structures M_L and M_R as usual. By Theorem 19, $\mathbf{M}, t \models P \cup X$ iff ($M_\mathrm{R}, t \models P \cup X$ and $M_\mathrm{L}, t \models (P \cup X)^{M_\mathrm{R},t}$) which in turn holds iff (($M_\mathrm{R}, t \models P$ and $M_\mathrm{R}, t \models X$) and ($M_\mathrm{L}, t \models P^{M_\mathrm{R},t}$ and $M_\mathrm{L}, t \models X^{M_\mathrm{R},t}$)) iff $\mathbf{M}, t \models P$ and $\mathbf{M}, t \models X$. □

The first equivalence characterization result states that two programs are strongly equivalent iff they possess the same bi-models.

Theorem 21 (Strong Equivalence) *Let $D = (T, \upsilon)$ be a data stream, $t \in T$, and let P and Q be $\mathrm{LARS}_\mathrm{bi}$ programs. Then, $P \equiv_s Q$ (for D at t) iff* $\mathrm{bi}(P) = \mathrm{bi}(Q)$ *(for D at t).*

For the following proofs we define, given a stream $S = (T, \upsilon)$,

$$\Gamma(S) = \{@_t a \mid a \in \upsilon(t) \cap \mathcal{A}^\mathcal{I}, t \in T\}, \tag{5.5}$$

i.e., a translation of the intensional part into a set of @-atoms. We also view $\Gamma(S)$ as conjunction. Observe that $M, t \models \Gamma(S)$ for all $t \in T$, where $M = \langle S, W, B\rangle$.

Proof. ($\Rightarrow$) Indirectly, assume $\mathrm{bi}(P) \neq \mathrm{bi}(Q)$, and w.l.o.g., let $\mathbf{M}_\mathrm{LR} = \langle (S_\mathrm{L}, S_\mathrm{R}), W, B\rangle$ be a bi-structure s.t. $\mathbf{M}_\mathrm{LR}, t \models P$ and $\mathbf{M}_\mathrm{LR}, t \not\models Q$. To show that P and Q are not strongly equivalent, we construct a program X such that $P \cup X$ and $Q \cup X$ do not have the same bi-equilibrium models, i.e., by Theorem 20, that they do not have the same answer streams. Let $\mathbf{M}_\mathrm{RR} = \langle (S_\mathrm{R}, S_\mathrm{R}), W, B\rangle$. We distinguish two cases:

Case $\mathbf{M}_\mathrm{RR}, t \not\models Q$: by $\mathbf{M}_\mathrm{LR}, t \models P$ and Lemma 3, we get $M_\mathrm{R}, t \models P$, and by Proposition 3, $\mathbf{M}_\mathrm{RR}, t \models P$. Now, let $X = \Gamma(S_\mathrm{R})$. Since $\mathbf{M}_\mathrm{RR}, t \models X$, we have $\mathbf{M}_\mathrm{RR}, t \models P \cup X$

and furthermore, by construction of X, $\mathbf{M}_{\mathrm{RR}}$ is a bi-equilibrium model of $P \cup X$. On the other hand, $\mathbf{M}_{\mathrm{RR}}, t \not\models Q$ and therefore $\mathbf{M}_{\mathrm{RR}}, t \not\models Q \cup X$.

Case $\mathbf{M}_{\mathrm{RR}}, t \models Q$: let

$$X = \Gamma(S_{\mathrm{L}}) \cup \{@_z b \leftarrow @_y a \mid a \in \upsilon_{\mathrm{R}}(y) \setminus \upsilon_{\mathrm{L}}(y),\, b \in \upsilon_{\mathrm{R}}(z) \setminus \upsilon_{\mathrm{L}}(z),\, y, z \in T\}.$$

We have $\mathbf{M}_{\mathrm{RR}}, t \models X$ and therefore $\mathbf{M}_{\mathrm{RR}}, t \models Q \cup X$. We now show that $\mathbf{M}_{\mathrm{RR}}$ is a bi-equilibrium model of (i) $Q \cup X$, but (ii) not of $P \cup X$.

(i) Consider any bi-model $\mathbf{M} = \langle (S, S_{\mathrm{R}}), W, B \rangle$ of $Q \cup X$, where $S = (T, \upsilon) \subset S_{\mathrm{R}}$. Since $\Gamma(S_{\mathrm{L}}) \subseteq X$, S_{L} must be a substream of S. Furthermore, we have $S \neq S_{\mathrm{L}}$, since $\mathbf{M}_{\mathrm{LR}}, t \not\models Q$. Thus, $S_{\mathrm{L}} \subset S \subset S_{\mathrm{R}}$. Now, let $a \in \upsilon(y) \setminus \upsilon_{\mathrm{L}}(y)$ and $b \in \upsilon_{\mathrm{R}}(z) \setminus \upsilon(z)$ for some $y, z \in T$. Clearly, $@_z b \leftarrow @_y a$ belongs to X, but $\mathbf{M}, t \not\models @_z b \leftarrow @_y a$. Hence, $\mathbf{M}, t \not\models Q \cup X$.

(ii) Consider the initial assumption $\mathbf{M}_{\mathrm{LR}}, t \models P$. Clearly, $\mathbf{M}_{\mathrm{LR}}, t \models \Gamma(S_{\mathrm{L}})$. Moreover, for every rule $r = @_z b \leftarrow @_y a \in X$, $\mathbf{M}_{\mathrm{LR}}, t \models r$. Hence, $\mathbf{M}_{\mathrm{LR}}, t \models P \cup X$ and $\mathbf{M}_{\mathrm{RR}}$ is not a bi-equilibrium model of $P \cup X$.

($\Leftarrow$) Assume $\mathrm{bi}(P) = \mathrm{bi}(Q)$. We then have for all programs X that $\mathrm{bi}(P) \cap \mathrm{bi}(X) = \mathrm{bi}(Q) \cap \mathrm{bi}(X)$ and hence by Lemma 6 that $\mathrm{bi}(P \cup X) = \mathrm{bi}(Q \cup X)$. Consequently, $P \cup X$ and $Q \cup X$ have the same answer sets, i.e., P and Q are strongly equivalent. □

Furthermore, we also characterize uniform equivalence in terms of bi-entailment.

Theorem 22 (Uniform Equivalence) *Let $D = (T, \upsilon)$ be a data stream, $t \in T$, and let P and Q be LARS$_{\mathrm{bi}}$ programs. Then, $P \equiv_u Q$ iff*

(i) for each $\mathbf{M}_{\mathrm{RR}}$, $\mathbf{M}_{\mathrm{RR}} \in \mathrm{bi}(P)$ iff $\mathbf{M}_{\mathrm{RR}} \in \mathrm{bi}(Q)$, and

(ii) for each $\mathbf{M}_{\mathrm{LR}}$, where $S_{\mathrm{L}} \subset S_{\mathrm{R}}$, $\mathbf{M}_{\mathrm{LR}} \in \mathrm{bi}(P)$ (resp. $\mathbf{M}_{\mathrm{LR}} \in \mathrm{bi}(Q)$) iff $\mathbf{M} \in \mathrm{bi}(Q)$ (resp. $\mathbf{M} \in \mathrm{bi}(P)$) for some $\mathbf{M} = \langle (S, S_{\mathrm{R}}), W, B \rangle$ s.t. $S_{\mathrm{L}} \subseteq S \subset S_{\mathrm{R}}$.

Proof. ($\Rightarrow$) Let $P \equiv_u Q$. Towards a contradiction, assume (i) does not hold and assume w.l.o.g. that $\mathbf{M}_{\mathrm{RR}}, t \models P$ but $\mathbf{M}_{\mathrm{RR}}, t \not\models Q$. Then, by construction of $\Gamma(S_{\mathrm{R}})$, $\mathbf{M}_{\mathrm{RR}}, t \models P \cup \Gamma(S_{\mathrm{R}})$ and there exists no $S_{\mathrm{L}} \subset S_{\mathrm{R}}$ such that $\mathbf{M}_{\mathrm{LR}}, t \models P \cup \Gamma(S_{\mathrm{R}})$, i.e., $\mathbf{M}_{\mathrm{RR}}$ is a bi-equilibrium model of $P \cup \Gamma(S_{\mathrm{R}})$. On the other hand, $\mathbf{M}_{\mathrm{RR}}, t \not\models Q \cup \Gamma(S_{\mathrm{R}})$, since $\mathbf{M}_{\mathrm{RR}}, t \not\models Q$. Hence, $\mathbf{M}_{\mathrm{RR}}$ is not a bi-equilibrium model of $Q \cup \Gamma(S_{\mathrm{R}})$. In summary, we obtain that the answer streams for $P \cup \Gamma(S_{\mathrm{R}})$ and $Q \cup \Gamma(S_{\mathrm{R}})$ do not coincide, contradicting the fact that $P \equiv_u Q$.

To show (ii), again suppose the contrary towards a contradiction. W.l.o.g., assume $\mathbf{M}_{\mathrm{LR}}, t \models P$ and $\mathbf{M}, t \not\models Q$ for all $\mathbf{M} = \langle (S, S_{\mathrm{R}}), W, B \rangle$ such that $S_{\mathrm{L}} \subseteq S \subset S_{\mathrm{R}}$. Since $\mathbf{M}_{\mathrm{LR}}, t \models P$, also $\mathbf{M}_{\mathrm{RR}}, t \models P$ and therefore, by (i), $\mathbf{M}_{\mathrm{RR}}, t \models Q$. Furthermore, by construction of $\Gamma(S_{\mathrm{L}})$, $\mathbf{M}_{\mathrm{RR}}, t \models Q \cup \Gamma(S_{\mathrm{L}})$. From $\mathbf{M}, t \not\models Q$ we get $\mathbf{M}, t \not\models Q \cup \Gamma(S_{\mathrm{L}})$ (for all considered $\mathbf{M}$ with $S \subset S_{\mathrm{R}}$). Consequently, $\mathbf{M}_{\mathrm{RR}}$ is a bi-equilibrium model of $Q \cup \Gamma(S_{\mathrm{L}})$. Furthermore, we have $\mathbf{M}_{\mathrm{LR}}, t \models P \cup \Gamma(S_{\mathrm{L}})$ and thus, $\mathbf{M}_{\mathrm{RR}}$ cannot be a bi-equilibrium model of $P \cup \Gamma(S_{\mathrm{L}})$. In conclusion, $\mathbf{M}_{\mathrm{RR}}$ is a bi-equilibrium model of $Q \cup \Gamma(S_{\mathrm{L}})$ but not of $P \cup \Gamma(S_{\mathrm{L}})$, i.e., they have different answer streams, contradicting the initial assumption that $P \equiv_u Q$.

($\Leftarrow$) Assume that (i) and (ii) hold. Towards a contradiction, suppose P and Q are not uniformly equivalent. Then, some set X consisting of @-atoms exists such that, w.l.o.g. $P \cup X$ has some bi-equilibrium model $\mathbf{M}_{\mathrm{RR}}$ which is not a bi-equilibrium model of $Q \cup X$. Since $\mathbf{M}_{\mathrm{RR}}, t \models P$ and $\mathbf{M}_{\mathrm{RR}}, t \models X$, and by (i) $\mathbf{M}_{\mathrm{RR}}, t \models Q$, we have also $\mathbf{M}_{\mathrm{RR}}, t \models Q \cup X$. Thus, some $S_{\mathrm{L}} \subset S_{\mathrm{R}}$ exists s.t. $\mathbf{M}_{\mathrm{LR}}, t \models Q \cup X$, which implies $\mathbf{M}_{\mathrm{LR}}, t \models Q$. Since (ii) holds, some $\mathbf{M} = \langle (S, S_{\mathrm{R}}), W, B \rangle$ exists, where $S_{\mathrm{L}} \subseteq S \subset S_{\mathrm{R}}$, s.t. $\mathbf{M}, t \models P$. We have $\mathbf{M}_{\mathrm{LR}}, t \models X$ and thus by construction of S also $\mathbf{M}, t \models X$. Hence, we obtain $\mathbf{M}, t \models P \cup X$. This implies that $\mathbf{M}_{\mathrm{RR}}$ is not a bi-equilibrium model of $P \cup X$, contradicting the assumption; we conclude that P and Q are uniformly equivalent. □

We now characterize data equivalence using *relativized uniform equivalence* [Wol04], given a fixed initial data stream.

Definition 27 *Let A be a set of @-atoms. We say two LARS programs P and Q are* uniformly equivalent relative to A (for D at t)*, denoted by $P \equiv_u^A Q$, iff $\mathcal{AS}(P \cup F, D, t) = \mathcal{AS}(Q \cup F, D, t)$ for all $F \subseteq A$.*

Given $S = (T, \upsilon)$ and a set A of @-atoms, the *restriction* $S|_A$ of S to A is the stream (T, υ') such that $\upsilon'(t) = \{a \mid a \in \upsilon(t) \text{ and } @_t a \in A\}$ for all $t \in T$.

Consider two streams $S_{\mathrm{L}} = (T, \upsilon_{\mathrm{L}})$ and $S_{\mathrm{R}} = (T, \upsilon_{\mathrm{R}})$ where $D \subseteq S_{\mathrm{L}}$. We call $\mathbf{M}_{\mathrm{LR}}$ an A-bi-*interpretation*, if either $S_{\mathrm{L}} = S_{\mathrm{R}}$ or $S_{\mathrm{L}} \subset S_{\mathrm{R}}|_A$. Moreover, $\mathbf{M}_{\mathrm{LR}}$ is a *(relativized)* A-bi-*model* of a LARS program P for data stream D at time point $t \in T$ if

(a) $\mathbf{M}_{\mathrm{RR}}, t \models P$,

(b) $M', t \not\models P^{M_{\mathrm{R}}, t}$, for all structures $M' = \langle S', W, B \rangle$ s.t. $D \subseteq S' \subset S_{\mathrm{R}}$ and $S'|_A = S_{\mathrm{R}}|_A$, and

(c) $S_{\mathrm{L}} \subset S_{\mathrm{R}}$ implies that $M'', t \models P^{M_{\mathrm{R}}, t}$ holds for some $M'' = \langle S'', W, B \rangle$ s.t. $D \subseteq S'' \subseteq S_{\mathrm{R}}$ and $S''|_A = S_{\mathrm{L}}$.

The set of all A-bi-models of P (relative to a data stream D and a time point t) is denoted by $\mathrm{bi}^A(P)$. In line with [EFW07], we present the following Lemma.

Lemma 7 *$P \equiv_u^A Q$ iff for every A-bi-interpretation $\mathbf{M}_{\mathrm{LR}}$ which is an A-bi-model of exactly one of the programs P and Q, it holds that*

(i) $\mathbf{M}_{\mathrm{RR}} \in \mathrm{bi}^A(P) \cap \mathrm{bi}^A(Q)$*, and*

(ii) there exists an A-bi-model $\mathbf{M}' = \langle (S', S_{\mathrm{R}}), W, B \rangle$ of the other program such that $S_{\mathrm{L}} \subset S' \subset S_{\mathrm{R}}$.

Proof. ($\Rightarrow$) Let $P \equiv_u^A Q$. We first show Condition (i) holds by first assuming the contrary; w.l.o.g., that $\mathbf{M}_{\mathrm{RR}} \in \mathrm{bi}^A(P)$ and $\mathbf{M}_{\mathrm{RR}} \notin \mathrm{bi}^A(Q)$. Moreover, we let $F = S_{\mathrm{R}}|_A$. We have $\mathbf{M}_{\mathrm{RR}}, t \models P$ and thus $\mathbf{M}_{\mathrm{RR}}, t \models P \cup \Gamma(F)$. Next, $\mathbf{M}_{\mathrm{RR}} \in \mathrm{bi}^A(P)$ implies that for each $M' = \langle (S', S_{\mathrm{R}}), W, B \rangle$, where $S' \subset S_{\mathrm{R}}$ and $S'|_A = S_{\mathrm{R}}|_A$, that $M', t \not\models P^{M_{\mathrm{R}}, t}$ (we also omit t in the notation of the reduct). That is, for each such M' we get

$M', t \not\models (P \cup \Gamma(F))^{M_\mathrm{R},t}$. Furthermore, for each $S_\mathrm{L} \subset S_\mathrm{R}$ with $S_\mathrm{L}|_A \subset S_\mathrm{R}|_A$, we have $\mathbf{M}_\mathrm{LR}, t \not\models \Gamma(F)$ and thus $\mathbf{M}_\mathrm{LR}, t \not\models P^{M_\mathrm{R},t} \cup \Gamma(F)$. Thus, $\mathbf{M}_\mathrm{RR}$ is a bi-equilibrium model of $P \cup \Gamma(F)$. However, we observe the following for $Q \cup \Gamma(F)$: since $\mathbf{M}_\mathrm{RR} \notin \mathrm{bi}^A(Q)$ we have $\mathbf{M}_\mathrm{RR}, t \not\models Q$ or $M', t \not\models Q^{M_\mathrm{R},t}$ for some structure $M' = \langle S', W, B\rangle$ s.t. $D \subseteq S' \subset S_\mathrm{R}$ and $S'|_A = S_\mathrm{R}|_A$. We then have $M', t \models (Q \cup \Gamma(F))^{M_\mathrm{R},t}$. That is, $\mathbf{M}_\mathrm{RR}$ is not a bi-equilibrium model of $Q \cup \Gamma(F)$. Since $\mathbf{M}_\mathrm{RR}$ is an bi-equilibrium model of $P \cup \Gamma(F)$ we conclude that $P \equiv_u^A Q$ cannot hold, which gives the contradiction.

For Condition (ii), we assume w.l.o.g. that $\mathbf{M}_\mathrm{LR} \in \mathrm{bi}^A(P)$ and $\mathbf{M}_\mathrm{LR} \notin \mathrm{bi}^A(Q)$. From (i) follows that $S_\mathrm{L} \subset S_\mathrm{R}$; and then $S_\mathrm{L} = S_\mathrm{L}|_A$. Towards a contradiction, suppose now that no bi-structure $\mathbf{M'} = \langle (S', S_\mathrm{R}), W, B\rangle$, where $S_\mathrm{L} \subset S' \subset S_\mathrm{R}$, is in $\mathrm{bi}^A(Q)$. Since for every $M'' = \langle S'', W, B\rangle$ s.t. $D \subseteq S'' \subset S_\mathrm{L}$ we have that $M'', t \not\models Q^{M_\mathrm{R}} \cup \Gamma(S_\mathrm{L})$, $\mathbf{M}_\mathrm{RR}$ is the only A-bi-model of $Q \cup \Gamma(S_\mathrm{L})$ with M_R as the right-stream. Consequently, $\mathbf{M}_\mathrm{RR}$ is a bi-equilibrium model of $Q \cup \Gamma(S_\mathrm{L})$. However, we observe the following for $P \cup \Gamma(S_\mathrm{L})$: from $\mathbf{M}_\mathrm{LR} \in \mathrm{bi}^A(P)$ follows that $M', t \models P^{M_\mathrm{R},t}$ for some $M' = \langle S', W, B\rangle$ s.t. $D \subseteq S' \subseteq Y$ and $S'|_A = S_\mathrm{L}|_A$. Moreover, $M', t \models (P \cup \Gamma(S'))^{M_\mathrm{R},t}$. It follows that $\mathbf{M}_\mathrm{RR}$ is not a bi-equilibrium model of $P \cup \Gamma(S_\mathrm{L})$, which contradicts the assumption that $P \equiv_u^A Q$. We obtain that for some stream S' s.t. $S_\mathrm{L} \subset S' \subset S_\mathrm{R}$ the bi-structure $\mathbf{M'} = \langle (S', S_\mathrm{R}), W, B\rangle$ in an A-bi-model of Q.

($\Leftarrow$) We assume Conditions (i) and (ii) hold for every A-bi-interpretation $\mathbf{M}_\mathrm{LR}$ which is an A-bi-model of exactly one of the programs P and Q. Suppose there exists a subset $F \subseteq A$ and a structure $M = \langle Z, W, B\rangle$ such that $\mathbf{M} = \langle (Z, Z), W, B\rangle$ is, w.l.o.g., (1) a bi-equilibrium model of $P \cup F$ and (2) not of $Q \cup F$. From (1) we get $F \subseteq \Gamma(Z)$, $M, t \models P$ and $M', t \not\models P^{M,t}$ for each $M' = \langle Z', W, B\rangle$ such that $Z' \subset Z$ and $Z'|_A = Z|_A$. Consequently, $\mathbf{M} \in \mathrm{bi}^A(P)$. From (2) we derive that either $M, t \not\models Q \cup F$ or there is some structure $M' = \langle Z', W, B\rangle$, where $Z' \subset Z$, such that $M', t \models (Q \cup F)^{M,t}$.

We first assume that $M, t \not\models Q \cup F$. From $F \subseteq \Gamma(Z)$ we obtain $M, t \not\models Q$ and thus $\mathbf{M} \notin \mathrm{bi}^A(Q)$ which is in conflict with Condition (i). We conclude that $M, t \models Q \cup F$ must hold and there exists some $M' = \langle Z', W, B\rangle$, where $Z' \subset Z$, such that $M', t \models (Q \cup F)^{M,t}$, i.e., $M', t \models Q^{M,t} \cup F$. We can exclude the case that $Z'|_A = Z|_A$ since this would imply that $\mathbf{M} \notin \mathrm{bi}^A(Q)$. That is, we have the following relations: $M, t \models Q$; $M', t \not\models Q^{M,t}$ for each $M' = \langle Z', W, B\rangle$ s.t. $Z'|_A = Z|_A$; and there exists some $M'' = \langle Z'', W, B\rangle$, where $Z''|_A \subset Z|_A$, s.t. $M'', t \models Q^{M,t}$. We obtain that $\mathbf{M}''_A = \langle (Z''|_A, Z), W, B\rangle \in \mathrm{bi}^A(Q)$ and $\mathbf{M}'' = \langle (Z'', Z), W, B\rangle \notin \mathrm{bi}^A(P)$. We also get that $\mathbf{M}''_A \notin \mathrm{bi}^A(P)$ since assuming the contrary would imply $\mathbf{M}''_A \in \mathrm{bi}^A(P \cup F)$, contradicting that $\mathbf{M}$ is a bi-equilibrium model of $P \cup F$. That is, $\mathbf{M}''_A$ is an A-bi-model only of Q. Due to Condition (ii) we obtain that $M' = \langle (S', Z), W, B\rangle \in \mathrm{bi}^A(P)$ for some $Z'' \subset S' \subset Z$. Since $F \subseteq \Gamma(Z)$, $M' \in \mathrm{bi}^A(P \cup F)$, which contradicts the assumption that $\mathbf{M}$ is a bi-equilibrium model of $P \cup F$.

In conclusion, we obtain that by assuming (i) and (ii) for every $\mathbf{M}_\mathrm{LR}$ that is an A-bi-model of exactly one of the programs P and Q, there is no set $F \subseteq A$ of @-atoms and stream Z such that Z is an answer stream of exactly one of $P \cup F$ and $Q \cup F$, i.e., $P \equiv_u^A Q$ holds. □

The following result based on Lemma 7 characterizes relativized uniform equivalence, and resembles the one for uniform equivalence, where bi-models are replaced by A-bi-models.

Theorem 23 (Relativized Uniform Equivalence) *$P \equiv_u^A Q$ iff*

(i) for each $\mathbf{M}_{\mathrm{RR}}$, $\mathbf{M}_{\mathrm{RR}} \in \mathrm{bi}^A(P)$ iff $\mathbf{M}_{\mathrm{RR}} \in \mathrm{bi}^A(Q)$, and

(ii) for each $\mathbf{M}_{\mathrm{LR}}$, where $S_{\mathrm{L}} \subset S_{\mathrm{R}}$, $\mathbf{M}_{\mathrm{LR}} \in \mathrm{bi}^A(P)$ (resp. $\mathbf{M}_{\mathrm{LR}} \in \mathrm{bi}^A(Q)$) iff $\mathbf{M} \in \mathrm{bi}^A(Q)$ (resp. $\mathbf{M} \in \mathrm{bi}^A(P)$), for some $\mathbf{M} = \langle (S, S_{\mathrm{R}}), W, B \rangle$ s.t. $S_{\mathrm{L}} \subseteq S \subset S_{\mathrm{R}}$.

The proofs of Theorems 21-23 are similar to those for answer set programs (cf. [LPV01], [EF03], [EFW07]) and exploit the following key properties:

(1) the reduct of the union of two programs P and Q is the union of their reducts, i.e., $(P \cup Q)^{M,t} = P^{M,t} \cup Q^{M,t}$,

(2) the reduct of a set of atoms (facts) F is F itself, i.e., $F^{M,t} = F$,

(3) an atom evaluates to true iff it is in the interpretation stream,

(4) a structure is a model of the union of two programs iff it is a model of both programs, i.e., $M, t \models P \cup Q$ iff $M, t \models P$ and $M, t \models Q$.

We now obtain data equivalence as special case of relativized uniform equivalence.

Corollary 5 *Let $D = (T, v)$ be a data stream and $A = \{@_t a \mid a \in \mathcal{A}^{\mathcal{E}} \text{ and } t \in T\}$. Then, $P \equiv_d Q$ iff $P \equiv_u^A Q$.*

Recall that plain LARS permits only intensional atoms and @-atoms with intensional atoms in rule heads. Since A only refers to extensional atoms, Condition (c) of an A-bi-model holds for every S_{L} such that $D \subseteq S_{\mathrm{L}} \subset S_{\mathrm{R}}|_A$.

5.5 LARS Here-and-There and Monotone Windows

In Definition 23, the semantics of the window operator $\boxplus^w$ was defined in bi-LARS by straight branching into separate evaluation of the left and the right stream. Consider the following alternative $\boxplus^w$ semantics.

Definition 28 (Recursive $\boxplus^w$) *We define the following alternative $\boxplus^w$ semantics for* bi-*LARS. Let* $\mathrm{w} \in \{\mathrm{L}, \mathrm{R}\}$.

$$\mathbf{M}, t, \mathrm{w} \Vdash \boxplus^w \alpha \;:\Leftrightarrow\; \mathbf{M}', t, \mathrm{w} \Vdash \alpha\,,$$

where $\mathbf{M}' = \langle (S'_{\mathrm{L}}, S'_{\mathrm{R}}), W, B \rangle$ and $S'_{\mathrm{w}} = w(S_{\mathrm{w}}, t)$.

This recursive variant may in general break the connection between left and right, i.e., the relation $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$.

Example 44 Consider streams $S_{\mathrm{L}} = ([0,4], v_{\mathrm{L}})$ and $S_{\mathrm{R}} = ([0,4], v_{\mathrm{R}})$ as depicted in Figure 5.1, where $S_{\mathrm{L}} \subset S_{\mathrm{R}}$. Applying a tuple-based window operator with size 3 at $t = 4$ returns $S'_{\mathrm{L}} = ([1,4], \{1 \mapsto \{a\},\ 3 \mapsto \{b\},\ 4 \mapsto \{a\}\})$ as substream of S_{L}, and $S'_{\mathrm{R}} = ([3,4], \{3 \mapsto \{b\}, 4 \mapsto \{a, c\}\}$ for S_{R}. We observe that $S'_{\mathrm{L}} \not\subseteq S'_{\mathrm{R}}$, i.e. the substream relation breaks. ∎

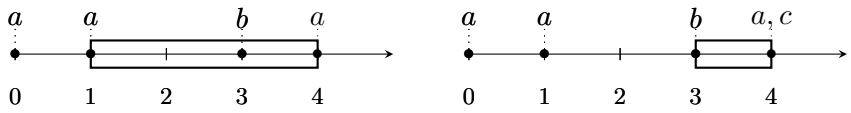


Figure 5.1: Tuple-based windows of size 3 at $t = 4$.

In Example 44, the window is nonmonotonic in the sense that by increasing the input stream, atoms may disappear from the output. When excluding such *nonmonotonic* windows, the recursive version for $\boxplus^w$ semantics may be equally used.

Definition 29 *We call a window function w* monotone*, if for every pair of streams $S_1 = (T_1, \upsilon_1)$ and $S_2 = (T_2, \upsilon_2)$ it holds that*

$$S_1 \subseteq S_2 \text{ implies } w(S_1, t) \subseteq w(S_2, t) \text{ for all } t \in T_1,$$

i.e., w preserves substreams, and

$$T_1 = T_2 \text{ implies } T_1' = T_2' \text{ for all } t \in T_1,$$

where $w(S_i, t) = (T_i', \upsilon_i')$ (for $i = 1, 2$).

Likewise, we call a window operator $\boxplus^w$ *monotone* if the underlying window function w is monotone. For instance, time-based window operators have this property. Using Definition 28 for a variant of bi-LARS on monotone windows can be seen as an extension of Here-and-There [Hey30] underlying Equilibrium Logic [Pea06].

Definition 30 (HT-entailment) HT-*entailment is defined as variant of* bi-*LARS entailment (Definition 23), using instead Definition 28 for the semantics of $\boxplus^w$ and $\neg\alpha := \alpha \rightarrow \bot$ for negation.*

Based on HT-entailment, we obtain a conservative extension of Pearce's Equilibrium Logic for LARS with monotone windows, that treats nested implications intuitionistically, and is thus different from FLP-based semantics. Under limited nesting of negation, the two semantics actually coincide; e.g., for the following class of formulas/programs:

Definition 31 ($\mathcal{F}_{\mathbf{HT}}$, $\mathbf{LARS}_{\mathbf{HT}}$) *By $\mathcal{F}_{\mathrm{HT}}$ we denote the class of LARS formulas where*

(i) each $\boxplus^w$ is monotone,

(ii) each subformula $\varphi \rightarrow \psi$ expresses negation (i.e., $\psi = \bot$), and

(iii) no negation occurs within the scope of $\Diamond$ or another negation.

By $LARS_{\mathrm{HT}}$ we denote the class of LARS programs P where all formulas in $\mathcal{F}(P)$ are in $\mathcal{F}_{\mathrm{HT}}$.

Note that nested negation must be excluded, as e.g. the rule $a \leftarrow \neg\neg a$ has the equilibrium models $(\emptyset, \emptyset)$ and $(\{a\}, \{a\})$. Only the first one amounts to an FLP-answer set.

If we drop $\rightarrow$ (i.e., negation) from HT-formulas, we get complete independence of the left-structure from the right-structure. Recall that we always consider the same timeline for the left-stream $S_{\mathrm{L}} = (T, \upsilon_{\mathrm{L}})$ and the right-stream $S_{\mathrm{R}} = (T, \upsilon_{\mathrm{R}})$ (cf. Section 5.2).

Lemma 8 *Let $\mathbf{M} = \langle \mathbf{S}, W, B \rangle$ s.t. $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ and let $\alpha \in \mathcal{F}_{\mathrm{HT}}$ be a formula in which $\rightarrow$ does not occur, i.e., without negation. Then, under* HT*-entailment, $\mathbf{M}, t \Vdash \alpha$ iff $M_{\mathrm{L}}, t \Vdash \alpha$.*

Proof. Intuitively, the lemma holds since $\rightarrow$ is the only connective establishing a connection from left to right. We argue formally by the structure of α.

($\Rightarrow$) We have shown this part already in Lemma 4, with exception of the alternative $\boxplus^w$ semantics considered here. We defined $\mathbf{M}, t, \mathrm{L} \Vdash \boxplus^w \varphi$ iff $\mathbf{M}', t, \mathrm{L} \Vdash \varphi$ in the bi-structure $\mathbf{M}'$ resulting from window application. For the latter we get $M'_{\mathrm{L}}, t \Vdash \varphi$ inductively, hence $M_{\mathrm{L}}, t \Vdash \boxplus^w \varphi$ by definition.

($\Leftarrow$) Let $M_{\mathrm{L}}, t \Vdash \alpha$. If α is an atom a, then $a \in \upsilon_{\mathrm{L}}(t)$ or $a \in B$, as accessed both from the structure M_{L} and the bi-structure $\mathbf{M}$, i.e., we get the base case $\mathbf{M}, t, \mathrm{L} \Vdash a$ by definition. For α of the form $\varphi \wedge \psi$, we get by definition that $M_{\mathrm{L}}, t \Vdash \varphi$ and $M_{\mathrm{L}}, t \Vdash \psi$, hence $\mathbf{M}, t, \mathrm{L} \Vdash \varphi$ and $\mathbf{M}, t, \mathrm{L} \Vdash \psi$ inductively, and then $\mathbf{M}, t, \mathrm{L} \Vdash \varphi \wedge \psi$ by definition. The cases for $\Diamond$, $\Box$, $@_{t'}$ and $\boxplus^w$ are shown with the same straightforward argument. $\square$

With an inductive argument, one can show that the central property of Proposition 4 carries over to formulas in $\mathcal{F}_{\mathrm{HT}}$ under HT-entailment.

Proposition 5 *Let $\mathbf{M} = \langle \mathbf{S}, W, B \rangle$ s.t. $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ and let $\alpha \in \mathcal{F}_{\mathrm{HT}}$. Then, under* HT*-entailment, $\mathbf{M}, t \Vdash \alpha$ iff $M_{\mathrm{L}}, t \Vdash \alpha$ and $M_{\mathrm{R}}, t \Vdash \alpha$.*

Proof. We consider $\mathbf{M} = \langle (S_{\mathrm{L}}, S_{\mathrm{R}}), W, B \rangle$ such that $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$; $M_{\mathrm{L}} = \langle S_{\mathrm{L}}, W, B \rangle$ and $M_{\mathrm{R}} = \langle S_{\mathrm{R}}, W, B \rangle$. The proposition is again shown by induction on the structure of α. The arguments given for Lemmas 3-5 carry over for the base case where α is an atom, and the cases for connectives $\wedge$, $\Box$, and $@_{t'}$. It remains to argue the cases $\boxplus^w$, $\Diamond$ and $\rightarrow$.

($\Rightarrow$) Let $\mathbf{M}, t \Vdash \alpha$. We first consider $\alpha = \boxplus^w \varphi$ and observe that Lemma 2 carries over for HT-entailment, in particular for the window operator; this is immediate from the new definition. Thus, we obtain $M_{\mathrm{R}}, t \Vdash \boxplus^w \varphi$, and $M_{\mathrm{L}}, t \Vdash \boxplus^w \varphi$ is given directly by the definition of $\boxplus^w$. Next, consider the form $\alpha = \varphi \rightarrow \bot$. By definition, we have $\mathbf{M}, t, \mathrm{L} \nVdash \varphi$ (or $\mathbf{M}, t, \mathrm{L} \Vdash \bot$, which is false) and $\mathbf{M}, t, \mathrm{R} \Vdash \varphi \rightarrow \bot$, i.e., $\mathbf{M}, t, \mathrm{R} \nVdash \varphi$. Since Lemma 2 carries over we obtain that $M_{\mathrm{R}}, t \nVdash \varphi$ and consequently $M_{\mathrm{R}}, t \Vdash \varphi \rightarrow \bot$. Since the considered formula class guarantees that φ does not contain $\rightarrow$, it follows from Lemma 8 that $M_{\mathrm{L}}, t \nVdash \varphi$ and we obtain $M_{\mathrm{L}}, t \Vdash \varphi \rightarrow \bot$. Finally, let α be of the form $\Diamond\varphi$. By definition, $\mathbf{M}, t', \mathrm{L} \Vdash \varphi$ for some $t' \in T$ and thus by induction $M_{\mathrm{L}}, t' \Vdash \varphi$. The considered HT-fragment excludes any occurrence of negation within φ. Together with the property that $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$ we thus obtain $M_{\mathrm{R}}, t' \Vdash \varphi$, and in conclusion that $M_{\mathrm{L}}, t \Vdash \Diamond\varphi$ and $M_{\mathrm{R}}, t \Vdash \Diamond\varphi$.

($\Leftarrow$) Let $M_{\mathrm{L}}, t \Vdash \alpha$ and $M_{\mathrm{R}}, t \Vdash \alpha$. We get the cases for $\boxplus^w$ and $\Diamond$ by Lemma 8, thus we consider $\alpha = \varphi \rightarrow \bot$. That is, we have $M_{\mathrm{L}}, t \nVdash \varphi$ and $M_{\mathrm{R}}, t \nVdash \varphi$. Since φ does not contain $\rightarrow$, we obtain again by Lemma 8 that $\mathbf{M}, t, \mathrm{L} \nVdash \varphi$, and thus $\mathbf{M}, t \Vdash \varphi \rightarrow \bot$. □

This allows one to establish the characterization in Theorem 19 for this setting. Thus, for $\mathrm{LARS}_{\mathrm{HT}}$ programs, FLP-based answers streams and HT-equilibrium models coincide, and the equivalence notions can equally be characterized by HT-entailment.

Theorem 24 (LARS Here-and-There) *Theorems 19-23 also hold for $\mathrm{LARS}_{\mathrm{HT}}$ programs under* HT*-entailment.*

Proof. With Proposition 5 at hand, the proof of Theorem 24 consists in just following the steps of the proofs of Theorems 19-23 using $\mathrm{LARS}_{\mathrm{HT}}$ programs under HT-entailment instead of $\mathrm{LARS}_{\mathrm{bi}}$ programs under bi-entailment. □

We note that $\mathrm{LARS}_{\mathrm{HT}}$ includes plain LARS programs with monotone windows such as time-based windows, hence we obtain the following.

Corollary 6 *For plain LARS with monotone windows, the window semantics of Definition 23 and Definition 28 coincide.*

In summary, we showed how major equivalence notions from Answer Set Programming carry over to a large class of LARS programs. We characterized strong equivalence, uniform equivalence, and relativized uniform equivalence. The introduced data equivalence was obtained as special case of the latter. All results depend on the property that the left-stream is included in the right-stream; where this property might fail to hold, bi-LARS has to branch into separate evaluation in the underlying LARS structures. For the considered fragment $\mathrm{LARS}_{\mathrm{HT}}$, which does not break the connection $S_{\mathrm{L}} \subseteq S_{\mathrm{R}}$, a more elegant, recursive definition of the central window operator is possible.

Next, we summarize the complexity results for the obtained characterizations.

5.6 Computational Complexity of Deciding Equivalences

We now give an overview of the results from [BDE16] regarding the complexity of deciding $P \equiv_e Q$ for equivalence notions $e = o, s, u, d$. In line with the definition of the considered notions of equivalence, i.e., ordinary/strong/uniform/data equivalence, we assume that the data stream D, programs P and Q as well as the time point t are given. We recall the result from Section 3.3 that, given a window nesting depth bounded by some constant $k \geq 0$, model checking for LARS formulas is feasible in polynomial time and satisfiability is NP-complete; for LARS programs we obtained co-NP-completeness and Σ_2^p-completeness, respectively. We consider plain LARS as guiding fragment, which falls into this category since it precludes nested windows (i.e., $k = 0$); and a stratified fragment (as in Section 6.5.1). Moreover, we distinguish monotone and non-monotone windows. The results are shown in Table 5.1; we now briefly summarize how they are obtained.

	$\boxplus^w$	cons.	$\equiv_o$	$\equiv_s$	$\equiv_u$	$\equiv_d$
plain LARS	non-mon.	Σ_2^p	Π_2^p	co-NP	Π_2^p	Π_2^p
	monotone	NP	co-NP	co-NP	co-NP	co-NP
stratified LARS	(both)	P	P	co-NP	co-NP	co-NP

Table 5.1: Complexity results (completeness) for consistency and equivalence checking

Based on the complexity of deciding $M, t \Vdash \alpha$ (resp. satisfiability), most upper bounds are derived directly from the characterizations above. Deciding strong equivalence is in co-NP, while uniform and data equivalence is in Π_2^p, since refutation is possible in nondeterministic polynomial time, given an NP oracle to verify a guess for a counter example to equivalence. When considering stratified programs, the answer stream is unique and can be computed by a fixed-point computation in polynomial time as usual. In this case, answer set existence is also feasible in polynomial time, as well as checking ordinary equivalence. We note that in case of monotone windows in plain LARS, negative literals can be deleted from the FLP-reduct. Thus, when restricting to monotone windows, the minimality check for a model can also be carried out by a fixed-point computation on the reduct. As a consequence, answer set existence and ordinary equivalence drop from Σ_2^p and Π_2^p to NP and co-NP, respectively.

By restricting all formulas in plain LARS to atoms, the class of normal logic program is obtained (cf. Section 4.1). This subsumption allows us to infer lower bounds from the analogous equivalence notions in ASP [EFW07], where answer set existence is NP-complete, and deciding strong/uniform equivalence are co-NP-complete. Regarding uniform and data equivalence, we observe the following relation in the context of ordinary logic programs P and Q.

Proposition 6 *Let A be a set of atoms, A' be a copy of A and let $P' = P \cup R'$ and $Q' = Q \cup R'$, where $R' = \{a \leftarrow a' \mid a \in A\}$. Then, $P \equiv_u Q$ (w.r.t. A) iff $P' \equiv_d Q'$ (w.r.t. A').*

Using rules of the form $@_t a \leftarrow @_t a'$, this approach extends to LARS. Consequently, it suffices to consider uniform equivalence, i.e., the Π_2^p-hardness result (for plain LARS with non-monotone windows).

Furthermore, it remains to argue the hardness results for ordinary equivalence (Π_2^p) and answer stream existence (Σ_2^p). All of these results can be shown with techniques from [EG95] and [EF03], where they are applied for disjunctive logic programs. In particular, the proofs carry over for plain programs without negations; however, negation can be emulated using non-monotone windows. For instance, consider a stream $S = (T, \upsilon)$ with auxiliary atoms a' at every time point. Now, consider a window operator $\boxplus^{\neg a}$ based on a window function $w_{\neg a}$ that will delete, when applied at time point t, atom a' from $\upsilon(t)$ if $a \in \upsilon(t)$. If a' appears as fact in a program P we obtain for every model M that $M, t \Vdash \boxplus^{\neg a} @_t\, a'$ iff $M, t \Vdash \neg a$, i.e., we effectively provide negation in form of a (non-monotone) window operator.

In conclusion, the complexity of deciding the considered equivalence is, for plain LARS with non-monotone windows, not harder than for disjunctive logic programs, and for most cases easier when restricting to monotone windows or a stratified fragment that permits fixed-point evaluation.

5.7 Discussion and Related Work

In contrast to bi-LARS, LARS includes also a reset operator $\triangleright$ by which one jumps back to the original stream during evaluation. We skipped an explicit study of $\triangleright$ in this chapter for multiple reasons. First, $\triangleright$ is relevant only for nested windows; the application of $\triangleright$ outside the scope of any window operator has no effect. Nevertheless, $\triangleright$ might be of practical interest, but we observe the following: bi-LARS entailment (Definition 23) branches for window operators into separate evaluation in the underlying LARS structures, hence the obtained results remain when using $\triangleright$ inside the scope of a window. On the other hand, introducing $\triangleright$ formally in bi-LARS makes notation more involved as we would have to carry the current stream left of $\Vdash$ as in the LARS entailment definition. That is, we could define the bi-LARS entailment for reset by

$$\mathbf{M}, \mathbf{S}, t, \mathrm{w} \Vdash \triangleright \alpha \;:\Leftrightarrow\; \mathbf{M}, \mathbf{S}^\star, t, \mathrm{w} \Vdash \alpha\,, \tag{5.6}$$

where $\mathbf{S}^\star = (S^\star_\mathrm{L}, S^\star_\mathrm{R})$ is the original bi-stream and $\mathbf{S} = (S_\mathrm{L}, S_\mathrm{R})$ contains substreams componentwise. We then require re-definitions like $\mathbf{M}, t \Vdash \alpha \;:\Leftrightarrow\; \mathbf{M}, \mathbf{S}^\star, t, \mathrm{L} \Vdash \alpha$, etc. Note that the definition due to (5.6) is analogous to the recursive window in Definition 28. As it cannot break the $\subseteq$-relation between left-stream and right-stream (but only reintroduces the original streams which are in this relation), the results for the monotone variant of Section 5.5 extend also for reset.

Related Work

Lifschitz et al. [LPV01] introduced strong equivalence of logic programs under ASP semantics and showed that it coincides with equivalence in the logic of Here-and-There. Inspired by this, Eiter et al. [EFW07] characterized uniform and relativized notions of equivalence in ASP in terms of HT-interpretations (H, T). We generalize this to the LARS framework with bi-structures $(S_\mathrm{L}, S_\mathrm{R})$ containing pairs of streams.

As regards optimization in stream reasoning, to our knowledge not much foundational work exists. Typically, the interest is concentrated on dealing with evolving data, and to develop incremental evaluation techniques, e.g., [MNPH15]; [RP11]; [GSS11]; [BBC$^+$10b]; [BDE15]. In the context of data processing, [ABW06] studied query equivalence for the Continuous Query Language (CQL), but at a very elementary level. Since CQL can essentially be captured by LARS programs (cf. [BDEF15] respectively Section 4.3), results on LARS equivalence may be fruitfully applied in this context as well.

Naturally, stream reasoning relates to temporal reasoning; in [CV07], nonmonotonic Linear Temporal Equilibrium Logic (TEL) was presented as an extension of Pearce's Equilibrium Logic [Pea06] to Linear Temporal Logic (LTL), defining temporal stable

models over infinite structures. More recently, a complexity analysis of TEL was given in [BP15]. Notably, strong equivalence for TEL theories amounts to equivalence in the underlying Temporal Here-and-There logic [ACPV08, CD14]. However, TEL differs from LARS in several respects. LARS aims primarily at finite (single-path) structures, and the notion of window, which requires to go beyond the HT setting, has no counterpart in TEL. Furthermore, the temporal operators in LARS are more geared to access of data in windows in practice.

Outlook

We have characterized FLP-based answer streams of LARS programs in logical terms by means of bi-LARS, as well as several notions of equivalence for LARS programs, as a basis for optimization. Furthermore, we have shown that a monotone variant of bi-LARS leads to an extension of the logic of Here-and-There for a large fragment of LARS including plain LARS.

In future work, our studies can be extended in several directions. Besides other notions of equivalence, specific program classes are of practical relevance. In particular, by confining to widely used time-based and tuple-based window operators one might exploit more specific properties than by the distinction of monotone vs. non-monotone ones. Related to this is identifying maximal fragments (relative to some criteria) where Here-and-There semantics coincides with bi-LARS, which is tailored for FLP. Furthermore, one might introduce window operators to the more expressive temporal equilibrium logic. Apart from potential extensions of bi-LARS to capture according semantics, combining nonmonotonic temporal reasoning with features to drop data is an intriguing issue.

CHAPTER 6

Incremental Reasoning for Plain LARS Programs

Reasoning over streaming data suggests a reconsideration of the concept of a *solution* to a problem, and how to obtain it. In contrast to static data, streams change continuously, and so may their evaluations: conclusions based on expired data must be retracted, additional information might arrive and yield additional derivations, even undermine previous ones, and so forth. In Chapter 3 we developed a model-based semantics which allows us to view the evolution of solutions as a sequence of models (resp. answer streams or sets thereof) over time. Assuming an algorithm that computes the answer streams at a given time point, we thus get an algorithm for stream reasoning (in our terms) by repeatedly computing models over time. However, such a naive method would also recompute those parts that can remain. That is to say, repeatedly computing models from scratch is hardly a practical solution.

Just like the data stream changes continuously, so might the solution to a formalized problem. Consequently, the idea of *incremental reasoning* is to efficiently update the previous solution instead of rederiving everything from scratch. The model computation task then gets replaced by the model update task, in which we have to find out which atoms of the previous model are still guaranteed to hold and which ones might change due to the update of the stream. If we are able to quickly determine large portions of the previous model that remains valid we can update solutions more efficiently.

In this chapter we explore the idea of incremental reasoning in terms of incremental model update for plain LARS programs (as defined in Section 4.1), a fragment that is a natural starting point for practical applications. Chapter 7 will then present an implementation and evaluation of the resulting algorithmic techniques.

Outline

Section 6.1 introduces the conceptual idea of the central algorithms developed in this chapter. In Section 6.2 we start by reviewing Doyle's Justification-based Truth Maintenance System (JTMS) [Doy79], which will serve as a building block. We present in Section 6.3 an encoding of plain LARS programs to ASP such that answer streams naturally correspond to answer sets. In Section 6.4 we show how a similar encoding can be updated incrementally when the stream progresses, i.e., when time passes by or when new atoms stream in. Given a model update technique that works by updating a program (such as JTMS), we thus obtain an incremental model update mechanism for plain LARS. The techniques of Sections 6.3 and 6.4 underlie the Ticker engine, as presented in Chapter 7. Briefly mentioning further work, we present in Section 6.5.1 an alternative approach which extends truth maintenance techniques directly for plain LARS. Furthermore, Section 6.5.2 outlines an efficient algorithm for incrementally grounding positive and stratified plain LARS programs.

Publications

Section 6.2 is for the largest part a copy of the technical report [Bec17], which was released as supplementary material to publication [BEF17]. The latter is the basis of Sections 6.3-6.4. Section 6.5 summarizes [BDE15] and Section 6.5.2 presents the central ideas of [BBU17].

6.1 Core Idea

We will now introduce the central motivation and idea behind the incremental reasoning technique for (plain) LARS programs as developed in this chapter.

Model Update Tick by Tick

Recall the model-based semantics of LARS programs, as defined in Chapter 3 (Section 3.2.2), which is based on the notion of *answer streams*: given a program P, a data stream $D = (T, \upsilon)$ and a time point $t \in T$, an answer stream S minimally adds intensional atoms to D such that S is a model of P at t. We denote by $\mathcal{AS}(P, D, t)$ all answer streams. In this formal model, we assume that the timeline T is a closed interval (in $\mathbb{N}$) but in practice a stream can rather be seen as a sequence of atoms assigned with time points without an explicit timeline. When we evaluate a stream at time t, we implicitly take $T = [t_0, t]$, where t_0 may either be the first evaluation time or the oldest time point where data is available. (Over time, the exact t_0 will not play a role, as long as no window expresses an interval that goes beyond it; we will discuss this.)

Thus, assume an evaluation of D at t from the formal point of view using $T = [t_0, t]$. Receiving an additional atom a at time t then amounts to augmenting the evaluation $\upsilon(t) := \upsilon(t) \cup \{a\}$. On the other hand, proceeding to the next time point $t + 1$ amounts to considering the augmented timeline $T = [t_0, t + 1]$ (where initially $\upsilon(t + 1) = \emptyset$). We

thus can consider an incremental change of the stream D towards a stream D' due to such a *tick*, i.e., (i) a single new atom or (ii) a single time point passing by. A tick thus is the minimal granularity for incremental reasoning, where we ask: given an answer stream $S \in \mathcal{AS}(P, D, t)$, how can we *update* S to a model S' for the next tick, i.e., such that (i) $S' \in \mathcal{AS}(P, D', t)$, respectively (ii) $S' \in AS(P, D', t')$ holds?

Example 45 Consider the data stream $D_7 = ([0, 7], \upsilon)$, where $\upsilon = \{7 \mapsto a(x)\}$, and the program P comprising the single rule

$$r\colon \quad b(X) \leftarrow \boxplus^2 \Diamond a(X)\,.$$

Evaluating at $t = 7$, we get $\mathcal{AS}(P, D_7, t) = \{(T, \upsilon')\}$, where $\upsilon'(7) = \{a(x), b(x)\}$. That is to say, P has a single answer stream which infers $b(x)$ to hold at evaluation time.

If we now proceed to time point $t' = 8$, where we formally consider the stream $D_8 = ([0, 8], \{7 \mapsto \{a(x)\}\})$, we obtain the unique answer stream $S_8 = ([0, 8], \{7 \mapsto \{a(x)\}, 8 \mapsto \{b(x)\}\}$. Data streams D_7 and D_8 can be seen as formalizations of intermediate states of an input stream D, and the timeline can be viewed implicitly, ranging from 0 to the evaluation time point. In that regard, we know that $b(x)$ will be in the model at t in all evaluation time points $t \in \{7, 8, 9\}$ due to the length of the sliding time-based window. In other words, there is no need to recompute $b(x)$ for time points 8 and 9; we can keep it in the model (relative to the evaluation time point), regardless of the data that might appear at these time points. ∎

Example 45 illustrated how parts of a model might be retained or carried over for the next tick. Recall that the pattern $\boxplus^w \Diamond a$ deserves special attention, as it formalized the prominent snapshot semantics, as discussed e.g. in Section 4.3. As instances for the window function w we shall consider time-based and tuple-based windows which are frequently used kinds of windows. More specifically, focusing on sliding versions of these windows, we hope to exploit their fully incremental nature during incremental model update.

Static Encoding to ASP

Towards our incremental reasoning technique, we first consider a natural encoding to Answer Set Programming.

Example 46 Rule r of Example 45 can be naturally translated into the following ASP rules, where we assume an additional predicate $now(t)$ to model the query time t. Moreover, $a_@(x, t)$ means that $a(x)$ holds at time t.

$$\begin{aligned}
r_b\colon &\quad b(X) \leftarrow \omega(X)\\
r_1\colon &\quad \omega(X) \leftarrow now(N), a_@(X, N)\\
r_2\colon &\quad \omega(X) \leftarrow now(N), a_@(X, N-1)\\
r_3\colon &\quad \omega(X) \leftarrow now(N), a_@(X, N-2)
\end{aligned}$$

Base rule r_b corresponds to the original LARS rule and replaces the window atom $\boxplus^2 \Diamond a(X)$ by an encoding atom $\omega(X)$ which shall be derived (for a substitution of X) iff

the window atom itself holds. To this end, rules r_1-r_2 derive $\omega(X)$ in case $a(X)$ holds at the current time point N, or at one of the two time points before. Clearly, given $a_@(x, 7)$ (of the data stream from Example 45), we obtain $b(x)$ for query times $t = 7, 8, 9$, which are reflected by providing the additional atom $now(7)$, $now(8)$, $now(9)$, respectively.

For instance, the answer stream $([0, 8], \{7 \mapsto \{a(x)\}, 8 \mapsto \{b(x)\}\}$ of P at 8 corresponds to the answer set $\{a_@(x, 7), \omega(x), b(x)\}$ (where $\omega(x)$ is auxiliary) of the program

$$\begin{aligned} b(X) &\leftarrow \omega(X) \\ \omega(X) &\leftarrow now(N), a_@(X, N) \\ \omega(X) &\leftarrow now(N), a_@(X, N-1) \\ \omega(X) &\leftarrow now(N), a_@(X, N-2) \\ now(8) &\leftarrow \\ a_@(x, 7) &\leftarrow ; \end{aligned}$$

i.e., the ground program

$$\begin{aligned} b(x) &\leftarrow \omega(x) \\ \omega(x) &\leftarrow now(8), a_@(x, 8) \\ \omega(x) &\leftarrow now(8), a_@(x, 7) \\ \omega(x) &\leftarrow now(8), a_@(x, 6) \\ now(8) &\leftarrow \\ a_@(x, 7) &\leftarrow . \end{aligned}$$

Note that $b(x)$ (to hold at time 8) could be also represented as $b_@(x, 8)$, but using the convention that for any predicate p, $p(\mathbf{x})$ equals $p_@(\mathbf{x}, t)$ for the current evaluation time point t, we can keep the first rule (corresponding to groundings of r_b). We later use explicit rules for this equivalence. ∎

The idea of this encoding essentially carries over for tuple-based windows and other temporal modalities; some cases however are more subtle. For the sake of motivation, we stick here to this simple case.

Incremental Encoding

We now illustrate the central idea for incremental update mechanism, which is based on the observation that a slight variation of the given ASP encoding can be adjusted incrementally from tick to tick.

Example 47 (cont'd) Assuming that we remove the auxiliary predicate *now* and directly deal with partial groundings, we get at time $t = 7$ the following rules.

$$\begin{aligned} r_b\colon&\ \ b(X) \leftarrow \omega(X) \\ r'_1\colon&\ \omega(X) \leftarrow a_@(X, 7) \\ r'_2\colon&\ \omega(X) \leftarrow a_@(X, 6) \\ r'_3\colon&\ \omega(X) \leftarrow a_@(X, 5) \end{aligned}$$

We observe that the rules r'_1, r'_2, r'_3 cover the window timeline $[5, 7]$. If we move to time 8, i.e. the timeline $[6, 8]$, we can keep any groundings for time points 6 and 7 as well as for the base rule r_b; only groundings for rule r'_3 need to be removed, and groundings for time 8 have to be added. This adjustment can be illustrated as follows:

$$\begin{array}{cll} & r_b\colon & b(x) \leftarrow \omega(x) \\ + & r_8\colon & \omega(x) \leftarrow a_{@}(x, 8) \\ & r_7\colon & \omega(x) \leftarrow a_{@}(x, 7) \\ & r_6\colon & \omega(x) \leftarrow a_{@}(x, 6) \\ - & r_5\colon & \omega(x) \leftarrow a_{@}(x, 5) \end{array}$$

In other words, ground rule r_8 is new, r_b, r_7 and r_6 remain, and r_5 will be deleted. ∎

We see that a new tick (in Example 47 an increase in time) is reflected in the incremental encoding by (i) generating a new template rule, (ii) grounding it, and (iii) removing expired rules. The latter can be done efficiently since it depends on syntactic information from the rules themselves. In the case of the example, ground rules stemming from template $\omega(X) \leftarrow a_{@}(X, N)$ are removed after three time points due to the window length 2 of the time-based window.

Exploiting Truth Maintenance

With respect to model maintenance, a benefit of this incremental program update arises if we exploit the part of the program that does not change.

Example 48 (cont'd) Analyzing the incremental grounding at time 7, we observe that $b(x)$ is in the answer set due to the following rules (the last one being the fact from the data stream):

$$\begin{aligned} b(x) &\leftarrow \omega(x) \\ \omega(x) &\leftarrow a_{@}(x, 7) \\ a_{@}(x, 7) &\leftarrow \end{aligned}$$

We note that for time points 8 and 9 atom $b(x)$ is still concluded due to the same rules. In fact, the answer set $\{a_{@}(x, 7), \omega(x), b(x)\}$ as obtained for the incremental encodings at time $t = 7, 8, 9$ is always based on these three rules; the other rules are irrelevant (i.e., their body is false):

$$\begin{array}{lll} t = 7: & t = 8: & t = 9: \\ b(x) \leftarrow \omega(x) & b(x) \leftarrow \omega(x) & b(x) \leftarrow \omega(x) \\ \omega(x) \leftarrow a_{@}(x, 7) & \omega(x) \leftarrow a_{@}(x, 8) & \omega(x) \leftarrow a_{@}(x, 9) \\ \omega(x) \leftarrow a_{@}(x, 6) & \omega(x) \leftarrow a_{@}(x, 7) & \omega(x) \leftarrow a_{@}(x, 8) \\ \omega(x) \leftarrow a_{@}(x, 5) & \omega(x) \leftarrow a_{@}(x, 6) & \omega(x) \leftarrow a_{@}(x, 7) \\ a_{@}(x, 7) \leftarrow & a_{@}(x, 7) \leftarrow & a_{@}(x, 7) \leftarrow \end{array}$$

In principle, the model can thus be retained from time 7 to time 9. Moreover, even if we had an appearance of $a(x)$ at time 8 or 9 (corresponding to additional input atoms

$a_@(x, 8)$ and $a_@(x, 9)$, respectively), we could still maintain the truth of $b(x)$ until time 9 due to the rules displayed in black. ∎

Our considerations so far reduce plain LARS reasoning to ASP based on an encoding which can be updated incrementally: when new data is streaming in or time passes by, we add some new rules and remove expired rules. We are thus interested in a model maintenance technique that is able to update models (answer sets) based on updating the program.

In fact, Doyle's Justification-based Truth Maintenance System (JTMS) [Doy79] serves this purpose (with restrictions that will be discussed). The core of this system deals with updating justifications of propositions; these justifications syntactically correspond to rules of normal logic programs. Moreover, the semantics of JTMS is defined in terms of so-called *admissible models*, which correspond to answer sets (under some restrictions).

Overall, we thus obtain recipes for two *reasoning strategies* for plain LARS programs.

1. Repeated one-shot solving based on a static encoding to ASP where the query time t is specified by $now(t)$.

2. Incremental model update based on the dynamic encoding to ASP, where after every tick, the previous model is updated by JTMS due to the addition of new rules and the removal of expired ones.

These two reasoning strategies are implemented in the Ticker engine, which we shall present in the next chapter.

The static encoding is explained in detail in Section 6.3, followed by the incremental encoding in Section 6.4. Before that, we review and formalize the core JTMS and extend it for rule removal, which has not been considered in the original presentation. Notably, our technique does not depend on JTMS as such. Any mechanism that is able to maintain an answer set of a normal logic program in light of new and deleted rules qualifies as a submodule in our conceptual architecture and its implementation.

6.2 Formalizing Justification-based Truth Maintenance Systems (JTMS)

Justification-based Truth Maintenance Systems (JTMS) trace back to Doyle's seminal paper [Doy79] which introduced techniques for maintaining consistent beliefs and their well-foundedness based on justifications. The central concern of JTMS is the incremental model update due to new information which may lead to the retraction of previous conclusions, i.e., nonmonotonic reasoning.

The motivation for this review is that truth maintenance is a classic technique in the area of Knowledge Representation and Reasoning (KR&R); yet a clear, modular formalization of Doyle's informal algorithm description seems to be lacking, as well an implementation in a state-of-the-art programming language.

Doyle's original algorithm is more involved and allows for modelling constraints by means of contradiction nodes. However, the backtracking procedure to resolve them can introduce auxiliary rules which may then lead to models that do not reflect the semantics of the original program. Since our specific interest here is the relationship to Answer Set Programming (ASP), we focus on the core of JTMS that employs no notion of constraints and thus always computes an admissible model, resp. answer set. (For introductions to ASP we refer to [BET11, EIK09].) Throughout, we often depart from (resp. add to) Doyle's original terminology. Used terminology either stems from [BK14], from logic programming, or was chosen to reflect the conceptual meaning (e.g. algorithm and function names).

6.2.1 Truth Maintenance Networks

We start by introducing key notions similarly as in [BK14]. A *truth maintenance network* (TMN) $\mathcal{T}$ is a pair $(N, \mathcal{J})$, where N is a set of *nodes* and $\mathcal{J}$ is a set of *justifications*, i.e., expressions of form

$$J = \langle I | O \to c \rangle \,,$$

where $I, O \subseteq N$ and $c \in N$. We call I the *in-list*, O the *out-list* and c the *consequent* of J. Nodes may be viewed as proposition. The intention of a justification is that the consequent c holds iff all nodes in the in-list hold and no node in the out-list holds. What holds is determined by a *model* $M \subseteq N$, i.e., a subset of the network's nodes. Then,

$$\mathcal{J}^M = \{\langle I | O \to c \rangle \mid I \subseteq M \text{ and } O \cap M = \emptyset\} \,,$$

is the set of justifications that are *valid in* M. We say a model M is

(i) *founded*, if there exists a total order $n_1 < \cdots < n_k$ of all elements in M s.t. each $n_j \in M$ has a *supporting justification*, i.e., some $\langle I | O \to n_j \rangle \in \mathcal{J}^M$ such that $I \subseteq \{n_1, \ldots, n_{j-1}\}$;

(ii) *closed*, if $n \in M$ for all $\langle I | O \to n \rangle \in \mathcal{J}^M$; and

(iii) *admissible*, if it is founded and closed.

Example 49 As a first example, we illustrate an application of modus ponens. Consider the truth maintenance network $\mathcal{T} = (N, \mathcal{J})$, where $N = \{x, y\}$ and $\mathcal{J}$ consists of the following two justifications.

$$\begin{array}{ll} (J_1) & \langle \emptyset | \emptyset \to x \rangle \\ (J_2) & \langle \{x\} | \emptyset \to y \rangle \end{array}$$

Justification J_1 is called a *premise* (or *fact*), since both I and O are empty. A premise is valid in every model. Consequently, every model not containing its consequent is not closed. Justification J_2 reflects the material implication $x \supset y$. Assuming we have x in the model, J_2 is valid, so y must be concluded. Thus, we get the admissible model $M = \{x, y\}$. ∎

In general, admissible models are not unique, as the next example shows.

Example 50 Consider the network $\mathcal{T} = (N, \mathcal{J})$, where $N = \{a, b, c\}$ and $\mathcal{J}$ consists of the following three justifications (we omit set braces for I and O):

$$\begin{array}{ll} (J_1) & \langle b|\emptyset \to a\rangle \\ (J_2) & \langle \emptyset|c \to b\rangle \\ (J_3) & \langle \emptyset|a \to c\rangle \end{array}$$

First consider the empty model. We observe that $\mathcal{J}^{\emptyset} = \{J_2, J_3\}$, since in-lists of these justifications are empty. Consequently, this model is not closed, since J_2 would require b and J_3 would require c to be concluded. Thus, consider next the model $\{b, c\}$. In this case, the valid justifications are $\mathcal{J}^{\{b,c\}} = \{J_1, J_3\}$. The consequent of justification J_1, node a, is not included in the model, so it is also not closed. Hence, consider model $\{a, b, c\}$, for which we get $\mathcal{J}^{\{a,b,c\}} = \{J_1\}$. In other words, this model is not founded: there are no valid justifications for nodes b and c, and J_1 is also not a supporting justification for a.

By removing c from the model, we get the admissible model $M_1 = \{a, b\}$, for which both J_1 and J_2 are valid. The absence of c suffices to conclude b via J_2, based on which a is founded via J_1. Notably, the network has another admissible model: $M_2 = \{c\}$. From the absence of node a alone, c is concluded by J_3, which is the only valid justification in M_2. ∎

Logic Programming Perspective

Truth maintenance networks resemble normal logic programs in the following way. Let $J = \langle I|O \to h\rangle$ be a justification such that $I = \{i_1, \ldots, i_n\}$ and $O = \{o_1, \ldots, o_m\}$, and let

$$r_J \;=\; h \leftarrow i_1, \ldots, i_n, \mathrm{not}\, o_1, \ldots, \mathrm{not}\, o_m \tag{6.1}$$

be the corresponding rule. Moreover, let $P_{\mathcal{T}} = \{r_J \mid J \in \mathcal{J}\}$ be the logic program obtained by this translation. The following theorem by Elkan [Elk90] allows for identifying admissible models with answer sets.

Theorem 25 (cf. [Elk90]) *Let $\mathcal{T} = (N, \mathcal{J})$ be a TMN and $M \subseteq N$. Then, (i) M is an admissible model of $\mathcal{T}$ iff it is an answer set of $P_{\mathcal{T}}$. (ii) Deciding whether $\mathcal{T}$ has an admissible model is NP-complete.*

Notably, the maintenance algorithm can ensure admissibility of computed models only in the absence of odd loops (noted in [Elk90]), and in particular, constraints. We will discuss this in more detail below.

Elkan points out that also incremental reasoning is NP-complete, i.e., given an admissible model M for P, deciding for a rule r whether $P \cup \{r\}$ has an admissible model.

Example 51 (cont'd) Consider again the network $\mathcal{T} = (N, \mathcal{J})$ of Example 50. We obtain the following translated program $P_{\mathcal{T}}$:

$$\begin{array}{lll} (r_1) & a & \leftarrow b \\ (r_2) & b & \leftarrow \text{not } c \\ (r_3) & c & \leftarrow \text{not } a \end{array}$$

This program has two answer sets, $\{a, b\}$ and $\{c\}$, corresponding to the admissible models of $\mathcal{T}$. ∎

In the sequel, we discuss truth maintenance techniques in terms of logic programs:

- a program P (i.e. a set of rules) replaces justifications, and
- atoms A_P occurring in P replace nodes.

For a rule of form (6.1), with atoms $\{h, i_1, \ldots, i_n, o_1, \ldots, o_m\}$, $H(r) = h$ is called the *head*, $B^+(r) = \{i_1, \ldots, i_n\}$ is the *positive body*, $B^-(r) = \{o_1, \ldots, o_m\}$ is the *negative body*, and

$$B(r) = B^+(r) \cup B^-(r)$$

is called the *body* of r. For a given rule r, we also write H, B, B^+, and B^-, respectively. We may also denote a rule by $h \leftarrow B$, etc.

JTMS Data Structures

By *JTMS* we will refer to the following data structures based on a program P.

Labels. Each atom $a \in A_P$ is assigned a unique *label*. A *model* corresponds to the set of atoms with label *in*, all others are (labeled with) *out*. During the algorithm, a third label *unknown* is assigned to atoms that are not yet determined.

Justifications, consequences. The *justifications* $J(a) = \{r \in P \mid H(r) = a\}$ of an atom a are the rules with head a. *Consequences* of an atom b are heads of rules where b appears in the body, i.e.,

$$cons(b) = \{H(r) \mid r \in P, b \in B(r)\}.$$

Rule validity. Due to a given labeling *label*, a rule r is

- *valid*, if the body holds, i.e., if $label(a) = in$ for all $a \in B^+$ and $label(a) = out$ for all $a \in B^-$;
- *invalid* if some $a \in B^+$ is *out* or some $a \in B^-$ is *in*; we will call any such atom a *spoiler* for r;
- *posValid* if all atoms in B^+ are *in* and no atom in B^- is *in*, however, B^- may contain *unknown* atoms.

Note that every *valid* rule is also *posValid*.

Support. The *support* of an atom a specifies the reason why the current label is assigned.

- If $\mathit{label}(a) = \mathit{in}$, then $\mathit{supp}(a)$ is the body of some *supporting justification*, denoted by $\mathit{suppJ}(a)$, i.e., a valid rule $r \in P$ with head a. If a appears as fact, we set $\mathit{supp}(a) = \emptyset$ since it then needs no further support.
- If $\mathit{label}(a) = \mathit{out}$, then $\mathit{supp}(a)$ contains a spoiler for each justification. Notably, the algorithm might fail to find a spoiler for a given rule. Then it can also use an *unknown* atom a (from B^+) assuming that a will be labeled *out* later. We will explore this below. If a appears only in rule bodies, then we again set $\mathit{supp}(a) = \emptyset$.
- If $\mathit{label}(a) = \mathit{unknown}$, then the conceptual intuition of support does not apply; we technically set $\mathit{supp}(a) = \emptyset$.

Repercussions. If atom a supports rule head h, a label change for a might entail one for h. Thus, h is said to be *affected* by a. Formally,

$$\mathit{affected}(a) = \{h \in A_P \mid a \in \mathit{supp}(h)\}.$$

The *repercussions* denote the transitive closure of this relation, i.e., the set of atoms which might change their label directly or indirectly due the label change of a given atom.

6.2.2 The Truth Maintenance Algorithm

Algorithm 6.1 presents the outline of Doyle's procedure for adding to a program P with model M a rule r. The goal is to update the data structures such that the model M' obtained by atoms with label *in* is an answer set of $P \cup \{r\}$.

Example 52 Consider the following rules.

$$r_1 = a \leftarrow b. \quad r_2 = b \leftarrow \text{not } c. \quad r_3 = a \leftarrow d. \quad r_4 = d \leftarrow c.$$
$$r_5 = c \leftarrow d. \quad r_6 = c \leftarrow \text{not } e. \quad r_7 = e \leftarrow .$$

Algorithm 6.1. We start with $\mathit{add}(r_1)$. For both atoms $x \in \{a, b\}$, the call $\mathit{register}(r_1)$ (Algorithm 6.2) assigns $\mathit{label}(x) := \mathit{out}$, $\mathit{supp}(x) := \emptyset$ and $\mathit{suppJ}(x) := \mathit{nil}$. Moreover, $\mathit{cons}(b) = \{a\}$. Head a is not *in* but rule r_1 is invalid, so we reassign $\mathit{supp}(a) := \{b\}$ in Line 4 of Algorithm 6.1 and halt with model $\emptyset$, since no atom has label *in*. Next, we call $\mathit{add}(r_2)$, where $r_2 = b \leftarrow \text{not } c$. Rule head b is not *in* but r_2 is valid. Due to r_1, a is affected by b, so *update* (Algorithm 6.4) is called for $A = \{a, b\}$. ∎

Algorithm 6.5. After *setUnknown* in Algorithm 6.4, we try to deterministically assign each atom label *in* (resp. *out*) due to a valid justification (resp. by invalidity of all justifications).

Algorithm 6.1: JTMS Algorithm: *add*(r: *rule*)

Input: A rule r with head h

```
1 register(r)
2 if label(h) = in: return
3 if invalid(r):
4 |   supp(h) := supp(h) ∪ {spoiler(r)}
5 |   return
6 A := repercussions(h) ∪ {h}
7 update(A)
```

Algorithm 6.2: *register*(r: *rule*)

Input: A rule $r \notin P$ with atoms A_r, head h, and body atoms B

```
1 foreach a ∈ A_r \ A_P    //new atoms
2 |   label(a) := out
3 |   cons(a) := ∅         //consequences: heads of rules with a in the body
4 |   supp(a) := ∅         //sufficient atoms supporting the current label of a
5 |   suppJ(a) := nil      //in case a is in, a valid rule with head a
6 foreach b ∈ B
7 |   cons(b) := cons(b) ∪ {h}
8 P := P ∪ {r}
```

Algorithm 6.3: *spoiler*(r: *rule*)

Input: A rule $r \in P$ with atoms A_r, pos./neg. body B^+/B^-
Output: If r is invalid, an atom $a \in A_r$ supporting the invalidity, else *nil*

```
1 if randomBoolean()
2 |   if ∃ a ∈ B+: label(a) = out  return a
3 |   else if ∃ a ∈ B−: label(a) = in  return a
4 |   else  return nil
5 else
6 |   if ∃ a ∈ B−: label(a) = in  return a
7 |   else if ∃ a ∈ B+: label(a) = out  return a
8 |   else  return nil
```

Example 53 (cont'd) Assume *findLabel* is first called for a. The justification set is $J(a) = \{r_1\}$, and r_1 is neither valid nor invalid. For *findLabel*(b), we have $J(b) = \{r_2\}$ and as $label(c) = out$, r_2 is valid and we call $setIn(r_2)$. This assigns $label(b) := in$, $supp(b) := \{c\}$ and $suppJ(b) := r_2$. As *labelSet* is true, $cons(b) = \{a\}$ and a is *unknown*, we recursively call *findLabel*(a) for propagation; now r_1 is valid. In $setIn(r_1)$ we set $label(a) := in$, $supp(a) := \{b\}$ and $suppJ(a) := r_1$. We have the model $\{a, b\}$ which stays after adding rules r_3, r_4 and r_5; adding r_6 leads to $\{a, c, d\}$. We now call $add(r_7)$, where

Algorithm 6.4: *update*(A: *set of atoms*)

1 **foreach** $a \in A$: *setUnknown*(a)
2 **foreach** $a \in A$: *findLabel*(a) //try to determine first
3 **foreach** $a \in A$: *chooseLabel*(a) //else make choices

$r_7 = e \leftarrow$. After *findLabel*, only e is assigned (*in*), and a, b, c, d remain *unknown*. ∎

Algorithm 6.6. Procedure *chooseLabel* will similarly assign *label*(x) := *out* if all justifications are not *posValid*. Otherwise, if a *posValid* justification is found for atom x, we check whether assigning *label*(x) := *in* is safe by ensuring that nothing is affected by x so far. This may happen only if x is already used as a spoiler and thus assumed to eventually be labeled *out*. Then, Lines 14-16 reset x and affected atoms to *unknown* and recursively calls the subprocedure.

Example 54 (cont'd) We call *chooseLabel*(d) and $r_4 \in J(d)$ is not *posValid* since c is *unknown*. We call *setOut*(d) and a, c are *unknown* consequences of d; *chooseLabel*(a) leads to *setOut*(a) and *chooseLabel*(c) leads to *setOut*(c). Now b is an *unknown* consequence of c and *chooseLabel*(b) finds *posValid*(r_2). However, the previous call *setOut*(a) set *supp*(a) := $\{b, d\}$, thus *affected*(b) = $\{a\} \neq \emptyset$. Hence, we call *setUnknown* for b, a. We enter *chooseLabel*(b), then *setIn*(b); finally *chooseLabel*(a), then *setIn*(a) and get model $\{a, b, e\}$. ∎

6.2.3 Extending JTMS: Removing Rules

Doyle provided no explicit means to remove rules. However, we can simulate removal: we can store a rule $r = h \leftarrow B$ internally as $h \leftarrow B, \text{not } rm_r$, where rm_r is a fresh atom. Then, removing r amounts to adding the fact $rm_r \leftarrow$. However, reactivating the rule can then only be done by adding a modified copy of form $h \leftarrow B, \text{not } rm'_r$, where rm'_r is another fresh atom. Consequently, this approach will lead to an inflation of the knowledge base.

A better solution is presented in Algorithm 6.7, which allows for removing a rule in analogy to Algorithm 6.1. First, we potentially remove in *deregister* (Algorithm 6.8) obsolete entries in the TMS data structures. This involves removal of atoms that are no longer in the program and updating consequences of the remaining ones. Procedure *remove* then halts without updates if rule head h no longer occurs in the program, h has label *out*, or if r was not the supporting justification of h.

Example 55 (cont'd) Suppose we delete $r_3 = a \leftarrow d$, i.e., we call *remove*(r_3). Both a and d remain in the program, so in *deregister*(r_3) we only remove a from *cons*(d). We then halt in Line 3 (of Algorithm 6.7) since *suppJ*(a) = r_1. Consequently, the model $\{a, b, e\}$ is maintained. ∎

If no exit criterion applies, we determine repercussions as in Algorithm 6.1 (*add*) and call the same update procedure.

Algorithm 6.5: *findLabel*(a: *atom*)

```
1  if label(a) ≠ unknown: return
2  labelSet := false
3  J(a) := {r ∈ P | H(r) = a}    //H(r) is the head of r
4  if ∃r ∈ J(a): valid(r)
5  |   setIn(r)
6  |   labelSet := true
7  else if ∀r ∈ J(a): invalid(r)
8  |   setOut(a)
9  |   labelSet := true
10 if labelSet
11 |   foreach u ∈ unknownCons(a): findLabel(u)
   ---------------------------------------------------------------
12 defn unknownCons(a) = {c ∈ cons(a) | label(c) = unknown}
13 def setIn(r):
14 |   h := H(r)
15 |   label(h) := in
16 |   supp(h) := B(r)
17 |   suppJ(h) := r
18 def setOut(a):
19 |   label(a) := out
20 |   supp(a) := { spoiler(r) | r ∈ P and H(r) = a }
21 |   suppJ(a) := nil
22 def setUnknown(a):
23 |   label(a) := unknown
24 |   supp(a) := ∅
25 |   suppJ(a) := nil
```

6.2.4 Analysis of JTMS

Doyle's algorithm faces some problem on its own, and some with respect to the (partial) correspondence with Answer Set Programming.

Algorithm Design

In [Doy79], Doyle described his algorithm only informally. Our formal presentation uses new vocabulary and is modularized into subprocedures. A design issue is the 2-step approach to deterministic and non-deterministic label assignment in Algorithm 6.4. Doyle points out that *findLabel* (in our terms) is subsumed by *chooseLabel* and only introduced

Algorithm 6.6: *chooseLabel*(a: *atom*)

```
1  if label(a) ≠ unknown: return
2  labelSet := false
3  J(a) := {r ∈ P | H(r) = a}
4  if ∃ r ∈ J(a): posValid(r)
5  |  if affected(a) = ∅
6  |  |  setIn(r)
7  |  |  labelSet := true
8  else
9  |  setOut(a)      //for support: view unknown as out
10 |  labelSet := true
11 if labelSet
12 |  foreach u ∈ unknownCons(a): chooseLabel(u)
13 else
14 |  C := affected(a) ∪ {a}
15 |  foreach c ∈ C: setUnknown(c)
16 |  foreach c ∈ C: chooseLabel(c)
```

Algorithm 6.7: JTMS Extension: *remove*(r: *rule with head* h)

```
1  deregister(r)
2  if h ∉ A_{P\{r}} or label(h) = out or suppJ(h) ≠ r
3  |  return
4  A := repercussions(h) ∪ {h}
5  update(A)
```

for efficiency. However, from this perspective, it is unclear why in Algorithm 6.6, Line 4, one would not first look for *valid* rules, then for *posValid* ones, etc. To first compute the deterministic effects after a *single* choice is natural, as this reduces the probability that retraction steps of Lines 14-16 will be needed.

Example 56 In Ex. 54, assume a deterministic step after the initial call *chooseLabel*(d), with the same order as before: By contrast, a is now examined in *findLabel* and remains *unknown*. Next, *findLabel*(c) will call *setOut*(c), leading label *in* for b and then a due to recursive calls of *findLabel*. ∎

Odd Loops

As pointed out in [McD91], the update may lead to inadmissible models. JTMS cannot handle *odd loops*, i.e., an odd number of negations due to which an atom's support (necessarily) depends on itself. The minimal case is shown in the next example.

Algorithm 6.8: *deregister*(r: *rule*)

Input: A rule r with atoms A_r, head h, and body atoms $B(r)$

```
1 if r ∉ P return
2 P := P \ {r}
3 foreach a ∈ A_r \ A_P    //deprecated rule atoms
4     remove key a in maps label, cons, supp, suppJ
5 foreach b ∈ B(r) ∩ A_P    //body atoms that are still in use
6     if {r' ∈ P | H(r') = h and b ∈ B(r')} = ∅
7         cons(b) := cons(b) \ {h}
```

Example 57 Consider the rule $r = x \leftarrow \text{not}\, x$, where x is a fresh atom; r is valid and *findLabel*(x) calls *setIn*(r_1) which assigns *supp*(x) := $\{x\}$, i.e., x supports itself. We get the inadmissible model $\{x\}$ instead of none. ∎

Note that for odd loops as in Example 57, simple ad-hoc tests (e.g. for self-support) would suffice. In general, however, odd loops cannot be solved that easily and require a different kind of maintenance technique.

Odd loops may not only result in inadmissible models, they may also cause non-termination.

Example 58 Consider an empty program, then *add*(r_1) for $r_1 = a \leftarrow \text{not}\, b$ and *add*(r_2) for $r_2 = b \leftarrow a$, which is *valid*. Since b is *out*, we call *update*($\{a, b\}$); atom a will be assigned *in* since r_1 is *posValid*; *supp*(a) := $\{b\}$. For atom b we have *posValid*(r_2) but *affected*(b) $\neq \emptyset$. Thus, a and b are reset to *unknown*, which also occurs when starting with b. ∎

Constraints and Inconsistency

More abstractly, JTMS cannot deal with inconsistency. In particular, a constraint $\bot \leftarrow i_1, \ldots, i_n, \text{not}\, o_1, \ldots, \text{not}\, o_m$ cannot be modelled by

$$x \leftarrow i_1, \ldots, i_n, \text{not}\, o_1, \ldots, \text{not}\, o_m, \text{not}\, x\,, \tag{6.2}$$

as usual as this rule introduces a (minimal) odd loop. Doyle presented a workaround: when a special *contradiction node (atom)* c is derived, a dependency-directed backtracking procedure will add new rules R to the program P to prevent the derivation of c. However, the admissible models of $P \cup R$ are not necessarily admissible for P. We thus skip a discussion of this procedure.

Non-termination

Another issue concerns the unsystematic backtracking in Algorithm 6.6, where the recursive call in Line 16 may lead to an infinite loop.

Example 59 Consider again the program from Example 52 and let the truth maintenance network be in the same state (in terms of contents of all data structures) as in Example 54 where b is the last remaining unknown atom. Let us call this specific state U_b. There, we found $\mathit{posValid}(r_2)$ (Line 4), but $\mathit{affected}(b) = \{a\}$ (Line 5), hence we reset a and b to unknown (Line 15). Let this state be denoted by U_{ab}. In Example 54 we assumed that in Line 16 we will first pick atom b. Let us now assume in state U_{ab}, the implementation first picks a, i.e., the call $\mathit{chooseLabel}(a)$. Now, neither of the justifications $r_1 = a \leftarrow b$ nor $r_3 = a \leftarrow d$ are *posValid* ($\mathit{label}(b) = \mathit{unknown}$, $\mathit{label}(d) = \mathit{out}$). Consequently, a is set to *out* and we again enter state U_b, leading to state U_{ab}, and so forth. ∎

In practice, such infinite loops can be avoided by shuffling the order of atoms for which the recursive call to *chooseLabel* is made.

Example 60 (cont'd) Assume set C of Line 14 is represented as list and randomly shuffled after initialization. Then, eventually we will first call $\mathit{chooseLabel}(b)$ in state U_{ab} as in Example 54, where the algorithm terminates with the correct model $\{a, b, e\}$. ∎

6.3 Static Encoding: Plain LARS to ASP

In this section we will first give a translation of LARS programs P to an ASP program $\hat{P}$. Toward incremental evaluation of P, we will then show (in Section 6.4) how $\hat{P}$ can be adjusted to accommodate new input signals and account for expiring information as specified by window operators.

6.3.1 Tick Streams

Central to our approach of fully incremental reasoning is the notion of a tick.

Definition 32 (Tick) *A pair $k = (t, c)$, where $t, c \in \mathbb{N}$, is called a* tick*; t is the* (tick) time *and c the* (tick) count*. Moreover, $(t+1, c)$ is called the* time increment *and $(t, c+1)$ the* count increment *of k. A sequence $K = \langle k_1, \ldots, k_m \rangle$, $m \geq 1$, of ticks is a* tick pattern*, if every tick k_{i+1} is either a time increment or a count increment of k_i.*

Intuitively, a tick pattern captures the incremental development of a stream in terms of time and tuple count, where at each step exactly one dimension increases by 1. For a set of ticks, at most one linear ordering yields a tick pattern. Thus, we can view a tick pattern K also as a set.

Definition 33 (Tick Stream) *A* tick stream *is a pair $\dot{S} = (K, v)$ of a tick pattern K and an* evaluation function *v such that $v(k_{i+1}) = \{a\}$ for some $a \in \mathcal{A}$, if k_{i+1} is a count increment of k_i, else $v(k_{i+1}) = \emptyset$.*

We say that a tick stream $\dot{S} = (K, v)$ with $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$ is *at tick* (t_m, c_m). By default, we assume $(t_1, c_1) = (0, 0)$ and thus c_m is the total number of atoms. We also write $v(t, c)$ instead of $v((t, c))$. Naturally, a *(tick) substream* $\dot{S}' \subseteq \dot{S}$ is a tick stream

(K', v'), where K' is a subsequence of K and v' is the restriction $v|_{K'}$ of v to K', i.e., $v'(t,c) = v(t,c)$ if $(t,c) \in K'$, else $v'(t,c) = \emptyset$.

Example 61 The sequence $K = \langle (0,0), (1,0), (2,0), (3,0), (3,1), (3,2), (4,2) \rangle$ is a "canonical" tick pattern starting at $(0,0)$, where $(3,1)$ and $(3,2)$ are the only count increments. Employing an evaluation $v(3,1) = \{a\}$ and $v(3,2) = \{b\}$, we get a tick stream $\dot{S} = (K, v)$ which is at tick $(4,2)$. ∎

Definition 34 (Ordering) *Let $\dot{S} = (K, v)$ be a tick stream and let $S = (T, \upsilon)$ be a stream such that $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$, $T = [t_1, t_m]$ and $\upsilon(t) = \bigcup \{v(t,c) \mid (t,c) \in K\}$ for all $t \in T$. Then, we say $\dot{S}$ is an* ordering *of S, and S* underlies *$\dot{S}$.*

Note that in general, a stream S has multiple orderings, but every tick stream $\dot{S}$ has a unique underlying stream. All orderings of a stream have the same tick pattern.

Example 62 (cont'd) Stream $S = ([0,4], \upsilon)$, where $\upsilon = \{3 \mapsto \{a, b\}\}$, is the underlying stream of $\dot{S}$ of Example 61. A further ordering of S is $\dot{S}' = (K, v')$, where $v' = \{(3,1) \mapsto \{b\}, (3,2) \mapsto \{a\}\}$. ∎

Recall the nature of sliding windows, which are purely incremental: a (standard) time-based window of size n always selects the last n time points (and all atoms there), and a tuple-based window of size n always selects the minimal recent window that contains the last n atoms. These definitions can essentially be carried over for tick streams. There are two formal differences. First, ticks replace time points as positions in a stream, and thus as second argument of the window functions. Second, tuple-based windows are now always unique, which also simplifies their definition.

Definition 35 (Sliding Windows over Tick Streams) *Consider a tick stream $\dot{S} = (K, v)$ such that $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$ and $(t,c) \in K$. Then, the*

(i) time window function τ_n, $n \geq 0$, *is defined by $\tau_n(\dot{S}, (t,c)) = (K', v|_{K'})$, where $K' = \{(t', c') \in K \mid \max\{t_1, t - n\} \leq t' \leq t\}$, and the*

(ii) tuple window function $\#_n$, $n \geq 1$, *is defined by $\#_n(\dot{S}, (t,c)) = (K', v|_{K'})$, where $K' = \{(t', c') \in K \mid \max\{c_1, c - n + 1\} \leq c' \leq c\}$.*

Note the analogy that arises due to the generalization to ticks. Time windows are determined by taking the most recent ticks such that the last n time points are covered, and tuple windows cover the last n atoms. Time windows are allowed to select only the current time point ($n \geq 0$), whereas tuple windows have to select at least one atom ($n \geq 1$). More precisely, our definition states that a time window of size n selects the current time point plus n time points of the past. This explains why there is no $+1$ in $\max\{t_1, t - n\}$ as in the case for the tuple window.

Restricting tuple windows to extensional atoms. For practical reasons, we consider tuple windows only on extensional data. Their intended use is counting input data, not inferences. Using them on intensional data is conceptually questionable.

Example 63 Consider the tick stream $D = (\langle(0,0),(0,1),(1,0)\rangle, \{(0,1) \mapsto \{a\}\})$ and the rule $r = b \leftarrow \boxplus^{\#1}\Diamond a$. The rule fires at tick $(1,0)$ and we must infer b for $(1,0)$. However, in this interpretation, $\boxplus^{\#1}\Diamond a$ does not hold anymore, if we also take into account the inference b, which then is the latest atom to be selected by $\boxplus^{\#1}$. Thus, the interpretation would not be minimal. Moreover, further inferences would not be founded. Hence, program $\{r\}$ has no answer stream. ∎

In contrast to tuple windows, time windows are useful and allowed on arbitrary data, as long as no cyclic positive dependencies through time window atoms $\boxplus^n\Box a$ occur. In Example 31 (page 64) we gave a use case that applied a time window on inferred data: the first rule associated atom *high* with every recent time point where a value exceeded a threshold. The time window of the second rule then accessed this derived information.

We consider windows over tick streams also implicitly at the end of the timeline and thus might drop the tick as argument in the respective functions.

Lemma 9 *If stream S underlies tick stream $\dot{S}$, then $\tau_n(S)$ underlies $\tau_n(\dot{S})$.*

Proof. Let $S = (T, \upsilon)$ be a stream that underlies tick stream $\dot{S} = (K, \upsilon)$, such that $K = \langle(t_1, c_1), \ldots, (t_m, c_m)\rangle$. By definition, $T = [t_1, t_m]$ and $\upsilon(t) = \bigcup\{v(t,c) \mid (t,c) \in K\}$ for all $t \in T$. We recall that $\tau_n(S)$ (resp. $\tau_n(\dot{S})$) abbreviates $\tau_n(S, t_m)$ (resp. $\tau_n(\dot{S}, (t_m, c_m))$. Thus, by definition, $\tau_n(\dot{S}) = (K', v|_{K'})$, where $K' = \{(t', c') \in K \mid \max\{t_1, t-n\} \leq t' \leq t\}$, and $\tau_n(S) = (T', v|_{T'})$, where $T' = [t', t_m]$ and $t' = \max\{t_1, t-n\}$. We observe that t' is the minimal time point selected also in K', i.e., $K' = \langle(t_k, c_k), \ldots, (t_m, c_m)\rangle$ implies $t_k = t'$. It remains to show that $(\upsilon|_{T'})(t) = \bigcup\{(v|_{K'})(t,c) \mid (t,c) \in K'\}$ for all $t \in T'$. This is seen from the fact that neither $\tau_n(S)$ nor $\tau_n(\dot{S})$ drops any data within T'. We conclude that $\tau_n(S)$ underlies $\tau_n(\dot{S})$. □

Example 64 (cont'd) Given $\dot{S}$ and S from Example 62, we have

$$\tau_1(\dot{S}, 4) = (\langle(3,0),(3,1),(3,2),(4,2)\rangle, \upsilon)$$

with underlying stream $\tau_1(S, 4) = ([3,4], \upsilon)$. ∎

Correspondence for tuple windows is more subtle due to the different options to realize them. That is to say, if a sliding tuple-based window (for a LARS stream) of size n will select the interval $T = [t_1, t]$ and there are more than n atoms in T, a choice needs to be made which atoms to drop at time t_1. No such ambiguity arises for tick streams.

Lemma 10 *Let stream S underlie tick stream $\dot{S}$ and assume the tuple window $\#_n(S)$ is based on the order in which atoms appeared in S. Then, $\#_n(S)$ underlies $\#_n(\dot{S})$.*

Proof. The argument is similar as for Lemma 9. The central observation is that a tick stream provides a more fine-grained control over the information available in streams by introducing an order on tuples in addition to the temporal order. Each time point in a stream is assigned a set of atoms, whereas each tick in a tick stream is assigned at most

one atom. The tuple-based window function $\#_n$ always counts atoms backwards (from right end to left) and then selects the timeline $[t_1, t]$ with the latest possible left time point t_1 required to capture n atoms. While for tick streams, the order is unique, but multiple options exist for streams in general. If the tuple window $\#_n(S)$ is based on the order in which atoms appeared in S, then it selects the same atoms as $\#_n(\dot{S})$, and thus the same timeline. Consequently, $\#_n(S)$ underlies $\#_n(\dot{S})$. □

Example 65 (cont'd) Stream S has two tuple windows of size 1: $S_a = ([3,4], \{3 \mapsto \{a\}\})$ and $S_b = ([3,4], \{3 \mapsto \{b\}\})$. We observe that S_b underlies $\#_1(\dot{S})$, given by $(\langle(3,2),(4,2)\rangle, (3,2) \mapsto \{b\})$. ∎

We can represent a stream $S = (T, \upsilon)$ alternatively by T and a set of *time-pinned* atoms, i.e., the set

$$\{a_@(\mathbf{x}, t) \mid a(\mathbf{x}) \in \upsilon(t), t \in T\}.$$

Similarly, we can represent a tick stream $\dot{S} = (K, \upsilon)$ by *tick-pinned* atoms of the form $a_\#(\mathbf{x}, t, c)$, where c increases by 1 for every incoming signal, i.e., by the set

$$\{a_\#(\mathbf{x}, t, c) \mid a(\mathbf{x}) \in \upsilon(t, c), (t, c) \in K\}.$$

Tick-pinned atoms provide the necessary information for expressing tuple-windows, i.e., the relation between the time t an atom occurred and its relative position to other atoms regardless of time (i.e., its count c). Since tuple windows will be realized based on time-pinned atoms, we chose to use symbol # for both of them.

Example 66 (cont'd) Given extra knowledge about the current time $t = 4$, stream S is fully represented by $\{a_@(3), b_@(3)\}$, whereas tick stream $\dot{S}$ can be encoded by the set $\{a_\#(3,1), b_\#(3,2)\}$. ∎

The notions of data/interpretation stream readily carry over to their tick analogues. Moreover, we say a tick interpretation stream I is an *answer stream* of program P (for tick data stream D at t), if the underlying stream I' of I is an answer stream of P (for the underlying data stream D' at t).

6.3.2 Translation

Plain LARS programs extend normal logic programs by allowing extended atoms in rule bodies, and also @-atoms in rule heads. Thus, if we restrict α and β_i in rule syntax (4.2) (page 82) to atoms, we obtain a normal rule. This observation is used for the translation of LARS to ASP as shown in Algorithm 6.9. The encoding has to take care of two central aspects. First, each extended atoms e is encoded by an (ordinary) atom a that holds iff e holds. Second, entailment in LARS is defined with respect to some data stream D and background data B at some time t. Stream signals and background data are encoded as facts, and temporal information by adding a time argument to atoms. Example 46 already illustrated the central ideas of the encoding.

Algorithm 6.9: *LarsToAsp*(P, t)

Input: Plain LARS program P (potentially non-ground), evaluation time point t
Output: ASP encoding $\hat{P}$, i.e., a set of normal logic rules

$Q := \{a(\mathbf{X}) \leftarrow \mathit{now}(\dot{N}), a_@(\mathbf{X}, \dot{N}); a_@(\mathbf{X}, \dot{N}) \leftarrow \mathit{now}(\dot{N}), a(\mathbf{X}) \mid \text{predicate } a \text{ is in } P\}$
$R := \bigcup_{r \in P} \mathit{aspRules}(r)$
return $Q \cup R \cup \{\mathit{now}(t)\}$

---

defn $\mathit{aspRules}(r) = \{\mathit{baseRule}(r)\} \cup \bigcup_{e \in B(r)} \mathit{windowRules}(e)$

defn $\mathit{baseRule}(h \leftarrow e_1, \ldots, e_n, \text{not } e_{n+1}, \ldots, \text{not } e_m) =$
  $\mathit{atm}(h) \leftarrow \mathit{atm}(e_1), \ldots, \mathit{atm}(e_n), \text{not } \mathit{atm}(e_{n+1}), \ldots, \text{not } \mathit{atm}(e_m)$

defn $\mathit{atm}(e) =$ **match** e
  case $a(\mathbf{X}) \implies a(\mathbf{X})$
  case $@_T a(\mathbf{X}) \implies a_@(\mathbf{X}, T)$
  case $\boxplus^w @_T a(\mathbf{X}) \implies \omega_e(\mathbf{X}, T)$ //ω_e is a fresh predicate associated with e
  case $\boxplus^w \Diamond a(\mathbf{X}) \implies \omega_e(\mathbf{X})$
  case $\boxplus^w \Box a(\mathbf{X}) \implies \omega_e(\mathbf{X})$

defn $\mathit{windowRules}(e) =$ **match** e
  case $\boxplus^n @_T a(\mathbf{X}) \implies$
    $\{\omega_e(\mathbf{X}, T) \leftarrow \mathit{now}(\dot{N}), a_@(\mathbf{X}, T), T = \dot{N} - i \mid i = 0, \ldots, n\}$
  case $\boxplus^n \Diamond a(\mathbf{X}) \implies \{\omega_e(\mathbf{X}) \leftarrow \mathit{now}(\dot{N}), a_@(\mathbf{X}, T), T = \dot{N} - i \mid i = 0, \ldots, n\}$
  case $\boxplus^n \Box a(\mathbf{X}) \implies \{\omega_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } \mathit{spoil}_e(\mathbf{X})\} \cup$
    $\{\mathit{spoil}_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \mathit{now}(\dot{N}), \text{not } a_@(\mathbf{X}, T), T = \dot{N} - i \mid i = 1, \ldots, n\}$
  case $\boxplus^{\#m} @_T a(\mathbf{X}) \implies$
    $\{\omega_e(\mathbf{X}, T) \leftarrow \mathit{cnt}(\dot{C}), a_\#(\mathbf{X}, T, D), D = \dot{C} - j \mid j = 0, \ldots, m-1\}$
  case $\boxplus^{\#m} \Diamond a(\mathbf{X}) \implies$
    $\{\omega_e(\mathbf{X}) \leftarrow \mathit{cnt}(\dot{C}), a_\#(\mathbf{X}, T, D), D = \dot{C} - j \mid j = 0, \ldots, m-1\}$
  case $\boxplus^{\#m} \Box a(\mathbf{X}) \implies \{\omega_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } \mathit{spoil}_e(\mathbf{X})\} \cup$
    $\{\mathit{spoil}_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \mathit{cnt}(\dot{C}), \mathit{tick}(T, D), \dot{C} - m + 1 \leq D \leq \dot{C}, \text{not } a_@(\mathbf{X}, T)\} \cup$
    $\{\mathit{spoil}_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \mathit{cnt}(\dot{C}), \mathit{tick}(T, D), D = \dot{C} - m + 1, a_\#(\mathbf{X}, T, D'), D' < D\}$
  else $\emptyset$

The ASP encoding $\hat{P}$ for (plain) LARS program P at time t is obtained by providing the query time via fact $\mathit{now}(t)$ (where *now* is an auxiliary predicate) and two rule sets: Q serves to equate two representations for atoms that hold at t, and R contains the actual rule encodings as determined by function *aspRules*. A tick data stream D is then easily encoded by time-pinned and tick-pinned atoms. The final encoding $\hat{D}$ additionally includes an auxiliary atom $\mathit{cnt}(c)$ to explicitly represent the current tick count c, as well as representations for each tick. We will then obtain that an answer stream of P for D at t corresponds to an answer set of $\hat{P} \cup \hat{D}$. We now discuss the details.

Program Encoding $\hat{P}$: Algorithm 6.9 (*LarsToAsp*)

Given a LARS program P, its ASP encoding $\hat{P}$ at time t is computed by the function *LarsToAsp*(P, t), as defined in Algorithm 6.9. It computes two sets of rules, Q in Line 1 and R in Line 2, as follows.

Q: Identity of $@_t a(\mathbf{x})$ and $a(\mathbf{x})$ at t. In addition to the input facts modelling the stream, an answer set A of $\hat{P}$ contains two kinds of atoms: ordinary atoms of form $a(\mathbf{x})$ and time-pinned atoms of form $a_@(\mathbf{x}, t')$. (Auxiliary atoms with predicates ω_e can be considered as ordinary atoms in this regard, albeit with a special meaning.) An atom $a(\mathbf{x}) \in A$ is an alternative representation for $a_@(\mathbf{x}, t)$, which then is also included in A. That is to say, what is contained in A holds now (i.e., at t); in particular for time-pinned atoms $a_@(\mathbf{x}, t') \in A$ this means that the mapping $t' \mapsto a(\mathbf{x})$ in the corresponding answer stream holds now (i.e., more precisely, the inclusion $a(\mathbf{x}) \in \upsilon(t')$). This corresponds to the @-atom $@_{t'} a(\mathbf{x})$ which in this case holds now.

It will later turn out to be benefitial to have atoms $a(\mathbf{x}) \in \upsilon(t)$ represented both as $a(\mathbf{x})$ and $a_@(\mathbf{x}, t)$ in A. To this end, we state their equivalence by the rules Q in Line 1.

R: Rule encodings (*aspRules*). To encode a LARS rule r of form (4.2) we replace it with a normal rule due to function *baseRule*, which in turn replaces every extended atom e by an (ordinary) auxiliary atom $atm(e)$ (Lines 8-12). Accordingly, for e of form $@_T a(\mathbf{X})$, we use $a_@(\mathbf{X}, T)$ (where T and $\mathbf{X}$ can be non-ground). For a window atom e, we use a new predicate ω_e for its encoding. If e has the form $\boxplus^w \star a(\mathbf{X})$, where $\star \in \{\Diamond, \Box\}$, we use a new atom $\omega_e(\mathbf{X})$, while for e of form $\boxplus^w @_T a(\mathbf{X})$, we use $\omega_e(\mathbf{X}, T)$ with the additional time argument.

For window atoms e we need to make sure that e holds now iff $atm(e)$ is contained in a corresponding answer set of the encoding. Thus, *aspRules* adds a set of *windowRules*(e) to derive them. To this end, Lines 14-23 carry out a case distinction based on the form of window atom e. (For uniformity, all extended atoms are considered, but for atoms and @-atoms, no rules are returned.)

In case $e = \boxplus^n @_T a(\mathbf{X})$, we have to test whether $a_@(\mathbf{X}, T)$ holds for some time T within the last n time points. For $\boxplus^n \Diamond a(\mathbf{X})$, we omit T in the rule head. Dually, if $\boxplus^n \Box a(\mathbf{X})$ holds for the same substitution $\mathbf{x}$ of $\mathbf{X}$ for all previous n time points, then in particular it holds now. So we derive $\omega_e(\mathbf{x})$ by the rule in Line 16 if $a(\mathbf{x})$ holds now and there is no *spoiler* i.e., a time point among $t-1, \ldots, t-n$ where $a(\mathbf{x})$ does not hold. This is established by the rule in Line 17. (We assume the window does not exceed the timeline and thus do not check $T - i \geq 0$.) Adding $a(\mathbf{X})$ to the body ensures safety of $\mathbf{X}$ in $a_@(\mathbf{X}, T)$.

For $\boxplus^{\#m} @_T a(\mathbf{X})$, we match every atom $a(\mathbf{x})$ with the time it occurs in the window of the last m tuples. Accordingly, we track the relation between arguments $\mathbf{x}$, the time t of occurrence in the stream, and the count c. To this end, we assume any input signal $a(\mathbf{x})$ is provided as $\{a_@(\mathbf{x}, t), a_\#(\mathbf{x}, t, c)\}$. Furthermore, the rules in Line 18 employ a predicate *cnt* that specifies the current tick count (as does *now* for the time tick). Based on this, the window is created analogously to a time window but counting back $m-1$ tuples instead of n time points. The case $\boxplus^{\#m} \Diamond a(\mathbf{X})$ is again analogous, but the

variable T is not included in the head.

For $\boxplus^{\#m}\Box a(\mathbf{X})$, Line 20 is as in the time-based analogue (Line 16); $a(\mathbf{X})$ must hold now and there must not exist a spoiler. First, Line 21 ensures that $a(\mathbf{X})$ holds at every time point T in the window's range, determined by reaching back $m-1$ tick counts to count D. To do so, we add to the input stream an auxiliary atom of form $tick(t,c)$ for every tick (t,c) of the stream. Second, Line 22 accounts for the cut-off position within a time point, ensuring a is within the selected range of counts. Finally, $windowRules(e) = \emptyset$ if e is an atom or an @-atom, as they do not need extra rules for their derivation.

Example 67 Consider a stream $\dot{S}'$, which adds to $\dot{S}$ from Example 61 tick $(4,3)$ with evaluation $v(4,3) = \{a\}$. We evaluate $\boxplus^{\#2}\Box a$. The tick-pinned atoms are $a_{\#}(3,1)$, $b_{\#}(3,2)$ and $a_{\#}(4,3)$; the window selects the last two, i.e., atoms with counts $D \geq 2$. It thus covers time points 3 and 4. While atom a occurs at time 3, it is not included in the window anymore, since its count is $1 < D$. ∎

Tick Stream Encodings $\hat{D}$

Let $O = (K, v)$ be a tick stream at tick (t_m, c_m). We define its encoding $\hat{O}$ as

$$\begin{aligned}\hat{O} = {} & \{a_{@}(\mathbf{x}, t) \mid a(\mathbf{x}) \in v(t,c), (t,c) \in K\} \cup \\ & \{a_{\#}(\mathbf{x}, t, c) \mid a(\mathbf{x}) \in v(t,c), (t,c) \in K, a(\mathbf{x}) \in \mathcal{A}^{\mathcal{E}}\} \cup \\ & \{cnt(c_m)\} \cup \{tick(t,c) \mid (t,c) \in K\}.\end{aligned}$$

Note that a tick data streams D only contain extensional atoms. The presented encoding also applies to interpretation streams, where we will not include tick-pinned atoms for intensional atoms. (We will consider tuple windows only over extensional atoms, i.e., counting data and not inferences.)

We may assume that rules access background data B only by atoms (and not with @-atoms or window atoms). Viewing B as facts in the program, we skip further discussion.

Correspondence

The following correspondence result implicitly disregards auxiliary atoms in the encoding.

Theorem 26 *Let P be a plain LARS program, $D = (K, v)$ be a tick data stream at tick (t,c) and let $\hat{P} = LarsToAsp(P,t)$. Then, S is an answer stream of P for D at t iff $\hat{S}$ is an answer set of $\hat{P} \cup \hat{D}$.*

Proof. The desired correspondence is based on two translations: a LARS program P (at a time t) into a logic program $\hat{P} = LarsToAsp(P,t)$ (due to Algorithm 6.9), and the encoding of a stream S as set $\hat{S}$ of atoms. Given a fixed timeline T, we may view a stream $S = (T, v)$ as a set of pairs $\{(a(\mathbf{x}), t) \mid a(\mathbf{x}) \in v(t), t \in T\}$. This is the essence of a stream encoding $\hat{S}$ for the tick stream $\dot{S} = (K, v)$; $\hat{S}$ includes the analogous time-pinned atoms: $\{a_{@}(\mathbf{x}, t) \mid a(\mathbf{x}) \in v(t,c), (t,c) \in K\}$. With respect to the correspondence, atoms of form $a_{\#}(\mathbf{x}, t, c)$, $cnt(c)$ and $tick(t,c)$ in $\hat{S}$ can be considered auxiliary, as well as the

specific counts used in the tick pattern K to obtain time-pinned atoms $a_{@}(\mathbf{x}, t)$. Counts play a role only for the specific selection of tuple-based windows, which are assumed to reflect the order of the tick stream. Thus, we may view a stream encoding $\hat{S}$ essentially as a different representation of stream S; additional atoms can be abstracted away as they have no correspondence in the original LARS stream or program. We thus consider only the time-pinned atoms in an encoded stream to read off a LARS stream.

Thus, it remains to argue the soundness of the transformation *LarsToAsp*, which returns a program of form $Q \cup R \cup \{now(t)\}$, where $now(t)$ is auxiliary. The set Q simply identifies time-pinned atoms $a_{@}(\mathbf{X}, \dot{N})$ with $a(\mathbf{X})$ in case $\dot{N}$ is the current time point. This is the information provided by predicate *now* for which a unique atom exists. Thus, Q ensures that a time-pinned atom $a_{@}(\mathbf{x}, t)$ is available if $a(\mathbf{x}, t)$ is derived, and vice versa; Q thereby only accounts for redundant representations of atoms that currently hold.

Towards R, we get the translation by the function *aspRules* which returns a set of encoded rules for every LARS rule r. First, the *baseRule* is the corresponding ASP rule, which introduces a new symbol $atm(e)$ for every extended atom in the rule that is not an ordinary atom. In order to ensure that the base rule $\hat{r}$ fires in an interpretation just if the original rule r fires in the corresponding interpretation of program P, for each body element $atm(e)$ in $\hat{r}$ the set of rules to derive $atm(e)$ in lines (14)-(21) is provided; the correspondence between $@_T a(\mathbf{X})$ and $a_{@}(\mathbf{X}, T)$ is already given by construction. Thus, each interpretation stream $I \supseteq D$ for P has a corresponding interpretation $\hat{I}$ for $LarsToAsp(P)$ in which besides the time-pinned atoms the atoms $atm(e)$ and $spoil_e(\mathbf{X})$ occur depending on support from (i.e., firing) of the rules in (14)-(21), such that they correctly reflect the value of the window atoms e in I.

As each atom in an answer of an ordinary ASP program must derived by a rule, it is not hard to see that every answer set of $\hat{P} = LarsToAsp(P, t) \cup \hat{D}$ is of the form $\hat{I}$, where $I \supseteq D$ is an interpretation stream for D. We thus need to show the following: $I \in \mathcal{AS}(P, D, t)$ holds iff $\hat{I}$ is an answer set of $\hat{P}$. We do this for ground P (the extension to non-ground P is straightforward).

($\Rightarrow$) For the only-if direction, we show that if $I \in \mathcal{AS}(P, D, t)$, that is, I is a minimal model for the reduct $P^{M,t}$ where $M = \langle I, W, B\rangle$, then (i) $\hat{I}$ is a model of the reduct $\hat{P}^{\hat{I}}$, and (ii) no interpretation $J' \subset \hat{I}$ is a model of $\hat{P}^{\hat{I}}$. As for (i), we can concentrate by construction of $\hat{I}$ on the base rules $\hat{r} = baseRule(r)$ in $\hat{P}^{\hat{I}}$ (all other rules will be satisfied). If $\hat{I}$ satisfies $B(\hat{r})$, then by construction I satisfies $B(r)$; as I is a model of $P^{M,t}$, it follows that I satisfies $H(r)$; but then, by construction, $\hat{I}$ satisfies $H(\hat{r})$. As for (ii), we assume towards a contradiction that some $J' \subset \hat{I}$ satisfies $\hat{P}^{\hat{I}}$. We then consider the stream $J \supseteq D$ that is induced by J', and any rule r in the reduct $P^{M,t}$. If J does not satisfy $B(r)$, then J satisfies r; otherwise, if J satisfies $B(r)$, then as $\hat{r}$ is in the reduct $\hat{P}^{\hat{I}}$, we have that $\hat{I}$ falsifies each atom $atm(e)$ in $B^-(\hat{r})$, and as $J' \subset \hat{I}$, also J' falsifies each such $atm(e)$. Furthermore, as J satisfies each atom $e \in B^+(r)$, from the rules for $atm(e)$ among (14)-(21) in the reduct $\hat{P}^{\hat{I}}$ we obtain that J' satisfies each atom $atm(e)$ in $B^+(\hat{r})$. That is, J' satisfies $B(\hat{r})$. As J' satisfies $\hat{r}$, we then obtain that J' satisfies $H(\hat{r})$. The latter means that J satisfies $H(r)$, and thus J satisfies r. As r was arbitrary from

the reduct $P^{M,t}$, we obtain that $J \subset I$ is a model of $P^{M,t}$; this however contradicts that I is a minimal model of $P^{M,t}$, and thus (ii) holds.

($\Leftarrow$) For the if direction, we argue similarly. Consider an answer set $\hat{I}$ of $\hat{P}$. To show that $I \in \mathcal{AS}(P, D, t)$, we establish that (i) I is a model of $P^{M,t}$ and (ii) no model $J \subset I$ of $P^{M,t}$ exists. As for (i), since in $\hat{I}$ the atoms $atm(e)$ correctly reflect the value of the window atoms e in I, for each r in $P^{M,t}$ the rule $\hat{r} = baseRule(r)$ is in $\hat{P}^{\hat{I}}$; as $\hat{I}$ satisfies $\hat{r}$, we conclude that I satisfies r. As for (ii), we show that every model J of $P^{M,t}$ must contain I, which then proves the result.

To establish this, we use the fact that $\hat{I}$ can be generated by a sequence $\rho = r_1, r_2, r_3 \ldots, r_k$ of rules from $\hat{P}^{\hat{I}}$ with distinct heads such that (a) $\hat{I} = \{H(r_1), \ldots H(r_k)\} =: \hat{I}_k$ and (b) $\hat{I}_{i-1} = \{H(r_1), \ldots, H(r_{i-1})\}$ satisfies $B^+(r_i)$, for every $i = 1, \ldots, k$.

In that, we use the assertion that no cyclic positive dependencies through time-based window atoms $\boxplus^n \Box a$ occur. Formally, positive dependency is defined as follows: an atom $@_{t_1} b$ positively depends on an atom $@_{t_2} a$ in a ground program P at t, if some rule $r \in P$ exists with $H(r) = @_{t_1} b$ and such that either (a) $@_{t_2} a \in B^+(r)$, or (b) $\boxplus^n @_{t_2} a \in B^+(r)$ or (c) $\boxplus^n \star a \in B^+(r)$, $\star \in \{\Box, \Diamond\}$, where in (b) and (c) $t_2 \in [t-n, t]$ holds. As in $LarsToAsp(P, t)$, all ordinary atoms a are here viewed as $@_t a$. A cyclic positive dependency through $\boxplus^n \Box a$ is then a sequence $@_{t_0} a_0$, $@_{t_1} a_1$, …, $@_{t_k} a_k$, $k \geq 1$, such that $@_{t_i} a_i$ positively depends on $@_{t_{(i+1) \bmod k}} a_{(i+1) \bmod k}$, for all $i = 0, \ldots, k$ and $a_0 = b$ and $a_1 = a$ for case (c) with $\star = \Box$.

Given that no positive cyclic dependencies through atoms $\boxplus^n \Box a$ occur in P at t, and thus in $P^{M,t}$, we can w.l.o.g. assume that whenever r_i in ρ has a head ω_e for a window atom $e = \boxplus^n \Box a$, each rule r_j in ρ with a head $a_@(t')$, where $t' \in [t-n, t]$, precedes r_i, i.e., $j < i$ holds.

By induction on $i \geq 1$, we can now show that if $H(r_i) = atm(e)$, then every model J of $P^{M,t}$ must satisfy e; consequently, at $i = k$, J must contain I. From the form of the rules $baseRule(r)$ and $windowRules(e)$, the correspondence between $\hat{P}^{\hat{I}}$ and $P^{M,t}$, and the fact that the external data are facts, only the case $e = \boxplus^n \Box a(\mathbf{X})$ needs a further argument. Now if r_i is the rule $\omega_e \leftarrow a(\mathbf{X}), \text{not } spoil_e(\mathbf{X})$ on line (16), then $\hat{I}$ must satisfy a and falsify $spoil_e(\mathbf{X})$; in turn, every $a_@(t', \mathbf{X})$ must be true in $\hat{I}$, for $t' \in [t-n, t]$. From the induction hypothesis, we obtain that $@_{t'} a(\mathbf{X})$ is true in every model J of $P^{M,t}$, $t' \in [t-n, t]$, and thus $e = \boxplus^n \Box a(\mathbf{X})$ is true as well. This proves the claim and concludes the proof of the if-case, which in turn establishes the claimed correspondence between $\mathcal{AS}(P, D, t)$ and the answer sets of $\hat{P} = LarsToAsp(P, t) \cup \hat{D}$.

Remark. The condition on cyclic positive dependencies excludes that rules $b \leftarrow \boxplus^n \Box a$ and $a \leftarrow b$ occur jointly in a program. A stricter notion of dependency that allows for co-occurrence is to request in (c) for $\star = \Box$ in addition $t_2 < t$; then e.g. any LARS program where the rule heads are ordinary atoms is allowed, and Theorem 26 remains valid. □

Example 68 We consider again program $P = \{r\colon\ b(X) \leftarrow \boxplus^2 \Diamond a(X)\}$ of Example 45. We indicated the translation already in Example 46, which we now present in full for time point $t = 8$. The translation $\hat{P} = LarsToAsp(P, 8)$ is given by the following rules,

where ω abbreviates $\omega_{\boxplus^2\Diamond a(X)}$:

$$
\begin{array}{rrcl}
r_0: & b(X) & \leftarrow & \omega(X) \\
r_1: & \omega(X) & \leftarrow & \mathit{now}(\dot{N}),\, a_{@}(X,T),\, T = \dot{N} - 0 \\
r_2: & \omega(X) & \leftarrow & \mathit{now}(\dot{N}),\, a_{@}(X,T),\, T = \dot{N} - 1 \\
r_3: & \omega(X) & \leftarrow & \mathit{now}(\dot{N}),\, a_{@}(X,T),\, T = \dot{N} - 2 \\
r_n: & \mathit{now}(8) & \leftarrow &
\end{array}
\qquad
\begin{array}{rrcl}
q_1: & a(X) & \leftarrow & \mathit{now}(\dot{N}),\, a_{@}(X,\dot{N}) \\
q_2: & a_{@}(X,\dot{N}) & \leftarrow & \mathit{now}(\dot{N}),\, a(X) \\
q_3: & b(X) & \leftarrow & \mathit{now}(\dot{N}),\, b_{@}(X,\dot{N}) \\
q_4: & b_{@}(X,\dot{N}) & \leftarrow & \mathit{now}(\dot{N}),\, b(X)
\end{array}
$$

Stream D_8 above (respectively the corresponding tick stream) has the encoding $\hat{D}_8 = \{a_{@}(x,7), a_{\#}(x,7,1), \mathit{cnt}(1), \mathit{tick}(0,0), \mathit{tick}(1,0), \ldots, \mathit{tick}(7,0), \mathit{tick}(7,1), \mathit{tick}(8,1)\}$. The unique answer stream $S_8 = ([0,8], \{7 \mapsto \{a(x)\}, 8 \mapsto \{b(x)\}\})$ corresponds to the set $\{a_{@}(x,7), b_{@}(x,8), b(x)\}$. In addition, the answer set $\hat{S}$ of $\hat{P} \cup \hat{D}$ contains auxiliary atoms from $\hat{D}$, $\mathit{now}(8)$ and $\omega(x)$ for the window atom. ∎

6.4 Incremental Encoding: Program and Model Update

In this section, we present an incremental evaluation technique by adjusting an incremental variant of the given ASP encoding.

6.4.1 Incremental Translation

We illustrate the central ideas by expanding on Example 47.

Example 69 (cont'd) Consider the following rules Π similar to $\hat{P}$ of Example 68 where predicate *now* is removed. Furthermore, we instantiate the *tick time variable* $\dot{N}$ with 8 to obtain so-called *pinned* rules. (Later, pinning also includes grounding the *tick count variable* $\dot{C}$ with the tick count.)

$$
\begin{array}{rrcl}
r'_0: & b(X) & \leftarrow & \omega(X) \\
r'_1: & \omega(X) & \leftarrow & a_{@}(X,8) \\
r'_2: & \omega(X) & \leftarrow & a_{@}(X,7) \\
r'_3: & \omega(X) & \leftarrow & a_{@}(X,6)
\end{array}
\qquad
\begin{array}{rrcl}
q'_1: & a(X) & \leftarrow & a_{@}(X,8) \\
q'_2: & a_{@}(X,8) & \leftarrow & a(X) \\
q'_3: & b(X) & \leftarrow & b_{@}(X,8) \\
q'_4: & b_{@}(X,8) & \leftarrow & b(X)
\end{array}
$$

Due to stream encoding $\hat{D}_8 = \{a_{@}(x,7), a_{\#}(x,7,1), \mathit{cnt}(1), \mathit{tick}(0,0), \ldots, \mathit{tick}(8,1)\}$ we obtain a ground program $\hat{P}_{\hat{D}_8}$ from Π by replacing variable X with constant x. The resulting answer set is $\hat{D}_8 \cup \{\omega(x), b(x), b_{@}(x,8)\}$.

Assume now that time moves on to $t' = 9$, i.e., a (tick) stream D_9 at tick $(9,1)$. We observe that rules $q'_1, \ldots, q'_4$ must be replaced by $q''_1, \ldots, q''_4$, which replace time pin 8 by 9. Rule r'_0 can be maintained since it does not contain values from ticks. The time window now covers time points $7, 8, 9$. This is reflected by removing r'_3 and instead adding $\omega(X) \leftarrow a_{@}(X,9)$.

That is, based on the time increment from $(8,1)$ to $(9,1)$ rules $E^- = \{q'_1, \ldots, q'_4, r'_3\}$ and their groundings G^- (with $X \mapsto y$) *expire*, and new rules $E^+ = \{q''_1, \ldots, q''_4, \omega(X) \leftarrow a_{@}(X,9)\}$ have to be grounded based on the remaining rules (and the data stream), yielding new ground rules G^+. We thus incrementally obtain a ground program $\hat{P}_{\hat{D}_9} = (\hat{P}_{\hat{D}_8} \setminus G^-) \cup G^+$, which encodes the program P for evaluation at tick $(9,1)$. ∎

Algorithm 6.10: *IncrementalRules*(t, c, Sig)

Input: Tick time t, tick count c, signal set Sig with at most one input signal, which is empty iff (t, c) is a time increment. (Global: LARS program P)
Output: Pinned incremental rules annotated with duration until expiration

$F := \{\langle(\infty,\infty), tick(t,c) \leftarrow\rangle\} \cup$
$\quad \{\langle(\infty,\infty), a_{@}(\mathbf{x},t) \leftarrow\rangle,\ \langle(\infty,\infty), a_{\#}(\mathbf{x},t,c) \leftarrow\rangle \mid a(\mathbf{x}) \in Sig\}$
$Q := \{\, \langle(1,\infty), a(\mathbf{X}) \leftarrow a_{@}(\mathbf{X},t)\rangle,\ \langle(1,\infty), a_{@}(\mathbf{X},t) \leftarrow a(\mathbf{X})\rangle \mid \text{predicate } a \text{ is in } P \,\}$
$R := \bigcup_{r\in P} incrAspRules(r)$
return $F \cup Q \cup R$

---

defn $incrAspRules(r) = \{\langle(\infty,\infty), baseRule(r)\rangle\} \cup \bigcup_{e\in B(r)} incrWindowRules(e)$

defn $incrWindowRules(e,t,c) =$ **match** e
  case $\boxplus^n @_T a(\mathbf{X}) \implies \{\, \langle(n+1,\infty), \omega_e(\mathbf{X},t) \leftarrow a_{@}(\mathbf{X},t)\rangle \,\}$
  case $\boxplus^n \Diamond a(\mathbf{X}) \implies \{\, \langle(n+1,\infty), \omega_e(\mathbf{X}) \leftarrow a_{@}(\mathbf{X},t)\rangle \,\}$
  case $\boxplus^n \Box a(\mathbf{X}) \implies \{\, \langle(\infty,\infty), \omega_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } spoil_e(\mathbf{X})\rangle \,\} \cup$
    $\{\, \langle(n,\infty), spoil_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } a_{@}(\mathbf{X},t-1)\rangle \,\}$ //only if $n \geq 1$
  case $\boxplus^{\#m} @_T a(\mathbf{X}) \implies \{\, \langle(\infty,m), \omega_e(\mathbf{X},t) \leftarrow a_{\#}(\mathbf{X},t,c)\rangle \,\}$
  case $\boxplus^{\#m} \Diamond a(\mathbf{X}) \implies \{\, \langle(\infty,m), \omega_e(\mathbf{X}) \leftarrow a_{\#}(\mathbf{X},t,c)\rangle \,\}$
  case $\boxplus^{\#m} \Box a(\mathbf{X}) \implies \{\, \langle(\infty,\infty), \omega_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } spoil_e(\mathbf{X})\rangle \,\} \cup$
    $\{\, \langle(\infty,\infty), spoil_e(\mathbf{X}) \leftarrow a(\mathbf{X}), tick(t,c), covers^{\tau}_e(t), \text{not } a_{@}(\mathbf{X},t)\rangle \,\} \cup$
    $\{\, \langle(\infty,\infty), spoil_e(\mathbf{X}) \leftarrow a_{\#}(\mathbf{X},t,c), covers^{\tau}_e(t), \text{not } covers^{\#}_e(c)\rangle \,\} \cup$
    $\{\, \langle(\infty,m), covers^{\tau}_e(t) \leftarrow tick(t,c)\rangle,\ \langle(\infty,m), covers^{\#}_e(c) \leftarrow tick(t,c)\rangle \,\}$
  else $\emptyset$

//Practically, non-expiring rules will also be deleted when they become irrelevant.
//See Section 7.2.2 for according refinements.

Before we formalize the illustrated incremental evaluation, we present its ingredients, i.e., the incremental analogue of the program encoding of Algorithm 6.9.

Incremental Encoding: Algorithm 6.10 (*IncrementalRules*)

Algorithm 6.10 shows the procedure *IncrementalRules* that obtains incremental rules based on a tick time t, a tick count c, and the *signal set* $Sig = v(t,c)$, where $Sig = \emptyset$, if (t,c) is a time increment of k. The resulting rules of are annotated with a tick that indicates how long the ground instances of these rules are applicable before they expire.

Definition 36 (Annotated rule) *Let (t,c) be a tick, where $t, c \in \mathbb{N} \cup \{\infty\}$, and r be a rule. Then, the pair $\langle(t,c), r\rangle$ is called an* annotated rule*, and (t,c) the* annotation *of r.*

Annotations are used in two ways. First, in Algorithm 6.10, they express a *duration* how long a generated rule is applicable. Then, in Algorithm 6.11 below this duration will be added to the current tick to obtain the *expiration tick* (annotation) of a rule. If a rule

expires at tick (t, c), i.e., if its expiration tick (t', c') fulfills $t' \geq t$ or $c' \geq c$, then it has to be deleted.

Example 70 (cont'd) Each rule q'_i, $1 \leq i \leq 4$, has duration $(1, \infty)$. That is, after 1 time point, these rules will expire, regardless of how many atoms appear at the current time point. Hence, the *time duration* is 1, and the *count duration* is infinite, since these rules cannot expire based on arrival of atoms. Similarly, rules r'_i, $1 \leq i \leq 3$, have duration $(2, \infty)$ due to the time window length 2. ∎

We will discuss expiration ticks based on these durations below. Algorithm 6.10 is concerned with generating the incremental rules and their durations.

Mandatory expiration vs. optional deletion. We emphasize that annotations determine when encoded rules *must* be deleted. That is to say, annotations serve only a semantic purpose. In addition to expiring rules, rules with annotation (∞, ∞) *may* additionally be deleted at some point for optimization purposes, i.e., to limit memory usage. The algorithm description is only concerned with correctness. However, our implementation will delete further rules when they become redundant to avoid an inflation of the incremental encoding. These additional deletions will be described in Section 7.2.2, as well as other optimizations that are mentioned in the following algorithm description.

F: Streaming data. On the first two lines we add the fresh tick information (corresponding to the incremental change in the data stream) to a fresh set F. We note that facts $tick(t, c) \leftarrow$ (Line 1) are only required if the program P contains tuple windows with all-quantification, i.e., window atoms of form $\boxplus^{\#m}\Box a(\mathbf{X})$. Window rules for the other cases do not need them (see function *incrWindowRules*). Thus, due to an initial static analysis, we can use $\emptyset$ in Line 1 in case the form $\boxplus^{\#m}\Box a(\mathbf{X})$ does not occur in P. Similarly, we can also spare the inclusion of the tick-pinned atoms ($a_{\#}(\mathbf{x}, t, c)$) in Line 2, if no tuple window occurs in the program. All of these facts can remain in the encoding and thus have the infinite expiration duration (annotation) (∞, ∞), i.e., they expire neither based on time nor count. Practically, however, we will delete them as soon as no window in P can reach them anymore.

Q: Identity of $@_t a(\mathbf{x})$ and $a(\mathbf{x})$ at t. These rules are similar as in the static encoding, dropping only the *now* predicate in the bodies. As illustrated in Example 70, rules of set Q expire after 1 time point, hence the annotation $(1, \infty)$. Recall the intuition: we can specify an atom $a(\mathbf{x})$ that holds now at time t equivalently by the time-pinned atom $a_{@}(\mathbf{x}, t)$, which represents an @-atom; if the latter is derived for t it also expresses that $a(\mathbf{x})$ holds now, and in this case we also want to be able to access it as such.

R: Incremental rule encodings (*incrAspRules*). In analogy to the set R in Algorithm 6.9, we also obtain a set of incremental rule encodings for earch rule r in the program. First, we also replace r by $baseRule(r)$, which gets annotation (∞, ∞): since we only need to make sure that the body is faithfully derived, the base rule as such can remain in the program. Secondly, we traverse the body to obtain incremental window

Algorithm 6.11: *IncrementTick*(Π, G, t, c, Sig)

Input: Set of annotated, cumulative incremental rules $\Pi \supseteq \hat{D}$ collected until previous tick; its annotated groundings $G = \bigcup_{\langle (t',c'),r\rangle \in \Pi} ground(\Pi, r)$, tick time t, tick count c and signal set Sig

Result: Updated Π and G

1 $I := IncrementalRules(t, c, Sig)$
2 $E^+ := \{\langle (t + t_\Delta, c + c_\Delta), r\rangle \mid \langle (t_\Delta, c_\Delta), r\rangle \in I\}$ //determine expiration for new rules
3 $E^- := \{\langle (t', c'), r\rangle \in \Pi \mid t' \leq t \text{ or } c' \leq c\}$ //expired incremental rules
4 $\Pi' := (\Pi \setminus E^-) \cup E^+$
5 $G^+ := \{\langle (t', c'), r'\rangle \mid \langle (t', c'), r\rangle \in E^+, r' \in ground(\Pi', r)\}$ //new ground rules;
6 $G^- := \{\langle (t', c'), r\rangle \in G \mid t' \leq t \text{ or } c' \leq c\}$ //expired ground rules with exp. annotation
7 $G' := (G \setminus G^-) \cup G^+$
8 **return** $\langle \Pi', G'\rangle$

rules for deriving the encoded window atoms in function *incrWindowRules*, analogously as for the static case.

We already gave the intuition for atoms $\boxplus^n \Diamond a(\mathbf{X})$. The case for $\boxplus^n @_T a(\mathbf{X})$ is similar. Like in the static translation, we additionally have to use the time information in the head. Similarly, $\boxplus^{\#m} \Diamond a(\mathbf{X})$ and $\boxplus^{\#m} @_T a(\mathbf{X})$ expire after m new incoming atoms, instead of n time points. For $\boxplus^n \Box a(\mathbf{X})$, we add a spoiler rule for the previous time point $t - 1$, which will be considered for the next n time points.

For $e = \boxplus^{\#m} \Box a(\mathbf{X})$ we maintain two spoiler rules as in the static case that ensure $a(\mathbf{X})$ occurs at all time points in the coverage of the window, and the occurrence of $a(\mathbf{X})$ at the leftmost time point is also covered by the tick count. At tick (t, c), we have a guarantee for the next m atoms that the tick time t will be covered within the window. This is expressed by a rule $covers^{\tau}_e(t) \leftarrow tick(t, c)$ with duration (∞, m). Likewise, $covers^{\#}_e(c) \leftarrow tick(t, c)$ will select tick count c within duration (∞, m). Notably, coverage for time increments $(t + k, c)$ may extend the tuple window arbitrarily long if no atoms appear. Since the spoiler rules are based on these cover atoms they do not expire. Practically, however, they may be deleted after the next m new atoms as well. Finally, *IncrementalRules* returns the $F \cup Q \cup R$, where R contains all base rules and incremental window rules.

6.4.2 Incremental Evaluation

Algorithm 6.11 gives the high-level procedure *IncrementTick* to incrementally adjust a program encoding. We assume a function $ground(\Pi, r)$ that returns all possible ground instances of a rule $r \in \Pi$ (due to constants in Π). In fact, *IncrementTick* maintains a program Π that contains the encoded data stream $\hat{D}$ and non-expired incremental rules as obtained by consecutive calls to *IncrementalRules*, tick by tick. Moreover, it maintains a grounding G of Π, i.e., the incremental encoding for the previous tick plus expiration annotations.

The procedure starts by generating the new incremental rules I based on Algo-

rithm 6.10 described above. Next, we add for each rule the current tick (t, c) to its duration (t_Δ, c_Δ) (componentwise). This way, we obtain new incremental rules E^+ with expiration tick annotations. Dually, we collect in E^- previous incremental rules that expire now, i.e., when the current tick reaches the expiration tick time t' or count c'. The new cumulative program Π results by removing E^- from Π and adding E^+. Based on Π', we obtain in Line 5 the new (annotated) ground rules G^+ based on E^+. As in Line 3, we determine in Line 6 the set G^- of expired (annotated) ground rules. After assigning G' the updated annotated grounding in Line 7, we return the new incremental evaluation state $\langle \Pi', G' \rangle$, from which the current incremental program is derived as follows.

Definition 37 (Incremental Program) *Let P be a LARS program and $D = (K, v)$ be a tick stream, where $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$. The* incremental program $\hat{P}_{D,k}$ of P for D at tick (t_k, c_k), $1 \leq k \leq m$, *is defined by $\hat{P}_{D,k} = \{r \mid \langle (t', c'), r \rangle \in G_k\}$, where*

$$\langle \Pi_k, G_k \rangle = \begin{cases} \mathit{IncrementTick}(\emptyset, \emptyset, t_1, c_1, \emptyset) & \text{if } k = 1, \\ \mathit{IncrementTick}(\Pi_{k-1}, G_{k-1}, t_k, c_k, v(t_k, c_k)) & \text{else.} \end{cases}$$

In the remainder of this chapter, body occurrences of form $@_t a(\mathbf{X})$ are viewed as shortcuts for window atoms of form $\boxplus^\infty @_t a(\mathbf{X})$. Moreover, subsequent results will implicitly disregard auxiliary atoms like $\mathit{tick}(t, c)$, $\mathit{covers}_e^\tau(t)$, etc.

The following proposition states the correspondence between the static and the incremental encoding.

Proposition 7 *Let P be a LARS program and D be a tick data stream at tick $m = (t, c)$. Furthermore, let $\hat{P} = \mathit{LarsToAsp}(P, t)$ and $\hat{P}_{D,m}$ be the incremental program at tick m. Then $S \cup \{\mathit{now}(t), \mathit{cnt}(c)\}$ is an answer set of $\hat{P} \cup \hat{D}$ iff there exists an answer set S' of $\hat{P}_{D,m}$ that coincides with S on non-auxiliary atoms.*

Proof. We argue based on the commonalities and differences of the static encoding $\hat{P} \cup \hat{D}$ and the incremental encoding $\hat{P}_{D,m}$. Instead of body predicates $\mathit{now}(\dot{N})$ and $\mathit{cnt}(\dot{C})$, that are instantiated in $\hat{P} \cup \hat{D}$ due to the predicates $\mathit{now}(t)$ and $\mathit{cnt}(c)$, $\hat{P}_{D,m}$ directly uses the instantiations of tick variables. In both encodings, the window atom is associated with a set of rules that needs to model the temporal quantifier ($\Diamond$,$\Box$,$@_t$) in the correct range of ticks as expressed by the LARS window atom. This window always includes the last tick. While $\hat{P} \cup \hat{D}$ is based on a complete definition how far the window extends, $\hat{P}_{D,m}$ updates this definition tick by tick. In particular, the oldest tick that is not covered by the window anymore corresponds to the expiration annotation in $\hat{P}_{D,m}$.

The case $\boxplus^n \Diamond a(\mathbf{X})$ is as follows: in the static rule encoding,

$$\omega_e(\mathbf{X}) \leftarrow \mathit{now}(\dot{N}), a_@(\mathbf{X}, T)\,,$$

given $\mathit{now}(t)$, time variable T will be grounded with $t - n, \ldots, t - 0$. That is, we get a set of rules

$$\begin{array}{lrcl} (r_0) & \omega_e(\mathbf{X}) & \leftarrow & \mathit{now}(t), a_@(\mathbf{X}, t) \\ & & \vdots & \\ (r_n) & \omega_e(\mathbf{X}) & \leftarrow & \mathit{now}(t), a_@(\mathbf{X}, t - n)\,, \end{array}$$

where arguments $\mathbf{X}$ will be grounded due to data and inferences. We observe that (r_0) is the rule that is inserted to the incremental program $\hat{P}_{D,m}$ at time t (minus predicate $now(t)$, since in $\hat{P}_{D,m}$ variable T is instantiated directly with t to obtain $a_{@}(\mathbf{X}, t)$), and all rules up to r_n remain from previous calls to *IncrementalRules*. Rule r_n will expire at $t+1$, i.e., the exact time when it will not be included in $\hat{P} \cup \hat{D}$ anymore. The cases for $\boxplus^n @_T a(\mathbf{X}), \boxplus^n \Box a(\mathbf{X}), \boxplus^{\#m} \Diamond a(\mathbf{X})$ and $\boxplus^{\#m} @_T a(\mathbf{X})$ are analogous; the remaining case $\boxplus^{\#m} \Box a(\mathbf{X})$ has been argued earlier.

Finally, $\hat{P}_{D,m}$ includes a stream encoding, which is also incrementally maintained: at each tick (t, c) the tick atom $tick(t, c)$ is added, and in case of a count increment, the time-pinned atom $a_{@}(\mathbf{X}, t)$ and the tick-pinned atoms $a_{\#}(\mathbf{X}, t, c)$ are added to $\hat{P}_{D,m}$ as in $\hat{D}$. This way, we have a full correspondence with the static stream encoding $\hat{D}$.

Thus, at every tick (t, c), $\hat{P} \cup \hat{D}$ and $\hat{P}_{D,m}$ have the same data and express the same evaluations. Disregarding auxiliary atoms, we conclude that their answer sets coincide. □

In conclusion, we obtain from Theorem 26 and Proposition 7 the desired correctness of the incremental encoding.

Theorem 27 *Let P be a LARS program and $D = (K, v)$ be a tick data stream at tick $m = (t, c)$. Then, S is an answer stream of P for D at t iff $\hat{S}$ is an answer set of $\hat{P}_{D,m}$ (modulo aux. atoms).*

Proof. Given a LARS program P, a tick data stream $D = (K, v)$ at tick (t, c) by Theorem 26 S is an answer stream of P for D at t iff $\hat{S}$ is an answer set of $\hat{P} \cup \hat{D}$, where $\hat{P} = LarsToAsp(P, t)$. By Prop. 7, for any set X we have that $X \cup \{now(t), cnt(c)\}$ is an answer set of $\hat{P} \cup \hat{D}$ iff X is an answer set of $\hat{P}_{D,m}$ (modulo auxiliary atoms). In particular this holds for $X = \hat{S}$. As $\{now(t), cnt(c)\} \subseteq \hat{S}$, we obtain that S is an answer stream of P for D at t iff $\hat{S}$ is an answer set of $\hat{P}_{D,m}$, which is the result. □

The next theorem states that to faithfully compute an incremental program from scratch, it suffices to start iterating *IncrementTick* from the oldest tick that is covered from any window in the considered program. Let $\mathcal{AS}^{\mathcal{I}}(\hat{P})$ denote the answer sets of $\hat{P}$, projected to intensional atoms.

Theorem 28 *Let $D = (K, v)$ and $D' = (K', v')$ be two data streams such that (i) $D' \subseteq D$, (ii) $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$ and (iii) $K' = \langle (t_k, c_k), \ldots, (t_m, c_m) \rangle$, $1 \leq k \leq m$. Moreover, let P be a LARS program and n^* (resp. m^*) be the maximal window length for all time (resp. tuple) windows; or ∞ if none exists. If $t_k \leq t_m - n^*$ and $c_k \leq c_m - m^* + 1$, then $\mathcal{AS}^{\mathcal{I}}(\hat{P}_{D,m}) = \mathcal{AS}^{\mathcal{I}}(\hat{P}_{D',m})$.*

Proof. Assume a LARS program P and two tick data streams $D = (K, v)$ and $D' = (K', v')$ at tick (t_m, c_m) such that $D' \subseteq D$ and $K' = \langle (t_k, c_k), \ldots, (t_m, c_m) \rangle$. Furthermore, assume that (*) all atoms/time points accessible from any window in P are included in D'. We want to show $\mathcal{AS}^{\mathcal{I}}(\hat{P}_{D,m}) = \mathcal{AS}^{\mathcal{I}}(\hat{P}_{D',m})$. The central observation is that rules need to fire in order for intensional atoms to be included in the answer set, and that no rules can fire based on outdated ticks. Thus, these ticks can also be dropped.

In more detail, we assume $\mathcal{AS}^{\mathcal{I}}(\hat{P}_{D,m}) \neq \mathcal{AS}^{\mathcal{I}}(\hat{P}_{D',m})$ towards a contradiction. That is to say, a difference in evaluation arises based on data in $D \setminus D'$, i.e., atoms appearing before tick (t_k, c_k). Consider any extended atom e of a (LARS) rule $r \in P$, where the body holds only for one of the two encodings (in the same partial interpretation). Due to the assumption (*), we can exclude a difference arising from a window atom of form $\boxplus^w \star a$, $\star \in \{\Diamond, \Box, @_T\}$.

If e is an atom a, it holds in $\hat{P}_{D,m}$ iff it holds in $\hat{P}_{D',m}$ since an ordinary atom in the answer set of the encoding corresponds to an atom holding at the current time point, and both D and D' include the current time point.

The last option is $e = @_T a$, which may reach back beyond (t_k, c_k) but is viewed in the incremental encoding as syntactic shortcut for $\boxplus^\infty @_T a$. That is, in this case we have $D' = D$ and thus the encodings coincide.

We conclude that assuming $\mathcal{AS}^{\mathcal{I}}(\hat{P}_{D,m}) \neq \mathcal{AS}^{\mathcal{I}}(\hat{P}_{D',m})$ is contradictory due to these observations. Spelling out the details fully involves essentially a case distinction on the incremental window encodings and arguing about the relationship between (t_k, c_k), the respective expiration annotations, and the fact that rules accessing atoms at ticks before (t_k, c_k) are have already expired. □

The result is due to the fact that in the incremental program $\hat{P}_{D,m}$, no rule can fire based on outdated information, i.e., atoms that are not covered by any window anymore. In order to obtain an equivalence between $\hat{P}_{D,m}$ and $\hat{P}_{D',m}$ on extensional atoms, we would have to drop all atoms of the stream encoding $\hat{D}$ during *IncrementTick*, as soon as no window can access them anymore. This observation is the basis for an optimization in the implementation which also deletes non-expiring rules (i.e., those with annotation (∞, ∞)) to limit the size of the encoding. We will discuss this further in Section 7.2.2.

We now interleave notes on other approaches to incremental reasoning for plain LARS.

6.5 Further Work

In the next two sections, we give summaries of two further works on incremental reasoning. Section 6.5.1 presents the main ideas of [BDE15], which extends JTMS directly for plain LARS. Section 6.5.2 then reviews [BBU17] which developed an algorithm for efficient model update for plain LARS program with unique models.

6.5.1 Truth Maintenance for Answer Streams

In previous sections, we laid out an incremental mapping from plain LARS to ASP such that answer stream update can be carried out utilizing JTMS as update procedure. The plain LARS fragment was first considered in [BDE15], where we presented an alternative approach for incremental reasoning. There, the JTMS algorithm was extended directly for the LARS semantics, yielding an algorithm called *answer update*. The aim of the answer update is to adapt at time t a given answer stream I of a previous time point t' to account for the new data that has streamed in (and the elapsed time) since t'. A central concept is

the introduced *window-stratification*,[1] which, analogously to the usual stratification based on negation, partitions the program into strata based on occurrences of window operators such that the model computation can be carried out stratum by stratum. This requires an according definition of a *window dependency graph*[2] and an underlying extension of JTMS that takes into account the temporal validities of labels. We now give a summary of the technical ingredients and the obtained results, where we deviate slightly from the original presentation in order to align with concepts and vocabulary from above; this concerns in particular the JTMS formalization from Section 6.2.

Window-stratified Programs

The idea of *window-stratification* is to split a (plain) LARS program P into strata $P_0, \ldots, P_n$ such that an answer stream of $P = P_0 \cup \cdots \cup P_n$ can equally be obtained by computing consecutive answer streams for each stratum P_i $(1 \leq i \leq n)$, where the answer stream for P_{i-1} serves as input stream; P_0 starts with the data stream as input as usual. Strata are obtained in analogy to the usual stratified programs, using essentially window operators as splitting points rather than negation; all rules that have a cyclic dependency involving a window operator then belong to the same stratum. More specifically, the underlying *window dependency graph* of program P is built using as nodes the set $\mathcal{A}^*(P)$ consisting of all (possibly nested) occurrences of extended atoms, i.e., all syntactic elements of the forms a, where a is an atom, $@_t a$, and $\boxplus^w \circ a$, where $\circ \in \{@_t, \Diamond, \Box\}$. To express dependencies between extended atoms from $\mathcal{A}^*(P)$, we then use the following labelled edges E of form $\rightarrow^{\succeq}$, where $\succeq$ indicates the eventual relation of associated strata:

- $\boxplus^w \circ a \rightarrow^{>} a$,
- $@_t a \rightarrow^{=} a$ and $a \rightarrow^{=} @_t a$, and
- $\alpha \rightarrow^{\geq} \beta$ whenever α is the head of a rule with body element β.

Intuitively, the edge $\boxplus^w \circ a \rightarrow^{>} a$ expresses that the window atom of $\boxplus^w \circ a$ can (in general) be evaluated only after a. Similarly $@_t a$ and a in its scope are given at the same time (given a the relation of t to the current time), and $\alpha \rightarrow^{\geq} \beta$ expresses for rules that rule heads cannot be computed before their bodies. A *window-stratification* for P is then a mapping μ from $\mathcal{A}^*(P)$ to some set $\{0, \ldots, n\}$ such that $\alpha \rightarrow^{\succeq} \beta \in E$ implies $\mu(\alpha) \succeq \mu(\beta)$ $(\succeq \in \{>, =, \geq\})$. Any stratum *stratum* P_i is then defined by the set of rules in P such that the head is mapped to i by μ. Then, an answer stream of P (for a data stream D at a time point t) can be evaluated stratum by stratum as explained above.

Extension of JTMS for Plain LARS

To extend JTMS for (plain) LARS, we maintain labels $\{in, out, unknown\}$ now for extended atoms. In addition, each labelling is associated with a set of intervals that

[1] The original term was *stream-stratification*; we use here instead window-stratification to emphasize that strata are obtained by static analysis of window operator occurrences in the program, independently from streaming data.

[2] Original term: stream-dependency graph.

express for which time points they hold. Concepts of *valid*, *invalid* and *posValid* rules (cf. Section 6.2.1) carry over; the evaluation is always relative to a time point. Other JTMS data structures are briefly extended for the LARS syntax as follows.

In addition to the structural dependency between body and head elements, *consequences* also relate atoms a to window atoms $\boxplus^w \circ a$, and @-atoms $@_t a$ to window atoms $\boxplus^w @_t a$; and $@_t a$ to a. We call the latter the @-*consequences* of $@_t a$. The *support* contains, in addition to the standard definition that carries over, the @-*support* of any atom a where $\mathit{label}(a) \neq \mathit{out}$, defined as those @-atoms $@_t a$ such that $\mathit{label}(@_t a) = \mathit{in}$. The extended atoms *affected* by an extended atom α again hold the rule heads that are supported by α, plus the @-consequences (which are static). The *repercussions* are again given by the transitive closure of this relation. Moreover, some additional data structures are needed to track the transitive dependencies starting with atoms and @-atoms along the chain of rule heads. In sequel, we assume access to the described concepts and methods via an extended JTMS data structure $\mathcal{M}$ as described.

Answer Update Algorithm

We now consider a program P with strata $P_0, \ldots, P_n$, a data stream D, and an answer stream I of P for D at a previous time point t', as reflect in the model of a JTMS data structure $\mathcal{M}$, i.e., by the atoms in $\mathcal{M}$ with label *in*. The *answer update* algorithm adapts $\mathcal{M}$ such that it reflects an answer stream at the current time point $t > t'$. It does so by a loop over strata $i = 0, \ldots, n$, where at each iteration the labels (and intervals) for all atoms are updated. This is done in the following steps:

1. *Expiration & Firing:* We determine based on the current time, the window content and firing atoms (see below) expiring window atoms ω, i.e., those that will change their label to *out* and update labels and intervals accordingly. The time of expiration depends on auxiliary functions that model the specific semantics of employed window operators; we consider them here from a generic point of view. Similarly, the firing part collects window atoms ω of the current stratum that now hold due to new atoms available from $t' + 1$ to t in the stream (for $i = 0$) or from the output of the previous stratum (for $i > 0$).
2. *Update timestamps:* Based on the window atoms processed in the previous step that did not change their label, rules can be identified that remain (in)applicable. For these, the temporal intervals for labels of respective rule heads are extended.
3. *Find & Choose:* We now continue similarly like in the original JTMS. First, we reset the label of atoms and @-atoms affected by incoming and expiring atoms to *unknown*. Then, we assign labels similarly as in procedures *findLabel* (Algorithm 6.5) and *chooseLabel* (Algorithm 6.6) discussed earlier. We explicitly handle the case for update failures, i.e., when the label of an atom shall be set to *in*/*out* but was already set to *out*/*in*, the algorithm halts and returns *fail*. (Recall that Doyle's original JTMS also might fail after these two steps; in this case a backtracking procedure is started which, however, does not guarantee to return a stable model of the considered program.)

4. *Push Up:* Rule heads from the current stratum that now have status *in* are selected as input for the next iteration, i.e., for rules of stratum $i + 1$.

The correctness of the generic procedure relies on functions that correctly handle the expiration semantics of the window functions in use. Under this assertion, an answer stream can be read off the resulting labels of the updated JTMS data structure. In the absence of cyclic negation, the algorithm is also complete. In general, it may not be able to update the model and return *fail* due to the limitation inherited from JTMS, i.e., in the presence of (odd) loops through negation. Hence, developing a suitable backtracking algorithm to ensure completeness would be a natural next step. Furthermore, other techniques for model update deserve attention. We present one such approach next.

6.5.2 The Laser Stream Reasoning Engine

Chapter 7 will present *Ticker*, the implementation of the evaluation techniques presented in Sections 6.4 and 6.3 above. We now briefly describe the core ideas of another prototype reasoning engine called *Laser*, presented in [BBU17].

In contrast to Ticker, which provides a solution where multiple models can be considered, Laser explicitly restricts to positive plain LARS programs and stratified programs, which ensure a unique model, and thus can be computed more efficiently. It also focuses on sliding time-based and tuple-based windows. Instead of working with encodings to ASP, we extend semi-naive evaluation techniques as in Datalog [AHV95], taking into account intervals in which formulas are guaranteed to hold due to employed window operators and temporal modalities. In essence, the aim is to efficiently compute variable substitutions based on streaming data and derivations as specified by the program, and to avoid redundant recomputations, exploiting the temporal information. Thereby, we obtain a mechanism that performs both rule grounding and model computation. More precisely, the set of ground atoms returned by the algorithm corresponds to the answer stream at the respective time point and since these atoms are maintained incrementally, we obtain an incremental answer stream update mechanism.

Recall that the incremental encoding technique (Algorithm 6.10) expires rules after a duration that follows from the length of the window. Here, we determine in a similar way how long a substitution formula will hold. More specifically, we consider *annotated ground formulas* (AGFs), i.e., expressions of form

$$\varphi\sigma_{[c,h]} ,$$

where φ is a formula, σ a substitution (written postfix), and $[c, h]$ an *annotation*, i.e., an interval from a *consideration time* c to a *horizon time* h in which the ground formula $\varphi\sigma$ necessarily holds. (We first consider only sliding time-based windows; tuple-based windows are discussed later.)

Incremental Evaluation

We first explain the main steps of the incremental model update by means of an example.

Example 71 Consider the following rule r, adapted from 3.6 of Example 27.

$$\underbrace{@_{T'}\, exp(Id, Y)}_{\alpha} \leftarrow \underbrace{\boxplus^5 @_T\, tram(Id, X)}_{\beta_1}, \underbrace{plan(X, Y, D)}_{\beta_2}, \underbrace{T' = T + D}_{\beta_3}$$

Assume we have a signal $tram(i, x)$ at $t = 36$ in the data stream. We thus can instantiate $tram(Id, X)$ in β_1 with the substitution $\sigma_1 = \{Id \mapsto i, X \mapsto x\}$. The ground atom $tram(i, x)$ $(= tram(Id, X)\sigma_1)$ only holds at time 36, hence the AGF $tram(Id, X)\sigma_{1[36,36]}$. On the other hand, background data does not expire. Assume a background fact $plan(x, y, 5)$ specifying that it takes 5 minutes for a tram to get from station x to station y. We thus get AGF $plan(X, Y, D)\sigma_{2[0,\infty]}$, where $\sigma_2 = \{X \mapsto x, Y \mapsto y, D \mapsto 5\}$, i.e., the horizon time of $plan(x, y, 5)$ is infinite.

Consider now the window atom β_1. The *tram* signal at time 36 matches the time variable T in the @-operator, we thus extend the substitution to $\sigma_1' = \sigma \cup \{T \mapsto 36\}$. We observe that $\beta_1\sigma_1'$ will hold until time point 41 due to he window length, so we obtain AGF $\beta_1\sigma'_{1[36,41]}$.

Next, $\beta_1\sigma_1'$ and $\beta_2\sigma_2$ can be joined since the shared variable X is uniformly mapped to constant x. We thus get a joint substitution $\sigma_{12} = \sigma_1' \cup \sigma_2$ for the formula $\beta_1 \wedge \beta_2$ as usual. Clearly, this conjunction holds whenever both conjuncts hold, i.e., we have to calculate the intersection of the annotations $([36, 41] \cap [0, \infty])$; this again is $[36, 41]$. Finally, the substitution for β_3 is created directly when joining with $\beta_1 \wedge \beta_2$, where $T' \mapsto 41$ is determined.

In total, we obtain for the entire body the AGF $\beta_1 \wedge \beta_2 \wedge \beta_3\, \sigma_{[36,41]}$, where $\sigma = \{Id \mapsto i, X \mapsto x, Y \mapsto y, T \mapsto 36, D \mapsto 5, T' \mapsto 41\}$. (For uniformity, we view β_3 as a logical predicate.) This expresses in particular a grounding for variables Id, Y and T' of the rule head that holds during the interval $[36, 41]$. Thus, we create another AGF $\alpha\sigma'_{[36,41]}$ for the rule head, where σ' is obtained from σ by a projection to head variables. At time 42, all ground atoms with horizon time 41 are deleted; only $plan(x, y, 5)$ will remain. ∎

Example 71 illustrated how atoms from data streams and background knowledge are used to compute a model of the program in a way that may avoid recomputation: assuming no data arrives within interval $[37, 41]$, no further computations are needed during that period, and derived conclusions remain valid. More generally, the algorithm for positive programs and sliding time-based windows works in the following steps at the turn from time point $t - 1$ to t. (We skip some details, which will be discussed below.)

1. *Expiration:* delete all AGFs $\varphi\sigma_{[c,h]}$ where $h < t$.
2. *Input:* for every input atom $a(\mathbf{x})$ in the data stream at t, create an AGF $a(\mathbf{x})\sigma_{[t,t]}$, where σ is empty.
3. *Base σ from input:* for every such AGF and a non-ground atom $a(\mathbf{X})$ (of the same predicate) occurring in a rule body create further AGFs $a(\mathbf{X})\sigma_{[t,t]}$, where σ replaces variables in $a(\mathbf{X})$ by constants in $a(\mathbf{x})$ as usual.

4. *Adapting σ by window atoms:* based on temporal modality $\star \in \{\Diamond, \Box, @_{t'}\}$ of any window atom $\varphi = \boxplus^n \star a(\mathbf{X})$, we create an AGF for φ based on σ of $a(\mathbf{X})\sigma_{[t,t]}$ as follows:
   (i) Case $\Diamond$: $\varphi\sigma_{[t,t+n]}$; i.e., the horizon time is extended by the window length.
   (ii) Case $\Box$: $\varphi\sigma_{[t,t]}$; no extension is possible.
   (iii) Case $@_T$: $\varphi'\sigma_{[t,t+n]}$, where $\sigma' = \sigma \cup \{T \mapsto t\}$; i.e., in addition to extending the horizon time as for $\Diamond$, the time variable is replaced by the current time t.[3]
5. *Body join:* for every new AGF of a body element β_i in a rule, we try to obtain a joint substitution σ for the entire body $\beta = \beta_1 \wedge \cdots \wedge \beta_n$ as usual. If such σ is created from AGFs with respective annotations $[c_1, h_1], \ldots, [c_n, h_n]$, we use the annotation $[c, h]$ obtained by their intersection. In case $[c, h]$ is non-empty, we obtain the AGF $\beta\sigma_{[c,h]}$.
6. *Head derivation:* for the newly derived body AGF $\beta\sigma_{[c,h]}$, we get for the rule head α a substitution σ' by dropping in σ all variables that do not occur in α and store $\alpha\sigma'_{[c,h]}$.
7. *Propagation:* Derived AGFs for rule heads are treated like input atoms' AGFs in further rules; i.e., Step 3 is applied uniformly for input atoms and head atoms. This procedure continues until a fixed-point is reached, i.e., when no new body AGF can be obtained and thus no new AGF for any rule head.

Thus, we essentially create substitutions from the data stream and from derived rule heads, using window lengths to (potentially) extend the horizon time, and calculating the intersection of body annotations to obtain the temporal guarantee when the rule fires. The evaluation is incremental in the sense that it will not recompute entailments that have been obtained already; only the horizon times might be updated in this case. By repeating the procedure iteratively we get an algorithm to update the model over time; assuming that at time t we have the set S of AGFs after the fixed-point computation, we obtain the answer stream $I = ([0, t], \upsilon)$ as follows. Let $a(\mathbf{X})\sigma_{[c,h]} \in S$. If σ does not contain a time variable (from an @-atom), then $a(\mathbf{X})\sigma \in \upsilon(t)$, else $a(\mathbf{X})\sigma \in \upsilon(t')$, where t' is the substitution for the time variable in σ.

Note that static data can be uniformly processed by initially assigning an AGF $a(\mathbf{x})\emptyset_{[0,\infty]}$ for every provided fact $a(\mathbf{x})$ (corresponding to Step 2); an according substitution σ for every matching occurrence $a(\mathbf{X})$ in a rule then results in an AGF $a(\mathbf{X})\sigma_{[0,\infty]}$ (corresponding to Step 3). Background facts are practically not considered to be used within the scope of windows but cases 4 (i)-(iii) could be adapted easily.

Extension for Stratified Plain LARS Programs

The presented incremental evaluation procedure works for positive programs which permit a fixed-point computation as described. The algorithm can also be applied for stratified

[3] The case $@_{t'}$, where t' is ground, is only of theoretical interest. In this case we have to use $\varphi\sigma_{[t,t+n]}$ if $t = t'$, else no AGF is created for φ.

plain LARS programs, using the following adaptions. We first recall that plain LARS programs may additionally have negation in front of window atoms. Notably, we can equivalently consider negation to occur directly in front of atoms: $\neg\boxplus^n\Diamond a(\mathbf{x})$ equals $\boxplus^n\Box\neg a(\mathbf{x})$ and $\neg\boxplus^n\Box a(\mathbf{x})$ equals $\boxplus^n\Diamond\neg a(\mathbf{x})$. In case the timeline contains t (which we can assume since the timeline is implicit) we also have that $\neg\boxplus^n@_t a(\mathbf{x})$ equals $\boxplus^n@_t\neg a(\mathbf{x})$. Thus, we can use literals instead of atoms as unit elements of the evaluation algorithm, and consider in case of negated atoms $\neg a(\mathbf{X})$ all substitutions σ such that $a(\mathbf{X})\sigma$ does not hold. The base set of possible substitutions to form the according complement set is given by the remainder of the rule; i.e., we can adopt the usual safety assumption that every variable occurring under negation needs to occur positively in the same rule. Restricting to stratified programs retains the uniqueness of models.

Extension for Tuple-based Windows

For ease of presentation, we focused on time-based windows, but (sliding) tuple-based window can be realized analogously. To this end, we consider for every formula in addition to the time-based annotation $[c, h]$ a tuple-based annotation $[\#c, \#h]$ with an analogous *consideration count* $\#c$ and a *horizon count* $\#h$. An AGF $\varphi\sigma_{[c,h],[\#c,\#h]}$ then expresses that the grounding $\varphi\sigma$ holds if the current time is in $[c, h]$ and the global tuple count is in $[\#c, \#h]$. Consequently, we delete before processing the n^{th} atom any AGF $\varphi\sigma_{[c,h],[\#c,\#h]}$ where $\#h < n$. The aim of the typical tuple-based window is to count input data, not inferences. Thus, as for Ticker, we restrict the use of tuple-based windows to extensional atoms. This allows us to use the presented update methods of time-based windows (which may range also over intensional data) for tuple-based windows.

Evaluation

The resulting implementation Laser[4] was evaluated empirically for a set of small benchmark programs, where we examined the processing time per triple for Laser and competing engines. For the evaluation of the prominent snapshot semantics (featuring only modality $\Diamond$ in window atoms), we evaluated the update of a single rule (with one, resp. two window atoms with join) in Laser, and the analogous queries in C-SPARQL and CQELS. For operators not available in these languages, we compared Laser with Ticker in one-shot solving mode. (The incremental mode of Ticker was developed at the same time when Laser was evaluated.) Laser and Ticker were also compared on a small benchmark program with multiple rules, requiring some propagation (Step 7). For all evaluations, a tailored data generator made sure that the different input handling of respective engines did not lead to unfair comparisons. In all of our evaluations, which used different stream rates and window sizes, Laser was orders of magnitude faster than C-SPARQL, CQELS, and Ticker. While these evaluations indicate the desired efficiency, we shall note, however, that the obtained empirical results are not conclusive. For more robust results, a larger and more diverse set of benchmark programs is required. In particular, the evaluation of programs with negation and tuple-based windows remain for future work.

[4]Code, written by co-author Hamid R. Bazoobandi: `https://github.com/karmaresearch/Laser`

6.6 Discussion and Related Work

In this chapter, we presented a formalization of Doyle's core JTMS algorithm and how it can be used to update the answer stream of a plain LARS program based on an incremental encoding to ASP; the latter is derived from a complete encoding for evaluating a program once at a given time point. These specific techniques come with some limitations and point towards future research issues, i.e., the chosen LARS fragment, the grounding procedure and JTMS as specific model update algorithm. We discuss them and finally mention some related work.

LARS Fragment

Given the very generic LARS framework as presented in Section 3, the restriction to plain LARS is natural as it can be seen as straightforward extension of normal ASP for streams: it simply replaces atoms in normal rules by extended atoms in rule bodies, and also considers @-atoms in rule heads. With a focus on incremental reasoning, we deliberately restricted further to sliding time windows and sliding tuple windows. For single tumbling windows, we cannot expect performance gains, since they do not share data from step to step. However, the combination of multiple tumbling windows (within or across rules) might also provide potential for incremental reasoning, when different window sizes are combined. Moreover, adding further window functions and allowing for nested use is of interest; in particular filter windows before tuple windows. In many use cases one does not want to count all tuples but only a specific subset. In fact, the presented encodings can be adapted in a straightforward way.

Example 72 Consider the rule $@_T\mathit{lastBus} \leftarrow \boxplus^{bus} \boxplus^{\#1} @_T\mathit{bus}(\mathit{Id}, X)$, which first filters atoms with predicate *bus* and then uses a tuple window of size 1 to select to most recent appearance. Without the filter window (and window nesting), we can only write $@_T\mathit{lastBus} \leftarrow \boxplus^{\#1} @_T\mathit{bus}(\mathit{Id}, X)$, which is applicable to select the time T of the last bus appearance only if the stream exclusively contains *bus* signals. ∎

Incorporating the filter window is more a matter of adapting the developed ideas above rather than requiring genuine new techniques; it basically boils down to generalizing the generation of incremental rules in Algorithm 6.10. Once the program is fixed, we can use for a formula of the form $\boxplus^f \boxplus^{\#m} \circ a(\mathbf{X})$, where $\circ \in \{@_T, \Diamond, \Box\}$, a fresh predicate $a_{f\#}$, similarly like the tick-pinned predicate $a_\#$ for the tuple windows as used above. In analogy to the global counter c in ticks (t, c), we would then require in addition to $a_\#(\mathbf{X}, t, c)$ the tailored one of the form $a_{f\#}(\mathbf{X}, t, c_{f\#})$, where $c_{f\#}$ is a counter restricted to predicate $a_{f\#}$. Similarly, we would have to adapt the generation of incremental rules, i.e., duplicate the cases for tuple windows for every such pre-filtering.

A different issue of more theoretical interest would be to provide encoding mechanisms for classes of window functions based on abstract properties. For instance, the tuple window encoding as used only works due to the restriction to extensional atoms. In a similar way, it should be possible to incorporate new windows functions (employed in the

plain LARS fragment in unnested window operators) by simply adding new auxiliary atoms and new incremental rules. On the other hand, as soon as windows also access intensional atoms, they interfere with the LARS semantics in a more complex way. Thus, one could try to single out properties that permit (respectively prohibit) modular extensions of the existing algorithm.

Grounding

The heart of the incremental model update is Algorithm 6.11, which adapts the encoding tick by tick. Due to our focus on the composition of rules as such, we simply assume a grounding function in Line 5 that takes schematic rules from Algorithm 6.10 and produces a grounding, which then serves as input for the JTMS. Clearly, developing algorithms for suitable grounding on-the-fly is a major research issue itself; with great importance for further development of rule-based stream reasoning. In particular, such a grounding mechanism must be highly efficient and thus should also work incrementally. In order to provide a research prototype based on the ideas presented in this chapter, we developed Ticker (cf. Chapter 7) using a (partial) pre-grounding; all remaining variables in the incremental processing then correspond to tick variables (time or count) and can be grounded in constant time by a replacement with values from the current tick. For further development grounding techniques such as in [LN09], [PDPR09], [DEF+12] and [Wei17] might be considered.

Model Update

The choice of JTMS for carrying out model update was a pragmatic one due to the known (partial) correspondence with ASP, as described earlier. However, the JTMS algorithm is not applicable for programs with odd loops and also by far less efficient than modern ASP solving techniques; in particular when a model needs to be computed from scratch. It is also limited to ground programs. Consequently, providing other incremental model update techniques based on changing programs would be not only of interest for stream reasoning with LARS, but also for ASP without an explicit timeline. Some related works were mentioned in Section 2.2.5.

Beyond the issue of runtime/performance gains due to incremental evaluation would be studies on criteria or guarantees regarding the semantics of model update, which we did not consider in this work. Arguably, notions of *maintainability* of models, or degrees thereof, would pave the way for intriguing future research issues on the theoretical side. In particular, in applications with a multitude of alternatives one would like to stick with the same solutions or small deviations over time, rather than jumping between various different solutions from one time point to the next. In fact, also JTMS may retract derivations even when it does not have to.

Example 73 Consider the rules $c \leftarrow$; $a \leftarrow c$; $a \leftarrow \text{not}\, b$; and $b \leftarrow \text{not}\, a$; to be inserted in this order to the JTMS network. It will hold the model $\{a, c\}$, where c supports a. When we now remove fact $c \leftarrow$, atom a is marked as *unknown* since it is a repercussion of c.

The choice between a and b due to the last two rules now may also pick b, although a can be maintained. In other words, there is no preference for model $\{a\}$ over $\{b\}$. ∎

Example 73 suggests a specific model maintenance criterion: to prefer $\{a\}$ over $\{b\}$ after $\{a, c\}$, with the intuition that a can be maintained. However, this is a specific criterion which cannot easily be argued for: in general, multiple (mutually exclusive) subsets of a previous model may be maintainable, so one would need a further criterion to argue which. Such a criterion can realistically only come from encoding domain knowledge; we speculate that usefuly such criteria are not expressible in generic terms at a low level of counting atoms or with similar uniform metrics. Depending on specific meanings of the propositions, $\{b\}$ could in fact be the natural update to $\{a, c\}$ after removal of c; e.g., when it is more likely to have $\{b\}$ rather than $\{a\}$ without c in the specific domain.

The most basic requirement for an update procedure is that it maintains the same model over time if possible. As a basic example, consider the two rules $a \leftarrow \text{not}\, b$; and $b \leftarrow \text{not}\, a$; where the output should not flip a and b arbitrarily without further changes. The same criterion naturally applies for projections which are important in practice. For instance, if only a and b in Example 73 above are atoms relevant to the user, and c is auxiliary, then $\{a, c\}$ practically amounts to $\{a\}$, and the output should remain (identical) after c was removed.

As we see already in this brief discussion, developing criteria how models shall be maintained opens up a large field for future research. With few exceptions such as the simple ones above, most criteria will require specific use cases for their assessment or even their definition. In other words, rather than exploring immediately available relations (such as naive model differences at the abstract level), going backwards from application demands seems a better strategy to find suitable model maintenance criteria (and methods) for practice.

Related Work

We finally point to some literature on incremental reasoning. In particular, we mention some work from database research, where the closely related problem of view updates has been studied extensively. Views on relational tables are typically specified with SQL `select` statements, or in case of deductive databases, by means of logical rules; most prominently Datalog. The general task is to update the materialized view (i.e., intensional predicates) in response to a change in the base relation (i.e., extensional predicates). The incremental variant of this task then needs to identify for the previous materialization R the new tuples R^+ to add, and potentially old tuples R^- to delete, and process the update $R' := (R \setminus R^-) \cup R^+$ efficiently.

In [CW91], the aim is to maintain a view materialization by a set of production rules that are generated automatically in order to ensure their correctness. The authors provide a mechanism that takes an SQL-based view definition and outputs view maintenance rules that are triggered by changes in the base relations, i.e., `insert`, `delete` and `update` commands. The maintenance itself also works with changes, i.e., incrementally. Notably, the production rules work operationally, and serve more or less as a declarative variant for

procedural code. In contrast to stating by incremental rules new commands how to update a previous materialization, our work uses incremental rules to update the declarative definition of a view, in that sense, where the JTMS is responsible to incrementally update the materialization in terms of a model.

A classic approach for updating (potentially recursive) views on a base relation is semi-naive evaluation [GUW09]. It is sufficient to derive tuples that have to be inserted to the view. We used this technique as basis for our work on Laser (cf. Section 6.5.2). However, straightforward semi-naive evaluation does not work for tuple deletions; a derived tuple of a deleted tuple from the base relation would always be deleted, even if it can still be derived by another one. The DRed (Delete and Rederive) algorithm [GMS93] refines the overestimate of potential deletions by taking into account the alternative derivations of removal candidates; deletes them and rederives them in case they still hold.

A variant of the DRed algorithm in declarative form was presented in [SJ96], where a maintenance program is obtained from the original program (i.e., the view definition) by a rewriting technique that classifies different kinds of rules: for the original view definition p (i.e., the head of a rule), p^{del} captures possible deletions of relation p; p^{red} captures the subset that has alternative derivations (and thus must not be deleted); and p^{ins} reflects the insertions. The incremental update of rules to be added and deleted then is expressed by the rules $p^{+} \leftarrow p^{ins}, \neg p$; and $p^{-} \leftarrow p^{del}, \neg p^{ins}, \neg p^{red}$; respectively. With the aim to separate the view generation from its materialization at the same storage location, the authors then present an algorithm to adapt an external materialization incrementally based on updates on demand. This work was extended in [VSM05] for maintenance of ontological entailments; it considers ontologies expressible in Datalog with stratified negation. Accordingly, when the ontology is updated, not only ground facts but rules need to be updated, for which the previous rewriting techniques have been extended. Algorithms are presented to maintain the rewritten rules in response to a change of rules encoding the ontology.

Building on works mentioned above, [BBC+10b] presents a reasoning method that incrementally maintains materializations of ontological entailments that depend on streaming data. Similarly like in [VSM05], ontological schemas (such as the RDFS) are assumed to be axiomatized in Datalog. Streaming RDF triples that are inserted to the materialization are annotated with timestamps that serve to expire them once they are no longer valid.

An improvement of the DRed algorithm was addressed by the Backward/Forward (B/F) algorithm [MNPH15], which avoids the costly overdeletion phase. Instead, it determines by a combination of backward and forward chaining which of the facts that may have to be removed have alternate proofs from remaining facts. The algorithm was shown to be significantly faster than DRed in some ontology query answering benchmarks. However, it is not applicable to programs with negation. Recently, both DRed and B/F have been advanced further in [HMH18], which presents hybrid solutions in combination with the Counting algorithm [GMS93]. The latter maintains a counter for distinct derivations of a fact, deletes a fact once the counter is zero and thus avoids the backward evaluation. However, it is not applicable to recursive rules. DRed and B/F

are combined with Counting in two variants, one tracks derivation counts recursive and non-recursive rules separetly, and another tracks only non-recursive ones. Both variants achieve significant performance benefits, as shown in different evaluations.

Incremental data stream evaluation of queries with explicit time-based sliding windows has previously been studied in [GHM+07], which compared two relational processing approaches with respect to efficiency. In the *input-triggered approach* (ITA) only the new tuples are processed and query operators use their timestamps to expire them. The price of ITA is a potentially significant output delay of the query answer. Second, the *negative tuples approach* (NTA) aims to reduce these delays. The conceptual idea is to model expiration of a tuple t with an explicit expiration tuple t' that is processed at the intended expiration time and then triggers the retraction of the effect of t. In the query pipeline, which consists of incremental relational operators, this doubles the number of tuples to be processed and causes an overhead in processing. To tackle these two issues, the paper presents two optimization techniques: *time-messaging* avoids unnecessary processing in incremental operators (mainly in joins), and *piggybacking* reduces the number of negative tuples by dynamically selecting either ITA or NTA execution modes based on incoming tuples. The first technique was shown to improve NTA by a factor of two, whereas piggybacking yields the respective performance benefits of ITA and NTA.

All of the above works share the principle of maintaining a unique view. This is in contrast with our work that aims at updating one view (model) among potentially many. In other words, the main semantic difference to works on view materialization updates lies in our need to select from potentially multiple solutions. To this end, we used JTMS, which is preferable over assumption-based truth-maintenance systems [dK86] for our task: they maintain multiple models and are less suitable for applications where only a single model is required at a time. TMS techniques have also been studied in [RP11] to deal with ontology streams; their work however departs from our setting since it is monotonic and features neither windows nor time references. Further related work in the area of Knowledge Representation was mentioned in Section 2.2.5, including the incremental features [GKKS17] of the ASP solver Clingo [GKKS14]. In Chapter 7, we will make use of Clingo only in its standard solving mode (by means of repeated one-shot solving), since multi-shot solving features did not match the demands of our encoding (cf. Section 7.5). Due to the increasing demand of streaming solutions, we can expect further developments of incremental evaluation techniques in research areas targeting expressive reasoning.

CHAPTER 7

The Ticker Engine

This chapter presents Ticker, a prototypical stream reasoning engine based on the techniques developed in Chapter 6.

Outline

We present in Section 7.1 Ticker programs and how they are translated to LARS programs. We then discuss the Ticker engine's different modes of operation explaining its runtime options. In Section 7.2 we describe the pre-grounding mechanism that is used as a pre-processing step of the incremental reasoning mode. We present more fine-grained incremental algorithms based on those of Chapter 6, taking into account practical space limits and the pre-grounding procedure. Section 7.3 then gives an overview of the implementation architecture, centered around the *reasoner* abstraction. We give some details how the reasoners are realized, i.e., utilizing either repeated one-shot solving with Clingo or incremental reasoning with an implementation of Doyle's justification-based truth maintenance system (JTMS). These two reasoning modes are then evaluated empirically in Section 7.4. We formulate research questions, followed by a setup and benchmark programs towards their examination. We also present and discuss the obtained results. Finally, we add some remarks in Section 7.5, including ideas for future research directions.

Publications

This chapter extends the practical and empirical part of publication [BEF17], the technical core of which has been presented in the previous chapter. We give here a description of the developed software and the implementation of the presented algorithms that could not be included in [BEF17] due to space constraints. One of the benchmark programs is derived from our work in [BBD+16] and [BBD+17].

7.1 Introduction

Ticker is a stream reasoning engine for the command line. It is written in Scala and is available at `https://github.com/hbeck/ticker`, published under the MIT license. We give a brief overview of Ticker's functionality, which is reflected in the runtime options, all of which are fixed once the engine starts.

Among the start-up parameters is the program to be evaluated, which amounts to a plain LARS program. The engine can read input atoms from standard input and/or sockets and may write its result likewise to standard out and/or sockets. Notably, the results are not answer streams (or sets thereof). Instead, we define the *answer* of Ticker at time t as the set of facts that currently hold in a randomly selected answer stream. More formally, if $I = (T, \upsilon)$ is the currently computed answer stream at time t, then the *answer (at t)* is defined as $\upsilon(t)$. The actual *output* is then a subset of an answer as specified by the given *filter*, i.e., a set of atoms to which the resulting predicates are projected. For instance, one may restrict the output to intensional atoms (i.e., inferences) or to an explicitly declared set of predicates.

In contrast to LARS, Ticker has an explicit notion of time. We therefore introduce the following concept.

Definition 38 (Clock time) *The* clock time $c \in \mathbb{N} \setminus \{0\}$ *defines the time that passes from one time point to the next. It is specified in a* time unit $u_c \in \{ms, s, min, h\}$.

The clock time is the time associated with a single time point (in LARS) and a single time tick during the run of the engine. It also serves as factor to translate the temporally specified time windows in Ticker to those in LARS (see below). For seconds (unit s), also unit symbol *sec* is allowed as alternative.

Independent from the clock time one can specify when output shall be written to the selected sinks. This *output timing* has three modes: reporting the output when it changed (after filtering), when a fixed amount of time passed, or after a fixed number of input signals.

Finally, Ticker reasoning can be controlled by a parameter, where one selects either repeated one-shot solving with Clingo [GKKS14] (based on techniques of Section 6.3) or incremental model update (as developed in Section 6.4). The implementation of these techniques will be studied in Section 7.3. Before that, we are now going to formally introduce Ticker programs (Section 7.1.1) and its runtime options (Section 7.1.2) where we explain the mentioned configuration variants in more detail.

7.1.1 Ticker Programs

We now present the syntax of Ticker programs, which amount to plain LARS programs with sliding time windows and tuple windows. We use type writer font to distinguish the Ticker syntax from its formalization in LARS. We thus write `p`, `p(x)`, `p(x,y)`, etc. for atoms (using upper case for variables, e.g., `p(X)`, `p(X,Y)`, etc). Moreover, we write `@T p` for an @-atom $@_T\, p$. Next, we write `[n #]` for a tuple window that selects the last `n`

Ticker	LARS
`p`	p
`p(X,Y)`	$p(X, Y)$
`@T p`	$@_T\, p$
`a :- b, not c.`	$a \leftarrow b, \text{not}\, c$
`a(X) :- b(X,Y), not c(Y).`	$a(X) \leftarrow b(X, Y), \text{not}\, c(Y)$
`[n u]`	$\boxplus^{n_c}$, where
− `n` $\in \mathbb{N}$	− $n_c = \mathtt{n}/c_u$ and
− `u` $\in \{\mathtt{ms}, \mathtt{s}, \mathtt{sec}, \mathtt{min}, \mathtt{h}\}$	− c_u is the clock time in unit `u`
`[m #]`, where `m` $\in \mathbb{N} \setminus \{0\}$	$\boxplus^{\#m}$
`p [...]`	$\boxplus^{\cdots} \Diamond\, p$
`always p [...]`	$\boxplus^{\cdots} \Box\, p$
`@T p [...]`	$\boxplus^{\cdots} @_T\, p$
`=, !=, <, <=, >, >=`	$=, \neq, <, \leq, >, \geq$
`+, -, *, /, ^, %`	$+, \text{-}, \cdot, /, \hat{}, \%$

Table 7.1: Syntax overview of Ticker programs

atoms (`n` $\in \mathbb{N} \setminus \{0\}$), and `[n u]` for a time window that selects the temporal duration `n` $\in \mathbb{N}$ in unit `u`, i.e., `ms` (milliseconds), `s` or `sec` (seconds), `min` (minutes) or `h` (hours). For instance, `[1 s]` and `[1000 ms]` denote time windows the select the last second. While `[n #]` can be directly translated to $\boxplus^{\#n}$, the conversion of Ticker time windows to LARS time windows requires conversion based on the clock time. It is set when the engine is started. We say time window `[n u]` is *compatible* with clock time c (in unit u_c), if size `n` is a multiple of c. More formally, let n_c be the window size `n` converted to unit u_c. Then, it must hold that $n_c \in \mathbb{N}$ and $n_c = k \cdot c$ for some $k \in \mathbb{N}$. We call n_c the *corresponding time points*. Consequently, in case time windows are used, the translation of a Ticker program to a LARS program depends on the clock time. We thus obtain the *Ticker window operators* (Ticker windows for short) and write

- `[n #]` for $\boxplus^{\#n}$, and
- `[n u]` for $\boxplus^{n_c}$, where n_c are the corresponding time points.

Example 74 Consider a time window `[3 s]` that selects the last 3 seconds. It is compatible with a clock time of $c = 1s$ (one second, $u_c = s$). In this case, it corresponds to the LARS window $\boxplus^3$. Using clock time $500ms$ would give a LARS window $\boxplus^6$. The time window `[3 s]` is not compatible with clock times $400ms$, $2s$ or $4s$. ∎

It remains to specify the syntax of *window atoms*. Given a schematic Ticker window `[win]` that corresponds to a LARS window operator $\boxplus^w$, we write

- `p [win]` for $\boxplus^w \Diamond p$,
- `always p [win]` for $\boxplus^w \Box p$, and
- `@T p [win]` for $\boxplus^w @_T\, p$.

where `p` is an atom and `@T p` is an @-atom.[1] Naturally, we say a variable is in the *scope* of a window (or window atom), if it appears in one of these expressions.

Example 75 Let $c = 100ms$. Then, `a(X) [1 s]` expresses $\boxplus^{10}\Diamond a(X)$, `always p [42 #]` stands for $\boxplus^{\#42}\Box p$, and `@T rel(x,y) [5 s]` translates to $\boxplus^{50}@_T\, rel(x, y)$. Variable `X` occurs in the scope of `[1 s]`. ∎

As for plain LARS, we define *extended atoms* by atoms, @-atoms and window atoms.

Definition 39 (Ticker program) *A* Ticker rule `r` *is an expression of the form*

`h :- b`$_1$`, ..., b`$_n$`, not b`$_{n+1}$`, ..., not b`$_m$`.,`

where head `h` *is an atom or an @-atom, and each body element* `b`$_i$ *(*$1 \leq i \leq m$*) is an extended atom. A* Ticker program *is a set of Ticker rules.*

As usual, the separator `:-` in a fact `h :- .` can be omitted. In this chapter, we sometimes omit the final dot (`.`) for better readability. Notably, with the exception of time windows, which involve a conversion from a temporal duration to a number of time points, Ticker programs are merely a syntactic variant of plain LARS programs with (sliding) time and tuple windows. Table 7.1 summarizes the syntax.

Example 76 Table 7.2 shows four exemplary Ticker rules and their translations to LARS relative to the given clock time c. For instance, the expression `p(X) [5 s]` in the body of Rule (1) corresponds to the window atom $\boxplus^5\Diamond p(X)$, using clock time $c = 1s$, i.e., 5 seconds correspond to 5 time points. For Rule (2), where $c = 15s$, `@T p(X,Y) [4 min]` amounts to $\boxplus^{16}\Diamond p(X, Y)$. For tuple windows, the clock time is irrelevant. As specific example, consider window atom `always q(Y) [7 #]` appearing under negation in Rule (2). It corresponds to $\boxplus^{\#7}\Box q(Y)$. ∎

Conventions and Restrictions

We now present restrictions and conventions that describe the subclass of programs accepted by the Ticker engine. In case any criterion mentioned above is not met, the engine terminates immediately after printing an according error message.

Implicit definition of signals. No declaration of extensional atoms and intensional atoms are needed. Stream signals are extensional by definition, and implicitly given by those predicates appearing in the scope of windows which are not used as rule heads. The latter are viewed as intentional predicates. We assume that background data is not used in the scope of windows. (This is in line with the entailment definition; windows cannot drop background data and using them nevertheless may be useful only in more complex formulas, which are beyond the plain LARS fragment). Accordingly, predicates

[1] Towards analogy with the case for $\Box$, the case for $\Diamond$ should arguably read `some p [win]`; however, we choose the shorter `p [win]` since $\Diamond$ is by far the most frequently used variant in practice.

(1)	`h(X,Y) :- g(X,Y), p(X) [5 s], always q(Y) [7 #].`	
$c = 1s$	$h(X,Y)$	$\leftarrow g(X,Y), \boxplus^{5}\Diamond p(X), \boxplus^{\#7}\Box q(Y)$
(2)	`@T h(X,Y) :- f(X), g(Y), @T p(X,Y) [4 min], not always q(Y) [2 #].`	
$c = 15s$	$@_T\, h(X,Y)$	$\leftarrow f(X), g(Y), \boxplus^{16}@_T\, p(X,Y), \text{not}\, \boxplus^{\#2}\Box q(Y)$
(3)	`h(Z) :- g(X,Y), p(X) [2 #], q(Y) [2 #], Z=X+Y, up(U), Z <= U.`	
$(c = 1s)$	$h(Z)$	$\leftarrow g(X,Y), \boxplus^{\#2}\Diamond p(X), \boxplus^{\#2}\Diamond q(Y), Z{=}X{+}Y, up(U), Z \leq U$
(4)	`@T h(X,Y) :- g(X), g(Y), @T q(X) [200 ms], not @T q(Y) [200 ms].`	
$c = 10ms$	$@_T\, h(X,Y)$	$\leftarrow g(X), g(Y), \boxplus^{20}@_T\, q(X), \text{not}\, \boxplus^{20}@_T\, q(Y)$

Table 7.2: Example Ticker rules and their translations to LARS

which appear neither as rule heads nor within the scope of windows are assumed to be background data.

Tuple windows: only on signals. Tuple windows in Ticker count and limit the signals of the stream, not inferences made by the program. As in Chapter 6, we thus restrict the use of tuple windows on extensional atoms. In other words, no predicate can occur both in a rule head and in the scope of a tuple window.

Safe negation. We use the usual safety of variables under negation. That is, every variable that occurs in the scope of negation must also occur outside the scope of negation (in the same body, to be explicit; variables are always local to rules). For instance, in rule `a :- g(X,Y), not b(X) [1 s], not c(Z)` (only) variable `Z` is not safe.

Guards. When the incremental reasoner is used, all variables, except time variables T within the form $@_T$, that occur in the scope of a window atom must occur outside the scope of a window (and outside negation). We call according atoms *guards*. They can be predicates of background facts but also rule heads, i.e., intensional. For instance, in Rule (1) of Table 7.2, `g(X,Y)` is a guard, as are `f(X)` and `g(X)` in Rule (2). Guards are required for a pre-grounding of all variables that do not stem from tick information (cf. Section 7.2.1).

Head variables. Every variable mentioned in the head must occur in the body. For instance, this rule is violated for variable `Z` in rule `a(X,Z) :- g(X,Y), b(X) [1 s]`.

Unique time variable per rule. Due to the pre-grounding procedure (cf. Section 7.2.1) employed for incremental reasoning, a rule can mention only a single time variable (which, however, can occur multiple times in the rule). For instances, Rules (2) and (4) could not be extended by any expression involving an @-atom with a variable different from `T`.

Arithmetic and relations. Variables in arithmetic expressions must also be guarded (as variables in the scope of windows). Since time variables are not guarded, they cannot be used within arithmetic operators and relations.

Clock time. All time windows must be compatible with the specified clock time (see runtime option `--clock` below).

```
-p --program <file>,<file>,... (mandatory)
-r --reasoner [ incremental | clingo ]
-f --filter [ none | inferences | <predicate>,<predicate>,...  ]
-c --clock <int><timeunit>
     <timeunit> = ms | s | sec | min | h
-e --outputEvery [ change | signal | time | <int>signals | <int><timeunit> ]
     specify when output is written:
     change: filtered model changed
     signal: new signal streamed in (push-based)
     time: a time point passed by (pull-based)
     <int>signals: given number signals streamed in (generalized push-based)
     <int><timeunit>: specified time passed by (pull-based)
-i --input <source>,<source>,...
     <source>: stdin | socket:<int>
-o --output <sink>,<sink>,...
     <sink>: stdout | socket:<int>
```

default: `-r incremental -f none -c 1s -e change -i stdin -o stdout`

Table 7.3: Overview of Ticker's runtime options. Each parameter can be alternatively specified with a short (-) or a long (--) descriptor.

Odd loops. Programs with odd loops are handled correctly only when one-shot solving with Clingo is used (see option `--reasoner clingo` below). There is no explicit syntax for constraints, i.e., all rules must have a non-empty head. As usual, a constraint of form `:- body` can be stated as `x :- body, not x`, using a fresh atom `x`, i.e., an odd loop of minimal length.

7.1.2 Configuration: Runtime Options

Table 7.3 enumerates all start-up parameters for Ticker. Each parameter can be specified following a short selector or its long version. Only the program parameter (`-p`, respectively `--program`) is mandatory; the others have default values.

Program: `-p --program <file>,<file>,...`. A LARS program can be loaded from multiple files, following the selector `-p` or, alternatively, `--program`. (Note that input files are separated only by a comma, no spaces are allowed.)

Example: `-p core.lars,domain.lars` starts Ticker for the program obtained by the set of rules found in files `core.lars` and `domain.lars`.

Reasoning strategy: `-r --reasoner [ incremental | clingo ]`. This parameter specifies one of the two available reasoning strategies. With option `clingo` Ticker uses Clingo in one-shot solving mode on the static ASP encoding (cf. Section 6.3). With configuration `incremental` Ticker uses JTMS (cf. Section 6.2) based on to the incremental

encoding (cf. Section 6.4). The implementations of both variants will be discussed further in Sections 7.3.2 and 7.3.3, respectively.

- Default: `-r incremental`.

Filter: `-f --filter [ none | inferences | <predicate>,<predicate>,... ]`. This parameters specifies the output, i.e., which predicates from the answer will be sent to the sinks. Filter `none` includes the entire answer, including current input signals. Limiting the output to intensional atoms is done with option `inferences`. Alternatively, the answer can be projected to a set of specific output predicates. For instance, `-f a,x` includes only atoms with predicates `a` and `x` in the output.

- Default: `-f none`.

Clock time: `-c --clock <int><timeunit>`. The argument specifies the clock time (Definition 38), i.e., the length of a time point. The default is one second (i.e., `1s` or `1sec`). Alternatively, a (positive natural) number of milliseconds (`ms`), minutes (`min`) or hours (`h`) can be specified as time unit, i.e., `<timeunit> = ms | s | sec | min | h`. For instance, `-c 250ms` specifies that a time point corresponds to 250 milliseconds.

- Default: `-c 1s`.

Note that any time window in the program implicitly specifies a number of time points, which is determined by this parameter. As a consequence, any size of a time window in the program must be a *multiple* of this parameter (when converted to the same unit). Otherwise, the engine will not be started.

Example: time window sizes `[1 s]` and `[3 s]` are compatible with `-c 500ms` and `-c 1s`, but not with `-c 2s`, respectively.

Output timing: `-e --outputEvery <timing>`. This parameter controls the timing of the output. The following options exist for `<timing>`:

- `change` will write an output whenever it changes (relative to the specified filter).
- `signal` returns a model after every new input. This amounts to a push-based evaluation.
- `time` returns a model every time a single time point passes, i.e., a unit of time due to the clock parameter `-c`. This amounts to periodic querying as in pull-based evaluation.
- `<int>signals` generalizes option `signal`; the latter amounts to `1signals`.
- `<int><timeunit>` likewise generalizes option `time` which amounts to `1U`, where `U` is the selected clock time. For instance, `-e 500ms` will write the output every 500 milliseconds. The specified length must be a multiple of the clock time.

- Default: `-e change`.

Input source: `-i --input <source>,<source>,...` Multiple input sources can be specified, including standard input `stdin` and sockets (`socket:<int>`).

- Default: `-i stdin`.

Output sink: `-o --output <sink>,<sink>,...` Likewise, standard out (`stdout`) and sockets (`socket:<int>`) can be used as output sinks.

- Default: `-o stdout`.

7.2 Incremental Encoding Revisited

We now describe the realization of Ticker's incremental reasoning mode and how it deviates from the presentation in Chapter 6. We recall the the main ideas of the incremental evaluation (cf. Section 6.4): at each tick (t, c) we obtain by Algorithm 6.10 (*IncrementalRules*) new rules that we have to add to the truth maintenance system (JTMS). Each rule is annotated with a tick (t_Δ, c_Δ), expressing the duration after which it has to be removed from the JTMS. Algorithm 6.11 (*IncrementTick*) then describes the expiration process. It first retrieves new incremental rules from Algorithm 6.10, then determines the expiration tick by adding for each rule the respective duration (t_Δ, c_Δ) to the current tick (t, c), and retrieves expiring rules from memory. Notably, *IncrementTick* assumes a grounding mechanism to obtain according ground instances.

Ticker realizes these algorithms with the following deviations. First, we use a *pre-grounding* step (cf. Section 7.2.1) that grounds all variables but tick variables and time variables, i.e., all variables that remain after the pre-grounding are reserved for a tick time t or a tick count c. The actual creation of incremental rules (based on *IncrementalRules*; Algorithm 6.10) uses pre-grounded rules and grounds the remaining tick variables. This step is called *pinning* and can be done efficiently. We present in Section 7.2.2 a variant of *IncrementalRules* that avoids an inflation of the incremental encoding. The resulting procedure *IncrementalRulesImpl* additionally expires rules that do not have to be deleted for semantic reasons. Moreover, it introduces some optimizations and already calculates the expiration tick (instead of returning annotation expressing the duration until expiration).

Section 7.2.3 then explains how the tick increment (from Algorithm 6.11) is realized, using ground rules from *IncrementalRulesImpl*.

7.2.1 Pre-grounding

In Algorithm 6.11, Line 5, we assume a grounder that instantiates pinned rules from Algorithm 6.10.[2] To provide according efficient techniques is a topic on its own. Thus, we restrict dynamic grounding to the pinning process in Algorithm 6.10, i.e., the replacement of tick variables by respective integers.

[2]When a rule is *pinned* it has no tick variables left. In Algorithm 6.11, pinned rules are non-ground in general. By contrast, due to pre-grounding, pinned rules in the implementation are ground.

To enable pre-grounding, we demand in each rule for every variable X in the scope of a window atom an additional *guard* atom that includes X. The guard is either background data or intensional. Based on this, the incremental rules in Algorithm 6.10 can be grounded upfront, apart from the tick variables $\dot{N}$ and $\dot{C}$ and time variables in @-atoms. We call such programs *pre-grounded*. A LARS program P is first translated into an encoding $\hat{P}$ with several data structures that differentiate Q, base rules R, and window rules W. During the initialization process, pre-groundings are prepared, where arithmetic expressions are represented by auxiliary atoms. During grounding, they are removed if they hold, otherwise the entire ground rule is removed.

Example 77 For rule $r = @_T high \leftarrow value(V), \boxplus^{30} @_T \alpha(V), V \geq 18$ of Example 31 (page 64), where $value(V)$ was added as guard, we get a base rule

$$\hat{r} = high_@(T) \leftarrow value(V), \omega_e(V,T), Geq(V,18)\,,$$

where $e = \boxplus^{30} @_T \alpha(V)$. Given facts $\{value(0), \ldots, value(25)\}$ (from background data or potential derivations), we obtain as pre-grounding the set of rules of form

$$high_@(T) \leftarrow value(x), \omega_e(x,T)\,,$$

where x is replaced by values from 18 to 25. ∎

We then use the pre-grounded program as replacement for the original input program, when considering the creation of incremental rules.

The employed technique comes with a limitation, however. Pre-grounding still leaves variables, i.e., time variables T from @-atoms of the form $@_T a$. For efficiency, the pinning process simply will replace any such T with the current time point, to obtain incremental rules as described in Algorithm 6.10, that are fully ground. This implies that we cannot use two (or more) distinct time variables per rule. For instance, consider rule $r = h \leftarrow \boxplus^3 @_T a, \boxplus^4 @_U b$, which is already pre-grounded. Its (incremental) encoding has a base rule of form $h \leftarrow \omega_a(T), \omega_b(U)$. For time 42, function *incrWindowRules* of Algorithm 6.10 will produce rule $\omega_a(42) \leftarrow a_@(42)$ for deriving ω_a, and likewise $\omega_b(42) \leftarrow a_@(42)$ for ω_b. However, we get no pinning for other values T and U, in particular, where $T \neq U$. We similarly only get $h \leftarrow \omega_a(42), \omega_b(42)$ as pinned instance of the base rule. Using the pinning mechanism, we consequently can use only a single time variable per rule, and no arithmetic operations/relations over time variables. Fixing this issue would be the first step towards incremental grounding on-the-fly, which is left for future work.

7.2.2 Incremental Translation

We now review Algorithm 6.10 (*IncrementalRules*) and develop an improved version as foundation for the implementation. The creation of incremental rules can be optimized in two ways. First, not all classes of programs require all auxiliary facts. Second, more importantly, we want to avoid an inflation of the encoding due to non-expiring rules (i.e.,

those with annotation (∞, ∞)), which can also be deleted when they become irrelevant: we observed in Theorem 28 that we can drop facts (that encode the tick stream) as soon as they are outside the reach of any window of the program. Moreover, rules that rely on such facts can be discarded as well.

Disjunctive and Conjunctive Annotations

We will analyze Algorithm 6.10 and derive a variant *IncrementalRulesImpl* that makes use of a conceptual reframing of duration and expiration annotations. In Algorithms 6.10 and 6.11, an annotation (t', c') expresses when a rule *must* be deleted to ensure the semantic correctness of the incremental encoding.[3] Accordingly, we made use of infinite annotations (∞, ∞) for rules that never expire in this sense. We now use annotations in a pragmatic view to state when a rule *will* be deleted; this way, optional removals can be captured as well. Moreover, we distinguish two classes of duration and expiration annotations: We call the ones from before *disjunctive*, since a rule annotated with expiration tick (t', c') will be deleted at tick (t, c) when $t \geq t'$ *or* $c \geq c'$ holds. Dually, we introduce *conjunctive* expiration annotations, denoted $[t', c']$, which state that the rule is deleted as soon as $t \geq t'$ *and* $c \geq c'$ hold. (The distinction between disjunctive and conjunctive annotations is likewise used for duration ticks.)

We now revisit procedure *IncrementalRules*, analyzing the potential for optimization.

Algorithm 6.10 Revisited

Line 1. As mentioned earlier, facts $\mathit{tick}(t, c) \leftarrow$ need to be included only when window atoms of the form $\boxplus^{\#m}\Box a(\mathbf{X})$ occur in the program. If $m^*_\Box$ is the maximal size of such a window, then we can delete $\mathit{tick}(t, c) \leftarrow$ after $m^*_\Box$ additional atoms have streamed in. Thus, we can use the existing deletion mechanism for expiration, taking the annotated rule $\langle(\infty, m^*_\Box), \mathit{tick}(t, c) \leftarrow\rangle$ instead.

Line 2. This line addresses the encoding of the data stream at a count increment. We distinguish three cases.

(a) Consider first time-pinned atoms (of form $a_@(\mathbf{x}, t)$) and assume that only time windows occur in the program, with a maximal size n^*. The maximal duration annotated in any incremental window rule is then (n^*+1, ∞). Consequently, after n^*+1 time points, no incremental window rule can access the fact anymore, unless the fact is mentioned as @-atom outside the scope of any window.

Example 78 Consider the rule $r = a \leftarrow \boxplus^3@_T\, b, @_T\, c$, where atoms b and c are streaming in. Window atom $e = \boxplus^3@_T\, b$ gets the incremental window rule $\langle(4, \infty), \omega_e(t) \leftarrow b_@(t)\rangle$. This means that r has to be deleted 4 time points after creation. For instance, assuming atom b appears at $t = 20$, window atom $\boxplus^3@_{20}\, b$ (respectively encoded atom $\omega_e(20)$) holds only until $t = 23$, i.e., at $t = 24$ the rule $\omega_e(20) \leftarrow b_@(20)$ must be deleted. Assuming no further rules access b we conclude that we can also delete fact $b_@(20) \leftarrow$ at

[3] Procedure *IncrementalRules* (Alg. 6.10) uses duration annotations, which are transformed to expiration annotations in *IncrementTick* (Alg. 6.11).

any time $t \geq 24$. (Formally speaking, the mapping $20 \mapsto b$ has to be contained in the answer stream as long as we consider a timeline that contains time point 20; but from a practical point of view, we will not be interested in displaying historic data that does not lead to any derivations.) For body element $@_T c$, however, the case is different. Since access to T is not bounded by a window, we in general need to store all auxiliary facts of form $c_{@}(t) \leftarrow$. ∎

Envisaged applications will likely target only recent information and thus will usually use some upper bound for the history of data. Nevertheless, we allow "unbounded" @-atoms in rule bodies (as $@_T\, c$ in the above example). Any such occurrence $@_T a(\mathbf{X})$ in a rule body will be viewed as abbreviation for $\boxplus^{\infty} @_T(\mathbf{X})$.

If no tuple windows occur in the program, we thus conclude that using instead $\langle (n^* + 1, \infty), a_{@}(\mathbf{x}, t) \leftarrow \rangle$ in Line 2 would work for the time-pinned atom, where n^* is the maximal window length used on predicate a (including the convention $n^* = \infty$ for unbounded @-atoms).

(b) Let us now dually consider the case that only tuple windows occur in the program. Tick-pinned atoms of the form $a_{\#}(\mathbf{x}, t, c)$ are easier to analyze since they do not appear as rule heads. Intuitively, if m^* is the maximal size of any tuple window, it should be possible to delete auxiliary fact $a_{\#}(\mathbf{x}, t, c) \leftarrow$ after m^* new atoms, i.e., the annotation (∞, m^*). However, in case the program contains the tuple-box combination (i.e., a tuple window followed by $\Box$), we need to ensure that tick-pinned atoms remain at least until the next time point, yielding the conjunctive duration annotation $[1, m^*]$. We will explain this later (cf. Example 81).

Notably, for the tuple-box combination we also need the time-pinned atom $a_{@}(\mathbf{x}, t)$. Since tuple windows are used only on extensional data, the (negated) body element $a_{@}(\mathbf{x}, t)$ (in the first spoiler rule, Line 15) relies on a fact from Line 2. Any such fact must be present when it is still in the range of the last m atoms (as specified in $e = \boxplus^{\#m} \Box a(\mathbf{X})$). That is to say, when we add an auxiliary fact $a_{@}(\mathbf{x}, t)$ then we need to keep it for the next $m^*_{\Box}$ atoms. Without the presence of time windows, only this tuple-box combination needs it, hence it suffices to keep it for $m^*_{\Box}$ more signals. We thus use the annotation $(\infty, m^*_{\Box})$.[4]

(c) A subtle case arises for tuple windows with $\Box$ when also time windows are present. In this case, the time-pinned atom $a_{@}(\mathbf{x}, t)$ must also be available for the next n^* time points. Crucially, we cannot simply revise the annotation for facts $a_{@}(\mathbf{x}, t)$ to $(n^*, m^*_{\Box})$: at current tick (t, c), a rule r with expiration tick (t', c') will be expired (and deleted) if $t \geq t'$ *or* $c \geq c'$. By contrast, we require here that $a_{@}(\mathbf{x}, t)$ is deleted at (t, c) if and only if $t \geq t'$ *and* $c \geq c'$. Thus, we need a conjunctive duration annotation $[n^*, m^*_{\Box}]$.

Line 6. The base rule (which encodes the original LARS rule) also has the annotation (∞, ∞). In principle, we can also delete it as soon as we have a guarantee that the body cannot hold anymore. In light of our pre-grounding, it makes sense to keep those base

[4] For simplicity, we spare a further separation of maximal tuple window sizes for different predicates used in their scope. In that regard, using m^* instead of $m^*_{\Box}$ would also suffice to ensure that tick stream facts are eventually deleted.

rules which do not contain time arguments: optimizing the total number of stored base rules is only relevant when they can grow indefinitely. This is the case when the LARS rule r contains an @-atom, then the base rule $\hat{r}$ will have a time parameter.

Example 79 Consider the LARS rule $r = x \leftarrow \boxplus^3 @_T\, y, \boxplus^{\#2} @_T\, z$, which is translated to a base rule of form $\hat{r} = x \leftarrow \omega_y(T), \omega_z(T)$. The instantiations $\omega_x(t)$ and $\omega_y(t)$ at a tick (t, c) (derived by incremental window rules of Lines 8 and 12, resp.) will expire after durations $(3 + 1, \infty)$ and $(\infty, 2)$, respectively. Thus, using an "joint" annotation $(4, 2)$ for $\hat{r}$ would work in this case: $\hat{r}$ cannot fire as soon as the time or count expires in which the respective window atom necessarily fails to hold. ∎

In fact, for window atoms of the forms $\boxplus^n @_T\, a(\mathbf{X})$ and $\boxplus^{\#m} @_T\, a(\mathbf{X})$, respectively, and respective maximal values $n^*_@$ and $m^*_@$ occurring in the positive body of rule r, we can employ the annotation $(n^*_@ + 1, m^*_@)$ for the base rule $\hat{r}$; where not applicable, we use ∞. Note that the disjunctive semantics of expiration (i.e., time *or* count) applies: as soon as the current tick surpasses *any* of the two tick dimensions, the entire (positive) body cannot hold anymore.

Example 80 Consider rule $r = @_{T_1} x \leftarrow \boxplus^7 @_{T_1}\, y_1,\ \boxplus^9 @_{T_2}\, y_2,\ \boxplus^{13} \Diamond y_3,\ \text{not}\, \boxplus^{\#5} @_{T_1}\, z$. We annotate base rule $\hat{r} = x_@(T_1) \leftarrow \omega_1(T_1),\ \omega_2(T_2),\ \omega_3,\ \text{not}\, \omega_4(T_1)$ with duration $(10, \infty)$ since 9 is the maximal size of a time window (in the positive body), and no tuple window with an @-atom occurs positively. ∎

Note that we do not explicitly cover the case where all window atoms with @-atoms are negated. If they contain variables, these variables must occur (for safety reasons, cf. Section 7.1.1) also in the positive body, which then determines the expiration. Using in rules @-atoms of the form $@_t a(\mathbf{X})$, where t is ground, could be solved easily in addition; however, these cases are only of theoretical interest (where (∞, ∞) suffices), so we refrain from further elaboration.

Lines 10 and 14. We observe that these rules do not contain a tick parameter. Apart from tick numbers, we consider a finite universe from which we obtain the pre-grounding. Consequently, the ground rules based on these lines are fully determined by the pre-grounding and can thus remain, i.e., we keep annotation (∞, ∞).

Lines 15 and 16. For the first spoiler rule (Line 15) we observe that $\mathit{covers}^\tau_e(t)$ does not hold after m more atoms have streamed in due to the expiration duration (∞, m) of the rule that derives it. Consequently, we can explicitly expire also this spoiler rule. The case for the second spoiler rule, which deals with more than m tuples at a single time point, is more subtle.

Example 81 Consider the tick stream where the first signals are b, c, d at time 5 (in that order), and the single rule $h \leftarrow \boxplus^{\#2} \Box b$. The tuple window selects the timeline $[5, 5]$ and, due to the order of appearance, only atoms c and d. Hence $\boxplus^{\#2} \Box b$ is false, and h is

not derived. In addition to the base rule of form $h \leftarrow \omega_b$,[5] the relevant incremental rules at tick $(5, 3)$ are the following:

$$
\begin{array}{lrcl}
(r_1) & \omega_b & \leftarrow & b, \text{not } \mathit{spoil}_b \\
(r_2) & \mathit{spoil}_b & \leftarrow & b_{\#}(5,1), \mathit{covers}_b^{\tau}(5), \text{not } \mathit{covers}_b^{\#}(1) \\
(r_3) & \mathit{covers}_b^{\tau}(5) & \leftarrow & \mathit{tick}(5,2) \\
(r_4) & \mathit{covers}_b^{\tau}(5) & \leftarrow & \mathit{tick}(5,3)
\end{array}
$$

At tick $(5, 3)$, Rule (r_1) must not fire. Since b holds (at all ticks for time point 5), spoil_b must be derived, namely due to the spoiler rule that covers the case where the atom in the evaluation of at the single time point that is returned by the window, but not within reach with respect to counting. Accordingly, Rule (r_2) needs to fire, i.e., $b_{\#}(5, 1)$ must still be available, $\mathit{covers}_b^{\tau}(5)$ still derivable, and $\mathit{covers}_b^{\#}(1)$ not. The derivation of the cover for time point 5 is ensured by Rules (r_3) and (r_4) (which are added to the JTMS along with signals c and d, respectively); and the rule $\mathit{covers}_b^{\#}(1) \leftarrow \mathit{tick}(5, 1)$, inserted with signal b, expires after a duration $(\infty, 2)$, hence, it is no longer present in the encoding. However, Rule (r_2) must still be present as well as the auxiliary fact $b_{\#}(5, 1)$. That is, during the time point at which they are inserted, they must not expire; after the first time increase they can be expired, as soon as $m = 2$ new signals streamed in. Consequently, we need the conjunctive duration annotation $[1, m]$. ∎

Example 81 illustrated the extremal case for the tuple-box encoding where more than m atoms (where m is the size of the window) stream in at a single time point. In essence, we can expire auxiliary tick-pinned facts as well as instances of the second spoiler rule (Line 16) as soon as there was a time increase and the next m^* signals streamed in, yielding the conjunctive annotation $[1, m^*]$. If the tuple-box combination does not occur in the program, we can use (∞, m^*) for the tick-pinned atoms.

Conclusion of the algorithm review. The above considerations allow us to revise procedure *IncrementalRules* in a way that does not lead to an inflation of the incremental encoding over time. Assuming a program that does not access the entire history (i.e., by a window of infinite size, or some $@_T a(\mathbf{X})$ that is not in the scope of a window), all encoded rules that are added during incremental evaluation can be deleted at some point (as opposed to static ones where pre-grounding is fully ground); and in particular the facts for the tick stream encoding. Any stream signal that cannot be reached anymore can as well be deleted, and likewise any rule whose body will remain false. Towards an improved version of *IncrementalRules* we present two algorithms. First, Algorithm 7.1 gives a more involved tick stream encoding that replaces Lines 1-2 in *IncrementalRules*. Following the above analysis, we only use those auxiliary facts that are needed for the given program, and we do not keep these facts in memory longer than necessary.

The new tick stream encoding is then used in Algorithm 7.2 (*IncrementalRulesImpl*), i.e., the revision of the incremental rule generation. It defines additional expiration ticks when rules become redundant.

[5] We abbreviate subscript $e = \boxplus^{\#2}\Box b$ (e.g. for the window atom encoding ω_e) by b.

Algorithm 7.1: *TickStreamEncoding*(t, c, Sig)

Global: Pre-grounded LARS program P, maximal size n^* (resp. m^*) in a time (resp. tuple) window, maximal size $m^*_\Box$ in a tuple window followed by $\Box$.
Input: Tick time t, tick count c, signal set *Sig* with at most one signal; empty iff (t, c) is a time increment
Output: Taylored tick stream encoding as required by the program

```
1  F := ∅
2  if ⊞^{#m}□ occurs in P
3      F := F ∪ {⟨(∞, m*_□), tick(t, c) ←⟩}
4  if Sig ≠ ∅
5      let a(x) ∈ Sig
6      F := {⟨(1, ∞), a(x) ←⟩}
7      if ⊞^{#m} occurs in P
8          if ⊞^{#m}□ occurs in P
9              F := F ∪ {⟨[1, m*], a_#(x, t, c) ←⟩}   //conjunctive annotation
10             if ⊞^n occurs in P
11                 F := F ∪ {⟨[n*+1, m*_□], a_@(x, t) ←⟩}   //conjunctive annotation
12             else
13                 F := F ∪ {⟨(∞, m*_□), a_@(x, t) ←⟩}
14         else
15             F := F ∪ {⟨(∞, m*), a_#(x, t, c) ←⟩}
16             if ⊞^n occurs in P
17                 F := F ∪ {⟨(n*+1, ∞), a_@(x, t) ←⟩}
18     else if ⊞^n occurs in P
19         F := F ∪ {⟨(n*+1, ∞), a_@(x, t) ←⟩}
```

Improving the Tick Stream Encoding: Algorithm 7.1 (*TickStreamEncoding*)

Procedure *TickStreamEncoding*, shown in Algorithm 7.1, uses some information available from static program analysis, i.e., the maximum sizes n^* and m^* of time and tuple windows, respectively. Moreover, we provide in value $m^*_\Box$ the maximal size of a tuple window that is followed by modality $\Box$.

We initialize the set F of tick stream facts in Line 1 and add in Line 3 the tick information (fact $tick(t, c) \leftarrow$) only if needed, i.e., when there is a tuple window in the program P, followed by $\Box$. The duration annotation states that the fact can be deleted after $m^*_\Box$ new signals have streamed in. The remaining lines deal with encoding the signal in case of a count increment, i.e., where $|Sig| = 1$. The fact $a(\mathbf{x}) \leftarrow$ must be included in the encoding for the duration of one time point (Line 6), as in Algorithm 6.10. For time-pinned and tick-pinned facts we introduce new case distinctions. In the block starting with the condition of Line 7 we consider the case that a tuple window exists. We then further distinguish whether any tuple window is followed by modality $\Box$ or not

in Lines 8 and 14, respectively. Within each block we then distinguish further whether their exists a time-based window or not, and we also test the latter condition in Line 18 for the case that there is no tuple window. All these cases differ in the set of required auxiliary facts (for time-pinned atoms and tick-pinned atoms) and according duration annotations as discussed above. Notably, conjunctive annotations are only required for the tuple-box case.

Remark. We note that for each disjunctive duration annotation of form (t, ∞), we could equivalently use the conjunctive annotation $[t, 0]$ to express the irrelevance of the count dimension. Here, 0 expresses that the rule *can* expire immediately with respect to count increments, and thus *will* expire after t time increments. The case is analogous for (∞, c) and $[0, c]$. The chosen representation is closer to the implementation, where disjunctive annotations are easier to manage since both dimensions can be dealt with independently (also when both dimensions are finite): we maintain two lookup maps, one for time and one for counts, where we collect rules expiring at the same time (resp. count) increment. Thus, the notation for the conjunctive annotation in Line 11 indicates that an additional mechanism is needed, as will be discussed below.

Improving Incremental Rule Generation: Algorithm 7.2 (*IncrementalRulesImpl*)

With the revised tick stream encoding in place, we proceed by discussing Algorithm 7.2 (*IncrementalRulesImpl*) that further improves the incremental rule generation.

After retrieving in the first line the stream facts from *TickStreamEncoding*, we collect in set Q rules that express the equivalence between atoms $a(\mathbf{x})$, holding now, and time-pinned atoms $a_@(\mathbf{x}, t)$, if t is the current time. We need such rules only for predicates that are in the scope of some $\boxplus$ or @ (Line 4). (The presentation with the loop is conceptual; the implementation will compute static information such as the considered subset of Q only once.)

Next, we collect in Line 5 all incremental rules R by a refined function *incrAspRules′*. It differs from the original one in two aspects. First, we calculate for the given LARS rule r a duration annotation $(n_r^* + 1, m_r^*)$ for the base rule (Lines 9-12) by the maximum size n_r^* in a time window of the positive body of r, and likewise the maximal size m_r^* in a tuple window. (Note that this is the only place where we use a disjunctive annotation where both dimension are finite.) We keep the generation of incremental window rules from Algorithm 6.10, with the exception of the tuple-box case, where we now also expire the spoiler rules in Lines 16 and 17. The first spoiler rule (Line 16) can always be safely deleted after m new input signals, whereas the second one cannot be deleted at the same time where it was added; we discussed this corner case in Example 81. Consequently, the spoiler rules get annotated with (∞, m) and $[1, m]$, respectively. Notably, we keep annotation (∞, ∞) for the rule in Line 15 since this rule is fully ground after the pre-grounding step; likewise for the dual time-box combination in Line 10 of Algorithm 6.10.

Finally, we call (directly before returning) in Line 6 the function *dur2exp* on the obtained annotated rules $F \cup Q \cup R$ that converts duration annotations to expiration annotations. To this end, the current tick is added componentwise to the given duration. The case distinction between the standard disjunctive annotations (Line 23) and the

Algorithm 7.2: *IncrementalRulesImpl*(t, c, Sig)

Global: Pre-grounded LARS program P
Input: Tick time t, tick count c, signal set Sig with at most one signal; empty iff (t, c) is a time increment
Output: Pinned incremental rules annotated with expiration tick

$F := TickStreamEncoding(t, c, Sig)$
$Q := \emptyset$
foreach *predicate* a *in* P *in the scope of* $@$ *or* $\boxplus$
  $Q := Q \cup \{\, \langle (1, \infty), a(\mathbf{x}) \leftarrow a_{@}(\mathbf{x}, t)\rangle,\ \langle (1, \infty), a_{@}(\mathbf{x}, t) \leftarrow a(\mathbf{x})\rangle \}$
$R := \bigcup_{r \in P} incrAspRules'(r)$
return $dur2exp(F \cup Q \cup R)$

---

defn $incrAspRules'(r)$
  $n_r^* := \infty;\ m_r^* := \infty$
  $N := \{n \mid \boxplus^{n} @_T a(\mathbf{x}) \text{ occurs in } B^+(r)\}$
  $M := \{m \mid \boxplus^{\#m} @_T a(\mathbf{x}) \text{ occurs in } B^+(r)\}$
  if $N \neq \emptyset$ **then** $n_r^* := \max(N)$
  if $M \neq \emptyset$ **then** $m_r^* := \max(M)$
  return $\{\langle (n_r^*{+}1, m_r^*), baseRule(r)\rangle\} \cup \bigcup_{e \in B(r)} incrWindowRules'(e)$

defn $incrWindowRules'(e, t, c) =$ **match** e
  case $\boxplus^{\#m} \Box a(\mathbf{x}) \implies \{\, \langle (\infty, \infty),\ \omega_e(\mathbf{x}) \leftarrow a(\mathbf{x}), \text{not } spoil_e(\mathbf{x})\rangle \,\} \cup$
    $\{\, \langle (\infty, m),\ spoil_e(\mathbf{x}) \leftarrow a(\mathbf{x}), tick(t, c), covers_e^{\tau}(t), \text{not } a_{@}(\mathbf{x}, t)\rangle \,\} \cup$
    $\{\, \langle [1, m],\ spoil_e(\mathbf{x}) \leftarrow a_{\#}(\mathbf{x}, t, c), covers_e^{\tau}(t), \text{not } covers_e^{\#}(c)\rangle \,\} \cup$
    $\{\, \langle (\infty, m),\ covers_e^{\tau}(t) \leftarrow tick(t, c)\rangle,\ \langle (\infty, m),\ covers_e^{\#}(c) \leftarrow tick(t, c)\rangle \,\}$
  //other cases as in *incrWindowRules* (in Alg. 6.10)

defn $dur2exp(R)$
  $R' := \emptyset$
  foreach $\langle d, r\rangle \in R$
    if $d = (t', c')$ **then** $R' := R' \cup \{\langle (t + t', c + c'), r\rangle\}$
    else if $d = [t', c']$ **then** $R' := R' \cup \{\langle [t + t', c + c'], r\rangle\}$
  return R'

conjunctive ones (Line 24) is only for notational reasons; conceptually it is the same operation.

We emphasize that it is essentially the tuple-box combination that complicates the presented refinements *TickStreamEncoding* and *IncrementalRulesImpl*. Excluding window atoms of form $\boxplus^{\#m}\Box a(\mathbf{x})$ would lead to a much simpler encoding; conjunctive annotations and some case distinctions are only needed for them. While probably of less importance in practice, they shed some light on the limitation of the presented approach regarding dependencies between counting time and atoms on the mere technical side. Tuple windows followed by $\Box$ are not only conceptually the most involved case, they are also the most expensive computationally (when reasoning incrementally); this will be shown in the empirical evaluation below (cf. Section 7.4).

7.2.3 Incremental Evaluation

Algorithm 6.11 (*IncrementTick*) presented a fully incremental procedure to update the ASP encoding tick by tick. With ground incremental rules available from Algorithm 7.2 (*IncrementalRulesImpl*), we can revise the procedure and directly update the JTMS. The resulting method *IncrementTickImpl* is shown in Algorithm 7.3 and works as follows.

We assume as a precondition that the tick stream is now at tick (t, c) and that the JTMS reflects the state of the previous tick, when procedure *IncrementTickImpl* is called. At the end of the procedure, the state of the JTMS reflects tick (t, c).

The algorithm has two parts: Lines 1-16 are concerned with deleting expired rules, and Lines 17-25 add new incremental rules to the JTMS. We control expiration by maps that associate with time points t, respectively tick counts c, the set of rules which must (respectively can) expire due a disjunctive annotation (t, c) (respectively conjunctive annotation $[t, c]$). That is, a rule r expires at tick (t, c), if it is contained in $X_\tau(t)$ or $X_\#(c)$ (disjunctive case), or if it is in $X'_\tau(t)$ and in $X'_\#(c')$ for a previous count $c' < c$ or vice versa, i.e., in $X'_\#(c)$ and in $X'_\tau(t')$ for an earlier time point $t < t'$. To track the respective second condition, we use a set Y of rules that maintains candidates of rules that can be released due to exactly one dimension of a conjunctive annotation.

The algorithm then works as follows. To determine rules expiring at current tick (t, c), we first initialize in Line 1 an empty set G^- of rules to be removed and a placeholder X' for a map from which (potential) expiring rules from conjunctive annotations will be retrieved. As before, the condition "$Sig = \emptyset$" in Line 2 tests whether (t, c) is a time increment. In this case, we retrieve in Line 3 rules from $X_\tau(t)$, i.e., the rules expiring at time t (independently from the tick count c). The key t is then deleted from X_τ. For further expirations based on conjunctive information below, we already select the time-based map X'_τ (Line 5). In case of a count increment, we similarly retrieve rules to be expired at count c from map $X_\#$ (in Line 7), followed by removing the entry from the map, and the selection of the count-based map $X'_\#$.

We then iterate in Lines 10-14 over rules that expire if both time and count exceed their respective thresholds, as specified in a conjunctive expiration annotation. We observe that at each tick, we can only expire due to a change in time or a change in count (but not both). Accordingly, we already selected the map X' which will retrieve

Algorithm 7.3: *IncrementTickImpl*(*Sig*)

State: current tick (t, c), JTMS reflecting previous tick, maps $X_\tau / X_\#$ (resp. $X'_\tau / X'_\#$) holding expiring rules per time point/count due to disjunctive (resp. conjunctive) expiration annotations; set Y of rules ready for removal due to a single tick dimension
Input: signal set *Sig* with at most one signal; empty iff (t, c) is a time increment
Result: JTMS holds model at (t, c); X_τ, $X_\#$ are updated

```
1  G⁻ := ∅; X' := nil
2  if Sig = ∅
3  |  G⁻ := X_τ(t)
4  |  X_τ(t) := nil   //delete key t in map
5  |  X' := X'_τ
6  else
7  |  G⁻ := X_#(c)
8  |  X_#(c) := nil   //delete key c in map
9  |  X' := X'_#
10 foreach r ∈ X'(r)
11 |  if r ∈ Y
12 |  |  G⁻ := G⁻ ∪ {r}
13 |  |  Y := Y \ {r}
14 |  else Y := Y ∪ {r}
15 foreach r ∈ G⁻
16 |  remove(r)   //JTMS (Alg. 6.7)
17 G⁺ := IncrementalRulesImpl(t, c, Sig)
18 foreach ⟨e, r⟩ ∈ G⁺
19 |  add(r)   //JTMS (Alg. 6.1)
20 |  if e = (t', c')
21 |  |  if t' < ∞ then X_τ(t') := X_τ(t') ∪ {r}
22 |  |  if c' < ∞ then X_#(c') := X_#(c') ∪ {r}
23 |  else if e = [t', c']
24 |  |  X'_τ(t') := X'_τ(t') ∪ {r}
25 |  |  X'_#(c') := X'_#(c') ∪ {r}
```

rules that can expire due to the time dimension (i.e., from X'_τ in case (t, c) is a time increment) or due to the count dimension (i.e., from $X'_\#$ in case of a count increment). Consequently, if a rule r can expire now due the currently considered dimension (Line 10) and it is already contained in Y (Line 11), then the criterion $t \geq t'$ *and* $c \geq c'$ holds, and we put r to G^- (Line 12), i.e., the set of rules to be removed. We then also delete it from Y (Line 13). Otherwise, we now encountered the first dimension that permits expiration of r, so we store this information by putting r to Y (Line 14). Finally, we

remove in Line 16 all rules of G^- from the JTMS, calling Algorithm 6.7.

The second part for new incremental rules starts in Line 17, where we retrieve (fully) ground incremental rules with expiration annotations from *IncrementalRulesImpl*. In the loop of Lines 18-25 we first add each obtained rule r to the JTMS, using Algorithm 6.1. We then prepare the later expiration of r due to a case distinction on the kind of the expiration annotation e. The usual case, covered in Lines 21-22, deals with disjunctive annotations (t', c'). We put the rule r in maps X_τ or $X_\#$ to register it for disjunctive expiration for finite values of according tick dimensions. Dually, we put r in the maps for conjunctive expiration in case of a conjunctive annotation in Lines 24 and 25.

Since r has to be deleted when for the current tick (t, c) it holds that $t \geq t'$ or $c \geq c'$, we use two maps that associate integers with sets of rules: X_τ stores r as expiring rule for time point t' and $X_\#$ likewise for count c'. (In either case we do so only if the respective expiration dimension is finite.) Dually, if the annotation is conjunctive (i.e., of form $[t', c']$), we similarly put them in maps X'_τ and $X'_\#$ (in Lines 24 and 25). The case is only used (and meaningful) when both t' and c' are finite, hence we spare according checks.

As observed for Algorithm 7.2, *IncrementTickImpl* would be considerably easier when disregarding the tuple-box combination. In this case, the algorithm would be about half the size, sparing all expiration control for conjunctive annotations, i.e., Lines 5, 9-14, 20, and 23-25.

The tick increment is the core of the incremental evaluation mode of the Ticker engine. We now proceed to a discussion of its implementation.

7.3 Implementation

We are now going to describe central components of the Ticker engine's implementation. In Section 7.3.1, we start with an overview of the reasoner architecture. We then explain in more detail the implementation of ASP-based reasoning in Section 7.3.2 and the incremental reasoning mode in Section 7.3.3.

7.3.1 Architecture

The core concept of the implementation is the *reasoner* entity, as shown in Figure 7.1. Classes not shown in this diagram serve to set up the specific reasoner instance, preparing the utilized LARS program encoding, receiving input from (at least one) sink, repeatedly invoking the reasoner, and sending the result to the output sinks.

When Ticker is started, the main procedure reads the input arguments (cf. Section 7.1.2), parses the specified Ticker program, creates an according LARS program representation and a configuration object for the remaining arguments. It then starts the engine, which handles the processing of input and output due to the given clock time and output timing. It initializes the logical time point with 0 and increases it by 1 every time the duration of the clock time passes. The engine is the runtime environment whose single job is to send new signals to selected reasoning method and retrieve the computed model based on the output timing. From the view point of the execution, all

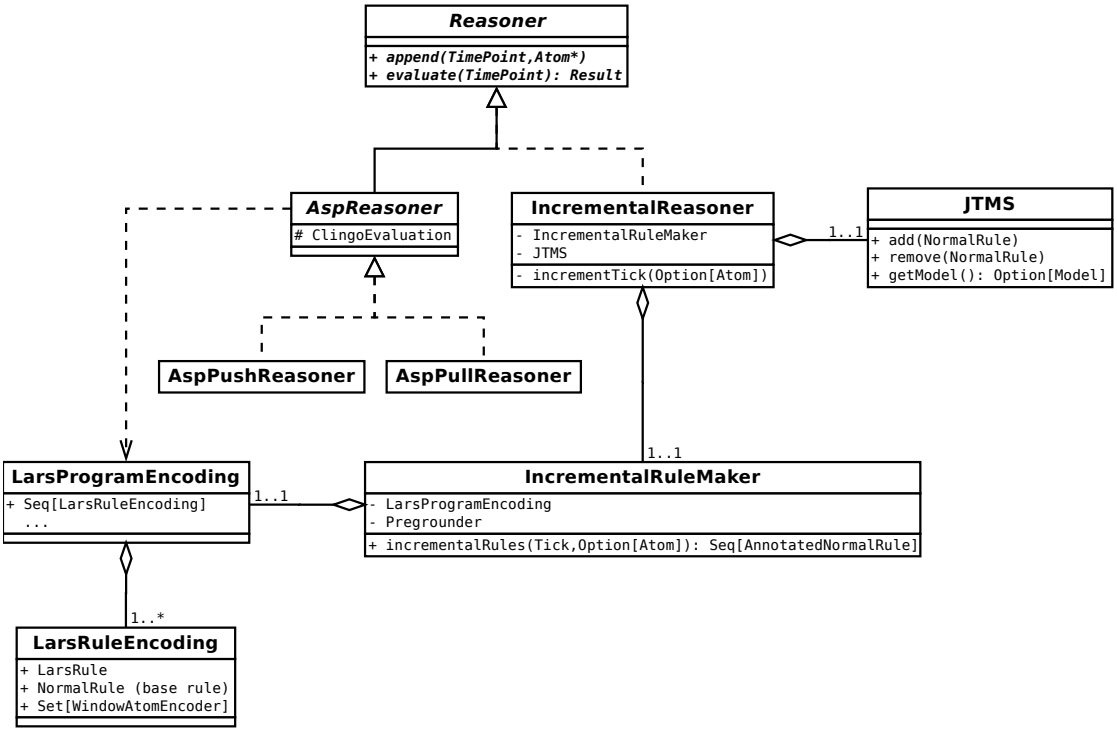


Figure 7.1: Ticker's core reasoning architecture

algorithmic details (and differences) are abstracted away in the `Reasoner` trait,[6] as shown in Figure 7.1.

```
trait Reasoner {
  def append(timepoint: TimePoint, signals: Atom*)
  def evaluate(timepoint: TimePoint): Result
}
```

Method `append`, which has no return type,[7] takes a `TimePoint` and stream signals in form of `Atom` objects. The notation `Atom*` indicates a sequence of `Atom` objects of arbitrary length (including zero). A `TimePoint` is essentially an integer (as in LARS) but represented explicitly as specific kind of numeric argument class, since time points also occur in @-atoms. The second method, `evaluate`, serves to return the output at the given time point. Its return type is `Result` which encapsulates an optional `Model`; the latter is a type alias for `Set[Atom]`, i.e., a set of objects of type `Atom`.[8]

[6] Scala traits are interfaces that may include (partial) implementations and can be used in well-defined multiple inheritance.

[7] More precisely, the return type is `Unit`, which can be omitted.

[8] Recall that using Clingo we can use programs with odd loops that may not have any answer stream. Such a case is then reflected in a `Result` object with *no* model. Scala has an explicit representation `Option[T]` for optional values of type `T`. If they are present, they materialize as `Some[T]`, else as `None`. This way, the often seen implicit, error-prone representation in form of a null value can be avoided.

Different classes of reasoners are provided, i.e., for repeated one-shot solving with Clingo and for incremental reasoning. Each reasoner can be wrapped by an instance of class `ReasonerWithFilter` that creates an output due to the specified filter in a post-processing step.

7.3.2 ASP Reasoner

Using repeated one-shot solving, Ticker repeatedly invokes the state-of-the-art ASP solver Clingo [GKKS14] on the static encoding (cf. Section 6.3). Reasoning with Clingo is a practical choice when the program is guaranteed to have a single model where from a purely semantic perspective the question of model update does not arise. Moreover, programs with odd loops can be handled only with Clingo, since the JTMS update procedure employed for the incremental reasoning mode cannot guarantee the stable semantics in this case (cf. Section 6.2.4). When Clingo reports multiple models, we simple take the first one.[9]

- `AspReasoner` is the trait that serves as abstraction for reasoning with Clingo. It extends `Reasoner` and requires only the following evaluation component.
- `ClingoEvaluation` is a class that contains a representation for the static ASP program (cf. Algorithm 6.9) that can be translated to Clingo input syntax. Moreover, it has a method that invokes Clingo (given the tick stream encoding; cf. page 142) and returns its `Result`.

Depending on the output timing, one of two classes, implementing the `AspReasoner` trait, is instantiated. They implement different policies when model computation with Clingo is invoked (using `ClingoEvaluation`). We recall that *push-based* processing typically refers to the behaviour of an engine that immediately reacts to incoming data, while *pull-based* reasoning is characterized by computing solutions in certain intervals of fixed length.

- `AspPushReasoner` is the class to that realizes push-based evaluation. It is used when output timing is `change` or `signal` (or `1signals`). This reasoner computes the model after every append call, which is then stored and replaced by another call to append, respectively returned by a call to `evaluate`.
- `AspPullReasoner` implements pull-based evaluation, which is employed for remaining output timings (i.e., `time`, `<int>time` and `Nsignals`, where `N` is greater than 1). In this case, model computation is carried out when `evaluate` is called by the engine. Method `append` only collects received data (with timestamps) in a buffer.

Pull-based evaluation saves unnecessary model computations for time points where the model is not needed, i.e., where no output is written anyway. For instance, consider a clock time of 100 milliseconds (`--clock 100ms`) and an output timing of 1 second

[9] Note that we do not employ the multi-shot features of Clingo, which are not applicable due to limitations regarding grounding and rule removal (see Section 7.5 for further discussion).

(`--outputEvery 1s`) and a stream that sends 5 signals per time point (100ms). A push-based evaluation mode would result in 50 model computations, among which 49 are irrelevant. Thus, we can save the computation time using pull-based evaluation. In some cases, this may give a practical solution if model computation is fast enough to fit in the intervals of the output timing but too slow to be computed at every tick. The model can then be written to the output sinks with some delay but still fast enough in terms of providing a correct sequence of results.

7.3.3 Incremental Reasoner

Our incremental approach naturally falls into the category of push-based processing, since every change in terms of time or data (i.e., new signals or expiring signals) may result in a changed model. Due to the fully incremental nature of the computation it is not possible to skip processing of any change and there is no advantage in delaying. We consequently provide a single reasoner implementation for this evaluation mode, realizing the algorithms of Section 7.2.

The resulting `IncrementalReasoner` is initialized with an `IncrementalRuleMaker` and a `JTMS` instance, i.e., an implementation of the extended justification-based truth maintenance system.

- `IncrementalRuleMaker`. This object has a method called `incrementalRules`, which realizes Algorithm 7.2 (*IncrementalRulesImpl*). The `IncrementalRuleMaker` is initialized with a `LarsProgramEncoding` and a `Pregrounder`; the latter substitutes all variables in the encoding that are not tick variables by potential constants available from background facts and intensional atoms. Pre-grounding is a pre-processing step run once during the start-up of the engine, after the input program has been parsed. The `LarsProgramEncoding` contains a representation of the LARS program and the mechanisms to create incremental rules. More specifically, it contains a sequence of `LarsRuleEncoding` objects, each of which contains (i) the original LARS rule, (ii) the corresponding normal ASP rule (the base rule), and (iii) a set of `WindowAtomEncoder` objects; they have methods to create incremental rules as described in function *incrWindowRules′* of Algorithm 7.2.

- `JTMS`. This is the realization of the truth maintenance system as described in Section 6.2, i.e., it has a method `add` to include a new rule in the network (Algorithm 6.1) and `remove` for dual rule removal (Algorithm 6.7). Moreover, a method `getModel` is available that returns the current model, i.e., the atoms with label *in*.

When the `IncrementalReasoner` object is created, static rules (which are also available from the `IncrementalRuleMaker`) are added to the `JTMS`. The internal tick counter variable `currentTick` is initially set to `(0,0)`, i.e., tick time 0 and tick count 0.

Incrementing Ticks

The central method of `IncrementalReasoner` is `incrementTick`, which realizes Algorithm 7.3 (*IncrementTickImpl*). It is called by the reasoner internally either by a method for a time increment or another for a count increment. A time increment first updates the `currentTick` of form `(time,count)` to `(time+1,count)` and then calls `incrementTick` with no argument (corresponding to an empty set *Sig* in Algorithm 7.3). Dually, the method for count increments, which takes as argument an atom a (corresponding to $Sig = \{a\}$), updates the `currentTick` to `(time,count+1)` and then calls `incrementTick` with a.

The `incrementTick` method itself first retrieves new annotated rules for the current tick from method `incrementalRules` of the `IncrementalRuleMaker` and registers them for expiration, i.e., they are put in maps for fast lookup (i.e., the maps X_τ, $X_\#$, X'_τ and $X'_\#$ in Algorithm 7.3). The rules are then added to the `JTMS` network. Next, expiring rules at the current tick are looked up from the expiration maps and removed in `JTMS`.

Incremental Evaluation

With the above procedures, the methods inherited from the `Reasoner` trait are implemented as follows. Assuming the engine calls `append` for time point t and atoms A, where the time of the `currentTick` is $t' \leq t$, we first call $t - t'$ time increments in a sub-procedure `updateToTimePoint`. This is followed by a count increment for each $a \in A$.[10] The `evaluate` method similarly first calls `updateToTimePoint` and then simply `getModel` in `JTMS`.

This concludes our overview of the Ticker's implementation. We refer the interested reader to the source code for further details.[11]

7.4 Empirical Evaluation

We will now proceed with the empirical evaluation of the presented algorithms. Since Ticker presents novel reasoning features for streaming data, a comparison with other stream reasoning approaches is of less interest, since they focus on different semantics. The aim of Chapter 6 has been to present incremental reasoning techniques for plain LARS, not engineering techniques towards a high-performance tool with previously existing stream processing semantics. In that regard, the overall aim is to assess the potential benefits of the presented incremental reasoning approach and to shed some light on limitations towards future research issues.

The presented prototypical implementation has some optimizations in place that avoid redundant recomputations. With further engineering efforts, absolute numbers (such as potential atoms per second) could be reduced further. However, we are interested here not in tuning the efficiency of the algorithm's implementation, but the effect of the algorithms in principle. In that spirit, the empirical assessment is geared towards relative performance measures. More specifically, we are interested in the following two questions.

[10] Note that these atoms are given as sequence which determines the result of tuple windows.

[11] `https://github.com/hbeck/ticker`

(Q1) What is the relative performance of the six different *window atom forms*, i.e., given an atom a, the instantiations of window atom

$$\boxplus^{w} \circ a\,, \tag{7.1}$$

where w is a time window function or a tuple window function, and $\circ \in \{@_t, \Diamond, \Box\}$.

(Q2) When is incremental reasoning faster than repeated one-shot solving with Clingo?

With respect to the first question, we observe that all window atoms are easy to compute, with the exception of the tuple-box combination, i.e., form $\boxplus^{\#m}\Box a$. It requires more auxiliary atoms and more rules to be computed and thus should intuitively be slower than the other five forms.

Regarding the second question, we recall the motivation of incremental reasoning, i.e., the aim to avoid recomputing derived information that can be maintained. The employed truth maintenance technique does not recompute the label of atoms (i.e., their containment in the model) when they are guaranteed not to change due to the network's data structures. Accordingly, when the update in the data stream does not lead to necessary changes, performance benefits should result.

This effect should be amplified with an increase of data volume and higher update frequency. For instance, given rules $r_1 = h \leftarrow \boxplus^{n}\Diamond b$ or $r_2 = h \leftarrow \boxplus^{\#m}\Diamond b$, the evaluation of h is agnostic about the number of appearances of b within respective windows; after the first appearance every further such signal is redundant. Such cases that integrate the typical snapshot semantics in LARS deserve special attention.

We should be able to observe further reduction of computation time the larger these windows become: for instance, if involved partial computation depends on the absence of (the derivability of) atom h, we can spare this computation for the next n time points, respectively the next $m-1$ signals once atom b streams in. This guarantee extends for further occurrences of b within these windows. In scenarios with probabilistic appearance of streaming signals, the likelihood for the need of re-evaluation then decreases with larger windows. (This not only holds for window atoms of form "$\boxplus^{w}\Diamond a$," which captures the standard snapshot semantics, but also for "$\boxplus^{w}@_T a$" and dually "not $\boxplus^{w}\Box a$.")

Towards the empirical assessment of these considerations, we present the following evaluation setup.

7.4.1 Setup

We are now going to analyze the requirements for the study the above questions and derive an according setup design.

Engine vs. Reasoning Algorithms

First, we make a distinction between the Ticker engine as such, and its algorithmic core. Recall that Ticker programs are parsed and translated to LARS programs, using a clock time that converts actual temporal durations to logical time points. Depending on the

reasoning mode (`incremental` or `clingo`) and the output configuration, we obtain a wide range of possibilities to assess the engine.

For instance, assume we want to determine the number of atoms that can be processed in one second in various scenarios.[12] We can use a clock time of 1 sec, Clingo mode and output after every second. This leads then to the internal pull-based ASP mode, where Ticker only collects received atoms in a buffer and trivially always has 1 second to compute the model. In this mode, we essentially evaluate Clingo (plus a small overhead introduced by invocation by Ticker). Towards a comparison between the reasoning modes, we would then have to measure the actual computation time of Clingo with the computation time of the incremental mode for the same set of new atoms since the last evaluation. This suggests to consider Clingo only in push-based mode, where each tick triggers model computation as in the incremental mode. This way, we can directly evaluate potential benefits on incremental reasoning.

Next, we observe that the LARS program representation has no notion of clock time. Since both reasoning modes work on time points, we can circumvent any overhead (such as losses in input/output handling) and directly compare the ASP push-based reasoner with the incremental reasoner. By purely processing each tick as fast as possible, we indirectly get a measure for the maximal performance in either mode. Instead of fixing a clock time and then testing how many atoms can be processed per time point, respectively how small the clock time may be set for a given throughput, we get these limits implicitly. We thus do not invoke the Ticker engine in its various configurations, but directly compare the speed of its internal reasoners. Further performance tuning and engineering efforts to reduce overheads are beyond the scope of this study, which focuses on clarifying the potential algorithmic benefit of incremental reasoning using the presented techniques.

Measurements and Environment

To evaluate Ticker's reasoning modes, we will use different benchmark programs: two analytic benchmarks specifically tailored for question (Q1), and two more application oriented programs for potentially obtaining additional insights regarding question (Q2).

We define a *benchmark program* as instantiation from a schematic program that leaves some parameters such as employed window sizes open. We then get a (benchmark) *instance* by considering a specific stream for a benchmark program. Throughout, we always consider a single definition of a stream that provides some randomization. We then consider 5 instances for the same benchmark program and measure the average performance of their processing, i.e., each evaluation metric is an average of 5 runs. To ensure that potential optimizations by the Java-Virtual-Machine (JVM) do not distort the measurements, we always warm up the environment with a number of pre-runs that we vary in incremental mode based on the instance size to ensure that computation does continue not speed up during the 5 evaluation runs; in Clingo mode, we always

[12]Note that there is a broad spectrum of pure processing speed for trivial cases to actual reasoning speed in involved programs, provided relevant atoms that trigger complex computation.

use a single pre-run since the model computation process (with Clingo) is not subject to JVM optimizations. The index of each run (iteration 1 to 5) is used to initialize the random seed to ensure that randomized input varies over iterations and that they are also reproducible. In particular, when comparing incremental and Clingo-based reasoning modes, we thus ensure that they evaluate the same streams (during the 5 runs that are evaluated).

Following the considerations above, we use as central performance metric the number of processed *time points per second*, since this allows us to compare the maximal speed of processing. We also measure the time to initialize the engine before the first time point can be processed (which includes time for pre-grounding in the incremental mode), and the resulting total time. In the evaluation tables we also show, as alternative to time points per second, the average processing time per time point.

All evaluations are executed on a Lenovo Thinkpad T440s with 12 GB DDR-3 RAM, 1.60GHz x4 i5-4200U CPU on ubuntu 16.04, using Scala 2.12.5 and the JVM version 1.8.0_112. We provide an initial memory (heap size) of 4 GB and limit it to 10 GB. The employed Clingo version is 5.1.0. All evaluations can be run via class `DissEvalMain`, which executes a given number of pre-runs and runs on a specific benchmark program. The realization of an instance adds to a program a procedure that generates input signals on the fly for each time point. Due to the controlled random seed (which is fixed per iteration) and the time point, the input at each time point is deterministic (but varies over different iterations). By generating input on demand, we do not have to store the entire stream in memory. Clearly, the time it takes to generate the stream should not influence the measurements. Thus, we measure only the time spent in the reasoner's `append` and `evaluate` methods to obtain the reasoning time after initialization; the latter is the time used to create the reasoner object itself. In addition to the generation of the encodings, this includes the pre-grounding step in case of the incremental reasoner.

All evaluations below were obtained with bash scripts that invoke the main program with different lists `ARGS` of arguments via calls of form

```
scala -J-Xms4g -J-Xmx10g -cp target/scala-2.12/classes evaluation.diss.DissEvalMain ARGS.
```

The standard arguments are

- `inst` to specify the instance,
- `reasoner` to specify the reasoning mode (i.e., `incremental` or `clingo`),
- `pre` and `runs` to declare the number of pre-runs and runs, respectively,
- `timepoints` to fix the length of the timeline of the processed stream,
- `scale`, a number to control the size of the program, and
- `winsize`, the window size.

For instance, replacing `ARGS` above with

```
reasoner incremental pre 10 runs 5 timepoints 2000 scale 32 winsize 50 inst INST
```

yields a complete call for a single evaluation of instance `INST`; potential values for `INST` will be presented in Section 7.4.2. In fact, additional parameters depend on the specific programs and are given as part of the instance name. All parameters and all benchmark programs can be found in class `evaluation.diss.Config`. The Ticker version used for this evaluation is annotated with Tag `v0.8.3`.[13]

7.4.2 Benchmark Programs

We present two categories of benchmarks: two analytic benchmarks are tailored for fine-grained investigations of the different window atom forms, whereas the other two are concerned with contrasting incremental reasoning with repeated one-shot solving with a focus on practically relevant reasoning features.

We use the following parameters in all benchmarks, corresponding to respective arguments `scale` and `winsize` from above:

- n - *program size*: controls the number of ground facts (and thus the size of the ground program); and
- k - *window size*: number of time points.

We uniformly use `timepoints 2000` in this evaluation, i.e., all instances process a timeline with 2000 time points.

Analytic Benchmarks

The first two programs are deliberately small and designed towards a fine-grained analysis of question (Q1) regarding the relative performance of window atom forms, but also allows us to study the relative performance of both reasoning modes. Both analytic benchmarks make use of the following additional parameters.

- a *window atom form* that replaces the schematic window atom $\boxplus^w \circ a$, where $\circ$ is a placeholder for a specific modality $@_T$, $\Diamond$ or $\Box$. Table 7.4 gives all six forms a name. The first character specifies the window (t stands for time and # for tuple windows), the second one for the modality.
- p - *signal probability*: at each time point, each signal $sig(j)$ $(1 \leq j \leq n)$ streams in with probability p, where we use values 0, 0.1, 0.5, 0.9 and 1.0.

Note that our evaluation setup requests model computations (respectively retrieval) after every tick; i.e., after each new signal and each change of a time point. Thus, with respect to considered performance metrics, it is of no additional interest to stream in multiple signals per time point: the switch from one time point to the next itself amounts to an update. (The picture would be different when evaluating a pull-based mode, where all streaming signals within one or even multiple time points can be consumed without a necessity for immediate model computation.)

[13] Direct link: `https://github.com/hbeck/ticker/releases/tag/v0.8.3`

Name	Window atom form
t@	$\boxplus^k @_T\, a$
t$\Diamond$	$\boxplus^k \Diamond\, a$
t$\Box$	$\boxplus^k \Box\, a$
#@	$\boxplus^{\#k} @_T\, a$
#$\Diamond$	$\boxplus^{\#k} \Diamond\, a$
#$\Box$	$\boxplus^{\#k} \Box\, a$

Table 7.4: Window atom forms as used in analytic benchmarks

$$a(X) \leftarrow g(X), \boxplus^w \circ sig(X) \tag{7.2}$$
$$g(1) \leftarrow \tag{7.3}$$
$$\vdots$$
$$g(n) \leftarrow , \tag{7.4}$$

Figure 7.2: Program Basic

Program Basic. The first program provides a minimum to evaluate window atoms. It is specified schematically[14] in Figure 7.2, where n is the program size parameter from above. Thus, $n = 1$ yields a schematic ground program with the fact $g(1)$ and the rule $a(1) \leftarrow g(1), \boxplus^w \circ sig(1)$.

The specific benchmark programs are then obtained by varying parameters n, k and p. In particular, program size $n = 1$ is of interest to obtain micro-benchmarks in order to evaluate the performance difference of the six window atom forms (as building blocks). We increase n up to 32 to study the effect of mere program size; no reasoning (apart from trivial rule firing) occurs in this program. Window size k, a central comparison parameter between the incremental mode and the Clingo mode, shall already be studied at this low level, which concerns mere input processing.

An instance is then obtained by a stream that adds at each time point each of the signals $sig(1), \ldots, sig(n)$ with probability p, e.g., with $n = 32$ this translates to 32 signals per time point when using $p = 1$ and 16 signals on average with $p = 0.5$.

Program Reach. The second analytic program, shown in Figure 7.3, carries out the computation of a transitive closure on top of received signals, i.e., a reachability relation as often seen in ASP. Due to the construction of background facts (predicate *edge*), the inequality relations in the bodies of the first two rules are redundant but help to reduce the size of the pre-grounding in the incremental mode. As before, we obtain a benchmark program from this schematic form by varying program size n and window size k. Like

[14] Since we directly evaluate Ticker's internal reasoning components (which are based on logical time points), we write programs in the formal LARS notation.

$$reach(X,Y) \leftarrow edge(X,Y),\ \boxplus^w \circ sig(X,Y),\ X < Y \tag{7.5}$$
$$reach(X,Z) \leftarrow reach(X,Y),\ reach(Y,Z),\ X < Y,\ Y < Z \tag{7.6}$$
$$edge(0,1) \leftarrow \tag{7.7}$$
$$\vdots$$
$$edge(n-1,n) \leftarrow \tag{7.8}$$

Figure 7.3: Program Reach

for Basic, we stream each signal $sig(j-1,j)$ $(1 \leq j \leq n)$ at each time point with the defined insert probability p and thus obtain the Reach instances.

Application Benchmarks

The second pair of programs shifts the focus from the examination of window atom forms to a combination of different forms of reasoning aspects in a single program. Both programs draw from our work in [BBD+16] and [BBD+17], which is concerned with research in the area of Content-centric Networking (CCN). Towards future internet architectures that better address modern usage demands, CCN is a particular approach where routers can store data locally, e.g., segments of video data that is currently highly popular. Program Strategy sketches the core idea from the simulation architecture of the aforementioned works and is concerned with selecting a suitable *caching strategy* at a given router, i.e., the order in which stored items are dismissed. The objective is to serve frequently requested data items directly from the cache of the router to the end user, instead of retrieving it repeatedly from the network, thereby increasing delay and operator costs. Program Content, similar to that in [BEF17], then selects target nodes for items which are not cached based on the quality of service. In contrast to the other programs, it has multiple models in general.

We now go into the specifics of the application programs.

Program Strategy. Figure 7.4 shows the third evaluation program. Its purpose is to select a caching strategy *lfu*, *lru*, *fifo* or *random*, based on the recent history of the so-called α-value which represents the distribution of interest over all requested data items. For the purpose of our evaluation, we abstract here from the original numbers (of [BBD+16] and [BBD+17]) and partition a scalable range of values $value(1), \ldots, value(n)$ into an upper, middle, and lower third. If during the last k time points the reported value is consistently within one of these sub-ranges, an specific strategy is selected, otherwise *random*.

In contrast to the analytic programs before, Strategy is not schematic with respect to window atom forms. It employs time windows with modalities @ and $\Box$. However, it is scalable due to program size parameter n as before and also has window size k as parameter.

$$
\begin{aligned}
nMax(V) &\leftarrow value(V),\ value(V'),\ V' > V && (7.9)\\
max(V) &\leftarrow value(V),\ \text{not}\ nMax(V) && (7.10)\\
third(V) &\leftarrow value(V),\ max(M),\ V = M/3 && (7.11)\\
upper(V) &\leftarrow value(V),\ third(X),\ value(Y),\ Y = 2 * X,\ Y < V && (7.12)\\
lower(V) &\leftarrow value(V),\ third(X),\ V \leq X && (7.13)\\
middle(V) &\leftarrow value(V),\ \text{not}\ upper(V),\ \text{not}\ lower(V) && (7.14)\\
@_T\, high &\leftarrow \boxplus^k @_T\, alpha(V),\ upper(V) && (7.15)\\
@_T\, mid &\leftarrow \boxplus^k @_T\, alpha(V),\ middle(V) && (7.16)\\
@_T\, low &\leftarrow \boxplus^k @_T\, alpha(V),\ lower(V) && (7.17)\\
lfu &\leftarrow \boxplus^k \Box\, high && (7.18)\\
lru &\leftarrow \boxplus^k \Box\, mid && (7.19)\\
fifo &\leftarrow \boxplus^k \Box\, low && (7.20)\\
specific &\leftarrow lfu && (7.21)\\
specific &\leftarrow lru && (7.22)\\
specific &\leftarrow fifo && (7.23)\\
random &\leftarrow \text{not}\ specific && (7.24)\\
value(1) &\leftarrow && (7.25)\\
&\vdots\\
value(n) &\leftarrow && (7.26)
\end{aligned}
$$

Figure 7.4: Program Strategy

Rules (7.9)-(7.14) compute the partition of values into three buckets, by determining a third of the maximum value, and then assigning either predicate *upper*, *lower*, or *middle* to each value. While this computation is trivial, it gives a good example of a basic use for incremental reasoning: there is no point in recomputing this static information; nevertheless, one may want to express it within the query/program itself.

Based on this static information, the next three rules – (7.15) to (7.17) – assign to the last k time points atom *high*, *mid*, or *low* with the time point of the respective streaming α-value. This way, we abstract away the specific number and can then universally quantify in Rules (7.18)-(7.20): in case of a consistent high, middle, or low value throughout the considered time span a specific strategy can be selected, otherwise *random* in Rule (7.24).

To obtain specific instances, we provide again a probability p. We always stream exactly one signal of the form $alpha(v)$ per time point. Initially, value v is randomly picked from $\{1, \ldots, n\}$ and then changes from one time point to the next with probability p. In this case, it decreases either to $v - 1$ or it increases to $v + 1$ (with equal chance); if $v = 1$

$$need(I,N) \leftarrow item(I),\ node(N),\ \boxplus^k\Diamond req(I,N) \tag{7.27}$$
$$avail(I,N) \leftarrow item(I),\ node(N),\ \boxplus^k\Diamond cache(I,N) \tag{7.28}$$
$$src(I,N,M) \leftarrow need(I,N),\ \text{not}\ avail(I,N),\ avail(I,M),\ N \neq M \tag{7.29}$$
$$getFrom(I,N,M) \leftarrow src(I,N,M),\ \text{not}\ dism(I,N,M) \tag{7.30}$$
$$dism(I,N,M) \leftarrow node(M),\ getFrom(I,N,M'),\ M \neq M' \tag{7.31}$$
$$dism(I,N,M) \leftarrow src(I,N,M),\ src(I,N,M'),\ worseThan(M,M') \tag{7.32}$$
$$worseThan(N,N') \leftarrow minQ(N,Q),\ minQ(N',Q'),\ N \neq N',\ Q < Q' \tag{7.33}$$
$$minQ(N,Q) \leftarrow node(N),\ qLev(Q),\ \boxplus^k\Diamond qual(N,Q),\ \text{not}\ nMinQ(N,Q) \tag{7.34}$$
$$nMinQ(N,Q) \leftarrow node(N),\ qLev(Q),\ qLev(Q'),\ \boxplus^k\Diamond qual(N,Q),\ \boxplus^k\Diamond qual(N,Q'),Q' < Q \tag{7.35}$$
$$node(1) \leftarrow \tag{7.36}$$
$$\vdots$$
$$node(n) \leftarrow \tag{7.37}$$
$$item(1) \leftarrow \tag{7.38}$$
$$\vdots$$
$$item(i) \leftarrow \tag{7.39}$$
$$qLev(1) \leftarrow \tag{7.40}$$
$$\vdots$$
$$qLev(q) \leftarrow \tag{7.41}$$

Figure 7.5: Program Content

or $v = n$ it changes to $v = 2$, respectively $v = n - 1$ to stay within the boundaries from 1 to n.

Program Content. The program to select sources for items is shown in Figure 7.5. This program builds from two kinds of static background data: n nodes and i items that can be requested and cached at each node. Once an item I is requested at a node N, we infer a need for the pair (I, N); which is trivially satisfied when I is available at N, i.e., when it is still cached there. For both requests and caches (Rules (7.27) and (7.28)) we consider any occurrence during the last k time points. If I is not available at N, but at a different node M, then M is a source for request (I, N), from which I can be retrieved (Rule (7.29)). Whether this choice is made (or a different source is selected) is captured in predicate *getFrom* (Rule (7.30)), which selects a source unless it is dismissed. For the latter, there are two reasons. First, we dismiss in Rule (7.31) a node in case some other is picked, to ensure a unique target node for the specific request of I at N. Second,

Program	Class	Instance Pattern
Basic	`ScalableRandomizedBasicInstance`	`srbasic_w`w`_n`n`_p`p
Reach	`ReachSigInstance`	`rs_w`w`_n`n`_p`p
Strategy	`StrategyInstance`	`strat_n`n`_p`p
Content	`ContentInstance`	`content_n`n`_i`i`_q`q`_pc`p_c`_pq`p_q

Table 7.5: Implementations and instance names for invocation; w is a window atom form `ta`, `td`, `tb`, `ca`, `cd`, or `cb`, encoding t@, t$\diamond$, t$\square$, #@, #$\diamond$ and #$\square$, respectively.

we dismiss in Rule (7.32) source M, if its quality of service is worse than some other source M'. The quality of service is characterized by the lowest quality level Q recently reported for the node. (Such a quality level might reflect, e.g., the download speed or general availability of the node.) To this end, we maintain in Rule (7.34) the recent minimal quality level Q reported for any node N, which is given by a signal $qual(N, Q)$ such that we cannot infer $nMinQ(N, Q)$, i.e., evidence that Q is not the recent minimal value at N. The latter may stem from Rule (7.35), where we compare recent quality level signals for each node.

To instantiate the program, we always use 5 quality levels. The streaming behaviour depends on two probabilities: similarly like for Strategy, we initially assign to each node a random quality level from $\{1, \ldots, 5\}$ and change this value at each time point (for each node) with a probability p_q. We stream according predicates *qual* only in case of such a value change. Moreover, at each time point each node caches a random item with probability p_c. For both probabilities p_c and p_q we use the fixed value 0.5. Finally, we issue a random request at every time point, i.e., a random item at a random node.

7.4.3 Results

We are now going through the results of evaluating various instantiations of the benchmark programs described above. Table 7.5 gives an overview of the class names implementing them (they can be found in package `evaluation.diss.instances`), along with the instance name pattern as used in the invocation. (The dispatch is handled in the `Config` class.) For instance, when invoking program Basic for window atom form $w = t\diamond$, size $n = 32$ and signal insert probability $p = 0.5$ the value for the `inst` argument in the aforementioned Scala invocation is `srbasic_wtd_n32_p0.5`.

All evaluation results are given in Tables B.1-B.13 in the Appendix, where we present the averages over all runs for the total runtime, the initialization time, and processing time per logical time point in seconds, and dually also the *average number of time points processed per second*. We take the latter as a uniform performance measure for the following charts which visualizes these tables' results.

In the following, we refer to the reasoning modes by their command line names, i.e., `incremental` and `clingo`.

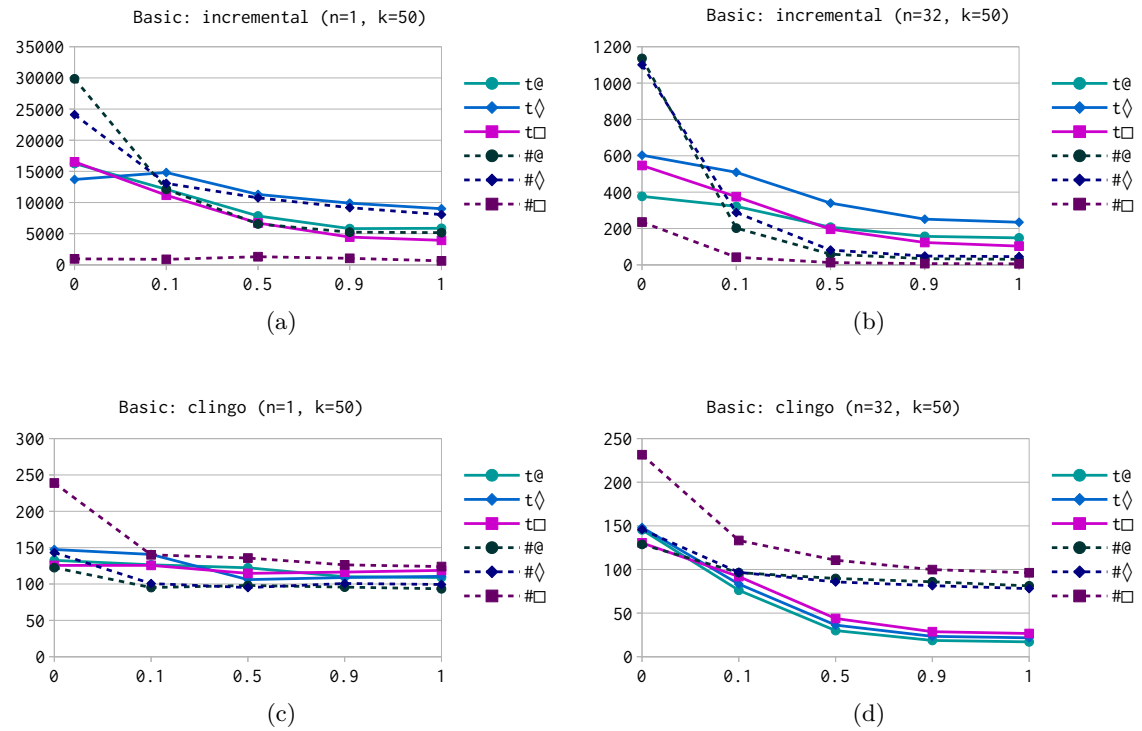


Figure 7.6: Program Basic. Difference in performance, measured in processed time points per second (y-axis), over different signal insert probabilities $p = 0, \ldots, 1$ (x-axis) for different window atom forms. Sources: Tables B.1 and B.2.

Program Basic

Figure 7.6 shows the effect of different signal insert probabilities on the performance of different window atom forms, where we fix window size $k = 50$. In Charts (a) and (b), results for `incremental` are shown for program sizes $n = 1$ and $n = 32$, respectively. Likewise, Charts (c) and (d) show respective results for `clingo`. We observe in Chart (a) that case #□ is the slowest, i.e., it has the least number of time points processed per second. This reflects the more involved encoding. Naturally, performance decreases with increasing throughput (from left to right) for the remaining five cases. We see in Chart (b) that all time-based cases (t@, t◇, t□) are faster (unless no data is streaming in). This is due to an increased number of incremental rules stemming from additional auxiliary atoms. Figure (c) reveals that there is little difference between window atom forms for the minimal case ($n = 1$) using `clingo`. Interestingly, Chart (d) indicates that `clingo` tends to be faster for the tuple-based cases (#@, #◇, #◇).

We now fix insert probability $p = 0.5$ and investigate the same programs (with $n = 1$ and $n = 32$, respectively) for both reasoning modes in Figure 7.7, where we vary the window size k from 1 to 500. Charts (a) and (b), depicting results for `incremental`, again show that #□ is the slowest window atom form, and all forms become slower with

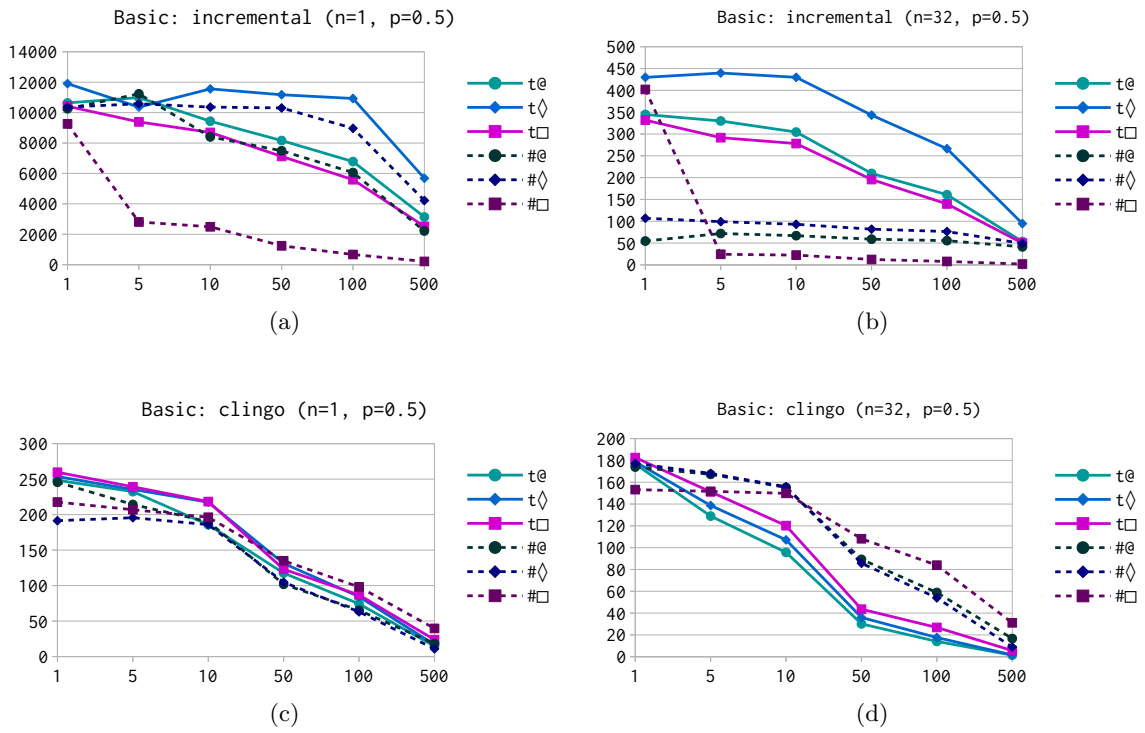


Figure 7.7: Program Basic. Effect of varying window size $k = 1, \ldots, 500$. Sources: Tables B.3 and B.4.

increasing window size due to the increased total volume of data that is being processed. The same holds for `clingo`, as shown in Charts (c) and (d), where we again see less difference between window atom forms than for `incremental`.

We discussed earlier that the practically most relevant modality is $\Diamond$, formalizing the prominent snapshot semantics. We thus single out the respective time-based and tuple-based cases for both reasoning modes in Figure 7.8 to study the relative performance of resulting window atoms forms, when the window size is increased. We see in Chart (a) for program Basic with $n = 32$ that `clingo` is faster for $\#\Diamond$ when window sizes are small, but when the window size is sufficiently big, `incremental` is faster, as for all entries of the time-based case ($t\Diamond$). Chart (b) depicts a variant where at every time point, every signal $sig(1), \ldots, sig(32)$ is streaming in with a probability of only 0.01. We find that `incremental` is consistently faster than `clingo`. Notably, this shows that `incremental` may be beneficial not only when data is streaming in with high frequency. The difference is best is explained by the overhead of having to repeatedly invoke `clingo`. In our setup, which focuses on immediate model evaluation, moving from one time point to the next has a similar effect than processing a signal.

Next, Figure 7.9 shows the effect of varying the program size, using the cases for modality $\Diamond$ from before. We see here a clear indication that the program size has a huge effect on the performance of `incremental`. It is orders of magnitude faster than `clingo`

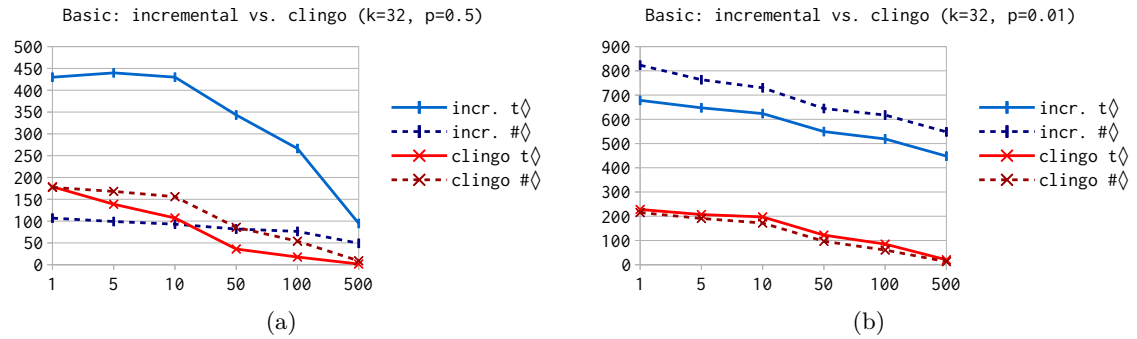


Figure 7.8: Program Basic. Relative effect of window size $k = 1, \ldots, 500$ for snapshot semantics in both reasoning modes. Sources: Tables B.4 and B.5.

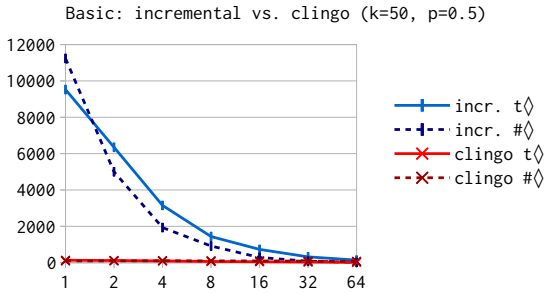


Figure 7.9: Program Basic. Effect of program sizes $n = 1, \ldots, 64$. Source: Table B.6.

when the program is small, and yields a similar performance when the program gets bigger. Eventually, `clingo` becomes faster (relative to a fixed window size, here $k = 50$) when the program gets large enough. (This can be seen in Table B.6, page 234.)

Program Reach

We observe similar results for program Reach that involves some computation beyond simple rule firing. Figure 7.10 shows, in analogy to Figure 7.6, the effect of varying signal insert probabilities, focusing on program size $n = 8$ and window size $k = 50$. Again, we observe that for `incremental` the time-based cases are faster than the tuple-based ones (consistently so for $p \geq 0.5$, and for `clingo` vice versa.) This is also observed in Figure 7.11, which shows the window atoms' influence over varying window sizes.

In Figure 7.12, we again study the relative performance of snapshot semantics using different window sizes. We use insert probability $p = 0.5$ and compare a program size $n = 8$ in Chart (a) with $n = 32$ in Chart (b). For the small program, we observe that `incremental` is faster when sufficiently large windows are in use. We also can observe the impact of the window size on the relative performance of window modes in Chart (b). There, however, `clingo` is significantly faster for small windows. We observe that for

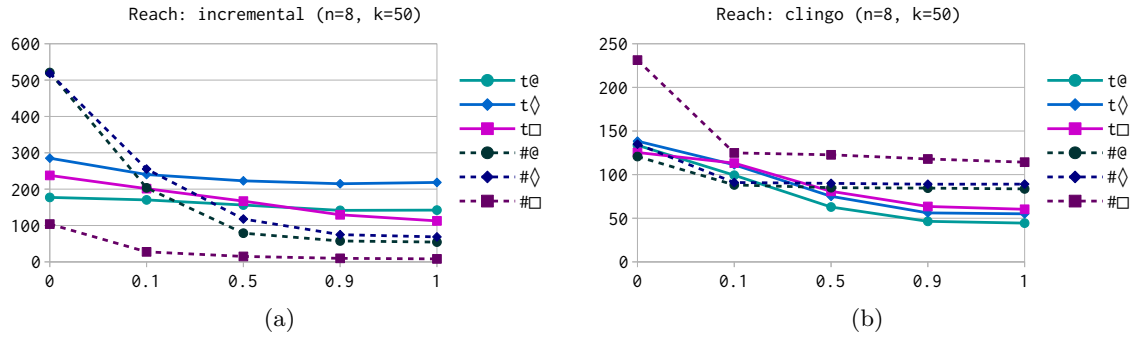


Figure 7.10: Program Reach. Effect of varying signal insert probability $p = 0, \ldots, 1$. Source: Table B.7.

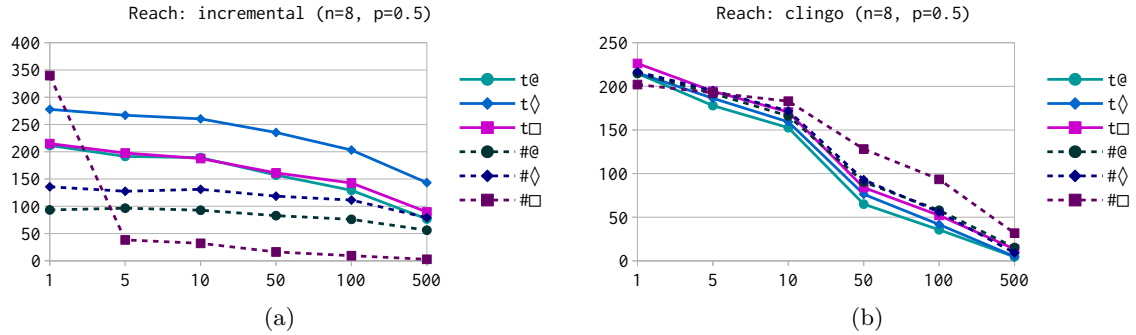


Figure 7.11: Program Reach. Effect of varying window size $k = 1, \ldots, 500$. Source: Table B.8.

small windows, incremental reasoning is closer to repeated one-shot solving. It is then no surprise that Clingo, the state-of-the-art ASP solver, has to be significantly faster than Doyle's reasoning algorithms, given that program Reach adds actual reasoning on top of the input processing of program Basic. However, as window size increases the benefit of model maintenance kicks in and the relative benefit of `clingo` decreases.

Figure 7.13 shows a chart for program Reach similarly as Figure 7.9 before, with the above finding that smaller programs favour `incremental`.

Program Strategy

Observations for the above micro-benchmarks carry over to program Strategy, which involves only window atom forms t@ and t□.

Figure 7.14 again shows the effect of window sizes, and directly compares reasoning modes for four (respectively two) different program sizes. In Chart (a), we see the evaluations for program sizes $n = 3, 9, 30, 90$, where `incremental` is significantly faster for smaller sizes. We zoom in on the competitive cases $n = 30, 90$ in Chart (b), where we again

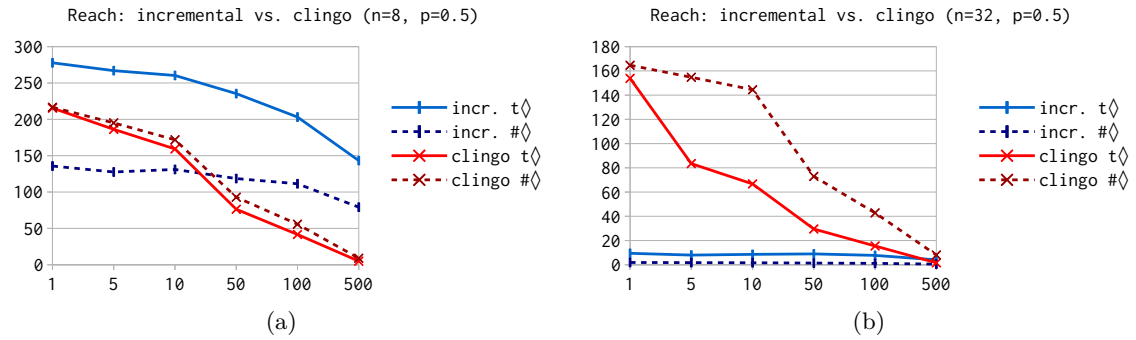


Figure 7.12: Program Reach. Effect of varying window size $k = 1, \ldots, 500$. Sources: Tables B.8 and B.9.

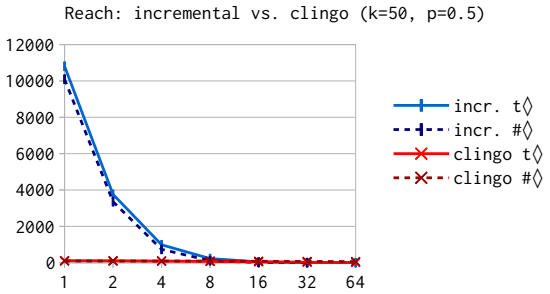


Figure 7.13: Program Reach. Effect of program size $n = 1, \ldots, 64$. Source: Table B.10.

observe a finding from above, i.e., that increasing window sizes tends to favour incremental reasoning. In particular, the largest instance ($n = 90$) is solved more efficiently with `clingo` for small windows, but the `incremental` is advantageous at window sizes $k \geq 100$.

Figure 7.15 presents a different view, where the x-axis varies the program size. Again, we see that `incremental` is significantly faster for small instances, and that the program size has less influence on the relative performances using `clingo`.

Program Content

The final evaluation benchmarks based on program Content, we make use of the same charts as for Strategy.

We show in Figure 7.16 a comparison of reasoning modes, where we parameterise the program size in two dimensions. We fix in Chart (a) (respectively Chart (b)) the number n of 10 (respectively 20) nodes, and then show the influence of handling $i = 4, \ldots, 64$ items in the resulting network. A similar finding reoccurs, showing superior performance of `clingo` for small window sizes, and increasingly beneficial behaviour of `incremental` when windows become larger. We observed before that `clingo` is more robust with respect to instance sizes. This is now visible in two ways. First, within each chart, the distance

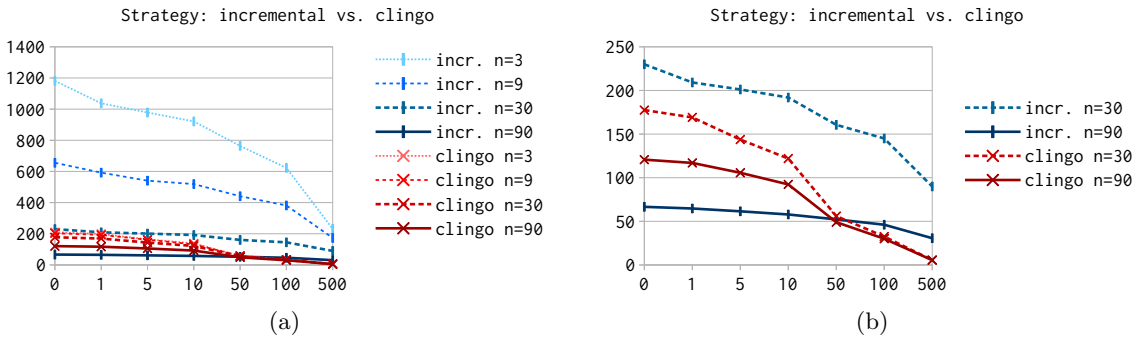


Figure 7.14: Program Strategy. Window sizes $k = 1, \ldots, 500$. Source: Table B.11.

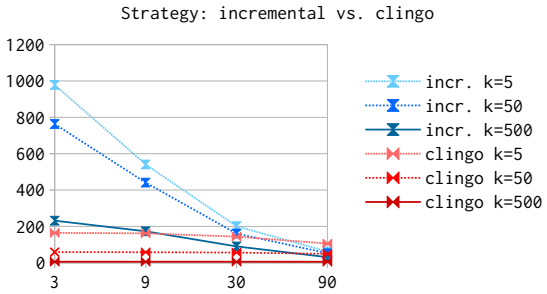


Figure 7.15: Program Strategy. Program sizes $n = 3, \ldots, 90$. Source: Table B.11.

between the red lines is smaller. Second, both charts show almost the same performance for `clingo`. Reasoner `incremental` behaves differently, showing bigger differences in both ways, i.e., when increasing parameters i and n.

Finally, Figure 7.17 again shows findings from above based on varying the number of items: `clingo` is faster the smaller the window is, and robust with respect to the program size. By contrast, `incremental` is less efficient the bigger the program becomes, but may perform better with large windows in place.

Summary of Results

Reviewing the above evaluation results, we can identify some robust patterns.

(i) *High throughput / fast update.* When the task involves a lot of low-level processing or fast updates, the overhead of repeatedly invoking the Clingo solver is too costly. As a consequence, incremental reasoning is significantly faster in such cases.

(ii) *Involved reasoning.* On the other hand, the more a task is concerned about repeatedly computing fresh models involving complex reasoning, mode `clingo` tends to be faster.

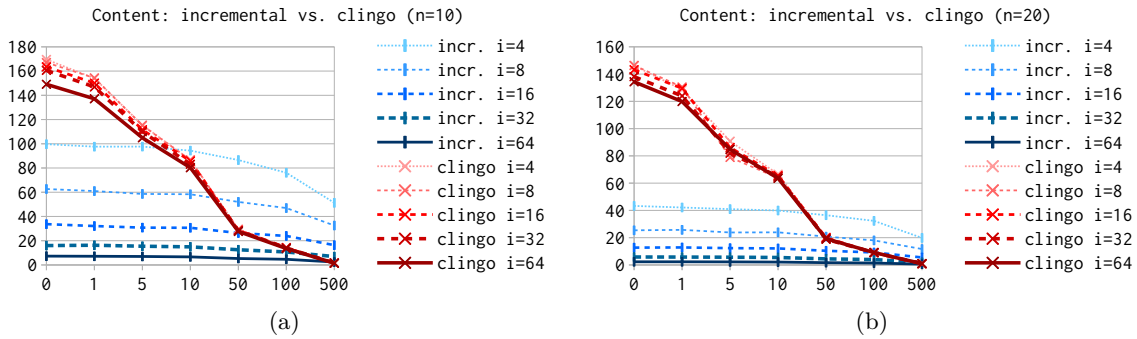


Figure 7.16: Program CONTENT. Window sizes $k = 1, \ldots, 500$. Source: Tables B.12 and B.13.

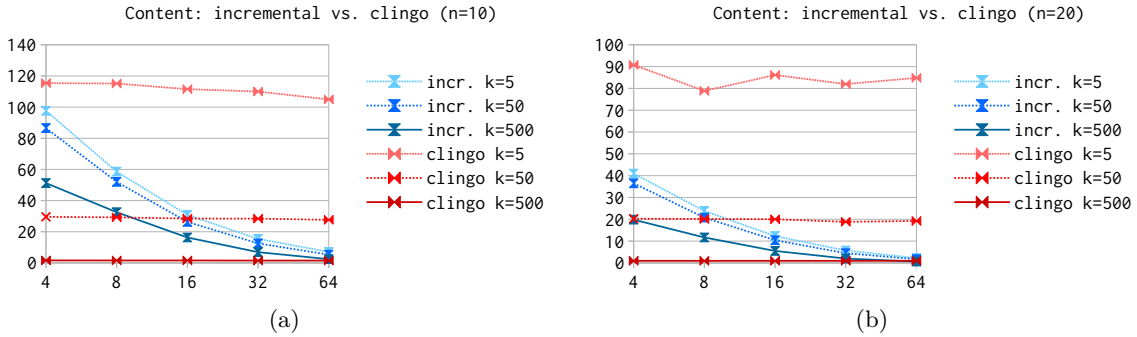


Figure 7.17: Program CONTENT. Number of items $i = 4, \ldots, 64$. Source: Tables B.12 and B.13.

(iii) *Window size.* Throughout, the relative performance between repeated one-shot solving and incremental reasoning tends to favour the latter with increasing window sizes. Also this result is natural: regardless of the quality of the incremental procedure as such, its potential benefit is limited with small window sizes. The larger the considered history gets, the smaller the relative difference in a data (or time) update, and thus the higher the potential for relative performance. In other words, the potential for model maintenance increases (in particular for snapshot semantics) with the size of the window, as corroborated in this study.[15]

[15] We note that the evaluations of the effect of varying window sizes could be more fine tuned: given size k, at any time point t of the first $k-1$ points in the timeline we evaluate (for time windows) a program with a window of size $|t| < k$ due to the cut-off position at time point 0. Since we always use 2000 time points, this means that for $k = 500$ we use the full window size only for three quarters of the timeline. A more fine-grained study of the influence of sliding windows could instead use a timeline of 2500 time points and start to record the evaluation time (for `append` plus `evaluate`) at $t = 500$, where the window extends to its full size. In light of the above findings, we infer however, that the key finding would not change, only the relative performance benefit of incremental reasoning would increase slightly.

(iv) *Program size.* By contrast, the mode using Clingo is very robust with respect to the program size and thus tends to be favourable for large programs, unless the window size becomes very large.

In conclusion, we studied the performance of reasoning incrementally, using Doyle's truth-maintenance update technique on an incremental encoding, and repeated one-shot solving using Clingo on a static encoding. With respect to our first question (Q1) regarding relative performance of window operators, we found that in the incremental mode the time-based cases are typically faster than the tuple-based ones, and that the tuple-box combination is the slowest. Using Clingo, the relative performance of all cases is closer, where tuple-based cases tend to be faster than time-based ones. Regarding question (Q2) on comparing reasoning modes, we found that small programs, high update frequency and large windows favour incremental reasoning, whereas scenarios with larger programs, more involved reasoning and small windows are more efficiently solved with Clingo. The results are intuitive: the size of a window can be seen as a yardstick in the spectrum from static reasoning to highly dynamic stream reasoning, and the smaller the window, the closer the setup is to reasoning with static data, where a state-of-the-art solver must be beneficial. Nevertheless, incremental reasoning can also be considered in terms of model maintenance as such, from one tick to the next, regardless of windows. For cases with small or minimal window sizes (where the incremental reasoning mode was faster) we did not examine to which degree performance benefits can be attributed to model maintenance, and to which degree they stem from the overhead of their competing mode (i.e., the repeated calls to Clingo). In fact, the definition of suitable metrics of *maintainability* of models is a separate research issue.

7.5 Discussion

We conclude this chapter by remarks on the relation between time and time points, multi-shot solving features of Clingo, and potential research directions.

Time vs. Time Points

We discussed in Section 7.2.1 a limitation regarding the expressivity with respect to *temporal* conditions due to the employed pre-grounding mechanism. However, even if we assume a fully incremental grounding procedure (for grounding on-the-fly), another difficulty remains: the semantics of Ticker is fully expressed in LARS which has no notion of time per se; thus temporal values or durations still have to be matched to time points.

Current implementation details aside, consider from a purely semantic perspective a scenario where we want to query at which during the last 10 seconds we received a warning signal. We naturally use a clock time $c = 1s$ and a rule `r`:

```
q(T) :- @T warn [10 s].
```

However, time variable `T` does not refer to seconds of the computer clock but to full seconds since the start of the engine. The first instant is associated with time point 0,

regardless of the computer time. That is to say, there is no way of referring explicitly to a specific time. In essence, even if a more sophisticated grounding algorithm were in place, temporal expressions would be limited to those expressible in terms of time points.

Now suppose we use rule `r` in an application that requires a clock time $c = 100ms$, i.e., each time point corresponds to a tenth of a second. Accordingly, time variable `T` does not refer to seconds anymore. However, we can virtually group signals appearing within chunks of 10 time points by integer division.

```
q_ms(S) :- q(T), int(S), S=T/10.
```

One problem with this solution is that the rule needs to replaced when a different clock time is used; there is no means to access the clock time in rules.

In summary, one has to keep in mind that any time variable `T` (as in `@T`) in Ticker refers to time points which have the duration due to the specified clock time. Future extensions, capable of handling relations and arithmetic over time variables in the first place, might consider the use of time units in expressions like `D = T + 10s` such that `D` is assigned a time point that is 10 seconds after `T`, regardless of the employed (compatible) clock time.

Notes on the Use of Clingo

We add two remarks regarding the ASP solver Clingo [GKKS14] used in this work; commenting on its multi-shot features and on the issue of model update.

Multi-shot features. For non-incremental reasoning, we use Clingo by repeated one-shot solving. That is, for each tick we call Clingo with a static schematic program, augmented by the stream representation and further auxiliary facts. We also considered using Clingo's reactive/multi-shot facilities[16] which are based on [GGKS11], have since evolved [GGK+12, GKKS14], and were successfully applied: e.g. see [GKOS15]. Unfortunately, for our purposes, control features in Clingo are not applicable.

First, the control features in Clingo allow addition of new rules, but not removal of existing ones. Technically, removing might be simulated by setting a designated switch atom to false. However, this approach would imply that the program keeps growing over time. Second, we considered using reactive features as illustrated for rule r of Example 77, using a program part that is parameterised for stream variables, including that of tick (t, c).

```
#program tick(t, c, v).
#external now(t).
#external cnt(c).
#external alpha_at(v,t).
high_at(t) :- w_time_2_alpha(v,t), t >= 18.
w_time_2_alpha(v,t) :- now(t), alpha_at(v,t).
w_time_2_alpha(v,t) :- now(t), alpha_at(v,t-1).
w_time_2_alpha(v,t) :- now(t), alpha_at(v,t-2).
```

[16] Clingo 5.1.0. API: `https://potassco.org/clingo/python-api/current/clingo.html`

However, this encoding is not applicable, since atoms in rule heads cannot be redefined, i.e., they cannot be grounded more than once.

Notably, it was not the goal of this work to explore various encodings to increase the range of applicability of Clingo's specific features, but to provide a second mode of reasoning that (i) reduces to ASP, and (ii) could be used as base line for empirically comparing the incremental reasoning mode.

Model update. For stratified programs (which have a unique model), repeatedly calling Clingo (by standard one-shot solving) on the encoded program is a practical solution. However, when a program has multiple models, we then have no link between the output of successive ticks, i.e., the model may change arbitrarily. For instance, consider the following program:

```
a :- not b, not c.  b :- not a, not c.  c :- not a, not b.
```

Using Clingo 5.1.0, the answer set of the program that is returned first is `{a}`, which remains an answer set if we add rule `a :- not c`. However, the first reported answer set now is `{c}`.

Future Research Issues

As a research prototype, Ticker indicates potential benefits of reasoning incrementally for ASP-like languages like LARS. Based on its architecture and the empirical evaluation, the following components stand out as entry points for future improvement.

The first module to be discussed concerns the pre-grounding, which was a pragmatic choice for the sake of establishing a research prototype. Practically, however, pre-grounding is limiting since providing according guards is less intuitive (when writing rules), comes with additional cost (to load according facts) and is sometimes not possible in the first place: in case incoming signals are not known upfront, the only way to define guards is to specify a naive set of all potential signals which then leads to an impractically large grounding. This may be the case even if potential input signals are known in principle. Essentially, pre-grounding also undermines the idea of forgetting as conceptualized by windows. In that regard, techniques for incremental grounding on-the-fly would by a practically important next research direction, which, due to the non-monotonicity of the language, is also technically challenging when an inflation of the incremental encoding shall be avoided. Full grounding on-the-fly would also solve the limitations on the use of time variables as discussed.

The second issue in the incremental reasoning mode concerns the use of JTMS, which cannot compete with modern ASP solving techniques in terms of efficiency; this was clearly confirmed in the empirical evaluation. Furthermore, JTMS guarantees correct model update only in the absence of constraints (and more generally, odd loops). In ASP, constraint satisfaction problems are often modelled with the *guess-and-check* paradigm, where the *guess* is a subprogram that declares a hypothetical space of solutions from which undesirable combinations are excluded via constrains in the *check* part. This modelling technique is currently is not available in the incremental mode due to Ticker's utilization

of JTMS, which nevertheless offered a first good show case algorithm for programs reducible to normal ASP without odd loops. Thus, as a second research direction, the development of other model maintenance techniques are of interest that likewise update a model based on an adaptation of its program. Notably, the presented incremental encoding is agnostic about the specific model update procedure; it only assumes the possibility to add and delete rules to express the evolution of an ASP program.

The third dimension for improving Ticker concerns the specific instantiation of LARS, i.e., the plain LARS fragment with sliding time and tuple windows. Here, one may first consider to incorporate additional window functions, i.e., different movement patterns (hopping and tumbling in addition to sliding) and other selection mechanisms such as the filter window. Another aspect is to improve the relation between time (temporal durations) in Ticker and logical time points as used in the underlying LARS semantics. For instance, one might enhance the input syntax for more convenience; e.g. in way such that time points in @-atoms have time units. Considering larger fragments beyond plain LARS as underlying formal language would be another starting point for research with a focus on theoretical boundaries. Towards improved usability of the employed fragment, including aggregation functions or extensions for handling multiple input streams would be of practical importance.

CHAPTER 8

Conclusion

We conclude with a recapitulation of the LARS framework and some of the established results based on it. We review the incremental evaluation techniques and the resulting stream reasoning tool Ticker. Finally, we point towards future research issues.

8.1 Summary

We now summarize the above work from the perspective of the initial problem statement (cf. Section 1.2) by discussing the three main research objectives.

A framework for expressive stream reasoning with window mechanisms. We discussed the previous lack of a common formal underpinning for advanced stream reasoning beyond low-level stream processing approaches. The latter typically employ window mechanisms that select recent data based on time or counting tuples. For advanced reasoning techniques and solvers as for Answer Set Programming or SAT, no such mechanisms were available as first class citizens of the respective languages. We observed a trade-off between low-level processing, focusing on tasks that are also typical for querying databases, and high-level reasoning that targets expressiveness, i.e., model generation, abstraction, nonmonotonicity, and the like. A formal framework for stream reasoning thus needs to cover a broad spectrum in terms of semantic features and thus must be modular and flexible. Towards analytic utility, it should be fully declarative and model-based, rather than operationally defined.

To tackle these objectives, we presented the LARS framework in Chapter 3 that comes in two incarnations. In its monotonic core, it allows one to evaluate *formulas* on streams; the latter are viewed as discrete timelines where time points are mapped to sets of atoms. LARS formulas extend propositional logic by generic window operators that select recent substreams, and different modalities for controlling the evaluation with respect to the temporal dimension. On top of this, LARS *programs* essentially extend Answer Set Programming by means of these formulas and evaluation mechanisms; the notion of an

answer set is lifted to an answer stream which intuitively adds inferred data to the input stream to minimally satisfy a given program. Due to this design choice, LARS programs inherit semantic features of ASP, such as full declarativity, nonmonotonicity, multiple models, or the possibility to express transitive relations.

There are two main dimensions which makes the LARS framework very flexible. The first one concerns the possibility to restrict to a fragment. Specifically, we defined the plain LARS fragment (cf. Section 4.1) as a straightforward extension of ASP, replacing body atoms by so-called window atoms and also considering @-atoms in heads; it served as guiding fragment and was used in multiple publications. The second dimension of flexibility lies in the generic definition of window operators. In fact, only by fixing a specific set of window functions, a concrete logic is obtained. While formula and program evaluation are PSpace-complete in general, complexities drop significantly when considering practically relevant syntactic or semantic restrictions on window operators, as we explored in Section 3.3; resulting fragments are then not harder than ASP.

We explored the analytic capabilities of LARS by studying the relationship to other formalisms in Chapter 4, where we covered continuous querying in the spirit of databases (CQL) and the semantic web (C-SPARQL, CQELS), reasoning as typical in runtime verification (LTL) and rule-based complex event processing (ETALIS). Towards optimizing programs, e.g. by rewriting for more efficient evaluation, we give a first study in Chapter 5. Building on according literature for ASP, we defined different notions of equivalence between LARS programs, i.e., ordinary, strong and uniform equivalence; and moreover a notion of data equivalence that reduces to relativized uniform equivalence. We provided semantic characterizations of these equivalence relations based on a logic called bi-LARS that evaluates two streams at the same time. Restricting the considered LARS fragment enabled an alternative definition that is closer to previous work in ASP, likewise establishing a connection to Heyting's logic of Here-and-There. We noted that deciding the defined equivalence notions for plain LARS is not harder than for ASP.

Techniques for incremental reasoning. When the evaluation of a query or program is subject to changing data, incremental computation is required to reduce output delays. Incremental algorithms that update previous results are especially important for expressive reasoning tasks, where computing solutions from scratch comes with even higher cost in general.

Towards incremental evaluation of plain LARS programs, we developed in Chapter 6 algorithms for updating a previous answer stream. We first reviewed Justification-based Truth Maintenance Systems (JTMS) that can update a program's answer set after the addition of a new rule. Unlike the original work, we specified JTMS unambiguously by means of formal definitions and precise algorithms and then extended it for rule removal. To compute the answer streams at a single time point, we then gave an encoding from plain LARS (with sliding time windows and sliding tuple windows) to ASP. Next, we showed how a slight adjustment leads to an incremental encoding, i.e., a set of rules that can be maintained stepwise by rules that are deleted and added at each tick; the latter conceptualizes a minimal change after which re-evaluation can be considered. We then obtain an incremental model update procedure by feeding these difference sets to

the JTMS. We concluded the chapter by mentioning two further works on incremental reasoning for plain LARS: the first directly extends JTMS for the LARS semantics, the second extends semi-naive evaluation techniques for programs with unique models.

A prototypical rule-based reasoning engine. Most available stream processing frameworks focus on low-level computation and high-performance. Naturally, providing reasoning features with more expressiveness is more difficult and provided by fewer tools. In particular, no system with rule-based stream reasoning in the spirit of ASP with explicit window mechanisms was available prior to this thesis.

In Chapter 7, we presented Ticker, a prototypical stream reasoning tool that can process streams in different push-based and pull-based evaluation modes. A Ticker program can be seen as a plain LARS program that (i) may use sliding time windows and sliding tuple windows, and (ii) has an explicit notion of time. That is, to obtain the semantics of a Ticker program, the temporal durations used in time windows of Ticker are translated to LARS time points based on the given configuration of the engine. To compute results, Ticker uses the techniques presented in Chapter 6, i.e., either the static encoding for repeated one-shot solving (using the ASP solver Clingo), or the incremental encoding and our own extended JTMS implementation for incremental model update. The Clingo mode is required for programs with odd loops, and tends to be superior when the use case can be viewed as repeated evaluation of a complex reasoning task after certain intervals. The higher the throughput (i.e., mere processing demand), or the bigger the employed window sizes, the bigger the relative benefit of incremental reasoning.

8.2 Outlook

During the development of this thesis, LARS has already been used as modelling language and framework. We mentioned in Section 3.2.4 its first application as controller language for a simulation software in Content-Centric Networking research. In our works on Ticker and Laser it has been successfully used as independent reasoning language, providing a formal semantics for new streaming engines. These research prototypes implement novel algorithms for incremental high-level reasoning based on fully declarative rule-based programs. With these first developments based on LARS in place, new research opportunities open up, which we already partially mentioned in concluding remarks of previous chapters.

On the purely theoretical side, LARS as framework provides many opportunities for further studies. Apart from natural investigations of fragments within the defined framework, a particular interest lies in deviations from some of the assumptions made by LARS. For instance, one may alternatively consider continuous timelines, interval-based semantics, or multiple input streams as part of the language. Due to our focus on model-based semantics, we did not discuss operational aspects that could be part of the formal framework as well. In particular, LARS has as a single notion of time and is thus less suitable for dealing with delays, outages, or out-of-order events. It is also not geared for studying the semantics or performance effects of distributed computation.

Another issue of theoretical interest, with potentially more immediate practical relevance, are further algorithms for incremental reasoning. It is clear that in the worst case full recomputation cannot be avoided. However, the output of many stream reasoning use cases show some continuity over time, in particular at a higher semantic level. While low-level and fine-granular data (such as GPS positions of cars) might change extremely fast, derived high-level information (such as existence of a traffic jam) is typically more robust. With the plain LARS fragment we essentially proposed a dividing line between the low-level processing part and the high-level reasoning part by means of window atoms; incremental reasoning then exploits the continuity at the higher levels. Improving incremental reasoning at the higher levels as such translates in our terms to incremental grounding and solving techniques for ASP, but other languages and formalisms are of interest as well. For languages with multiple models, the question of model maintenance arises also from a semantic perspective, i.e., which model to pick as successor of a previous one. Practical definitions of according criteria along with incremental algorithms to compute them with suitable heuristics seem to be lines of research of potentially high impact.

APPENDIX A

Proofs

A.1 LARS: A Logic-based Framework for Analytic Reasoning over Streams

In this section, we provide proofs of the complexity results and further details on computation.

LARS Formulas

Proof of Theorem 7. Let $M = \langle S^\star, W, B\rangle$, $S = (T^\star, v^\star)$, be a structure, let $S \subseteq S^\star$ be a substream of $S^\star$ and let α be a ground formula. Let N denote the size of M plus S.

PSpace *membership.* We show that the space used to determine $M, S, t \Vdash \alpha$ is bounded by $O(|\alpha| * N + N^k)$, where $|\alpha|$ is the size (length) of formula α and $k \geq 1$ is some constant.

Indeed, a ground formula α can be represented as a tree whose leaf nodes are atoms from $\mathcal{A}$ and whose intermediate nodes are operators from $\{\neg, \wedge, \vee, \rightarrow, \Diamond, \Box, @_t, \boxplus^w, \triangleright\}$, where $t \in T$. For example, the formula $\boxplus^{10}\Box(\boxplus^{\#3}\Diamond a \wedge (\boxplus^4\Diamond b \rightarrow \boxplus^5\Box(\neg c \wedge d)))$ can be represented by the tree in Figure 3.12.

The following strategy guarantees that evaluating a ground formula α remains in $O(|\alpha| * N + N^k)$ space. We travel the tree in a depth-first-search manner.

(1) When encountering a logical connective: evaluate the truth value of its sub-tree(s) and then combine the result using the semantics of the corresponding connective.

(2) When encountering a window operator $\boxplus^w$: extract a substream $S' = w(S, t)$ of the current stream S and store S' in a new place for evaluating the sub-tree of this operator.

(3) When encountering a $\Box$ or $\Diamond$ operator: iterate over the timeline T of the current window to determine the truth value of the sub-tree for each $t \in T$.

(4) When encountering an $@_t$ operator where t is a time point (and $t \in T$ where $S = (T, \upsilon)$): evaluate the sub-tree with reference to this time point.

(5) When encountering a leaf node: check the occurrence of the atom in the evaluation function at the current reference time point.

case (2) extends the space for setting up the environment for further checking of the sub-tree. The space for storing computed window $S' = w(S, t)$ is bounded by N, as a window is a substream of S. Furthermore, the computation of S' itself can be done in space N^k, for some constant $k \geq 1$, as it takes polynomial time in the size of S.

Furthermore, when visiting a node in the tree, we only need to consider the windows constructed by the window operators appearing on the path from the root to the current node. For other already visited branches, the space allocated for storing windows can be released. In case (3), we loop over all time points $t \in T$ and need only an iteration counter.

Therefore, at any time, the space used is bounded by the depth of the tree times N, plus the space for window evaluation; this yields $O(|\alpha| * N + N^k)$.

PSpace *hardness.* Given a QBF of the form

$$\Phi = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi(x_1, x_2, \ldots, x_n), \tag{A.1}$$

where $Q_i \in \{\exists, \forall\}$, we translate it into a LARS formula

$$\alpha = W_1 \boxplus^{\text{set}:x_1} W_2 \boxplus^{\text{set}:x_2} \cdots W_n \boxplus^{\text{set}:x_n} \phi(x_1, x_2, \ldots, x_n), \tag{A.2}$$

where for $1 \leq i \leq n$, $W_i = \Diamond$ if $Q_i = \exists$, $W_i = \Box$ if $Q_i = \forall$, and $\boxplus^{\text{set}:x_i}$ is a window operator with an associated window function set^{x_i} defined as follows. Given a stream $S = (T, \upsilon)$ and a time point $t \in \{0, 1\}$:

$$\text{set}^{x_i}(S, t) = (T', \upsilon'),$$

where $T' = T$ and for all $j \in T$:

$$\upsilon'(j) = \begin{cases} \upsilon(j) \setminus \{x_i\} & \text{if } t = 0, \\ \upsilon(j) & \text{if } t = 1. \end{cases}$$

That is, set^{x_i} removes x_i from the input stream S, if it is called at time $t = 0$, and it leaves S unchanged if it is called at $t = 1$; informally, this amounts to setting x_i to false (=0) and to true (=1), respectively.

Let now $S^\star = (T^\star, \upsilon^\star)$, where $T^\star = [0, 1]$ and $\upsilon^\star(0) = \upsilon^\star(1) = \{x_1, x_2, \ldots, x_n\}$ and let $M = \langle S^\star, W, \emptyset \rangle$, where $W = \{\boxplus^{\text{set}:x_1}, \ldots, \boxplus^{\text{set}:x_n}\}$.

Informally, $\Diamond \boxplus^{\text{set}:x_1} \alpha'$ is entailed at $t = 0$ (likewise, at $t = 1$), if either at $t = 0$ or $t = 1$, after applying the window function $\text{set}^{x_i}(S, t)$, the formula α' evaluates to true (=1) at t; that is, after either setting x_1 false (0) or to true (1), respectively. Dually, $\Box \boxplus^{\text{set}:x_1} \alpha'$ is entailed at $t = 0$ (likewise, at $t = 1$) iff α' evaluates to true for both setting x_1 to false and to true. The nesting of the formula (A.2) thus mimics the QBF Φ in (A.1), as follows:

(i) the two time points 0 and 1 encode the truth values false and true, respectively.

(ii) By starting with the function $\upsilon^\star$ as the set $\{x_1, x_2, \ldots, x_n\}$ and by removing in $\boxplus^{\text{set}:x_i}$ the atom x_i on the 0 branch and keeping x_i on the 1-branch, the evaluation of α can be seen as traversing a binary evaluation tree where the substream at each leaf node represents a complete truth assignment to $x_1, \ldots, x_n$. Figure 3.13 shows the tree with three variables.

(iii) The operator $\Diamond$ (resp. $\Box$) in front of $\boxplus^{\text{set}:x_i}$ simulates the quantifier $\exists$ (resp., $\forall$): some (resp. every) of the subtrees, rooted at the 0 or 1 child, must evaluate to true.

A subtree of the tree starting at the root that fulfills the condition (iii) each satisfies the formula $\phi(x_1, \ldots, x_n)$ at each leaf witnesses then that Φ evaluates to true.

More formally, we show by induction on $i = 0, \ldots, n$ that if $S_{n-i} = (T^\star, \upsilon_{n-i})$ is a substream of $S^\star$ such that $\upsilon_{n-i}(0) = \upsilon_{n-i}(1) \supseteq \{x_{n-i+1}, \ldots, x_n\}$, then for the truth assignment σ to $x_1, \ldots, x_{n-i}$ such that $\sigma(x_j) \Leftrightarrow x_j \in \upsilon_{n-i}(0)$, it holds that

$$M, S_{n-i}, t \Vdash W_{n-i+1} \boxplus^{\text{set}:x_{n-i+1}} \cdots W_n \boxplus^{\text{set}:x_n} \phi(x_1, x_2, \ldots, x_n) \tag{A.3}$$

for any $t \in \{0, 1\}$ iff

$$Q_{n-i+1} x_{n-i+1} \cdots Q_n x_n \phi(\sigma(x_1), \ldots, \sigma(x_{n-i}), x_{n-i+1}, \ldots, x_n), \tag{A.4}$$

evaluates to true; here $S_0 = S^\star$.

For $i = 0$, the stream S_n is a complete truth assignment to $x_1, \ldots, x_n$ and by construction the claim holds. For the inductive step, suppose the statement holds for i and consider $i + 1$. By applying $\text{set}^{x_{n-i}}$ on $S_{n-(i+1)+1} = S_{n-i}$ at $t = 0$, a stream of form S_{n-i+1} results, where x_{n-i} is set to false; thus by the induction hypothesis

$$\begin{aligned}
& M, S_{n-i}, 0 \Vdash \boxplus^{\text{set}:x_{n-i}} W_{n-i+1} \boxplus^{\text{set}:x_{n-i+1}} \cdots W_n \boxplus^{\text{set}:x_n} \phi(x_1, x_2, \ldots, x_n) \\
\text{iff} \quad & M, S_{n-i+1}, 0 \Vdash W_{n-i+1} \boxplus^{\text{set}:x_{n-i+1}} \cdots W_n \boxplus^{\text{set}:x_n} \phi(x_1, x_2, \ldots, x_n) \\
\text{iff} \quad & Q_{n-i+1} x_{n-i+1} \cdots Q_n x_n \phi(\sigma(x_1), \ldots, \sigma(x_{n-i}), x_{n-i+1}, \ldots, x_n) \text{ evaluates to true,}
\end{aligned}$$

where $\sigma(x_{n-i}) = 0$; for $t = 1$ the argument is analogous, where "0" is replaced by "1". Hence by definition of $\Diamond$ resp. $\Box$, we obtain that (A.3) holds iff (A.4) holds. This proves the claim.

For $n = 0$, as $S_0 = S^\star$ we then obtain that $M, S^\star, t \Vdash \alpha$ for α in (A.2) and $t \in \{0, 1\}$ iff Φ in (A.1) evaluates to true. As $M, S^\star$, α and t are computable in polynomial time from Φ, it follows that deciding $M, S, t \Vdash \alpha$, i.e., model checking for LARS formulas, is PSpace-hard.

Proof of Theorem 8. PSpace membership follows from the fact that we can guess an evaluation function υ on T and then perform a model check $M, S, t \Vdash \alpha$ where $S = (T, \upsilon)$; relative to the set $\mathcal{A}'$ of atoms, the guess for υ has polynomial size, and thus the combined guess and check algorithm can run in NPSpace; as NPSpace = PSpace by Savitch's result [Sav70], it is thus in PSpace.

The PSpace-hardness follows from a simple reduction of model checking $M, S, t \Vdash \alpha$, where from the proof of Theorem 7 w.l.o.g. $S = S^\star = (T^\star, \upsilon^\star)$: we construct

$$\alpha_S = \alpha \wedge \bigwedge_{t \in T^\star, p \in \upsilon(t)} @_t p \wedge \bigwedge_{t \in T^\star, p \in \mathcal{A}' \setminus \upsilon(t)} @_t \neg p$$

i.e., fix the possible valuation to υ, and ask for an evaluation function υ' on $T^\star$ s.t. $M, (T^\star, \upsilon'), t \Vdash \alpha_S$.

A.2 Relating LARS to other Formalisms

A.2.1 Continuous Query Language (CQL)

In this section, we give additional details on the translation from CQL to Datalog, resp. LARS, and the presented CQL query q of Example 35. First, applying *rel* on q gives us the following SQL query:

```
SELECT ID, plan.Y, TY
FROM  tram_part_ID_rows_1 , line, plan
WHERE tram_part_ID_rows_1.ID=line.ID AND
      line.L=plan.L AND
      tram_part_ID_rows_1.ST=plan.X AND
      TY=tram_part_ID_rows_1.T+plan.D AND
      NOT EXISTS
        (SELECT * FROM jam_range_20
         WHERE jam_range_20.ST=tram_part_ID_rows_1.ST)
```

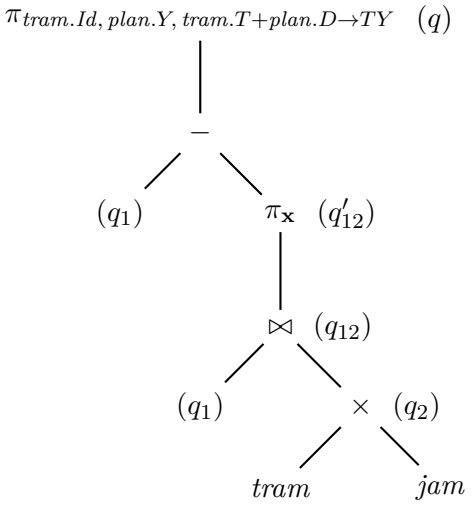
Translating this CQL query to relational algebra according to [DS90], we get the following relational algebra expression $\mathit{RelAlg}(\mathit{rel}(q))$, where *tram* abbreviates the relation name *tram_part_**ID**_rows_**1***, and *jam* abbreviates *jam_range_**20***:

$$q = \pi_{\mathit{tram.Id},\, \mathit{plan.Y},\, \mathit{tram.T+plan.D} \to \mathit{TY}}(q_1 - q'_{12}),$$

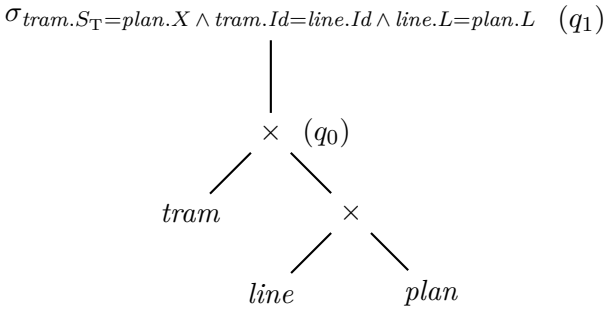
where

$$\begin{aligned}
q_1 &= \sigma_{\mathit{tram.S_T=plan.X} \,\wedge\, \mathit{tram.Id=line.Id} \,\wedge\, \mathit{line.L=plan.L}}\, q_0\,, \\
q_0 &= \mathit{tram} \times \mathit{line} \times \mathit{plan}\,, \\
q'_{12} &= \pi_{\mathit{tram.Id},\mathit{tram.S_T},\mathit{tram.T},\mathit{line.Id},\mathit{line.L},\mathit{plan.L},\mathit{plan.X},\mathit{plan.Y},\mathit{plan.D}}\, q_{12}\,, \\
q_{12} &= q_1 \bowtie q_2\,, \\
q_2 &= \mathit{tram} \times \mathit{jam}\,.
\end{aligned}$$

Figure A.1 shows the syntactic expression tree. Note that different such translations might be considered, e.g., by considering other orders in cross products, or joining earlier, etc. However, no semantic differences arise from such optimizations and thus no further discussion is needed for our purposes.



(a) Main tree



(b) Sub tree for (q_1)

Figure A.1: Relational algebra expression in tree representation. In projection node (q'_{12}) $\mathbf{x}$ is $tram.Id$, $tram.S_{\mathrm{T}}$, $tram.T$, $line.Id$, $line.L$, $plan.L$, $plan.X$, $plan.Y$, $plan.D$.

Using the translation in [GUW09] the translated Datalog program

$$\Delta_D(q) = Dat(RelAlg(rel(q)))$$

is the one in Example 36.

Without loss of generality, i.e., due to possible renamings, we assume in the sequel that relation names $\mathtt{B}_i$ and stream names $\mathtt{S}_i$ are pairwise distinct.

Proof of Theorem 17. Before going into the details, let us note that the relational algebra expression is tree-shaped and thus $\Delta_D(Q)$ is an acyclic program (i.e., there is not cyclic recursion through rules). Furthermore, *Dat* only creates atomic rule heads; therefore $\Delta_D(Q)$ is also definite (i.e., each rule head consists of an atom). The translation

$\Delta_L(Q)$ only adds a stratified layer to $\Delta_D(Q)$, i.e., the snapshot rules. Thus, both translations $\Delta_D(Q)$ and $\Delta_L(Q)$ amount to stratified theories and have a unique answer set resp. answer stream relative to given input data.

A correspondence between the CQL results $res(Q,t)$ and the answer set of $\Delta_L(Q)$ at time t obtained by the following steps.

(1) First, we construct the input for the translated Datalog program, i.e., the atoms reflecting the static relations and those obtained from snapshots. Correctness and completeness of the (compound) translation – from SQL to Relational Algebra and from the latter to Datalog – establishes a correspondence between the unique answer set of the Datalog program and the results of the CQL queries (Lemma 11).

(2) Lemma 12 shows that these atoms need not be provided as such, but can be derived by snapshot rules in LARS itself.

(3) Lemma 13 guarantees that the unique answer set of the Datalog encoding $\Delta_D(Q)$, given snapshot relations as input, corresponds with the unique answer stream of the LARS encoding $\Delta_L(Q)$, given the stream as input.

(4) By combining these lemmas, we obtain the desired correspondence between the results of CQL queries and the answer stream of respective LARS programs.

More formally, let Q be a set of CQL queries evaluated on static relations $\mathtt{B} = \mathtt{B}_1, \ldots, \mathtt{B}_m$ and input streams $\mathtt{S} = \mathtt{S}_1, \ldots, \mathtt{S}_n$ at a time point t. Without loss of generality, assume that to each input stream $\mathtt{S}_i$ only one of the CQL window functions in the first column of Table 4.1 (with window parameters replaced by values) is applied. We denote by $\mathtt{WINDOW}_i$ the CQL window function applied on stream $\mathtt{S}_i$ and by $\mathtt{WINDOW}_i(\mathtt{S}_i, t)$ the snapshot obtained for $\mathtt{S}_i$ at time t after the S2R operator. That is, $\mathtt{WINDOW}_i(\mathtt{S}_i, t)$ contains the selected tuples. (In case there is the need to apply two different windows on the same input stream, one can equivalently take a renamed copy of the stream.) The following defines an according set of input facts for the Datalog program $\Delta_D(Q)$ at t:

$$F(\mathtt{B}, \mathtt{S}, t) = \Delta(\mathtt{B}) \cup \{rel(\mathtt{S}_i)(\mathbf{c}) \mid \mathbf{c} \in \mathtt{WINDOW}_i(\mathtt{S}_i, t)\}$$

where $\Delta(\mathtt{B}) = \{b_i(\mathbf{c}) \mid \mathbf{c} \in \mathtt{B}_i\}$ as before. That is, $rel(\mathtt{S}_i)(\mathbf{c})$ is an atom corresponding to tuple $\mathbf{c}$ from the (snapshot of) stream $\mathtt{S}_i$. For any query $q \in Q$, let $\hat{q}$ denote the head predicate of the rule in $\Delta_D(q)$ corresponding to the root of the relational algebra expression for q. The following lemma establishes the correspondence between the answer set of $\Delta_D(Q)$ and CQL results $res(Q,t)$ for Q at t; it follows from the correctness and completeness of *RelAlg* and *Dat*.

Lemma 11 *Let A be the unique answer set of $\Delta_D(Q) \cup F(\mathtt{B}, \mathtt{S}, t)$, i.e., the translated Datalog program plus input facts as obtained from static relations and snapshot relations at time t. Then, for every query $q \in Q$, $\hat{q}(\mathbf{c}) \in A$ iff $\mathbf{c} \in res(Q,t)$.*

The next lemma states that the snapshot semantics of CQL's S2R operator is faithfully captured by LARS formulas as given in Table 4.1.

Lemma 12 *Assume static relations* $\mathtt{B} = \mathtt{B}_1, \ldots, \mathtt{B}_m$ *and streams* $\mathtt{S} = \mathtt{S}_1, \ldots, \mathtt{S}_n$*, LARS window functions* W *corresponding to those in CQL queries* Q*, and let* $M = \langle S, W, \Delta(\mathtt{B})\rangle$ *be a structure such that* $\Delta(\mathtt{S}) \subseteq S$*. Moreover, let* w_i *be the LARS window function corresponding to* $\mathtt{WINDOW}_i$ *due to Table 4.1. Then,* $\mathbf{c} \in \mathtt{WINDOW}_i(\mathtt{S}_i, t)$ *iff* $M, \Delta(\mathtt{S}), t \Vdash \boxplus^{w_i}\Diamond s(\mathbf{c})$*.*

Proof. Consider an element $\langle \mathbf{c}, t'\rangle$ in stream $\mathtt{S}_i$ which corresponds to the inclusion $s(\mathbf{c}) \in \upsilon(t')$ in $\Delta(\mathtt{S}_i) = (T, \upsilon)$. That is, we can view LARS streams as notational variant of CQL streams, and vice versa. It follows directly from their definitions that window functions $\mathtt{WINDOW}_i$ and w_i (due to Table 4.1) select the same elements. Thus, $\langle \mathbf{c}, t'\rangle$ is in the CQL window iff it is in the LARS window. The appearance time t' is abstracted away in CQL by the S2R operator, which amounts to existential quantification with $\Diamond$ in LARS. Thus, $\mathbf{c} \in \mathtt{WINDOW}_i(\mathtt{S}_i, t)$ iff $\boxplus^{w_i}\Diamond s(\mathbf{c})$. □

We now establish correspondence between the encodings $\Delta_D(Q)$ and $\Delta_L(Q)$ at a time point t.

Lemma 13 *Let* Q *be a set of CQL queries,* $\mathtt{B} = \mathtt{B}_1, \ldots, \mathtt{B}_m$ *static relations,* $\mathtt{S} = \mathtt{S}_1, \ldots, \mathtt{S}_n$ *be input streams, and let* t *be a time point. Moreover, let* A *be the unique answer set of* $\Delta_D(Q) \cup F(\mathtt{B}, \mathtt{S}, t)$ *and* $I = (T, \upsilon)$ *the unique answer stream of* $\Delta_L(Q)$ *for* $\Delta(\mathtt{S})$ *at* t *(using structure* $M = \langle I, W, \Delta(\mathtt{B})\rangle$*). Then, for all* $q \in Q$*,* $\hat{q}(\mathbf{c}) \in A$ *iff* $\hat{q}(\mathbf{c}) \in \upsilon(t)$*.*

Proof. From Lemma 12 we obtain that snapshot relations $rel(s)$ can be derived directly from the input stream $\Delta(\mathtt{S})$, using snapshot rules of form (4.4), instead of providing them explicitly. That is, the LARS subprogram $\Delta_L(Q) \setminus \Delta_D(Q)$ essentially computes $F(\mathtt{B}, \mathtt{S}, t)$ and associates these snapshot atoms of form $rel(s)(\mathbf{c})$ with time point t in the answer stream $I = (T, \upsilon)$: if an atom $s(\mathbf{c})$ is contained in the window, $\boxplus^{w_i}\Diamond s(\mathbf{c})$ holds and in order for the snapshot rule to be satisfied, $rel(s)(\mathbf{c})$ must be contained in $\upsilon(t)$. Due to the minimality and supportedness of I, $rel(s)(\mathbf{c})$ is contained in $\upsilon(t)$ *only* in this case. Moreover, no time point $t' \neq t$ will be assigned with any snapshot atom $rel(s)(\mathbf{c})$ (due to the form of snapshot rules). The snapshot atoms occur as rule heads only in snapshot rules and no rules other than snapshot rules distinguish $\Delta_D(Q)$ and $\Delta_L(Q)$. We thus conclude for any element $\langle \mathbf{c}, t\rangle$ that will be selected by $\mathtt{WINDOW}_i$, $\mathbf{c} \in F(\mathtt{S}, \mathtt{B}, t)$ by definition, and $s_i(\mathbf{c}) \in \upsilon(t)$ as argued. As the semantics after the S2R operator is captured by $\Delta_D(Q)$, we obtain the desired correspondence; in particular for output predicate $\hat{q}$ that $\hat{q}(\mathbf{c})$ is in the answer set of the Datalog encoding iff it is in $\upsilon(t)$ of the answer stream of the LARS encoding. □

A.2.2 Complex Event Processing: ETALIS

We now develop the results of Section 4.4. We start by formal definitions for expressing temporal intervals in LARS, give a review of ETALIS, and then present the translation form ETALIS to LARS.

Intervals in LARS

The formal definition of the interval window used in $\boxplus^{[\ell,u]}$ (i.e., $\boxplus^{w_{[\ell,u]}}$) follows.

Definition 40 (Interval window) *Let $S = (T, \upsilon)$ be a stream, where $T = [t_{min}, t_{max}]$, $t \in T$ and let $\ell \leq u \in \mathbb{N} \cup \{\infty\}$. The* interval window for $[\ell, u]$ *(at time t) is defined by*

$$w_{[\ell,u]}(S, t) = (T', \upsilon|_{T'}), \tag{A.5}$$

where $T' = [t_\ell, t_u]$ such that $t_\ell = \max\{t_{min}, \ell\}$ and $t_u = \min\{u, t_{\max}\}$.

Based on this, we introduce a syntactic shortcut $[\![\ell, u]\!]$ to test whether a formula α holds throughout an interval, and $\langle\!\langle \ell, u \rangle\!\rangle$ that ensures that α does not hold in a bigger interval.

Definition 41 *Let α be a formula and $\ell, u \in \mathbb{N} \cup \{\infty\}$. Then, we define*

$$[\![\ell, u]\!]\alpha \;:=\; \boxplus^{w_{[\ell,u]}} \Box\alpha\,, \tag{A.6}$$

and

$$\langle\!\langle \ell, u \rangle\!\rangle\alpha \;:=\; [\![\ell, u]\!]\alpha \;\wedge\; @_{\ell-1}\neg\alpha \;\wedge\; @_{u+1}\neg\alpha \tag{A.7}$$

That is, to test whether the formula α always holds in the window of the timeline $[\ell, u]$, we can evaluate $[\![\ell, u]\!]\alpha$. Similarly, if $\langle\!\langle \ell, u \rangle\!\rangle\alpha$ holds, then $[\ell, u]$ is the largest window containing $[\ell, u]$ in which α always holds.

We observe that for the evaluation of a formula $[\![\ell, u]\!]\alpha$ or $\langle\!\langle \ell, u \rangle\!\rangle\alpha$, the current reference time point t is irrelevant. In the sequel, we assume $W = \{w_{[\ell,u]} \mid \ell, u \in \mathbb{N},\, l \leq u\}$.

Lemma 14 *Let $M = \langle S, W, \emptyset \rangle$ be a structure, $S = (T, \upsilon)$, and α be a formula. Moreover, let $\phi \in \{[\![\ell, u]\!]\alpha, \langle\!\langle \ell, u \rangle\!\rangle\alpha\}$, where $\ell \leq u \in \mathbb{N} \cup \{\infty\}$ and let $t \in T$. If $M, S, t \Vdash \phi$ holds, then $M, S, t' \Vdash \phi$ holds for all $t' \in T$.*

Thus, we can arbitrarily choose any time point $t \in T$ for the evaluation of a formula of form $[\![\ell, u]\!]\alpha$, respectively $\langle\!\langle \ell, u \rangle\!\rangle\alpha$.

Review: ETALIS

We recall ETALIS definitions from [AFR+10], restricting to the essential ground case and natural numbers for time points.

An ETALIS *event stream* ϵ maps *atomic events* (i.e., atoms) to sets of time points, i.e., non-negative natural numbers. Based on atomic events, rules can make use of *event patterns* to express interval relations similar to those in [All83]. For instance the pattern x SEQ y matches interval $\langle t_1, t_4 \rangle$, if there are intervals $\langle t_1, t_2 \rangle$ and $\langle t_3, t_4 \rangle$ assigned to events x and y, respectively, such that $t_2 < t_3$. The pattern x AND y selects the temporal overlaps, x EQUALS y selects intervals assigned to both x and y, etc. We formalize these patterns later (cf. Table A.1). Notably, a *complex event* is an atom that is associated with (closed) intervals $[t_1, t_2]$, which are represented as pairs $\langle t_1, t_2 \rangle$ of integers (time

points). We deliberately use two kinds of notations for intervals, as we will elaborate on different possibilities to employ them. An interpretation $\mathcal{I}$ is a function that maps atoms to sets of pairs $\langle t_1, t_2\rangle \in \mathbb{N} \times \mathbb{N}$ representing intervals $[t_1, t_2]$. Let $r = a \leftarrow pt$ be a rule, where a is an atom and pt an event pattern. Then, interpretation $\mathcal{I}$ *satisfies* r, denoted by $I \models_r \epsilon$, if all intervals assigned to pt are also assigned to a. Moreover, given a *rule base* $\mathcal{R}$ (i.e., a set of rules), $\mathcal{I}$ is a *model* of $\epsilon, \mathcal{R}$, denoted by $\mathcal{I} \models_\epsilon \mathcal{R}$, if for all ground atoms $a \in \mathcal{A}$

(C1) $\langle t, t\rangle \in \mathcal{I}(a)$ for all $t \in \epsilon(a)$, and

(C2) $\mathcal{I}(pt) \subseteq \mathcal{I}(a)$ for every rule $a \leftarrow pt \in \mathcal{R}$.[1]

That is, by (C1), an atomic event a appearing at time point t must be captured by an interval $\langle t, t\rangle$. Condition (C2) says that the intervals that match a rule body must be assigned to the rule head.

For the definition of minimal models, we recall the notation $\mathcal{I}|_n$ from [AFR+10], where $n \in \mathbb{N}$: for an atom $a \in \mathcal{A}$, $\mathcal{I}|_n(a) = \mathcal{I}(a) \cap \{\langle t_1, t_2\rangle \mid t_2 - t_1 \leq n\}$, i.e., the subset of the intervals assigned to a that have a length of at most n. A model $\mathcal{J}$ is *preferred* to model $\mathcal{I}$, if there exists a time point $n \in \mathbb{N}$ such that $\mathcal{J}|_n \subset \mathcal{I}|_n$; more formally,

$$\exists n \in \mathbb{N}\ \left(\forall\, a \in \mathcal{A}\ (\mathcal{J}|_n(a) \subseteq \mathcal{I}|_n(a))\ \wedge\ \exists\, a \in \mathcal{A}\ (\mathcal{J}|_n(a) \subset \mathcal{I}|_n(a))\right).$$

Throughout, we will only be interested in minimal models $\mathcal{I}$, i.e., where there does not exist a preferred model $\mathcal{J}$. Assuming that every rule $a \leftarrow pt$ has a unique head atom a, this means that the intervals of a are exactly those of pt.

Example 82 Consider the rule $a \leftarrow x$ SEQ y and the event stream ϵ, where $\epsilon(x) = \{2\}$ and $\epsilon(y) = \{5, 6\}$. The minimal model $\mathcal{I}$ is given as follows: $\mathcal{I}(x) = \{\langle 2, 2\rangle\}$, $\mathcal{I}(y) = \{\langle 5, 5\rangle, \langle 6, 6\rangle\}$. This covers (C1). For (C2), we have $\mathcal{I}(a) = \{\langle 2, 5\rangle, \langle 2, 6\rangle\}$. Note that for a model $\mathcal{I}'$ which assigns for a only $\mathcal{I}'(a) = \{\langle 2, 6\rangle\}$ we get $\mathcal{I}'|_3 = \emptyset \subset \mathcal{I}|_3 = \{\langle 2, 5\rangle\}$. That is, $\mathcal{I}'$ is a smaller interpretation. However, $\mathcal{I}'$ is not a *model* of $\epsilon, \mathcal{R}$, since the interval $\langle 2, 5\rangle$ matches the pattern x SEQ y and thus must be assigned to a. ∎

Notably, the minimal model according to the above definition is computable by a fixed-point iteration.

Lemma 15 *Let $\mathcal{R}$ be a program without negation and let $\mathcal{I}$ be the minimal model of $\epsilon, \mathcal{R}$ and $\mathcal{I}'$ be an arbitrary model of $\epsilon, \mathcal{R}$. Then, for each $a \in \mathcal{A}$, $\mathcal{I}(a) \subseteq \mathcal{I}'(a)$, i.e., the minimal model is contained in every model.*

The original set of event patterns (such as SEQ) can be found in [AFR+10]. We use a ground version of these, as shown in Table A.1 along with their LARS translation. We will discuss this later. Comments on the NOT pattern, which we exclude, will follow in the final discussion (cf. Section A.2.2).

[1] In [AFR+10] and in other papers on ETALIS, the containment is (as confirmed by an author) mistakenly stated in the other direction.

r	$a \leftarrow n \quad (n \in \mathbb{N})$
$\mathcal{I}(a)$	$\{\langle n, n\rangle\}$
Δ^r	$[\![t,t]\!]a \leftarrow \langle\!\langle t,t\rangle\!\rangle \mu^n$. (aux. atom $\mu^n \in \upsilon(t)$ if $t = n$)
r	$a \leftarrow (x).n$
$\mathcal{I}(a)$	$\mathcal{I}(x) \cap \{\langle t_1, t_2\rangle \mid t_2 - t_1 \leq n\}$
Δ^r	$[\![t_1,t_2]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x,\ t_2 - t_1 \leq n.$
r	$a \leftarrow x$ SEQ y
$\mathcal{I}(a)$	$\{\langle t_1, t_4\rangle \mid \langle t_1,t_2\rangle \in \mathcal{I}(x),\ \langle t_3,t_4\rangle \in \mathcal{I}(y),\ t_2 < t_3\}$
Δ^r	$[\![t_1,t_4]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x, \langle\!\langle t_3,t_4\rangle\!\rangle y, t_2 < t_3.$
r	$a \leftarrow x$ AND y
$\mathcal{I}(a)$	$\{\langle \ell, u\rangle \mid \langle t_1,t_2\rangle \in \mathcal{I}(x), \langle t_3,t_4\rangle \in \mathcal{I}(y),$ $\ell = \min\{t_1,t_3\}, u = \max\{t_2,t_4\}\}$
Δ^r	$[\![\ell,u]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x, \langle\!\langle t_3,t_4\rangle\!\rangle y,$ $\ell = \min\{t_1,t_3\}, u = \max\{t_2,t_4\}.$
r	$a \leftarrow x$ PAR y
$\mathcal{I}(a)$	$\{\langle \ell, u\rangle \mid \langle t_1,t_2\rangle \in \mathcal{I}(x), \langle t_3,t_4\rangle \in \mathcal{I}(y),$ $\ell = \min\{t_1,t_3\}, u = \max\{t_2,t_4\}$ $s = \max\{t_1,t_3\}, e = \min\{t_2,t_4\}, s < e\}$
Δ^r	$[\![\ell,u]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x, \langle\!\langle t_3,t_4\rangle\!\rangle y,$ $\ell = \min\{t_1,t_3\}, u = \max\{t_2,t_4\}$ $s = \max\{t_1,t_3\}, e = \min\{t_2,t_4\},\ s < e.$
r	$a \leftarrow x$ OR y
$\mathcal{I}(a)$	$\mathcal{I}(x) \cup \mathcal{I}(y)$
Δ^r	$[\![t_1,t_2]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x.$
	$[\![t_1,t_2]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle y.$
r	$a \leftarrow x$ EQUALS y
$\mathcal{I}(a)$	$\mathcal{I}(x) \cap \mathcal{I}(y)$
Δ^r	$[\![t_1,t_2]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x, \langle\!\langle t_1,t_2\rangle\!\rangle y.$
r	$a \leftarrow x$ MEETS y
$\mathcal{I}(a)$	$\{\langle t_1, t_3\rangle \mid \langle t_1,t_2\rangle \in \mathcal{I}(x),\ \langle t_2,t_3\rangle \in \mathcal{I}(y)\}$
Δ^r	$[\![t_1,t_3]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x, \langle\!\langle t_2,t_3\rangle\!\rangle y.$
r	$a \leftarrow x$ DURING y
$\mathcal{I}(a)$	$\{\langle t_3, t_4\rangle \mid \langle t_1,t_2\rangle \in \mathcal{I}(x),\ \langle t_3,t_4\rangle \in \mathcal{I}(y),$ $t_3 < t_1 < t_2 < t_4\}$
Δ^r	$[\![t_3,t_4]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x, \langle\!\langle t_3,t_4\rangle\!\rangle y,$ $t_3 < t_1,\ t_1 < t_2,\ t_2 < t_4.$
r	$a \leftarrow x$ STARTS y
$\mathcal{I}(a)$	$\{\langle t_1, t_3\rangle \mid \langle t_1,t_2\rangle \in \mathcal{I}(x),\ \langle t_1,t_3\rangle \in \mathcal{I}(y),\ t_2 < t_3\}$
Δ^r	$[\![t_1,t_3]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x, \langle\!\langle t_1,t_3\rangle\!\rangle y,\ t_2 < t_3.$
r	$a \leftarrow x$ FINISHES y
$\mathcal{I}(a)$	$\{\langle t_1, t_3\rangle \mid \langle t_2,t_3\rangle \in \mathcal{I}(x),\ \langle t_1,t_3\rangle \in \mathcal{I}(y),\ t_1 < t_2\}$
Δ^r	$[\![t_1,t_3]\!]a \leftarrow \langle\!\langle t_2,t_3\rangle\!\rangle x, \langle\!\langle t_1,t_3\rangle\!\rangle y,\ t_1 < t_2.$
r	$a \leftarrow x$ WHERE b
$\mathcal{I}(a)$	$\mathcal{I}(x)$, if $b = \mathit{true}$; else $\emptyset$
Δ^r	$[\![t_1,t_2]\!]a \leftarrow \langle\!\langle t_1,t_2\rangle\!\rangle x,$ if $b = \mathit{true}$; else void

Table A.1: Definitions for ground, *unnested* ETALIS rules r, rule head interpretations $\mathcal{I}(a)$, and their LARS translations Δ^r.

Definition 42 (Unnested rules) *A rule r is called* unnested*, if its form is listed in Table A.1. A rule base $\mathcal{R}$ is called* unnested*, if each rule $r \in \mathcal{R}$ is unnested.*

This restriction serves to ease technical presentation but imposes no semantic limitations.

For an interpretation $\mathcal{I}$ and a rule base $\mathcal{R}$, let $\mathcal{I}|_{\mathcal{R}}$ be the interpretation defined as

$$\mathcal{I}|_{\mathcal{R}}(a) = \begin{cases} \mathcal{I}(a) & \text{if } a \text{ appears in } \mathcal{R} \\ \emptyset & \text{else}\,, \end{cases}$$

for every atom $a \in \mathcal{A}$, i.e., $\mathcal{I}|_{\mathcal{R}}$ limits the interpretation of $\mathcal{I}$ to atoms appearing in $\mathcal{R}$.

Lemma 16 *Let ϵ be an event stream. For every rule base $\mathcal{R}$, there exists an* unnested rule *base $\mathcal{R}'$ such that $\mathcal{I}'$ is a minimal model of $\epsilon, \mathcal{R}'$ iff $\mathcal{I}'|_{\mathcal{R}}$ is a minimal model of $\epsilon, \mathcal{R}$.*

Intuitively, one can make use of a fresh atom for each nested appearance of an ETALIS pattern, and create a new rule for this. In a model, these atoms will be assigned the same set of intervals as an according pattern within a different rule (appearing there as sub-pattern). Consequently, when filtering out these auxiliary atoms, we arrive at the same minimal model.

Example 83 Consider the rule base $\mathcal{R}$ containing the single nested rule

$$r : a \leftarrow x \text{ PAR } (y \text{ SEQ } z).$$

Instead, we can use a rule base $\mathcal{R}'$ containing the following unnested rules r_1 and r_2, employing an auxiliary atom h:

$$\begin{aligned} r_1 &: a \leftarrow x \text{ PAR } h \\ r_2 &: h \leftarrow y \text{ SEQ } z \end{aligned}$$

For any event stream ϵ, a *minimal* model of $\mathcal{R}'$ will assign *exactly* the matching intervals for y SEQ z to h, if h is not the head of another rule and not appearing in ϵ. Given these assumptions, h can thus be alternatively used to describe the pattern y SEQ z. ∎

Moreover, the following concept will be useful.

Definition 43 (Head unique) *An ETALIS rule base $\mathcal{R}$ is called* head unique*, if for every pair of rules $a \leftarrow pt$ and $a' \leftarrow pt'$ in $\mathcal{R}$, $pt \neq pt'$ implies $a \neq a'$.*

By assuming head uniqueness, we have for a rule $a \leftarrow pt$ in the minimal model $\mathcal{I}$ the identity $\mathcal{I}(pt) = \mathcal{I}(a)$. The following lemma states that confining to head unique rules bases also comes without semantic restrictions.

Lemma 17 *Let ϵ be an event stream. For every rule base $\mathcal{R}$, there exists a* head unique *rule base $\mathcal{R}'$ such that $\mathcal{I}'$ a minimal model of $\epsilon, \mathcal{R}'$ iff $\mathcal{I}'|_{\mathcal{R}}$ is a minimal model of $\epsilon, \mathcal{R}$.*

Proof (Sketch). We can replace every set of rules

$$a \leftarrow pt_1 \quad \ldots \quad a \leftarrow pt_n$$

in $\mathcal{R}$ with a common head atom a by a set of rules

$$a_1 \leftarrow pt_1 \quad \ldots \quad a_n \leftarrow pt_n$$

in $\mathcal{R}'$ with fresh, distinct atoms $a_1, \ldots, a_n$ and introduce another rule

$$a \leftarrow a_1 \text{ OR } \ldots \text{ OR } a_n.$$

If $\mathcal{I}$ and $\mathcal{I}'$ are minimal models of $\mathcal{R}$ and $\mathcal{R}'$, respectively, then we have $\langle t_1, t_2 \rangle \in \mathcal{I}(a)$ iff $\langle t_1, t_2 \rangle \in \mathcal{I}'(a)$. □

We define the following class of rule bases.

Definition 44 *By* **R** *we denote the class of ETALIS rule bases that are (i) unnested and (ii) head unique.*

Due to Lemmas 16 and 17 we get the following result.

Proposition 8 *For every rule base $\mathcal{R}$ there exists a corresponding rule base $\mathcal{R}' \in \mathbf{R}$ such that for any event stream ϵ, $\mathcal{I}'$ is a minimal model of $\epsilon, \mathcal{R}'$ iff $\mathcal{I}'|_{\mathcal{R}}$ is a minimal model of $\epsilon, \mathcal{R}$.*

For simplicity, we will thus confine to class **R** in the sequel. We shall note that ETALIS does not distinguish between extensional predicates (of the input stream) and intensional predicates (of the rules' heads), as we do. For strict compliance with our framework, we can always map any atom a to a designated intensional atom a' and use a' instead of a in the according rule translations. Given an ETALIS event stream ϵ, we assume that all rule heads do not make use of atoms appearing in the event stream ϵ.

Translation from ETALIS to LARS

Given an ETALIS event stream ϵ, respectively an interpretation $\mathcal{I}$, we use the following translations to obtain a LARS stream Δ^ϵ, respectively LARS interpretation $\Delta^\mathcal{I}$.

Translation of stream: $\epsilon \mapsto \Delta^\epsilon$. Let $\tau(\epsilon) = \{t \in \mathbb{N} \mid a \in \mathcal{A}, t \in \epsilon(a)\}$, i.e., the time points at which atomic events occur in ϵ. For an event stream $\epsilon : \mathcal{A} \to 2^\mathbb{N}$, we define a data stream $\Delta^\epsilon = (T^\epsilon, \upsilon^\epsilon)$, where

$$\begin{aligned} T^\epsilon &= [\min \tau(\epsilon), \max \tau(\epsilon)], \text{ and} \\ \upsilon^\epsilon &= \{t \mapsto a \mid a \in \mathcal{A}, t \in \epsilon(a)\}. \end{aligned}$$

Translation of interpretation: $\mathcal{I} \mapsto \Delta^\mathcal{I}$. In the translation of an ETALIS interpretation $\mathcal{I} : \mathcal{A} \to 2^{\mathbb{N}\times\mathbb{N}}$, we use a representation which maps time points to according sets of

atoms. We use $\tau(\mathcal{I}) = \{t_1, t_2 \mid a \in \mathcal{A}, \langle t_1, t_2\rangle \in \mathcal{I}(a)\}$ and define the LARS interpretation stream $\Delta^{\mathcal{I}} = (T, \upsilon)$, where

$$\begin{aligned} T &= [\min \tau(\mathcal{I}), \max \tau(\mathcal{I})], \text{ and} \\ \upsilon &= \{t \mapsto a \mid a \in \mathcal{A}, \langle t_1, t_2\rangle \in \mathcal{I}(a), t \in [t_1, t_2]\}. \end{aligned}$$

We observe that ϵ and Δ^{ϵ} equivalently formalize streams.

Lemma 18 *The mapping from ϵ to Δ^{ϵ} is bijective.*

The translation for interpretations defines a pointwise correspondence. The next lemma also follows directly from the definition of the translation. Furthermore, we let

$$\begin{aligned} tps(\mathcal{I}, a) &:= \bigcup_{\langle t_1, t_2\rangle \in \mathcal{I}(a)} [t_1, t_2] \\ &= \{t \in \mathbb{N} \mid \langle t_1, t_2\rangle \in \mathcal{I}(a), t \in [t_1, t_2]\}. \end{aligned}$$

Lemma 19 *Let $\mathcal{I}$ be an ETALIS interpretation, and let $\Delta^{\mathcal{I}} = (T, \upsilon)$ be the translated LARS interpretation stream. Then, $t \in tps(\mathcal{I}, a)$ iff $a \in \upsilon(t)$.*

This lemma serves to establish the correspondence between LARS and a stream reasoning view on ETALIS, where only single time points are considered instead of intervals (cf. Corollary 4, page 92). Using the presented intuitive translation, ETALIS can be captured fully only when for each atom a the assigned intervals $\mathcal{I}(a)$ are disjoint and have at least one time point between them. This is the intuition behind the following notion.

Definition 45 (Separability) *We say two (ETALIS) intervals $\langle t_1, t_2\rangle$ and $\langle t_3, t_4\rangle$ are* separable*, if $t_2 + 1 < t_3$ or $t_4 + 1 < t_1$. An interpretation $\mathcal{I}$ is said to be* separable*, if for all intensional atoms $a \in \mathcal{A}$, all intervals in $\mathcal{I}(a)$ are pairwise separable.*

Interpretations with overlapping or adjacent intervals (for the same atom) cannot be distinguished in LARS, where they are implicitly merged. If intervals are separable, no ambiguity arises.

Lemma 20 *Let $\mathcal{I}$ be a separable interpretation. Then, the mapping from $\mathcal{I}$ to $\Delta^{\mathcal{I}}$ is bijective.*

Proof. Given a LARS interpretation stream $I = (T^I, \upsilon^I)$, we define the following. First, we let

$$\begin{aligned} mxi(I, a) = \{[t_1, t_2] \subseteq T^I \mid\ & a \in \upsilon^I(t) \text{ for all } t \in [t_1, t_2], \\ & a \notin \upsilon^I(t_1{-}1), a \notin \upsilon^I(t_2{+}1)\}, \end{aligned}$$

i.e., the set of maximal intervals $[t_1, t_2]$ in T^I where a always holds. Then, we define an ETALIS interpretation $et(I)$ by

$$et(I) = \{a \mapsto \langle t_1, t_2\rangle \mid [t_1, t_2] \in mxi(I, a)\}.$$

ETALIS	LARS
event stream ϵ	data stream $\Delta^\epsilon = (T^\epsilon, \upsilon^\epsilon)$
interpretation $\mathcal{I}$	interpretation stream $\Delta^\mathcal{I} = (T, \upsilon)$
	interpretation $M = \langle \Delta^\mathcal{I}, W, \emptyset \rangle$,
	where $W = \{w_{[\ell,u]} \mid \ell, u \in \mathbb{N}, \ell \leq u\}$
rule base $\mathcal{R}$	program $\Delta^\mathcal{R}$

Table A.2: Mapping notation from ETALIS to LARS and back.

Let $\mathcal{I}$ be a separable interpretation, $\Delta^\mathcal{I} = (T, \upsilon)$ be the translated LARS interpretation stream, and let $\mathcal{J} = et(\Delta^\mathcal{I})$. We show that $\mathcal{I} = \mathcal{J}$, where we confine to intensional atoms. Bijection on the extensional part is covered by Lemma 18.

We first show $\mathcal{I} \subseteq \mathcal{J}$. Let $\langle t_1, t_2 \rangle \in \mathcal{I}(a)$ such that a is intensional. By definition of the translation from $\mathcal{I}$ to $\Delta^\mathcal{I} = (T, \upsilon)$, $a \in \upsilon(t)$ for all $t \in [t_1, t_2]$. Since $\mathcal{I}$ is separable, all other intervals $\langle t_3, t_4 \rangle \in \mathcal{I}(a)$ are separable w.r.t. $\langle t_1, t_2 \rangle$. As a consequence, $a \notin \upsilon(t')$ for $t' \in \{t_1 - 1, t_2 + 1\}$ and thus $[t_1, t_2] \in mxi(\Delta^\mathcal{I}, a)$. That is to say, $[t_1, t_2]$ is a maximal interval in T where a always holds, and thus $a \mapsto \langle t_1, t_2 \rangle \in et(\Delta^\mathcal{I})$, i.e., $\langle t_1, t_2 \rangle \in \mathcal{J}(a)$. Thus, $\mathcal{I}(a) \subseteq \mathcal{J}(a)$ holds for all atoms $a \in \mathcal{A}$.

We now show that $\mathcal{J} \subseteq \mathcal{I}$ holds. For the sake of contradiction, assume there exists an intensional atom $a \in \mathcal{A}$ and a pair $\langle t_1, t_2 \rangle \in \mathcal{J}(a)$ such that $\langle t_1, t_2 \rangle \notin \mathcal{I}(a)$. Since $\langle t_1, t_2 \rangle \in \mathcal{J}(a)$, $\langle t_1, t_2 \rangle$ is a maximal interval in T where a holds (by construction of function et). Thus, in $\Delta^\mathcal{I}$, a is assigned to all time points in $[t_1, t_2]$. Translation $\Delta^\mathcal{I}$ does not assign atoms to time points that are not covered in intervals in $\mathcal{I}$. That is to say, since $\langle t_1, t_2 \rangle \notin \mathcal{I}(a)$ there must be at least two intervals assigning (at least) interval $[t_1, t_2]$ to a. More formally, we have that set of intervals $\langle t_1^1, t_2^1 \rangle \ldots \langle t_1^n, t_2^n \rangle$ in $\mathcal{I}(a)$, where $n \geq 2$, such that $[t_1, t_2] \subseteq \bigcup_{k=1}^{n} [t_1^k, t_2^k]$. This means that the assignment of $[t_1, t_2]$ to a is due to an overlap of intervals for a, i.e., $\mathcal{I}(a)$ is not separable. This gives the contradiction, and we conclude that $\mathcal{J}(a) \subseteq \mathcal{I}(a)$ for all atoms $a \in \mathcal{A}$, and in conclusion that $\mathcal{I} = \mathcal{J}$. □

Similarly as for ETALIS event streams and interpretations, we define a translation to obtain LARS programs for ETALIS programs $\mathcal{R} \in \mathbf{R}$.

Translation of rule base $\mathcal{R} \mapsto \Delta^\mathcal{R}$. Table A.1 defines ground ETALIS rules r of form $a \leftarrow pt$, where the atom a is the head of the rule. The symbols x, y, z in the body patterns pt also denote atoms, $n \in \mathbb{N}$ and b is a boolean value. The ETALIS semantics defines that the atom a must be assigned at least to the declared intervals $\mathcal{I}(a)$.

Finally, the table gives translations to according LARS rules Δ^r. The atoms of the original rules are used there as well. In addition, time variables t_i encode the interpretation function of ETALIS. With this, we define $\Delta^\mathcal{R} = \bigcup_{r \in \mathcal{R}} \Delta^r$.

The following lemma notes a link between programs and their reducts. Recall that $\beta(r)$ is the conjunction representing the body of rule r.

Lemma 21 *An interpretation $M = \langle \Delta^\mathcal{I}, W, \emptyset \rangle$ is a model of $\Delta^\mathcal{R}$ at $t \in T$ iff M is a model of the reduct of $\Delta^\mathcal{R}$ w.r.t. M at t.*

Proof. The reduct of $\Delta^{\mathcal{R}}$ w.r.t. $M = \langle \Delta^{\mathcal{I}}, W, \emptyset \rangle$ at time t is the program

$$R = \{\Delta^r \in \Delta^{\mathcal{R}} \mid M, \Delta^{\mathcal{I}}, t \Vdash \beta(\Delta^r)\}.$$

The only-if-part is straightforward, since $R \subseteq \Delta^{\mathcal{R}}$. For the if-part, consider a rule $\Delta^r \in \Delta^{\mathcal{R}} \setminus R$ with head h. Since $M, \Delta^{\mathcal{I}}, t \nVdash \beta(\Delta^r)$, we get $M, \Delta^{\mathcal{I}}, t \Vdash \beta(\Delta^r) \rightarrow h$ by the entailment definition for implication ($\rightarrow$). □

Lemma 22 ($\mathcal{I}$ sep. model ⇒ $\Delta^{\mathcal{I}}$ model) *Let ϵ be an event stream, $\mathcal{R} \in \mathbf{R}$ and let $\mathcal{I}$ be a model of $\epsilon, \mathcal{R}$. Moreover, let $M = \langle \Delta^{\mathcal{I}}, W, \emptyset \rangle$, where $\Delta^{\mathcal{I}} = (T, \upsilon)$, and $t \in T$. If $\mathcal{I}$ is separable, then M is a model of $\Delta^{\mathcal{R}}$ for Δ^{ϵ} at t.*

Proof. Let ϵ be an event stream, $\mathcal{R} \in \mathbf{R}$, $\mathcal{I}$ be a model of $\epsilon, \mathcal{R}$, $M = \langle \Delta^{\mathcal{I}}, W, \emptyset \rangle$, where $\Delta^{\mathcal{I}} = (T, \upsilon)$, and let $t \in T$. Assume that $\mathcal{I}$ is separable. We have to show that M is a model of $\Delta^{\mathcal{R}}$ for Δ^{ϵ} at t, i.e., (i) $\Delta^{\epsilon} \subseteq \Delta^{\mathcal{I}}$, and (ii) $M, t \models \Delta^r$ for each $\Delta^r \in \Delta^{\mathcal{R}}$.

For (i), we recall that, for $t' \in \mathbb{N}$, $a \in \upsilon(t')$ if there exists some interval $\langle t_1, t_2 \rangle \in \mathcal{I}(a)$ where $t_1 \leq t' \leq t_2$. Let $\Delta^{\epsilon} = (T^{\epsilon}, \upsilon^{\epsilon})$. Since $\mathcal{I}$ is a model of $\epsilon, \mathcal{R}$, $t' \in \epsilon(a)$ implies $\langle t', t' \rangle \in \mathcal{I}(a)$. Consequently, we have by definition of $\Delta^{\mathcal{I}}$ that $a \in \upsilon^{\epsilon}(t')$ implies $a \in \upsilon(t')$, i.e., $\Delta^{\epsilon} \subseteq \Delta^{\mathcal{I}}$.

To show (ii), let $\Delta^r \in \Delta^{\mathcal{R}}$. We show that $M, t \models \Delta^r$ holds by a case distinction on the form of rule r.

- $a \leftarrow x$ SEQ y. We have, by definition,

  $$\mathcal{I}(a) = \{\langle t_1, t_4 \rangle \mid \langle t_1, t_2 \rangle \in \mathcal{I}(x), \langle t_3, t_4 \rangle \in \mathcal{I}(y),\ t_2 < t_3\}.$$

  Hence, the ETALIS semantics assigns to atom a the set of intervals $\langle t_1, t_4 \rangle$, where x holds in the interval from $\langle t_1, t_2 \rangle$, and y holds in a later interval $\langle t_3, t_4 \rangle$. Since $\mathcal{I}$ is separable, we know that for x, there is no interval $\langle t, t' \rangle$ in $\mathcal{I}(x)$, such that $[t_1 - 1, t_2 + 1] \cap [t, t'] \neq \emptyset$. Thus, $\langle t_1, t_2 \rangle$ encodes a *definite* interval in which x holds, in the sense that for any $d_1, d_2 \in \mathbb{N}$ where $d_1 + d_2 > 0$,

  - $\langle t_1 - d_1, t_2 + d_2 \rangle \notin \mathcal{I}(a)$, and
  - $\langle t_1 + d_1, t_2 - d_2 \rangle \notin \mathcal{I}(a)$.

  That is, under our assumptions, $\langle t_1, t_2 \rangle \in \mathcal{I}(x)$ means there is no greater or smaller $\langle \ell, u \rangle \in \mathcal{I}(x)$ with a common time point $n \in [\min\{t_1, \ell\}, \max\{t_2, u\}]$. Similarly, $\langle t_3, t_4 \rangle$ is a definite interval for y. Consequently, we can encode each tuple $\langle \ell, u \rangle \in \mathcal{I}(a)$ for an atom $a \in \mathcal{A}$ by associating a with each time point in $[\ell, u]$, i.e., $a \in \upsilon(t')$ for all $t' \in [\ell, u]$. This is established by the mapping from $\mathcal{I}$ to $\Delta^{\mathcal{I}}$. For the rule, we defined the translation

  $$\Delta^r = [\![t_1, t_4]\!] a \leftarrow \langle\!\langle t_1, t_2 \rangle\!\rangle x, \langle\!\langle t_3, t_4 \rangle\!\rangle y, t_2 < t_3.$$

  This LARS rule says: if $[t_1, t_2]$ is a maximal interval in which x holds at every time point, and $[t_3, t_4]$ is a maximal later interval where y holds at every time point, then a must hold at every time point in the interval $[t_1, t_4]$.

The translation $\Delta^{\mathcal{I}}$ for $\langle t_1, t_2\rangle \in \mathcal{I}(x)$ will define $x \in \upsilon(t')$ for every $t' \in [t_1, t_2]$. Since no greater interval $\langle t_1 - d_1, t_2 + d_2\rangle$ is in $\mathcal{I}(x)$, the time points $t_1 - 1$ and $t_2 + 1$ are not labelled with x, i.e., $x \notin \upsilon(t')$ for $t' \in \{t_1 - 1, t_2 + 1\}$. Consequently, $\langle\!\langle t_1, t_2\rangle\!\rangle x$ holds, and likewise, $\langle\!\langle t_3, t_4\rangle\!\rangle y$. Since the rule body holds, $[\![t_1, t_4]\!]a$ must hold. This is the case, since $\langle t_1, t_4\rangle \in \mathcal{I}$, which is encoded as $a \in \upsilon(t')$ for all $t' \in [t_1, t_4]$. Consequently, $M, t \models \Delta^r$.

Note that by separability of $\mathcal{I}$, $[t_1, t_4]$ is also a maximal interval in which a holds, and thus also in $\Delta^{\mathcal{I}}$. Consequently, any translation $\Delta^{r'}$ of a rule r' with a in the body correctly captures the intervals assigned to a.

- By the same argumentation, we get the result for the rule patterns $a \leftarrow (x).n$ and $a \leftarrow x$ BIN y, where

  $$\text{BIN} \in \{\ \text{AND},\ \text{PAR},\ \text{OR},\ \text{EQUALS},\ \text{MEETS},\ \text{DURING},\ \text{STARTS},\ \text{FINISHES}\ \}.$$

  Note that we could alternatively use the single, simple rule $a \leftarrow x \vee y$ for OR and $a \leftarrow x, y$ for EQUALS. We presented the other definitions for the sake of uniformity.

- $a \leftarrow n$, where $n \in \mathbb{N}$. LARS distinguishes atoms and time points. However, we can encode each time point $n \in \mathbb{N}$ by an auxiliary (fresh) atom μ^n and always add to the interpretation $\mu^t \in \upsilon(n)$ if $t = n$. Since $\mathcal{I}$ is a model of r, $\langle n, n\rangle \in \mathcal{I}(a)$, and by the translation, $a \in \upsilon(n)$. Consider the translated rule $\Delta^r = [\![t, t]\!]a \leftarrow \langle\!\langle t, t\rangle\!\rangle \mu^n$. The body $\langle\!\langle t, t\rangle\!\rangle \mu^n$ will hold exactly for $t = n$, thus a must hold everywhere in the interval $[t, t]$, i.e., at time point n, which is the case, since $a \in \upsilon(n)$. Consequently, Δ^r is satisfied. Note that the specific translation is again for uniformity. Alternatively, we could use the more direct translation $@_t a \leftarrow @_t \mu^n$.

- $a \leftarrow x$ WHERE b. Similarly like for SEQ, the translated rule Δ^r will assign a to the interval $[t_1, t_2]$, if $[t_1, t_2]$ is a maximal interval in which x holds. To account for the WHERE condition, Δ^r will be included in $\Delta^{\mathcal{R}}$ only if $b = \mathit{true}$. □

Lemma 23 ($\mathcal{I}$ sep., $\Delta^{\mathcal{I}}$ model $\Rightarrow$ $\mathcal{I}$ model) *Let $\epsilon, \mathcal{R}$ and $\mathcal{I}$ be separable, $\mathcal{R} \in \mathbf{R}$, and let $M = \langle \Delta^{\mathcal{I}}, W, B\rangle$, where $\Delta^{\mathcal{I}} = (T, \upsilon)$, and $t \in T$. If M is a model of $\Delta^{\mathcal{R}}$ for Δ^{ϵ} at t, then $\mathcal{I}$ is a model of $\epsilon, \mathcal{R}$.*

Proof. Indirectly, we show that $\Delta^{\mathcal{I}}$ is not a model (of $\Delta^{\mathcal{R}}$ for Δ^{ϵ} at t), assuming that $\mathcal{I}$ is not a model of $\epsilon, \mathcal{R}$, which implies that one of the following conditions is violated:

(i) $\langle t', t'\rangle \in \mathcal{I}(a)$ for all $t' \in \epsilon(a)$, and

(ii) $\mathcal{I}(pt) \subseteq \mathcal{I}(a)$ for every rule $a \leftarrow pt \in \mathcal{R}$.

Let us first assume a violation of (i), i.e., $\langle t', t'\rangle \notin \mathcal{I}(a)$ for some $t' \in \epsilon(a)$. Event atoms do not appear as rule heads. Consequently, our definitions imply that $\Delta^{\epsilon} \not\subseteq \Delta^{\mathcal{I}}$, i.e., $\Delta^{\mathcal{I}}$

is not an interpretation stream for Δ^ϵ, hence structure M is not an interpretation for stream Δ^ϵ, and thus not a model.

For (ii) we assume that $\mathcal{I}(pt) \not\subseteq \mathcal{I}(a)$ for some rule $a \leftarrow pt$ in $\mathcal{R}$. By an analogous case distinction on the specific rule form $\Delta^r \in \Delta^\mathcal{R}$ as in Lemma 22 we find that this implies $M, t \not\models \Delta^r$: the essential property is again that the intervals matching the LARS rule bodies correspond to the ETALIS patterns; they can be reflected in both languages due to separability of $\mathcal{I}$ (Lemma 20). □

With the above lemmas in place we can formalize the correspondence of considered ETALIS programs with LARS, given separability.

Proof of Theorem 18

We restate the theorem as follows. Let $\epsilon, \mathcal{R}$ be separable and $\mathcal{R} \in \mathbf{R}$. Then, $\mathcal{I}$ is a minimal model of $\epsilon, \mathcal{R}$ iff $\Delta^\mathcal{I} = (T, \upsilon)$ is an answer stream of $\Delta^\mathcal{R}$ for Δ^ϵ at t for all $t \in T$.

Proof. We first observe that, based on our translation $\Delta^\mathcal{R}$ (cf. Table A.1) and Lemma 14, it suffices to consider an arbitrary time point $t \in T$ for the evaluation of $\Delta^\mathcal{R}$.

($\Rightarrow$) Let $\mathcal{I}$ be the separable, minimal model of $\epsilon, \mathcal{R}$, $\Delta^\mathcal{I} = (T, \upsilon)$ and $t \in T$. We show that $\Delta^\mathcal{I}$ is an answer stream of $\Delta^\mathcal{R}$ for Δ^ϵ at t. By Lemma 22 we have that $\Delta^\mathcal{I}$ is a model of $\Delta^\mathcal{R}$ for Δ^ϵ at t. By Lemma 21, we obtain that it is also a model of the reduct R (of $\Delta^\mathcal{R}$ for Δ^ϵ at t) and it remains to show that $\Delta^\mathcal{I}$ is a *minimal* model of R.

For the sake of contradiction, assume that $\Delta^\mathcal{J} \subset \Delta^\mathcal{I}$ is a smaller model of R. Then, by Lemma 21, $\Delta^\mathcal{J}$ is also a model of $\Delta^\mathcal{R}$. By Lemma 23, we get that $\mathcal{J}$ is a model of $\epsilon, \mathcal{R}$. Since $\Delta^\mathcal{J} = (T, \upsilon') \subset \Delta^\mathcal{I} = (T, \upsilon)$ there exists some $t' \in T$ and some atom $a \in \mathcal{A}$ such that $a \in \upsilon(t')$ and $a \notin \upsilon'(t')$. Let $[t_1, t_2]$ be the maximal closed interval such that $a \in \upsilon(t_i)$ for all $t_i \in [t_1, t_2]$. By construction, $\langle t_1, t_2 \rangle \in \mathcal{I}(a)$ but $\langle t_1, t_2 \rangle \notin \mathcal{J}(a)$. This contradicts minimality of $\mathcal{I}$ and we conclude that $\Delta^\mathcal{I}$ is an answer stream of $\Delta^\mathcal{R}$.

($\Leftarrow$) Let $\Delta^\mathcal{I} = (T, \upsilon)$ be an answer stream (of $\Delta^\mathcal{R}$ for Δ^ϵ at $t \in T$). We have to show that $\mathcal{I}$ is a minimal model of $\epsilon, \mathcal{R}$, which is assumed to be separable. Interpretation $\Delta^\mathcal{I}$ is a minimal model of the reduct R of $\Delta^\mathcal{R}$, and by Lemma 21, a model of $\Delta^\mathcal{R}$. Thus, by Lemma 23, we get that $\mathcal{I}$ is a model of $\epsilon, \mathcal{R}$. It remains to show that $\mathcal{I}$ is a *minimal* model of $\epsilon, \mathcal{R}$.

For the sake of contradiction, assume that there is a model $\mathcal{J}$ of $\epsilon, \mathcal{R}$ that is preferred to $\mathcal{I}$. That is to say, there exists an $n \in \mathbb{N}$ such that $\mathcal{J}|_n \subset \mathcal{I}|_n$, i.e.,

$$\forall\, a \in \mathcal{A} :\ (\mathcal{J}|_n(a) \subseteq \mathcal{I}|_n(a)) \ \wedge\ \exists\, a \in \mathcal{A} :\ (\mathcal{J}|_n(a) \subset \mathcal{I}|_n(a)).$$

Without loss of generality, let $a \in \mathcal{A}$ such that $\langle t_1, t_2 \rangle \in \mathcal{I}(a) \setminus \mathcal{J}(a)$ and $t_2 - t_1 = n$. Model $\mathcal{I}$ is separable and since $\mathcal{J}$ assigns fewer intervals, $\mathcal{J}$ is also separable. Since $\mathcal{J}$ is a model of $\epsilon, \mathcal{R}$, we have by Lemma 22 that $\Delta^\mathcal{J} = (T, \upsilon^\mathcal{J})$ is a model of $\Delta^\mathcal{R}$ for Δ^ϵ at t. We observe that model $\Delta^\mathcal{J}$ does not assign atom a to any of the time points in $[t_1, t_2]$, i.e., $a \notin \upsilon^\mathcal{J}(t')$ for all $t' \in [t_1, t_2]$. We obtain $\Delta^\mathcal{J} \subset \Delta^\mathcal{I}$ which contradicts the minimality of model $\Delta^\mathcal{I}$ of $\Delta^\mathcal{R}$ for Δ^ϵ and we conclude that $\mathcal{I}$ is a minimal model for $\epsilon, \mathcal{R}$. □

Given Theorem 18 and Lemma 19 we immediately obtain the pointwise correspondence, as stated in Corollary 4 (page 92).

Discussion

We gave an intuitive translation from positive ETALIS rule bases $\mathcal{R}$ to LARS programs $\Delta^{\mathcal{R}}$ that produce the same results for any event stream ϵ if $\mathcal{R}, \epsilon$ is separable; i.e., the condition that for all atom a the assigned intervals $\mathcal{I}(a)$ do not overlap or meet in the minimal (ETALIS) model $\mathcal{I}$. We now comment on the restrictions to positive programs and non-overlapping interpretations, respectively.

The NOT-pattern. ETALIS employs negation by means of expression of the form $\mathrm{NOT}(p_1).[p_2, p_3]$, where all p_i are patterns [AFR+10]. Intuitively, the NOT-pattern selects every interval p_2 SEQ p_3 where p_1 does not occur in between. There is no general mechanism to condition the derivation of new intervals due to the absence of another, as one might expect. The advantage of this limited form of negation is that it enables the monotonic growth of derived intervals over time: the pattern p_1 that may prevent the derivation of p_2 SEQ p_3 necessarily stems from the past. Using a straightforward translation of the NOT-pattern in LARS would yield multiple models in general. A tailored encoding that maps the semantics of the NOT-pattern is possible but beyond the interest of this study. The focus here is on the possibilities to express intervals in LARS naturally, beyond its capabilities inherited from ASP. This leads to the next point.

Overlapping intervals. The presented translation from ETALIS to LARS made use of intuitive interval windows $\boxplus^{w_{[\ell,u]}}$ that always select the (maximal) substream of the specified timeline $[\ell, u]$. This is a natural way to restrict the evaluation of formulas to fixed temporal intervals but comes with the limitation that overlapping (or adjacent) intervals for the same atoms cannot be represented.

Another way to represent intervals would be by directly encoding their limits within atoms. For instance, if an atom a holds during the interval $[7, 19]$, this can be reflected by a fresh atom $a'(7, 19)$. Following up on Example 37, we then express the ETALIS rule $r = z \leftarrow x$ SEQ y, where x, y, and z are propositions, as follows.

$$z'(T_1, T_4) \leftarrow x'(T_1, T_2),\ y'(T_3, T_4),\ T_2 < T_3$$

Clearly, intervals as viewed in ETALIS (i.e., pairs of integers) can be encoded this way, which also permits the expression of overlaps. For instance, the above rule can fire both for $x'(1, 4)$ and $x'(3, 5)$. Given also $y'(6, 8)$, we obtain $z'(1, 8)$ and $z'(3, 8)$. To bootstrap the process, one then simply needs a rule of the form $a'(T, T) \leftarrow @_T\, a$ for every atom a occurring in the original program. We thus obtain in essence an encoding in Datalog, respectively ASP if also the NOT-pattern is used; and ASP (and thus Datalog) is subsumed by LARS.

Finally, we note that more expressive complex event processing, in particular with nonmonotonic semantics and multiple models, would result by equipping an extended version of LARS with an interval-based evaluation function $\upsilon : T \times T \to 2^{\mathcal{A}}$.

APPENDIX B

Ticker

B.1 Detailled Evaluation Results

We present here the detailled results for the empirical evaluation of Ticker's reasoning modes, underlying the charts in Section 7.4.

Each table shows performance metrics of a single benchmark program, where some parameters are fixed as indicated in the header of the table, and some vary. The latter are indicated in the left-most column, where we indicate two dimensions of instantiated parameters. For instance, Table B.1 iterates the window atom forms $t@$, $t\Diamond$, $t\Box$, $\#@$, $\#\Diamond$ and $\#\Box$, and within each block, each row holds the results for a specific insert probability p. All tables show the results for the incremental reasoning mode on the left side, and those for Clingo mode on the right. Each reasoning mode's performance measure is detailled in the columns *total*, *init*, *tp*, and *tp/s*, all of which give the average of 5 runs. The entries are specified as follows.

- *total*: Total runtime in seconds for the given parameterization. It is essentially the sum of *init* and 2000 times *tp*,[1] or more precisely, *init* plus 2000 divided by the entry in *tp/s*. (Typically, *total* once adds to this sum a small overhead to finish processing the entire chain of calls.)
- *init*: Initialization time (in seconds) before the processing of the first time point starts. This includes the time for pre-grounding in the incremental mode.
- *tp*: Average time (in seconds) needed to process a single timepoint; we use ε for entries below 0.001s.
- *tp/s*: Processed time points per second. This is the main performance metric and used in the charts in Section 7.4. For each value v here, the previous column *tp* dually holds $1/v$.

[1] Recall that all instances run over 2000 time points.

Basic: $n = 1$, $k = 50$

		incremental				clingo			
	p	total	init	tp	tp/s	total	init	tp	tp/s
t@	0.0	0.127	0.003	ε	16260.16	15.107	0.018	0.007	132.55
	0.1	0.169	0.003	ε	12121.21	15.837	0.015	0.007	126.41
	0.5	0.259	0.004	ε	7843.14	16.366	0.016	0.008	122.33
	0.9	0.348	0.003	ε	5813.95	18.193	0.015	0.009	110.03
	1.0	0.345	0.003	ε	5865.10	18.378	0.014	0.009	108.91
t◇	0.0	0.150	0.004	ε	13698.63	13.581	0.015	0.006	147.43
	0.1	0.139	0.004	ε	14814.81	14.233	0.014	0.007	140.66
	0.5	0.181	0.003	ε	11299.44	18.880	0.017	0.009	106.03
	0.9	0.206	0.003	ε	9900.99	18.381	0.017	0.009	108.91
	1.0	0.226	0.003	ε	9009.01	18.114	0.016	0.009	110.52
t□	0.0	0.126	0.004	ε	16528.93	15.944	0.016	0.007	125.57
	0.1	0.184	0.004	ε	11173.18	15.940	0.016	0.007	125.60
	0.5	0.302	0.003	ε	6711.41	17.469	0.019	0.008	114.62
	0.9	0.453	0.004	ε	4454.34	17.204	0.016	0.008	116.37
	1.0	0.512	0.004	ε	3937.01	16.859	0.016	0.008	118.74
#@	0.0	0.072	0.004	ε	29850.75	16.341	0.015	0.008	122.50
	0.1	0.169	0.004	ε	12121.21	21.041	0.014	0.010	95.12
	0.5	0.309	0.003	ε	6557.38	20.423	0.013	0.010	97.99
	0.9	0.386	0.004	ε	5249.34	20.913	0.016	0.010	95.71
	1.0	0.390	0.003	ε	5167.96	21.425	0.013	0.010	93.41
#◇	0.0	0.088	0.004	ε	24096.39	13.983	0.016	0.006	143.19
	0.1	0.157	0.003	ε	13071.90	19.901	0.016	0.009	100.58
	0.5	0.189	0.003	ε	10752.69	21.046	0.015	0.010	95.10
	0.9	0.222	0.003	ε	9174.31	19.872	0.014	0.009	100.72
	1.0	0.252	0.003	ε	8064.52	20.145	0.016	0.010	99.36
#□	0.0	2.071	0.004	0.001	968.05	8.379	0.008	0.004	238.92
	0.1	2.269	0.004	0.001	883.00	14.279	0.008	0.007	140.14
	0.5	1.534	0.004	ε	1308.04	14.748	0.008	0.007	135.69
	0.9	1.912	0.004	ε	1048.22	15.840	0.008	0.007	126.33
	1.0	3.104	0.004	0.001	645.16	16.137	0.008	0.008	124.00

Table B.1: Program Basic ($n = 1$). Effect of varying insert probability p.

Only one evaluation did not finish; it had to be stopped manually after a single run exceeded 4 hours (see Table B.10). According entries are marked with a minus (−). This is was caused by a garbage collection overhead, where 10 GB of heap space did not suffice.

BASIC: $n = 32$, $k = 50$

		incremental				clingo			
	p	total	init	tp	tp/s	total	init	tp	tp/s
t@	0.0	5.322	0.014	0.002	376.79	13.764	0.013	0.006	145.44
	0.1	6.230	0.011	0.003	321.65	26.279	0.014	0.013	76.15
	0.5	9.665	0.012	0.004	207.19	66.742	0.015	0.033	29.97
	0.9	12.772	0.012	0.006	156.74	107.251	0.016	0.053	18.65
	1.0	13.488	0.011	0.006	148.40	117.496	0.013	0.058	17.02
t◇	0.0	3.329	0.013	0.001	603.14	13.517	0.014	0.006	148.12
	0.1	3.940	0.011	0.001	509.16	24.027	0.015	0.012	83.30
	0.5	5.902	0.011	0.002	339.56	55.221	0.013	0.027	36.23
	0.9	7.971	0.012	0.003	251.29	85.693	0.014	0.042	23.34
	1.0	8.550	0.012	0.004	234.27	92.215	0.014	0.046	21.69
t□	0.0	3.675	0.016	0.001	546.75	15.343	0.016	0.007	130.49
	0.1	5.349	0.016	0.002	375.02	21.794	0.018	0.010	91.84
	0.5	10.227	0.015	0.005	195.87	45.652	0.016	0.022	43.83
	0.9	16.288	0.018	0.008	122.93	69.706	0.015	0.034	28.70
	1.0	19.424	0.014	0.009	103.04	75.150	0.016	0.037	26.62
#@	0.0	1.774	0.012	ε	1135.72	15.556	0.013	0.007	128.68
	0.1	9.913	0.012	0.004	202.00	20.759	0.012	0.010	96.40
	0.5	34.045	0.011	0.017	58.77	22.319	0.014	0.011	89.67
	0.9	58.902	0.012	0.029	33.96	23.306	0.012	0.011	85.86
	1.0	65.023	0.010	0.032	30.76	24.641	0.012	0.012	81.21
#◇	0.0	1.831	0.014	ε	1100.72	13.726	0.013	0.006	145.85
	0.1	6.963	0.009	0.003	287.65	20.700	0.016	0.010	96.70
	0.5	24.620	0.010	0.012	81.27	23.350	0.014	0.011	85.71
	0.9	41.278	0.011	0.020	48.47	24.575	0.013	0.012	81.43
	1.0	44.257	0.009	0.022	45.20	25.642	0.017	0.012	78.05
#□	0.0	8.507	0.016	0.004	235.54	8.648	0.009	0.004	231.51
	0.1	47.794	0.015	0.023	41.86	15.022	0.010	0.007	133.24
	0.5	159.234	0.015	0.079	12.56	18.089	0.008	0.009	110.62
	0.9	292.849	0.013	0.146	6.83	20.040	0.008	0.010	99.84
	1.0	345.484	0.015	0.172	5.79	20.800	0.008	0.010	96.19

Table B.2: Program BASIC ($n = 32$). Effect of varying insert probability p.

Basic: $n = 1$, $p = 0.5$

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
t@	1	0.191	0.002	ε	10638.30	8.058	0.006	0.004	248.42
	5	0.186	0.003	ε	10989.01	8.615	0.008	0.004	232.37
	10	0.215	0.003	ε	9433.96	10.782	0.014	0.005	185.74
	50	0.248	0.003	ε	8163.27	17.026	0.018	0.008	117.59
	100	0.298	0.003	ε	6779.66	26.966	0.018	0.013	74.22
	500	0.641	0.004	ε	3139.72	136.447	0.022	0.068	14.66
t◇	1	0.172	0.004	ε	11904.76	7.892	0.006	0.003	253.65
	5	0.197	0.004	ε	10362.69	8.502	0.007	0.004	235.46
	10	0.176	0.003	ε	11560.69	9.203	0.007	0.004	217.51
	50	0.182	0.003	ε	11173.18	15.267	0.014	0.007	131.13
	100	0.186	0.003	ε	10928.96	23.733	0.016	0.011	84.33
	500	0.355	0.003	ε	5681.82	118.916	0.018	0.059	16.82
t□	1	0.195	0.003	ε	10416.67	7.708	0.006	0.003	259.67
	5	0.217	0.004	ε	9389.67	8.369	0.009	0.004	239.23
	10	0.234	0.004	ε	8695.65	9.171	0.010	0.004	218.32
	50	0.284	0.003	ε	7117.44	16.286	0.018	0.008	122.94
	100	0.362	0.004	ε	5586.59	23.061	0.017	0.011	86.79
	500	0.798	0.004	ε	2518.89	84.407	0.029	0.042	23.70
#@	1	0.199	0.003	ε	10256.41	8.148	0.007	0.004	245.67
	5	0.182	0.004	ε	11235.96	9.343	0.008	0.004	214.27
	10	0.242	0.003	ε	8403.36	10.532	0.009	0.005	190.06
	50	0.270	0.003	ε	7490.64	19.657	0.013	0.009	101.81
	100	0.333	0.002	ε	6060.61	30.696	0.015	0.015	65.19
	500	0.906	0.002	ε	2214.84	110.238	0.023	0.055	18.15
#◇	1	0.197	0.003	ε	10362.69	10.457	0.009	0.005	191.44
	5	0.193	0.004	ε	10582.01	10.233	0.010	0.005	195.64
	10	0.196	0.003	ε	10362.69	10.765	0.008	0.005	185.93
	50	0.198	0.003	ε	10309.28	19.121	0.013	0.009	104.67
	100	0.226	0.003	ε	8968.61	31.763	0.015	0.015	63.00
	500	0.478	0.003	ε	4219.41	183.837	0.023	0.091	10.88
#□	1	0.219	0.002	ε	9259.26	9.194	0.006	0.004	217.68
	5	0.716	0.003	ε	2808.99	9.671	0.007	0.004	206.95
	10	0.806	0.004	ε	2493.77	10.200	0.009	0.005	196.27
	50	1.616	0.004	ε	1241.46	14.834	0.008	0.007	134.91
	100	2.983	0.003	0.001	671.37	20.396	0.009	0.010	98.10
	500	9.354	0.004	0.004	2139.0	50.492	0.007	0.025	39.62

Table B.3: Program Basic ($n = 1$). Effect of varying window size k.

BASIC: $n = 32$, $p = 0.5$

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
t@	1	5.814	0.012	0.002	344.71	11.333	0.010	0.005	176.63
	5	6.076	0.011	0.003	329.82	15.516	0.009	0.007	128.97
	10	6.582	0.012	0.003	304.41	20.918	0.009	0.010	95.66
	50	9.552	0.010	0.004	209.62	66.409	0.015	0.033	30.12
	100	12.453	0.012	0.006	160.77	142.691	0.016	0.071	14.02
	500	37.457	0.017	0.018	53.42	1392.870	0.025	0.696	1.44
t◇	1	4.664	0.011	0.002	429.83	11.201	0.008	0.005	178.70
	5	4.560	0.011	0.002	439.75	14.431	0.009	0.007	138.68
	10	4.662	0.010	0.002	429.92	18.679	0.009	0.009	107.12
	50	5.838	0.012	0.002	343.29	55.595	0.011	0.027	35.98
	100	7.522	0.011	0.003	266.31	113.645	0.015	0.056	17.60
	500	21.149	0.018	0.010	94.65	1210.333	0.02	0.605	1.65
t□	1	6.044	0.016	0.003	331.79	10.963	0.008	0.005	182.58
	5	6.876	0.016	0.003	291.59	13.236	0.010	0.006	151.22
	10	7.207	0.014	0.003	278.05	16.652	0.011	0.008	120.19
	50	10.241	0.015	0.005	195.60	45.924	0.015	0.022	43.56
	100	14.340	0.016	0.007	139.64	74.545	0.017	0.037	26.84
	500	40.070	0.021	0.020	49.94	356.312	0.030	0.178	5.61
#@	1	36.659	0.011	0.018	54.57	11.510	0.008	0.005	173.90
	5	27.829	0.010	0.013	71.89	11.971	0.008	0.005	167.20
	10	29.887	0.010	0.014	66.94	12.898	0.010	0.006	155.18
	50	33.996	0.010	0.016	58.85	22.418	0.013	0.011	89.27
	100	36.056	0.009	0.018	55.48	33.964	0.016	0.016	58.91
	500	48.470	0.015	0.024	41.28	119.903	0.025	0.059	16.68
#◇	1	18.734	0.012	0.009	106.83	11.296	0.009	0.005	177.21
	5	20.215	0.011	0.010	99.00	11.908	0.007	0.005	168.07
	10	21.525	0.011	0.010	92.96	12.841	0.009	0.006	155.87
	50	24.447	0.010	0.012	81.84	23.351	0.014	0.011	85.70
	100	26.183	0.010	0.013	76.42	37.079	0.016	0.018	53.96
	500	40.743	0.013	0.020	49.10	224.333	0.022	0.112	8.92
#□	1	4.986	0.012	0.002	402.09	13.061	0.008	0.006	153.22
	5	82.605	0.015	0.041	24.22	13.200	0.010	0.006	151.64
	10	89.567	0.015	0.044	22.33	13.359	0.009	0.006	149.81
	50	161.691	0.015	0.080	12.37	18.497	0.009	0.009	108.18
	100	255.304	0.018	0.127	7.83	23.845	0.008	0.011	83.91
	500	1059.166	0.022	0.529	1.89	64.331	0.012	0.032	31.10

Table B.4: Program BASIC ($n = 32$). Effect of varying window size k.

Basic: $n = 32$, $p = 0.01$

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
t◇	1	2.963	0.014	0.001	678.20	8.784	0.008	0.004	227.92
	5	3.102	0.012	0.001	647.46	9.671	0.010	0.004	207.02
	10	3.220	0.013	0.001	623.64	10.160	0.011	0.005	197.06
	50	3.653	0.012	0.001	549.45	16.387	0.016	0.008	122.17
	100	3.869	0.014	0.001	518.94	23.534	0.018	0.011	85.05
	500	4.478	0.015	0.002	448.13	98.607	0.019	0.049	20.29
#◇	1	2.444	0.014	0.001	823.38	9.300	0.011	0.004	215.33
	5	2.635	0.015	0.001	763.36	10.492	0.008	0.005	190.79
	10	2.756	0.015	0.001	729.93	11.636	0.011	0.005	172.04
	50	3.117	0.014	0.001	644.54	20.687	0.014	0.010	96.74
	100	3.252	0.014	0.001	617.67	33.027	0.017	0.016	60.59
	500	3.663	0.014	0.001	548.10	157.552	0.019	0.078	12.70

Table B.5: Program Basic ($n = 32$). Effect of varying window size k. (Snapshot semantics, little data.)

Basic: $k = 50$, $p = 0.5$

		incremental				clingo			
	n	total	init	tp	tp/s	total	init	tp	tp/s
t◇	1	0.214	0.004	ε	9523.81	15.076	0.014	0.007	132.78
	2	0.318	0.003	ε	6349.21	16.672	0.014	0.008	120.07
	4	0.640	0.004	ε	3149.61	20.414	0.013	0.010	98.03
	8	1.395	0.005	ε	1438.85	30.311	0.016	0.015	66.02
	16	2.743	0.004	0.001	730.46	37.865	0.021	0.018	52.85
	32	6.248	0.007	0.003	320.46	58.106	0.014	0.029	34.43
	64	14.278	0.008	0.007	140.15	106.983	0.016	0.053	18.70
#◇	1	0.181	0.003	ε	11235.96	19.899	0.015	0.009	100.59
	2	0.403	0.003	ε	5000.00	19.549	0.014	0.009	102.38
	4	1.041	0.004	ε	1930.50	19.370	0.014	0.009	103.33
	8	2.188	0.004	0.001	916.17	20.156	0.016	0.010	99.30
	16	6.951	0.005	0.003	287.94	20.853	0.015	0.010	95.98
	32	24.390	0.005	0.012	82.02	24.242	0.016	0.012	82.56
	64	104.043	0.012	0.052	19.23	27.497	0.016	0.013	72.78

Table B.6: Program Basic. Effect of varying program size n.

REACH: $n = 8$, $k = 50$

		incremental				clingo			
	p	total	init	tp	tp/s	total	init	tp	tp/s
t@	0.0	11.298	0.013	0.005	177.23	14.978	0.015	0.007	133.66
	0.1	11.750	0.013	0.005	170.40	20.160	0.016	0.010	99.29
	0.5	12.815	0.013	0.006	156.23	31.841	0.016	0.015	62.85
	0.9	14.134	0.014	0.007	141.64	43.090	0.015	0.021	46.43
	1.0	14.069	0.014	0.007	142.31	45.073	0.014	0.022	44.39
t◇	0.0	7.025	0.011	0.003	285.14	14.469	0.015	0.007	138.37
	0.1	8.342	0.014	0.004	240.15	17.993	0.014	0.008	111.24
	0.5	8.987	0.013	0.004	222.89	26.645	0.013	0.013	75.10
	0.9	9.321	0.012	0.004	214.85	35.660	0.015	0.017	56.11
	1.0	9.166	0.011	0.004	218.48	36.346	0.012	0.018	55.05
t□	0.0	8.431	0.017	0.004	237.73	15.973	0.015	0.007	125.33
	0.1	9.952	0.017	0.004	201.31	17.695	0.017	0.008	113.14
	0.5	11.986	0.015	0.005	167.08	24.762	0.017	0.012	80.83
	0.9	15.456	0.017	0.007	129.55	31.521	0.017	0.015	63.48
	1.0	17.761	0.015	0.008	112.70	33.217	0.018	0.016	60.24
#@	0.0	3.852	0.011	0.001	520.83	16.585	0.012	0.008	120.69
	0.1	9.838	0.011	0.004	203.54	22.702	0.016	0.011	88.16
	0.5	25.368	0.012	0.012	78.88	23.543	0.015	0.011	85.01
	0.9	34.795	0.013	0.017	57.50	23.685	0.016	0.011	84.50
	1.0	36.749	0.011	0.018	54.44	23.935	0.014	0.011	83.61
#◇	0.0	3.869	0.013	0.001	518.67	14.841	0.015	0.007	134.90
	0.1	7.845	0.011	0.003	255.30	22.008	0.015	0.010	90.94
	0.5	16.988	0.011	0.008	117.81	22.244	0.013	0.011	89.96
	0.9	26.738	0.011	0.013	74.83	22.492	0.014	0.011	88.98
	1.0	29.134	0.013	0.014	68.68	22.463	0.013	0.011	89.09
#□	0.0	19.269	0.020	0.009	103.90	8.657	0.007	0.004	231.24
	0.1	72.536	0.018	0.036	27.58	16.03	0.011	0.008	124.85
	0.5	133.368	0.018	0.066	15.00	16.315	0.010	0.008	122.66
	0.9	206.648	0.033	0.103	9.68	16.977	0.009	0.008	117.88
	1.0	243.625	0.033	0.121	8.21	17.517	0.008	0.008	114.23

Table B.7: Program REACH ($n = 8$). Effect of varying insert probability p.

Reach: $n = 8$, $p = 0.5$

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
t@	1	9.468	0.012	0.004	211.51	9.322	0.007	0.004	214.73
	5	10.462	0.013	0.005	191.42	11.253	0.009	0.005	177.87
	10	10.582	0.013	0.005	189.23	13.118	0.011	0.006	152.59
	50	12.734	0.013	0.006	157.22	30.804	0.015	0.015	64.96
	100	15.484	0.014	0.007	129.29	56.243	0.015	0.028	35.57
	500	26.107	0.011	0.013	76.64	437.628	0.023	0.218	4.57
t◇	1	7.211	0.011	0.003	277.78	9.290	0.007	0.004	215.47
	5	7.504	0.012	0.003	266.99	10.744	0.007	0.005	186.27
	10	7.694	0.012	0.003	260.35	12.566	0.008	0.006	159.26
	50	8.506	0.010	0.004	235.40	26.224	0.015	0.013	76.31
	100	9.858	0.013	0.004	203.15	48.090	0.018	0.024	41.61
	500	13.970	0.013	0.006	143.31	404.440	0.020	0.202	4.95
t□	1	9.319	0.014	0.004	214.96	8.850	0.007	0.004	226.19
	5	10.133	0.013	0.005	197.63	10.300	0.009	0.005	194.36
	10	10.675	0.012	0.005	187.56	11.735	0.010	0.005	170.58
	50	12.425	0.015	0.006	161.17	23.844	0.015	0.011	83.93
	100	14.030	0.013	0.007	142.69	38.411	0.017	0.019	52.09
	500	22.344	0.013	0.011	89.57	149.124	0.029	0.074	13.41
#@	1	21.451	0.009	0.010	93.28	9.314	0.008	0.004	214.92
	5	20.732	0.009	0.010	96.52	10.432	0.009	0.005	191.90
	10	21.578	0.012	0.010	92.74	12.055	0.009	0.006	166.03
	50	24.155	0.012	0.012	82.84	22.154	0.014	0.011	90.33
	100	26.310	0.013	0.013	76.05	34.522	0.017	0.017	57.96
	500	35.635	0.014	0.017	56.15	130.945	0.022	0.065	15.28
#◇	1	14.746	0.010	0.007	135.72	9.239	0.007	0.004	216.64
	5	15.689	0.011	0.007	127.57	10.260	0.008	0.005	195.10
	10	15.272	0.010	0.007	131.05	11.648	0.009	0.005	171.84
	50	16.883	0.011	0.008	118.55	21.548	0.015	0.010	92.88
	100	17.966	0.010	0.008	111.39	35.869	0.014	0.017	55.78
	500	25.274	0.012	0.012	79.17	221.984	0.022	0.110	9.01
#□	1	5.899	0.011	0.002	339.67	9.909	0.008	0.004	202.02
	5	52.163	0.020	0.026	38.36	10.393	0.008	0.005	192.60
	10	62.335	0.020	0.031	32.10	10.940	0.008	0.005	182.95
	50	123.081	0.019	0.061	16.25	15.630	0.008	0.007	128.03
	100	212.413	0.017	0.106	9.42	21.427	0.009	0.010	93.38
	500	716.654	0.017	0.358	2.79	63.169	0.008	0.031	31.67

Table B.8: Program Reach ($n = 8$). Effect of varying window size k.

REACH: $n = 32$, $p = 0.5$

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
t◇	1	211.677	0.244	0.105	9.46	13.007	0.010	0.006	153.88
	5	250.470	0.271	0.125	7.99	23.977	0.012	0.011	83.46
	10	233.356	0.237	0.116	8.58	30.005	0.012	0.014	66.68
	50	223.631	0.259	0.111	8.95	67.786	0.020	0.033	29.51
	100	259.301	0.239	0.129	7.72	129.050	0.025	0.064	15.50
	500	491.682	0.296	0.245	4.07	1251.249	0.033	0.625	1.60
#◇	1	1071.230	0.237	0.535	1.87	12.155	0.010	0.006	164.68
	5	1129.816	0.336	0.564	1.77	12.935	0.011	0.006	154.75
	10	1212.501	0.289	0.606	1.65	13.858	0.013	0.006	144.46
	50	1432.245	0.271	0.715	1.40	27.440	0.021	0.013	72.94
	100	1706.193	0.303	0.852	1.17	46.774	0.025	0.023	42.78
	500	3857.366	0.342	1.928	0.52	248.795	0.033	0.124	8.04

Table B.9: Program REACH ($n = 32$). Effect of varying window size k.

REACH: $k = 50$, $p = 0.5$

		incremental				clingo			
	n	total	init	tp	tp/s	total	init	tp	tp/s
t◇	1	0.190	0.004	ε	10810.81	17.212	0.016	0.008	116.31
	2	0.537	0.005	ε	3759.40	18.579	0.013	0.009	107.73
	4	2.035	0.008	0.001	987.17	21.674	0.013	0.010	92.34
	8	8.847	0.012	0.004	226.37	27.339	0.015	0.013	73.20
	16	41.184	0.034	0.020	48.60	41.122	0.017	0.020	48.66
	32	242.725	0.261	0.121	8.25	68.427	0.017	0.034	29.24
	64	1807.202	1.485	0.902	1.11	159.319	0.016	0.079	12.55
#◇	1	0.202	0.004	ε	10101.01	21.736	0.019	0.010	92.09
	2	0.602	0.005	ε	3355.70	21.514	0.016	0.010	93.04
	4	2.729	0.006	0.001	734.48	21.603	0.015	0.010	92.64
	8	18.021	0.011	0.009	111.05	22.540	0.017	0.011	88.80
	16	155.265	0.045	0.077	12.88	24.605	0.016	0.012	81.34
	32	1567.877	0.323	0.783	1.28	29.052	0.018	0.014	68.89
	64	-	-	-	-	31.323	0.023	0.015	63.90

Table B.10: Program REACH. Effect of varying program size n.

Strategy

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
$n = 3$	0	1.707	0.014	ε	1181.33	9.544	0.012	0.004	209.82
	1	1.942	0.013	ε	1037.34	10.095	0.015	0.005	198.41
	5	2.059	0.015	0.001	978.47	12.195	0.016	0.006	164.22
	10	2.185	0.014	0.001	921.23	14.522	0.019	0.007	137.91
	50	2.635	0.014	0.001	763.36	34.270	0.026	0.017	58.40
	100	3.223	0.012	0.001	622.86	59.838	0.032	0.029	33.44
	500	8.653	0.014	0.004	231.54	357.822	0.081	0.178	5.59
$n = 9$	0	3.069	0.018	0.001	655.52	9.833	0.011	0.004	203.62
	1	3.397	0.019	0.001	592.24	10.396	0.015	0.005	192.68
	5	3.718	0.019	0.001	540.69	12.381	0.018	0.006	161.77
	10	3.873	0.017	0.001	518.81	15.102	0.018	0.007	132.60
	50	4.557	0.020	0.002	440.92	34.619	0.025	0.017	57.81
	100	5.256	0.018	0.002	381.83	60.228	0.029	0.030	33.22
	500	11.571	0.017	0.005	173.12	358.06	0.068	0.178	5.59
$n = 30$	0	8.781	0.083	0.004	229.94	11.282	0.012	0.005	177.48
	1	9.658	0.095	0.004	209.14	11.843	0.015	0.005	169.09
	5	10.046	0.099	0.004	201.09	13.934	0.014	0.006	143.69
	10	10.512	0.088	0.005	191.86	16.424	0.019	0.008	121.91
	50	12.551	0.094	0.006	160.57	35.760	0.025	0.017	55.97
	100	13.892	0.095	0.006	144.96	61.594	0.034	0.030	32.49
	500	22.335	0.092	0.011	89.92	359.552	0.071	0.179	5.56
$n = 90$	0	31.659	1.624	0.015	66.59	16.592	0.013	0.008	120.64
	1	32.421	1.484	0.015	64.65	17.121	0.014	0.008	116.91
	5	34.058	1.503	0.016	61.44	18.960	0.017	0.009	105.58
	10	36.113	1.516	0.017	57.81	21.695	0.017	0.010	92.26
	50	39.904	1.605	0.019	52.22	40.857	0.025	0.020	48.98
	100	44.989	1.575	0.021	46.07	66.629	0.037	0.033	30.03
	500	66.906	1.579	0.032	30.62	363.250	0.083	0.181	5.51

Table B.11: Program Strategy. Effect of program size n and window size k.

CONTENT: $n = 10$

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
$i = 4$	0	20.294	0.226	0.010	99.67	11.805	0.010	0.005	169.56
	1	20.690	0.199	0.010	97.60	12.994	0.012	0.006	154.06
	5	20.697	0.213	0.010	97.64	17.342	0.014	0.008	115.42
	10	21.418	0.217	0.010	94.34	23.098	0.016	0.011	86.65
	50	23.316	0.209	0.011	86.55	67.700	0.018	0.033	29.55
	100	26.512	0.223	0.013	76.08	138.462	0.022	0.069	14.45
	500	39.299	0.256	0.019	51.23	1258.050	0.046	0.629	1.59
$i = 8$	0	32.285	0.356	0.015	62.64	11.983	0.010	0.005	167.06
	1	33.159	0.357	0.016	60.97	12.974	0.012	0.006	154.30
	5	34.518	0.383	0.017	58.59	17.384	0.013	0.008	115.13
	10	34.679	0.380	0.017	58.31	23.113	0.016	0.011	86.60
	50	38.842	0.379	0.019	52.00	68.458	0.020	0.034	29.22
	100	42.977	0.363	0.021	46.93	139.683	0.022	0.069	14.32
	500	61.890	0.395	0.030	32.52	1265.964	0.052	0.632	1.58
$i = 16$	0	60.110	0.707	0.029	33.67	12.245	0.010	0.006	163.47
	1	63.041	0.699	0.031	32.08	13.337	0.012	0.006	150.09
	5	65.711	0.730	0.032	30.78	17.944	0.014	0.008	111.54
	10	65.995	0.747	0.032	30.65	23.341	0.015	0.011	85.74
	50	76.718	0.728	0.037	26.32	70.333	0.020	0.035	28.44
	100	84.346	0.721	0.041	23.92	141.280	0.022	0.070	14.16
	500	123.229	0.704	0.061	16.32	1264.973	0.060	0.632	1.58
$i = 32$	0	126.470	1.430	0.062	15.99	12.467	0.011	0.006	160.57
	1	124.617	1.393	0.061	16.23	13.630	0.013	0.006	146.89
	5	131.001	1.380	0.064	15.43	18.200	0.015	0.009	109.99
	10	135.500	1.414	0.067	14.92	24.098	0.016	0.012	83.05
	50	161.189	1.457	0.079	12.52	70.437	0.021	0.035	28.40
	100	188.337	1.405	0.093	10.70	142.674	0.022	0.071	14.02
	500	291.593	1.485	0.145	6.89	1342.446	0.073	0.671	1.49
$i = 64$	0	277.638	2.922	0.137	7.28	13.429	0.012	0.006	149.08
	1	282.700	2.967	0.139	7.15	14.599	0.013	0.007	137.12
	5	289.352	2.950	0.143	6.98	19.074	0.016	0.009	104.94
	10	304.602	2.988	0.150	6.63	24.946	0.015	0.012	80.22
	50	380.437	2.951	0.188	5.30	72.416	0.019	0.036	27.63
	100	430.986	3.135	0.213	4.67	147.587	0.023	0.073	13.55
	500	840.437	2.981	0.418	2.39	1340.007	0.074	0.669	1.49

Table B.12: Program CONTENT ($n = 10$). Effect of number of items i and window size k.

Content: $n = 20$

		incremental				clingo			
	k	total	init	tp	tp/s	total	init	tp	tp/s
$i = 4$	0	47.582	1.372	0.023	43.28	13.693	0.014	0.006	146.21
	1	48.879	1.394	0.023	42.12	15.572	0.018	0.007	128.59
	5	50.256	1.405	0.024	40.94	22.055	0.020	0.011	90.76
	10	51.508	1.412	0.025	39.92	30.088	0.020	0.015	66.52
	50	56.157	1.310	0.027	36.47	98.679	0.031	0.049	20.27
	100	63.276	1.366	0.030	32.31	217.596	0.026	0.108	9.19
	500	102.524	1.588	0.050	19.81	2115.053	0.090	1.057	0.95
$i = 8$	0	81.415	2.680	0.039	25.40	13.680	0.014	0.006	146.35
	1	80.367	2.631	0.038	25.73	15.342	0.015	0.007	130.50
	5	86.638	2.673	0.041	23.82	25.386	0.024	0.012	78.86
	10	86.352	2.739	0.041	23.92	31.359	0.021	0.015	63.82
	50	98.936	2.693	0.048	20.78	99.370	0.027	0.049	20.13
	100	113.889	2.859	0.055	18.01	221.187	0.054	0.110	9.04
	500	174.454	2.886	0.085	11.66	2115.357	0.082	1.057	0.95
$i = 16$	0	163.155	5.499	0.078	12.69	13.986	0.014	0.006	143.15
	1	161.433	5.550	0.077	12.83	15.450	0.014	0.007	129.57
	5	168.092	5.797	0.081	12.32	23.240	0.021	0.011	86.14
	10	170.579	5.613	0.082	12.12	30.581	0.020	0.015	65.45
	50	198.099	5.624	0.096	10.39	100.103	0.028	0.050	19.99
	100	223.663	5.645	0.109	9.17	223.530	0.033	0.111	8.95
	500	371.251	7.047	0.182	5.49	2122.224	0.064	1.061	0.94
$i = 32$	0	355.356	11.679	0.171	5.82	14.498	0.016	0.007	138.10
	1	358.126	11.694	0.173	5.77	16.155	0.017	0.008	123.93
	5	364.459	11.680	0.176	5.67	24.417	0.020	0.012	81.98
	10	378.060	11.761	0.183	5.46	30.896	0.023	0.015	64.78
	50	461.341	12.183	0.224	4.45	106.255	0.028	0.053	18.83
	100	521.474	11.965	0.254	3.93	221.297	0.033	0.110	9.04
	500	1020.162	12.421	0.503	1.98	2124.759	0.079	1.062	0.94
$i = 64$	0	892.615	25.514	0.433	2.31	14.911	0.021	0.007	134.32
	1	894.315	26.183	0.434	2.30	16.702	0.014	0.008	119.85
	5	929.521	27.040	0.451	2.22	23.606	0.021	0.011	84.80
	10	974.860	26.961	0.473	2.11	31.493	0.022	0.015	63.55
	50	1215.322	26.041	0.594	1.68	104.139	0.030	0.052	19.21
	100	1452.145	25.840	0.713	1.40	219.451	0.034	0.109	9.12
	500	3231.036	25.693	1.602	0.62	2124.738	0.084	1.062	0.94

Table B.13: Program Content ($n = 20$). Effect of number of items i and window size k.

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 277–289, 2005.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Database Programming Languages - 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[ACÇ+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[ACD+17] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP System DLV2. In *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, volume 10377 of *Lecture Notes in Computer Science*, pages 215–221. Springer, 2017.

[ACPV08] Felicidad Aguado, Pedro Cabalar, Gilberto Pérez, and Concepción Vidal. Strongly equivalent temporal logic programs. In Steffen Hölldobler, Carsten Lutz, and Heinrich Wansing, editors, *Logics in Artificial Intelligence, 11th European Conference, JELIA 2008, Dresden, Germany, September 28 - October 1, 2008. Proceedings*, volume 5293 of *Lecture Notes in Computer Science*, pages 8–20. Springer, 2008.

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In Wang [Wan08], pages 147–160.

[ADT+18] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 601–613. ACM, 2018.

[AFH96] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The Benefits of Relaxing Punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[AFR+10] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In *Web Reasoning and Rule Systems - 4th International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2010.

[AFRS11] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28-April 1, 2011*, pages 635–644. ACM, 2011.

[Agh90] Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press Series in Artificial Intelligence. MIT Press, 1990.

[AH89] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 164–169. IEEE Computer Society, 1989.

[AH93] Rajeev Alur and Thomas A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*, volume 8. Addison-Wesley, Reading, 1995.

[AKK+17] Alessandro Artale, Roman Kontchakov, Alisa Kovtunova, Vladislav Ryzhikov, Frank Wolter, and Michael Zakharyaschev. Ontology-Mediated Query Answering over Temporal Data: A Survey (Invited Talk). In Sven Schewe, Thomas Schneider, and Jef Wijsen, editors, *24th International Symposium on Temporal Representation and Reasoning, TIME 2017, October 16-18, 2017, Mons, Belgium*, volume 90 of *LIPIcs*, pages 1:1–1:37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[All83] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[ARFS12] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012.

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[BBC+09] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: SPARQL for continuous querying. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 1061–1062. ACM, 2009.

[BBC+10a] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: a Continuous Query Language for RDF Data Streams. *International Journal of Semantic Computing*, 4(1):3–25, 2010.

[BBC+10b] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental Reasoning on Streams and Rich Background Knowledge. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I*, volume 6088 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.

[BBCG10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for C-SPARQL queries. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 441–452. ACM, 2010.

[BBD+16] Harald Beck, Bruno Bierbaumer, Minh Dao-Tran, Thomas Eiter, Hermann Hellwagner, and Konstantin Schekotihin. Rule-based Stream Reasoning for Intelligent Administration of Content-Centric Networks. In Loizos Michael and Antonis C. Kakas, editors, *Logics in Artificial Intelligence - 15th European Conference, JELIA 2016, Larnaca, Cyprus, November 9-11, 2016, Proceedings*, volume 10021 of *Lecture Notes in Computer Science*, pages 522–528, 2016.

[BBD+17] Harald Beck, Bruno Bierbaumer, Minh Dao-Tran, Thomas Eiter, Hermann Hellwagner, and Konstantin Schekotihin. Stream Reasoning-based Control of Caching Strategies in CCN Routers. In *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–6. IEEE, 2017.

[BBU17] Hamid R. Bazoobandi, Harald Beck, and Jacopo Urbani. Expressive Stream Reasoning with Laser. In Claudia d'Amato, Miriam Fernández, Valentina A. M. Tamma, Freddy Lécué, Philippe Cudré-Mauroux, Juan F. Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, volume 10587 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2017.

[BDD+18] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios E. Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, tools and applications. In Bartocci and Falcone [BF18], pages 135–175.

[BDE15] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Answer Update for Rule-Based Stream Reasoning. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2741–2747. AAAI Press, 2015.

[BDE16] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Equivalent Stream Reasoning Programs. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 929–935. IJCAI/AAAI Press, 2016.

[BDEF15] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. LARS: A Logic-Based Framework for Analyzing Reasoning over Streams. In Bonet and Koenig [BK15], pages 1431–1438.

[BDEF18] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Christian Folie. Stream Reasoning with LARS. *KI*, 32(2-3):193–195, 2018.

[BDG+07] Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker M. White. Cayuga: a high-performance event processing engine. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1100–1102. ACM, 2007.

[BDTE17] Harald Beck, Minh Dao-Tran, and Thomas Eiter. LARS: A Logic-Based Framework for Analytic Reasoning over Streams. Technical Report INFSYS RR-1843-17-03, Institute of Information Systems, TU Vienna, October 2017.

[BDTE18] Harald Beck, Minh Dao-Tran, and Thomas Eiter. LARS: A Logic-based framework for Analytic Reasoning over Streams. *Artificial Intelligence*, 261:16–70, 2018.

[BDTEF14a] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. Towards a Logic-Based Framework for Analyzing Stream Reasoning. In Irene Celino, Óscar Corcho, Daniele Dell'Aglio, Emanuele Della Valle, Markus Krötzsch, and Stefan Schlobach, editors, *Proceedings of the 3rd International Workshop on Ordering and Reasoning Co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 20th, 2014.*, volume 1303 of *CEUR Workshop Proceedings*, pages 11–22. CEUR-WS.org, 2014.

[BDTEF14b] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. Towards Ideal Semantics for Analyzing Stream Reasoning. In *International Work-*

shop on Reactive Concepts in Knowledge Representation, August 19, 2014, Prague, Czech Republic, 2014.

[Bec13] Harald Beck. Inconsistency Management for Traffic Regulations. Master's thesis, TU Wien, Institut für Informationssysteme, TU Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria, 10 2013.

[Bec17] Harald Beck. Reviewing Justification-based Truth Maintenance Systems from a Logic Programming Perspective. Technical Report INFSYS RR-1843-17-02, Institute of Information Systems, TU Vienna, July 2017.

[BEF17] Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A System for Incremental ASP-based Stream Reasoning. *Theory and Practice of Logic Programming*, 17(5-6):744–763, 2017.

[BET11] Gerd Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer Set Programming at a Glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BET16] Gerd Brewka, Thomas Eiter, and Miroslaw Truszczyński, editors. *AI Magazine: Special Issue on Answer Set Programming*. AAAI Press, 2016. Volume 37, number 3 (Fall issue).

[BF18] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In Bartocci and Falcone [BF18], pages 1–33.

[BGAH07] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 363–374, 2007.

[BK98] Fahiem Bacchus and Froduald Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.

[BK14] Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme - Grundlagen, Algorithmen, Anwendungen (5. Aufl.)*. Computational intelligence. SpringerVieweg, 2014.

[BK15] Blai Bonet and Sven Koenig, editors. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. AAAI Press, 2015.

[BKK+17] Sebastian Brandt, Elem Güzel Kalayci, Roman Kontchakov, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Ontology-Based Data Access with a Horn Fragment of Metric Temporal Logic. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 1070–1076. AAAI Press, 2017.

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of Real-Time Properties. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006.

[BP15] Laura Bozzelli and David Pearce. On the Complexity of Temporal Equilibrium Logic. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 645–656. IEEE Computer Society, 2015.

[BW01] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.

[CA15] Jean-Paul Calbimonte and Karl Aberer. Reactive Processing of RDF Streams of Events. In Fabien Gandon, Christophe Guéret, Serena Villata, John G. Breslin, Catherine Faron-Zucker, and Antoine Zimmermann, editors, *The Semantic Web: ESWC 2015 Satellite Events - ESWC 2015 Satellite Events Portorož, Slovenia, May 31 - June 4, 2015, Revised Selected Papers*, volume 9341 of *Lecture Notes in Computer Science*, pages 457–468. Springer, 2015.

[Cad92] Marco Cadoli. The Complexity of Model Checking for Circumscriptive Formulae. *Information Processing Letters*, 44(3):113–118, 1992.

[CÇC+02] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 215–226. Morgan Kaufmann, 2002.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR 2003, First Biennial Conference on Innovative Data Systems*

Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings. www.cidrdb.org, 2003.

[CCG10] Jean-Paul Calbimonte, Óscar Corcho, and Alasdair J. G. Gray. Enabling Ontology-Based Access to Streaming Data Sources. In *9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, volume 6496 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2010.

[CD14] Pedro Cabalar and Martín Diéguez. Strong equivalence of non-monotonic temporal theories. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press, 2014.

[CDE+16] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 379–390. ACM, 2000.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in computer science. Springer, 1990.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.

[CM94] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog (4. ed.)*. Springer, 1994.

[CM10] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In Jean Bacon, Peter R. Pietzuch, Joe Sventek, and Ugur Çetintemel, editors, *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 50–61. ACM, 2010.

[CM12] Gianpaolo Cugola and Alessandro Margara. Complex Event Processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[CV07] Pedro Cabalar and Gilberto Pérez Vega. Temporal equilibrium logic: A first approach. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *Computer Aided Systems Theory - EUROCAST 2007, 11th International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, February 12-16, 2007, Revised Selected Papers*, volume 4739 of *Lecture Notes in Computer Science*, pages 241–248. Springer, 2007.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In Lohman et al. [LSC91], pages 577–589.

[Day88] Umeshwar Dayal. Active Database Management Systems. In Catriel Beeri, Joachim W. Schmidt, and Umeshwar Dayal, editors, *Proceedings of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness, June 28-30, 1988, Jerusalem, Israel. Morgan Kaufmann, 1988*, pages 150–169, 1988.

[DBE15a] Minh Dao-Tran, Harald Beck, and Thomas Eiter. Contrasting RDF Stream Processing Semantics. In Guilin Qi, Kouji Kozaki, Jeff Z. Pan, and Siwei Yu, editors, *Semantic Technology - 5th Joint International Conference, JIST 2015, Yichang, China, November 11-13, 2015, Revised Selected Papers*, volume 9544 of *Lecture Notes in Computer Science*, pages 289–298. Springer, 2015.

[DBE15b] Minh Dao-Tran, Harald Beck, and Thomas Eiter. Towards Comparing RDF Stream Processing Semantics. In Daniela Nicklas and Özgür Lütfü Özçep, editors, *Proceedings of the 1st Workshop on High-Level Declarative Stream Processing co-located with the 38th German AI conference (KI 2015), Dresden, Germany, September 22, 2015.*, volume 1447 of *CEUR Workshop Proceedings*, pages 15–27. CEUR-WS.org, 2015.

[DCvF09] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intelligent Systems*, 24:83–89, 2009.

[DDC+16] Daniele Dell'Aglio, Minh Dao-Tran, Jean-Paul Calbimonte, Danh Le Phuoc, and Emanuele Della Valle. A Query Model to Capture Event Pattern Matching in RDF Stream Processing Query Languages. In Eva Blomqvist, Paolo Ciancarini, Francesco Poggi, and Fabio Vitali, editors, *Knowledge Engineering and Knowledge Management - 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings*, volume 10024 of *Lecture Notes in Computer Science*, pages 145–162, 2016.

[DDvB17] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream Reasoning: A Survey and Outlook. *Data Science*, 1(1-2):59–83, 2017.

[DEF+12] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. OMiGA : An Open Minded Grounding On-The-Fly Answer Set Solver. In Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors, *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, volume 7519 of *Lecture Notes in Computer Science*, pages 480–483. Springer, 2012.

[DFM13] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient Robust Monitoring for STL. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 264–279. Springer, 2013.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[DGH+06] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards Expressive Publish/Subscribe Systems. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 627–644. Springer, 2006.

[DGP+07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A General Purpose Event Monitoring System. In *CIDR 2007, Third Biennial Conference on*

Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings, pages 412–422, 2007.

[dK86] Johan de Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[DK08] Patrick Doherty and Jonas Kvarnström. Temporal Action Logics. In Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 709–757. Elsevier, 2008.

[DKH09] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A Temporal Logic-based Planning and Execution Monitoring Framework for Unmanned Aircraft Systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[dlBP08] Maria Garcia de la Banda and Enrico Pontelli, editors. *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*. Springer, 2008.

[dLH14] Daniel de Leng and Fredrik Heintz. Towards on-demand semantic event processing for stream reasoning. In *17th International Conference on Information Fusion, FUSION 2014, Salamanca, Spain, July 7-10, 2014*, pages 1–8. IEEE, 2014.

[dLH16] Daniel de Leng and Fredrik Heintz. Qualitative Spatio-Temporal Stream Reasoning with Unobservable Intertemporal Spatial Relations Using Landmarks. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the 30th AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 957–963. AAAI Press, 2016.

[DLL11] Thang M. Do, Seng Wai Loke, and Fei Liu. Answer Set Programming for Stream Reasoning. In *Advances in Artificial Intelligence - 24th Canadian Conference on Artificial Intelligence, St. John's, Canada, May 25-27, 2011. Proceedings*, volume 6657 of *Lecture Notes in Computer Science*, pages 104–109. Springer, 2011.

[Doy79] Jon Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231–272, 1979.

[DS90] Mohammad Dadashzadeh and David W. Stemple. Converting SQL queries into relational algebra. *Information & Management*, 19(5):307–323, 1990.

[DS02] Stéphane Demri and Philippe Schnoebelen. The Complexity of Propositional Linear Temporal Logics in Simple Cases. *Information and Computation*, 174(1):84–103, 2002.

[DTM+13] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. Modeling the execution semantics of stream processing engines with SECRET. *The VLDB Journal*, 22(4):421–446, 2013.

[DVCC14] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Óscar Corcho. RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. *International Journal on Semantic Web and Information Systems*, 10(4):17–44, 2014.

[EF03] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.

[EFW07] Thomas Eiter, Michael Fink, and Stefan Woltran. Semantical Characterizations and Complexity of Equivalences in Answer Set Programming. *ACM Transactions on Computational Logic*, 8(3), July 2007.

[EG95] Thomas Eiter and Georg Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems - 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30-September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[EIST06] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A Prover for Semantic-Web Reasoning under the Answer-Set Semantics. In *IEEE / WIC / ACM International Conference on Web Intelligence, 18-22 December 2006, Hong Kong, China*, pages 1073–1074. IEEE Computer Society, 2006.

[Elk90] Charles Elkan. A Rational Reconstruction of Nonmonotonic Truth Maintenance Systems. *Artificial Intelligence*, 43(2):219–234, 1990.

[EMRS15] Thomas Eiter, Mustafa Mehuljic, Christoph Redl, and Peter Schüller. User Guide: dlvhex 2.X. Technical Report INFSYS RR-1843-15-05, Institute of Information Systems, TU Vienna, September 2015.

[FLP04] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence - 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 200–212. Springer, 2004.

[Gab87] Dov M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Temporal Logic in Specification, Altrincham, UK, April 8-10, 1987, Proceedings*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1987.

[GADI08] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On Supporting Kleene Closure over Event Streams. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 1391–1393. IEEE Computer Society, 2008.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: the System S Declarative Stream Processing Engine. In Wang [Wan08], pages 1123–1134.

[GGK+12] Martin Gebser, Torsten Grote, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, and Torsten Schaub. Stream reasoning with answer set programming: Preliminary report. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press, 2012.

[GGKS11] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In James P. Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2011.

[Gha96] Malik Ghallab. On chronicles: Representation, on-line recognition and learning. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996.*, pages 597–606. Morgan Kaufmann, 1996.

[GHM+07] Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref, and Ahmed K. Elmagarmid. Incremental Evaluation of Sliding-

Window Queries over Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):57–72, 2007.

[GKK+08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an Incremental ASP Solver. In de la Banda and Pontelli [dlBP08], pages 190–205.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + Control: Preliminary Report. *CoRR*, abs/1405.3694, 2014. Theory and Practice of Logic Programming, Online Supplement.

[GKKS17] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017.

[GKOS15] Martin Gebser, Roland Kaminski, Philipp Obermeier, and Torsten Schaub. Ricochet Robots Reloaded: A Case-Study in Multi-shot ASP Solving. In Thomas Eiter, Hannes Strass, Miroslaw Truszczynski, and Stefan Woltran, editors, *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation - Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, volume 9060 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2015.

[GL88] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993.

[GÖ10] Lukasz Golab and M. Tamer Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[GPLT91] Michael Gelfond, Halina Przymusinska, Vladimir Lifschitz, and Miroslaw Truszczynski. Disjunctive Defaults. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991.*, pages 230–237. Morgan Kaufmann, 1991.

[GPSS80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the Temporal Analysis of Fairness. In Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors, *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 163–173. ACM Press, 1980.

[GRU17] Alejandro Grez, Cristian Riveros, and Martín Ugarte. Foundations of Complex Event Processing. *CoRR*, abs/1709.05369, 2017.

[GSS11] Martin Gebser, Orkunt Sabuncu, and Torsten Schaub. An Incremental Answer Set Programming Based System for Finite Model Computation. *AI Communications*, 24(2):195–212, 2011.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book (2. ed.)*. Pearson Education, 2009.

[HD04] Fredrik Heintz and Patrick Doherty. DyKnow: An Approach to Middleware for Knowledge Processing. *Journal of Intelligent and Fuzzy Systems*, 15(1):3–13, 2004.

[HD12] Fredrik Heintz and Zlatan Dragisic. Semantic information integration for stream reasoning. In *15th International Conference on Information Fusion, FUSION 2012, Singapore, July 9-12, 2012*, pages 1454–1461. IEEE, 2012.

[HdL13] Fredrik Heintz and Daniel de Leng. Semantic information integration with transformations for stream reasoning. In *Proceedings of the 16th International Conference on Information Fusion, FUSION 2013, Istanbul, Turkey, July 9-12, 2013*, pages 445–452. IEEE, 2013.

[HdL14] Fredrik Heintz and Daniel de Leng. Spatio-Temporal Stream Reasoning with Incomplete Spatial Information. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 429–434. IOS Press, 2014.

[Hei09] Fredrik Heintz. *DyKnow : A Stream-Based Knowledge Processing Middleware Framework*. PhD thesis, Linköping University, Sweden, 2009.

[Hey30] Arend Heyting. Die formalen Regeln der intuitionistischen Logik. In *Sitzungsberichte der preußischen Akademie der Wissenschaften. phys.-math. Klasse*, pages 42–65, 57–71, 158–169, 1930.

[HKD10a] Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Bridging the sense-reasoning gap: DyKnow - Stream-based middleware for knowledge processing. *Advanced Engineering Informatics*, 24(1):14–26, 2010.

[HKD10b] Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Stream-Based Reasoning in DyKnow. In Gerhard Lakemeyer, Hector J. Levesque, and Fiora Pirri, editors, *Cognitive Robotics, 21.02. - 26.02.2010*, volume 10081 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

[HKD10c] Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty. Stream-Based Reasoning Support for Autonomous Systems. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 183–188. IOS Press, 2010.

[HMH18] Pan Hu, Boris Motik, and Ian Horrocks. Optimised Maintenance of Datalog Materialisations. In McIlraith and Weinberger [MW18].

[HOW14] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online Monitoring of Metric Temporal Logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2014.

[HSP13] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. *W3C recommendation*, 21(10), 2013.

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. Towards a Streaming SQL Standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, 2008.

[JST+09] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca Braynard. Networking Named Content. In Jörg Liebeherr, Giorgio Ventre, Ernst W. Biersack, and Srinivasan Keshav, editors, *Proceedings of the 2009 ACM Conference on Emerging Networking Experiments and Technology, CoNEXT 2009, Rome, Italy, December 1-4, 2009*, pages 1–12. ACM, 2009.

[KBJ+16] Evgeny Kharlamov, Sebastian Brandt, Ernesto Jiménez-Ruiz, Yannis Kotidis, Steffen Lamparter, Theofilos Mailis, Christian Neuenstadt, Özgür L. Özçep, Christoph Pinkel, Christoforos Svingos, Dmitriy Zheleznyakov, Ian Horrocks, Yannis E. Ioannidis, and Ralf Möller. Ontology-Based Integration of Streaming and Static Relational Data with Optique. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2109–2112. ACM, 2016.

[KHD08] Jonas Kvarnström, Fredrik Heintz, and Patrick Doherty. A Temporal Logic-Based Planning and Execution Monitoring System. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen, editors, *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, pages 198–205. AAAI, 2008.

[Koy90] Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.

[Lem06] Daniel Lemire. Streaming Maximum-Minimum Filter Using No More than Three Comparisons per Element. *Nordic Journal of Computing*, 13(4):328–339, 2006.

[Lif08] Vladimir Lifschitz. Twelve Definitions of a Stable Model. In de la Banda and Pontelli [dlBP08], pages 37–51.

[LMS02] François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal Logic with Forgettable Past. In *17th IEEE Symposium on Logic in Computer Science, LICS 2002, 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 383–392. IEEE Computer Society, 2002.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 311–322. ACM, 2005.

[LN09] Claire Lefèvre and Pascal Nicolas. The First Version of a New ASP Solver : ASPeRiX. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer, 2009.

[LPF+06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[LPV01] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The Glory of the Past. In Rohit Parikh, editor, *Logics of Programs, Conference, Brooklyn*

College, New York, NY, USA, June 17-19, 1985, Proceedings, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[LS95] François Laroussinie and Philippe Schnoebelen. A Hierarchy of Temporal Logics with Past. *Theoretical Computer Science*, 148(2):303–324, 1995.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[LSC91] Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors. *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*. Morgan Kaufmann, 1991.

[Luc96] David C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. In Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors, *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 329–358. DIMACS/AMS, 1996.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[MAPH13] Alessandra Mileo, Ahmed Abdelrahman, Sean Policarpio, and Manfred Hauswirth. StreamRule: A Nonmonotonic Stream Reasoning System for the Semantic Web. In *Web Reasoning and Rule Systems - 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013. Proceedings*, volume 7994 of *Lecture Notes in Computer Science*, pages 247–252. Springer, 2013.

[Mar03] Nicolas Markey. Temporal logic with past is exponentially more succinct, Concurrency Column. *Bulletin of the EATCS*, 79:122–128, 2003.

[MC11] Alessandro Margara and Gianpaolo Cugola. Processing flows of information: from data stream to complex event processing. In David M. Eyers, Opher Etzion, Avigdor Gal, Stanley B. Zdonik, and Paul Vincent, editors, *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11-15, 2011*, pages 359–360. ACM, 2011.

[MCCD18] Alessandro Margara, Gianpaolo Cugola, Dario Collavini, and Daniele Dell'Aglio. Efficient Temporal Reasoning on Streams of Events with DOTR. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, volume 10843 of *Lecture Notes in Computer Science*, pages 384–399. Springer, 2018.

[McD91] Drew V. McDermott. A General Framework for Reason Maintenance. *Artificial Intelligence*, 50(3):289–329, 1991.

[MDEF17] Alessandra Mileo, Minh Dao-Tran, Thomas Eiter, and Michael Fink. Stream Reasoning. In *Encyclopedia of Database Systems, 2nd edition*, page 7 pp. Springer Science+Business Media, 2017.

[MN04] Oded Maler and Dejan Nickovic. Monitoring Temporal Properties of Continuous Signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004.

[MN13] Oded Maler and Dejan Nickovic. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15(3):247–268, 2013.

[MNPH15] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of datalog materialisation: the backward/forward algorithm. In Bonet and Koenig [BK15], pages 1560–1568.

[MT91] V. Wiktor Marek and Miroslaw Truszczynski. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.

[MW18] Sheila A. McIlraith and Kilian Q. Weinberger, editors. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*. AAAI Press, 2018.

[NM07] Dejan Nickovic and Oded Maler. AMT: A Property-Based Monitoring Tool for Analog Systems. In Jean-François Raskin and P. S. Thiagarajan, editors, *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, volume 4763 of *Lecture Notes in Computer Science*, pages 304–319. Springer, 2007.

[NM14] Matthias Nickles and Alessandra Mileo. Web Stream Reasoning Using Probabilistic Answer Set Programming. In *Web Reasoning and Rule Systems - 8th International Conference, RR 2014, Athens, Greece, September 15-17, 2014. Proceedings*, volume 8741 of *Lecture Notes in Computer Science*, pages 197–205. Springer, 2014.

[NM15] Matthias Nickles and Alessandra Mileo. A Hybrid Approach to Inference in Probabilistic Non-Monotonic Logic Programming. In *Proceedings of the 2nd International Workshop on Probabilistic Logic Programming co-located with 31st International Conference on Logic Programming, ICLP 2015, Cork, Ireland, August 31st, 2015*, volume 1413 of *CEUR Workshop Proceedings*, pages 57–68. CEUR-WS.org, 2015.

[OJ06] Emilia Oikarinen and Tomi Janhunen. Modular Equivalence for Normal Logic Programs. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *17th European Conference on Artificial Intelligence, ECAI 2006, August 29-September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems, PAIS 2006, Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 412–416. IOS Press, 2006.

[ÖMN15] Özgür Lütfü Özçep, Ralf Möller, and Christian Neuenstadt. Stream-Query Compilation with Ontologies. In Bernhard Pfahringer and Jochen Renz, editors, *Advances in Artificial Intelligence - 28th Australasian Joint Conference, AI 2015, Canberra, ACT, Australia, November 30 - December 4, 2015, Proceedings*, volume 9457 of *Lecture Notes in Computer Science*, pages 457–463. Springer, 2015.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):16:1–16:45, 2009.

[PD99] Norman W. Paton and Oscar Díaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[PDP+12] Danh Le Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter A. Boncz, Thomas Eiter, and Michael Fink. Linked Stream Data Processing Engines: Facts and Figures. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, volume 7650 of *Lecture Notes in Computer Science*, pages 300–312. Springer, 2012.

[PDPH11] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 370–388. Springer, 2011.

[PDPR09] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Answer Set Programming with Constraints Using Lazy Grounding. In Patricia M. Hill and David Scott Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2009.

[Pea06] David Pearce. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence*, 47(1-2):3–41, 2006.

[PMA17] Thu-Le Pham, Alessandra Mileo, and Muhammad Intizar Ali. Towards Scalable Non-Monotonic Stream Reasoning via Input Dependency Analysis. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1553–1558. IEEE Computer Society, 2017.

[PNPH12] Danh Le Phuoc, Hoan Quoc Nguyen-Mau, Josiane Xavier Parreira, and Manfred Hauswirth. A Middleware Framework for Scalable Management of Linked Streams. *Journal of Web Semantics*, 16:42–51, 2012.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, FOCS 1977, Providence, Rhode Island, USA, 31 October-1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[Pol07] Axel Polleres. From SPARQL to rules (and back). In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 787–796. ACM, 2007.

[Pri67] Arthur Prior. *Past, Present and Future.* Oxford University Press, 1967.

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[RCN+17] Xiangnan Ren, Olivier Curé, Hubert Naacke, Jérémy Lhez, and Li Ke. Strider^{r}: Massive and distributed RDF graph stream reasoning. In Jian-Yun Nie, Zoran Obradovic, Toyotaro Suzumura, Rumi Ghosh, Raghunath Nambiar, Chonggang Wang, Hui Zang, Ricardo A. Baeza-Yates, Xiaohua Hu, Jeremy Kepner, Alfredo Cuzzocrea, Jian Tang, and Masashi Toyoda, editors, *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 3358–3367. IEEE, 2017.

[Rei80] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[RKG+18] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, Boris Motik, and Ian Horrocks. Stream Reasoning in Temporal Datalog. In McIlraith and Weinberger [MW18].

[RN01] Jochen Renz and Bernhard Nebel. Efficient Methods for Qualitative Spatial Reasoning. *Journal of Artificial Intelligence Research*, 15:289–318, 2001.

[RP11] Yuan Ren and Jeff Z. Pan. Optimising Ontology Stream Reasoning with Truth Maintenance System. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 831–836. ACM, 2011.

[Sav70] Walter J. Savitch. Relationships Between Nondeterministic and Deterministic Tape Complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[SC85] A. Prasad Sistla and Edmund M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, 1985.

[SJ96] Martin Staudt and Matthias Jarke. Incremental Maintenance of Externally Materialized Views. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 75–86. Morgan Kaufmann, 1996.

[SPAM91] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In Lohman et al. [LSC91], pages 469–478.

[Ste97] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[SW04] Utkarsh Srivastava and Jennifer Widom. Flexible Time Management in Data Stream Systems. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 263–274. ACM, 2004.

[SW18] Torsten Schaub and Stefan Woltran. Special Issue on Answer Set Programming. *KI*, 32(2-3):101–103, 2018.

[TGNO92] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. Continuous Queries over Append-Only Databases. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992.*, pages 321–330. ACM Press, 1992.

[TH16] Mattias Tiger and Fredrik Heintz. Stream Reasoning Using Temporal Logic and Predictive Probabilistic State Models. In Curtis E. Dyreson, Michael R. Hansen, and Luke Hunsberger, editors, *23rd International Symposium on Temporal Representation and Reasoning, TIME 2016, Kongens Lyngby, Denmark, October 17-19, 2016*, pages 196–205. IEEE Computer Society, 2016.

[Tur01] Hudson Turner. Strong Equivalence for Logic Programs and Default Theories (Made Easy). In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2001.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):619–649, July 1991.

[VSM05] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. *Journal on Data Semantics*, 2:1–34, 2005.

[Wan08] Jason Tsong-Li Wang, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. ACM, 2008.

[WC96] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance Complex Event Processing over Streams. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 407–418. ACM, 2006.

[Wei17] Antonius Weinzierl. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In Marcello Balduccini and Tomi Janhunen, editors,

Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings, volume 10377 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2017.

[Wol04] Stefan Woltran. Characterizations for relativized notions of equivalence in answer set programming. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 161–173. Springer, 2004.

[WYG+07] Kun-Lung Wu, Philip S. Yu, Bugra Gedik, Kirsten Hildrum, Charu C. Aggarwal, Eric Bouillet, Wei Fan, David George, Xiaohui Gu, Gang Luo, and Haixun Wang. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1185–1196. ACM, 2007.

[XCK+18] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-Based Data Access: A Survey. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 5511–5519. ijcai.org, 2018.

[Zan12] Carlo Zaniolo. Logical Foundations of Continuous Query Languages for Data Streams. In *Datalog in Academia and Industry - 2nd International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*, pages 177–189. Springer, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65, 2016.