Università della Calabria
Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica
XXXV Ciclo

Settore Disciplinare INF/01 INFORMATICA

Tesi di Dottorato

Design and Implementation of an ASP-based Stream Reasoner

Elena Mastria

Supervisori
Prof. Francesco Calimeri
Prof.ssa Simona Perri
Dott.ssa Jessica Zangari

Coordinatore
Prof. Giorgio Terracina

A.A. 2022/2023

Università della Calabria
Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica
XXXV Ciclo

Settore Disciplinare INF/01 INFORMATICA

Tesi di Dottorato

Design and Implementation of an ASP-based Stream Reasoner

Elena Mastria

Elena Mastria
03.07.2023 12:40:56
GMT+00:00

Supervisori

Prof. Francesco Calimeri

FRANCESCO CALIMERI
03.07.2023 14:55:21
GMT+01:00

Prof.ssa Simona Perri

SIMONA PERRI
03.07.2023 13:52:49
GMT+00:00

Dott.ssa Jessica Zangari

Jessica Zangari
03.07.2023 14:48:25 GMT+01:00

Coordinatore

Prof. Giorgio Terracina

Terracina
Giorgio

A.A. 2022/2023

Design and Implementation of an ASP-based Stream Reasoner

Elena Mastria

Dipartimento di Matematica e Informatica,
Università della Calabria
87036 Rende, Italy
email: elena.mastria@unical.it

Sommario

Lo Stream Reasoning (SR) è un campo di ricerca relativamente giovane che si è evoluto dallo Stream Processing (SP) più di un decennio fa. Esso tratta principalmente lo studio e lo svuluppo di approcci e tecniche avanzate per effettuare inferenze su flussi di dati altamente dinamici in modo continuo. Tali flussi consistono in una sequenza infinita di *informazioni* che cambiano *dinamicamente* nel tempo. Essi vengono generati da sorgenti (come sensori, dispositivi, reti sociali, ecc) che monitorano un ambiente, fisico o virtuale, registrandone continuamente lo stato e i cambiamenti. Mentre lo SP mira a processare velocemente i flussi di dati valutando query sui loro elementi, lo SR mira a qualcosa di più complesso. In particolare, esso deriva nuove informazioni tenendo in considerazione sia il contenuto dei flussi di dati che la conoscenza di base sul dominio di applicazione.

Di recente, lo SR è stato studiato in diversi campi ed è diventato sempre più rilevante scenari applicativi come: IoT, Città Intelligenti, Gestione delle Emergenze, e Assistenza Sanitaria. In questi tipi di contesti, è richiesto che le applicazioni rispondano a query complesse in un breve lasso di tempo: *real-time* ($<$ 1 secondo) o *near real-time* ($<$ 1 minuto) a seconda dei vincoli del dominio applicativo trattato. Per tanto, un sistema per lo SR (chiamato *stream reasoner*) deve essere in grado di effettuare task di ragionamento complesso continuando a processare efficientemente flussi di dati *eterogenei* insieme a *vaste* basi di conoscenza.

Diversi approcci per lo SR sono stati proposti in campi come Data Stream Management System (DSMS), Complex Event Processing (CEP), Semantic Web e Knowledge Representation and Reasoning (KRR). Fra i paradigmi KRR, l'Answer Set Programming (ASP) è un formalismo ben noto sviluppato nell'area della programmazione logica e ragionamento non monotono. Grazie all'esistenza di implementazioni efficienti, l'ASP è stato impiegato con successo al di fuori

del campo accademico per risolvere problemi derivanti da applicazioni reali. Inoltre, recentemente, esso ha guadagnato attenzioni come una possibile base per lo SR. In tal senso, sono stati fatti notevoli passi in avanti e diverse soluzioni basate su ASP sono state proposte: alcune combinano implementazioni per lo SP e ASP all'interno dello stesso sistema, altre estendono nativamente ASP con construtti per ragionare su flussi di dati. Tuttavia, gli stream reasoner basati su ASP risultano essere ancora immaturi per poter affrontare i requisiti derivanti dallo SR.

Questa tesi si focalizza nella progettazione e implementazione di un nuovo e affidabile stream reasoner basato su ASP. L'obbiettivo principale è quindi quello di ottenere un sistema che abbia le seguenti caratteristiche: *(i)* scali efficientemente su domini applicativi reali; *(ii)* supporti un linguaggio che erediti la natura altamente dichiarativa e la semplicità di utilizzo da ASP; *(iii)* sia facilmente estendibile con nuovi costrutti rilevanti in scenari pratici.

In questo lavoro viene presentato lo stream reasoner I-DLV-sr. Il suo linguaggio è una lineare estensione di ASP con costrutti che consentono di ragionare su flussi di dati. La sua implementazione si basa su una stretta interazione fra due soluzioni all'avanguardia nel campo dello SP e ASP: $\mathcal{I}^2$-*DLV* e *Apache Flink*, rispettivamente. I-DLV-sr è stato testato su diversi domini reali e artificiali al fine di esplorare le sue capacità nel modellare scenari SR e verificare le sue prestazioni. Negli esperimenti condotti, il sistema ha ottenuto buoni risultati dimostrando che l'approccio proposto in questa tesi è valido e che la sua implementazione è robusta.

Abstract

Stream Reasoning (SR) is a relatively young research field that evolved from Stream Processing (SP) more than a decade ago. It focuses on studying and developing advanced approaches and techniques for the continuous application of inference techniques to highly dynamic data streams. Data streams are (theoretically) *infinite* streams of information that *dynamically* change over time. These are generated by sources (e.g., sensors, devices, social networks, etc.) that monitor a physical or virtual environment, continuously reporting the relative state and changes. While SP aims at quickly processing data streams while answering continuous queries on their elements, SR tackles inferencing new information taking into account the content of data streams along with background knowledge on the application domain.

Recently, SR has been studied in several fields, and has become more and more relevant in diverse application scenarios, such as IoT, Smart Cities, Emergency Management, and Healthcare. In such types of context, applications require complex query answering in a minimal amount of time. This amount is defined from the application domain at hand and is typically *real-time* (< 1 second) or *near real-time*(< 1 minute). Therefore, an SR system (i.e., *stream reasoner*) must be able to perform complex reasoning tasks while efficiently processing *heterogeneous* data streams together with *large* background knowledge bases.

Different SR approaches have been proposed in fields such as Data Stream Management Systems (DSMS), Complex Event Processing (CEP), Semantic Web, and Knowledge Representation and Reasoning (KRR). Among declarative KRR paradigms, Answer Set Programming (ASP) is a well-established formalism developed in the area of logic programming and non-monotonic reasoning. Thanks to the availability of robust and efficient implementations, ASP is successfully employed outside of academia to implement several real-world

applications. Recently, ASP gained attention as a basis for SR, and significant steps in this direction have been taken.

Several ASP-based solutions have been proposed: some combining SP and ASP implementations into a single engine, others natively extending ASP with SR constructs. However, existing ASP-based stream reasoners appear not mature enough concerning the desirable requirements for SR.

This thesis focuses on designing and implementing a novel, reliable ASP-based stream reasoner. The main goal is to obtain a system featuring the following properties: *(i)* efficiently scale over real-world application domains; *(ii)* support a language that inherits the highly declarative nature and ease of use from ASP; *(iii)* easily extendable with new constructs that are relevant for practical SR scenarios. Therefore, we herein present the stream reasoner I-DLV-sr. The input language is a straightforward extension of ASP with constructs to reason over data streams. The implementation relies on a tight interaction between two state-of-the-art solutions in ASP and SP: $\mathcal{I}^2$-*DLV* and *Apache Flink*, respectively.

We tested I-DLV-sr on several real-world and synthetic domains to explore its capabilities in modeling SR scenarios and assess its performance. In the conducted experiments, the system obtained good results, proving the viability of the proposed approach and the robustness of the implementation herein presented.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, we face a data-centered world; thanks in part to the spread of the Internet and interconnected technologies, modern environments are rich in sources that generate, continuously and rapidly, high-volume data. For instance, data can constantly be produced by software applications (messages, statuses), sensors (measurements of the environment), devices (parameters, errors), and online social interactions (likes, comments, updates). All these data constitute an *infinite* stream of information that *dynamically* changes over time according to the environment they come from.

States of the environment change, and observations that are true in one moment cease to be valid in another. The same dynamism applies to the implicit consequences of such observations. Consequently, reasoning over streams requires new considerations and pondering about the specific challenges they pose.

Several modern applications in real-world contexts, such as IoT, Smart Cities, Emergency Management, and Healthcare, need solutions that integrate advanced reasoning techniques and efficient data stream management. This need has given rise to *Stream Reasoning (SR)*: a research field which gained more and more popularity in the last decade. *SR* has been defined as *the application of inference techniques to highly dynamic data streams* [52].

Different SR approaches have been proposed [17, 108, 83, 105] in contexts like Data Stream Management Systems (DSMS), Complex Event Processing (CEP), Semantic Web, and Knowledge Representation and Reasoning (KRR). DSMS and CEP offered continuous query application and data stream monitoring solutions but lacked expressiveness and complex reasoning capabilities. The

Semantic Web community focused on extending RDF [110] and SPARQL [112] for representing and querying streaming data. Many powerful *RDF Stream Processing* (RSP) engines have been proposed [33, 17, 13, 108]. In the KRR field, Answer Set Programming (ASP) [30] gained attention as a particularly attractive basis for SR. ASP is a powerful declarative paradigm for KRR developed in the area of non-monotonic reasoning and logic programming. Beyond the academic scope, ASP has been successfully employed to deal with real-world problems and applications, thanks to the availability of robust and efficient implementations [72].

1.1 Stream Reasoning via Answer Set Programming

While existing data-stream management systems allow for high-throughput stream processing, these solutions lack complex reasoning capacities. At some point in recent years, it became apparent that more advanced and powerful knowledge representation and reasoning mechanisms are needed, going beyond description logic or semi-formal query languages. In this respect, with its rich modeling and reasoning capabilities, ASP became an influential paradigm in the field. The use of ASP for stream reasoning offers considerable advantages. Thanks to years of research in the area, it relies on solid foundations and comes with well-defined semantics and optimized inference engines. Knowledge-intensive stream reasoning based on ASP shifts the emphasis from rapid data processing to complex reasoning [65]. Different lines of research on adapting and extending ASP for stream reasoning exist.

On the one hand, extensions of the language have been proposed. In time-decaying logic programs based on Reactive ASP [67, 65, 66], a set of annotations has been introduced in order to reason over data and rules that change in time. The proposal of LARS (a Logic-based framework for Analytic Reasoning over Streams) [23] natively extends ASP by providing flexible expiration control (window operators that allow selecting relevant subsets of data) and temporal modalities ("always", "sometimes" and "at" modifiers that can be added to logic formulas).

On the other hand, the integration of ASP reasoners with stream processing solutions has been studied, starting from simple architectures [53] up to optimization techniques based on window size adaptations, input data analysis and

parallelization [99, 78, 107], or data noise and incompleteness management [101].

These two approaches interleave in the research landscape of stream reasoning with ASP. Particular implementations adopt various theoretical proposals and integrate engines that support different languages. However, also solutions relying on both approaches exist. C-ASP [105] is a language that extends the ASP syntax and semantics with constructs for continuous reasoning requests over RDF streams; a designated implementation that combines an ASP system with an RSP engine supports such formalism.

1.2 Main Contribution

State-of-the-art ASP-based stream reasoners appear not mature enough concerning the desirable requirements for SR [52]. Hence, there is still room for improvement, especially when dealing with real applications. For instance, some systems are weakly usable in practice (e.g., enforce strict assumptions in input programs), and others suffer from efficiency/scalability issues.

This thesis aims to propose a solution relying on ASP that can be successfully and efficiently used in real-world SR scenarios. Therefore, we herein propose I-DLV-sr: a novel ASP-based stream reasoner [42, 44].

I-DLV-sr is the result of extensive research activity focused on different aspects:

- study of real-world SR applications aiming at identifying main problems that have to be modeled/solved;
- designing an input language expressive enough to address typical SR problems while being effectively supported by the implementation;
- designing and developing an efficient and reliable system that can be used in practical contexts;
- study of state-of-the-art reasoning and optimization techniques helpful to deal with the high complexity typical of SR field.

The input language of I-DLV-sr inherits the declarative nature and ease of use from ASP, extending the basic language with many constructs to reason over streams. Specifically, it allows users to use *streaming literals*, ***@now*** terms, *trigger rules*, and *temporary rules* in logic programs. Streaming literals can be used within the rule bodies. These create windows (i.e., snapshots of the

stream) and evaluate logic formulas over inputs received at different time points. Streaming literals can feature the operators: *in*, *always*, *count*, *at least*, and *at most*. Recursion involving streaming literals is allowed. The **@now** construct enables users to model the concept of time in rule bodies explicitly. It is a special term that, at each evaluation time point t, takes t as its value. Trigger rules are special rules whose evaluation is performed (i.e., triggered) with a frequency specified by the user. Temporary rules make it possible to mark derivations obtained through their evaluation as temporary, i.e., these can be safely forgotten from one evaluation time point to the next one.

The implementation of I-DLV-sr relies on the proper integration of two well-established solutions in the field of Stream Processing and ASP: *Apache Flink* (*Flink*) [85] and $\mathcal{I}^2$-*DLV* [86], respectively. *Flink* is a powerful distributed stream processor, while $\mathcal{I}^2$-*DLV* is an ASP grounder and a full-fledged deductive database system that enables incremental ASP evaluation via overgrounding techniques [34]. On the one hand, a Java application built on top of *Flink* is used to manage data streams efficiently. On the other hand, $\mathcal{I}^2$-*DLV* is used for performing reasoning tasks.

We tested the system to assess its reliability, performance, and scalability and explore ease of modeling and reasoning capabilities. In particular, we conducted several experiments on both real-world and synthetic domains. The results obtained from this experimental activity are encouraging, proving the viability of the proposed approach and the robustness of the implementation herein presented.

Eventually, we participated in the *Stream Reasoning Hackathon 2021*[1], an event part of the *Stream Reasoning Workshop 2021*. This Hackathon was designed as a "model and solve" challenge on scenarios concerning the implementation of Intelligent Transport Systems (ITS) [117]. In this event, we used the herein presented system to deal with the proposed tasks. I-DLV-sr demonstrated good modeling and solving capabilities allowing reasoning on both streaming and background knowledge. These features have been awarded by the organization committee along with others participants so that our solution ranked first.

[1] `https://streamreasoning.org/events/stream-reasoning-hackathon-2021/` (url date: 18/01/2022)

1.3 Organization of the Thesis

This thesis consists of five parts:

- The first part provides an overview of Stream Processing and Stream Reasoning, highlighting their main purposes and differences. In addition, it describes the stream processor *Apache Flink*, focusing on the main aspects, crucial for the presentation of I-DLV-sr.
- The second part introduces ASP foundations and recalls state-of-the-art implementations; moreover, it discusses rewriting techniques for optimizing ASP programs.
- The third part explores state-of-the-art ASP-based solutions for Stream Reasoning; in particular, we recall the LARS framework and illustrates some ASP-based stream reasoning systems, discussing their modeling capabilities and evaluation mechanisms.
- The fourth part presents I-DLV-sr. In particular, it first defines the input language, then it describes the system architecture and provides details on its implementation; finally, it demonstrates its capabilities in modeling several SR scenarios and discusses the results of an experimental activity carried out for assessing its performance.
- The fifth part compares related work and concludes, outlining future and ongoing work.

Part I

Stream Processing and Stream Reasoning

Chapter 2

Processing and Reasoning over Data Streams

This chapter discusses Stream Processing and Stream Reasoning, focusing on their motivation and summarizing the leading solutions proposed in both fields.

Section 2.1 is dedicated to Stream Processing, while Section 2.2 is centered on Stream Reasoning.

2.1 Stream Processing

Stream Processing (SP) consists of continuously processing data events in real-time, i.e., as soon as they are received.

Early SP solutions come from Data Stream Management Systems (DSMSs) and Complex Event Processing (CEP) [48].

DSMSs born as a natural evolution of Database Management Systems (DBMSs) with the substantial difference that DSMSs deal with transient data that are continuously updated and execute continuous queries providing updated answers as new data are available. Relying on DBMS foundations, DSMSs use SQL-based operators, like selections, aggregates, joins, and relational algebra, for processing incoming data.

CEP systems are an extension to traditional publish-subscribe systems, which allow subscribers to express their interest in composite events. Publish-subscribe systems adopt a message-oriented interaction paradigm: on the one hand, users express their interest in receiving some information by subscribing to specific

classes of events; on the other hand, the publish-subscribe engine addresses events coming from sources of data based on user subscriptions. Therefore, CEP systems focus on identifying specific patterns of low-level streaming events that represent high-level events in which users expressed some interest. Specifically, they collect input data representing eternal world events that are subsequently analyzed and combined to derive new meaningful composite events, i.e., explanations or information on what is happening in the observed environment.

In general, SP solutions allow getting real-time, valuable insights on business trends useful to make fast decisions and changes. Therefore, businesses relying on real-time analytics and monitoring for decision-making processes are interested in advancing this field. In such a context, SP events involve business transactions or logs from different platforms, information reports from external sources that are useful for decision-makers, and data collected from sensors [58].

Business analytics is only one of many domains where SP can be successfully employed. For instance, healthcare, smart cities, transportation, finance, and many others use SP solutions for continuous monitoring, resource management and maintenance, fault detection, etc.

Modern SP solutions perform multiple low-level actions on streaming events like filters, aggregations, counting, analytics, transformations, enrichment, branching, joining, and so on. Existing SP framework mainly fall into two categories [58]: *Native Streaming* and *Micro-batching*.

Native Streaming refers to systems that continuously process streaming events coming from different sources as soon as they arrive, i.e., without any latency. Examples of native streaming systems are Storm [114, 87, 60], Flink [46, 85, 113], Kafka Streams [115, 25], and Samza [1, 102]. These feature a minimum latency, a high throughput, and easy state management, but they also suffer from a challenging fault tolerance, i.e., restore the complete state of an application and resume processing in the case of failures.

Micro-batching refers to systems that adopt a continuous processing approach based on micro-batches. This approach groups streaming events into a predefined time window called *batch* and processes the contained events only when a batch time window is complete. An example of a micro-batching system is Spark Streaming [2]. In contrast with native streaming systems, micro batching systems suffer from a higher latency, a lower throughput, and more challenging state management, but they feature a better fault tolerance.

2.2 Stream Reasoning

Stream Reasoning (SR) [52] consists in the application of inference techniques to highly dynamic data streams.

SR is an active research field that evolved from SP more than a decade ago. While SP targets answering different types of queries considering only data stream contents, SR infers new information by combining the data stream at hand with some background knowledge.

As stated in [52], "Increasingly, applications require real-time processing of heterogeneous data streams together with large background knowledge bases". These applications include scenarios from smart cities, smart industry, social networks analysis, etc. Hence, SR systems (a.k.a. Stream Reasoners) must be able to efficiently manage theoretically infinite data streams and perform complex reasoning tasks while handling large domain models that provide background knowledge and context.

Early SR solutions [15, 16] came from the fields of Data Stream Management Systems (DSMS) and Complex Event Processing [48] (CEP), which provided the foundations for continuous query application and data stream monitoring. However, both DSMS and CEP lacked to handle the increasing complexity arising from typical SR contexts, and more advanced and powerful knowledge representation and reasoning mechanisms became necessary. In this respect, several solutions have been proposed in the Semantic Web [17, 108, 83, 105, 18, 33] and Knowledge Representation and Reasoning (KRR) [122, 126, 29, 103, 23] fields.

In Semantic Web, the *RDF Stream Processing (RSP)* W3C community group proposed extensions to RDF and SPARQL for representing streaming data and their semantics. RDF [110] is a general framework for data representation on the Web; it unifies data from diverse sources in triples $< subject, predicate, object >$ where *subject*, and *object* are two data nodes and *predicate* is a predicate representing their relationship. SPARQL [112] is a query language for RDF that can join data from any source that expresses its knowledge as a directed labeled graph.

RSP solutions mainly consist of querying engines based on a continuous version of SPARQL; we mention here Streaming SPARQL (SPARQL_{Stream}) [33], Continuous SPARQL (C-SPARQL) [17], Event Processing SPARQL (EP-SPARQL) [13] and Continuous Query Evaluation over Linked Streams (CQELS) [108].

In KRR, Answer Set Programming (ASP) [30] is a well-established proposal that gained attention also outside of academia thanks to the availability of

robust and efficient implementations [72]. ASP is acknowledged as a particularly attractive basis for SR. Indeed, some significant steps forward in this direction have been taken to enable ASP to reason over data streams.

Typical ASP-based solutions integrate ASP engines with stream processing solutions and/or natively extend ASP language and semantics with the temporal concept and advanced constructs (e.g., window operators) useful to deal with data stream; we mention here the LARS framework [23] and its implementations [19, 24, 57, 116], StreamRule [99], C-ASP [105] and others [53, 67].

2.2.1 Handling (Infinite) Data Streams

Several methods exist to process data streams. Specifically, the paradigm shifts from "batch-like" approaches, which store data over a period of time and then analyze them in batch, towards timely and scalable streaming approaches, which analyze data as soon as the source produces them. The latter typically leverages on the natural temporal order of data stream elements.

Although streams are theoretically infinite, it is often neither viable nor necessary to process all the data produced by a device or a sensor at every time point. Sometimes only the most recent data are relevant; other times, a sequence of events in a limited time period is significant. Therefore, only a subset of data should be selected according to the appropriate criteria. In such a setting, window-based processing comes naturally, according to which a relevant part of the data stream is selected through the so-called window operators.

Window operators allow access to the stream by partitioning it into finite *snapshots*, namely windows, that stream processors or reasoners use to perform processing or reasoning tasks, respectively. So, a window contains the portion of the input streams representing the data needed to solve the task at the current time instant. Each item in a window is time-stamped.

Several types of window operators exist, initially defined in the fields of DSMS and CEP. As stated in [100], the most common are:

- **time-based** windows: contain input data arriving within a fixed amount of time
- **tuple-based** windows: contain the latest fixed number of input data
- **partition-based** windows: split the input data based on different attributes and apply tuple-based windows on each substream

2.2.2 SR Requirements and Open Challenges

A comprehensive survey by D. Dell'Aglio et al. published in 2017 [52] reported about the main requirements that a stream reasoner should meet to cope with real-world applications. We list below these requirements.

R1. **handle volume**: the typical SR scenarios are characterized by a large number of data sources. For example, Meta company reported an estimation of 2.88 *billion* daily active people on its platforms (i.e., Facebook, Instagram, Messenger, and/or WhatsApp) for the last quarter of 2022 [111].

R2. **handle velocity**: data sources produce a great amount of data at each moment that can easily exceed thousands of observations per minute; Facebook's users, as of April 2022, posted more than 1 million[1] of contents per minute, on average.

R3. **handle variety**: different data sources typically use different representation formats, and static data can also be subject to different data models.

R4. **cope with incompleteness**: generated data are not always complete, source of data might meet disparate problems. For example, sensors may suffer damage, wear and tear, run out of battery, or lose connection for a while.

R5. **cope with noise**: generated data can be subject to errors, outliers, imperfections, etc.

R6. **provide answers in a timely fashion**: answers should be generated within an amount of time, typically small (e.g., real-time, i.e., $< 1sec$, or near real-time, i.e., $< 1min$), that is compliant with the requirements specified by the application domain at hand.

R7. **support fine-grained information access**: different levels of information access should be provided, ranging from querying a group of sources to a single source.

R8. **integrate complex domain models**: domains might be characterized by very complex models that provide background knowledge and context.

[1] See `https://www.statista.com/statistics/195140/new-user-generated-content\-uploaded-by-users-per-minute/`, (url date: 10/01/2022)

R9. **capture what users want**: the system language should be expressive enough to support complex queries on both data streams and background knowledge.

Although the most advanced solutions address some stream reasoning challenges, and years have passed since the survey mentioned above was published [52], there are still open problems. Current solutions tackle the stream reasoning challenges from different perspectives and focus on various aspects of them.

Stream processing solutions proposed by DSMS and CEP were already able to process high-volume (R1) dynamic data (R2) in a timely fashion (R6) and to provide reactive fine-grained information access (R7) in the presence of noisy data (R5). However, DSMS and CEP systems faced limitations when dealing with heterogeneous data with complex domain models and the need to combine rich background knowledge. In those cases, users were required to put a great manual effort in developing complex networks of queries. Research in KRR and Semantic Web provides relevant input to cope with these problems.

Flexible information representation with RDF, querying language SPARQL for graph-based databases, and logic-based ontologies for knowledge representation and reasoning contributed to challenges R1, R3, R4, R7, and R8. DSMS, CEP, and Semantic Web allow users to use complex constructs in queries. However, sometimes more refined features are required (R9). The KRR community took some steps in this direction, extending logical formalisms to carry out SR tasks. The high expressiveness of logic formalisms allows to capture what users want (R9), handle incomplete knowledge (R4), and deal with complex domain models (R8), which can be efficiently supported thanks to efficient reasoning implementations. However, due to reasoning complexity, KRR solutions struggle to deal with high-volume (R1) dynamic data stream (R2) and, at the same time, comply with answer time requirements (R6). To overcome this issue, recently, researchers put much effort into designing advanced optimization techniques like incremental reasoning, parallel and distributed reasoning, caching strategies, answer approximation, and integration of inductive reasoning techniques [27, 109, 64, 6, 54, 106, 94, 57, 21].

Chapter 3

Apache Flink

This chapter gives an overview of the stream processor Apache Flink (*Flink*) [46, 85, 113], which is used as a main component of to efficiently handle data streams by the system presented in Part IV.

Flink is a distributed stream processor for processing both streaming and batch data with high throughput and low latency. It emerges as a strong contender in the stream processing landscape, providing several features that make it a compelling choice for building scalable and reliable streaming applications. In particular, it comes with a rich set of easy-to-use APIs that allow users to implement sophisticated stream processing applications. Moreover, its fault-tolerance mechanisms, low latency, unified processing model, advanced windowing, and event time handling, along with dynamic scaling and resource management, position *Flink* as a versatile and powerful stream processing framework.

Section 3.1 introduces the dataflow graph that represents the heart of how *Flink*-based applications work; Section 3.2 describes the principal stream operations that users can include in a *Flink* dataflow graph; Section 3.3 outlines the notions of time handled by *Flink* along with the *watermark* mechanism.

3.1 Dataflow Graph

Flink functioning consists in the execution of a dataflow program. A dataflow is a Directed Acyclic Graph (DAG) where nodes are *operators* and edges are data dependencies between operators. Incoming edges represent input streams to operators; outcoming edges represent output streams resulting from the appli-

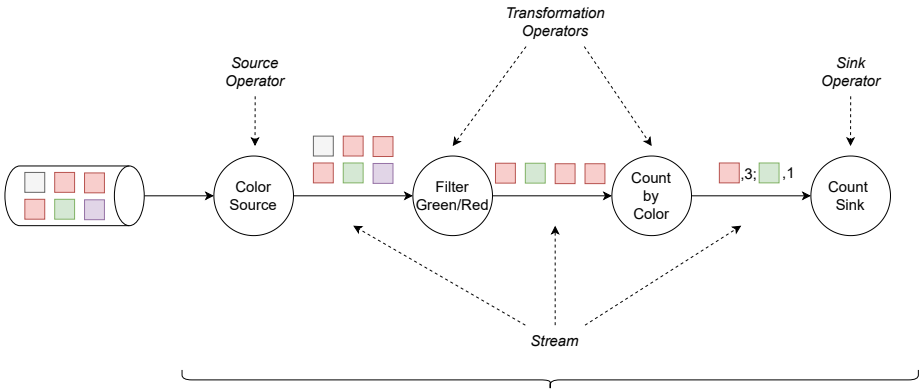


Figure 3.1: An intuitive *Flink* dataflow graph

cation of operators. An operator is a functional unit that consumes input events and performs some computation on them, thus producing an output stream. Events in the output stream can undergo further transformations through the next operators in the dataflow.

If an operator has no incoming stream, it is called *source*. Conversely, if an operator has no outcoming edges, it is called *sink*. Each dataflow graph must have one or more sources from which the data streams originate and at least one sink that can either emit the final output or persistently store it.

Example 3.1.1 Without going into the technical details, let us consider an intuitive example. "Given an input stream composed of colored boxes, count how many red and green boxes there are."

Figure 3.1 depicts the dataflow graph that extracts the required information. □

3.1.1 Parallel Dataflow Graph and Data Exchange Patterns

Given the parallel and distributed nature of *Flink*, each operator can have one or more independent *operator subtasks* (i.e., instances) executed in different threads and possibly on different machines. An operator subtask receives one or more *stream partitions* over which it applies the transformation at hand, producing partial results. The latter will then be properly reconstructed to form the final result.

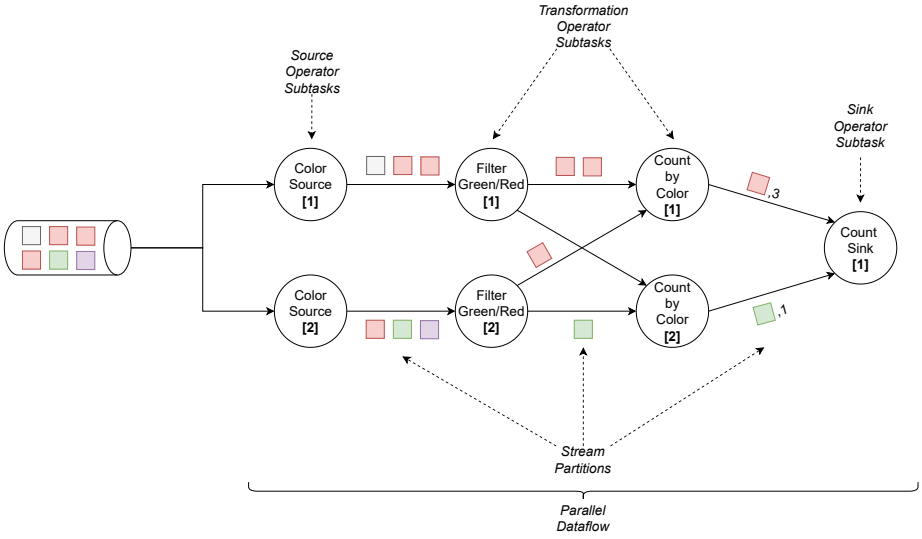


Figure 3.2: An intuitive parallel *Flink* dataflow graph

Given an operator, the number of the related operator subtasks represents its parallelism. Operators in a program may have different parallelism. *Flink* gives the possibility to set the desired maximum parallelism of each operator; otherwise, the engine determines the best parallelism according to the configuration of the machine running the program.

Example 3.1.2 Let us consider the dataflow introduced in Example 3.1.1. Figure 3.2 depicts a parallel version of it. Notice that operators *Color Source*, *Filter Green/Red*, and *Count by Color* have parallelism 2, while *Count Sink* has parallelism 1. □

Data are exchanged from the subtasks of one operator to the next ones following specific patterns. In general, *Flink* automatically selects which type of pattern to use between two operators according to their nature. However, users can define their own configurations if necessary. The available patterns are:

- *Forward*: is a one-to-one exchange pattern, from one operator subtask to the next one, that preserves the partition as well as the order of stream elements.
- *Broadcast*: is a redistributing exchange pattern where data are sent from one operator subtask to all the subtasks of the successive operators. Broad-

cast is the most expensive strategy as data are replicated, and network communication is required to send elements from one subtask to another.

- *Key-based*: is a redistributing exchange pattern that re-partitions stream elements according to their key. This strategy ensures that elements with the same key are always processed using the same subtask.

- *Rebalance*: is a redistributing exchange pattern designed to distribute the load across subtasks. In particular, elements are equally distributed from one subtask to the next ones in a random way.

In redistributing patterns, the order of elements is preserved only between each pair of sending and receiving subtasks. Therefore, when a subtask receives events from multiple sending subtasks, this type of exchange introduces non-determinism related to the order in which elements are received.

Example 3.1.3 Let us consider the parallel dataflow in Figure 3.2.

- Between the subtasks of *Color Source* and the subtasks of *Filter Green/Red*, there is a *forward* pattern. Specifically, substask 1 of the operator *Filter Green/Red* receive elements in the exact order they are produced by subtask 1 of the operator *Color Source*. The same happens between subtasks 2.

- Between the subtasks of *Filter Green/Red* and the subtasks of *Count by Color*, there is a *key-based* pattern. In particular, in our example, boxes have their color as key. According to this, they are uniformly redistributed among *Count by Color* subtasks, i.e., guaranteeing that boxes having the same color are always processed by the same subtask. In the considered dataflow, *Count by Color* subtask 1 processes red-colored boxes, while *Count by Color* subtask 2 processes green-colored ones. Furthermore, the order in which the subtasks receive boxes is not deterministic: *Count by Color* subtask 1 may receive boxes from *Filter Green/Red* subtask 2 before the ones produced by *Filter Green/Red* subtask 1, or vice versa. The same happens to *Count by Color* subtask 2.

- Between *Count by Color* subtasks and the *Count Sink* operator exists a *broadcast* pattern. As before, the order in which the sink receives aggregate elements is not deterministic.

□

3.2 Stream Operations

Flink provides several built-in operators as well as a set of APIs that enables users to develop custom ones. Such operators can be combined to form sophisticated dataflows.

There exist two categories of operators: *stateless* and *stateful*.

Stateless operators only look at one event at a time, do not store any information, and perform simple stream transformations. These characteristics make them easy to distribute and execute in parallel.

Stateful operators store information across multiple events in a *state*. Every time a new event arrives, the *state* can be accessed and updated to implement advanced transformations. *Flink* stores the application state locally in memory or an embedded database.

In the following, we present the principal *Flink* stream operations.

3.2.1 Source and Sink

Source and sink operators allow *Flink* to communicate with external systems.

The source operator consumes data from external sources and feeds them to *Flink* streams. The user can define how raw data must be transformed into stream elements.

The sink operator emits the output resulting from the execution of the dataflow. This output is in the form of stream that external systems can then consume.

Flink provides many predefined source/sink operators. The available source operators can read stream elements from files, directories, sockets, collections, and iterators. The available sink operator can publish output stream elements writing to files, stdout and stderr, and sockets.

3.2.2 Process Function

The ProcessFunction operator allows the implementation of low-level stream processing operations. This operator has complete access to events in the stream, manages an internal state, and can handle a timer. The latter can be used to register callbacks for future instants so that applications autonomously react to changes in time.

Flink allows users to define their own PocessFunction. Then, the ProcessFunction is invoked for each element in the input stream over which it applies

the user-defined transformations.

3.2.3 Windows

Windows are crucial to deal with infinite streams of events. Window operators allow for partitioning events into *buckets* of finite size, over which it is possible to perform computations. Events are assigned to buckets on the basis of data properties or the associated time.

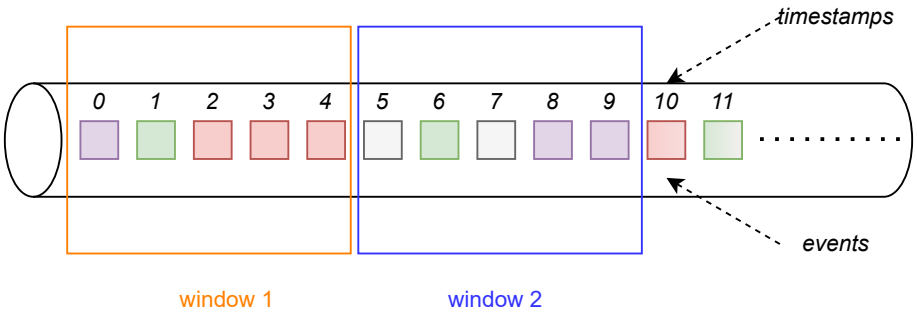
In general, window operators have a set of policies associated that defines when a new bucket must be created, how events are assigned to buckets, and when the computation on a bucket must be triggered. When a window operator is triggered, elements in the bucket are sent to an *evaluation function*, which in turn applies some computation on them and produces results. *Flink* allows users to define their own evaluation functions.

The most common window functions used to create buckets are:

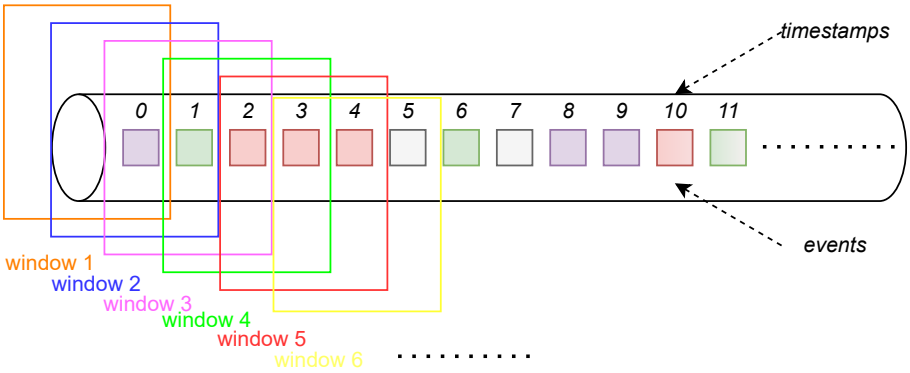
- *Tumbling* windows: events are assigned to buckets of fixed size that do not overlap, i.e., an element can be assigned only to one bucket. To define this window, the user must define just its size. For example, a tumbling window with a size of 5 minutes allows getting, every 5 minutes, a bucket that contains all events that arrived during the last 5 minutes (see Figure 3.3a).

- *Sliding* windows: events are assigned to buckets of fixed size that overlap, i.e., an element may be assigned to more than one bucket. To define this window, the user must define its size and slide, i.e., how frequently a sliding window must be started. For example, a sliding window with a size of 3 minutes and a slide of 1 minute allows getting, every 1 minute, a bucket that contains all events that arrived during the last 5 minutes (see Figure 3.3b).

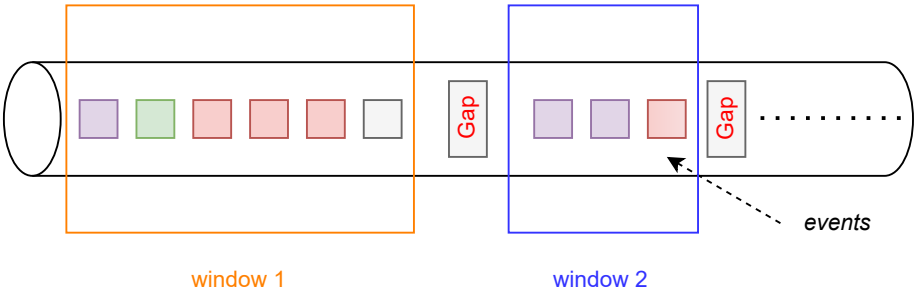
- *Session* windows: events are assigned to buckets on the basis of sessions of activity. Here, buckets do not overlap and do not have a fixed start and end time. In general, a bucket is complete when the window operator does not receive stream events for a given period of time, i.e., there is a gap in the arrival of events. As an example, consider the illustration in Figure 3.3c.



(a) An example of tumbling window



(b) An example of sliding window



(c) An example of session window

Figure 3.3: Example of windows

3.2.4 Window Join and Window CoGroup

Flink allows joining two streams based on windows through the *window join* or the *window CoGroup* operators.

The window join operator takes as input two streams and joins the contained events that fall within the same window and have the same key. The applied join corresponds to an *inner-join*: if an event from one stream does not join with any event from the other stream, this is not produced in output. Therefore, the join operation results in all the pairwise combinations of elements that have been joined, where each element is associated with the largest timestamp within the scope of the related window. The joined elements are then passed to a user-defined *joining function* that produces the output stream by emitting only results that comply with the user's specification. Figure 3.4 shows two examples of window joining based on tumbling (3.4a) and sliding (3.4b) windows.

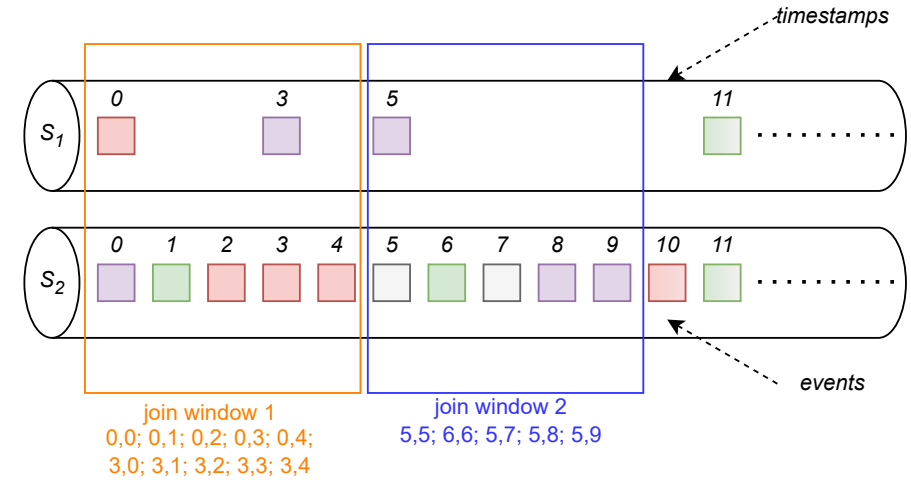
The window CoGroup operator works similarly but is more generic. It allows the implementation of different types of join, including the *outer-join*. The CoGroup operator takes as input two streams and groups all the events that fall within the same window and have the same key. A group is produced even if one of the two streams has no element for a considered key and window. Groups are then passed to a user-defined *CoGroup function* that produces the output stream according to the criteria specified by the user.

3.3 Time Semantics

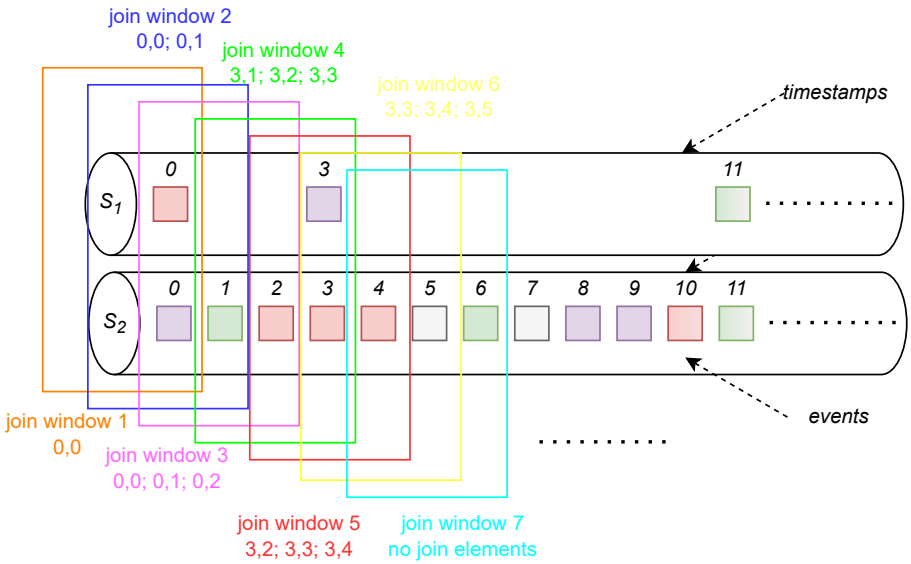
Flink allows to perform timely stream processing, i.e., a stream processing approach where time plays a crucial role in the computation; for instance, for creating windows, performing analysis based on the moment in which events are generated or received by the system, and so on. To do this, it provides three different notions of time:

- *Event time*: is the moment in which an event actually occurred, i.e., when the data source has registered it. This type of time is generally attached to the event outside of *Flink* and is extracted once the engine processes that event. In this case, time progresses on the bases of data arrival and not on the basis of any system clock time.

- *Processing time*: is the time in which an operator processes an event, and it corresponds to the system time of the machine that executes the



(a) An example of tumbling window join



(b) An example of sliding window join

Figure 3.4: Examples of window joining

operator.

- *Ingestion time*: is the moment in which an event arrives at *Flink*, and the engine internally records it.

The type of time to use depends on the streaming application that must be developed, as each type has different implications. The ones most commonly used are event time and processing time.

Event time allows obtaining deterministic, reproducible, and consistent results as computations are independent of both arrival time and order of events. However, event time processing leads to some latency when dealing with out-of-order events, as it is necessary to wait some time to avoid data loss. Furthermore, given that it is possible to wait only for a limited amount of time, deterministic results are not guaranteed in case of very late events. These issues do not occur if events are known to always arrive in order.

Processing time allows for achieving the highest performance and the lowest latency. However, deterministic or reproducible results are not feasible since computations depend on the speed at which events arrive at the system or flow from one operator to another.

3.3.1 Dealing With Event Time Processing - Watermarks

In the case of applications relying on event time, a way to define the progress of time is necessary. *Flink* makes use of the so-called *watermarks*. Watermarks are special events that flow within the stream, embedding a timestamp t and indicating that event time has reached the time t in that stream. This means that all the events with timestamp $t' < t$ have been received.

Watermarks are generated by source operators or by some operator immediately after. When a source operator features more than one subtask, each subtask generates its own watermark on the basis of events within the stream partition they received. In this case, the watermark models the time progress on that specific source instance. When generated, watermarks propagate within the dataflow from one operator to the next, notifying the progress of the time to each of them. If an operator consumes more than one input stream, it considers as its current time the smallest watermark among the ones received.

The watermark mechanism is crucial to trigger event time windows and operators that deal with out-of-order events. Once operators receive a watermark, they deduce that all events falling within a specific time interval have been re-

ceived and either trigger computation or order received events. For example, let us suppose that a window operator handles a window in progress and that this window ends at time t. That operator will close the window and "fire" the related computation only when it receives a watermark t' with $t < t'$. In such a way, it is sure that all the events falling in the considered window have been received.

Part II

Answer Set Programming

Chapter 4

Foundations

This chapter introduces the syntax and semantics of Answer Set Programming (ASP). The main core of this formalism consists of Disjunctive Datalog with non-monotonic negation under stable model semantics. During the last decades, the scientific community spent much effort extending the basic language of ASP to increase its expressiveness. Eventually, the main features have been formally collected in the *ASP-Core-2* standard [35], which is also used in official ASP competitions [41, 9, 38, 74, 75].

Sections 4.1 and 4.2 present a formal definition of the syntax and semantics of ASP.

4.1 Syntax

4.1.1 Terms

A **term** can be *simple*, *functional*, or *arithmetic*. In turn, a simple term is either a *constant* or a *variable*.

Constant Term. A *constant* term is either an *integer* (e.g., `1`, `231`, `100`), a *symbolic constant* or a *string constant*. Specifically, a symbolic constant is a string starting with a lowercase letter that contains only alphanumeric symbols and the symbol "_" (underscore) (e.g., `x`, `home`, `person_1`); a string constant is a quoted string (e.g., ``` ``The sky is cloudy'' ```, ``` ``231'' ```, ``` ``person 1'' ```).

Variable Term. A *variable* term is a non-empty string starting with an uppercase letter that contains only alphanumeric symbols and the symbol "_" (underscore) (e.g., `X`, `Home`, `Person_1`). A variable represented only by the symbol "_" is called *anonymous variable*, and it is intended to be a *fresh* i.e., a variable that does not appear elsewhere.

Functional Term. A *functional* term has the form $f(t_1, ..., t_n)$ where f is the function name, $t_1, ..., t_n$ are terms, and $n > 0$ is the arity of the function.

Examples of functional terms are: `f(1)`, `g(X)`, `foo("s",X)`.

Arithmetic Term. An *arithmetic* term has form $-(t)$ or $(t \diamond u)$ where t, u are terms and $\diamond \in \{"+", "-", "*", "/"\}$. In general, parentheses can be omitted; in this case, standard priorities apply.

Examples of arithmetic terms are: `X+Y`, `X*2`, `6/3`.

Ground Term. A term is *ground* if it does not contain any variable. For instance, the terms `1`, `231`, `100`, `x`, `home`, `person_1`, `''The sky is cloudy''`, `''231''`, `''person 1''`, `f(1)` and `6/3` introduced above are ground; the remaining ones are *non-ground*.

4.1.2 Atoms and Literals

Predicate Atom. A *predicate atom* has form $p(t_1, ..., t_n)$, where p is a *predicate name*, $t_1, ..., t_n$ are terms, and $n \geq 0$ is the arity of the predicate. If $n = 0$, the parenthesis are omitted, and a predicate atom $p()$ is represented by its predicate name p. The *predicate* of a predicate atom is generally identified as p/n where p is the predicate name and n is the arity.

Examples of predicate atoms are: `son(X,Y)`, `reLu(max(0,X))`, and `f(g(1))` whose predicates are respectively `son/2`, `reLu/1`, and `f/1`.

Classical Atom. A *classical atom* is either $-a$ or a, where a is a predicate atom and $-$ is the *strong negation* symbol.

Examples of classical atoms are: `son(X,Y)`, `reLu(max(0,X))` and `-f(g(1))`.

Built-in Atom. A *built-in* atom has form $t \triangleright u$ where t,u are terms and $\triangleright \in \{<, <=, =, <>, !=, >, >=\}$.

Examples of built-ins are: `X<=Y`, `X=Y*2`, and `1!=2`.

Naf-Literal. A *naf-literal* can be either a built-in atom or have the form a or *not* a where a is a classical atom and *not* is the *negation as failure* symbol.

Examples of naf-literals are: `X<=Y`, `not b(X)`, `son(X,Y)`.

Aggregate Atom. Aggregate atoms enable the application of aggregation operations over multi-sets of terms naturally and easily.

An *aggregate element* has form $t_1, ..., t\_m : l_1, ..., l_n$, where $t_1, ..., t\_m$ are terms, $l_1, ..., l_n$ are naf-literals, $n \geq 0$ and $m \geq 0$.

An *aggregate atom* has form $\#aggr\ \{e_1; ...; e_n\} \triangleright\ t$, where (i) $\#aggr \in \{\#count, \#sum, \#max, \#min\}$; (ii) $e_1, ..., e_n$ are aggregate elements; (iii) $\triangleright \in \{<, <=, =, <>, != , >, >=\}$; (iv) t is a term.

Example of aggregate atoms are:

- `#max {X:a(X)} > 10`,
- `#count {X,Y: son(X,Y)} = Z`
- `#min {1:a(1),2:b(2)} > 0`

Aggregate Literals. An *aggregate literal* is either a or *not* a, where a is an aggregate atom.

Example of aggregate literals are:

- **not** `#max {X:a(X)} > 10`
- `#count {X,Y: son(X,Y)} = Z`
- `#min {1:a(1),2:b(2)} > 0`.

In the following, we use *atom* and *literal* to refer to all types of atoms and literals, respectively. Moreover, a literal a (respectively *not* a) is said *positive* (respectively *negative*).

Ground Atom and Literal. An atom is *ground* if it contains no variable. Therefore, a literal is ground if its atom is ground.

4.1.3 Rules, Strong Constraints, Weak Constraints, and Queries

A **rule** r has the form:

$$a_1 \,|\, ... \,|\, a_k :- b_1, ...,\ b_n, not\ b_{n+1}, ...,\ not\ b_m.$$

where (i) $k >= 0$, $n >= 0$, and $m >= 0$; (ii) $a_1 |...| a_k$ is the head of r, denoted by $H(r)$, and it is a disjunction of atoms; (iii) $b_1, ..., b_n, not\ b_{n+1}, ..., not\ b_m$ is the body of r, denoted by $B(r)$, and it is a conjunction of literals. Specifically, $b_1, ..., b_n$ is the *positive* body of r, and it is denoted by $B(r)^+$; while $not\ b_{n+1}, ..., not\ b_m$ is the negative body of r, and it is denoted by $B(r)^-$.
When $m = 0$ and $k = 1$, the rule is a **fact**, while when $k = 0$, the rule is a **strong constraint**, defined as:

$$:- b_1, ..., b_n, not\ b_{n+1}, ..., not\ b_m.$$

Example 4.1.1 Examples of rules are:

r_1: `accept(X) | decline(X) :- proposal(X).`

r_2: `city(``Rome''):-.`

r_3: `:- you_jog, you_are_starving.`

In this example, r_2 is a fact, while r_3 is a strong constraint. □

In general, the sign `:-` is omitted for facts .

A **weak constraint** is a special type of rule, and it has the form:

$$:\sim\ b_1, ..., b_n, not\ b_{n+1}, ..., not\ b_m.\ [\ w@l\ ,\ t_1, ..., t_j\ ].$$

where (i) $n >= 0$, $m >= 0$, and $j >= 0$; (ii) $b_1, ..., b_n, not\ b_{n+1}, ..., not\ b_m$ are literals; (iii) $t_1, ..., t_j$, w, and l are terms; w and l represent, respectively, the *weight* and the *level* for the weak constraint. When $l = 0$, $@l$ can be omitted.

Example 4.1.2 An example of weak constraint is the following.

`:~ you_jog, you_are_starving.[1]` □

Intuitively, the main difference between strong and weak constraints regards their rigidity.

Strong constraints, also known as *integrity constraints*, are used to define conditions that *must not* be violated; given the strong constraint `:- B.`, its informal meaning is "It is not possible B is true" or "B must be false".

Weak constraints are used to define conditions that *should not* be violated; given the weak constraint `:~ B.`, its informal meaning is "It is preferable B is not true" or "B should be preferably false".

Example 4.1.3 Let us consider the strong constraint (r_3) in Example 4.1.1 and the weak constraint in Example 4.1.2, they have the same body, and they both are a type of constraint, but the first is meant to be stronger than the second.

Indeed the strong constraint can be read as "You *must not* jog while you are starving", while the weak constraint can be read as "You *should not* jog while you are starving".

□

In the following, we denote rules and constraints (weak or strong) simply as rules.

A rule is *ground* if it contains no variable; for instance, the rule r_3 in Example 4.1.1 and the one in Example 4.1.2 are ground rules.

A **query** has the form a? where a is a classical atom. If a is ground, then the query is also ground. For instance, `city(''Rome'')?` is a (ground) query.

4.1.4 What is an ASP program

A **program** is formed by a finite set of rules, strong constraints and weak constraints, and possibly a single query.

Example 4.1.4 (**Hamiltonian Path**.) Given a graph $G = \langle V, E \rangle$ where V is the set of nodes and E is the set of edges of G, and a starting node $n \in V$, find a path that visits each node $v \in V$ exactly once.

Suppose that the problem is defined through the following input predicates: (i) `node/1` defines the set of nodes of G; (ii) `edge/2` defines the set of edges of G; (iii) `start/1` defines the starting node.
The following ASP program models the problem at hand.

```
     %%%% Guess a path%%%%
r1   in_path(X, Y) | out_path(X, Y):- edge(X, Y).
     %%%% Check the guessed path%%%%
     % A node can be visited only once
r2   :- in_path(X, Y), in_path(X, Y1), Y <> Y1.
r3   :- in_path(X, Y), in_path(X1, Y), X <> X1.
     % The path is not cyclic
r4   :- in_path(X, Y), start(Y).
     % Compute the reachable nodes in the path
r5   reached(X):- start(X).
r6   reached(X):- reached(Y), in_path(Y, X).
     % All nodes must be reached
r7   :- node(X), not reached(X).
```

In the program above, the predicate `in_path/2` is meant to contain all the edges

forming a solution path, while `out_path/2` contains the edges not included in such a path. Moreover, the `reached/1` is a utility predicate indicating the set of nodes reachable along the path, as mentioned earlier. The rule r_1 guesses a path, i.e., given an edge, it is selected to be either part of the path or not. The rules in r_2-r_7 check the validity of the guessed path. Specifically, (i) the strong constraints r_2-r_3 ensure that a node is visited only via a single edge in the path, i.e., only once; (ii) the strong constraint r_4 verifies that the guessed path is not a cycle; (iii) the strong constraint in r_7 checks if all nodes are visited at least one time in the guessed path. To do so, the latter uses the utility predicate `reached/1`, whose values are derived by the rules in r_5-r_6. These rules recursively compute the set of reachable nodes in the guessed path.

Notice that lines starting with the `%` symbol are comments. □

4.2 Semantics

The semantics of ASP programs is based on the answer set semantics, which has been originally defined in [77] as a generalization of *stable models* semantics [76], and subsequently extended to aggregates [62, 63].

ASP can be used to declaratively encode computational problems by means of logic rules intended to define the features of the desired solution. Given an ASP program encoding a specific problem, its answer sets correspond one-to-one to the solution of that problem. As ASP is a fully declarative formalism, the order of rules and literals in rule bodies and heads does not impact the answer sets. It is worth noting that answer sets are only defined for ground programs. However, for every non-ground program, a semantically equivalent ground version can be defined through the *instantiation* (or *grounding*) process.

In the following, a formal overview of the ASP semantics is provided.

4.2.1 Herbrand Universe and Base

Let P be an ASP program.

The **Herbrand Universe** of P, denoted by U_P, is the set of all ground terms contractible from constants and function symbols appearing in P. If P contains no constant, an arbitrary constant c is added to U_P.

Example 4.2.1 Let us consider the ASP program P_1:

```
p(X) | q(X) :- u(X), not z(X).
u(c1).  z(c1).  u(c2).
```

its Herbrand Universe is U_{P_1}={c1, c2} □

The **Herbrand Base** of P, denoted by B_P, is the set of all ground classical atoms obtainable by combining predicates in P with terms from U_P as arguments.

Example 4.2.2 The Herbrand Base of P_1 from Example 4.2.1 is B_{P_1}={p(c1), p(c2), q(c1), q(c2), u(c1), u(c2), z(c1), z(c2)} □

A *substitution* for a rule $r \in P$ is a mapping from the set of variables of r to the set U_P of ground terms. Therefore, a *ground instance* of a rule r is obtained by applying a substitution to r. The **full instantiation** (grounding) of P, denoted by *Ground(P)*, corresponds to the set of all ground instances of rules appearing in P over U_P.

Example 4.2.3 Let us consider again P_1 from Example 4.2.1, its ground version P_1^G=*Ground*(P_1) consists in:

```
p(c1) | q(c1) :- u(c1), not z(c1).
p(c2) | q(c2) :- u(c2), not z(c2).
u(c1).  z(c1).  u(c2).
```

□

4.2.2 Interpretations, Models, and Answer Sets

Definition 4.2.1 (Herbrand Interpretation). A *(Herbrand) interpretation* I for P is a *consistent* subset of B_P. I is consistent if, for each predicate atom $a \in B_P$, $\{a, -a\} \not\subseteq I$ holds.

Example 4.2.4 Examples of interpretations for P_1 from Example 4.2.1 are:

- I_1={u(c1), u(c2), z(c1)};
- I_2={p(c2), u(c1), u(c2), z(c1)};
- I_3={p(c2), q(c2), u(c1), u(c2), z(c1)}.

□

Let I and r be an interpretation and a ground rule, respectively. A positive literal l is true w.r.t. I if $l \in I$; it is false otherwise. A negative literal *not* l is true w.r.t. I if $l \notin I$; it is false otherwise. The head of r is true w.r.t. I if at least one classical atom $a \in H(r)$ is true w.r.t I. The body of r is true w.r.t. I if all the literals in $B(r)$ are true w.r.t. I. Finally, r is *satisfied* w.r.t I if either its head is true w.r.t I or its body is false w.r.t. I.

Example 4.2.5 Let us consider the ground program P_1^G in Example 4.2.3 and the interpretation I_2 in Example 4.2.4:

- the literals {`p(c2), u(c1), u(c2), z(c1),` **`not`** `z(c2)`} are true w.r.t. I_2
- the literals {`p(c1), q(c1), q(c2),` **`not`** `z(c1), z(c2)`} are false w.r.t. I_2
- I_2 satisfies all the rules in P_1^G. The first rule is satisfied because the body literal **`not`** `z(c2)` is false w.r.t. I_2; the second one is satisfied because the head atom `p(c2)` is true w.r.t. I_2. Moreover, also all the facts in the third line are satisfied as {`u(c1), z(c1), u(c2)`}$\in I_2$.

□

Definition 4.2.2 (Model). A *model* for P is an interpretation M for P such that every rule $r \in Ground(P)$ is satisfied w.r.t. M.

Definition 4.2.3 (Minimal Model). A model M is *minimal* if there is no model N for P such that $N \subseteq M$.

The set of all minimal models for P is denoted by $MM(P)$.

Example 4.2.6 Let us consider P_1 from Example 4.2.1 along with its ground version P_1^G from Example 4.2.3 and the interpretations from Example 4.2.4: *(i)* I_1 is not a model for P_1 as I_1 does not satisfy the first two rules of P_1^G; *(ii)* I_2 is a model for P_1 according to the third point in Example 4.2.5; similarly, *(iii)* I_3 is a model for P_1 as it satisfies all its rules. Additionally, I_2 is a minimal model for P_1, while I_3 is not given that $I_2 \subseteq I_3$. □

Let P^G be a ground ASP program. The *reduct* of P^G, w.r.t. an interpretation I, is the positive program $P^I \subseteq P^G$ obtained from P^G by deleting all the rules whose body is false w.r.t. I. This definition of reduct, proposed in [62],

is a simplification of the *Gelfond-Lifschitz* reduct [77]. However, they are fully equivalent for the definition of answer sets [62].

Definition 4.2.4 (Answer Set). Given an interpretation I for a program P. I is an *answer set* for P if $I \in MM(P^I)$ (i.e., if I is a minimal model for the reduct P^I of P).

The set of all answer sets for P is denoted by $AS(P)$ and corresponds to its semantics.

Example 4.2.7 Let us consider P_1^G in Example 4.2.3 and the interpretation I_2 in Example 4.2.4. The reduct $P_1^{I_2}$ is:

```
p(c2) | q(c2) :- u(c2), not z(c2).
u(c1).  z(c1).  u(c2).
```

I_2 is a minimal model for $P_1^{I_2}$ since there exists no proper subset of I_2 that is also a model for $P_1^{I_2}$. Therefore, I_2 is an answer set for $P_1^{I_2}$ and consequently also for P_1^G and its non-ground version P_1 in Example 4.2.1. Moreover, given that P_1 is a *disjunctive* program, I_2 is not the unique answer set for it. Indeed such a type of program admits more than one answer set. In our example the interpretation I_4={`q(c2), u(c1), u(c2), z(c1)`} is also an answer set for $P_1^{I_2}$. □

Chapter 5

ASP Computation and Implementations

Since the founding of Answer Set Programming as a powerful paradigm for knowledge representation and reasoning, the research community has spent much effort designing and implementing advanced techniques to support its semantics efficiently. Throughout the years, reliable and high-performance implementations [73, 38] have been released.

This chapter provides an overview of the existent evaluation strategies and state-of-the-art ASP systems in Section 5.1, focusing on the $\mathcal{I}$-*DLV* system and its incremental version $\mathcal{I}^2$-*DLV* in Section 5.2.

5.1 Main Evaluation Strategies

ASP systems typically adopt the so-called "ground & solve" evaluation strategy. Intuitively, this approach computes the semantics of ASP programs by splitting the evaluation process into two subsequent phases: *instantiation* (or *grounding*) and *solving* (or *answer set search*). The first phase is carried out by a module named *grounder* (or *instantiator*) and consists in generating a propositional theory semantically equivalent to the input program. The subsequent second phase takes as input the program instantiation, previously computed by the grounder, and generates its answer sets by applying proper propositional techniques. The module responsible for the solving phase is named *solver*. State-of-the-art systems based on this approach have been released either as monolithic,

i.e., internally embedding grounding and solving modules, or as combinations of standalone grounders and solvers. Among the most commonly used ASP systems, there are the *I-DLV* [37] and Gringo [68] standalone grounders, the Wasp [11] and Clasp [69] standalone solvers, and the DLV2 [10] and Clingo [70] monolithic systems. DLV2 is the new version of the DLV system and integrates *I-DLV* and Wasp; Clingo instead integrates Gringo and Clasp.

Besides the "ground & solve approach", there exists the so-called "lazy grounding" [123, 124] approach. Systems adopting this method do not split the evaluation process into two steps; grounding and solving are interleaved instead, and rules are grounded on-demand during solving. The aim is to overcome the so-called grounding bottleneck that occurs on problems characterized by a huge instantiation, which makes the traditional approach unsuitable in practice [50]. Systems that implement lazy grounding techniques are Gasp [104], Asperix [93, 92], Omiga [51], and Alpha [124].

5.2 The ASP system I-DLV

I-DLV [37] is the new grounder of DLV [97, 10]. It is based on the solid theoretical foundations of its predecessor and integrates all the techniques that proved to be effective in the old grounder. In particular, it uses the semi-naïve approach [61, 121] with a bottom-up evaluation strategy but properly extended with several optimization techniques.

In general, *I-DLV* is not simply limited to producing the full instantiation of an input program; on the contrary, it implements advanced intelligent techniques to optimize both the grounding process and the resulting instantiation. Also, *I-DLV* is a complete and efficient deductive database system; in the case of programs with a unique solution, i.e., non-disjunctive and stratified w.r.t. negation programs, it provides the complete unique answer set.

The language supported by the system is fully compliant with the standard ASP-Core-2 [35]. Additionally, it provides a set of features that make DLV suitable in industrial domains [95, 4] and Semantic Web contexts [8, 12]; indeed, it can perform powerful ontology-based reasoning. For instance, it allows the users to express SPARQL queries in logic programs, therefore, to reason on RDF knowledge bases and to query SPARQL endpoints accordingly.

5.2.1 Incremental Evaluation for Stream Reasoning

With an eye to stream reasoning scenarios and, in general, applications where *multi-shot reasoning* [24, 71] is required, *I-DLV* has recently been released in an incremental version, named $\mathcal{I}^2$-*DLV* (or *I-DLV*-incr) [34, 86]. Multi-shot reasoning consists in repeated executions of reasoning tasks over varying inputs.

As anticipated, generating a propositional theory for a non-ground input program can be quite expensive in terms of time and memory [50]. Therefore, in contexts where reasoning over data streams or via multiple subsequent "shots" is required, starting the grounding process from scratch at each reasoning request is unworkable — also, considering that such reasoning is typically asked to be done within a short amount of time (e.g., real-time or near real-time).

Aiming to overcome this problem, $\mathcal{I}^2$-*DLV* incrementally evaluates pure ASP programs using an incremental grounding approach. Specifically, given a non-ground input program, the system maintains a semantically equivalent ground program, called *overgrounded*, that monotonically grows over time. This overgrounded program is stored in memory and is updated at each shot by adding only new ground rules derived from the submitted inputs. In such a way, it becomes more and more general while moving from one shot to the next. This approach allows reusing the same propositional theory to reason on different sets of inputs, possibly avoiding the grounding process at all. Eventually, subsequent intermediate updates to the ground program are considerably less time-consuming as new reasoning shots are performed.

The $\mathcal{I}^2$-*DLV* implementation behaves as a process staying alive and providing a service-oriented behavior, waiting for requests. In particular, given a fixed input program, it remains "listening" for input facts. Every time new facts arrive, it computes a ground program by properly updating the one resulting from previous shots. When the end-user asks for a reasoning result, $\mathcal{I}^2$-*DLV* returns the answer sets of the shot at hand. In this case, just like *I-DLV*, if the input program is a normal and stratified w.r.t. negation ASP program, $\mathcal{I}^2$-*DLV* directly computes the full semantics (i.e., returns the unique answer set); otherwise, it interfaces with an external ASP solver.

Chapter 6

Rewriting Techniques for Optimizing ASP Programs

As we will see in Part IV, the herein presented system is an ASP-based stream reasoner leveraging on the incremental version of the *I-DLV* grounder for performing complex reasoning tasks. In SR contexts, it is crucial to perform such tasks in a small amount of time. Therefore, as part of this project, we worked on studying and designing optimization techniques for *I-DLV*. In particular, we focused on automatic rewriting techniques for ASP programs.

Typically, the same computational problem can be encoded by means of many different ASP programs which are semantically equivalent; however, real ASP systems may perform very differently when evaluating each of them. This is because the structural properties of a logic program can make easier or harder the computation of its answer sets; furthermore, specific aspects and features of the ASP system at hand might significantly impact performance, such as the adopted algorithms and optimizations. As a result, some expert knowledge can be required in order to select the "best" encoding; this, in a certain sense, conflicts with the declarative nature of ASP that, ideally, should free users from the burden of computational issues. For this reason, ASP systems tend to be endowed with pre-processing means aiming at making performance less encoding-dependent.

Section 6.1 presents preliminary work about a Machine Learning (ML) strategy devised for automatically optimizing ASP programs. Such a strategy relies on an adaptation of hypergraph tree-decompositions techniques for rewriting

rules and inductively estimates whether a rule decomposition is convenient or not. Section 6.2 discusses the possibility of extending the application of this approach to other rewriting techniques inspired by the *projection* rewriting, used in the database field.

6.1 Tree Decompositions for Rewriting ASP Rules

An ASP rule r can be represented as a *hypergraph* [26] $HG(r)$ and be decomposed according to a tree decomposition $TD(r)$ of $HG(r)$ into a set $RD(r)$ of new rules that are equivalent to the original one, yet typically shorter. Such technique is adopted in *lpopt* [26] to rewrite a program before it is fed to an ASP system. In more details, a (undirected) hypergraph is a generalization of a (undirected) graph in which an edge can join two or more vertices. $HG(r)$ has a hyperedge for each literal l in the body and in the head of r containing all variables in l.

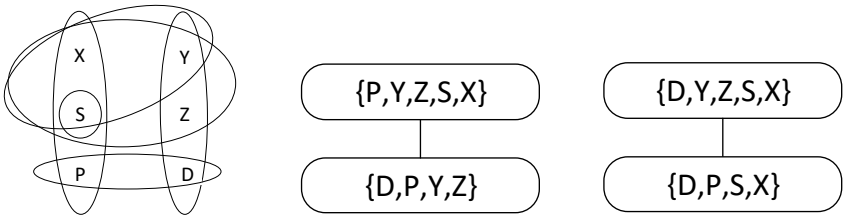
A tree decomposition of a hypergraph $HG(r)$ is a tuple $(TD(r), \chi)$, where $TD(r) = \langle V(TD(r)), E(TD(r))\rangle$ is a tree and $\chi : V(TD(r)) \rightarrow 2^{V(HG(r))}$ is a function mapping a set of vertices $\chi(t) \subseteq V(HG(r))$ to each vertex t of the decomposition tree $TD(r)$, such that for each $e \in E(HG(r))$ there is a node $t \in V(TD(r))$ such that $e \subseteq \chi(t)$, and for each $v \in V(HG(r))$ the set $\{t \in V(TD(r))|v \in \chi(t)\}$ is connected in $TD(r)$. Intuitively, a tree decomposition $TD(r)$ of $HG(r)$ is a tree such that each vertex is associated to a *bag*, i.e., a set of nodes of $HG(r)$, and such that each hyperedge of $HG(r)$ is covered by some bag, and for each node of $HG(r)$ all vertices of $TD(r)$ whose bag contains it induce a connected subtree of $TD(r)$.

A tree decomposition $TD(r)$ induces a set of rules that rewrites r, called *rule decomposition*, and denoted by $RD(r)$ containing a fresh rule for each vertex v of $TD(r)$. Roughly, each body literal l in r, such that the set of variables in l is contained in v, is added to the body of the rule generated for v. Moreover, some rules may be generated to guarantee safety. In general, more than one decomposition is possible for each rule.

Example 6.1.1 Let us consider the rule:

r_1
```
p(X,Y,Z,S) :- s(S), a(X,Y,S-1), c(D,Y,Z), f(X,P,S-1), P >= D.
```

Figure 6.1 depicts the conversion of rule r_1 into the hypergraph $HG(r_1)$ and two possible decompositions $TD_1(r_1)$ and $TD_2(r_1)$. According to $TD_1(r_1)$, r_1 can be decomposed into the set of rules $RD_1(r_1)$:



(a) $HG(r_1)$ (b) $TD_1(r_1)$ (c) $TD_2(r_1)$

Figure 6.1: Decomposing a rule.

```
r1  p(X,Y,Z,S) :- s(S), a(X,Y,S-1), f(X,P,S-1), fresh_pred_1(P,Y,Z).
r2  fresh_pred_1(P,Y,Z) :- c(D,Y,Z), P>=D, fresh_pred_2(P).
r3  fresh_pred_2(P) :- s(S), f(_,P,S-1).
```

The rule r_2 has the same head of r_1 and in the body all literals covering the first node of $TD_1(r_1)$ with variables $\{P, Y, Z, S, X\}$; r_3 has in the head the fresh predicate `fresh_pred_1` that links it to r_2. The body of r_2 contains the literals having as variables $\{D, P, Y, Z\}$ appearing in the other node of $TD_1(r_1)$. The rule r_4 ensures safety in r_3: `fresh_pred_2(P)` is added in the body of r_3 and to the head of r_4, whose body has a set of literals coming from r_1 and covers P. Intuitively, a different rewriting could be obtained by differently handling safety: e.g., by adding the literals `s(S)` and `f(_,P,S-1)` to the body of r_3 and avoiding to introduce r_4. Similarly, according to $TD_2(r_1)$, r_1 can be rewritten into $RD_2(r_1)$:

```
r1  p(X,Y,Z,S) :- a(X,Y,S-1),c(D,Y,Z), fresh_pred_1(D,S,X).
r2  fresh_pred_1(D,S,X) :- s(S),f(X,P,S-1),P>=D,fresh_pred_2(D).
r3  fresh_pred_2(D) :- c(D,_,_).
```

□

6.1.1 ML-guided Tree Decomposition Rewriting

I-DLV [37] employs a heuristic-guided tree decomposition algorithm [45] aiming at optimizing the instantiation process. Roughly, *I-DLV* possibly decomposes input rules into multiple smaller ones according to the technique sketched in Section 6.1 on the basis of a formula that estimates the cost of joining body literals. This deductive heuristic relies on internally computed grounding statistics such as the number of generated atoms or arguments' selectivities (cfr. [45]). The

technique proved to be beneficial in both grounding and solving performance, permitting the mitigation of the so-called grounding bottleneck issue and the instantiation of programs that cannot be grounded otherwise.

In this thesis, in contrast with the proposal above, we present an approach relying on an inductive heuristic. We still aim at properly deciding whether decomposing rules might improve grounding performance, but via an ML heuristic that considers only "static" information on the non-ground structure of the input program. In particular, this new heuristic is based on a predictive model purposefully designed and trained able to classify each input rule as: "better to decompose", "better not to decompose" or "indifferent" (i.e., applying or not decompositions has almost the same effect on performance).

Work-flow

As anticipated, we opted for classification. We recall that in such a task input data consists of a set of records known as *examples*; each record (or *example*) is a tuple of the form (X, y), where $X = \{x_1, \cdots, x_n\}$ for $n > 0$ is a set of attributes and y is a class label, namely, the so-called *target attribute*. Classification lies in learning a target function $f : X \mapsto y$ that maps a set of attributes X to a class label y. In our context, for a rule r, X is the features described in Section 6.1.1 computed on r, while y can be either "better to decompose", "better not to decompose" or "indifferent".

We chose a set of features by focusing only on easily computable non-ground structural properties and domain information. The set of examples has been created by selecting all decomposable rules from a large set of widely spread ASP benchmarks. We obtained an example from each selected rule by computing features on it and associating a class label. The association has been done by taking into account the *I-DLV* grounding times when the considered rule is decomposed or not (see Section 6.1.1). Once we achieved a consistent data set, an Artificial Neural Network (ANN) has been designed and trained to build the classifier. Eventually, we experimentally evaluated the quality of the resulting model.

Feature Selection

In total, we devised 19 features involving information about non-ground properties, tree decomposition structures, and input facts. We herein report the 6 features that showed a higher correlation with class labels. The features,

reported below, are defined via the following notations.

Let P be an ASP program, we indicate as $Facts^I(P)$ the set of input facts of P. We denote as $EDB(P)$ the set of all predicates in P defined only by facts and as $IDB(P)$ the remaining ones. Let r be a rule, $H(r)$ is the set of atoms in the *head* of r, as $B(r)$ the set of literals in the *body* of r, and as $RD(r)$ a possible tree decomposition of r. We denote as $\#sharedVars(r)$ the number of joins in $B(r)$, i.e., the number of times each pair of atoms in the rule shares the same variable. Let p be a predicate, we indicate as $arity(p)$ the arity of p.

The most relevant features are:

- $|Facts^I(P)|$: the number of input facts;
- $|B(r)|$: the body length;
- $|RD(r)|$: the number of rules in which r can be decomposed;
- $\frac{\sum_{r_i \in RD(r)} |B(r_i)|}{|RD(r)|}$: the body average length of the rules in the rule decomposition of r;
- $\sum_{r_i \in RD(r)} \#sharedVars(r_i)$: the total number of joins in the rule decomposition of r;
- $\frac{\sum_{p_i \in IDB(P)} arity(p_i)}{|IDB(P)|}$: the average arity of IDB predicates.

Data Set Creation

We collected decomposable rules from benchmarks of the 3rd [40], the 4th [9], and the 6th [74] official ASP competitions, as well as from the grounding-intensive 2-QBF domain [26]. Since rules can often not be decomposed, as tree decomposition is applicable only on the basis of intrinsic structural rule properties, we tried to enrich the set of examples. To this end, we performed a pre-processing step in which we applied techniques inspired by *unfolding* [119] to encodings. The aim was to obtain additional rules with longer bodies that are more likely to be decomposable.

At this point, we computed the features over the so collected decomposable rules, and then we properly associated a class label to each one. Labels have been assigned by first generating for each considered rule an *example program* consisting of its non-decomposed version along with the set of required input

facts, and then by measuring $\mathcal{I}$-*DLV* times when asked to instantiate the example program when decomposition is disabled and forcibly activated. If the difference in instantiation times of two versions is lower than 10%, the features set is labeled as "indifferent"; otherwise, the assigned label is either "decomp" or "do-not-decomp", depending on which version led to the lowest grounding time.

The resulting data set is formed by 3852 examples: 3106 have been labeled as "indifferent", 417 as "do-not-decomp" and the remaining 329 as "decomp", with class distributions of $80,63\%$, $10,83\%$, and $8,54\%$, respectively. We observe unbalanced distributions that, in general, could make learning about minority classes tougher and, in turn, worsen the model quality. In Subsection 6.1.1, we describe our countermeasures to mitigate this issue.

Model Design

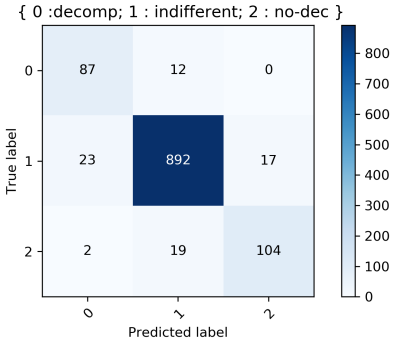
To build the classifier, we adopted a *Multilayer Perceptrons (MLPs) Neural Network* [79]. MLPs are commonly used for such tasks, as they often permit high accuracy by "learning" complex implicit relationships within data.

In classifications, the learning process of a neural network is guided by a *loss function* that, in general, determines the quality of the model prediction. More specifically, during the training phase, the internal network configuration undergoes a series of transformations aiming at minimizing the loss function. Since it is computed at each training step, the loss value clearly indicates how well the current configuration performs the task for which the network was designed; in our case, it is a multi-class classification.

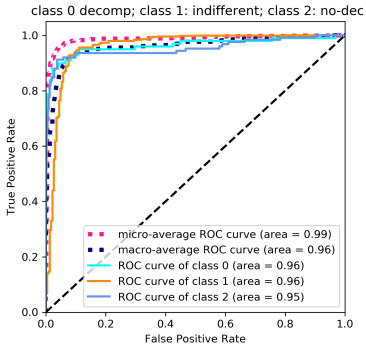
Given that we wanted to maintain the natural data set configuration, we adopted a cost-sensitive learning method to deal with the imbalance issue. Instead of modifying the distribution of training data, such an approach assigns different weights to classes in the loss function so that minority class misclassifications are more penalized; this commonly used adjustment allows to deal with the imbalance directly into the learning algorithm itself. In particular, we implemented as a custom loss function the α-*balanced focal loss* for multi-classes classification [98]. Experimentally, this focal loss variant has proved to be suitable for classifying better the examples belonging to minority classes. We used the adaptive learning rate optimization algorithm *Adam* as an optimizer for the loss function minimization process.

Class	Precision	Recall	F1-*score*
"indifferent"	0.97	0.96	0.96
"do-not-decomp"	0.86	0.83	0.85
"decomp"	0.78	0.88	0.82
Cumulative			
Avg method	Precision	Recall	F1-*score*
macro	0.87	0.89	0.88
weighted	0.94	0.94	0.94

(a) Performance measures



(b) Confusion Matrix



(c) AUC-ROC curves

Figure 6.2: Model validation results.

Model Evaluation

The model has been trained over 300 epochs. As convention, 70% of the examples have been used as *training set* and the remaining 30% as *test set*. In this splitting, we carefully maintained the original class distributions.

Since we are dealing with an unbalanced data set, accuracy cannot be considered an appropriate performance measure. In this metric, the impact of the classification errors of minority cases is reduced by the proper classification of majority cases. Thus, the quality of the model has been assessed by means of the *F1-score*, defined as $F1 = 2 \times (Precision \times Recall)/(Precision + Recall)$, which provides us with more information about the effectiveness of the model in correctly predicting the instances belonging to minority classes [28]. The $F1$ value is high when both *Precision* and *Recall* are high; the former indicates

the proportion of cases classified as relevant that are actually relevant, while the latter measures the proportion of relevant cases classified among all the relevant ones.

The Receiver Operator Characteristic (*ROC*) curve examines the model's capability of detecting *True Positives (TP)* instances and compares it with *False Positive (FP)* predictions. The ROC curve plots the *TP rate* against the *FP rate* on the vertical axis and on the horizontal axis, respectively. The larger the Area Under the Curve (*AUC*) is, the higher the quality of the model (i.e., AUC=1.0).

Figure 6.2a reports *precision*, *recall*, and *F1* scores, both class by class and aggregated using macro and weighted as average methods. Despite unbalance, the model achieves good performance also when dealing with minority classes. Figure 6.2b shows the *confusion matrix* summarizing distributions of model predictions. The *AUC - ROC* plot in Figure 6.2c evidences the model's capability of distinguishing among classes: an AUC very close to 1.0 suggests that, in most cases, the model correctly identifies when it is convenient to decompose a rule or not.

Experimental Evaluation

Next, we present the experimental activity carried out to analyze the impact of the proposed inductive heuristic on *I-DLV* performance. We compared four versions of *I-DLV*:

- *I-DLV* $_{never}$ with decomposition disabled;
- *I-DLV* $_{always}$ with decomposition always enabled;
- *I-DLV* $_{deduct}$ with decomposition applied according to the internal deductive heuristic;
- *I-DLV* $_{induct}$ with decomposition guided by the inductive model.

The latter version has been externally implemented thanks to the capability of *I-DLV* to customize its grounding process via annotations [37]. More in detail, the model communicates with *I-DLV* via an external module which, given an encoding, for each rule r: first, invokes the model to determine whether r has to be decomposed, and then accordingly, annotates r as to decompose or not. Eventually, these annotated encodings are fed to *I-DLV*. We report in Table 6.1 the results on the 4th competition. For each version, the table details the

Table 6.1: 4th Competition - number of grounded instances and average grounding times in seconds.

Problem		$\mathcal{I}$-DLV $_{never}$		$\mathcal{I}$-DLV $_{always}$		$\mathcal{I}$-DLV $_{deduct}$		$\mathcal{I}$-DLV $_{induct}$	
Name	# instances	#	Time	#	Time	#	Time	#	Time
Permutation Pattern Matching	30	28	58.69	30	63.90	30	64.83	30	63.89
Valves Location	30	30	4.11	30	4.09	30	4.10	30	4.11
Connected Still Life	10	10	0.10	10	0.10	10	0.10	10	0.10
Graceful Graphs	30	30	0.38	30	0.38	30	0.39	30	0.38
Bottle Filling Problem	30	30	4.07	30	4.31	30	4.33	30	4.33
Nomystery	**30**	**30**	**34.86**	**30**	**18.92**	**30**	**35.66**	**30**	**18.84**
Sokoban	30	30	2.68	30	2.76	30	2.77	30	2.85
Ricochet Robots	30	30	0.27	30	0.31	30	0.31	30	0.31
Crossing Minimization	30	30	0.10	30	0.10	30	0.10	30	0.10
Reachability	30	30	102,46	30	101,65	30	102,72	30	102,72
Strategic Companies	30	30	0,21	30	0,22	30	0,21	30	0,41
Solitaire	27	27	0.13	27	0.19	27	0.20	27	0.20
Weighted-Sequence Problem	**30**	**30**	**2.83**	**30**	**9.50**	**30**	**2.90**	**30**	**11.16**
Stable Marriage	30	30	27.72	30	2.54	30	2.54	30	2.47
Incremental Scheduling	30	12	295.62	21	219.97	21	222.00	21	214.59
Qualitative Spatial Reasoning	30	30	2.84	30	2.84	30	2.83	30	2.85
Chemical Classification	30	30	88.49	30	88.50	30	88.67	30	87.28
Abstract Dialectical Frameworks	30	30	0.13	30	0.13	30	0.13	30	0.13
Visit-all	30	30	0.13	30	0.13	30	0.13	30	0.14
Complex Optimization	29	29	34.89	29	34.23	29	35.15	29	35.51
Knight Tour with Holes	30	20	177.76	20	173.03	20	174.90	20	180.98
Maximal Clique	30	30	0.32	30	0.32	30	0.33	30	0.31
Labyrinth	**30**	**30**	**1.47**	**30**	**1.39**	**30**	**1.48**	**30**	**0.71**
Minimal Diagnosis	30	30	2.54	30	2.22	30	2.57	30	2.90
Hanoi Tower	30	30	0.22	30	0.23	30	0.23	30	0.23
Graph Colouring	30	30	0.10	30	0.10	30	0.10	30	0.10
Total	696	666	22.46	677	22.39	677	23.07	677	22.47

number of grounded instances and the average instantiation times per problem. Experiments have been performed on a NUMA machine equipped with two 2.8 GHz AMD Opteron 6320 and 128 GiB of main memory, running Linux Ubuntu 14.04.4; the memory limit has been set to 15 GiB and time limit to 600 seconds per instance.

The proposed method generally behaves consistently with the well-established deductive method used in $\mathcal{I}$-*DLV*. On the one side, we observe cases such as *Nomystery* in which the inductive heuristic identifies benefits of applying decompositions: $\mathcal{I}$-*DLV* $_{deduct}$ rewrites the input encoding in a way similar to $\mathcal{I}$-*DLV* $_{never}$, while the $\mathcal{I}$-*DLV* $_{induct}$ rewriting is comparable to that performed by $\mathcal{I}$-*DLV* $_{always}$. An improvement is also gained in problem *Labyrinth*: $\mathcal{I}$-*DLV* $_{induct}$ is faster than the other versions. On the other side, there is the case of the *Weighted-Sequence Problem* in which the proposed method causes a significant worsening because it does not recognize some decompositions as convenient. In the remaining problems, $\mathcal{I}$-*DLV* $_{induct}$ performance is in line with others.

In summary, the performance of the experimental ML-based approach is comparable to the ones obtained with well-assessed methods.

6.2 Projecting Variables for Rewriting ASP Rules

In addition to the aforementioned rewriting technique, other approaches based on variable *projection* have proved to be effective in optimizing ASP programs. Such techniques are inspired by the database field and consist in projecting variables in rule bodies, on the basis of different criteria. In [82], the authors proposed two different types of projection, named α and β. Similarly to the decomposition, these projections rewrite a rule by splitting it into a set of new rules with smaller body sizes and preserving the original semantics. Roughly, given a rule r and let V be a subset of the variables appearing in $B(r)$ but not in $H(r)$:

- the α-projection moves all literals in $B(r)$ that have in common some variable with V in a new rule r' called *replacement rule*. If the body of r' is not safe, let V^u be the set of the unsafe variables in $B(r')$. The α-projection moves in $B(r')$ all the literals of $B(r)$ that share some variable with V^u. This process is repeated until r' becomes safe. The head of r' is an atom a' having a fresh predicate and all the variables appearing in $B(r')$ but not in V as terms. Then, the atom a' is added to the body of r in place of the moved literals. If possible, the projection process is repeated iteratively. In particular, each time, the α-projection is applied to replacement rules until no further α-projection are possible.

- the β-projection selects from $B(r)$ every literal such that all its variables are contained in V, and uses them to create the body of the replacement rule r'. If the body of r' is not safe, let V^u be the set of the unsafe variables in $B(r')$. The β-projection adds to $B(r')$ each atom in $B(r) \setminus B(r')$ that shares all its variables with the set $V \cup V^u$. This process is repeated until r' becomes safe, each time, by adding the unsafe variables to V^u. The head of r' is obtained as in the case of the α-projection. Eventually, $H(r')$ is inserted within the body of r in place of all the literals that the α-projection would select when projecting $V \cup V^u$. Also in this case, the β-projection is iteratively applied on replacement rules until no further projections are possible.

Given the similar aim of tree-decompositions and projections, both of the aforementioned heuristics can be straightforwardly applied to projections. We are working on integrating also heuristic-driven projection techniques. Moreover, we are also working on additional projection and rewriting approaches.

Part III

State-of-the-art ASP-based SR Solutions

Chapter 7

LARS

LARS [23], Logic-based Framework for Analyzing Reasoning over Streams, is a comprehensive extension of ASP to reason over data streams. It extends ASP for streams by providing flexible expiration control (it introduces window operators) and temporal modalities (it introduces controls to handle temporal information). These extensions allow reasoning "over streams" of data, i.e., pose questions about the validity of formulas at certain time points, during a certain "window", or combine the above cases, e.g., by asking if, during a certain time period, a given formula was valid (entailed by the program) at some point in time. In general, LARS defines a logic for modeling stream-related axioms.

LARS models the notion of *stream* as a sequence of time-annotated formulas. In addition to the usual boolean operators (and, or, implies, not), the language introduces operators such as $\diamond$, $\Box$, and @. These operators check if a formula holds respectively at some time in the past, at every time in the past, and at a specific time instant. In general, LARS formulas can be evaluated against the whole stream, or their scope can be limited through the usage of the window operator.

This chapter presents basic notions of LARS, originally defined in [23]. Where appropriate, we provide only informal descriptions. Therefore, we refer the reader to the original paper for additional details. Section 7.1 defines the notions of stream and windows. Section 7.2 provide a brief overview of how LARS extends the syntax and semantics of ASP for SR.

7.1 Notions of Streams and Windows

In LARS, a stream associates a time point with a set of ground atoms. In the following, $\mathcal{A}$, $\mathcal{G}$, and $\mathcal{U}$ denote a finite set of atoms, ground atoms, and *time variables*, respectively.

Atoms can be either *extensional* or *intensional*. The former collects the set of ground atoms whose predicate is extensional, i.e., that does not appear in any rule head. The latter collects the set of ground atoms whose predicate is intensional, i.e., that appears in some rule head; notably, they represent inferred information.

Definition 7.1.1 (LARS Stream [23]). Let T be a closed nonempty interval in $\mathbb{N}$ and $\upsilon : \mathbb{N} \mapsto 2^{\mathcal{G}}$ 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*.

If a stream S only contains extensional atoms, it is said *data stream*. Along with streams, it is possible to have a finite set $\mathcal{G}^{\varepsilon}_{\mathcal{B}}$ of extensional ground atoms representing the *background knowledge* (i.e., static data that do not change over time).

Besides the stream concept, the one of a *window* constitutes the core of LARS formalism. A window is typically a continuous small selection of later streaming data and can be seen as a substream of the stream over which it is created.

Definition 7.1.2 (Window [23]). 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 $S' \subseteq S$.

A *window function* allows the creation of a window over a stream

Definition 7.1.3 (Window Function [23]). Any (computable) function w that returns, given a stream $S = (T, \upsilon)$, and a time point $t \in T$, a window S' of S, is called a window function.

LARS defines four types of window functions that create windows based on different criteria: time-based, tuple-based, partition-based, and filter windows. The first two are presumably the most used and are informally reported below [23, 21, 24, 57, 19].

- *Time-based window.* Given an integer n representing the window size, a time-based window returns all atoms appearing in the last n time points.

- *Tuple-based window.* Given an integer n representing the window size, a tuple-based window returns the most recent n atoms.

7.2 Extending ASP Syntax and Semantics for Stream Reasoning

7.2.1 LARS Programs

Definition 7.2.1 (Formulas [23]). 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:

$$\alpha ::= a \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \alpha \rightarrow \alpha \mid \diamond\alpha \mid \Box\alpha \mid @_t\alpha \mid \boxplus^w \alpha \mid \triangleright\alpha$$

$\boxplus^w$ is called *window operator*, and $\boxplus^w\alpha$ means that the formula α will be evaluated on the window returned by w, which is computed on the current stream S and at the current time point t. A formula is ground if it contains no variable; the set of ground formulas is denoted by $\mathcal{F}_\mathcal{G}$.

A LARS program P is a set of rules of the form $\alpha \leftarrow \beta_1, \cdots, \beta_n.$ where α is a formula and $\beta_1, \cdots, \beta_n$ is a conjunction of formulas. Given a rule r, α is the head of r, and it is denoted by $H(r)$, $\beta_1, \cdots, \beta_n$ represents the body of r, and is denoted by $\beta(r)$.

7.2.2 Entailment Relation

Given a ground formula, it can be satisfied at a time point in a *structure* defined as a triple $M = \langle S, W, B\rangle$ where $S = (T, \upsilon)$ is a stream, called *interpretation stream*, W is a set of windows, and $B \subseteq \mathcal{G}^\varepsilon_\mathcal{B}$ is a set of facts representing the background data of M. Let a stream $S' = (T', \upsilon')$ such that $S' \subseteq S$, the entailment relation, denoted by $\Vdash$, between (M, S', t) and formulas is defined next [23].

Let $a \in \mathcal{G}$ be an atom, $\alpha, \beta \in \mathcal{F}_\mathcal{G}$ be ground formulas, $w \in W$ and $t \in T'$ then:

- $M, S', t \Vdash a$ iff $a \in \upsilon'(t)$ or $a \in B$,
- $M, S', t \Vdash \neg\alpha$ iff $M, S', t \nVdash \alpha$,

- $M, S', t \Vdash \alpha \wedge \beta$ iff $M, S', t \Vdash \alpha$ and $M, S', t \Vdash \beta$,
- $M, S', t \Vdash \alpha \vee \beta$ iff $M, S', t \Vdash \alpha$ or $M, S', t \Vdash \beta$,
- $M, S', t \Vdash \alpha \rightarrow \beta$ iff $M, S', t \nVdash \alpha$ or $M, S', t \Vdash \beta$,
- $M, S', t \Vdash \diamond\alpha$ iff $M, S', t' \Vdash \alpha$ for some $t' \in T'$,
- $M, S', t \Vdash \Box\alpha$ iff $M, S', t' \Vdash \alpha$ for all $t' \in T'$,
- $M, S', t \Vdash @_{t'}\alpha$ iff $M, S', t' \Vdash \alpha$ and $t' \in T'$,
- $M, S', t \Vdash \boxplus^{w}\alpha$ iff $M, S'', t \Vdash \alpha$ where $S'' = w(S', t)$,
- $M, S', t \Vdash \triangleright\alpha$ iff $(M, S, t) \Vdash \alpha$

If $M, S', t \Vdash \alpha$, we say that (M, S', t) *entails* α. If (M, S, t) *entails* α, we write $M, t \models \alpha$, and say that M *is a model of* α *at time* t.

7.2.3 Models and Answer Stream

Let P be a LARS program and D be a data stream.

An *interpretation stream for* D is a stream $I = (T, \upsilon)$ such that $D \subseteq I$, and for all $t \in T$ the set $\upsilon(t) \setminus D$ contains only intensional atoms.

Given a structure $M = \langle I, W, B\rangle$, called *interpretation (for D)*, M is a model of P for D at time t, denoted by $M, t \models P$, if for each rule $r \in P$, $M, t \models \beta(r) \rightarrow H(r)$. Moreover, M is a *minimal model* if there is no model $M' = \langle S', W, B\rangle$ of P for D at time t such that $S' = (T, \upsilon')$ and $S' \subseteq S$. Finally, the *reduct* of P with respect to M at time t is defined by $P^{M,t} = \{r \in P | M, t \models \beta(r)\}$, i.e., all program rules whose body is satisfied.

Definition 7.2.2 (Answer Stream [23]). 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.

Chapter 8

ASP-based SR Systems

This chapter overviews state-of-the-art approaches based on ASP for Stream Reasoning. Section 8.1 presents recent LARS-based implementations. Sections 8.2 and 8.3 describe the systems *StreamRule* and *C-ASP*, respectively.

8.1 Implementations of Subsets of LARS

LARS allows defining formulas over the whole stream or limiting the reasoning effect to the window content. The language is very expressive, but it comes at a price of high-complexity reasoning. Consequently, as demonstrated in [23], the full language which is intractable in practice.

For practical realization, it is necessary to identify practical subsets of LARS for tractable reasoning. Over the years, several experimental prototypes have been proposed that support smaller yet practically relevant fragments of LARS.

We provide an overview of the existent LARS fragments and implementations next.

Plain Lars

Plain Lars is a tractable fragment of LARS defined in [19] and then implemented in several prototypes under different further restrictions. This fragment relies on the following limitations: *(i)* windows can only be time-based and tuple-based; *(ii)* windows in rule heads are not allowed; *(iii)* nested windows in rule bodies are not supported;

Specifically, it considers formulas according to the following grammar:

$$\alpha ::= a \mid @_t a \mid \boxplus^w @_t a \mid \boxplus^w \diamond a \mid \boxplus^w \Box a \tag{8.1}$$

where a is an atom, t is a time point, and w can be either a time-based window or a tuple-based window.

Therefore a plain LARS program is a set of rules of form $\alpha \leftarrow \beta_1, \cdots, \beta_n$ where α is a formula of form a or $@_t a$, and each β_i is a formula as in 8.1 that can also be *negative*, i.e., preceded by the *not* symbol.

8.1.1 Laser

Laser [19] implements the fragment *Plain LARS* with a further restriction on accepted programs. In particular, it admits only *positive* programs, i.e., without negation in rule bodies, that ensure a unique model.

The system implementation runs on the ASP solver Clingo [70], and it comes with an evaluation process that enables an incremental model update. This process avoids the recomputation of ground formulas that hold at multiple time points by marking them with temporal annotations. Temporal annotations consist of two markers representing the lower and upper bound of the time interval where formulas are guaranteed to hold. The designed algorithm, while evaluating a program over an input stream along the time, efficiently updates and propagates these annotations. As a result, an incremental model computation is obtained, avoiding many unnecessary recomputations.

In [19], the authors discuss how it is possible to extend this approach to support negation in front of atoms. However, they only consider stratified negation to avoid the generation of multiple models, and the related implementation is still in a prototypical version.

8.1.2 Ticker

Ticker [24] implements the Plain Lars fragment and comes with two evaluation modes: static and incremental. In both cases, programs must respect a limitation on their variable; specifically, any variable appearing in the scope of a window atom (i.e., formulas where $\boxplus^w$ is present) must be "guarded" by some standard atom (i.e., formulas where $\boxplus^w$ is not present). In other words, each variable in a window atom must also appear in a standard one. Such variables are grounded upfront in a so-called *pre-grounding phase*.

In the static approach, at each time point, first, the input plain LARS program P is translated in a static ASP one P', then P' is evaluated using the ASP system Clingo [70] as back-end. The static ASP encoding is valid only for the time point in which it is generated; therefore, the evaluation is carried out from scratch at each time point. If multiple models exist, the system always returns the first one. For this reason, this mode is most suitable for programs having a single model.

In the incremental approach, the static translation from plain LARS to ASP is made incremental through an annotation that provides rules with expiration controls. This mechanism is based on the notion of *tick streams*. A tick stream captures the evolution of a stream along the time in terms of time and received input. Specifically, each element of a tick stream (called *tick*) register either an increase in the time dimension (i.e., a transition from a time point t to the time point t) or in the number of received tuples (i.e., a transition from n tuples to $n + 1$ tuples). Therefore, each derived ground rule is annotated with a tick, representing how long that rule is valid. This incremental encoding is evaluated using a truth maintenance system (TMS) [55] under the ASP semantics, which continuously maintains the related model. This mode is suitable for programs with multiple models; in this case, the system computes and maintains one randomly chosen model.

8.1.3 Distributed-SR

Distributed-SR [57] implements the plain LARS fragment and adopts a distributed reasoning approach by efficiently splitting computations among different solvers. Each solver evaluates a part of the input program over a data stream and communicates its results with the others through data streams. Distributed-SR can use LARS-based (Ticker [24]) or ASP-based solvers (Clingo [70]).

In order to pursue such distribution, this implementation requires an additional limitation on the class of accepted programs. In particular, it requires plain LARS programs to be *stream-stratified*. Stream stratification, defined in [21], allows splitting logic programs into subprograms that can be evaluated separately and sequentially, similarly to classical stratification [14]. In this case, strata are searched w.r.t. dependencies among window atoms instead of w.r.t. negation. In a stream-stratified program, the computation of every window in a stratum only depends on the computation of windows in preceding strata. This stratification lets the system assign each subprogram to a different solver. It

is worth noting that such a stratification is only possible if recursion through window atoms is not present. Eventually, as Ticker, Distributed-SR requires that variables appearing in window atoms are guarded by standard ones and performs the pre-grounding phase.

With the aim of unburdening communications among solvers, the system uses an *interval-based* version of the LARS semantic. Specifically, interval-based streams assign each ground atom a with a set of nonempty time intervals that intuitively groups the set of time points where a was true in a compact way. Solvers communicate using this type of stream, which is only updated when something changes (rather than at each time point).

8.1.4 BigSR

BigSR [116] implements a very small fragment of plain LARS where: only time-based window atoms and $\diamond$ operator are supported, rule heads can only contain simple atoms (i.e., without $@_t$ operator), and negation is not allowed. Despite the low expressiveness, this system is the first step towards applying distributed computing techniques on logic-based stream reasoning addressed via modern Big Data frameworks. Indeed, the primary purpose of this research work was to demonstrate that solutions of this type are possible in practice, although not trivial to realize. It is an ontology-based stream reasoner that accepts RDF input streams and is released in two versions built on top of state-of-the-art stream processors: Spark Streaming [125] and Apache Flink [47] respectively.

BigSr introduces an additional practical limitation on the basis of the stream processor at hand. On the one hand, the version leveraging on Spark Streaming accepts only programs with a global window, i.e., all the window atoms must share the same window operator. On the other hand, the version relying on Apache Flink accepts non-recursive programs with global windows at rule scope, i.e., all formulas in the body of a rule must share the same window operator. Recursion is disabled in this version due to the usage of the multi-core/distributed nature of Apache Flink, which makes the handling of synchronization of clocks, task progress, and window trigger mechanisms complicated.

In both cases, the system implements internal reasoning algorithms to compute the semantics of accepted programs natively; hence it does not use any ASP system as a back-end.

8.2 StreamRule

StreamRule [99] system enables expressive reasoning for the Semantic Web using ASP and builds on a two-layer approach. On the one hand, a CQELS-based RSP is used to filter raw data that are relevant for complex reasoning tasks. On the other hand, the reactive ASP system OClingo [67, 66] is used as a reasoner. These two components communicate through a middle layer consisting of a Linked Sensor Middleware (LMS) [91]. Specifically, first, the RSP executes the event filtering CQELS queries specified by end-users, then the middle layer transforms the query results in ASP input facts and forwards them to the ASP system along with the logic program at hand, and finally, the ASP system evaluates such a program on the provided facts and emits the solutions in the form of answer set. In general, StreamRule abstracts from time in the sense that it considers computation at a certain moment: the reasoner is triggered whenever new input arrives, the portion of input data is considered together with the program, and there is no explicit timestamp.

In the original proposal of StreamRule, the ASP grounding is a bottleneck. In fact, the system does not scale when results are returned slower than the new input window arrives. Therefore, in [78], the authors proposed a method to dynamically adapt the tuple-based window size to changing streaming rates by implementing a processing pipeline bi-directional or adaptive. Later, the authors of [107] proposed an *Extended* StreamRule that addresses the scalability challenge via input-driven parallelization. The goal is to partition the input and compute the answer set in parallel based on input dependency analysis. The latter consists in analyzing the structure of the input program and constructing an "extended dependency graph". The extended dependency graph relates not only to the intensional database but also to the input data and represents how data items in a window relate to each other. It allows partitioning input data in a window that can be processed with the whole program by parallel reasoners.

8.3 C-ASP

C-ASP [105] system is designed to apply complex reasoning in the Semantic Web by combining RSP and ASP-based reasoning in one single framework. It natively extends the core ASP language and semantics with constructs that handle RDF background knowledge bases and streams, and performs continuous reasoning over them. For instance, it defines RDF streams input data type,

window operators, and streaming operators.

The computation workflow of C-ASP is the following. Firstly, users express their continuous reasoning requests using the C-ASP language and submit such requests to the system. Subsequently, C-ASP evaluates the continuous reasoning requests (using its own reasoning model) over one or more RDF input streams along with static RDF background knowledge. Finally, the system emits the outputs in the form of streams.

The reasoning model of C-ASP relies on a combination of windowing-based models originally developed in the field of DSMS with models adopted in rule-based systems of CEP for producing new events from timestamped data.

8.4 Summary

This chapter provided an overview of state-of-the-art ASP-based stream reasoners, including LARS-based implementations, such as Laser, Ticker, Distributed-SR, and BigSR, as well as StreamRule and C-ASP. Laser, Ticker, and Distributed-SR have been developed within the scope of KRR and handle streams in an ASP-like format, while BigSR, StreamRule, and C-ASP are ontology-based proposals coming from the Semantic Web and handle streams in RDF. These solutions offer different features and approaches leveraging ASP for enabling expressive and continuous reasoning over data streams.

LARS extends ASP by providing flexible window operators and temporal modalities like “always”, “sometimes”, and “at” modifiers. It provides high expressiveness but is intractable in practice. Therefore, the discussed LARS-based implementations support tractable fragments of the full language. Among the others, Ticker supports the larger fragment, known as Plain LARS (see Section 8.1), while Distributed-SR, Laser, and BigSr introduce further restrictions on the accepted programs. Distributed-SR requires Plain LARS programs to be stream stratified, thus prohibiting recursion through windows. Laser restricts the class of accepted Plain LARS programs to the positive ones, thus prohibiting negation in rule bodies. Eventually, BigSr supports the smaller fragment of LARS, implementing only time-based windows and the $\diamond$ operator, and prohibiting @−atoms in the head of rules.

C-ASP represents a further proposal in which ASP language and semantics is extended for Stream Reasoning. It particular, it provides constructs designed for handling and reasoning on RDF background knowledge bases and streams.

As an alternative approach, StreamRule does not directly extend ASP, but it uses the functionalities of the ASP system o-clingo for enabling expressive reasoning in Semantic Web applications.

From an implementation perspective, these solutions also differ in terms of reasoning and evaluation mechanisms. Laser employs the ASP solver Clingo as back-end and utilizes an incremental model update process that avoids unnecessary recomputations by means of temporal annotations. Ticker supports static and incremental evaluation modes. The static approach translates plain LARS programs into standard ASP programs and evaluates them from scratch at each time point. The incremental approach uses an annotation mechanism based on tick streams and a truth maintenance system for efficient evaluation. Distributed-SR adopts a distributed reasoning approach that efficiently splits the computation among different solvers, and employs interval-based streams for implementing communication among them. Furthermore, it uses Ticker or Clingo as back-ends. BigSR uses the state-of-the-art stream processors Spark Streaming and Apache Flink to handle data streams, and implements its own evaluation algorithm for computing the semantics of the accepted programs. StreamRule adopts a two-layer approach, integrating a CQELS-based RSP for data filtering and o-clingo as the reasoning engine. It uses a Linked Sensor Middleware (LMS) as communication layer between the RSP and ASP system and addresses scalability issues through adaptive window sizing and input parallelization techniques. C-ASP provides a single framework that combines RSP and ASP-based reasoning. It performs continuous reasoning over RDF input streams, leveraging windowing-based models and CEP techniques.

ASP-based stream reasoners offer diverse approaches to stream reasoning. However, each reasoner has its own limitations, including restrictions on window types, negation support, recursion, and scalability. To date, a solution capable of fully covering all the SR requirements [52] is still missing. This thesis aims to present a novel ASP-based SR solution, designed with the aim of meeting SR demand and overcoming main limitations of currently available solutions.

Part IV

I-DLV-sr: an ASP-based Stream Reasoner

Chapter 9

Input Language

The input language of I-DLV-sr consists of a fragment of ASP enriched with a set of constructs for reasoning over streams.

The design and definition of the basic input language is a joint work, published in the *Theory and Practice of Logic Programming* journal [42]. Afterwards, the language has been extended with new constructs [44].

I-DLV-sr accepts ASP programs that are non-disjunctive and stratified w.r.t. negation, and is compliant with ASP-Core-2. Indeed, all term and literal types of ASP-Core-2 (see Sections 4.1.1 and 4.1.2) are fully supported. Additionally, this fragment has been enriched with: (*i*) a set of *streaming literals*, enabling the possibility to evaluate logic formulae on streaming data; (*ii*) some temporal constructs like *temporary* and *trigger* rules, allowing to infer new knowledge temporarily or with a predefined frequency; (*iii*) the ***@now*** term allowing to explicitly reason about time.

This chapter illustrates the complete I-DLV-sr input language: Section 9.1 intuitively introduces basic concepts and intuitions on the syntax and semantics of the system, which are then formally defined in sections 9.2 and 9.3, respectively.

9.1 Basic Concepts and Intuitions

When designing a language for SR, several concepts need to be considered and included in the formalism definition. These concepts include events and streams of events, their representation, constructs for accessing the stream and reasoning

about time, and rules for expressing sophisticated inference over streams. All of these elements have been taken into account in the definition of the I-DLV-sr language and semantics. We will identify as event an information that is true at a specific time point and it is essentially represented as a predicate atom. A stream is a sequence of sets events (i.e., predicate atoms) $\Sigma = \langle S_0, \ldots, S_n \rangle$, where each set is associated with a *time point*, i.e., a specific moment in time. Therefore, if a predicate atom a is contained in the set S_i, it means that a is true at the i-th time point.

Example 9.1.1 An example of stream is depicted in Figure 9.1

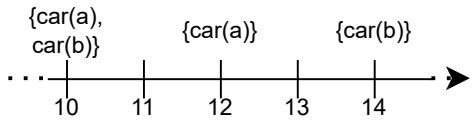


Figure 9.1: An Example of Stream

in which, e.g., the predicate atom `car(a)` is (an event) true in the time points 10 and 12. □

I-DLV-sr allows access to the stream through *streaming literals* which allow to evaluate formulae on an input stream. A streaming literal consists of a *streaming atom* possibly preceded by the symbol `not` intended to be the negation as failure. The general form of a streaming atom is `a` **`op`** $\{\mathtt{d}, \cdots, \mathtt{d_k}\}$, where `a` is a predicate atom representing the event of interest, **`op`** is an *operator* representing the formula to evaluate, and $\{\mathtt{d}, \cdots, \mathtt{d_k}\}$ is a *lookup set* indicating the set of time points of the input stream to be selected when evaluating the atom at hand.

To provide users with the ability to model practical SR scenarios, we have implemented the operators most commonly used in this context. **`op`** can be one of: `always`, `at least`, `at most`, `count`. Intuitively, given an event of interest a, `always`, `at least`, and `at most` respectively check if a is *always true*, *true at least a given number of times*, or *true at most a given number of times*, while `count` counts how many times a is true. All these operators are applied over the sets of events associated with the time points specified by the lookup set.

Example 9.1.2 An example of streaming literal is `car(a)` **`always in`** {*0, 2*}. Let us assume we want to evaluate this streaming literal at time point $n = 12$ of the input stream in Figure 9.1. The `always` operator must be evaluated over the sets of events received at time points $n - 0$ and $n - 2$, i.e., 12 and 10. In

this case, the considered streaming literal holds because the event of interest `car(a)` is always true within the considered sets. Conversely, if the lookup set were $\{1, 2\}$, the streaming literal would not hold because `car(a)` is false at time point 11. □

With I-DLV-sr, users can implement sophisticated SR tasks by defining declarative programs consisting of logic rules whose bodies can contain the streaming literals. This allows capturing complex events present within an input stream and inferring new information from them.

Example 9.1.3 Let us consider a traffic monitoring scenario. Here, a possible question one can ask when analyzing the traffic flow is "*How many cars have passed in the last 4 time points?*". The I-DLV-sr program answering this question follows.

```
r1  car_passing(C,N) :- car(C) count N in {0,1,2,3}.
r2  tot(T) :- #sum{N,C: car_passing(C,N)}=T.
```

In this program, rule r_1 counts the number of passes for each car, while r_2 uses an ASP aggregate to compute the total sum. □

Note that, besides streaming literals, I-DLV-sr also supports built-in atoms and aggregate literals as defined in the ASP-Core-2 standard [35]. Currently, the only restriction is that aggregate elements cannot feature (non-degenerate) streaming literals.

When evaluating a I-DLV-sr program over an input stream Σ at a given time point n, the output is a *streaming model*. This corresponds to the set of ground (i.e., variable free) predicate atoms that are considered true at n according to Σ. Therefore, the streaming model includes the predicate atoms received as input at n along with the ones derived by evaluating the program rules over Σ at n.

Example 9.1.4 Let us suppose we want to evaluate the program presented in Example 9.1.3 over the stream reported in Figure 9.1 at the time point 14.

In order to evaluate the streaming literal contained within the body of r_1, we have to consider the true predicate atoms received at the time points from 11 $(14-3)$ to 14 $(14-0)$. According to the functioning of the `count` operator, for each car `C`, it is counted how many times `C` has been observed in these time points. Then, such a value is assigned to N. In our case, we have two possible cars `a` and `b` both of them occurring once: `car(a)` at 12, while `car(b)` at 14. Therefore, from r_1 we derive the predicate atoms `car_passing(a,1)` and

`car_passing(b,1)`. These new predicate atoms are considered in the evaluation of the aggregate literal in the body of r_2, which applies the sum aggregation function over the second term of predicate `car_passing/2`. The result of this operation is 2, thus deriving the predicate atom `tot(2)`.

In the end, the streaming model of this program computed on Σ at the time point 14 is the set of ground predicate atoms {`car(b)`, `car_passing(a,1)`, `car_passing(b,1)`, `tot(2)`}. □

9.2 Syntax

In the following, we use the definitions of *terms* and *atoms* introduced in Section 4.1.

Let P, V, and C be finite sets containing respectively *predicate names*, *variables*, and *constants* where $V \cap C = \emptyset$. We denote as G the set of all ground *predicate atoms* obtainable by combining predicate names in P and constants in C as arguments.

9.2.1 Streaming Atoms and Literals

A *lookup set*, denoted by D, is a non-empty set of numbers $\{d_1, \ldots, d_m\} \subset \mathbb{N}$.

Streaming Atoms. A *streaming atom* has one of the following forms:

(1) a **always in** $\{d_1, \ldots, d_m\}$

(2) a **at least** t **in** $\{d_1, \ldots, d_m\}$

(3) a **at most** t **in** $\{d_1, \ldots, d_m\}$

(4) a **count** t **in** $\{d_1, \ldots, d_m\}$

where *(i)* a is a *predicate atom*; *(ii)* t is a *term*, named *counting term*, such that $t \in \{C \cap \mathbb{N}^+\} \cup V$; *(iii)* $\{d_1, \ldots, d_m\}$ is a lookup set.

Example 9.2.1 For each possible form, an example of streaming atom is reported next, along with an indication of its eventual counting term and lookup set.

□

`b(1) always in {2,3}`	counting term: *none*; lookup set: {2,3}.
`color("Red") at least X in {0,1,2,3,4}`	counting term: X; lookup set: {0,1,2,3,4}.
`warning(X) at most 1 in {5,10}`	counting term: 1; lookup set: {5,10}.
`action_1 count 2 in {2,5,8}`	counting term: 2; lookup set {2,5,8}.

Streaming Literals. A *streaming literal* is either α or *not* α, where α is a streaming atom, and the *not* symbol denotes the *negation as failure*. A streaming literal is *negative* if the *not* symbol is present, otherwise it is *positive*.

Given a streaming literal l, we denoted as *base(l)* the predicate atom appearing in l. Let L be a set of streaming literals, we use *preds*(L) and *base(L)* to denote respectively the set of predicates and the set of predicate atoms appearing in L.

Example 9.2.2 The table below reports some examples of streaming literals. For each example, we specify the type of streaming literal shown (i.e., positive or negative) and the related predicate atom. □

l_1: `b(1) always in {2,3}` l_2: **not** `b(1) always in {2,3}`	l_1: is a positive streaming literal; l_2: is a negative streaming literal; *base*(l_1) = *base*(l_2) = `b(1)`.
l_3: `color("Red") at least X in {0,1}` l_4: **not** `color("Red") at least X in {0,1}`	l_3: is a positive streaming literal; l_4: is a negative streaming literal; *base*(l_3) = *base*(l_4) = `color("Red")`.
l_5: `warning(X) at most 1 in {5,10}` l_6: **not** `warning(X) at most 1 in {5,10}`	l_5: is a positive streaming literal; l_6: is a negative streaming literal; *base*(l_5) = *base*(l_6) = `warning(high)`.
l_7: `action_1 count 2 in {2,5,8}` l_8: **not** `action_1 count 2 in {2,5,8}`	l_7: is a positive streaming literal; l_8: is a negative streaming literal; *base*(l_7) = *base*(l_8) = `action_1`.

Ground Streaming Atoms and Literals. A streaming atom is *ground* if both its predicate atoms and counting term are ground. Therefore, a streaming literal is ground if its streaming atom is ground.

Example 9.2.3 Let us consider the streaming literals in Example 9.2.2.

The streaming literals l_1, l_2, l_7, and l_8 are ground as they contain no variable; on the contrary, l_3, l_4, l_5, and l_6 are non-ground as they contain the variable X. □

Harmless Literals. A streaming literal is said to be *harmless* if it has the form a **always in** $\{d_1, \ldots, d_m\}$, a **at least** c **in** $\{d_1, \ldots, d_m\}$, or *not* a **at most** c **in** $\{d_1, \ldots, d_m\}$ where $c \in C$ is a constant; otherwise, it is said to be *non-harmless*.

Example 9.2.4 Let us consider the streaming literals in Example 9.2.2 again.

The streaming literals l_1 and l_6 are harmless, while the remaining ones are non-harmless. □

9.2.2 Syntactic Shortcuts

Specific streaming atoms can be written using a shorter syntax. In particular, we can write:

- a **in** $\{d_1, \ldots, d_m\}$ in place of a **at least** *1* **in** $\{d_1, \ldots, d_m\}$;
- a in place of a **at least** *1* **in** $\{0\}$ (this is called *degenerate* form of a streaming literal);
- a **at least** c **in** $\{d_1, \ldots, d_m\}$ in place of *not* a **at most** c' **in** $\{d_1, \ldots, d_m\}$ where $c' = c - 1$.
- a **at most** c **in** $\{d_1, \ldots, d_m\}$ in place of *not* a **at least** c' **in** $\{d_1, \ldots, d_m\}$ where $c' = c + 1$ and $c' > 1$.

From the second shortcut, it is worth noting that predicate atoms are special cases of streaming atoms.

Given a streaming atom of any type, if its lookup set is $D = \{n \in \mathbb{N} \mid 0 \leq n \leq w \wedge w > 0\}$, we write $[w]$ in place of D.

Example 9.2.5 For example, consider the following streaming literals (on the left) with their shorthand (on the right).

- `warning` **`at least`** *`1`* **`in`** $\{0, 2\} \rightarrow_s$ `warning` **`in`** $\{0, 2\}$
- `warning` **`at least`** *`1`* **`in`** $\{0\} \rightarrow_s$ `warning`
- **`not`** `warning` **`at most`** *`1`* **`in`** $\{0, 2\} \rightarrow_s$ `warning` **`at least`** *`2`* **`in`** $\{0, 2\}$
- **`not`** `warning` **`at least`** *`2`* **`in`** $\{0, 2\} \rightarrow_s$ `warning` **`at most`** *`1`* **`in`** $\{0, 2\}$
- `warning` **`at least`** *`2`* **`in`** $\{0, 1, 2\} \rightarrow_s$ `warning` **`at least`** *`2`* **`in [2]`**

□

9.2.3 Rules and Programs

Rules. A *rule* r has one out of the following three forms:

1. $a :- l_1, \ldots, l_b.$
2. **#trigger_frequency(f)** $a :- l_1, \ldots, l_b.$
3. **#temp** $a :- l_1, \ldots, l_b.$

where: *(i)* $b \geq 0$; *(ii)* a is the *head* of r, denoted by $H(r)$, which is a predicate atom; *(iii)* $l_1, \ldots, l_b$ is the *body* of r, denoted by $B(r)$, which is a conjunction of streaming literals; *(iv)* $f \in \mathbb{N}^+$ is a number. The body of a rule can contain both positive and negative streaming literals. The set of positive (respectively, negative) streaming literals in $B(r)$ is called the *positive* (respectively, *negative*) body, and is denoted by $B^+(r)$ (respectively, $B^-(r)$).

We call rules of the form (1), (2), and (3) *normal rules*, *trigger rules*, and *temporary rules*, respectively. Moreover, given a trigger rule r, we denote as $f(r)$ the trigger frequency f of r.

Intuitively, trigger rules are applied only in some time points according to the associated frequency. Moreover, atoms inferred through temporary rules are considered only in the current time point and then forgotten. An example is reported below, whereas the semantics of each rule type is formally defined in Subsection 9.3.

Example 9.2.6 Example of rules are:

r_1: `warning :- rain, power(off)` **`at least`** *`2`* **`in`** {*`0,1,2`*}`.`

r_2: **`#trigger_frequency(`**`4`**`)`**
`working(w01) :- status(w01,"Production")` **`always in [`***`3`***`].`**

r_3: **`#temp`** `bus_stops(b01,s01,N):- stopped_bus(b01,s01)` **`count`** *`N`* **`in`**
`[`*`10`***`].`**

Let us suppose we are evaluating these rules at the time point 4 and that we derived the heads: `warning`, `working(w01)`, `bus_stop(b01,s01,4)`. The information that `warning` and `working(w01)` have been derived at the time point 4 is globally visible and can be considered in reasoning during next time points. On the contrary, the information that `bus_stop(b01,s01,4)` has been derived at the time point 4 is only visible at this time point and cannot be be considered in reasoning during next time points.

□

Ground Rules. For a predicate atom, a streaming atom, or a streaming literal o, $var(o)$ is the set of all variables appearing in o. For a non-empty set of streaming literals L, $var(L)$ is the set of variables appearing in every streaming literal $l \in L$.

A rule r is *ground* if the predicate atom in $H(r)$ is ground and all the streaming literals in $B(r)$ are ground, i.e., if $var(H(r)) \cup var(B(r)) = \emptyset$.

Example 9.2.7 Let us consider the rules introduced in Example 4.1.1. The rule r_1 is ground since its head and body are ground, while the other two rules, r_2 and r_3, are non-ground. □

Safe Rules. For a predicate atom, a streaming atom, or a streaming literal o, we denote as $var'(o) \subseteq var(o)$ the set of variables in o occurring outside of arithmetic terms. Let a non-empty set of streaming literals L, $safe(L) \subseteq var(L)$ denotes the set of safe variables of L.

Given a rule r, the set of its *safe* variables can be computed according to the following schema:

1. $safe(B(r)) = \emptyset$;
2. for each streaming literal $l \in B^+(r)$ having either form (1) a **always in** $\{d_1, \ldots, d_m\}$ or (4) a **count** t **in** $\{d_1, \ldots, d_m\}$,

$$safe(B(r)) = safe(B(r)) \cup var'(a)$$

3. for each streaming literal $l \in B^+(r)$ having form (2) a **at least** t **in** $\{d_1, \ldots, d_m\}$,

$$safe(B(r)) = safe(B(r)) \cup var'(base(a))$$

4. for each streaming literal *not* $\alpha \in B^-(r)$ such that α has form (3) a **at most** t **in** $\{d_1, \ldots, d_m\}$,

$$safe(B(r)) = safe(B(r)) \cup var'(base(l))$$

A rule r is safe if $var(H(r)) \subseteq safe(B(r))$, and every streaming literal $l \in B(r)$ is safe, i.e., $var(l) \subseteq safe(B(r))$.

Example 9.2.8 For instance, the rules in Example 4.1.1 are all safe, while the following rules are non-safe:

```
p(X):- q(Y) at least X in {0,3}

p(X):- q(X) at most 3 in {0,1,3,5}.

p(X):- q(Y) at least X in {0,3}, not z(Y) at most X in {1,2}.
```

□

9.2.4 Programs and Program Stratification

A program is a finite set of rules. It is said *flat* if it consists of rules which contain only streaming literals in the degenerate form. It is said *restricted* if it consists of only rules of the form (1) and (2).

Let Γ be an I-DLV-sr program. Γ is safe if every rule $r \in \Gamma$ is safe.

Program Stratification. A *stratification* for Γ is a partition of disjoint sets of its rules $\Gamma = \Pi_1 \cup \cdots \cup \Pi_k$, called *strata*, such that for $i \in \{1, \ldots, k\}$ both these conditions hold:

(i) for each harmless literal in the body of a rule in Π_i with predicate p, $\{r \in \Gamma \mid H(r) = \{p(t_1, \ldots, t_n)\}\} \subseteq \bigcup_{j=1}^{i} \Pi_j$;

(ii) for each non-harmless literal in the body of a rule in Π_i with predicate p, $\{r \in \Gamma \mid H(r) = \{p(t_1, \ldots, t_n)\}\} \subseteq \bigcup_{j=1}^{i-1} \Pi_j$.

Γ is *stratified* if a stratification $\Pi_1 \cup \cdots \cup \Pi_k$ exists for Γ, in this case Γ is said stratified by $\Pi_1 \cup \cdots \cup \Pi_k$.

Example 9.2.9 Let us consider the following I-DLV-sr program Γ_1.

```
r1  a(X) :- b(X).
r2  c(X,Y) :- a(X) always in [2], Y=X-1, not a(Y).
r3  e(X,Y) :- c(X,Y), d(Y), a(X) at most 2 in [3].
```

There exist three possible stratifications for Γ_1:

- $\{r_1\}_1 \cup \{r_2\}_2 \cup \{r_3\}_3$;
- $\{r_1\}_1 \cup \{r_3\}_2 \cup \{r_2\}_3$;
- $\{r_1\}_1 \cup \{r_2, r_3\}_2$.

Γ_1 contains two non-harmless literals, i.e., `not a(Y)` $\in B(r_2)$ and `a(X)` **`at most`** *`2`* **`in [`***`3`***`]`** $\in B(r_3)$; both of them feature $a/1$ as predicate, which is defined through $H(r_1)$. Therefore, according to the second point of the definition of stratification above, r_1 must be contained in a stratum preceding the ones where r_2 and r_3 are contained. On the contrary, r_2 and r_3 can be contained in the same stratum according to the condition *(i)* in the definition of stratification. Indeed, even if the body of r_3 features the predicate `c/2` defined by r_2, this is used by an harmless literal; therefore, condition *(i)* applies. □

9.3 Semantics

This section provides an operational semantics of I-DLV-sr programs that are safe and stratified.

9.3.1 Stream and Backward Observation

Definition 9.3.1 (Stream). A *stream* Σ is a sequence of sets of ground predicate atoms $\langle S_0, \ldots, S_n \rangle$ such that for $0 \leq i \leq n$, $S_i \subseteq G$; each natural number i is called *time point*.

Let Σ be a stream $\langle S_0, \ldots, S_n \rangle$. Given a predicate atom a, a is true w.r.t. Σ at the i-th time point if $a \in S_i$. Given another stream $\Sigma' = \langle S'_0, \ldots, S'_n \rangle$, $\Sigma = \Sigma'$ iff $S_i = S'_i$ for each $i \in \{0, \ldots, n\}$.

Example 9.3.1 An example of stream is $\Sigma_1 = \langle \{ \{a(2), b(5)\}_0, \{a(3), c(7)\}_1, \{b(5)\}_2, \{a(3)\}_3 \} \rangle$ where 0, 1, 2 and 3 are time points, and:

- `a(2)` is true at the time point 0
- `a(3)` is true at the time points 1 and 3;
- `b(5)` is true at the time points 0 and 2;
- `c(7)` is true at the time point 1.

The streaming atoms above are false in all the time points in Σ_1 where they are not explicitly present (e.g., `a(2)` is false from time point 1 to time point 3). □

Definition 9.3.2 (Backward Observation). Given a stream $\Sigma = \langle S_0, \ldots, S_n \rangle$ and a lookup set D, the *backward observation of* Σ *w.r.t.* D is the set $\{S_i \mid i = n - d \text{ with } d \in D \wedge i \geq 0\}$, and we denote it as $O(\Sigma, D)$.

Definition 9.3.3 (Window). Given $w \in \mathbb{N}$, a backward observation of Σ w.r.t. $[w]$ is called a *window*.

Intuitively, a backward observation identifies the sets of ground predicate atoms that are true w.r.t. Σ at the time points identified by means of D preceding the n-th time point.

Example 9.3.2 Let us consider the stream Σ_1 defined in Example 9.3.1.

- Given the lookup set $D_1 = \{0, 1, 3\}$, the backward observation of Σ_1 w.r.t. D_1 is $O(\Sigma_1, D_1) = \{\{a(3)\}, \{b(5)\}, \{a(2), b(5)\}\}$.
- Given the lookup set $D_2 = [2]$ (i.e., $\{0, 1, 2\}$), the backward observation of Σ_1 w.r.t. D_2 is $O(\Sigma_1, D_2) = \{\{a(3)\}, \{b(5)\}, \{a(3), c(7)\}\}$

□

9.3.2 Entailment of Streaming Literals

Given a stream $\Sigma = \langle S_0, \ldots, S_n \rangle$ and a ground streaming literal l whose lookup set and counting term are respectively D and c, the truth value of l at the n-th time point is defined by the backward observation $O(\Sigma, D)$. Table 9.1 reports when Σ *entails* either a ground streaming atom α, denoted by $\Sigma \models \alpha$, or its negation *not* α, denoted by $\Sigma \models \mathit{not}\ \alpha$. If $\Sigma \models \alpha$ (respectively, $\Sigma \models \mathit{not}\ \alpha$), we say that α is *true* (respectively, *false*) at the time point n.

α	$\Sigma \models \alpha$	$\Sigma \models \mathit{not}\ \alpha$
a **always in** $\{d_1, \ldots, d_m\}$	$\forall S \in O(\Sigma, D),\ a \in S$	$\exists S \in O(\Sigma, D) : a \notin S$
a **at least** c **in** $\{d_1, \ldots, d_m\}$	$\lvert\{S \in O(\Sigma, D) : a \in S\}\rvert \geq c$	$\lvert\{S \in O(\Sigma, D) : a \in S\}\rvert < c$
a **at most** c **in** $\{d_1, \ldots, d_m\}$	$\lvert\{S \in O(\Sigma, D) : a \in S\}\rvert \leq c$	$\lvert\{S \in O(\Sigma, D) : a \in S\}\rvert > c$
a **count** c **in** $\{d_1, \ldots, d_m\}$	$\lvert\{S \in O(\Sigma, D) : a \in S\}\rvert = c$	$\lvert\{S \in O(\Sigma, D) : a \in S\}\rvert \neq c$

Table 9.1: Entailment of ground streaming literals.

Example 9.3.3 Let us consider the stream Σ_1 defined in Example 9.3.1 and the streaming atoms: `b(5) at least 2 in {0,1,3}`, `a(3) at most 2 in [2]`, and `c(7)`. In order to determine whether Σ_1 entails them, we have to identify their lookup sets and compute the related backward observation on Σ_1.

The lookup sets of the first two streaming atoms coincide respectively with D_1 and D_2 in Example 9.3.2, where the related backward observations on Σ_1 are also defined. Given that it is a shortcut for the streaming atom `c(7) at least`

`1 in {0}`, the lookup set of `c(7)` is $D_3 = \{0\}$, and the backward observation of Σ_1 w.r.t. D_3 is $O(\Sigma_1, D_3) = \{\{a(3)\}\}$.

Eventually, we have that:

- $\Sigma_1 \models$ `b(5) at least 2 in {0,1,3}` as the cardinality of the set $\{S \in O(\Sigma_1, D_1) : b(5) \in S\}$ is 2, i.e., `b(5)` is true at least 2 times in the backward observation of Σ_1 w.r.t. $\{0, 1, 3\}$;
- $\Sigma_1 \models$ `a(3) at most 2 in [2]` as the cardinality of the set $\{S \in O(\Sigma_1, D_2) : a(3) \in S\}$ is 2, i.e., `a(3)` is true at most 2 times in the backward observation of Σ_1 w.r.t. $\{0, 1, 2\}$;
- $\Sigma_1 \models$ `not c(7)` as the cardinality of the set $\{S \in O(\Sigma_1, D_3) : c(7) \in S\}$ is 0, i.e., `c(7)` is false in the backward observation of Σ_1 w.r.t. $\{0\}$.

□

9.3.3 Rule Substitution and Application

Definition 9.3.4 (Substitution). A *substitution* is a mapping $\sigma : V \to C$ from the set of variables V to the set of constants C.

Let σ be a substitution. Given a predicate atom a, $\sigma(a)$ is the ground predicate atom a' obtained by replacing each occurrence of every variable $v \in var(a)$ by $\sigma(v)$. Given a streaming literal l, $\sigma(l)$ is the ground streaming literal l' obtained by applying, for every variable $v \in var(l)$, $\sigma(v)$ to the predicate atom *base(l)* and to the possible counting term t of l if $t = v$ and $\sigma(v) \in \{C \cap \mathbb{N}^+\}$.

Definition 9.3.5 (Rule Application). Given a stream $\Sigma = \langle S_0, \ldots, S_n \rangle$ and a rule r, a *rule application* of r on Σ is a substitution σ such that $\Sigma \models \sigma(l)$ for every $l \in B(r)$. If such a σ exists, then r is *applicable* on Σ via σ in the following cases:

i. r has the form (1) or (3), i.e., it is a normal or temporary rule, respectively;

ii. r has the form (2), i.e., it is a trigger rule, $n > 0$ is the current time point, and n **mod** $f(r) = 0$.

Intuitively, given a steam $\Sigma = \langle S_0, \ldots, S_n \rangle$, the applicability of rules on Σ identifies new ground predicate atoms that are true at the time point n; that is, a rule r fires implying the truth of the ground predicate atom $\sigma(a)$, with

$a \in H(r)$. According to the case i. above, a normal or temporary rule r fires, i.e., is applicable, in all time points of Σ for which a rule application of r exists. On the contrary, according to the case ii., if r is a trigger rule it may not fire in all time points of Σ, even if a rule application of r exists. Specifically, r can only be applicable in every time point n of Σ in which r must be triggered according to $f(r)$, i.e., when n **mod** $f(r) = 0$. This implies that r fires with a predefined frequency punctuated by $f(r)$ rather than in all time points. Notice that a trigger rule r does not fire in the first time point 0. By design, it start firing only when the first $f(r)$ time points passed.

Example 9.3.4 Let us consider the Σ_1 from Example 9.3.1 and the I-DLV-sr program Γ_2 next.

r_1
```
c(X) :- b(X) at least 2 in {0,1,3}.
```

As an example, consider the substitution σ_1 such that $X \mapsto 5$. The rule r_1 is applicable on Σ_1 via σ_1 given that $\Sigma_1 \models \sigma_1(\texttt{b(X) at least 2 in \{0,1,3\}})$, i.e., $\Sigma_1 \models \texttt{b(5) at least 2 in \{0,1,3\}}$ as we saw in Example 9.3.3.

□

9.3.4 Trigger and Trigger Application

Let $\Sigma = \langle S_0, \ldots, S_n \rangle$ be a stream and Γ an I-DLV-sr program.

Definition 9.3.6 (Trigger). A *trigger for* Γ *on* Σ is a pair $\langle r, \sigma \rangle$ where σ is a substitution and $r \in \Gamma$ is applicable on Σ via σ.

Definition 9.3.7 (Trigger Application.). Let $\langle r, \sigma \rangle$ be a trigger for Γ on Σ. A *trigger application* of $\langle r, \sigma \rangle$ to Σ is the stream $\Sigma' = \langle S_0, \cdots, S_{n-1}, S_n \cup \sigma(a) \rangle$, with $a \in H(r)$, and it is denoted as $\Sigma \langle r, \sigma \rangle \Sigma'$.

Example 9.3.5 Let us consider the Σ_1 from Example 9.3.1 and the I-DLV-sr program Γ_2 along with the substitution σ_1 defined in Example 9.3.4. The pair $\langle r_1, \sigma_1 \rangle$ is a trigger for Γ_2 on Σ_1. The application of $\langle r_1, \sigma_1 \rangle$ on Σ_1 returns the stream $\Sigma'_1 = \langle \{ \{a(2), b(5)\}_0, \{a(3), c(7)\}_1, \{b(5)\}_2, \{a(3), c(5)\}_3 \} \rangle$.

Additionally, let us take into account the coming I-DLV-sr program Γ_3.

r'_1
```
#trigger_frequency(2) c(X) :- b(X) at least 2 in {0,1,3}.
```

Given that $n = 3$, $f(r'_1) = 2$, and 3 **mod** $2 \neq 0$, Γ_3 is not applicable on Σ_1 via σ_1. Therefore, the pair $\langle r'_1, \sigma_1 \rangle$ is not a trigger for Γ_3 on Σ_1.

Consider now the I-DLV-sr program Γ_4.

r_1''
```
#trigger_frequency(3) c(X) :- b(X) at least 2 in {0,1,3}.
```

Since $n = 3$, $f(r_1'') = 3$, and 3 **mod** $3 = 0$, Γ_4 is applicable on Σ_1 via σ_1. Therefore, the pair $\langle r_1'', \sigma_1 \rangle$ is a trigger for Γ_4 on Σ_1. □

9.3.5 Stratum Application, Application Outcome, and Outcome over Strata

In order to define the concept of streaming model of an I-DLV-sr program Γ on a stream Σ, it is crucial to establish the order of application of the rules in Γ according to a stratification for Γ.

Let Σ be a stream and Γ be an I-DLV-sr program stratified by $\Pi_1 \dots \Pi_k$.

Definition 9.3.8 (Stratum Application). Given a stratum Π_s in $\Pi_1 \dots \Pi_k$, a *stratum application* of Π_s on Σ is a finite sequence of streams $\Sigma_0, \dots, \Sigma_h$ with $\Sigma_0 = \Sigma$ and $h \geq 0$ such that:

- for each $0 \leq i < h$, there is a trigger $\langle r_i, \sigma_i \rangle$ for Π_s on Σ_i such that $\Sigma_i \langle r_i, \sigma_i \rangle \Sigma_{i+1}$;
- for each $0 \leq i < j < h$, if $\Sigma_i \langle r_i, \sigma_i \rangle \Sigma_{i+1}$, $\Sigma_j \langle r_j, \sigma_j \rangle \Sigma_{j+1}$ and $r_i = r_j$, then $\sigma_i \neq \sigma_j$;
- there is no trigger $\langle r, \sigma \rangle$ for Π_s on Σ_h such that $\langle r, \sigma \rangle \notin \{\langle r_i, \sigma_i \rangle\}_{0 \leq i \leq h}$.

Intuitively, starting from a stream Σ, distinct triggers are applied by considering only those having different substitutions for the same rule as long as there is a new trigger.

In general, for each stratum Π_s in $\Pi_1 \dots \Pi_k$, more than one stratum application exists. However, for any pair of stratum applications of Π_s on Σ, their last streams coincide.

Proposition 9.3.1 Given a stratum Π_s in $\Pi_1 \dots \Pi_k$ and two stratum applications of Π_s on Σ, $\Sigma_0, \dots, \Sigma_h$ and $\Sigma_0', \dots, \Sigma_t'$, we have that $\Sigma_h = \Sigma_t'$.

Proof. We first remark that by definition of stratum application on Σ, $\Sigma_0 = \Sigma_0' = \Sigma$ and that a trigger application adds ground predicate atoms only to the last set of a stream. For convenience, given a stream $\Sigma = \langle S_0, \dots, S_n \rangle$, we denote the last set S_n as $last(\Sigma)$. We prove that if a predicate atom a belongs

to the last set of one of the two streams Σ_h or Σ'_t, then a necessarily belongs to the last set of the other stream. Therefore, let us suppose that $a \in last(\Sigma_h)$, we prove that $a \in last(\Sigma'_t)$. In particular, we show that $\forall i \in \{0,\ldots,h\}, a \in last(\Sigma_i) \implies \exists j_a \in \{0,\ldots,t\} : a \in last(\Sigma'_{j_a})$. We proceed by induction:

- $a \in last(\Sigma_0)$. Since $\Sigma_0 = \Sigma'_0 = \Sigma$, we have that $j_a = 0$.
- We assume that $a \in last(\Sigma_n) \implies \exists j_a \in \{0,\ldots,t\} : a \in last(\Sigma'_{j_a})$.
- If $a \in last(\Sigma_{n+1})$ we can have that either $a \in last(\Sigma_n)$ and by inductive hypothesis there exists $j_a \in \{0,\ldots,t\} : a \in last(\Sigma'_{j_a})$ or Σ_{n+1} is the result of the application of the trigger $\langle r_n, \sigma_n\rangle$ on Σ_n, i.e., $\Sigma_n\langle r_n, \sigma_n\rangle\Sigma_{n+1}$, with $a = \sigma_n(l)$ where $l \in H(r_n)$. In the latter case, we have that $\Sigma_n \models \sigma_n(b)\forall b \in B(r_n)$. If $b \in B(r_n)$ is a non-harmless literal its truth value cannot depend on rules belonging to stratum Π_s. Then $\Sigma_n \models \sigma_n(b)$ iff $\Sigma \models \sigma_n(b)$. If $b \in B(r_n)$ is an harmless literal with predicate atom $p(t_1,\ldots,t_p)$ we can have that $\Sigma \models \sigma_n(b)$ or we can have that $\sigma_n(p(t_1,\ldots,t_p)) \in last(\Sigma_n)$. By inductive hypothesis we have that $\exists j_b \in \{0,\ldots,t\} : \sigma_n(p(t_1,\ldots,t_p)) \in \Sigma'_{j_b}$. Hence, there exists a stream Σ'_m, with $m \in \{0,\ldots,t-1\}$ such that $\Sigma'_m \models \sigma_n(b)\forall b \in B(r_n)$ and there exists $m < j_a \leq t$ such that $a \in \Sigma'_{j_a}$. □

Definition 9.3.9 (Application Outcome). Given a stratum Π_s in $\Pi_1 \ldots \Pi_k$, an *application outcome* of Π_s on Σ is the last stream in a stratum application of Π_s on Σ, and it is denoted by $outcome(\Pi_s, \Sigma)$.

We now define the outcome over strata of Γ on Σ, obtained, starting from the outcome of the first stratum Π_1 of Γ and considering, one after the other, subsequent strata according to the stratification.

Definition 9.3.10 (Outcome Over Strata). Let $\Sigma_{\Pi_1},\ldots,\Sigma_{\Pi_k}$ be the sequence of streams computed as follows; for all $i \in \{1,\ldots,k\}$:

$$\Sigma_{\Pi_i} = \begin{cases} outcome(\Pi_i, \Sigma) & \text{if } i = 1 \\ outcome(\Pi_i, \Sigma_{\Pi_{i-1}}) & \text{if } 1 < i \leq k \end{cases}$$

The *outcome over strata* of Γ on Σ, denoted by $\mathcal{R}(\Gamma, \Sigma)$, is the n-th element in the stream Σ_{Π_k}.

Note that $\mathcal{R}(\Gamma, \Sigma)$ is independent from the chosen stratification.

9.3.6 Streaming Models

Streaming model of I-DLV-sr restricted programs. For simplicity, we first define the notion of streaming model for programs where only rules of the forms (1) and (2) appear.

Definition 9.3.11 Given a stream $\Sigma = \langle S_0, \ldots, S_n \rangle$ and an I-DLV-sr restricted program Γ stratified by $\Pi_1 \ldots \Pi_k$. Let $\Sigma' = \langle S'_0, \ldots, S'_{n-1}, S_n \rangle$ be the stream constructed as follows; for all $i \in \{0, \ldots, n-1\}$:

$$S'_i = \begin{cases} \mathcal{R}(\Gamma, \langle S_0 \rangle) & \text{if } i = 0 \\ \mathcal{R}(\Gamma, \langle S'_0, \ldots, S'_{i-1}, S_i \rangle) & \text{if } 0 < i \leq n-1 \end{cases}$$

The streaming model of Γ *on* Σ is $\mathcal{R}(\Gamma, \Sigma')$.

Intuitively, the streaming model of Γ for a stream $\Sigma = \langle S_0, \ldots, S_n \rangle$ is the set of the ground predicate atoms derived as true at the n-th time point. These latter come from the evaluation of Γ over a stream that iteratively accumulates the evaluations at previous time points.

Example 9.3.6

Let us consider the stream $\Sigma_2 = \langle \{ \{b(5)\}_0, \{c(7)\}_1 \} \rangle$ and the following I-DLV-sr program Γ_5:

r_1 `c(X) :- b(X).`
r_2 `d(X) :- c(X) in [1].`

Γ_5 is stratified by the single stratum $\{r_1, r_2\}$; thus, the outcome over strata of Γ_5 on Σ_2, $\mathcal{R}(\Gamma_5, \Sigma_2)$, is the n-th element in the stream $outcome(\Gamma_5, \Sigma_2)$. Then, we have $\Sigma'_2 = \langle S'_0, S_1 \rangle$ where $S'_0 = \mathcal{R}(\Gamma_5, \langle S_0 \rangle) = \mathcal{R}(\Gamma_5, \langle \{b(5)\}_0 \rangle) = \{b(5), c(5), d(5)\}$ and $S_1 = \{c(7)\}$. Since $\mathcal{R}(\Gamma_5, \Sigma'_2) = \mathcal{R}(\Gamma_5, \langle \{b(5), c(5), d(5)\}_0, \{c(7)\}_1 \rangle) = \{c(7), d(7), d(5)\}$, the streaming model of Γ_5 for Σ_2 is the set $\{c(7), d(7), d(5)\}$. □

Streaming model of I-DLV-sr programs. In this paragraph, the definition of the streaming model is also extended to I-DLV-sr non-restricted programs. To this aim, first, the notion of *persistent outcome* is introduced, and then a slightly different definition of streaming model is presented, which takes into account the eventual presence of #**temp** in rule heads.

Intuitively, the marking #**temp** stands for temporary: indeed, the truth of heads of rules in the form (3) is, in a sense, limited to the current time point.

Let $\Sigma = \langle S_0, \ldots, S_n \rangle$ be a stream, and Γ be an I-DLV-sr program stratified by $\Pi_1 \ldots \Pi_k$.

Definition 9.3.12 (Persistent Outcome Over Strata). Let $\Gamma_{(1,2)} = \{r \in \Gamma \mid r \text{ is of the form (1)} \vee r \text{ is of the form (2)}\}$. The *persistent outcome over strata* of Γ on Σ, denoted by $\mathcal{P}(\Gamma, \Sigma)$, is the set of ground predicate atoms $a \in \mathcal{R}(\Gamma, \Sigma)$ such that either $a \in S_n$ or there exists a rule $r \in \Gamma_{(1,2)}$ applicable to Σ_{Π_k} via a substitution σ, and $a = \sigma(h)$ where $h \in H(r)$.

Basically, for an input stream Σ, $\mathcal{P}(\Gamma, \Sigma)$ identifies the set of ground predicate atoms in S_n or in $\mathcal{R}(\Gamma, \Sigma)$ deriving from heads of rules of the form (1) or (2).

Definition 9.3.13 Let Σ' be the stream $\Sigma' = \langle S'_0, \ldots, S'_{n-1}, S_n \rangle$ defined as follows; for all $i \in \{0, \ldots, n-1\}$:

$$S'_i = \begin{cases} \mathcal{P}(\Gamma, \langle S_0 \rangle) & \text{if } i = 0 \\ \mathcal{P}(\Gamma, \langle S'_0, \ldots, S'_{i-1}, S_i \rangle) & \text{if } 0 < i \leq n-1 \end{cases}$$

The streaming model of Γ *on* Σ is $\mathcal{R}(\Gamma, \Sigma')$.

Definition 9.3.13 differs from Definition 9.3.11 as it considers the persistent outcome over strata for time points up to $n-1$ and the outcome over strata for n including the heads of rules of form (3).

Example 9.3.7 Let Σ_2 be the stream of Example 9.3.6 and Γ_6 be the following program:

```
r1  #temp c(X) :- b(X).
r2  d(X) :- c(X) in [1].
```

Since the first rule of Γ_6 is of the form (3), we have that $S'_0 = \mathcal{P}(\Gamma_6, \langle S_0 \rangle) = \mathcal{P}(\Gamma_6, \langle \{b(5)\}_0 \rangle) = \{b(5), d(5)\}$ and $\Sigma'_2 = \langle S'_0, S_1 \rangle = \langle \{b(5), d(5)\}_0, \{c(7)\}_1 \rangle$. The streaming model of Γ_6 on Σ_2 is the set $\mathcal{R}(\Gamma_6, \Sigma'_2) = \mathcal{R}(\Gamma_6, \langle \{b(5), d(5)\}_0, \{c(7)\}_1 \rangle) = \{c(7), d(7)\}$. □

9.4 An Insight About the Differences with LARS

Although I-DLV-sr and LARS extend ASP for reasoning over streams, they deeply differ in syntax and semantics. In a recent work [96], I-DLV-sr and LARS languages have been compared in terms of expressiveness. The authors

formally proved that, without any restriction, they are incomparable in practice. Some insights about their similarities and differences highlighted in the aforementioned work are reported next.

I-DLV-sr and LARS adopt a similar management of the stream timeline and elements but a different management of information that is always true. In particular, they both rely on a punctuated timeline where time is discretized in time points and represent elements into the stream via sets of ASP ground predicate atoms associated with different time points. I-DLV-sr represents information always true as standard ASP facts associated with all the time points; conversely, LARS uses *background atoms* that are not necessarily associated with every time point.

Eventually, I-DLV-sr and LARS produce different types of output. On the one hand, the former returns a *streaming model* that is a single set of information related to the most recent time point. On the other hand, the latter returns another stream for each time point of the input stream, which is the *answer stream* at that time point.

Example 9.4.1 Let us consider the I-DLV-sr program Γ_7 next.

r_1
```
a(Y) :- b(Y).
```
r_2
```
c(Y) :- a(Y) in [1]
```

To some extent, Γ_7 can be translated in a LARS program Γ'_7.

r_1 $@_T$ a(Y) $\leftarrow$ $@_T$ b(Y).

r_2 c(Y) $\leftarrow \boxplus^1 \Diamond$ a(Y).

The following table reports the results of evaluating Γ_7 and Γ'_7 over different input streams.

Formalism	**Input Stream**	**Output**
LARS	$\{0 \mapsto \{b(5)\}$	$0 \mapsto \{a(5), b(5), c(5)\}$
I-DLV-sr	$\langle\{\ \{b(5)\}_0\}\rangle$	$\{a(5), b(5), c(5)\}$
LARS	$\{0 \mapsto \{b(5)\}, 1 \mapsto \{b(10)\}$	$0 \mapsto \{a(5), b(5), c(5)\}$ $1 \mapsto \{c(5), a(10), b(10), c(10)\}$
I-DLV-sr	$\langle\{\ \{b(5)\}_0,\ \{b(10)\}_1\}\rangle$	$\{c(5), a(10), b(10), c(10)\}$

Here we can observe that, while the streaming model of I-DLV-sr only consists of the derivations obtained in the last time point of the input stream, the

answer stream of LARS consists of the derivations obtained in all the time points of the input stream. □

Given these differences, the authors of the work cited above designed a formal framework to determine whether both formalisms may produce the same output starting from the same input stream. Therefore, using this framework and introducing restrictions to the languages, they identified large fragments of each of the two languages that can be expressed via the other one. Such fragments consist of rules using specific constructs of one language that can be expressed in the other through some rewriting process.

9.5 Advanced Constructs

The @now construct. In an I-DLV-sr program, usually, the user is not required to handle time (points) explicitly; everything is transparently handled so that, at each time point t, each atom that is either inferred or part of the input stream is automatically associated with t. However, some applications require the capability of explicitly reasoning over time and time points. To this aim, we introduced the **@now** construct, which consists of a special term that, at each time point t, is automatically assigned with the value of t. For instance, one could write the following rule:

r_1
```
grant_permission(X):- allowed(X), validity_interval(X,T1,T2),
          T1 <= @now, @now <= T2.
```

Note that the term **@now** can be either (1) numeric, i.e., an integer number representing t in *seconds*, *minutes*, or *hours* or (2) textual, i.e., a string in the *datetime* format:`''yyyy-MM-ddTHH:mm:ss.SSS''`, where milliseconds (`.SSS`) can be omitted if time points are expressed in larger time units. By default, **@now** is in seconds, but the format can be set via the option: `--now-format = <sec|min|hrs|datetime>`.

Intuitively, the **@now** term allows for accessing the last time point of the input stream. It behaves similarly to classical ASP terms, therefore it can be used in both the body or the head of a rule. In the former case, it is useful for checking temporal conditions and relations and, on the basis of these, deriving new information. In the latter case, it is useful for explicitly representing time, enabling the implementation of more sophisticated reasoning tasks.

Example 9.5.1 Let us consider the following program:

```
r1  grant_permission(X, @now):- allowed(X), validity_interval(X,T1,T2),
              T1 <= @now, @now <= T2.
r2  granted_before(X,Y) :- grant_permission(X, T1) in [2],
              grant_permission(Y, T2) in [2], T1<T2.
```

Intuitively, r_1 determines whether it is possible to grant permission to a given entity `X` based on its validity interval (indicated by `validity_interval/3`) at the current time point, i.e., the one associated with **@now**. Therefore, the second term of the predicate `grant_permission/2` indicates the time point at which an entity obtains the permission. On the basis of this, r_2 determines if an entity `X` has obtained permission before an entity `Y` within the last window of size 2, using the temporal information made explicit through r_1. □

External Sources of Computation. I-DLV-sr supports *external literals* [36] in rule bodies that can be used to call external sources of computation via Python3.

We recall that an *external atom* $\boldsymbol{e}$ is of the form

$$\&p(i_0, \ldots, i_n; o_0, \ldots, o_m)$$

where $n, m \geq 0$, $\&p$ is a predicate name, $i_0, \ldots, i_n$ are input terms and $o_0, \ldots, o_m$ are output terms; input and output terms can be both constants or variables.

An *external literal* is either $\boldsymbol{e}$ or ***not*** $\boldsymbol{e}$. The semantics is externally defined by the user via Python, and output terms are computed based on input ones by applying such user-defined semantics.

This construct is useful in scenarios where the reasoning tasks should also consider the results of domain-specific algorithms. Furthermore, it is worth noting that they allow using all the Python features, thus paving the way to a number of interesting possibilities, such as the integration of Machine Learning solutions within deductive SR applications.

Chapter 10

System Design and Architecture

This chapter outlines the design and architecture of I-DLV-sr. Section 10.1 describes the workflow of the system, including information on the configuration and execution phases. Sections 10.2 and 10.3 provide the reader with additional details on the functioning of the main modules of I-DLV-sr's architecture and their interaction.

I-DLV-sr functioning relies on continuous cooperation between two components: a custom *Flink*-based Java application (Chapter 3) [46, 85, 113] and $\mathcal{I}^2$-*DLV* (Chapter 5) [73, 38]. The former is in charge of processing real-time data streams, handling temporal information, and computing the semantics of SR-related language constructs introduced in Chapter 9. The latter performs complex reasoning tasks necessary for evaluating I-DLV-sr programs.

Figure 10.1 reports the high-level architecture of the system, mainly composed of three modules: the *Execution Manager*, the *Stream Manager*, and the *Subprogram Manager*, all making ad-hoc use of *Flink* APIs. In this Chapter: *(i)* Section 10.1 provides the reader with a high-level description of the computation workflow performed by I-DLV-sr when it must evaluate an input program Γ over an input stream Σ; *(ii)* Section 10.2 describes in detail the operations carried out by the *Execution Manager* to properly process Γ; *(iii)* Section 10.3 discusses the interaction between the *Stream Manager* and *Subprogram Manager* to evaluate Γ.

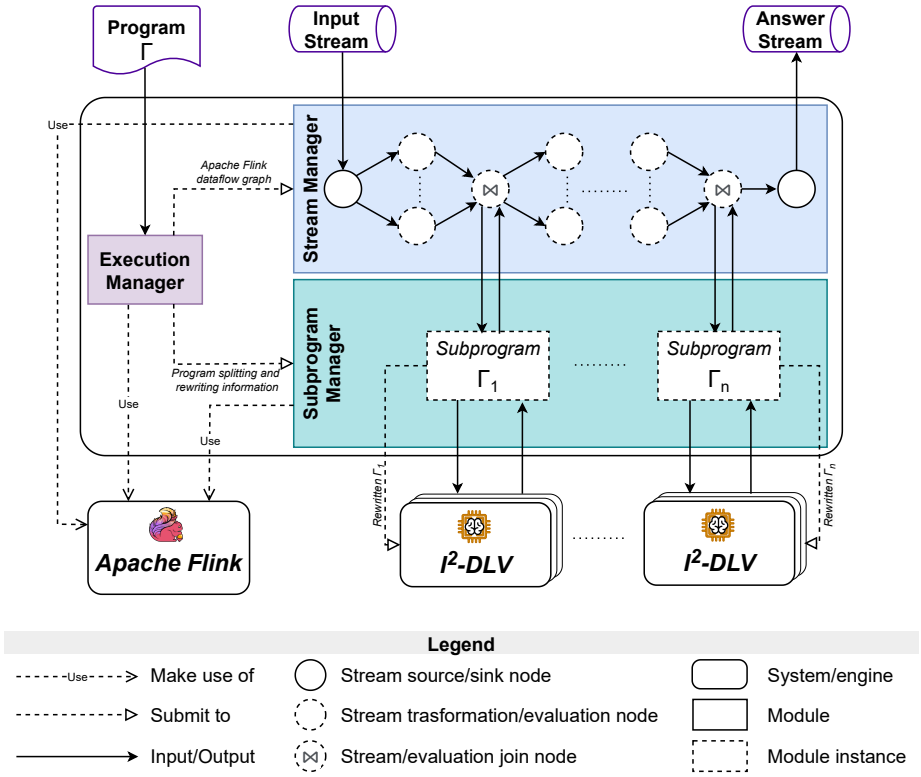


Figure 10.1: The high-level architecture of I-DLV-sr

10.1 The Workflow

The system takes as input an I-DLV-sr program Γ and a stream $\Sigma = \langle S_0, \ldots, S_n \rangle$, and iteratively builds an output stream $O = \langle O_0, \ldots, O_n \rangle$ such that each $O_t \in O$ contains the result of the evaluation of Γ at the time point t. Γ is a static input that I-DLV-sr receives at the beginning of its execution and never changes; conversely, Σ is a dynamic stream (theoretically infinite) whose elements are continuously consumed by the system as soon as the data source produce them. Therefore, it is worth noting that elements in O are gradually computed in real-time while receiving new elements of Σ without waiting for its end.

The workflow of I-DLV-sr consists of two phases: the *configuration* phase and the *execution* phase. During the first phase, the *Execution Manager* and the *Subprogram Manager* process the input program Γ and set up the system so that it can evaluate Γ. During the second phase, the *Stream Manager*, the *Subprogram Manager*, and $\mathcal{I}^2$-*DLV* continuously interact to evaluate Γ.

The Configuration Phase

The configuration phase starts when the *Execution Manager* receives an I-DLV-sr program Γ as input. It defines the sets of tasks that must be accomplished to evaluate Γ and distributes such tasks among the remaining modules.

Initially, the *Execution Manager* splits Γ in subprograms $\Gamma_1, \ldots, \Gamma_n$ (i.e., disjoint sets of rules), taking into account dependencies among predicates of the streaming literals appearing in Γ. Subsequently, according to such dependencies, an evaluation order among subprograms is established and used to create the *Flink* dataflow graph (see Section 3.1). The *Flink* dataflow graph represents, through a set of nodes and directed arcs, the sequence of operations that must be performed over the stream to evaluate Γ. It is constructed incrementally following the subprogram ordering as described next. Each subprogram Γ_i is compiled in a *Flink* dataflow sub-graph G^{Γ_i} corresponding to the sequence of operations for evaluating Γ_i. G^{Γ_i} is possibly connected to every other sub-graph G^{Γ_j} such that *(i)* $j < i$, and *(ii)* G^{Γ_j} corresponds to a subprogram Γ_j on which Γ_i depends.

Additionally, once subprograms are individuated, they undergo a rewriting process. Each Γ_i is rewritten by mapping every streaming literal $l \in \Gamma$ to a standard ASP literal, thus obtaining a "flattened" version Γ'_i of Γ_i.

Eventually, the *Execution Manager* submits the *Flink* dataflow graph to the *Stream Manager*, and the program splitting along with the rewriting information to the *Subprogram Manager*.

The *Subprogram Manager* for each subprogram Γ_i, instantiates at least one instance of $\mathcal{I}^2$-*DLV* to which the "flattened" version Γ'_i is submitted. For the whole execution of I-DLV-sr, each instance of $\mathcal{I}^2$-*DLV* is in charge of evaluating only the subprogram it received during the configuration phase. In general, the *Subprogram Manager* may instantiate multiple instances of the reasoner for the same subprogram Γ_i because of the following reason. The *Flink* operators in charge of evaluating subprograms may have multiple instances that are executed by different threads (see Subsection 3.1.1). Consequently, Γ_i could be evaluated simultaneously by multiple threads for different time points. Therefore, in order to avoid the bottleneck that would be created using a shared reasoner, each thread has its own $\mathcal{I}^2$-*DLV* instance.

Once this configuration phase ends, I-DLV-sr is up and running, ready to

consume the input stream and produce the output stream.

The Execution Phase

In the execution phase, the *Stream Manager*, the *Subprogram Manager*, and $\mathcal{I}^2$-*DLV* continuously collaborate to evaluate, at each time point, every subprogram $\Gamma_i \in \Gamma$ over Σ. To achieve this, they follow the instructions received from the *Execution Manager* and the chosen evaluation order, as described next.

The *Stream Manager* connects to the data source and starts to consume the input stream, forwarding its elements along the *Flink* dataflow graph over which its nodes (namely, operators) are executed.

For each $\Gamma_i \in \Gamma$, the *Stream Manager* executes the related *Flink* dataflow sub-graph G^{Γ_i}. The latter computes the semantics of the set L_i of streaming literals appearing in Γ_i. Specifically, it performs such computation considering both Σ and the results of previous evaluations using properly customized *Flink*-based stateful operators. The execution of all the nodes in G^{Γ_i} produces a set of true ground streaming literals R_i. Subsequently, the *Stream Manager* forwards R_i to the *Subprogram Manager* in order to obtain the evaluation of Γ_i.

The *Subprogram Manager*, in turn: *(i)* translates each element in R_i in a ground standard ASP literal producing the set R'_i; *(ii)* feeds R'_i to one of the $\mathcal{I}^2$-*DLV* instances in charge of handling Γ_i, which incrementally evaluates the flattened version Γ'_i on the set R'_i. The *Subprogram Manager* creates R'_i according to the rewriting information received from the *Execution Manager*. Intuitively, R'_i represents the set of ground instances of the standard ASP literals used in Γ'_i in place of the streaming literals in L_i. When the *Subprogram Manager* gets the new derivations from $\mathcal{I}^2$-*DLV*, it feeds them back to the *Stream Manager*.

The *Stream Manager* forwards the new derivations in the remaining part of the *Flink* dataflow graph and starts the evaluation of the next subprogram Γ_{i+1}.

Eventually, for the given time point t, the output of I-DLV-sr consists of the composition of the evaluation results of each subprogram Γ_i at t; continuous evaluations of Γ, time point by time point, form the output stream. Notably, independent subprograms might be evaluated in parallel, and a subprogram might be evaluated more times in case of recursion.

10.2 *Execution Manager*

This section provides details on the *Execution Manager* tasks.

As we introduced in the previous section, the *Execution Manager* is in charge of setting up the evaluation of an I-DLV-sr program Γ. First, it divides Γ into subprograms $\Gamma_1 \cdots \Gamma_n$, taking into account dependencies among all rules in Γ caused by streaming atoms. In such a way, each subprogram can be separately processed, limiting the interplay between *Flink* and $\mathcal{I}^2$-*DLV*. According to the program splitting, it constructs the *Flink* dataflow graph (see Section 3.1). Moreover, it determines a mapping τ from streaming atoms in Γ to fresh predicate atoms so that it can compute from each Γ_i a flat program Γ'_i. Finally, it provides the *Stream Manager* with the dataflow graph and the *Subprogram Manager* with the program splitting and τ.

10.2.1 Program Splitting and Processing Order

Let Γ be an I-DLV-sr program.

Stream Dependency Graph. The *Stream Dependency Graph* (SDG) of Γ, denoted by G_Γ^{SD}, is a directed labeled graph whose nodes are the predicates in rule heads of Γ, and for each pair of nodes p and q, there is an arc (p, q) if there exists a rule $r \in \Gamma$ such that $preds(H(r)) = \{q\}$ and $p \in preds(B(r))$; moreover, if p occurs in a streaming literal that is not in the degenerate form, then the arc is labeled with "<"; no label is added otherwise.

Intuitively, relying on the SDG, it is possible to identify *streaming dependencies* among predicates. If an arc (p, q) is labeled with "<" it means that the evaluation of predicate q depends on the evaluation results of predicate p at the *past* time points of the stream.

A rule $r \in \Gamma$ is *streaming-recursive* if there is a cycle in G_Γ^{SD} between two nodes p and q, where $p \in preds(B(r))$ and $q \in preds(H(r))$, and such a cycle contains at least one arc labeled with "<".

Example 10.2.1 As an example, let us consider the I-DLV-sr program Γ_8:

```
r1  a(X) :- b(X) always in [2].
r2  b(Y) :- a(X) in [1], Y=X+1, c(Y).
r3  d(X) :- b(X) at least 2 in [4].
r4  e(X,Y) :- a(X), b(Y).
```

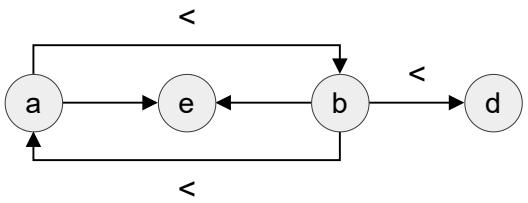


Figure 10.2: Stream Dependency Graph $G^{SD}_{\Gamma_8}$ of Γ_8

Figure 10.2 reports the stream dependency graph $G^{SD}_{\Gamma_8}$, which is composed of 4 nodes as the number of head predicates in Γ_8 and 5 arcs representing the dependencies among these predicates. For instance, given that $H(r_3) = d$ and $b \in B(r_3)$, there is an arc from node b to node d; also, (b, d) is labeled with "<" because d depends on b via the non-degenerate streaming literal `b(X) at least 2 in [4]`. Intuitively, this indicates the presence of a streaming dependency of d on b: in order to derive all the values of d, it is necessary to look backward into the stream and analyze the values of b that have been previously derived, specifically, at the time points indicated by the lookup set of `b(X) at least 2 in [4]`.

Notice that r_1 and r_2 are streaming recursive rules due to the streaming dependencies between predicates a and b, which form a cycle in $G^{SD}_{\Gamma_8}$ where both arcs (a, b) and (b, a) have "<" as label. □

Stream Component Graph. Given a stream dependency graph G^{SD}_{Γ}, we define an additional directed labeled graph, called *Stream Component Graph* (SCG) and denoted by G^{SC}_{Γ}, that is computed on the basis of G^{SD}_{Γ}. Specifically, the nodes of G^{SC}_{Γ} are the strongly connected components (SCCs) of G^{SD}_{Γ} (i.e., sets of predicates), and for each pair of components A and B, it contains an arc from A to B if there exists an arc $(p,\ q)$ in G^{SD}_{Γ} such that $p \in A$ and $q \in B$. Each arc (A, B) in G^{SC}_{Γ} is labeled with "<" if there exists at least a pair of predicates $p,\ q$ such that: $p \in A,\ q \in B$ and the arc (p, q) is present in G^{SD}_{Γ} with "<" as label; no label is added otherwise.

Example 10.2.2 Consider the program Γ_8 in Example 10.2.1 and its stream dependency graph in Figure 10.2. Figure 10.3 depicts the stream component graph $G^{SC}_{\Gamma_8}$.

Given that a and b are recursive predicates forming an SCC of the graph $G^{SD}_{\Gamma_8}$, they collapse into a single node in $G^{SC}_{\Gamma_8}$. On the contrary, predicates d

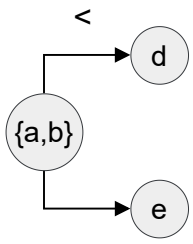


Figure 10.3: Stream Component Graph $G^{SC}_{\Gamma_8}$ of Γ_8

and e constitute separately two SCCs of $G^{SD}_{\Gamma_8}$; therefore, they still have their own node in $G^{SC}_{\Gamma_8}$. Additionally, the arc $(\{a, b\}, d)$ is labeled with "$<$" because of the arc (b, d) in $G^{SD}_{\Gamma_8}$, which features "$<$" as label.

□

Similarly to what happens in the SDG, if two components A and B are connected via an arc (A, B) labeled with "$<$", this means that there is a streaming dependency between them and, in particular, that the evaluation of B depends on the evaluation of A at the previous time points of the stream.

Precedence relationships among Stream Components Given a G^{SC}_{Γ}, according to the labeling of its arcs, it is possible to establish streaming precedence relationships among all the contained stream components. In particular, for any pair of nodes A and B of G^{SC}_{Γ}, we say that A *precedes* B (denoted $A \prec B$) if there exists a path in G^{SC}_{Γ} from A to B containing at least one arc labeled with "$<$"; we say A *is alongside* B (denoted $A \approx B$) otherwise. Such relationships induce a partial ordering among stream components. Intuitively, in the first case, A must be evaluated before B, as B is streaming dependent on A, while in the second case, A can possibly be evaluated with B.

Example 10.2.3 Let us consider the SCG in Figure 10.3; the precedence relationships among its nodes are: $\{a, b\} \prec \{d\}$, $\{a, b\} \approx \{e\}$,$\{e\} \approx \{a, b\}$, $\{d\} \approx \{e\}$ and $\{e\} \approx \{d\}$. For instance, we have that $\{d\}$ *is alongside* $\{e\}$ because there is no dependency between these two stream components; thus, they can be evaluated together. On the contrary, $\{a, b\}$ *precedes* $\{d\}$ due to the streaming dependency existing between b and d, which prevents them from being evaluated together and forces the stream component $\{a, b\}$ to be evaluated before the stream component d. □

Macro-nodes creation and ordering Relying on the graph G_Γ^{SC}, rules of Γ that can be processed together are grouped, and a processing ordering among groups of rules is established. We identify an ordering $C_1, \ldots, C_n$ of the nodes of G_Γ^{SC} such that, for each $i < j$, it does not hold that $C_j \prec C_i$. According to such an ordering $C_1, \ldots, C_n$, we collect the predicates of some consecutive nodes in order to form a macro-node as follows: for each pair of nodes C_i and C_k, with $i \leq k \leq n$, we construct the macro-node $M = \bigcup_j C_j$ with $i \leq j \leq k$, and such that one of the following conditions holds:

i. $i = k$, i.e., the macro-node actually consists of a single node;

ii. for each $j \neq k$, $C_j \approx C_{j+1}$, i.e., the macro-node consists of nodes that are all alongside each other.

We can therefore define an ordered sequence of all the maximal macro-nodes such that for any pair of macro-nodes $M_1 = \bigcup_{j_1} C_{j_1}$ with $i_1 \leq j_1 \leq k_1 \leq n$ and $M_2 = \bigcup_{j_2} C_{j_2}$ with $i_2 \leq j_2 \leq k_2$, if M_1 precedes M_2 in the sequence, then $k_1 < i_2$. This ordered sequence induces both a splitting of Γ into subprograms and a processing order for them. Specifically, for each macro-node M:

- the subprogram Γ_M is the set of all rules r of Γ such that $preds(H(r)) \subseteq M$, i.e., the predicate occurring in $H(r)$ belongs to M;
- the processing order for the subprograms coincides with the ordering in the sequence of the corresponding macro-nodes.

Note that for an I-DLV-sr program, several orderings of the nodes of G_Γ^{SC} might exist in general, and therefore, different processing orders might be obtained. In the following, we will refer to one of these processing orders as Ord_Γ.

Example 10.2.4 Let us consider the program Γ_8 in Example 10.2.1 and its SCG in Figure 10.3 again. According to the precedence relationships introduced in Example 10.2.3, there are three possible orderings of the nodes in $G_{\Gamma_8}^{SC}$:

$o_1 = \{\{e\}, \{a, b\}, \{d\}\}$;

$o_2 = \{\{a, b\}, \{e\}, \{d\}\}$;

$o_3 = \{\{a, b\}, \{d\}, \{e\}\}$.

Let us focus on o_1. The maximal macro-nodes that we can build are $\{a, b, e\}$ and $\{d\}$ and the resulting program splitting is: $\Gamma_{8\{a,b,e\}} = \{r_1, r_2, r_4\}$, $\Gamma_{8\{d\}} = \{r_3\}$. Note that o_2 would induce the same maximal macro-nodes and thus the same splitting of o_1. In both cases, $\{r_1, r_2, r_4\}$ are processed before $\{r_3\}$. □

10.2.2 Flink Dataflow Graph Creation

As described in the previous section, given an I-DLV-sr program Γ, the *Execution Manager* splits it into subprograms and computes a processing order $Ord_\Gamma = \Gamma_1, ..., \Gamma_n$. According to Ord_Γ, it builds a *Flink* dataflow graph (FDG) incrementally, subprogram by subprogram, by adding one or more nodes for each subprogram and each of its streaming atoms. As shown in Figure 10.1, there are three types of nodes represented by empty circles and empty or $\bowtie$-filled dashed circles. Every type of node is an abstraction of a set of *Flink* operators designed, developed, and properly combined for different purposes.

- Empty circles represent source or sink operators. The first is designated to consume the input stream from the data source and feed its elements into the FDG. The second is designated to produce the output stream collecting all the results coming from the execution of FDG.

- Empty dashed circles correspond to sets of operators designated for evaluating the semantics of specific streaming atoms. Therefore, the composition of a set of this type depends on the type of streaming atom at hand. Intuitively, it contains both predefined *Flink* operators for filtering, partitioning, transforming, and processing elements into the stream and custom operators for computing the semantics of the streaming atom under consideration. By means of *Flink* APIs, we defined a number of custom operators in order to implement the semantics of *in*, *always*, *count*, *at least*, and *at most*.

- $\bowtie$-filled circles correspond to a set of operators intended to join results of the evaluation of all streaming atoms in a subprogram and to set up the evaluation of such subprogram. This set contains custom operators we implemented for performing the low-level transformations needed for filtering and joining outcomes of linked circles.

An I-DLV-sr FDG always contains exactly two empty circles, one acting as a *sink* and the other acting as a *source*. In addition, a number of empty and $\bowtie$-filled dashed circles are used to compose the FDG according to the process described next. Each subprogram Γ_i is transformed in a *Flink* dataflow subgraph $G^{\Gamma_i} =< N, A >$ where N is the set of nodes and A is the set of arcs. N contains an empty dashed circle for each streaming atom a appearing in Γ_i and a single $\bowtie$-filled dashed circle. A contains a directed arc (e, j) from every

empty dashed circle $e \in N$ to the only $\bowtie$-filled dashed circle $j \in N$. Once G^{Γ_i} is created, it is connected to the part of the FDG already created by possibly using the following set of arcs:

- if Γ_i depends on a previous subprogram Γ_k (i.e., with $k < i$), every empty dashed circle e in G^{Γ_i} is connected to the $\bowtie$-filled dashed circle j of G^{Γ_k} with a directed arc from j to e. Intuitively, this provides the sub-graph G^{Γ_i} with the evaluation results of Γ_k required for evaluating streaming atoms in Γ_i.

- if Γ_i does not depend on any previous subprogram, every empty dashed circle e in G^{Γ_i} is connected to the *source* node s through a directed arc from s to e.

- if there exists no subprogram Γ_k subsequent to Γ_i (i.e., with $i < k$) such that Γ_k depends on Γ_i, the $\bowtie$-filled dashed circle j of G^{Γ_i} is connected to the *sink* node s via a directed arc form j to s.

This process continues with the next subprogram Γ_i+1 until all the subprograms are processed.

10.2.3 Program Rewriting

Given an I-DLV-sr program Γ, the *Execution Manager* rewrites each of its subprograms in plain ASP as described next. It starts rewriting Γ in an intermediate I-DLV-sr program $\Gamma^{r \mapsto r'}$ where each rule $r \in \Gamma$ is rewritten in a rule r' having the same head as r (i.e., $H(r') = H(r)$), and whose body, initially empty (i.e., $B(r') = \emptyset$), is constructed on the basis of streaming literals in $B(r)$. For each streaming literal $l \in B(r)$, if l has one out of the forms:

(i) $p(t_1, \cdots, t_n)$ **at least** X **in** $\{d_1, \cdots, d_m\}$;

(ii) $p(t_1, \cdots, t_n)$ **at most** X **in** $\{d_1, \cdots, d_m\}$;

(iii) *not* $p(t_1, \cdots, t_n)$ **at least** X **in** $\{d_1, \cdots, d_m\}$

(iv) *not* $p(t_1, \cdots, t_n)$ **at most** X **in** $\{d_1, \cdots, d_m\}$.

it is transformed to $p(t_1, \cdots, t_n)$ **count** Y **in** $\{d_1, \cdots, d_m\}$, where Y is a fresh variable. Then, this new atom is added to $B(r')$ along with a built-in atom b expressing a condition on Y according to the form of l; b is respectively:

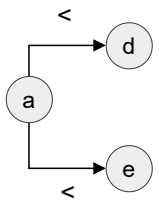


Figure 10.4: Stream Dependency and Component Graph of Γ_9 and $\Gamma_9^{r\mapsto r'}$ in Example 10.2.5

$Y >= X$ for *(i)*; $Y <= X$ for *(ii)*; $Y < X$ for *(iii)*; $Y < X$ for *(iv)*. If l has none of the forms in the list above, then it is added to $B(r')$ as it is.

Notice that this first rewriting only changes the operator of a streaming literal from *at least* and *at most* to *count*. Such a transformation leaves unchanged the streaming dependencies among predicates in rule heads; therefore, Γ and $\Gamma^{r\mapsto r'}$ have the same SDG, SCG, program splitting, and processing order.

Example 10.2.5 Let us consider the I-DLV-sr program Γ_9 reported below.

r_1 `a(Y) :- b(X) in {2}, Y=X+1.`
r_2 `d(Y) :- a(Y) at least X in {1,3,4}, c(X).`
r_3 `e(Y) :- not a(Y) at most X in {0,2,5}, c(X).`

Γ_9 is rewritten in an I-DLV-sr program $\Gamma_9^{r\mapsto r'}$:

r'_1 `a(Y) :- b(X) in {2}, Y=X+1.`
r'_2 `d(Y) :- a(Y) count Z in {1,3,4}, c(X), Z>=X.`
r'_3 `e(Y) :- a(Y) count Z in {0,2,5}, c(X), Z>X.`

Figure 10.4 reports the SDG of Γ_9 that coincides with its SCG as it does not contain streaming recursive rules. Moreover, if we also construct the SDG and SCG of $\Gamma_9^{r\mapsto r'}$, we can see that $G^{SD}_{\Gamma_9}$, $G^{SC}_{\Gamma_9}$, $G^{SD}_{\Gamma_9^{r\mapsto r'}}$, and $G^{SC}_{\Gamma_9^{r\mapsto r'}}$ are all the same.

The streaming precedence relationships among the nodes in SDG of Γ_9 and $\Gamma_9^{r\mapsto r'}$are: $\{a\} \prec \{d\}$, $\{a\} \prec \{e\}$, $\{a\} \approx \{d\}$. These induce two possible orderings $o_1 = \{\{a\},\{d\},\{e\}\}$ and $o_2 = \{\{a\},\{e\},\{d\}\}$. In both cases, the maximal macro-nodes we can build are $\{a\}$ and $\{d,e\}$; therefore, Γ_9 (respectively, $\Gamma_9^{r\mapsto r'}$) has two subprograms $\Gamma_{9\{a\}} = \{r_1\}$ (respectively, $\Gamma^{r\mapsto r'}_{9\{a\}} = r'_1$) and $\Gamma_{9\{d,e\}} = \{r_2, r_3\}$ (respectively, $\Gamma^{r\mapsto r'}_{9\{d,e\}} = \{r'_2, r'_3\}$) where $\{r_1\}$ (respectively, $\{r'_1\}$) is processed before $\{r_2, r_3\}$ (respectively, $\{r'_2, r'_3\}$). □

Subsequently, for each subprogram $\Gamma_i^{r\mapsto r'}$ of $\Gamma^{r\mapsto r'}$, the *Execution Manager*

produce a flat version $\Gamma_i'^{r\mapsto r'}$ of $\Gamma_i^{r\mapsto r'}$. In particular, it computes a mapping τ from each streaming atom a appearing in $\Gamma^{r\mapsto r'}$ to a fresh standard ASP atom a', and uses such a mapping to replace each occurrence of a in $\Gamma^{r\mapsto r'}$ with a'. τ is computed as follows. Each streaming atom $p(t_1,\ldots,t_n)\diamond$ **in** $\{d_1,\ldots,d_m\}$, with $\diamond \in \{$ **at least** c, **at most** c, **always** , **count** $c\}$, which is not in the degenerate form and so that c is a constant, is mapped to $p'(t_1,\ldots,t_n)$; each streaming atom $p(t_1,\ldots,t_n)$ **count** X **in** $\{d_1,\ldots,d_m\}$ where X is a variable, is mapped to $p'(t_1,\ldots,t_n,X)$; in both cases, p' is a fresh predicate name.

The *Subprogram Manager* will generate ground instances of fresh predicates on the basis of the evaluation of corresponding streaming atoms performed by the *Stream Manager*.

Example 10.2.6 Let us consider again the program Γ_8 in Example 10.2.1 with its subprograms defined in Example 10.2.4. Since Γ_8 does not contain any streaming literal that must undergo the first rewriting introduced before, we can produce the flat subprograms directly from Γ_8. A possible replacement mapping is τ= {`b(X) always in [2]` $\mapsto$ `b_aux1(X)`, `a(X) in [1]` $\mapsto$ `a_aux1(X)`, `b(X) at least 2 in [4]` $\mapsto$ `b_aux2(X)` }, and according to it:

- $\Gamma_{8\{a,b,e\}} = \{r_1, r_2, r_4\}$ is rewritten as:

```
r'1  a(X) :- b_aux1(X).
r'2  b(Y) :- a_aux1(X), Y=X+1, c(Y).
r4   e(X,Y) :- a(X), b(Y).
```

- $\Gamma_{8\{d\}} = \{r_3\}$ is rewritten as:

```
r'3  d(X) :- b_aux2(X).
```

□

10.3 *Stream Manager* and *Subprogram Manager*

This section provides the reader with additional details on the interaction between the *Stream Manager* and the *Subprogram Manager* during the execution phase. These two modules continuously collaborate to evaluate an I-DLV-sr program Γ, at every time point, over an input stream Σ, thus producing an answer stream O.

As depicted in Figure 10.1, the *Stream Manager* executes the scheduled operators in the *Flink* dataflow graph to evaluate Γ. In particular, for each subprogram Γ_i of Γ, firstly, it applies the operators scheduled to evaluate every streaming atom in Γ_i, then interacts with the *Subprogram Manager* to obtain the evaluation of the whole Γ_i. To achieve this, the *Subprogram Manager* mediates communications with the $\mathcal{I}^2$-*DLV* instances responsible for evaluating the flat program Γ'_i. A flat subprogram Γ'_i is always a normal and stratified w.r.t. negation ASP program. Therefore, $\mathcal{I}^2$-*DLV* always returns the only answer set of Γ'_i, which the *Subprogram Manager* properly interprets to produce the evaluation results of Γ_i on behalf of the *Stream Manager*.

Let us assume to be at the time point n. The *Stream Manager* handles the input stream $\Sigma = \langle S_0, \ldots, S_n \rangle$ and the stream $O = \langle O_0, \ldots, O_n \rangle$ where $O_0, \ldots, O_{n-1}$ collect the results obtained from the evaluations in previous time points and O_n, initially empty, is filled in while subprograms are evaluated.

More in detail, let Γ_M be the first subprogram in Ord_Γ. The *Stream Manager* executes the operators in the dataflow graph corresponding to the streaming atoms in Γ_M over Σ and O. Then, it passes to the *Subprogram Manager* S_n, O_n, and the set of holding streaming atoms H, i.e., those evaluated as true, at the time point n. The *Subprogram Manager*, in turn, performs the steps reported next:

1. according to the mapping τ, generates from H the set H' of holding ground instances of the fresh predicates introduced by the *Execution Manager*;
2. provides $S_n \cup O_n \cup H'$ as input to one of the $\mathcal{I}^2$-*DLV* instances dedicated to Γ_M, which incrementally evaluates Γ'_M over such input and computes the corresponding unique answer set A;
3. receives A (i.e., a set of ground (predicate) atoms) and adds A to O_n.

Subsequently, the *Stream Manager* takes control back, evaluates the next nodes in the dataflow graph, and performs the evaluation of the next subprogram in Ord_Γ by interacting with the *Subprogram Manager*. The process continues until all subprograms are evaluated according to Ord_Γ. The output of I-DLV-sr at n is $S_n \cup O_n$. Note that the smaller it is the number of subprograms, the smaller it is the number of iterations between $\mathcal{I}^2$-*DLV* and *Flink*: indeed, maximal macro-nodes are designed for limiting such interplay.

In case a subprogram contains some streaming-recursive rules R, the *Stream Manager* has to repeatedly evaluate the streaming atoms in R, i.e., every time

O_n is enriched by the *Subprogram Manager*. When no more new ground (predicate) atoms can be derived for the predicates in the heads of R, the process described above goes on with the next subprogram in Ord_P.

Note that the set $S_n \cup O_n$ actually coincides with the streaming model of Γ on Σ. Indeed, the streaming model is independent from the stratification of choice (see Section 9.3); thus, if we consider the stratification where each stratum is the smallest possible, then each subprogram consists of rules that might belong to one or more consecutive strata; when evaluating a subprogram Γ_M, *$\mathcal{I}^2$-DLV* by design takes into account dependencies among the strata and processes rules in Γ_M accordingly. Thus, the output computed for Γ_M coincides with the output that we would obtain when evaluating, one after the other, the strata contained in Γ_M as well as in all subprograms preceding Γ_M, according to the definition of streaming model. In other words, the output computed for Γ_M coincides with the streaming model of the I-DLV-sr program constituted by all subprograms up to Γ_M on Σ. It is worth noting that the *Stream Manager* implements a queuing mechanism for the subtasks that have to be performed for evaluating Γ on Σ exploiting *Flink* APIs. Thanks to this, I-DLV-sr is able to manage backpressure [89], guaranteeing no loss of data and, thus, the correctness of the output.

Example 10.3.1 Consider Γ_8 of Example 10.2.1 again. Figure 10.5 illustrates the evaluation process of Γ_8 by referring to the general architecture in Figure 10.1. The *Execution Manager* splits Γ_8 and identifies the processing order for its subprograms. Suppose as shown in Example 10.2.4 that the splitting is $\Gamma_{8\{a,b,e\}} = \{r_1, r_2, r_4\}$ and $\Gamma_{8\{d\}} = \{r_3\}$, and that $\Gamma_{8\{a,b,e\}}$ precedes $\Gamma_{8\{d\}}$. The *Execution Manager* accordingly creates the *Flink* dataflow graph and passes it to the *Stream Manager*. Moreover, it rewrites both $\Gamma_{8\{a,b,e\}}$ and $\Gamma_{8\{d\}}$ in plain ASP programs (respectively, $\Gamma'_{8\{a,b,e\}}$ and $\Gamma'_{8\{d\}}$) as shown in Example 10.2.6, and then it passes the rewriting information along with the program splitting to the *Subprogram Manager*. Subsequently, the *Subprogram Manager* creates at least one instance of *$\mathcal{I}^2$-DLV* for each subprogram to which the related plain version is submitted.

As we can see from the figure, in our example, the *Flink* dataflow graph contains three empty dashed circles for the operators needed for evaluating streaming atoms in $\Gamma_{4\{a,b,e\}}$ and $\Gamma_{4\{d\}}$, and two $\bowtie$-filled ones that receive results from the linked circles and forward them to *Subprogram Manager*.

Now, let us intuitively discuss the computation workflow for evaluating Γ_8

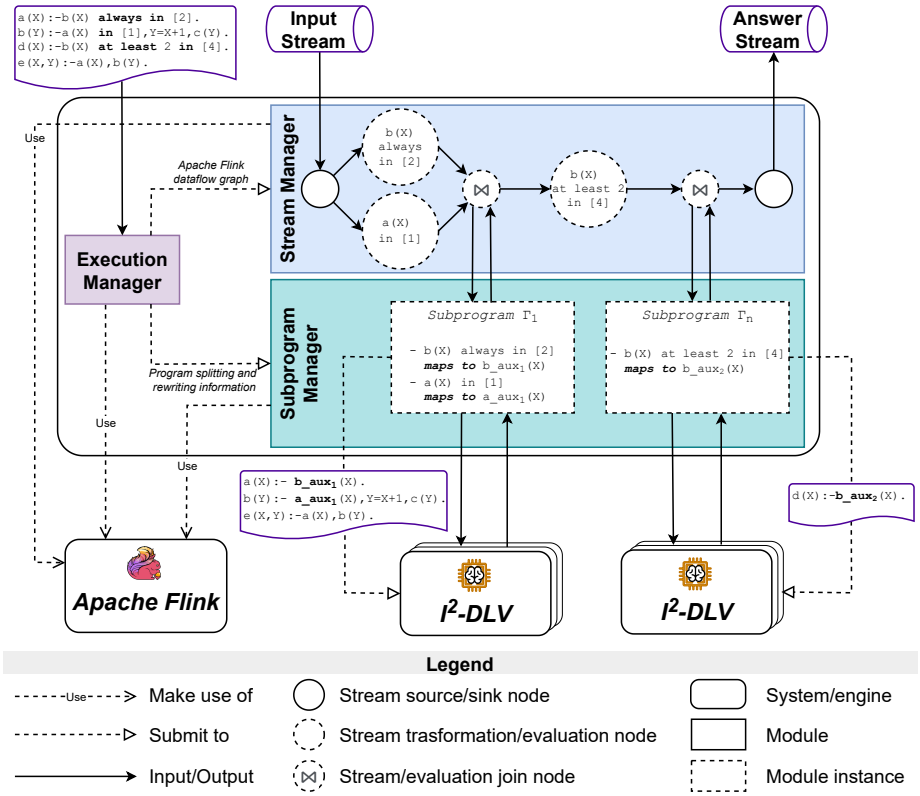


Figure 10.5: The high-lever architecture of I-DLV-sr on an input program

while executing the *Flink* dataflow graph. We assume to be at the n-th time point and that $\Sigma = \langle S_0, \ldots, S_n \rangle$ is the input stream, while $O = \langle O_0, \ldots, O_n \rangle$ with $O_n = \emptyset$ is the stream iteratively built so far.

The *Stream Manager* first evaluates the empty dashed circles relative to $\Gamma_{4\{a,b,e\}}$, and then, the *Subprogram Manager* is required to take into account $\Gamma_{4\{a,b,e\}}$ hence, producing the ground instances for `a_aux1` and `b_aux1` and properly invoking $\mathcal{I}^2$-*DLV*. The result received by the *Subprogram Manager* is forwarded back to the *Stream Manager* that updates O_n. Such loop between *Stream Manager* and *Subprogram Manager* for processing of $\Gamma_{4\{a,b,e\}}$ continues until nothing new can be inferred for the predicates $\{a, b, e\}$ and only after $\Gamma_{4\{d\}}$ is processed. In the end, O_n is emitted in the output stream. □

Chapter 11

Implementation

This chapter provides some implementation details of I-DLV-sr. Section 11.1 describes how *Flink* operators have been extended and used for evaluating I-DLV-sr programs over input streams. Section 11.2 provides an overview of the significant enhancement and optimization introduced in I-DLV-sr since its first release.

11.1 Custom Stateful *Flink* Operators for Evaluating I-DLV-sr Programs

As we introduced in Chapter 10, we used *Flink* APIs to develop custom operators for implementing the semantics of the streaming atoms (Chapter 9) and the evaluation of I-DLV-sr subprograms (Chapter 10). Therefore, we also designed how such operators must be combined to form a dataflow that properly evaluates an I-DLV-sr program on an input stream. In particular, we designed a chain (i.e., a sequence) of custom operators for each type of streaming atom and subprogram (i.e., normal and streaming recursive). These chains are then combined in a dataflow according to the process described in Subsection 10.2.2.

11.1.1 Deal With Normal I-DLV-sr Subprograms

Let Γ_i be a normal I-DLV-sr subprogram (i.e., that is not streaming recursive); we handle its evaluation via a number of linked chains of operators: one chain for each streaming atom a appearing in Γ_i, which computes the semantics of a,

and one chain for Γ_i itself, which collects results from other chains and evaluates Γ_i. We recall that *Flink* possibly executes operators in the chain simultaneously and on different machines (Subsection 3.1.1). Therefore, this approach allows us to take maximum advantage of its parallel and distributed nature, allowing for low evaluation times.

The central operator of the chains designed to evaluate the semantics of streaming atoms is a *Window* operator (see Subsection 3.2.3). The *window* handled by this operator is a sliding window. The size of this window is the one that the user specifies via the input program, and its slide is always of 1 *time unit* (i.e., 1 millisecond, 1 second, 1 minute, or 1 hour, according to the system configuration; see Subsection 11.2.3). The *evaluation function* changes on the basis of the nature of the streaming atom at hand. Indeed, a specific evaluation function has been designed for each available streaming atom to implement the related semantics. If the lookup set of a streaming atom is not a window (see Subsection 9.2.2), the window handled by the *Window* operator has a size equal to the largest number contained in the lookup set. Therefore, when computing the semantics, the *evaluation function* will take care of considering only inputs received at the time points of interest.

Example 11.1.1 Let us consider the following streaming atoms: `a` **`always in [`***`3 sec`***`]`** and `a` **`always in`** {*0, 2, 4*}, and assume that time points are in seconds.

The *Window* operator associated with the first streaming atom will handle a sliding window of size 3 seconds and slide 1 second. Given a bucket of events, the window evaluation function will compute the *always* semantics on all the contained events.

The *Window* operator associated with the second streaming atom will handle a sliding window of size 4 seconds and slide 1 second. In this case, given a bucket generated by a window whose upper bound is *ub*, the window *evaluation function* will compute the *always* semantics only on the events within the bucket associated with the time points: $ub - 0$, $ub - 2$, $ub - 4$. □

The central operator of the chain designed to evaluate normal subprograms is a *ProcessFunction* operator (see Subsection 3.2.2) that is associated with a custom *processing function*. This function takes as input a stream containing results from the chains mentioned above, namely, all the chains that evaluate the streaming atoms contained within the subprogram under consideration. Each element in this stream corresponds to the set of streaming atoms hold-

ing at a specific time point t. The output stream of this operator contains the new derivations resulting from evaluating the handled subprogram at each considered time point. The *processing function* gets these new derivations by communicating with the *Subprogram Manager*. In particular, for each element within the input stream, the function sends the set of holding atoms at the time point t to the *Subprogram Manager*. The latter, in turn, evaluates the considered subprogram over this set and replies with the new derivations for the time point t (see Section 10.3).

Joining Multiple Input Streams. Typically, the *Window* and the *ProcessFunction* operators described above must evaluate their functions on events coming from different sources. Such sources are other operators in the dataflow on which they depend. Consequently, strategies for joining two or more streams are required to provide them with a single input stream. To achieve this, where appropriate, we use a sequence of custom *CoGroup* operators (see Subsection 3.2.4). This sequence is inserted into the chains before the *Window* and *ProcessFunction* operators, as described next.

Suppose we have a normal subprogram Γ_i that depends on a set of subprograms M such that $|M| > 1$ and for each $\Gamma_k \in M$, $k < i$. For each streaming atom in Γ_i, the associated *Window* operator depends on all the *ProcessFunction* operators related to the subprograms in M. In order to join their output streams into a single one, we add $|M| - 1$ custom *CoGroup* operators into the chains designated for evaluating the streaming atoms of Γ_i; specifically, before their *Window* operators. In general, $|M| - 1$ *CoGroup* operators are required because this type of operator allows joining only two streams at a time. Therefore, it is necessary to start joining the first pair of streams and continue joining each subsequent stream with the stream resulting from previous *CoGroup* operations. We designed the used *CoGroup function* to filter only elements of the streams that are useful for the computation of the window evaluation functions at hand.

Similarly, let Γ_i be a normal subprogram containing n streaming atoms. The *ProcessFunction* operator in charge of evaluating Γ_i depends on all the output streams of the *Window* operators associated with the streaming atoms contained in Γ_i. As before, to provide this operator with a single input stream, it is preceded by $n - 1$ *CoGroup* operators into the chain. We designed the *CoGroup* function to filter only elements necessary for evaluating Γ_i. Additionally, it appropriately formats the output elements to be sent to the *Subprogram*

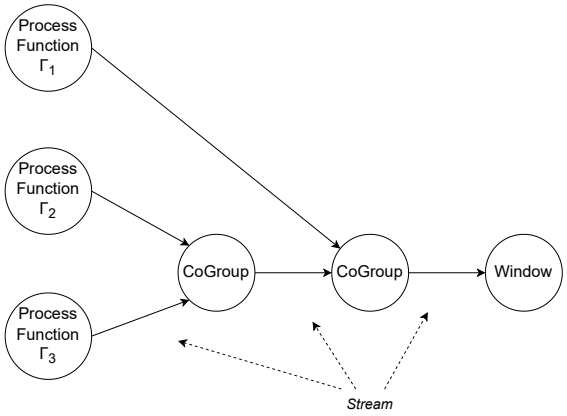


Figure 11.1: An example of joining multiple streams

Manager.

In both cases, the window used for the *CoGroup* operator is a *tumbling window* with a size of 1 time unit (i.e., 1 millisecond, 1 second, 1 minute, 1 or hour, according to the system configuration; see Subsection 11.2.3). Moreover, during the *CoGroup* computation, events are grouped according to their timestamp; in other words, the event time is used as the grouping key.

Example 11.1.2 Suppose we have a subprogram that depends on three other subprograms and must evaluate one of its streaming atoms. Figure 11.1 depicts the part of dataflow that provides the related *Window* operator with the input stream. In this example, to join the output streams coming from the *ProcessFunction* operators of the previous subprograms, two *CoGroup* operators are necessary. □

11.1.2 Deal With Streaming Recursive I-DLV-sr Subprograms

Streaming recursive programs include recursive rules over predicates appearing in non-degenerate streaming atoms (Subsection 10.2.1). These rules introduce a cyclic dependency between windows operators involved in the recursion. Since *Flink* only allows the definition of acyclic dataflows, this type of dependency cannot be transformed into a dataflow using the process described in the previous subsection. Therefore, we need to design an alternative approach to handle cyclic dependencies.

In order to deal with streaming recursive subprograms, we made compromises between language expressiveness and evaluation efficiency. Specifically, we designed a chain of operators where the streaming recursive subprogram and all the contained streaming atoms are evaluated by the same operator iteratively and sequentially on a single machine.

Let Γ_i be a streaming recursive program and A be the set of its streaming atoms. A single *Window* operator performs all the streaming computation needed to evaluate Γ_i. This operator controls a sliding window with a slide of 1 *time unit* and size equal to the largest window appearing in the streaming atoms in A. In such a way, we guarantee that all the windows of these streaming atoms are covered.

Since we deal with a streaming recursive subprogram, several subsequent evaluations may be required to derive all the possible events for a time point. Therefore, we defined an *evaluation function* that iteratively evaluates both the streaming atoms in A and Γ_i multiple times until new events can be derived. This function works according to the process described next.

When a new bucket of events is generated, the *evaluation function* is invoked. First, it starts computing the semantics of each streaming atom in A, thus generating the set of holding atoms H. Then, the function communicates with the *Subprogram Manager* to obtain the evaluation of Γ_i on the basis of H. This process continues until the *Subprogram Manager* returns new derivations. Due to the streaming dependencies among the streaming atoms involved in the recursion, the evaluation function evaluates streaming atoms in A iteratively by considering the events coming from: *(i)* the bucket currently received; *(ii)* the evaluations currently performed; *(iii)* the evaluations performed on previously received buckets.

Example 11.1.3 Let us consider the streaming recursive subprogram composed only of the following rule: `a(X):- a(Y) in [2], X=Y+1, X<=3.` If we consider the input stream $\Sigma_3\langle\{\ \{a(1)\}_0\}\rangle$, accordig to the I-DLV-sr semantics, the output stream is $\Sigma_4\langle\{\ \{a(1), a(2), a(3)\}_0\}\rangle$.

In order to evaluate this subprogram, a *Window* operator with a window of size 2 is used. If we evaluate the streaming atom `a(Y) in [2]` only on the input stream of this operator, we will derive only `a(2)`. In order to also derive `a(3)`, we have to re-evaluate the streaming atom by taking into account the new derivation `a(2)`. Notice that this is not part of the input stream but results from computations performed by the operator at hand.

Eventually, if we would also evaluate this subprogram at the time point 1, we must take into account also `a(2)` and `a(3)` when computing the semantics of `a(Y)` **`in [`***`2`***`]`** as they fall within its window. □

The *evaluation function* of the *Window* operator above shares the events that it derives during an invocation with subsequent invocations using an internal state. Each time the function is executed, it dynamically updates this state by adding new derivations obtained during the current execution and deleting old derivations that are no longer useful for future derivations.

If necessary, the input stream of this operator is constructed using a chain of *CoGroup* operators as described in the previous subsection.

11.1.3 Dealing With Idle Operators

I-DLV-sr relies on the event time semantics; therefore, the clock of the system progresses on data arrival according to the watermark mechanism described in Subsection 3.3.1. Thus, operators in the dataflow only are triggered when they receive specific watermarks from operators on which they depend.

Let us consider two connected operators. If the sending operator stops to produce output events (i.e., it is idle), the receiving operator does not receive any watermark. Consequently, the latter will no longer be triggered until the first operator wakes up. This situation is very likely to happen in our case. For instance, if the chains of operators that compute the semantics of streaming atoms do not derive any holding streaming atom, or if a subprogram has no solutions for a while. In these cases, the system waits until something is generated to emit previous outputs. However, it should be able to emit its answer as soon as it is complete, even if this is empty (i.e., an I-DLV-sr program has no solution at a time point).

To overcome this situation, we use an approach inspired by the watermark mechanism of *Flink*. Specifically, we generate a *special event* at the *source* operator for each considered time point t. This special event is associated with t and emitted into the dataflow. Therefore, it is forwarded from each operator to the next ones. Operators add this event into their output stream even if they do not generate any actual event. In such a way, they notify subsequent operators that they performed their computation over events received until the time point t. This special event will trigger the watermark update as well as the computation of the receiving operator.

11.2 Enhancements and Optimizations

Since its first release, I-DLV-sr has undergone continuous changes to improve its performance and usability in real-world domains. Next, we describe significant enhancements and optimizations introduced from the first to the second release of the system.

The overall computational process performed by I-DLV-sr to evaluate a given input program over a data stream is designed in order to take advantage of both *Flink* parallel and distributed nature and $\mathcal{I}^2$-*DLV* incremental evaluation. According to the program splitting discussed in Chapter 10, I-DLV-sr identifies all the activities for evaluating an input program properly; then, it defines multiple tasks to be executed in parallel, each one incorporating an ordered sequence of activities. Such activities correspond to custom *Flink* operators that I-DLV-sr employs to form the dataflow graph responsible for evaluating the I-DLV-sr program on an input stream. We recall that *Flink* operators can have multiple instances and run in parallel in different threads. Note that, typically, operators depend on the results from others, requiring related threads to communicate and synchronize. Consequently, it turns out that a kind of trade-off between the granularity of tasks and the dependencies among them must be found. Indeed, the smaller the tasks in terms of the number of activities, the higher the number of operators in the dataflow; consequently, the higher can be the overall level of inter-dependency among tasks. Such dependency level directly influences performance since synchronization and communication among threads are expensive. The cost of such operations depends on the used communication pattern and how *Flink* distributes operators among the available machines (Subsection 3.1.1).

In the first I-DLV-sr version, much effort was spent on designing strategies to define tasks at a finer granularity to increase parallelism as much as possible. The second I-DLV-sr version has been deeply re-engineered. Conversely to the first version, it comes with new strategies that induce tasks at a coarser granularity in some points, reducing synchronization and communication duties.

11.2.1 A New Watermark Assignment Strategy

One of the main changes introduced in the second I-DLV-sr version regards the strategies adopted for managing *idle operators* (Subsection 11.1.3) and *watermarks* (Subsection 3.3.1).

Both I-DLV-sr versions make use of a *source* operator that transforms input row data in events with a form suitable to be processed by the I-DLV-sr and handles watermarks. Besides how they implement such operators, these two versions also differ in how they manage the production of the special events (used to deal with idle operators) and assign watermarks to them. The first I-DLV-sr version adopts a chain of parallel operators; this emits special events and watermarks once input events enter the dataflow. On the contrary, the second I-DLV-sr version manages both tasks directly within the *source* operator while reading inputs from the data source, i.e., before events enter the dataflow.

Management of Idle Operators and Watermarks in the First I-DLV-sr version. In the first version of I-DLV-sr, the source operator has more than one subtask. In particular, *Flink* set its parallelism automatically on the basis of the machine that runs the system. Therefore, each subtask receives a partition of the input stream and handles its own local watermark. All events produced by the *source* operator are then passed to a custom operator named *StreamFiller*, which emits the *special events*. In particular, while processing input events, this operator produces one special event for each time point between the first and the last event time registered. The *StreamFiller* operator emits special events also for those time points without input events associated. As in the case of the *source* operator, the *StreamFiller* operator has multiple subtasks.

Given that both *source* and *StreamFiller* operators are parallel, the partitions of events produced by the *source* subtasks must be broadcasted to all the *StreamFiller* subtasks. This introduces non-determinism in the order in which events arrive at the *StreamFiller* operator. Therefore, in order to properly produce *special events*, the *StreamFiller* operator must restore the temporal order of input events; specifically, it must order them according to their time point.

It is worth noting that the *StreamFiller* operator generates special events after the *source* operator has assigned watermarks to input events. Consequently, *Flink* internally associates them with the watermark held by the *StreamFiller* operator at the moment they have been emitted. Event times and watermarks of *special events* may thus not correspond at this stage. In order to solve this mismatch, *special events* pass through an additional operator that performs the watermark assignment process again.

As can be imagined, this approach has two main issues. On the one hand, stream broadcasting and ordering are very expensive operations. On the other hand, watermark needs to be assigned twice.

Management of Idle Operators and Watermarks in the Second I-DLV-sr version. The second version of I-DLV-sr, carries out all the operations described above within the *source* operator, whose maximum parallelism is set to 1. Therefore, the *source* operator generates the s*pecial events* and assigns them watermarks in real time as actual events are processed. This approach allows for avoiding wasting time on communication and duplicated operations.

11.2.2 An Improved Interplay Between *Flink* and I^2-DLV

Since the first release of I-DLV-sr, the interplay between *Flink* and $\mathcal{I}^2$-*DLV* has been significantly improved. In the first I-DLV-sr version, different threads interact with a unique $\mathcal{I}^2$-*DLV* instance. The latter represents a shared resource to be accessed in mutual exclusion in order to preserve semantics. The second I-DLV-sr version uses instead a number of instances of $\mathcal{I}^2$-*DLV* at least equal to the number of subprograms individuated (see Chapter 10). In particular, each thread in charge of executing the task (i.e., the dataflow operator) relative to the evaluation of a subprogram has its own $\mathcal{I}^2$-*DLV* instance. Incrementality is still preserved but fruited at the subprogram level rather than at the program level. Indeed, every $\mathcal{I}^2$-*DLV* instance always considers the same subprogram over different inputs. The goal is again to reduce the synchronization duties among threads, as they can directly communicate with their own reasoner instance without waiting for others to finish.

11.2.3 A New Timeline Management

The second version of I-DLV-sr comes with new timeline management. In particular, it allows the user to set the reasoning granularity, i.e., if it must reason in milliseconds, seconds, minutes, or hours. This feature allows for avoiding useless computations when intermediate results are not required. It changes how the system clock progresses and the frequency with which windows and the other operators in the dataflow are triggered and evaluated.

The possibility to set the granularity of the system timeline is not available in the first version of I-DLV-sr.

Example 11.2.1 Let us consider the following I-DLV-sr program, and suppose we are interested in receiving its solution at each minute.

r_1
```
a(X) :- b(X) in [2 min].
```

If the system is asked to reason in terms of milliseconds, it emits the solution of this program at every millisecond. Therefore, even if we receive input events at each minute, it evaluates the program at all the milliseconds within that minute. This leads to a large number of useless computations that reflect on the execution of the *Flink* dataflow graph. For instance, let us focus only on the *Window* operator in charge of evaluating the semantics of `b(X) in [2 min]`. This handles a sliding window of size 2 minutes that slides of 1 millisecond at a time (i.e., the *time unit* of the system timeline). Therefore, to produce the answer for one time point of interest (1 minute), 60.000 windows are triggered, thus firing the computation of the *evaluation function* 60.000 times. This number drops to 60 if the reasoning granularity is set to seconds and to 1 if it is set to minutes. □

Chapter 12

Usage

This chapter provides an overview of the usage of I-DLV-sr. Section 12.1 describes the format of the input stream accepted by the system. Section 12.2 explains how users can execute the system.

12.1 The Input Stream

I-DLV-sr relies on the event time semantics. Therefore, it requires that the input stream consists of chronologically ordered elements. The accepted input has the form:

$$\textit{timestamp}\ \texttt{p}_1\texttt{;}\ \cdots\texttt{;}\ \texttt{p}_n\texttt{;}$$

where *timestamp* indicates a time point and $\texttt{p}_1\texttt{;}\ \cdots\texttt{;}\ \texttt{p}_n\texttt{;}$, with $n \geq 0$, is a list of ground predicate atoms. This list indicates that the contained predicate atoms are true at that time point.

timestamp can have either *textual* or *numeric* format. In the first case, it must be a *datetime* string formatted according to the pattern: "`yyyy-MM-ddTHH:mm:ss.SSS`", where milliseconds (.SSS) can be omitted. In the second case, it must be an integer *number* indicating a timestamp in milliseconds, seconds, minutes or hours.

By default, the system accepts time points in *seconds* expressed using the textual format. The user can change this setting by command line via the options `--t-unit` and `--t-format`. The first indicates the time unit of time points, i.e., the timeline granularity of the system; the second specifies their formatsee Subsection 12.2.3 for details on the usage of these options). Note

that once the system is running, it accepts only timestamps with the same format; an execution error is raised otherwise.

Example 12.1.1 Let us consider the portion of the input stream reported next.

```
2020-05-26T12:16:18 a; a(1); c(3);
2020-05-26T12:16:23 b(2,1);
2020-05-26T12:16:25 e("fg");
2020-05-26T12:16:17 a(0); c(1);
```

The last line reports an out-of-order set of events that is not accepted by the system.

An example of a well-formed input stream follows.

```
2020-05-26T12:16:18 a; a(1); c(3);
2020-05-26T12:16:23 b(2,1);
2020-05-26T12:16:24
2020-05-26T12:16:25 e("fg");
```

Note that time points can be missed.

□

12.1.1 Input Streams with Duplicate Timestamps

I-DLV-sr can be configured to collect fragmented inputs for a time point. In such a way, the data source can send the events of a given time point via several consecutive messages. Optionally, it can communicate the end of such a sequence of messages by adding the special event `@end` to the last one. The only restriction is that each message must contain the timestamp of the relative time point.

The user can enable the system with this capability via the command line option `--t-duplicate`. In this case, I-DLV-sr evaluates the input program at a time point t only when it is sure that all the events associated with t have been received. Therefore, the evaluation at t triggers in one of the following cases:

- I-DLV-sr receives one event for the time point t' with $t' > t$;
- the source sends an `@end` event associated with t.

Example 12.1.2 Let us consider the following stream featuring fragmented inputs for some time point.

```
2020-05-26T12:16:18 a;
2020-05-26T12:16:18 b; @end;
2020-05-26T12:16:19 a;
2020-05-26T12:16:19 c;
2020-05-26T12:16:24 b;
2020-05-26T12:16:24 @end;
```

Here, the input program is evaluated for the time point `2020-05-26T12:16:18` as soon as the message `2020-05-26T12:16:18 b; @end;` is received, while for the time point `2020-05-26T12:16:19` it is evaluated only after the message `2020-05-26T12:16:24 b;` is received. □

12.2 Running the System

The latest release of I-DLV-sr for Linux x86-64 is available at the dedicated repository [43]. To run the system, Java 11 is required; optionally, Graphviz[1] is required to export graphs computed in the configuration phase (i.e., the Stream Dependency Graph and the Stream Componen Graph).

I-DLV-sr can run in two different modes: **socket-based** and **file-based**.

12.2.1 Socket-based

In the socket-based mode, I-DLV-sr reads the input stream from a source socket. Once running, it waits for input events until the socket is open.

To run the system in the socket-based mode, the user must perform the following steps:

1. Start up the service providing the input stream. For instance, one can run *netcat* from a command line to start a socket based-service: `nc -l <ip_address> <port_number>`.

2. Start up I-DLV-sr as follows:

```
java -jar I-DLV-sr.jar --program=<path_to_the_program>
   --hostname=<ip_address> --port=<port_number>
```

Note that the port number and the IP address of the socket-based service and I-DLV-sr must coincide. The port number and the IP address can change at will; by default, in I-DLV-sr `hostname=localhost` and `port=9000`.

[1] `https://graphviz.org/download/`

12.2.2 File-based

In the file-based mode, I-DLV-sr reads the input stream from a log file. Once running, it reads all input events up to the end of the file.

In the file-based mode, one should just execute the following command.

```
java -jar I-DLV-sr.jar --program=<path_to_the_program>    --
log=<path_to_the_input_stream_file>
```

12.2.3 Command-line Options

A lot of I-DLV-sr features can be easily configured via the command line. Next, we report the set of available options.

General Options

- `--hostname=<ip>` set the source IP address (default: "localhost")
- `--port-number=<int>` set the source port number (default: 9000)
- `--program` (required) path to the program file
- `--log` path to the input log file
- `--py-script=<../script/path.py[, ../script`$_n$`/path.py]>` paths to the python scripts containing the definition of the external atoms
- `--parallelism=<int>` set the default parallelism of the execution environment, i.e., a default parallelism for all operators within the Apache Flink dataflow (default: the number of processors of the machine that runs the system)
- `--windows-unit=<sec|min|hrs>` assign the specified timeunit to streaming atoms whose timeunit is not explicitly declared within the program
- `--complete-windows` disable the evaluation of "always" streaming atoms over partial windows, i.e., windows whose lower bound is less than the first event's timestamp
- `--export-graphs` export the stream dependency and component graphs

Time Point Options

- `--t-format=<msec|sec|min|hrs>` set the time unit to be used to interpret the timestamps contained in the input stream properly. By default, the system accepts timestamps in string format, i.e., a datetime following the pattern `yyyy-mm-ddThh:mm:ss.SSS` where the part `.SSS` can be omitted. Use this option if, instead, time points are in numeric format.
- `--t-unit=<msec|sec|min|hrs>` set the time unit of the system timeline, i.e., at which granularity the system has to reason (default: sec)
- `--now-format=<sec|min|hrs|datetime>` set the format to be used for assigning values to the **`@now`** term (default: datetime, i.e., a string following the pattern `yyyy-mm-ddThh:mm:ss.SSS` where the part `.SSS` is omitted when it is not required)
- `--t-duplicate` allows handling input streams having time points appearing multiple (consecutive) times, provided that they are chronologically ordered.

Log Options

- `--print-extended-log` print an extended version of the output log. It is possible to set the verbosity level:
  - `=0`: maximal verbosity (default)
  - `=1`: minimal verbosity
- `--print-reasoning-info` print an extended version of the output log containing information about the reasoning as well as related statistics. It is possible to set the verbosity level:
  - `=0`: maximal verbosity (default)
  - `=1`: minimal verbosity
- `--print-operators-info` print an extended version of the output log containing information about the evaluation of streaming atoms as well as related statistics. It is possible to set the verbosity level:
  - `=0`: maximal verbosity (default)
  - `=1`: minimal verbosity

- `--print-rewriting` print the rewritten program
- `--verbose` set the logger to verbose mode, i.e., enable all the printing options. It is possible to set the verbosity level:
  - `=0`: maximal verbosity (default)
  - `=1`: minimal verbosity
- `--help` print the help

Chapter 13

Modeling Capabilities

This chapter shows the modeling capabilities of I-DLV-sr. In particular, Sections 13.1 and 13.2 illustrate how I-DLV-sr can be used to model several Smart City applications, and to control caching strategies in Content-Centric Networks, respectively.

13.1 Smart Cities

A smart city [80, 84] is an urban area in which data collected via modern digital technologies are strategically exploited to implement efficient management of infrastructures, resources, and services, aiming to improve the life quality of residents.

First visions of smart cities were static and infrastructure-centered (i,e., relying on installation and management of edge devices coupled with analytics over such data), thus leaning towards single-purpose, vertical solutions. Recently, more complex visions have been proposed based on rich ensembles of physical, human, and ICT infrastructures and tight integration of research and technological solutions from different areas [49, 56, 81]; across such areas (e.g., Internet of Things, Social Computing, Artificial Intelligence, etc.) Stream Reasoning plays a crucial role.

This section presents several smart city scenarios where I-DLV-sr can be effectively used to address related real-world applications and problems.

13.1.1 Intelligent Energy Management

Sustainability is one of the main goals pursued in smart city contexts [81]. To this end, modern cities are characterized by increasingly widespread employment of renewable energy resources to supply the energy power demand of citizens

In this context, solar photo-voltaic systems are particularly attractive renewable energy resources thanks to their high efficiency and low environmental impact. However, just like classic electricity generation systems, they are susceptible to various problems. For instance, faults can involve system components like photo-voltaic modules, cables, protections, or inverters. Therefore, to ensure high-performance energy production, faults must be promptly detected and diagnosed [120]. In this respect, knowledge-based real-time energy monitoring, diagnostics and controlling technologies and their application for intelligent energy management represent a good starting point for achieving sustainable smart cities [90].

Intelligent Photo-voltaic System Monitoring

Suppose we need to build an Intelligent Monitoring System (IMS) for a photo-voltaic system (PVS) to detect malfunctions promptly. Without going into technical details, for the sake of simplicity, let us suppose that the PVS is composed of a grid of solar panels interconnected via solar cables and that each panel is provided with a sensor. The latter measures the amount of energy produced by the associated panel and continuously sends data to the IMS.

Each panel continuously produces energy to be transferred to a Central Energy Accumulator (CEA). Specifically, the CEA exploits the network of solar cables in order to accumulate the energy produced by the whole PVS. Therefore, it must be able to reach all the working panels directly or indirectly (e.g., via a path between neighbor panels across the grid).

Let us assume that a time point corresponds to a second. A panel is working if it produced an amount of energy greater than a given threshold within the last 4 seconds. A working panel should always be reachable by the CEA (i.e., there exists a path of working panels linking it to the CEA); therefore, if some unreachable working panels have been detected more than 2 times in the last 3 seconds, an alert must be raised for an identified malfunction. Furthermore, the IMS must request a maintenance intervention if failure is continuously observed for 5 seconds.

I-DLV-sr Modeling. The scenario described in the previous section can be modeled via an I-DLV-sr program P_{pvs}, as reported next.

```
r1 workingPanel(P) :- energyThreshold(Et),
       energyDelivered(P,W) at least 1 in [4 sec], W>=Et.
r2 reachable(cea,P2) :- link(cea,P2), workingPanel(P2).
r3 reachable(P1,P3) :- reachable(P1,P2), link(P2,P3), workingPanel(P3).
r4 unlinked :- workingPanel(P), not reachable(cea,P).
r5 regularFunctioning :- unlinked at most 2 in [3 sec].
r6 alert :- not regularFunctioning.
r7 callMaintenance :- alert always in [5 sec].
```

The predicates `link` and `energyThreshold` represent the PVS configuration and the threshold defining a working panel; these data do not change, i.e., they are part of the background knowledge. The predicate `energyDelivered` represents the amount of energy produced by each panel; at each time point, the current values are sent to the IMS, thus producing a stream.

A panel is defined as working by rule r_1, if it has transmitted an amount of energy greater than the threshold at least once in the latest 4 seconds. r_2 and r_3 recursively define the set of reachable working panels starting from the CEA. r_4 detects if there are `unlinked` working panels, and r_5 defines proper functioning by checking that the atom `unlinked` appeared no more than two times in the last 3 seconds. Eventually, r_6 raises an alert if there is not a regular functioning, and r_7 asks to call maintenance if an alert has been raised in all the latest 5 seconds.

13.1.2 Intelligent Transport Systems

Smart Mobility is one of the hot topic in smart cities; it is quite a challenging task that the Public Administration is trying to embrace, aiming at improving the Quality of Life (QoL) of its residents. Specifically, smart mobility collects approaches, decisions, and technologies enabling efficient and sustainable transportation of people and goods in urban areas. Most proposed solutions rely on IT, such as vehicle navigation systems, e-parking, e-ticket, info-mobility signalization, demand-responsive transport, car sharing, bike sharing, and public transport live tracking [127]. In this context, SR-based solutions help implement advanced systems for traffic management, traveler information, and public transportation.

Underground Train Traffic Monitoring

Let us imagine we want to build a monitoring system for the underground trains in the city of Milan. One of its components might be a simple control system that warns passengers when traffic regularity is broken in different ways (i.e., mild/grave irregularity). This would allow, e.g., to properly mark each station on a map within a mobile/web app with such information. In this example, we suppose that a time point corresponds to a minute.

Given a station, passengers expect to see a train stopping every 3–6 minutes during rush hours. If this does not happen, the traffic regularity is considered to be broken. Therefore, the system must raise a *mild alert* if, within the last 30 minutes, a traffic irregularity has been observed from 2 to 5 times; if the number of detected irregularities is more than 5, then a *severe alert* must be aroused.

I-DLV-sr Modeling. The I-DLV-sr program that models the control system (executed at each station) previously described is the following.

```
r1  irregular  :- trainPass, trainPass at least 1 in {1,2}.
r2  irregular  :- not trainPass in [6 min].
r3  #temp numAnomalies(X) :- irregular count X in [30 min].
r4  mild_alert   :- numAnomalies(X), X>2, X<=5.
r5  severe_alert :- numAnomalies(X), X>5.
```

Rules r_1 and r_2 detect irregularities: trains arriving too early or too late. Rule r_3 counts the number of irregular situations in the last half an hour, producing an instance of the `numAnomalies`; r_4 and r_5 raise the proper warning. Note that `numAnomalies` is an auxiliary predicate whose instance is used to determine irregularities by watching only at the current time point; thus, it is marked as **#temp**. In such a way, its instance only contributes to the current streaming model and is no longer considered in the next time points.

Traffic Management for Cooperative Intelligent Transportation Systems

Within the scope of the *Stream Reasoning Workshop 2021*[1] the *Stream Reasoning Hackathon 2021* (SR Hackathon 2021)[2] took place. The Hackathon was

[1] `https://streamreasoning.org/events/srw2021/` (url date: 18/01/2022)

[2] `https://streamreasoning.org/events/stream-reasoning-hackathon-2021/` (url date: 18/01/2022)

designed as a "model and solve" challenge on scenarios concerning the implementation of Intelligent Transport Systems (ITS). For additional details, we refer the reader to the paper [117] and the official GitHub repository [118], where an overview of both the proposed scenarios and the hackathon organization is available.

This section shows how I-DLV-sr can deal with the proposed tasks. First, we introduce the provided static and streaming models; then, we describe the I-DLV-sr modeling for some selected queries.

Static Model. The static model represents the background knowledge on the application domain, such as information on a road network, traffic lights, and vehicle taxonomy.

Roads are split into segments of the same length. Therefore a road network is modeled through two predicates, `node/1` and `link/3`, defining its set of nodes and edges, respectively. Specifically, `node(X)` defines a connection point between two edges (i.e., road segments) `link(Y,X,D)` and `link(X,Z,D1)` where `Y` and `Z` are road segments, `X` is a connection point, and `D` and `D1` indicate the direction of the segment. The latter can only assume `we` or `ew` as value.

For each road segment, the predicate `maxSpeed/3` provides the relative speed limit in m/s; for instance, `maxSpeed(X,Y,S)` indicates a speed limit of `S` m/s on the link `X,Y`.

For each connection node corresponding to an intersection, the predicate `conflict_tl/3` defines the set of traffic light conflicts between pairs of links connected to it. A traffic light conflict exists when two links cannot simultaneously have the same traffic light state. For example, `conflict_tl(I,X,Y)` means a traffic light conflict exists between the links `X` and `Y` on the intersection node `I`.

Eventually, the predicates `vehType/1` and `isSubType/2` define the vehicle taxonomy. The first lists the set of possible vehicle types, while the second defines a hierarchical relation for them, e.g., `isSubType(T1,T2)` means that the type `T1` is a sub-type of the type `T2`.

Streaming Model. Table 13.1 reports details about the streaming information available.

I-DLV-sr Modeling. In the traffic management scenario, many tasks have been proposed. In this paragraph, we show the I-DLV-sr modeling for some of

Streaming Message	Description
`vehModel(V,M)`	the model `M` of a vehicle `V`
`speed(V,S,T)`	the speed `S` of a vehicle `V` at the time point `T`
`position(V,X,Y,T)`	the location `X`,`Y` of a vehicle `V` at the time point `T`
`onLane(V,Ls,Le,T)`	the vehicle `V` is on the segment (Ls,Le) at the time point `T`
`heading(V,O,T)`	the orientation `O` of the vehicle `V` at the time point `T`
`accel(V,A,T)`	the acceleration `A` of the vehicle `V` at the time point `T`
`tlight(I,TL,S,T)`	the state `S` of the traffic light `TL` on an intersection `I`

Table 13.1: Streaming information of the ITS model proposed in [117]

them.

- "Calculate the number of vehicles (NoV) and average speed of all vehicles on each edge".

  I-DLV-sr Modeling:

```
r1 nOv(X,Y,D,N) :- link(X,Y,D), #count{Veh:onLane(Veh,X,Y,D,T)}=N.
r2 speedAVG(X,Y,D,Avg) :- nOv(X,Y,D,N), N>0,
       #sum{Sp,Veh: onLane(Veh,X,Y,D,T),speed(Veh,Sp,T)}=Sum,
       Avg=Sum/N.
```

  For each edge, r_1 computes the number of vehicles, and, accordingly, r_2 computes the average speed of that vehicles. Both rules make use of standard aggregate atoms for their computation.

- "Find the time intervals where vehicles exceed the top speed defined in `maxSpeed(X,Y,D)`.".

  I-DLV-sr Modeling:

```
r1 violationStart(Veh,X,Y,Tn) :- speed(Veh,Sp,Tn),
       onLane(Veh,X,Y,D,Tn), not onLane(Veh,X,Y,D,Tp) in {1},
       Tp=Tn-1, maxSpeed(X,Y,Msp), Sp>Msp.
r2 violationStart(Veh,X,Y,Tn) :- speed(Veh,Sp,Tn),
       onLane(Veh,X,Y,D,Tn), onLane(Veh,X,Y,D,Tp) in {1},
       speed(Veh,Sp1,Tp) in {1}, maxSpeed(X,Y,Msp), Sp>Msp,
       Sp1<=Msp.
```

```
r3  violationEnd (Veh,X,Y,Tp) :- onLane(Veh,X,Y,D,Tp) in {1},
        speed(Veh,Sp,Tp) in {1}, not onLane(Veh,X,Y,D,Tn), Tn=Tp+1,
        maxSpeed(X,Y,M), Sp>M.
r4  violationEnd (Veh,X,Y,Tp) :- onLane(Veh,X,Y,D,Tn),
        speed(Veh,Sp,Tn), onLane(Veh,X,Y,D,Tp) in {1},
        speed(Veh,Spp,Tp) in {1}, maxSpeed(X,Y,Msp), Spp>Msp,
        Sp<=Msp.
```

On the one hand, r_1 and r_2 derive the start of time intervals in which vehicles committed a speed violation on an edge. The latter happens in two cases:

i. a vehicle enters a new edge, and its speed is greater than the speed limit of that edge (i.e., the case modeled by r_1);

ii. a vehicle exceeds the speed limit at a time point t, but it was within the limit at the previous time point $t-1$ (i.e., the case modeled by r_2).

On the other hand, r_3 and r_4 derive the end of such time intervals. Specifically, a violation ends in two possible cases:

i. a vehicle is not on an edge anymore (i.e., the case covered by r_3);

ii. a vehicle exceeded the speed limit at the previous time point $t-1$, but this is no longer true at the current time point t (i.e., the case covered by r_4).

- "Detect the vehicles that made a short stop, i.e., with a duration less than 5 seconds".

  I-DLV-sr Modeling:

```
r1  speed(Veh,Sp) :- speed(Veh,Sp,T).
r2  shortStop(Veh) :- speed(Veh,0), speed(Veh,0) in [4 sec],
        not speed(Veh,0) always in [5 sec].
```

  r_1 is an auxiliary rule that projects only `speed` terms useful to answer the query. r_2 derives a short stop if a vehicle had a speed equal to 0 at least once in the last 4 seconds but not always in the last 5 seconds.

- "Detect the vehicles that make a u-turn traffic violation".

  I-DLV-sr Modeling:

```
r1  paired(N1,N2,N2,N1):- link(N1,N2,D).
r2  paired(N1,N2,N2,N1):- paired(N2,N1,N1,N2).

r3  violationUTurn(Veh):- onLane(Veh,N1,N2,D,Tn) in {1},
          onLane(Veh,M1,M2,D,Tn+1), paired(N1,N2,M1,M2).
```

r_1 and r_2 are auxiliary rules identifying the two different orientations of the same edge. r_3 derives a u-turn violation for a vehicle if it changes orientation on the same edge in two consecutive time points.

13.1.3 CityBench

CityBench [7] is a benchmark designed to evaluate the usability of SR systems in real-world smart city scenarios. It was conceived for RDF-based stream processors but brings several requirements and challenges that also fit to other kinds of SR systems, such as the ASP-based ones. CityBench collects real-time data streams[3] from sensors spread along Aarhus a city of Denmark. Furthermore, it features 13 continuous queries that require reasoning on large background knowledge and dynamic data streams. More in detail, the queries are envisioned in the context of the smart city applications described below.

Multi-modal Context-aware Travel Planner: this application aims to find a set of suitable alternative paths based ona user's preferences, travel cost, and other factors, along with continuously monitored events that change over time (e.g., traffic, weather, etc.), so to provide the best up-to-date option in real-time. Five queries are part of this application and regard the traffic congestion level, the weather conditions, the travel time, and the proximity of social events to the users as they travel along the planned paths.

Parking Space Finder Application: this application aims to help the user find a parking spot based on parking data streams and availability estimations. There are four queries proposed in this application to identify current parking conditions in places close to the user, its destination, or some social event of interest.

Smart City Administration Console: this application aims to ease the identification of both anomalies and trends by city administrators. Four are the queries proposed in this application to notify the administrators of critical events re-

[3] `http://iot.ee.surrey.ac.uk:8080/datasets.html` (url date: 15/01/2023)

lated to pollution, weather, and traffic congestion level so that they can apply immediate actions in case of need. In the following, we illustrate how some CityBench queries can be expressed via the I-DLV-sr language.

Query 5

Let us consider the query: "What is traffic congestion level on the road where a given cultural event X is happening? Notification for congestion level should be generated every minute starting from 10 minutes before the event X is planned to end, till 10 minutes after".

We can model this query as follows.

```
r1 eventsToConsider(E) :- eventOfInterest(E),
      eventDuration(E,StartTime,EndTime), CurrentTime=@now,
      CurrentTime>=EndTime-10, CurrentTime<=EndTime+10.
r2 closeRoads(RoadID) :- eventsToConsider(E),
      eventLocation(E,X2,Y2), roadLocation(RoadID,X1,Y1),
      &areClose(X1,Y1,X2,Y2;).
r3 congestionLevel(RoadID, CL) :- closeRoads(RoadID),
      vehicleCount(RoadID,VC), distanceInMeters(RoadID,DM),
      &floatDivision(VC,DM;CL).
```

where:

- `eventDuration`, `eventLocation`, `roadLocation`, `distanceInMeters` represent the background knowledge about cultural events and roads, i.e., the location of an event, the end time of an event, the start location of a road, and the length of a road, respectively;
- `vehicleCount` represents the streaming information on the traffic state of the city roads;
- `&areClose` is an *external predicate* holding if two positions are considered to be close to each other;
- `&floatDivision` is another external predicate that performs the division between two integer numbers preserving the decimal part of the result[4].

In this program, r_1 filters the events that must be considered according to the query condition: it uses **@now** to identify the current time and determine if

[4]The use of the external atom to compute the division is needed as floating points numbers are not supported yet, and native division among integers would only produce truncated integer results.

an event ended at most 10 minutes ago from now, or if it will end in 10 minutes from now. r_2 selects the roads close to the filtered social events. r_3 computes the congestion level on the selected roads based on the traffic information. Note that this encoding assumes times (i.e., those referred to as `StartTime` and `EndTime`) to be expressed in minutes; hence the `--now-format=min` command-line option is needed. Furthermore, since all the information required to determine if a location is exactly on a road is not available in the provided data set, we suppose an event is on a road if they are close.

The semantics of the streaming external atoms used above, i.e., `&areClose` and `&floatDivision`, are defined via two Python functions, as reported below.

```
import haversine as hs

def areClose(Lat1, Lon1, Lat2, Lon2):
   loc1=(Lat1,Lon1)
   loc2=(Lat2,Lon2)
   return hs.haversine(loc1,loc2) <= 1

def floatDivision(VC,DM):
   CL=VC/DM
   return ":.3f".format(CL)
```

The `areClose` function makes use of the `haversine` Python package[5] to determine the geographic distance between two locations and considers them close if their distance is at most 1 Kilometer.

The `floatDivision` function computes the division between two integers and returns a string containing the result with a three-digit precision. Notably, returning a string permits keeping the precision while still handling the division result in $\mathcal{I}^2$-*DLV*.

Query 10. Let us now consider the query "Notify me every 10 minutes, about the most polluted area in the city".

We can model this problem according to the *European Air Quality Index (EAQI)*. As reported in Table 13.2, the EAQI is based on concentration values of five pollutants: *particulate matter (PM10)*, *fine particulate matter (PM2.5)*, *ozone (O3)*, *nitrogen dioxide (NO2)*, *sulphur dioxide (SO2)* ; each of which has six

[5] `https://pypi.org/project/haversine/` (url date: 15/01/2022)

[6] `https://www.eea.europa.eu/themes/air/air-quality-index/index` (url date: 15/01/2023)

Pollutant	Good	Fair	Moderate	Poor	Very Poor	Extremely Poor
Particles $\leq$ 2.5 µm (PM2.5)	0-10	10-20	20-25	25-50	50-75	75-800
Particles $\leq$ 10 µm (PM10)	0-20	20-40	40-50	50-100	100-150	150-1200
Nitrogen dioxide (NO2)	0-40	40-90	90-120	120-230	230-340	340-1000
Ozone (O3)	0-50	50-100	100-130	130-240	240-380	380-800
Sulphur dioxide (SO2)	0-100	100-200	200-350	350-500	500-750	750-1250

Table 13.2: European Air Quality Index levels as defined by the European Environment Agency[6]– Classification levels are based on pollutant concentrations in $\mu g/m^3$

classification levels ranging from *Good* to *Extremely Poor*.

To determine the EAQI level of each pollutant measured in every road, we define a set of rules that straightforwardly encode Table 13.2, representing as integers the classification levels, from 0 (meaning *Good*) up to 5 (meaning *Extremely Poor*). For instance, we encode the *Good* classification of the ozone (`"O3"`) (row 4, column 1) with the rule:

```
roadPollutantLevel(RoadID, "O3", 0) :-
      pollutionMeasurement(RoadID,O3,PM10,SO2,NO2), O3<50.
```

where `pollutionMeasurement` represents the streaming information about the air quality indexes of the pollutants measured in each road of the city.

Since the pollution level of a road is given by the worst EAQI level measured in that road, r_1 determines the highest pollutant level for each road, and r_2 finds the city areas whose pollution level is the maximum.

```
r1  #trigger_frequency(10 min) worstAirQualityLevel(Level) :-
       #max{L: roadPollutantLevel(RoadID,P,L)}=Level.
r2  #trigger_frequency(10 min) mostPollutedArea(RID,P,L) :-
       roadPollutantLevel(RID,P,L), worstAirQualityLevel(L).
```

Query 12. Let us consider the query: "Notify me whenever the congestion level on a given road goes beyond a predefined threshold more than 3 times within the last 20 minutes".

We can model this query as follows.

```
r1  congestionLevel(RoadID, CL) :- roadOfInterest(RoadID),
       vehicleCount(RoadID,VC), distanceInMeters(RoadID,DM),
       &floatDivision(VC,DM;CL).
r2  beyondThreshold :- congestionLevel(_, CL), threshold(T),
```

```
    CL>T.
r3  warningBeyondThreshold :- minTimesTrigger(X),
    beyondThreshold at least X in [20 min].
```

where:

- `roadOfInterest` specifies the identifier of the road for which notifications on congestion level must be produced;
- `vehicleCount` represents the streaming information about the number of vehicles currently passing on a road;
- `distanceInMeters` represents the length of a road;
- `&floatDivision` is the external atom introduced in Query 5 above;
- `threshold` contains the threshold that causes a notification to be produced if the number of times specified via `minTimesTrigger` is exceeded.

In this program, r_1 computes the traffic congestion level on the road of interest; r_2 derives the event `beyondThreshold` if it goes beyond the specified threshold; r_3 arises a warning if the event `beyondThreshold` has been derived at least `X` times within the last time window of `20 min`.

Notice that we use the `threshold` and `minTimesTrigger` predicates to keep the encoding more generic. In such a way, it is possible to update these constraints without directly operating on the encoding of rules relying on them. For instance, supposing that the threshold is 1 and that it must not be exceeded more than 3 times, the following facts can be used to model such information without any need for updates in the rules: `threshold(1)` and `minTimesTrigger(4)`.

13.1.4 *Metropolitan Area Monitoring*

Besides CityBench, another work based on the same Aarhus data streams has been recently proposed [64]: an approach based on Deep Neural Networks allows high-speed query answering at the price of approximating results. In the following, we refer to this work as *Metropolitan Area Monitoring*.

Metropolitan Area Monitoring includes 5 queries in smart city scenarios, each focused on identifying the occurrence of special events in temporal windows by analyzing weather, parking, traffic, and pollution data streams. Additionally, sensors producing such data streams are grouped into *sectors* based on their

Pollution Trend	Pollution Index	Traffic Trend	Traffic Indicator	Classification
increasing	between 214 and 215 (included)	decreasing	between 10 and 11 (included)	industrial
decreasing	between 0 and 15 (included)	increasing	between 10 and 11 (included)	urban
increasing	between 214 and 215 (included)	increasing	between 250 and 300 (included)	highway

Table 13.3: Sector classification based on stream events observed during the last 9 minutes

geographic position. Next, we illustrate how some of the *Metropolitan Area Monitoring* queries are, to some extent, translated into the I-DLV-sr language. For the sake of comparison, predicate and variable names are left unchanged.

Query 1. Let us consider the query "Are there any metropolitan events in the last 9 time steps?". A metropolitan event occurs in a sector within a given time window if at least one industrial event, one urban event, and one highway event are detected in that sector in that time window. In this query, we suppose the time steps to be 1 minute.

An industrial (respectively urban/highway) event happens when a sector is always classified as industrial (respectively urban/highway) within the last time window of 3 minutes. Table 13.3 reports the sector classifications according to their observed events within the last time window of 9 minutes.

The I-DLV-sr encoding of this query is reported below.

```
r1  poll_measurementAt(MES,VAL,SEC,T) :- pollution(MES,VAL,SEC), T=@now.
r2  traff_measurementAt(MES,VAL,SEC,T) :- traffic(MES,VAL,SEC), T=@now.

r3  poll_inc(MES,SEC) :- poll_measurementAt(MES,VAL,SEC,T) in [9 min],
        poll_measurementAt(MES,VAL1,SEC,T1) in [9 min],T1=T+1,VAL1>=VAL.
r4  poll_dec(MES,SEC) :- poll_measurementAt(MES,VAL,SEC,T) in [9 min],
        poll_measurementAt(MES,VAL1,SEC,T1) in [9 min],T1=T+1,VAL1<=VAL.

r5  poll_low(MES,SEC) :- pollution(MES,VAL,SEC) in [9 min], VAL>=0,
        VAL<=15.
r6  poll_high(MES,SEC) :- pollution(MES,VAL,SEC) in [9 min], VAL>=214,
        VAL<=215.

r7  traff_inc(MES,SEC) :- traff_measurementAt(MES,VAL,SEC,T) in [9 min],
        traff_measurementAt(MES,VAL1,SEC,T1) in [9 min], T1=T+1,
        VAL1>=VAL.
```

```
r8  traff_dec(MES,SEC) :- traff_measurementAt(MES,VAL,SEC,T) in [9 min],
        traff_measurementAt(MES,VAL1,SEC,T1) in [9 min], T1=T+1,
        VAL1<=VAL.

r9  traff_low(MES,SEC) :- traffic(MES,VAL,SEC) in [9 min], VAL>=10,
        VAL<=11.
r10 traff_high(MES,SEC) :- traffic(MES,VAL,SEC) in [9 min], VAL>=250,
        VAL<=300.

r11 industrial_area(SEC) :- poll_inc(MES,SEC) in [9 min],
        traff_dec(MES10,SEC) in [9 min], poll_high(MES1,SEC) in [9 min],
        traff_low(MES2,SEC) in [9 min], MES!=MES10.
r12 industrial_box(SEC) :- industrial_area(SEC) always in [3 min].

r13 urban_area(SEC) :- traff_inc(MES, SEC) in [9 min],
        poll_dec(MES10,SEC) in [9 min], poll_low(MES2,SEC) in [9 min],
        traff_low(MES4,SEC) in [9 min], MES != MES10.
r14 urban_box(SEC) :- urban_area(SEC) always in [3 min].

r15 highway_area(SEC) :- traff_inc(MES, SEC) in [9 min],
        poll_inc(MES10,SEC) in [9 min], poll_high(MES2,SEC) in [9 min],
        traff_high(MES4,SEC) in [9 min], MES!= MES10.
r16 highway_box(SEC) :- highway_area(SEC) always in [3 min].

r17 city(SEC) :- industrial_box(SEC), highway_box(SEC), urban_box(SEC).
```

where `pollution` and `traffic` represent the streaming information about the air quality index (`VAL`) of a pollutant (`MES`) in a sector (`SEC`) and the value (`VAL`) of a traffic indicator (`MES`) in a sector, respectively.

r_1 and r_2 are auxiliary rules designated to associate each pollution and traffic measurement with the time point in which it has been observed. r_3 derives a pollution increment in a sector if, during the last 9 minutes, there are two subsequent pollution measurements where the value that came second is greater than the one that came first; similarly, r_4 detects a pollution decrement when the value that came second is smaller than the one that came first. r_5 (respectively r_6) determines if, within a sector, a low (respectively a high) pollution index has been observed at least once during the last 9 minutes. Rules r_{7-10} are in order very similar to r_{3-6} with the difference that the first couple of rules reason about the traffic stream instead of the pollution one. r_{11}, r_{13}, and r_{15} encode each row of Table 13.3 in order, thus classifying sectors. r_{12}, r_{14}, and r_{16} derive if, within the last 3 minutes, a sector has always been classified as industrial, urban, and highway, respectively. r_{17} answers the query checking if a metropolitan event happened on the basis of industrial, urban and highway

events derived by the other rules.

Query 5. Let us consider the query "In the last four time steps, in which sectors has an anomaly been detected?". A sector can be ether a *city* or a *town*, where a town is a *suburb* of a city. Additionally, a proximity relation defines if two suburbs belong to the same city. An anomaly occurs in a sector if it is a city and it is classified as an industrial sector together with two of its suburbs. In this query, we suppose time steps of 1 minute.

A sector is considered industrial if, within the last 4 minutes, high levels of carbon monoxide (CO) and particulate matter (PM) are always registered at least once by three different sensors. Pollutants are considered high if their concentration levels exceed 125.

The I-DLV-sr encoding of this query is reported below.

```
r1  close(A, C, B) :- suburb(A, B), suburb(C, B) , A != C.

r2  highPollutionCo(SENS, SEC) :-
        pollution(2, MES, SENS, SEC) in [4 min], MES>125.
r3  highPollutionPM(SENS, SEC) :-
        pollution(1, MES, SENS, SEC) in [4 min], MES>125.

r4  highPollutionCo_cont(SENS, SEC) :-
        highPollutionCo(SENS, SEC) always in [4 min].
r5  highPollutionPM_cont(SENS, SEC) :-
        highPollutionPM(SENS, SEC) always in [4 min].

r6  industrialSens(SENS, SEC) :- highPollutionCo_cont(SENS, SEC),
        highPollutionPM_cont(SENS, SEC).
r7  industrialSec(SEC) :- industrialSens(SENS1, SEC) ,
        industrialSens(SENS2, SEC), industrialSens(SENS3, SEC),
        industrialSens(SENS4, SEC), SENS1 != SENS2, SENS1!= SENS3,
        SENS1!= SENS4, SENS2!= SENS3, SENS2!= SENS4, SENS3!= SENS4.

r8  anomaly(CITY) :- industrialSec(SEC1), industrialSec(SEC2),
         close(SEC1, SEC2, CITY).
r9  industrialArea(SEC) :- industrialSec(SEC), city(SEC).
r10 alert(SEC) :- industrialArea(SEC), anomaly(SEC).
```

where `suburb` represents the background knowledge on suburbs (first term) of a given city (second term), and `pollution` represents streaming information about the level (second term) of a given pollutant (first term) measured by a sensor (third term) in a sector (fourth term).

r_1 derives suburbs belonging to the same city. r_2 and r_3 check, respectively, if high levels of CO and PM have been observed at least once in a sector within a time window of 9 minutes, and r_4 and r_5 infer if such high levels always happened during the last 4 minutes. r_6 determines the set of industrial sensors, and r_7 defines the set of industrial sectors accordingly. r_8 derives the eventual anomaly in a city if two of its suburbs are classified as industrial. r_9 classifies a city as an industrial area taking into account the generic classification done by r_7. r_{10} answers the query by raising an alert for a city if it has been classified as industrial and if an anomaly has been registered in its suburbs.

13.2 Control of Caching Strategies in Content-Centric Networking

Content-Centric Networking (CCN) [88] is a protocol for network communications that handles requests of users on the basis of the requested content name instead of the location address. Specifically, packets are addressed on the network according to their content names, not the IP address of the network node handling such content. Additionally, communication is driven by data consumers. When a user requests for a content, a request packet (called *interest* packet) is broadcasted on the network. Here, it is forwarded from one node to the next one until any node receiving the interest packet contains a content named like the requested one. In this case, the provider node replies with a packet (called *data* packet) containing the desired content. Given that packages are addressed by content name, this protocol allows the users asking for the same content to share the relative transmission in broadcast.

Problem Definition

In CCN networks, content can be placed on a server or the caches of network routers. Therefore, adopting a good caching strategy on each router can significantly improve network traffic performance. A caching strategy determines weather a content should be cached and which content should be eventually removed.

In [20], a rule-based stream reasoning solution has been proposed to automatically adapt caching strategies of routers according to the stream of requests they receive. The authors considered the following caching strategies:

- *Least Recently Used (LRU)*: this strategy stores contents in a list sorted by their access time stamps. When a new content should be cashed and the list is full, it replaces the oldest content with the new one.
- *First-In-First-Out (FIFO)*: this strategy stores contents in a queue where the earliest inserted content is replaced when there is no space to store newer content.
- *Least Frequently Used (LFU)*: this strategy counts how often a content in the cache is accessed, and, when a new content should be stored, the one with the smallest access count is replaced if the storage is full.
- *Random*: this strategy simply replaces a random content in the cache with a new one when necessary.

Such a solution is modeled using a LARS program and subsequently also implemented in the fragment (Section 8.1) of Ticker [24] and Distributed-sr [57].

In this section we show how the encoding proposed in [57] can be, to some extent, translated into I-DLV-sr. This version aims to manage the caching policy of a video content over an incoming stream that describes the evolving popularity level of the content. It is worth noting that Distributed-sr and I-DLV-sr rely on different languages and semantics [96]; thus the mining of rules may not always coincide.

I-DLV-sr Modeling

The following is the I-DLV-sr encoding for the content caching problem introduced in [57].

```
r1  high :- alpha(V) in [3 sec],18 <= V.
r2  mid :- alpha(V) in [3 sec],12 <= V,V < 18.
r3  low :- alpha(V) in [3 sec],V <= 12.
r4  lfu :- high always in [3 sec].
r5  lru :- mid always in [3 sec].
r6  fifo :- low always in [3 sec], rtm50 always in [3 sec].
r7  done :- lfu.
r8  done :- lru.
r9  done :- fifo.
r10 random :- not done.
r11 finish :- off in [1 sec], done.
r12 finish :- off in [1 sec], random.
```

where `alpha`, `rtm50`, and `off` are streaming information: `alpha` registers popularity levels of the content at hand, `rtm50` indicates if the asked content

is a real-time content, like broadcasts, video calls, etc., and `off` notifies the reasoner that no reasoning is required anymore.

r_1, r_2 and r_3 derive, respectively, if it has been registered a high, mid, and low popularity level within a time window of 3 seconds. On the basis of this, r_4, r_5, and r_6 determine which caching policy should be applied. Specifically, r_4 infers `lfu` if a high popularity level has always been registered within the last 3 seconds; r_5 infers `lru` if a mid popularity level has always been registered within the last 3 seconds; r_6 infers `fifo` if, within the last 3 seconds, a low popularity level has always been registered together with the streaming information `rtm50`. If one policy among `lfu`, `lru`, and `fifo` is derived, rules r_{7-9} infer the fact `done`; otherwise, r_{10} infers `random`, meaning that the default policy should be applied. Rules r_{11-12} establish the end of evaluation if a policy has been fixed and an `off` signal has been observed within the last second.

Chapter 14

Experimental Evaluation

In this chapter, we report the results of an experimental activity we carried out to assess the performance and usability of I-DLV-sr in real-world, data-intensive, and reasoning-intensive SR applications. All experiments have been performed on a `NUMA` machine equipped with two 2.8GHz AMD Opteron 6320 CPUs, with 16 cores and 128GB of RAM, and are available at the dedicated repository [43].

Section 14.1 describes the analysis carried out. Section 14.2 provides details on the benchmarks used through the experimental evaluation to test and compare the SR systems. Therefore, Sections from 14.3 to 14.7 discuss the results obtained for each considered benchmark.

14.1 Analysis Description

We conducted two kinds of analysis.

On the one hand, we wanted to compare the performance of I-DLV-sr with some available logic-based stream reasoners. As reported in Chapter III, many implementations exist; however, they considerably differ in syntax and/or semantics, and also from an architectural/implementation point of view. Such differences make achieving a fair comparison rather challenging. Given the distributed nature of I-DLV-sr, we compared it to Distributed-SR [57] (Subsection 8.1.3), the most recent LARS-based implementation which supports a large set of features and relies on a distributed architecture. In particular, we executed the latest Distributed-SR version[1].

[1] available at: `http://distributed-stream-reasoner.ainf.at/`

On the other hand, we wanted to assess the effectiveness of the introduced enhancements described in Section 11.2. Therefore, we compared the first version of I-DLV-sr with its latest version. For the sake of readability, with I-DLV-sr-v1,[2] we refer to the first version of the system, while with I-DLV-sr-v2,[3] we refer to the second one.

Our analysis focuses on three measures: `Total Time`, `#Accepted Requests`, and `Latency`: `Total Time` represents the total elapsed time for each execution, excepting the initial time spent for the setup; `#Accepted Requests` is the number of accepted incoming requests computed by checking the system logs and counting the number of time points whose corresponding input is read by the system; `Latency` is the processing-time latency, i.e., the interval between the time at which the system receives the input relative to a time point and the time at which the system returns the corresponding output.

14.2 Benchmark Description

For the experimental evaluation, we used 5 benchmarks:

- *Content Caching* [20, 57]: is a real-world benchmark that requires to manage the caching policy of a video content over an incoming stream that describes the evolving popularity level of that content (see Section 13.2). Besides the original problem [20], we also considered a slightly different version that deals with more than one event[4] per time point. Therefore, the encoding is adapted to handle more than one video content.

- *Heavy Join*: is an artificial problem conceived to test scalability. It consists of the single rule: `a(X,Y):-b(X,Z) in [w],c(Z,Y) in [w].`, where w is the window size, and that requires computing a join over evaluation results within two windows.

- *Photo-voltaic System*: is a benchmark concerning the problem introduced in Subsection 13.1.1. In particular, it contains several instances of this problem with grids of increasing size ranging from 20×20 panels with $11,970$ links up to 30×30 panels with $60,682$ links. Therefore, each instance is coupled with a designated input stream providing streaming information for each panel within the grid.

[2] `https://github.com/DeMaCS-UNICAL/I-DLV-sr/releases/tag/v1.0`
[3] `https://github.com/DeMaCS-UNICAL/I-DLV-sr/releases/tag/v1.1.0`
[4] we refer to a true atom at a time point in the stream as an "event"

Benchmark/ Query	**Recursive**	**Bakground Knowledge**	**Time-based Windows**	**Join Time-based Windows**	**Advanced Constructs**
Content Caching					
Single Content	No	None	Yes	Yes	None
Catalog Content	No	None	Yes	Yes	None
Heavy Join	No	None	Yes	Yes	None
Photo-voltaic System	Yes	Large	Yes	No	None
CityBench					
Q1	No	Small	No	No	External Atoms
Q2	No	Small	No	No	External Atoms
Q3	Yes	Small	No	No	External Atoms
Q4	No	Medium	No	No	@now, External Atoms
Q5	No	Large	No	No	@now, External Atoms
Q6	No	Small	No	No	External Atoms
Q7	No	Small	No	No	External Atoms
Q8	No	Medium	No	No	External Atoms
Q9	No	Medium	No	No	External Atoms
Q10	No	None	No	No	Trigger Rules
Q11	No	None	Yes	No	None
Q12	No	Small	Yes	No	External Atoms
Q13	No	Small	No	No	External Atoms
Metropolitan Area Monitoring					
Q1	No	None	Yes	Yes	@now
Q2	No	None	Yes	No	None
Q3	No	None	Yes	Yes	@now
Q4	No	None	Yes	Yes	@now
Q5	No	Small	Yes	No	None

Table 14.1: Comparing the key features of the benchmarks used in the experiments

- *CityBench* [7] is a benchmark designed to evaluate the usability of SR systems in real-world smart city scenarios (Subsection 13.1.3). It features 13 queries requiring reasoning on background knowledge and dynamic data streams from sensors spread along the city of Aarhus in Denmark[5]. Each query is coupled with the input stream it must reason on.

  - Queries $Q1$, $Q2$, $Q3$, $Q5$, $Q12$, and $Q13$, reason about events in the traffic stream; therefore, they are coupled with a stream containing information on about 100 roads. In particular, per each road, the stream contains the number of vehicles, the average speed, the average travel time, and the median travel time. Additionally, for $Q2$, the incoming stream also contains information on weather conditions.
  - $Q4$ identifies cultural events closest to the user's location; hence, its stream contains the GPS coordinates of the user.
  - Queries $Q6$, $Q7$, $Q8$, and $Q9$ monitor parking conditions; thus, they are evaluated together with an input stream containing streaming information on about 8 parking garages.
  - $Q10$ identifies the most polluted areas, then it is coupled with a stream containing measurements for each pollutant (e.g., particulate matter, ozone sulphur dioxide, etc.) on about 100 roads.
  - $Q11$ monitor weather condition, so its stream features weather measurements like wind speed, humidity, temperature etc.

- *Metropolitan Area Monitoring*: is a benchmark that proposes 5 queries to identify complex events in metropolitan areas based on pollution and traffic streaming information (Subsection 13.1.4). As in the case of *CityBench*, *Metropolitan Area Monitoring* uses streams coming from sensors within the city of Aarhus in Denmark. The input stream features about $2,680$ events per time point.

While both *Content Caching* and *Metropolitan Area Monitoring* deal with smart city domain and rely on the same source of data, their queries significantly differ in terms of reasoning complexity. Indeed, in *Metropolitan Area Monitoring*, all the queries use several windows, which are often joined or temporally dependent. This requires, in general, a more complex reasoning effort.

[5] `http://iot.ee.surrey.ac.uk:8080/datasets.html`

Table 14.1 reports key features of the considered benchmarks; specifically, if they contain recursive rules, require to also reason on background knowledge along with an intuition about its size, make use of time-based windows, need to join different time-based windows, and employ advanced constructs.

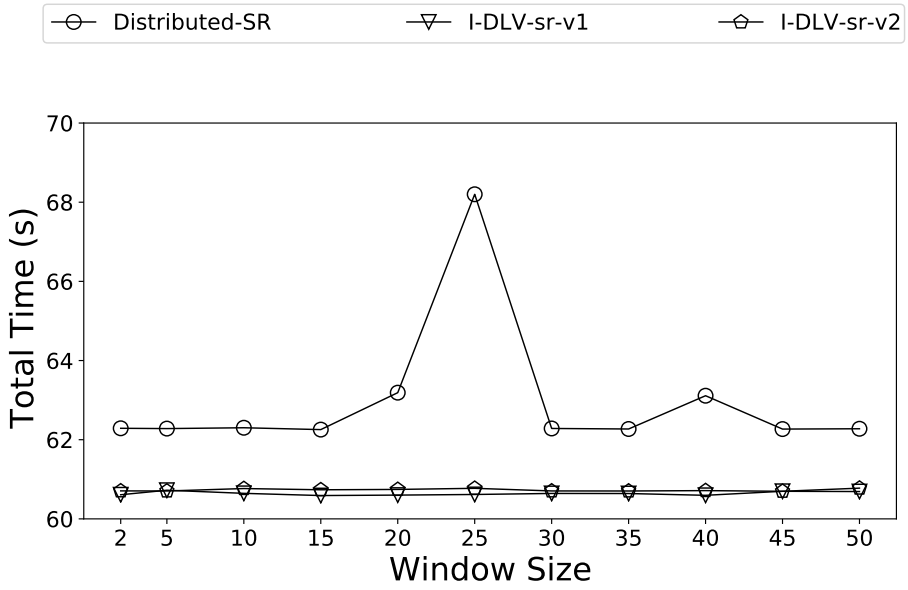
As introduced, Distributed-SR, I-DLV-sr-v1 and I-DLV-sr-v2 differ in supported features and/or semantics, so we tested them only on the domains where a fair comparison is possible. In particular, we compared all of them on *Content Caching* and *Heavy Join*, and I-DLV-sr-v1 and I-DLV-sr-v2 on *Photo-voltaic System* and *Metropolitan Area Monitoring*. Finally, I-DLV-sr-v2 is the only one tested on *CityBench*.

14.3 Results on *Content Caching*

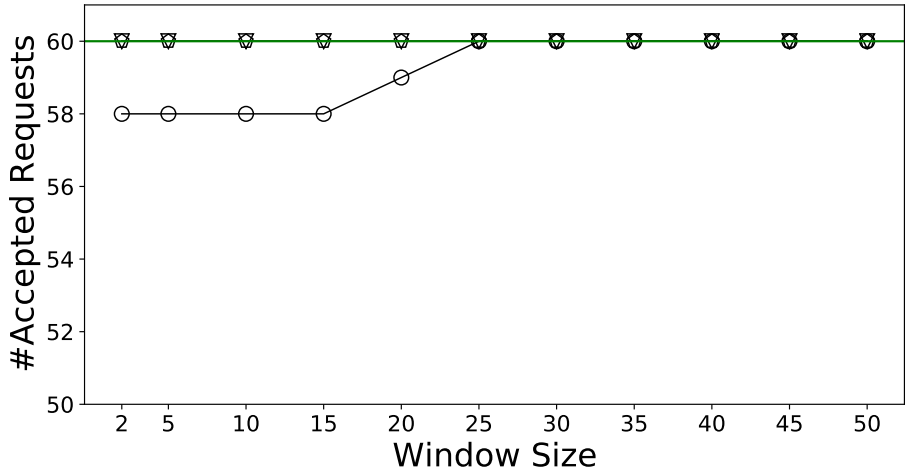
This section discusses the results obtained by I-DLV-sr-v1, I-DLV-sr-v2, and Distributed-SR on the domain *Content Caching*. In particular, we conducted two experiments. The first aims at assessing system performance on the original *Content Caching* encoding by varying the size of the windows from 2 seconds to 50 seconds. We recall that this version determines the caching policy of a single video content; thus, the systems receive an input stream containing one event per time point. The second uses the modified encoding to compare how systems performance changes when it is required to reason about more than one content and, thus, deal with more than one event per time point. In this setting, the systems are required to evaluate the modified encoding over different incoming streams. Such streams contain a number of events, representing the popularity levels for each content, ranging from 50 to 500.

Figure 14.1 and Figure 14.2 report the results obtained by Distributed-SR, I-DLV-sr-v1, and I-DLV-sr-v2 in the first and second experiment, respectively. Subfigures (a) show `Total Time`, while Subfigures (b) show `#Accepted Requests`. In each plot, a marker in a line corresponds to the average of the results of three execution. In every execution, the systems receive inputs for 60 time points at a frequency fixed to 1 time point per second.

Let us analyze the results obtained by the systems in the window variation experiment (Figure 14.1). According to Plot 14.1a, in most cases, the three systems appear to have similar behaviors w.r.t. `Total Time`, slightly above 60 seconds on average. We recall that here the frequency is one time point per second. Therefore, any execution cannot last less than 60 seconds, representing



(a) Total execution time in seconds

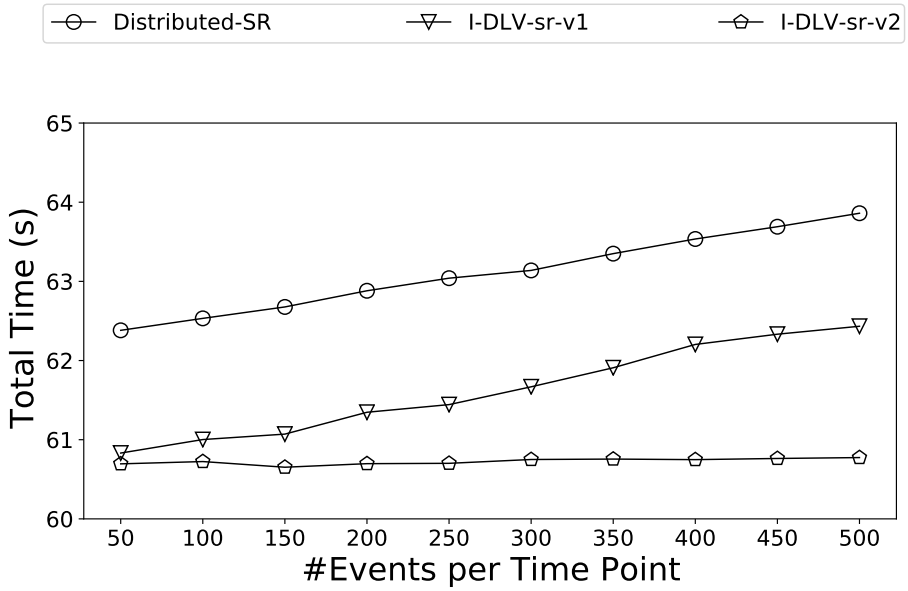


(b) Number of accepted requests out of 60 received in total

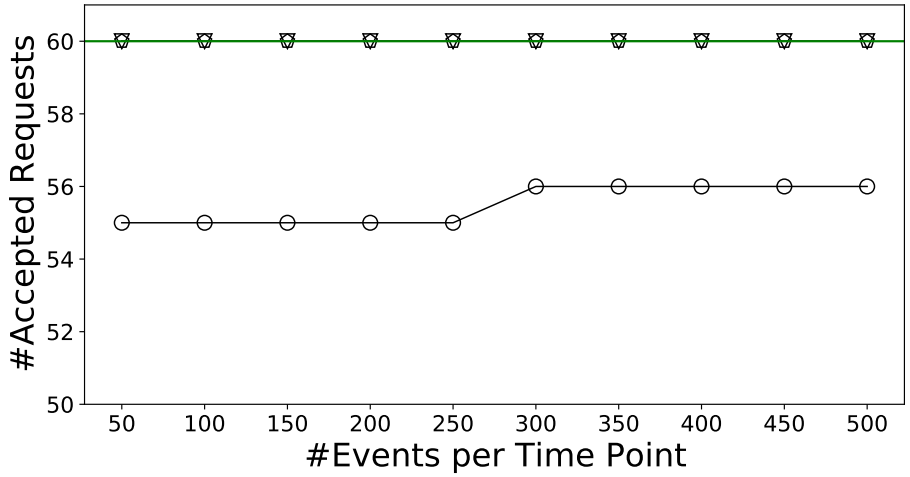
Figure 14.1: Results on *Content Caching*, window variation experiment - events per time point: 1 - input frequency: 1 time point per second - total input received: 60

the minimum amount of time needed to wait for all the incoming 60 requests. Generally, the two versions of I-DLV-sr do not suffer from window size variation; their trend is almost equal and very close to the minimum execution time possible. On the contrary, the trend of Distributed-SR comes with a peak quite pronounced when dealing with the encoding having a window size of 25 seconds. In addition, Plot 14.1b reports an important difference between I-DLV-sr and Distributed-SR. While both I-DLV-sr-v1 and I-DLV-sr-v2 accept and correctly return the expected output for all the 60 requests regardless of window size, Distributed-SR fails in handling them all and seems to work better with encodings having larger window sizes.

Similar results can be observed for the event variation experiment (Figure 14.2). Let us look at the total times in Plot 14.2a. Distributed-SR and I-DLV-sr-v1 show similar trends that grow along with the number of events for time points. I-DLV-sr-v2 presents instead a linear trend that is again very close to the minimum execution time possible for all the considered settings. Moreover, I-DLV-sr-v2 outperforms both I-DLV-sr-v1 and Distributed-SR. Differences between I-DLV-sr-v1 and I-DLV-sr-v2 in total times suggest that the introduced enhancements and optimization make I-DLV-sr more stable w.r.t. the size of input streams. As in the previous case, a loss of incoming requests is reported for Distributed-SR in Plot 14.2b, while I-DLV-sr-v1 and I-DLV-sr-v2 properly handle all of them.



(a) Total execution time in seconds



(b) Number of accepted requests out of 60 received in total

Figure 14.2: Results on *Content Caching*, experiment: number of events per time point variation - window size: 5s - input frequency: 1 time point per second - total input received: 60

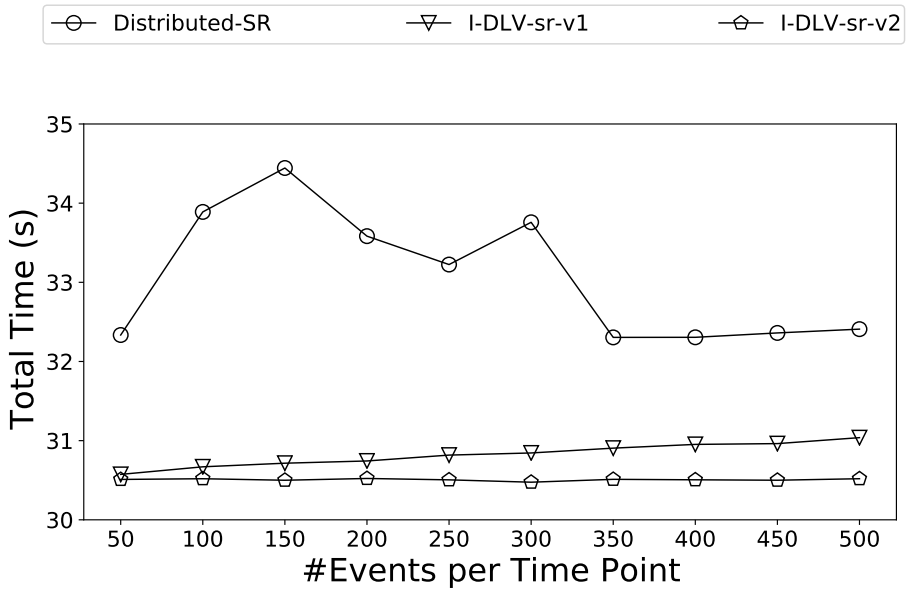
14.4 Results on *Heavy Join*

This section focuses on the results obtained by I-DLV-sr-v1, I-DLV-sr-v2, and Distributed-SR on the artificial domain *Heavy Join*.

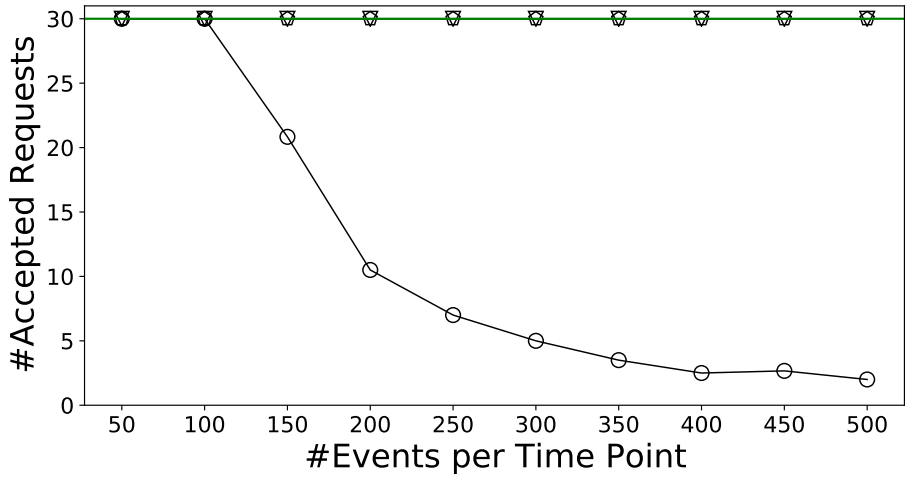
For this experiment, we considered two versions of the *Heavy Join* rule where the window size can be either 2 or 20 seconds. Furthermore, the systems are required to evaluate these rules on different input streams where the number of events per time point varies from 50 to 500. The considered input streams feature an equal number of instances of predicates b and c.

Figure 14.3 and Figure 14.4 report the results on the encoding having a window size of 2 and 20 seconds, respectively. In each figure, Subfigure (a) contains the plot showing `Total Time` results, while Subfigure (b) includes the plot representing `#Accepted Requests` results. In general, a marker in a line indicates the average of the results of three execution. In every execution, the systems receive inputs for 30 time points at a frequency fixed to 1 time point per second.

As we can see from these plots, the results are very similar regardless of the window size considered. In terms of `Total Time`, the fact that both versions of I-DLV-sr perform better than Distributed-SR is more evident than in the *Content Caching* case previously discussed. I-DLV-sr-v1 and I-DLV-sr-v2 maintain the same trend in both cases, while Distributed-SR exhibits a fluctuating behavior. Again I-DLV-sr-v2 is better than the other systems, displaying a linear trend that is not influenced by the input stream size. This confirms the effectiveness of the introduced improvements in the stream management processes. As for `#Accepted Requests`, Distributed-SR "misses" a relevant number thereof: the loss is limited when the number of events per time point is small, but it rapidly grows with the number of events for a time point, so that in the end almost all incoming requests are ignored. I-DLV-sr-v1 and I-DLV-sr-v2 also in these experiments properly handle all incoming requests confirming their reliability.

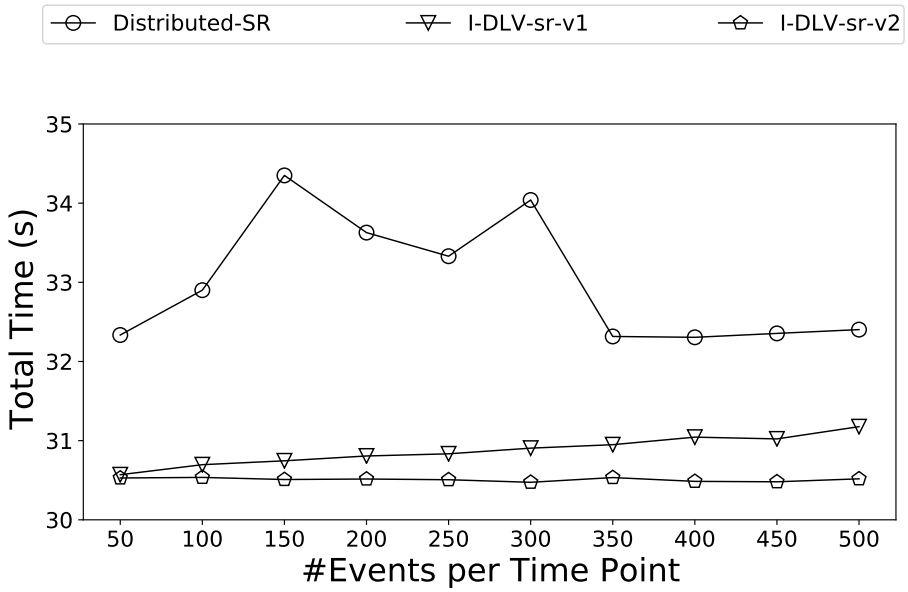


(a) Total execution time in seconds

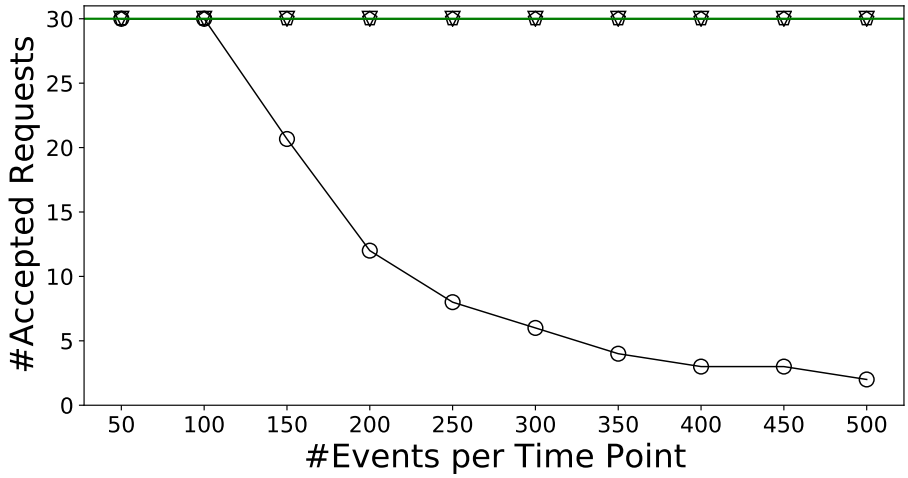


(b) Number of accepted requests out of 30 received in total

Figure 14.3: Results on *Heavy Join-* window size: 2s, input frequency: 1 time point per second



(a) Total execution time in seconds



(b) Number of accepted requests out of 30 received in total

Figure 14.4: Results on *Heavy Join*- window size: 20s, input frequency: 1 time point per second

14.5 Results on *Photo-voltaic System*

Given that the *Photo-voltaic System* problem (see Subsection 13.1.1) relies on specific features of I-DLV-sr, we tested and compared only I-DLV-sr-v1 and I-DLV-sr-v2 in this domain.

The main goal of this experiment is to show how total execution times and processing-time latency change varying the problem complexity and the frequency at which inputs for time points arrive. We vary the complexity of the problem by considering different grids with increasing sizes, ranging from 20×20 panels with $11,970$ links up to 30×30 panels with $60,682$ links. This is a quite expensive domain as rule r_2 and rule r_3 encode a reachability task among working grid panels that vary over time. Therefore, the more links and panels considered, the more complex it will be to calculate their reachability. We vary the input frequency by varying the *period* according to which we send time points to the systems. The considered periods range from 0.2 to 1.0; intuitively, when the period is equal to 0.2, the systems receive one time point every 0.2 seconds; in other words, they receive 5 time points per second: one every 200 milliseconds.

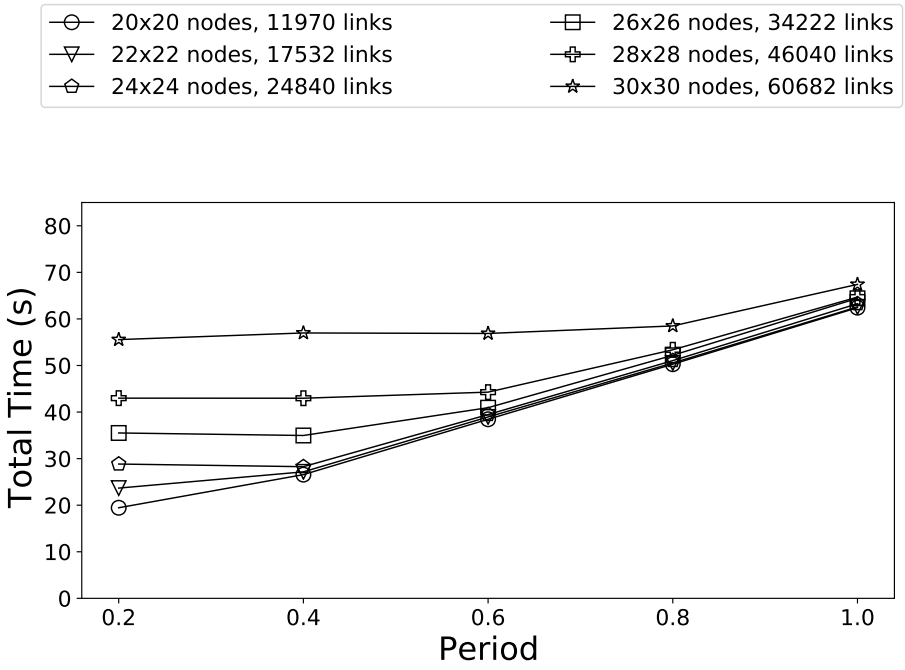
Figure 14.5 and Figure 14.6 report total execution times and processing time latencies, respectively. Subfigures (a) concern results obtained by I-DLV-sr-v1, while Subfigures (b) regard results of I-DLV-sr-v2. In every plot, each line corresponds to a different grid size, and a marker in a line is the average of results obtained from three executions. In each execution, the tested systems received inputs for 60 time points. The x-axis reports the considered periods. In this experiment, we do not report `#Accepted Requests` results as the systems correctly process all the 60 requests.

Let us analyze the `Total Time` plots of I-DLV-sr-v1 and I-DLV-sr-v2, the six lines show almost the same trend in both cases: constant up to a certain period P, and then linearly growing. A similar but flipped trend can also be observed in `Latency` plots, where the six lines start decreasing up to the period P after which they become constant. This is expected as such a P represents the starting point beyond which we observe the so-called sustainable throughput [89]. Intuitively, for period values smaller than P, the system is asked to process requests that come more often than it can sustain. In this case, it starts to enqueue pending requests to avoid data loss, which causes higher latency values. On the contrary, when the period is greater than P, the system is able to consider a request as soon as it arrives; thus, no queuing is

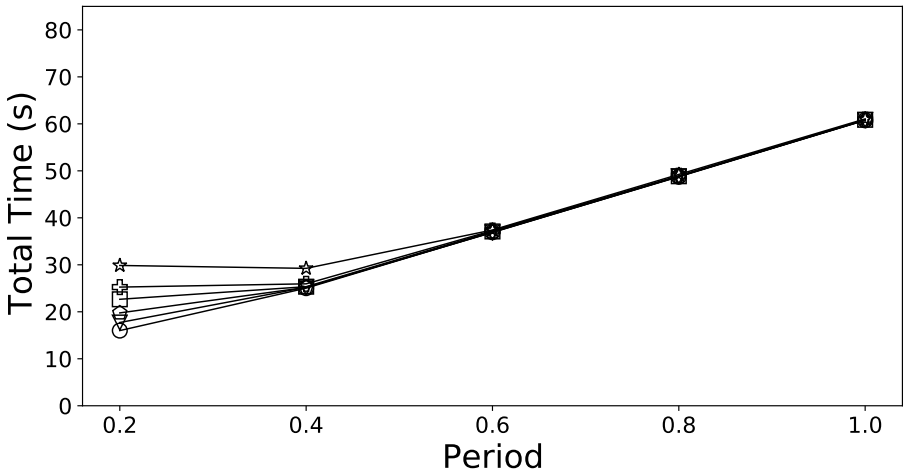
needed, and low latency values are obtained. Finally, the greater the period is w.r.t. P, the greater the idle time between the complete processing of a request and the incoming of the next one, which causes higher total execution times.

In general, periods close to P are ideal, as idle time is close to 0 and no queuing occurs, obtaining both low latency and total execution times.

The plots also show that I-DLV-sr-v2 clearly outperforms I-DLV-sr-v1; it enjoys a significant reduction of both `Latency` (up to more than 250%) and `Total Time` (up to 90%). Once again, the positive impact of re-engineering and introduced enhancements is proven.

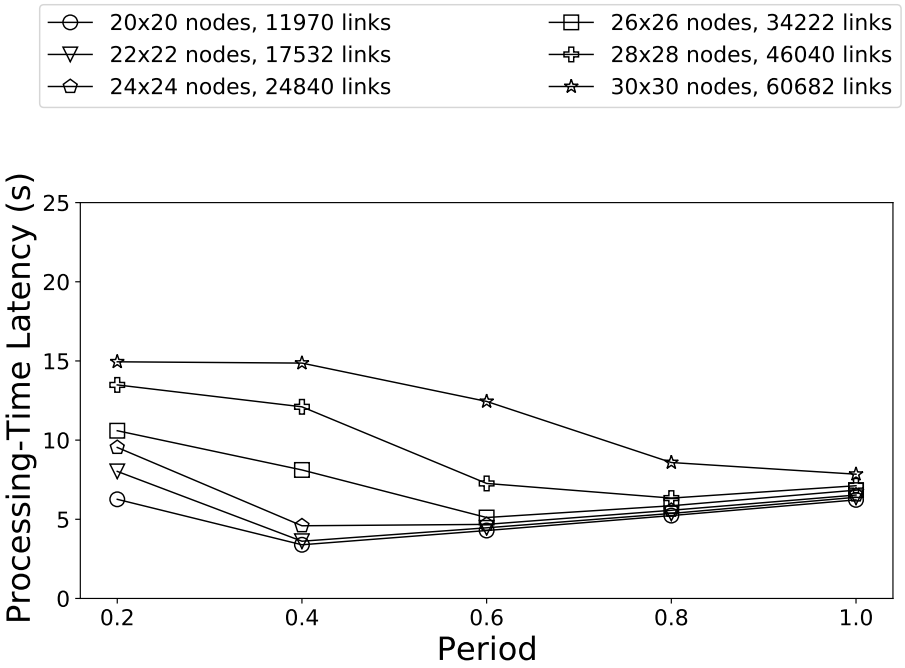


(a) Results of I-DLV-sr-v1

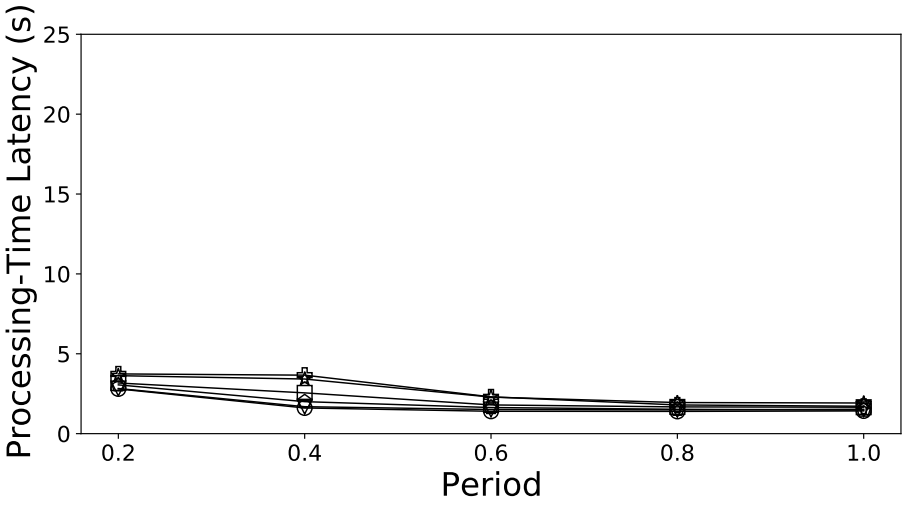


(b) Results of I-DLV-sr-v2

Figure 14.5: Total execution time results on PV-System



(a) Results of I-DLV-sr-v1



(b) Results of I-DLV-sr-v2

Figure 14.6: Latency results on PV-System

14.6 Results on *Metropolitan Area Monitoring*

In this section, we present the results obtained by I-DLV-sr-v1 and I-DLV-sr-v2 on the domain *Metropolitan Area Monitoring*.

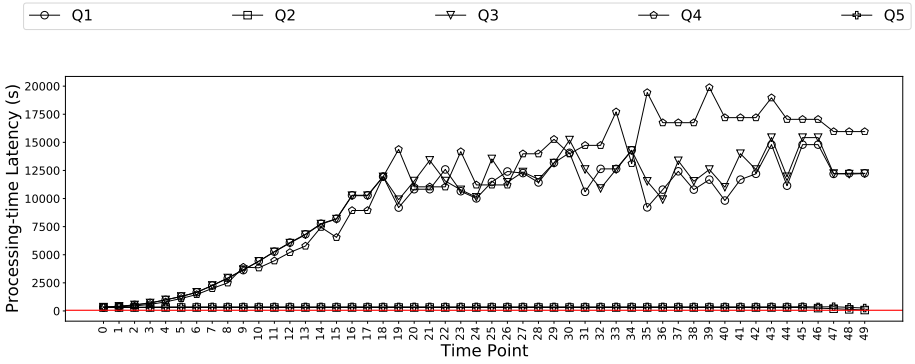
As reported in Table 14.1, almost all queries in this benchmark make use of advanced constructs introduced in the second version of I-DLV-sr. Therefore, to carry out the comparative analysis between I-DLV-sr-v1 and I-DLV-sr-v2, we modified the input encoding and the input streams considered. In such a way, I-DLV-sr-v1 can properly accept and process them. In particular, we replaced every occurrence of the **@now** construct in the encodings with the predicate atom `t_point(T)`.The instances of this predicate atom are sent within the input streams along with the other events. At each time point t, the input stream at hand will contain the fact `t_point(t)`, where t is an integer number representing the time point in terms of minutes. For the sake of fairness, we tested I-DLV-sr-v2 on the original version of *Metropolitan Area Monitoring* queries as well as the modified ones used to test I-DLV-sr-v1.

Figure 14.7 reports the obtained results where:

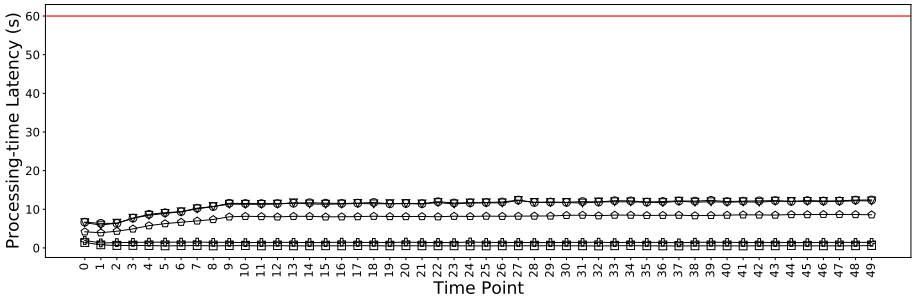
- Subfigure 14.7a plots the results of I-DLV-sr-v1 on the modified queries and input streams;
- Subfigure 14.7b plots the results of I-DLV-sr-v2 on the modified queries and input streams;
- Subfigure 14.7c plots the results of I-DLV-sr-v2 on the original queries and input streams.

The plots depict how the `Latency` (y-axis) varies over the time points (x-axis) per each query. We tested all the systems when receiving input events at each minute; hence, the red line indicates the time limit within which the system has to evaluate one time point before the next one arrives.

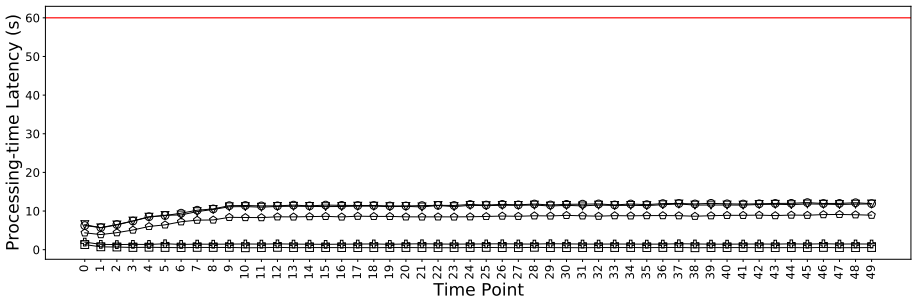
From the plots, we can see no notable difference in running I-DLV-sr-v2 on the original setting (Plot 14.7c) or the modified one (Plot 14.7b). In general, I-DLV-sr-v2 maintains the `Latency` far below the red line for all the queries. For $Q2$ and $Q5$, which make use of fewer and smaller windows w.r.t. the others (i.e., 3 and 4 minutes, respectively), `Latency` is always less than 1.6 seconds for $Q2$ and 2.1 seconds for $Q5$. Higher latency times are registered for $Q1$, $Q3$, and $Q4$, with performance on $Q1$ and $Q3$ basically overlapping: these three queries make use of wider time windows (9 minutes), thus causing the system



(a) Results of I-DLV-sr-v1 on the modified encodings and streams



(b) Results of I-DLV-sr-v2 on the modified encodings and streams



(c) Results of I-DLV-sr-v2 on the original settings

Figure 14.7: Latency variation along time in *Metropolitan Area Monitoring* domain

Query	From	To
$Q1$	339.2 s	14797.0s ($> 4\mathbf{h}$)
$Q2$	80.6 s	306.4s
$Q3$	340.7 s	15420.6s ($> 4\mathbf{h}$)
$Q4$	327.4 s	19873.7s ($> 5\mathbf{h}$)
$Q5$	255.7 s	375.7s

Table 14.2: Ranges of `Latency` times obtained by I-DLV-sr-v1 on *Metropolitan Area Monitoring*

to reason on a considerably large amount of events at each time point. For $Q1$ and $Q3$, `Latency` ranges from 5.6 to 12.8 seconds, whereas for $Q4$ from 3.7 to 9.5 seconds.

Results obtained by I-DLV-sr-v1, reported in Plot 14.7a, are completely different. In general, I-DLV-sr-v1 can never answer within the limit of 60 seconds for none of the five queries. The `Latency` trend of $Q2$ and $Q5$ is linear, while the one of $Q1$, $Q3$, and $Q4$ is fluctuating. Here, `Latency` times are several orders greater than the ones registered for I-DLV-sr-v2. In particular, for most complex queries $Q1$, $Q3$, and $Q4$, I-DLV-sr-v1 processes requests in the order of hours. This can also be observed in Table 14.2, which reports ranges of `Latency` times in numeric format.

Besides smarter stream management, the huge improvement from I-DLV-sr-v1 to I-DLV-sr-v2, demonstrated by this experiment, is attributable to two important enhancement introduced in the second version of the system (see Section 11.2).

On the one hand, I-DLV-sr-v2 allows to change the reasoning granularity, i.e., the user can specify if the system should reason in terms of milliseconds, seconds, minutes, or hours. Therefore, in this domain where reasoning should happen at each minute, I-DLV-sr-v2 simply reasons in terms of minutes. This feature is not available in I-DLV-sr-v1, which is only able to reason in terms of seconds. Specifically, when I-DLV-sr-v1 must reason on the input relative to the minute t, it produces an answer for all the seconds from $t - 1$ to t. This causes a high overwork and the production of a very large output stream, which results in additional management costs.

On the other hand, I-DLV-sr-v2 comes with a new window trigger management. In order to avoid data loss, I-DLV-sr-v1, before triggering a time window, shall ensure that all the events falling into that time window have been received. To do this, it waits for the first event in the scope of the next time window be-

fore evaluating the previous one. This introduces a delay in getting answers that depends form the frequency at which events arrive. In the *Metropolitan Area Monitoring* case, this explains why I-DLV-sr-v1 is not able to process requests with a `Latency` of less than 60 seconds, also for the simplest queries. Given that inputs are received each minute, it must wait at least this amount before evaluating a time point. Although the waiting mechanism is proper to avoid data loss, there exist cases in which this could and should be avoided, especially when fast answers are required. For instance, in some domains, there are no duplicate time points, i.e., the data source sends all the events for a time point at once; in others, the data source can communicate to the system when it has finished sending the events for a time point. I-DLV-sr-v2 allows the user to configure these types of settings so that it can use them to determine when to trigger time windows, reducing the waiting time automatically.

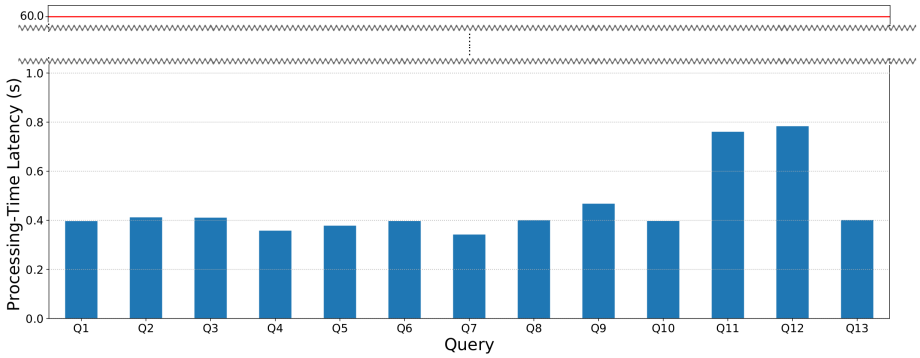


Figure 14.8: Results of I-DLV-sr-v2 on *CityBench* banchmark

14.7 Results on *CityBench*

I-DLV-sr encodings for queries in *CityBench* rely on advanced constructs supported only by I-DLV-sr-v2; hence these cannot be translated in the language supported by I-DLV-sr-v1 or Distributed-SR. For this reason, we tested only I-DLV-sr-v2 on this benchmark.

Figure 14.8 reports the results. In particular, for each of the 13 queries, this figure shows the average `Latency` computed over all time points. In this experiment, I-DLV-sr-v2 receives input events at each minute. As before, the red line represents the amount of time available to evaluate one time point before the next one arrives.

As we can see, I-DLV-sr-v2 is able to answer in less than 0.8 seconds in general, much before events for the next time point arrive (i.e., 1 minute). We also note that $Q11$ and $Q12$ represent the most complex queries as, differently from the others, they require to reason over wide windows of 10 and 20 minutes, respectively.

14.8 Summary

The analysis carried out in this experimental evaluation compare the performance of I-DLV-sr with other logic-based stream reasoners. Due to significant differences in syntax, semantics, and implementation approaches among existing implementations, achieving a fair comparison was challenging. In particular, I-DLV-sr was compared with Distributed-SR, the most recent LARS-based implementation with a distributed architecture. The analysis also focused on as-

sessing the effectiveness of the enhancements introduced in I-DLV-sr. A comparison was made between the first version (I-DLV-sr-v1) and the latest version (I-DLV-sr-v2). The analysis considered three measures: `Total Time`, `#Accepted Requests`, and `Latency`, and five benchmarks: *Content Caching*, *Heavy Join*, *Photo-voltaic System*, *CityBench*, and *Metropolitan Area Monitoring*.

The results in *Content Caching* show that the two versions of I-DLV-sr perform consistently well, while Distributed-SR has a pronounced peak in execution time with a window size of 25 seconds. I-DLV-sr handles all incoming requests, while Distributed-SR misses a significant number of them.

In the *Heavy Join* benchmark, both versions of I-DLV-sr outperform Distributed-SR in terms of `Total Time`. Distributed-SR fails to handle a large number of incoming requests as the number of events per time point increases, while I-DLV-sr handles all requests effectively.

The *Photo-voltaic System* benchmark showed that I-DLV-sr-v2 outperforms I-DLV-sr-v1 in terms of both `Latency` and `Total Time`. The introduced enhancements and optimization in I-DLV-sr-v2 resulted in a significant reduction in `Latency` and `Total Time` compared to I-DLV-sr-v1.

For the *Metropolitan Area Monitoring* benchmark, I-DLV-sr-v2 consistently maintains `Latency` below the specified limit for all queries. In contrast, I-DLV-sr-v1 presents high `Latency` times, particularly for complex queries, making it unable to process requests within the desired time limit. The improvements in I-DLV-sr-v2, including reasoning granularity and window trigger management, significantly enhanced its performance.

Overall, the results indicate that I-DLV-sr-v2 performs better than I-DLV-sr-v1 and Distributed-SR in terms of `Total Time`, `#Accepted Requests`, and `Latency` in all the considered benchmark domains. This demonstrates the enhancements and optimizations introduced in I-DLV-sr-v2 contribute to its improved performance and reliability.

Part V

Related Work and Conclusions

Chapter 15

Related Work

Stream Reasoning has been subject of a number of researchers over the latest years. Nevertheless, there are no standardized formalisms nor techniques for SR to date. The lack of standards makes the comparison among approaches relying on different semantics and technologies rather difficult.

Apart from several relevant solutions stemmed in the semantic-web context [17, 108, 105, 83], the proposals that relate most with ours are the ones based on ASP. With this respect, one well-established is LARS: Logic for Analytic Reasoning over Streams [23], a formal framework enriching ASP with temporal modalities and window operators. LARS theoretically consists of a full-fledged non-monotonic formalism for reasoning over streams; indeed, the full language is computationally intractable. Current implementations, such as Laser [19], Ticker [24], and its recent distributed version [57], support smaller, yet practically relevant fragments. For performance reasons, Laser handles only negation-stratified and stream-stratified programs, i.e., recursion is not supported if it involves negation or windows [21]. Ticker comes with two evaluation modes: one makes use of the state-of-the-art ASP system clingo [71] as back-end, and is intended for stratified programs, i.e., programs having only a single model; the other mode uses incremental truth maintenance techniques under ASP semantics, and, in case of multiple solutions, computes and maintains one single model randomly chosen. Distributed-SR is an additional version of Ticker, recently released; in order to increase the throughput, it implements an interval-based semantics of LARS that relies on Ticker as internal engine. Moreover, it adopts distributed computation, at the price of disabling the support for recursion through window operators. All Ticker versions require any

variable appearing in the scope of a window atom to be "guarded" by some standard atom including it; such variables are grounded upfront in a so-called pre-grounding phase.

I-DLV-sr supports the ASP fragment stratified w.r.t. negation, which is extended with streaming literals over temporal intervals, and other constructs such as the **@now** term, trigger rules and temporary rules. Recursion involving streaming literals is allowed. The implementation is designed for supporting incremental evaluation of I-DLV-sr programs, thanks to the integration with $\mathcal{I}^2$-*DLV* [34, 86], and parallel/distributed computation, thanks to the integration with *Flink* [46, 113, 85]. On the one hand, incrementality allows to efficiently evaluate logic programs at each time point and to avoid the need for pre-grounding. Also, apart from safety, no restriction is required for variables in streaming literals. On the other hand, parallelism and distribution pave the way to the efficient evaluation of streaming atoms. During execution, each operator in the *Flink* dataflow can have one or more operator subtasks. The latter are independent from each other; hence, they can be executed in different threads and possibly on different machines or containers. In case of recursive subprograms, distribution is limited by evaluating the involved streaming atoms via single-threaded, undistributed operators.

BigSR [116] is a distributed stream reasoner that, like I-DLV-sr, makes use of a stream processor to deal with data streams. It is released in two versions built on top of the state-of-the-art stream processors Spark Streaming [2] and *Flink*, respectively. BigSR implements internal reasoning algorithms to compute the semantics, and significantly differs from I-DLV-sr on the supported language, the input format and the provided features. In particular, it is oriented towards ontology-based reasoning and accepts RDF input streams. Moreover, input programs fall in the positive fragment of plain LARS and can only contain the *in* window operator. In addition, depending on the stream processor, further limitations are required. The version relying on Spark Streaming accepts stratified programs that can be recursive but have only global windows, i.e., the rules must share the same window operator. The version built on top of *Flink*, instead, accepts non-recursive stratified programs, but with global windows at rule scope, i.e., all literals in a rule must share the same window operator. In this latter version, such limitations are due to the BigSR usage of the multi-core/distributed nature of *Flink*, that makes the handling of synchronization of clocks, task progress and window trigger mechanisms difficult. Furthermore, I-DLV-sr and BigSR differs for the adopted notion of time, as the former relies

on *event time* notion while the latter on the *ingestion time* one [85].

The topics dealt in this thesis have some connections also with temporal logic programming [59, 3]. Temporal reasoning has been investigated also in the ASP community, where solutions that extend Equilibrium Logic, a logical characterization of ASP, with Linear-Time Temporal Logic (LTL) operators have been recently proposed [5, 32, 31]. Similarly to LARS, the herein presented proposal is based on the concept of window and on a discretized notion of time. In particular, the temporal operators included in I-DLV-sr language are intended to be used within the scope of a time-based window and currently, they do not allow to reason about the whole timeline or the future. Conversely, LTL does not have the notion of window, but permits to reason also about the whole timeline and the future [22, 5, 23].

When designing I-DLV-sr, Spark Streaming and Apache Storm [87, 114, 60] were evaluated as alternative stream processors. Spark Streaming was excluded as, when receiving live input streams, it divides the data into batches, which are lately processed by the Spark engine to generate the final stream in batches. On the other hand, *Flink* does not require the a-priori creation of batches; it works in real-time, record by record, rather than batch by batch. Apache Storm can handle data processing record by record but, it does not provide the event time processing and does not natively offers the exactly-once semantics as instead *Flink* does. This semantics ensures to I-DLV-sr that each incoming event affects the final outcome exactly once and, even in case of a machine or software failure, there is no data duplication nor unprocessed data.

Chapter 16

Ongoing Work - Beyond Deterministic Programs

As illustrated in Chapter 9, the latest release of I-DLV-sr extends ASP with constructs for reasoning over streams. The basic construct of the accepted input language is the *streaming atom*, which allows to access the stream. Streaming atoms consist of a classical ASP atom, an operator, and a lookup set. The first indicates the "event" of interest, the second defines the semantics to apply, and finally, the third identifies the part of the stream to consider in the semantics evaluation. Currently, we support 4 different operators such us "always", "count", "at least", and "at most". A streaming literal is either a streaming atom or a streaming atoms preceded by the symbol `not`, i.e., the negation as failure. Streaming literals can be used within the body of rules, which can have three different forms: *normal*, *trigger*, *temporary*. Normal rules behave similarly to classical ASP rules, while trigger and temporary rules have peculiar behaviors. The former is evaluated with a frequency specified by the user rather than at each time point. The latter only derives temporary information, meaning it last only for the time point where it is derived. The system version presented in this work only supports the fragment of ASP consisting of programs that are non-disjunctive and stratified w.r.t. negation, i.e., programs that have only one solution at each time point.

With the aim of support a wider class of programs, we are working to enhance the expressive power of I-DLV-sr. In particular, in this chapter, we present an extension of the language obtained by relaxing the constraint on stratified

negation and expanding the set of accepted rules. The chapter is organized as follows. Section 16.1 describes the extension of I-DLV-sr language and Section 16.2 illustrates some examples of problems that can be modeled thanks to such extension.

16.1 Extending the Language Towards Non-determinism

Beside the rule forms introduced in Subsection 9.2.3, a rule r can also have one of the following forms:

5. $a_1 | \cdots | a_h :- l_1, \ldots, l_b.$

6. $:\sim\ l_1, \ldots, l_b.[w@s]$

where: *(i)* $b \geq 0$, $h \geq 0$; *(ii)* $a_1, \cdots, a_h$ is a disjunction of predicate atoms; *(iii)* $l_1, \ldots, l_b$ is a conjunction of streaming literals; *(iv)* w and s are terms standing for a weight and a level, respectively. If $h = 0$, the rule is referred to as a *strong constraint*, if $h = 1$, the rule is a *normal rule* as stated in Subsection 9.2.3, finally, if $h > 1$, the rule is referred to as a *disjunctive rule*. A rule of form (6) is a *weak constraint*. These rules behaves like the ones originally defined in ASP with the only difference that they can features streaming literals in their bodies.

The formal definition of the semantics extended with disjunction and non-stratified negation is an ongoing work. Hereafter, we provides some intuitions.

The idea is that at each evaluation time point, we fix a single model among the possible ones. This allow us to avoid considering multiple equivalent solutions when evaluating non-degenerate streaming literals over non-deterministic predicates. The selected model is the first (optimal) one provided by the ASP solver. The streaming model is composed of *deterministic* and *non-deterministic* predicate atoms. The former are predicate atoms coming from the input stream or derived through deterministic predicates, while the latter are predicate atoms derived through non-deterministic predicates. A streaming model always includes the deterministic predicate atoms, even if the evaluated program has no solution. This is because we do not want to retract input events or events directly derived from them.

Example 16.1.1 Let us consider the stream $\Sigma_5 = \langle\{\ \{c\}_0,\ \{c\}_1,\ \{c, a\}_2\ \}\rangle$ and the following I-DLV-sr program Γ_{10}:

```
r1  a | b :- c.
r2  :- a in {1}, c.
```

The streaming model of Γ_{10} on Σ_5 is the set $\{c, a\}$. In the following, we intuitively describe how this output is obtained.

When evaluating the program over S_0, there are two possible solutions: $\{c, a\}$ and $\{c, b\}$. Since we assume to fix one model at each time point, S'_0 can be either $\{c, a\}$ or $\{c, b\}$.

On the one hand, assume we select $S'_0 = \{c, a\}$. Γ_{10} is inconsistent at the time point 1 due to r_2. We recall that, at each time point, I-DLV-sr programs are evaluated over the predicate atoms provided by the input stream along with the ones derived at previous time points (see Section9.3). Hence, at the time point 1, Γ_{10} is evaluated on the partial stream $\langle S'_0, S_1\rangle$. In this case, `a in {1}` holds from S'_0, and `c` holds from S_1; thus, the strong constraint r_2 fires making Γ_{10} inconsistent at 1. Since the semantics defined for this type of programs does not retract deterministic predicates atoms, we have that $S'_1 = \{c\}$. Then, the evaluation of Γ_{10} over $\langle S'_0, S'_1, S_2\rangle$ has only one solution, i.e., $\{c, a\}$. Eventually, the streaming model of Γ_{10} on Σ'_2 is the set $\{c, a\}$.

On the other hand, assume we select $S'_0 = \{c, b\}$. Γ_{10} has two possible solutions on $\langle S'_0, S_1\rangle$, i.e., $\{c, a\}$ and $\{c, b\}$. If we select the former, $S'_1 = \{c, a\}$, the program is inconsistent at the time point 2. However, since the predicate atoms in S_2 are not retracted by definition, the streaming model of Γ_{10} on Σ_5 is again the set $\{c, a\}$. If we select the latter, $S'_1 = \{c, b\}$, the program has only one solution at the time point 2, i.e., $\{c, a\}$. Thus, the streaming model of Γ_{10} on Σ_5 is always the set $\{c, a\}$. □

16.2 Modeling Examples

Thanks to this enhancement, I-DLV-sr is able to model non-deterministic problems like the ones presented in the next examples.

Example 16.2.1 Let us consider the *Content Retrieval* example defined in [24]. This example aims to model the problem of retrieving contents in a network where items can be cached and requested at every node. When an item is requested to a node where it is unavailable, that item must be retrieved from the network by choosing the best node among the available ones. This problem is encoded with the following I-DLV-sr program.

```
r1  need(I,N) :- item(I), node(N), req(I,N) in [n].
r2  avail(I,N) :- item(I), node(N), cache(I,N) in [n].
r3  get(I,N,M) :- source(I,N,M), not nGet(I,N,M).
r4  nGet(I,N,M) :- node(M), get(I,N,M0), M<>M0.
r5  nGet(I,N,M) :- source(I,N,M0), qual(M,L), M<>M0, L<L0, qual(M0,L0),
    source(I,N,M).
r6  source(I,N,M) :- need(I,N), avail(I,M), reach(N,M), not avail(I,N).
r7  reach(N,M) :- conn(N,M).
r8  reach(N,M) :- reach(N,M0), conn(M0,M), M0<>M, N<>M.
r9  conn(N,M) :- edge(N,M), not down(M) always in [n].
r10 qual(N,L) :- lev(L0), node(N), L0<L, lev(L), qLev(N,L) in [n], not
    qLev(N,L0) in [n].
```

If a user has recently requested item `I` at node `N` (rule r_1), it is either available at `N` (r_2) or has to be retrieved from some other node `M` ($r_3 - r_6$). A single node is selected (r_3) that provides the best quality level (e.g. connection speed) among all reachable nodes having `I` (r_5). Connecting paths (r_7, r_8) work unless the end node of an edge was down during the last n time points (r_9). Finally, nodes repeatedly report their quality level, among which the best recent value is selected (r_10). □

Example 16.2.2 In a city, the 5th International Festival of Comics and Animated Films will take place. The festival lasts for 3 days, starting from 9 a.m. to 8 p.m., and includes a number of indoors and/or outdoor activities, exhibitions, and exposures. Each event has a duration expressed in minutes, a popularity level that ranges from 0 to 5 (where 0 means *unpopular* and 5 means *popular*), and the best accommodation, i.e., *indoor*, *outdoor* or *both*. The festival location consists of *pavilions* and *open spaces*, each with an approximate capacity categorized as *small*, *medium* or *large*. The organization committee has drafted a schedule of the events on the following criteria:

- **duration**: a set of events is scheduled in a day to fit the total duration within the opening hours, considering 10-minute pause between the end of one event and the start of a new one in the same place;
- **accommodation**: an event is assigned to a place based on the popularity level of the event, the best accommodation of the event, and the capacity of the space. Specifically, events with a popularity level of 0 or 1 are located in small-sized places, events with a popularity level of 2 or 3 are located in medium-sized places, and events with a popularity level of 4 or 5 are located in large-sized places. Furthermore, if the best accommodation of an event is one between *indoor* and *outdoor*, it must be located in

pavilions and open spaces, respectively; otherwise, only the place capacity is considered.

To ensure the success of the festival, the organizers want the schedule to be automatically rearranged based on changing weather conditions, specifically if it is raining. The following criteria must be used for rescheduling events:

- If the rain intensity is *light*, outdoor events are preferably postponed to the same day they were originally scheduled. If not possible, they should be assigned to a free pavilion if available or postponed to the closest following day;
- If the rain intensity is *moderate*, outdoor events should be assigned to a free pavilion if possible, or else postponed to the closest following day;
- If the rain intensity is *heavy*, outdoor events must be postponed to the closest following day.

When defining a new schedule, it should adhere to the criteria set by the organization committee and be as similar as possible to the previous one.

The I-DLV-sr program modeling this problem is reported next.

```
r1  day_to_plan(Day):- day(Day), current_day(C_day), Day >= C_day.
r2  taken_place_events(ID) :- taken_place_events(Day, ID, Place, Order).
r3  event_to_schedule(Event):- event(Event,_,_,_,_),
      not taken_place_events(Event).

    % Guess a possible schedule for the events
r4  schedule_in_day(Day,Event,Place) |
      not_schedule_in_day(Day,Event,Place) :- it_is_raining,
      day_to_plan(Day), event_to_schedule(Event),
      suitable_place(Event,Place).

r5  scheduled_or_already_done(Event):-schedule_in_day(_,Event,_).
r6  scheduled_or_already_done(Event):-taken_place_events(_,Event,_).
r7  :-event(Event,_,_,_,_), not scheduled_or_already_done(Event).

r8  :- schedule_in_day(Day1,Event, _), schedule_in_day(Day2,Event,_),
      Day1<>Day2.
r9  :- schedule_in_day(_,Event, Place1), schedule_in_day(_,Event, Place2),
      Place1<>Place2.

r10 :~ schedule_in_day(Day,_,Place), schedule_in_day(Day,_,Place2),
      Place<>Place2.  [1@1]

    % Guess a possible schedule order
```

```
r11 order_position(Day, Place, 0) :-
      number_of_events_per_day_in_place(Day, Place, _).
r12 order_position(Day, Place, X) :- order_position(Day, Place, Y),
      number_of_events_per_day_in_place(Day, Place, Number_of_events),
      X=Y+1,X<Number_of_events.
r13 already_scheduled_timeslot(Day, Place, Order) :- day_to_plan(Day),
      taken_place_events(Day, _, Place, Order).
r14 schedule_order(Day,Event,Place,Order_position) |
      non_schedule_order(Day,Event,Place,Order_position):-
      schedule_in_day(Day,Event,Place),
      order_position(Day, Place, Order_position),
      not already_scheduled_timeslot(Day, Place, Order_position).

r15 :- schedule_order(Day,Event,_,Order_position),
      schedule_order(Day,Event,_,Order_position2),
      Order_position2<>Order_position.
r16 :- schedule_order(Day,Event,_,Order_position),
      schedule_order(Day,Event2,_,Order_position), Event<>Event2.

    %% Basic requirement 1
    % the total duration of events scheduled in a day must mostly fit
    % the opening hours per day
r17 event_in_day_and_place(Day, Event, Place):-
      schedule_in_day(Day, Event, Place).
r18 event_in_day_and_place(Day, Event, Place):-
      taken_place_events(Day, Event, Place, _), day_to_plan(Day).

r19 number_of_events_per_day_in_place(Day,Place,Number_of_events):-
      day(Day), place(Place,_,_),
      #count{Event: event_in_day_and_place(Day, Event, Place)} =
      Number_of_events.
r20 duration_scheduled_events_per_day_in_place(Day, Place, Tot_duration):-
      number_of_events_per_day_in_place(Day, Place, Number_of_events),
      Number_of_events>0,
      #sum{Duration,Event: event_in_day_and_place(Day,Event,Place),
      event(Event,_, Duration,_,_)} = Sum_durations,
      Tot_duration=Sum_durations+10*(Number_of_events-1).

r21 penalty_per_day_and_place(Day,Place, Cost):-
      duration_scheduled_events_per_day_in_place(Day, Place,
      Tot_duration), hours_per_day(H), H_min=H*60,
      Cost=Tot_duration-H_min, Cost>0.
r22 :~ penalty_per_day_and_place(Day,Place, Cost). [Cost@1]

    %% Basic requirement 2
    % the new schedule must be as similar as possible to the previous
    schedule
```

```
r23 rescheduled_event_at(Event,PrevDay,PrevPlace,NewDay,NewPlace):-
      schedule_in_day(PrevDay,Event,PrevPlace) in {1},
      schedule_in_day(NewDay,Event,NewPlace),PrevDay<>NewDay.
r24 rescheduled_event_at(Event,PrevDay,PrevPlace,NewDay,NewPlace):-
      schedule_in_day(PrevDay,Event,PrevPlace) in {1},
      schedule_in_day(NewDay,Event,NewPlace),PrevPlace<>NewPlace.

r25 :~ rescheduled_event_at(Event,PrevDay,PrevPlace,NewDay,NewPlace),
      Cost=NewDay-PrevDay, NewDay>PrevDay.  [Cost@3]
r26 :~ rescheduled_event_at(Event,PrevDay,PrevPlace,NewDay,NewPlace),
      Cost=PrevDay-NewDay, NewDay<PrevDay.  [Cost@3]

r27 event_today(Event,Place):- current_day(Day),
      schedule_in_day(Day,Event,Place).
r28 prev_scheduled(Event,Event2):-
      schedule_order(Day,Event,Place,Order_position),
      schedule_order(Day,Event2,Place,Order_position2),
      Order_position2<Order_position.
r29 num_prev_scheduled(Event,Num):- schedule_in_day(_,Event,_),
      #count{PrevEvent:  prev_scheduled(Event,PrevEvent)}=Num.
r30 starting_hour(Event,Starting_hour) :-
      schedule_in_day(_,Event,Place), Prev_event_duration=
      #sum{Duration,PrevEvent:  prev_scheduled(Event,PrevEvent),
        event(PrevEvent,_,Duration,_,_)},
      num_prev_scheduled(Event,Num),
      Starting_min=( Prev_event_duration + ( 10 * Num ) ),
      Hour=( Starting_min / 60 ), Starting_hour=@now.hour+Hour.

r31 event_at_new_time(Event,PrevH,NewH):-
      starting_hour(Event,PrevH) in {1}, starting_hour(Event,NewH),
      PrevH<>NewH.
r32 :~ event_at_new_time(Event,PrevH,NewH), Cost=NewH-PrevH,
      NewH>PrevH.[Cost@2]
r33 :~ event_at_new_time(Event,PrevH,NewH), Cost=PrevH-NewH,
      NewH<PrevH.[Cost@2]

r34 :~ rescheduled_event_at(Event,PrevDay,PrevPlace,NewDay,NewPlace),
      PrevPlace<>NewPlace.[1@1]

    % Rescheduling based on rain level
r35 :- it_is_raining, event_at_new_time(Event,PrevH,NewH), NewH<PrevH.

r36 can_be_reassigned(Event):-
      rescheduled_event_at(Event,PrevDay,PrevPlace,NewDay,NewPlace),
      PrevDay<>NewDay, event(Event, _, EDuration, _, both),
      duration_scheduled_events_per_day_in_place(PrevDay, Place,
      Duration), suitable_place(Event,Place), place(Place, pavilion, _),
```

```
    hours_per_day(H), H_min=H*60, Duration+10+EDuration<H_min.

r37 :~ rain_level(light),
    rescheduled_event_at(Event,PrevDay,PrevPlace,NewDay,NewPlace),
    place(PrevPlace, open_space, _), PrevDay<>NewDay.  [1@4]
r38 :~ rain_level(light), event(Event,_,_,_,_),
    can_be_reassigned(Event).  [1@3]

r39 :~ rain_level(moderate), event(Event,_,_,_,_),
    can_be_reassigned(Event).  [1@4]

r40 :- rain_level(heavy), current_day(Day),
    schedule_in_day(Day,_,open_space).
```

Rules $r_1 - r_3$ are auxiliary rules that define the days and events to be considered. r_4 is a disjunctive rule that *guesses* a possible `Day` and `Place` for each `Event` if it is raining (`it_is_raining`). Rules $r_5 - r_7$ ensure that all events of the festival are either scheduled or already done. Rules r_8 and r_9 express that an `Event` cannot be scheduled more the ones or assigned to more than one place, respectively. Rule r_{10} is a weak constraint that minimize the used places. Rules $r_{11} - r_{14}$ guess a possible order of the events planned for each day. Rules r_{15} and r_{16} are two constraints that ensure each event is associated with only one time slot and, viceversa, a time slot has only one event associated. Rules $r_{17} - r_{22}$ model the first basic requirement regarding the total duration of events scheduled in a day. Specifically, rules r_{19} and r_{20} calculate the total time of the events scheduled in a day and place. This amount is minimized by rule r_{21} and the weak constraint r_{22}. Rules $r_{23} - r_{34}$ model the second basic requirement of similarity between subsequent schedules. This task involves non-degenerate streaming literals over non-deterministic predicates and several weak constraints. In particular, rules r_{23} and r_{24} access the schedule guessed at the previous time point and compare it with the one guessed at the current time point. Rules r_{25} and r_{26} are the weak constraints with the higher priority that penalize the number of days an event is postponed. Rules r_{32} and r_{33} are the weak constraint with medium priority that penalize the number of hours an event is postponed within the same day it was originally planned. Finally, rule r_{34} is the weak constraint with the lower priority that penalizes for events assigned to a different place. Rules $r_{35} - r_{40}$ model the rescheduling criteria based on the rain level. Rule r_{35} is a generic constraints that specify it is not possible to anticipate an outdoor event in case of raining. For light rain, rule r_{37} applies a penalty, with higher priority, each time an outdoor event is post-

poned to another day, while rule r_{38} applies a penalty each time an event can be reassigned to another place but is instead postponed to another day. Rule r_{39} is similar to r_{38} but fires in the case of moderate rain and has a higher priority level. Finally, rule r_{40} prevents the scheduling of any event on the current day in an open space.

□

Both these example are non-deterministic. Example16.2.1 involves non-stratified negation (see predicates `get/3` and `nGet/3` in rules r_3 and r_4). Example 16.2.2 heavily uses disjunctive rules to implement a scheduling task, which is modeled as a minimization problem using weak constraints. Strong constraints are also used to check the consistency of the guessed solution w.r.t. the problem definition. It is worth noting this version of the language also supports non-degenerate streaming literals over non-deterministic predicates, i.e., predicates defined by disjunctive or non-stratified rules (see r_{23}, r_{24}, and r_{31}). Furthermore, it allows recursion through this type of streaming literals.

Chapter 17

Conclusions and Future Work

To date, a solution capable of fully covering all the SR requirements [52] is still missing. This thesis presents a novel ASP-based SR solution designed with the aim of meeting SR demand and overcoming main limitations of currently proposed solutions. We started studying the state-of-the-art, currently available solutions and emerging applications. We thus conceived I-DLV-sr, leveraging on benefits stemming from the SP and ASP worlds [42]. In order to fulfill the SR requirements, I-DLV-sr relies on a tight interaction between the ASP system $\mathcal{I}^2$-*DLV* and an SP-based *Flink* application. Moreover, I-DLV-sr has been designed to be easily extensible in both language and architecture. Indeed, proper extensions, inspired by Smart City scenarios, have been recently included proving its flexibility [44]. I-DLV-sr features were awarded during the *Stream Reasoning Hackathon 2021*[1] [117], an event part of the *Stream Reasoning Workshop 2021*, ranking the first position. For assessing reliability, performance, and scalability, we tested I-DLV-sr on both real-world and synthetic benchmarks. The conducted experimental activity confirmed that combining state-of-the-art solutions from SP and ASP permit efficient and effective reasoning over dynamic data streams. I-DLV-sr has undergone continuous optimization and re-engineering processes [44]. We released two stable versions, namely I-DLV-sr-v1 and I-DLV-sr-v2, available at the dedicated repository [43].

[1] `https://streamreasoning.org/events/stream-reasoning-hackathon-2021/` (url date: 18/01/2022)

We plan to enrich I-DLV-sr with further linguistic constructs and features relevant to practical SR scenarios. We are working to extend the language as well as the system to deal with programs having more than one solution. This enhancement would expand the class of accepted programs, also spanning the ones that are disjunctive or non-stratified w.r.t. negation, thus increasing the domains in which I-DLV-sr can be employed. In this respect, on the one hand, we recently integrated into I-DLV-sr the incremental ASP reasoner Incremental-*DLV2* [39]. Incremental-*DLV2* is the incremental version of the monolithic "ground & solve" ASP system *DLV2* [10], which integrates $\mathcal{I}^2$-*DLV* [34] as incremental grounder and *wasp* [11] as solver. This version employs Incremental-*DLV2* in place of $\mathcal{I}^2$-*DLV* to perform reasoning tasks as described in Chapter 10. On the other hand, we are studying and designing proper means to handle non-deterministic streams resulting from the evaluation of such classes of programs. Preliminary results of this research line have been reported in Chapter 16.

Moreover, we aim to further facilitate the applicability of I-DLV-sr in the SR field. SR applications are typically characterized by many data sources, which can have different natures and adopt different data models. Therefore, we are working to enable I-DLV-sr to manage multiple data sources and formats. In particular, we are studying techniques for properly joining events coming from different types of input stream. We are thus facing the need of preserving the chronological order among events, and proper means to synchronize data sources are necessary. In addition, in order to identify the most commonly used data formats, we are analyzing different real-world application domains.

Eventually, we plan to enrich I-DLV-sr with the capability of hybridly combining its deductive nature with inductive reasoning over streaming data. To this end, we are considering ML, which among the inductive techniques, is particularly suitable for efficiently recognizing meaningful patterns and connections from several observations.

Bibliography

[1] Apache samza - a distributed stream processing framework, `https://samza.apache.org/`

[2] Apache spark, `https://spark.apache.org/docs/latest/streaming-programming-guide.html`

[3] Abadi, M., Manna, Z.: Temporal logic programming. J. Symb. Comput. **8**(3), 277–295 (1989)

[4] Adrian, W.T., Alviano, M., Calimeri, F., Cuteri, B., Dodaro, C., Faber, W., Fuscà, D., Leone, N., Manna, M., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV: advancements and applications. Künstliche Intell. **32**(2-3), 177–179 (2018)

[5] Aguado, F., Cabalar, P., Diéguez, M., Pérez, G., Schaub, T., Schuhmann, A., Vidal, C.: Linear-time temporal answer set programming. Theory Pract. Log. Program. **23**(1), 2–56 (2023)

[6] Ajileye, T., Motik, B., Horrocks, I.: Streaming partitioning of RDF graphs for datalog reasoning. In: ESWC. Lecture Notes in Computer Science, vol. 12731, pp. 3–22. Springer (2021)

[7] Ali, M.I., Gao, F., Mileo, A.: Citybench: A configurable benchmark to evaluate RSP engines using smart city datasets. In: ISWC (2). Lecture Notes in Computer Science, vol. 9367, pp. 374–389. Springer (2015)

[8] Allocca, C., Alviano, M., Calimeri, F., Costabile, R., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Leone, N., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., Zangari, J.: Reasoning over ontologies with DLV. In: IC3K. Communications in Computer and Information Science, vol. 1222, pp. 114–136. Springer (2018)

[9] Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The fourth answer set programming competition: Preliminary report. In: LPNMR. Lecture Notes in Computer Science, vol. 8148, pp. 42–53. Springer (2013)

[10] Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: LPNMR. Lecture Notes in Computer Science, vol. 10377, pp. 215–221. Springer (2017)

[11] Alviano, M., Dodaro, C., Leone, N., Ricca, F.: Advances in WASP. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 40–54. Springer (2015)

[12] Alviano, M., Leone, N., Veltri, P., Zangari, J.: Enhancing magic sets with an application to ontological reasoning. Theory Pract. Log. Program. **19**(5-6), 654–670 (2019)

[13] Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW. pp. 635–644. ACM (2011)

[14] Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann (1988). https://doi.org/10.1016/b978-0-934613-40-8.50006-3, `https://doi.org/10.1016/b978-0-934613-40-8.50006-3`

[15] Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006)

[16] Babu, S., Widom, J.: Continuous queries over data streams. SIGMOD Rec. **30**(3), 109–120 (2001)

[17] Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-SPARQL: a continuous query language for RDF data streams. Int. J. Semantic Comput. **4**(1), 3–25 (2010)

[18] Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: ESWC (1). Lecture Notes in Computer Science, vol. 6088, pp. 1–15. Springer (2010)

[19] Bazoobandi, H.R., Beck, H., Urbani, J.: Expressive stream reasoning with laser. In: ISWC (1). Lecture Notes in Computer Science, vol. 10587, pp. 87–103. Springer (2017)

[20] Beck, H., Bierbaumer, B., Dao-Tran, M., Eiter, T., Hellwagner, H., Schekotihin, K.: Stream reasoning-based control of caching strategies in CCN routers. In: ICC. pp. 1–6. IEEE (2017)

[21] Beck, H., Dao-Tran, M., Eiter, T.: Answer update for rule-based stream reasoning. In: IJCAI. pp. 2741–2747. AAAI Press (2015)

[22] Beck, H., Dao-Tran, M., Eiter, T.: Equivalent stream reasoning programs. In: IJCAI. pp. 929–935. IJCAI/AAAI Press (2016)

[23] Beck, H., Dao-Tran, M., Eiter, T.: LARS: A logic-based framework for analytic reasoning over streams. Artif. Intell. **261**, 16–70 (2018)

[24] Beck, H., Eiter, T., Folie, C.: Ticker: A system for incremental asp-based stream reasoning. Theory Pract. Log. Program. **17**(5-6), 744–763 (2017)

[25] Bejeck, B.: Kafka Streams in Action: Real-time apps and microservices with the Kafka Streams API. Simon and Schuster (2018)

[26] Bichler, M., Morak, M., Woltran, S.: The power of non-ground rules in answer set programming. Theory Pract. Log. Program. **16**(5-6), 552–569 (2016)

[27] Bonte, P., Turck, F.D., Ongenae, F.: Bridging the gap between expressivity and efficiency in stream reasoning: a structural caching approach for iot streams. Knowl. Inf. Syst. **64**(7), 1781–1815 (2022)

[28] Branco, P., Torgo, L., Ribeiro, R.P.: A survey of predictive modeling on imbalanced domains. ACM Comput. Surv. **49**(2), 31:1–31:50 (2016)

[29] Brandt, S., Kalayci, E.G., Ryzhikov, V., Xiao, G., Zakharyaschev, M.: Querying log data with metric temporal logic. J. Artif. Intell. Res. **62**, 829–877 (2018)

[30] Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)

[31] Cabalar, P., Kaminski, R., Morkisch, P., Schaub, T.: telingo = ASP + time. In: LPNMR. Lecture Notes in Computer Science, vol. 11481, pp. 256–269. Springer (2019)

[32] Cabalar, P., Kaminski, R., Schaub, T., Schuhmann, A.: Temporal answer set programming on finite traces. Theory Pract. Log. Program. **18**(3-4), 406–420 (2018)

[33] Calbimonte, J., Corcho, Ó., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: ISWC (1). Lecture Notes in Computer Science, vol. 6496, pp. 96–111. Springer (2010)

[34] Calimeri, F., Ianni, G., Pacenza, F., Perri, S., Zangari, J.: Incremental answer set programming with overgrounding. TPLP **19**(5-6), 957–973 (Sep 2019). https://doi.org/10.1017/s1471068419000292, `http://dx.doi.org/10.1017/S1471068419000292`

[35] Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., Schaub, T.: Asp-core-2 input language format. Theory Pract. Log. Program. **20**(2), 294–309 (2020)

[36] Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: External computations and interoperability in the new DLV grounder. In: AI*IA. Lecture Notes in Computer Science, vol. 10640, pp. 172–185. Springer (2017)

[37] Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: I-DLV: the new intelligent grounder of DLV. Intelligenza Artificiale **11**(1), 5–20 (2017)

[38] Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artif. Intell. **231**, 151–181 (2016)

[39] Calimeri, F., Ianni, G., Pacenza, F., Perri, S., Zangari, J.: Asp-based multi-shot reasoning via DLV2 with incremental grounding. In: PPDP. pp. 2:1–2:9. ACM (2022)

[40] Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. Theory Pract. Log. Program. **14**(1), 117–135 (2014)

[41] Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The third answer set programming competition: Preliminary report of the system competition track. In: LPNMR. Lecture Notes in Computer Science, vol. 6645, pp. 388–403. Springer (2011)

[42] Calimeri, F., Manna, M., Mastria, E., Morelli, M.C., Perri, S., Zangari, J.: I-dlv-sr: A stream reasoning system based on I-DLV. Theory Pract. Log. Program. **21**(5), 610–628 (2021)

[43] Calimeri, F., Mastria, E., Perri, S., Zangari, J.: The i-dlv-sr system (2022), `https://github.com/DeMaCS-UNICAL/I-DLV-sr`

[44] Calimeri, F., Mastria, E., Perri, S., Zangari, J.: The stream reasoning system i-dlv-sr: Enhancements and applications in smart cities. In: RuleML+RR. Lecture Notes in Computer Science, vol. 13752, pp. 38–53. Springer (2022)

[45] Calimeri, F., Perri, S., Zangari, J.: Optimizing answer set computation via heuristic-based decomposition. Theory Pract. Log. Program. **19**(4), 603–628 (2019)

[46] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4) (2015)

[47] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)

[48] Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. **44**(3), 15:1–15:62 (2012)

[49] D'Aniello, G., Gaeta, M., Orciuoli, F.: An approach based on semantic stream reasoning to support decision processes in smart cities. Telematics Informatics **35**(1), 68–81 (2018)

[50] Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3), 374–425 (2001)

[51] Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., Weinzierl, A.: Omiga : An open minded grounding on-the-fly answer set solver. In: JELIA. Lecture Notes in Computer Science, vol. 7519, pp. 480–483. Springer (2012)

[52] Dell'Aglio, D., Valle, E.D., van Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook. Data Sci. **1**(1-2), 59–83 (2017)

[53] Do, T.M., Loke, S.W., Liu, F.: Answer set programming for stream reasoning. In: Canadian Conference on AI. Lecture Notes in Computer Science, vol. 6657, pp. 104–109. Springer (2011)

[54] Dodaro, C., Eiter, T., Ogris, P., Schekotihin, K.: Managing caching strategies for stream reasoning with reinforcement learning. Theory Pract. Log. Program. **20**(5), 625–640 (2020)

[55] Doyle, J.: A truth maintenance system. Artificial Intelligence **12**(3), 231–272 (1979). https://doi.org/https://doi.org/10.1016/0004-3702(79)90008-0, https://www.sciencedirect.com/science/article/pii/0004370279900080

[56] Dustdar, S., Nastic, S., Scekic, O.: Smart Cities - The Internet of Things, People and Systems. Springer (2017)

[57] Eiter, T., Ogris, P., Schekotihin, K.: A distributed approach to LARS stream reasoning (system paper). Theory Pract. Log. Program. **19**(5-6), 974–989 (2019)

[58] El-Kafrawy, P.M., Bennawy, M.: Walk through event stream processing architecture, use cases and frameworks survey. In: Robotics and AI for Cybersecurity and Critical Infrastructure in Smart Cities, Studies in Computational Intelligence, vol. 1030, pp. 57–71. Springer (2022)

[59] Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 995–1072. Elsevier and MIT Press (1990)

[60] Evans, R.: Apache storm, a hands on tutorial. In: 2015 IEEE International Conference on Cloud Engineering. pp. 2–2. IEEE (2015)

[61] Faber, W., Leone, N., Perri, S.: The intelligent grounder of DLV. In: Correct Reasoning. Lecture Notes in Computer Science, vol. 7265, pp. 247–264. Springer (2012)

[62] Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA. Lecture Notes in Computer Science, vol. 3229, pp. 200–212. Springer (2004)

[63] Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. **175**(1), 278–298 (2011)

[64] Ferreira, J., Lavado, D., Gonçalves, R., Knorr, M., Krippahl, L., Leite, J.: Faster than LASER - towards stream reasoning with deep neural networks. In: EPIA. Lecture Notes in Computer Science, vol. 12981, pp. 363–375. Springer (2021)

[65] Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Preliminary report. In: KR. AAAI Press (2012)

[66] Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Answer set programming for stream reasoning. CoRR **abs/1301.1392** (2013)

[67] Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. In: LPNMR. Lecture Notes in Computer Science, vol. 6645, pp. 54–66. Springer (2011)

[68] Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract gringo. Theory Pract. Log. Program. **15**(4-5), 449–463 (2015)

[69] Gebser, M., Kaminski, R., Kaufmann, B., Romero, J., Schaub, T.: Progress in clasp series 3. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 368–383. Springer (2015)

[70] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. CoRR **abs/1405.3694** (2014)

[71] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory Pract. Log. Program. **19**(1), 27–82 (2019)

[72] Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F., Schaub, T.: Evaluation techniques and systems for answer set programming: a survey. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. pp. 5450–5456. ijcai.org (2018). https://doi.org/10.24963/ijcai.2018/769, https://doi.org/10.24963/ijcai.2018/769

[73] Gebser, M., Maratea, M., Ricca, F.: What's hot in the answer set programming competition. In: AAAI. pp. 4327–4329. AAAI Press (2016)

[74] Gebser, M., Maratea, M., Ricca, F.: The sixth answer set programming competition. J. Artif. Intell. Res. **60**, 41–95 (2017)

[75] Gebser, M., Maratea, M., Ricca, F.: The seventh answer set programming competition: Design and results. Theory Pract. Log. Program. **20**(2), 176–204 (2020)

[76] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. pp. 1070–1080. MIT Press (1988)

[77] Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991)

[78] Germano, S., Pham, T., Mileo, A.: Web stream reasoning in practice: On the expressivity vs. scalability tradeoff. In: RR. Lecture Notes in Computer Science, vol. 9209, pp. 105–112. Springer (2015)

[79] Graupe, D.: Principles of Artificial Neural Networks - 3rd Edition, Advanced Series in Circuits and Systems, vol. 7. World Scientific (2013)

[80] Hall, R.E., Bowerman, B., Braverman, J., Taylor, J., Todosow, H., Von Wimmersperg, U.: The vision of a smart city. Tech. rep., Brookhaven National Lab.(BNL), Upton, NY (United States) (2000)

[81] Haque, A.K.M.B., Bhushan, B., Dhiman, G.: Conceptualizing smart city applications: Requirements, architecture, security issues, and emerging trends. Expert Syst. J. Knowl. Eng. **39**(5) (2022)

[82] Hippen, N., Lierler, Y.: Automatic program rewriting in non-ground answer set programs. In: PADL. Lecture Notes in Computer Science, vol. 11372, pp. 19–36. Springer (2019)

[83] Hoeksema, J., Kotoulas, S.: High-performance distributed stream reasoning using s4. In: Ordring Workshop at ISWC (2011)

[84] Hollands, R.G.: Will the real smart city please stand up? City **12**(3), 303–320 (2008)

[85] Hueske, F., Kalavri, V.: Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications. O'Reilly Media (2019)

[86] Ianni, G., Pacenza, F., Zangari, J.: Incremental maintenance of overgrounded logic programs with tailored simplifications. TPLP **20**(5), 719–734 (2020)

[87] Iqbal, M.H., Soomro, T.R., et al.: Big data analysis: Apache storm perspective. International journal of computer trends and technology **19**(1), 9–14 (2015)

[88] Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N.H., Braynard, R.: Networking named content. In: CoNEXT. pp. 1–12. ACM (2009)

[89] Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., Markl, V.: Benchmarking distributed stream data processing systems. In: ICDE. pp. 1507–1518. IEEE Computer Society (2018)

[90] Kim, H., Choi, H., Kang, H., An, J., Yeom, S., Hong, T.: A systematic review of the smart energy conservation system: From smart homes to sustainable smart cities. Renewable and Sustainable Energy Reviews **140**, 110755 (2021). https://doi.org/https://doi.org/10.1016/j.rser.2021.110755, `https://www.sciencedirect.com/science/article/pii/S1364032121000502`

[91] Le-Phuoc, D., Quoc, H.N.M., Parreira, J.X., Manfred, Hauswirth: The linked sensor middleware — connecting the real world and the semantic web (2011)

[92] Lefèvre, C., Béatrix, C., Stéphan, I., Garcia, L.: Asperix, a first-order forward chaining approach for answer set computing. Theory Pract. Log. Program. **17**(3), 266–310 (2017)

[93] Lefèvre, C., Nicolas, P.: The first version of a new ASP solver : Asperix. In: LPNMR. Lecture Notes in Computer Science, vol. 5753, pp. 522–527. Springer (2009)

[94] de Leng, D., Heintz, F.: Approximate stream reasoning with metric temporal logic under uncertainty. In: AAAI. pp. 2760–2767. AAAI Press (2019)

[95] Leone, N., Allocca, C., Alviano, M., Calimeri, F., Civili, C., Costabile, R., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Cuteri, B., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., Zangari, J.: Enhancing DLV for large-scale reasoning. In: LPNMR. Lecture Notes in Computer Science, vol. 11481, pp. 312–325. Springer (2019)

[96] Leone, N., Manna, M., Morelli, M.C., Perri, S.: A formal comparison between datalog-based languages for stream reasoning. In: Datalog. CEUR Workshop Proceedings, vol. 3203, pp. 151–165. CEUR-WS.org (2022)

[97] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7**(3), 499–562 (2006)

[98] Lin, T., Goyal, P., Girshick, R.B., He, K., Dollár, P.: Focal loss for dense object detection. In: ICCV. pp. 2999–3007. IEEE Computer Society (2017)

[99] Mileo, A., Abdelrahman, A., Policarpio, S., Hauswirth, M.: Streamrule: A nonmonotonic stream reasoning system for the semantic web. In: RR. Lecture Notes in Computer Science, vol. 7994, pp. 247–252. Springer (2013)

[100] Mileo, A., Dao-Tran, M., Eiter, T., Fink, M.: Stream reasoning (2017)

[101] Nickles, M., Mileo, A.: Web stream reasoning using probabilistic answer set programming. In: RR. Lecture Notes in Computer Science, vol. 8741, pp. 197–205. Springer (2014)

[102] Noghabi, S.A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., Campbell, R.H.: Samza: stateful scalable stream processing at linkedin. Proceedings of the VLDB Endowment **10**(12), 1634–1645 (2017)

[103] Özçep, Ö.L., Möller, R., Neuenstadt, C.: A stream-temporal query language for ontology based data access. In: Description Logics. CEUR Workshop Proceedings, vol. 1193, pp. 696–708. CEUR-WS.org (2014)

[104] Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: GASP: answer set programming with lazy grounding. Fundam. Informaticae **96**(3), 297–322 (2009)

[105] Pham, T., Ali, M.I., Mileo, A.: C-ASP: continuous asp-based reasoning over RDF streams. In: LPNMR. Lecture Notes in Computer Science, vol. 11481, pp. 45–50. Springer (2019)

[106] Pham, T., Ali, M.I., Mileo, A.: Enhancing the scalability of expressive stream reasoning via input-driven parallelization. Semantic Web **10**(3), 457–474 (2019)

[107] Pham, T., Mileo, A., Ali, M.I.: Towards scalable non-monotonic stream reasoning via input dependency analysis. In: ICDE. pp. 1553–1558. IEEE Computer Society (2017)

[108] Phuoc, D.L., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: ISWC (1). Lecture Notes in Computer Science, vol. 7031, pp. 370–388. Springer (2011)

[109] Phuoc, D.L., Eiter, T., Tuán, A.L.: A scalable reasoning and learning approach for neural-symbolic stream fusion. In: AAAI. pp. 4996–5005. AAAI Press (2021)

[110] RDF Core Working Group: Resource description framework (RDF) (2014), `https://www.w3.org/2001/sw/#rdf`

[111] Meta - Meta Investor Relations: Q3 2022 earnings (2022), `https://investor.fb.com/investor-events/event-details/2022/Q3-2022-Earnings/default.aspx`

[112] Steve Harris, Garlik, a part of Experian Andy Seaborne, The Apache Software Foundation: SPARQL 1.1 query language (2013), `https://www.w3.org/TR/sparql11-query/`

[113] The Apache Software Foundation: Apache flink — stateful computations over data streams, `https://flink.apache.org/`

[114] The Apache Software Foundation: Apache storm, https://storm.apache.org/index.html

[115] The Apache Software Foundation: Kafka streams, https://kafka.apache.org/documentation/streams/

[116] Ren, X., Curé, O., Naacke, H., Xiao, G.: Bigsr: real-time expressive RDF stream reasoning on modern big data platforms. In: IEEE BigData. pp. 811–820. IEEE (2018)

[117] Schneider, P., Alvarez-Coello, D., Le-Tuan, A., Duc, M.N., Phuoc, D.L.: Stream reasoning playground. In: ESWC. Lecture Notes in Computer Science, vol. 13261, pp. 406–424. Springer (2022)

[118] Schneider, P., Alvarez-Coello, D., Le-Tuan, A., Duc, M.N., Phuoc, D.L.: Stream reasoning playground (2022), https://github.com/patrik999/stream-reasoning-challenge

[119] Seki, H.: Unfold/fold transformations of stratified programs. Theor. Comput. Sci. **86**(1), 107–139 (1991)

[120] Taghezouit, B., Harrou, F., Larbes, C., Sun, Y., Semaoui, S., Arab, A.H., Bouchakour, S.: Intelligent monitoring of photovoltaic systems via simplicial empirical models and performance loss rate evaluation under labview: A case study. Energies **15**(21) (2022). https://doi.org/10.3390/en15217955, https://www.mdpi.com/1996-1073/15/21/7955

[121] Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume II. Computer Science Press (1989)

[122] Walega, P.A., Kaminski, M., Grau, B.C.: Reasoning over streaming data in metric temporal datalog. In: AAAI. pp. 3092–3099. AAAI Press (2019)

[123] Weinzierl, A.: Blending lazy-grounding and cdnl search for answer-set solving. In: Balduccini, M., Janhunen, T. (eds.) Logic Programming and Nonmonotonic Reasoning. pp. 191–204. Springer International Publishing, Cham (2017)

[124] Weinzierl, A.: Blending lazy-grounding and CDNL search for answer-set solving. In: LPNMR. Lecture Notes in Computer Science, vol. 10377, pp. 191–204. Springer (2017)

[125] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI. pp. 15–28. USENIX Association (2012)

[126] Zaniolo, C.: Logical foundations of continuous query languages for data streams. In: Datalog. Lecture Notes in Computer Science, vol. 7494, pp. 177–189. Springer (2012)

[127] Šurdonja, S., Giuffrè, T., Deluka-Tibljaš, A.: Smart mobility solutions – necessary precondition for a well-functioning smart city. Transportation Research Procedia **45**, 604–611 (2020). https://doi.org/https://doi.org/10.1016/j.trpro.2020.03.051, `https://www.sciencedirect.com/science/article/pii/S2352146520302131`, transport Infrastructure and systems in a changing world. Towards a more sustainable, reliable and smarter mobility.TIS Roma 2019 Conference Proceedings