Provided by the author(s) and University of Galway in accordance with publisher policies. Please cite the published version when available.

Title	Expressive RDF stream reasoning via data parallelism in answer set programming
Author(s)	Pham, Le Thi Anh Thu
Publication Date	2020-02-26
Publisher	NUI Galway
Item record	http://hdl.handle.net/10379/15806

Downloaded 2024-05-13T07:18:54Z

Some rights reserved. For more information, please see the item record link above.

National University of Ireland, Galway

Doctoral Thesis

Expressive RDF Stream Reasoning via Data Parallelism in Answer Set Programming

***Author*: Le Thi Anh Thu Pham**

Supervised by: Dr. Muhammad Intizar Ali and Dr. Alessandra Mileo
Co-supervised by: Dr. Matthias Nickles

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Insight Centre for Data Analytics
College of Engineering and Informatics

February 2020

Declaration of Authorship

I, Le Thi Anh Thu Pham, declare that this thesis titled, 'Expressive RDF Stream Reasoning via Data Parallelism in Answer Set Programming' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 25/02/2020

"I have not failed. I've just found 10,000 ways that won't work."

Thomas A. Edison

NATIONAL UNIVERSITY OF IRELAND, GALWAY

Abstract

The Insight Centre for Data Analytics
College of Engineering and Informatics

Doctor of Philosophy

Expressive RDF Stream Reasoning via Data Parallelism in Answer Set Programming

by Le Thi Anh Thu Pham

The Web nowadays is highly dynamic with massive amounts of data being continuously generated from a huge number of devices and services across the Internet. Various application scenarios in several domains, such as environment monitoring, health care systems, and smart transportation, can hugely benefit from the ability to efficiently integrate and query data streams from these sources to provide better services. However, in such applications, it is not only capturing data streams that is important, but also the ability to extract insights from such streams, and use them to target users' needs, preferences and constraints. For this reason, different types of complex reasoning tasks need to be efficiently designed and executed on such streams to capture the sophisticated requirements of users. Stream Reasoning is an emerging research area which focuses on providing continuous complex reasoning capabilities over data streams. However, Stream Reasoning faces many challenges not only due to their heterogeneity but also due to the exponential growth in the availability of streaming data on the Web, which severely limits the complexity of reasoning that can be used to extract actionable knowledge in a scalable and reliable way.

The key challenge addressed in this thesis is to enable expressive reasoning over massive, distributed, heterogeneous data streams in a scalable way. I address this problem by integrating Semantic Web for semantic integration, Answer Set Programming (ASP) for expressive reasoning, and Data Stream Management Systems for stream processing. The trade-off between scalability and expressivity in Stream Reasoning is considered, and parallel reasoning techniques are proposed to enhancing scalability while maintaining some of the key reasoning capabilities that are more expressive but also computationally more expensive.

The thesis addresses two research questions related to how the expressivity and scalability of a reasoner can be improved when reasoning on Semantic Web data streams. For

the first research question which targets expressivity, I propose C-ASP, a language extended from the ASP language with Resource Description Framework (RDF) streaming operators, which allows users to express complex requirements in terms of preferences and constraints, as a continuous reasoning request. The C-ASP reasoner is implemented to continuously evaluate such reasoning request when new data arrives. The experimental evaluation shows that the C-ASP engine outperforms the state-of-the-art RDF stream processing engine C-SPARQL. For the second research question which focuses on the scalability, I optimize the reasoning process of the C-ASP reasoner with a parallel approach based on data-level parallelism, and I demonstrate how the correctness of the results can be maintained. To do so, a clear characterization and formal definitions for analyzing the dependencies among input data streams are provided. The algorithms are developed to create a partitioning plan for guiding the parallel reasoning process to split data streams on-the-fly. Experiments show that applying this data-level parallelism improves the reasoning process significantly.

The research discussed in this thesis has been deployed in two real-world scenarios in the context of Smart Cities where event-driven contextual knowledge extraction is introduced, and Smart Enterprise where an Internet of Things-enabled meeting management system is developed. The former aims at continuously identifying and filtering critical events that might affect the decision making of users while the latter investigates how to enhance users' experience in online meetings on-the-go by using mobile sensors embedded in a communication platform. By addressing the requirements of such scenarios, the prototypes demonstrate the validity and feasibility of the approach proposed in this thesis.

Acknowledgements

I would like to express sincere gratitude to Dr. Alessandra Mileo, my thesis supervisor, for her patience and excellence in guiding me through my PhD journey. The time she dedicated and the advice she provided towards the completion of this work are priceless. Her expertise and positive attitude encouraged me to overcome all struggles occurred during my PhD process. I am thankful to my co-supervisor, Dr. Muhammad Intizar Ali, not only for funding the extension period of my PhD but also for his great support during the completion of this thesis. My thanks also go to Dr. Matthias Nickles for his time to review the thesis and slides.

I would like to thank my Graduate Research Committee members: Prof. Dietrich Rebholz-Schuhmann, Dr. John Breslin, Dr. Edward Curry, and Dr. Brian Davis. Their insightful feedback at various stages of this research help keeping this work on the track towards completion. I am also very grateful to my examiners Prof. Dr. Emanuele Della Valle and Prof. Dr. Mathieu D'Aquin for their time to evaluate this work and their precious feedback and comments, and thanks to Dr. Paul Buitelaar for chairing the process.

My acknowledgement goes to the funding agencies who supported this work at different stages: Science Foundation Ireland, Enterprise Ireland, and the European Commission. Many thanks to friends and colleagues who I have experienced great discussions and collaborations with. I deeply appreciate Hugo Hromic for his help in setting up environments for experiments in this thesis, and Andrea Barraza and Andrea Yanez for their proof-reading comments on the thesis and slides.

Finally, I would like to send my sincerest gratitude to my parents and my sisters who always love me unconditionally. To my sisters, thank you all for taking care of mom and dad so that I could pursue my career abroad. Thanks to my dear friend, Hau Bui, who takes care of me as her sister and cooks a lot of delicious Vietnamese food to me.

Contents

Declaration of Authorship **i**

Abstract **iii**

Acknowledgements **v**

Contents **vi**

List of Figures **ix**

List of Tables **xi**

Listings **xii**

Abbreviations **xiv**

1 Introduction **1**
- 1.1 Motivation & Problem Description . . . 2
- 1.2 Research Questions . . . 6
- 1.3 Hypotheses . . . 7
- 1.4 Overview of The Proposed Approach . . . 8
- 1.5 Contributions . . . 8
- 1.6 Thesis Outline . . . 9
- 1.7 Publications . . . 10

2 Background **13**
- 2.1 Semantic Web . . . 13
  - 2.1.1 Data Models . . . 13
  - 2.1.2 Semantic Web Streams . . . 15
- 2.2 Answer Set Programming . . . 17
  - 2.2.1 Syntax . . . 17
  - 2.2.2 Semantics . . . 19
  - 2.2.3 The ASP solving paradigm . . . 21
  - 2.2.4 Optimization in ASP . . . 22
- 2.3 Stream Reasoning . . . 24

2.3.1 Conceptual Architecture 25
2.3.2 Stream Reasoning Development 25
2.3.3 Stream Reasoning Categories 30
2.4 Summary 32

3 Related Work **33**
3.1 Stream Reasoning 33
3.1.1 Languages and Engines 33
3.1.1.1 SPARQL-based approaches 34
3.1.1.2 CEP-based approaches 36
3.1.1.3 Logic-based approaches 38
3.1.2 Comparative Analysis 41
3.1.2.1 RSP Benchmarks 41
3.1.2.2 Reasoning Benchmarks 44
3.2 Optimization via Parallel Strategy 46
3.3 Summary 50

4 C-ASP: Continuous Extension of ASP for RDF Stream Reasoning **51**
4.1 The C-ASP Processing Model 53
4.1.1 Windowing: from streams to relations 54
4.1.2 Evaluating: from relations to relations 56
4.1.3 Streaming: from relations to streams 57
4.2 Implementation: the C-ASP Language 57
4.2.1 C-ASP Reasoning Request 58
4.2.2 Examples of a C-ASP Reasoning Request 59
4.3 Evaluation 61
4.4 Summary 63

5 Characterizing Input-driven Dependency **65**
5.1 Reasoning over Independent Data Streams 66
5.1.1 Experiment Setting 67
5.1.2 Experiment Discussion 68
5.2 Reasoning over Dependent Data Streams 71
5.2.1 Running Example: Traffic Management 71
5.2.2 Assumptions 72
5.2.3 Input Dependency Analysis 73
5.2.4 Building Input Dependency Graph 77
5.3 Summary 80

6 Input-driven Parallel Reasoning **82**
6.1 Partitioning Plan 83
6.1.1 Unconnected Input Dependency Graph 83
6.1.2 Connected Input Dependency Graph 85
6.2 Parallel Reasoning in C-ASP 90
6.3 Evaluation 91
6.3.1 Experiment 1: Recursive positive rules 92
6.3.2 Experiment 2: Stratified negation rules 95
6.4 Summary 96

7 Use Cases and Prototypes **98**
7.1 Contextual Event Filtering System . . . 98
7.1.1 Contextual Filtering & Requirements . . . 100
7.1.2 Implementation of C-ASP Reasoner for Contextual Filtering . . . 101
7.1.3 Context-aware Travel Planner . . . 103
7.1.4 Context-aware Parking Planner . . . 103
7.2 IoT-enabled Meeting Management System . . . 104
7.2.1 Motivating Scenario . . . 105
7.2.2 IoT-MMS Architecture . . . 107
7.2.3 Stream Reasoning in IoT-MMS . . . 107
7.2.4 IoT-MMS Application Interface . . . 110
7.2.4.1 Android Application and User Login . . . 111
7.2.4.2 From Meeting Creation to Notification . . . 111
7.3 Summary . . . 115

8 Conclusion **116**
8.1 Contributions . . . 116
8.2 Limitations . . . 118
8.3 Future Work . . . 119

Bibliography **122**

List of Figures

2.1 The graphical representation of an RDF triple . . . 14
2.2 The example of the RDFS inference . . . 14
2.3 An RDF stream of booking hotels . . . 16
2.4 Declarative problem solving paradigm [1] . . . 17
2.5 A directed graph . . . 19
2.6 The ASP solving process . . . 21
2.7 A example of a dependency graph [2] . . . 24
2.8 The conceptual architecture of a SR system . . . 25
2.9 The conceptual architecture of DSMSs [3] . . . 27
2.10 The CQL model [4] . . . 27
2.11 The CEP conceptual architecture . . . 29
2.12 CEP operators [5] . . . 30
2.13 SR Layers [6] . . . 30

4.1 The C-ASP Processing Model . . . 53
4.2 An RDF stream of booking hotels . . . 56
4.3 C-ASP syntax . . . 58
4.4 Q1C & R1C (f=1) . . . 64
4.5 Q1C & R1C (f=2) . . . 64
4.6 Q2C & R2C (f=1) . . . 64
4.7 Q2C & R2C (f=2) . . . 64
4.8 Q10C & R10C (f=1) . . . 64
4.9 Q10C & R10C (f=2) . . . 64

5.1 Reasoning time . . . 70
5.2 Extended dependency graph G_P . . . 75
5.3 Input dependency graph $G_P^{inpre(P)}$ . . . 76
5.4 Types of dependencies . . . 77
5.5 The process to build an input dependency graph . . . 77

6.1 Input dependency graph $G_{P'}^{inpre(P')}$ . . . 86
6.2 Output of the decomposing process for $G_{P'}^{inpre(P')}$ . . . 89
6.3 The Extended StreamRule . . . 90
6.4 Latency (recursive rules with static setting) . . . 94
6.5 Memory consumption (recursive rules with static setting) . . . 94
6.6 Latency (recursive rules with streaming setting) . . . 95
6.7 Memory consumption (recursive rules with streaming setting) . . . 95
6.8 Latency (recursive and stratified negation rules) . . . 96

6.9 Memory consumption (recursive and stratified negation rules) 97

7.1 The components of CityPulse framework with their APIs [7] 99
7.2 Route selection 103
7.3 Route constraints 103
7.4 Optimal routes 104
7.5 Ciritcal Event 104
7.6 Parking selection 105
7.7 Parking constraint 105
7.8 Optimal parking spaces 105
7.9 IoT-enabled communication system architecture [8] 108
7.10 Stream Reasoning Layer Architecture 109
7.11 Application login interfaces 111
7.12 Available sensor list 112
7.13 Create new meeting event from OM 113
7.14 Adding agendas for meeting event 113
7.15 Setting sensor thresholds for the event 113
7.16 IoT panel in meeting room for each user 114
7.17 User's IoT Capabilities Notification 114

List of Tables

3.1 Comparison of SR systems. (Note: ST stands for Single timestamp and SB for SPARQL-based) 40

Listings

2.1 An example of a basic SPARQL query 15
2.2 Samples for the enrichment of the RDF data model 16
2.3 An ASP program for graph 3-coloring (with Clingo format input) 19
4.1 RR1 59
4.2 RR2 59
4.3 RR3 60
4.4 RR4 61
4.5 Snapshot of query Q1C 61
4.6 Snapshot of reasoning request R1C 63
5.1 An example of a weather event 68
5.2 An example of user's context 68
5.3 The reasoning request for notifying critical events 69
5.4 Sample rules for detecting events 71
6.1 The reasoning request with positive recursive rules inspired from LUBM . 93
6.2 Negation-as-failure rules 96
7.1 An example of an annotated event in Contextual Filtering 102
7.2 Rules for contextual filtering of events with ranking through linear combination 102
7.3 Rules of Event Detection 109
7.4 Rules of User Reasoner 110
7.5 Rules of Meeting Reasoner 111

Abbreviations

ADM Active Database Management.

APIs Application Programming Interfaces.

ASP Answer Set Programming.

BFS Breadth-First Search.

CEP Complex Event Processing.

CQL Continuous Query Language.

CWA Closed World Assumption.

DSMS Data Stream Management Systems.

EDB extensional database.

GL Gelfond-Lifschitz.

GPUs Graphical Processing Units.

IDB intensional database.

IoT Internet of Things.

IoT-MMS IoT-enabled Meeting Management System.

IRI Internationalized Resource Identifier.

KRR Knowledge Representation & Reasoning.

LSD Linked Sensor Data.

OM OpenMeetings.

OWL Web Ontology Language.

RDF Resource Description Framework.

RDFS RDF Schema.

RSP RDF Stream Processing.

SPARQL Simple Protocol and RDF Query Language.

SR Stream Reasoning.

SSN Semantic Sensor Network.

SW Semantic Web.

WWW World Wide Web.

Dedicated to my parents and sisters.

Chapter 1

Introduction

Massive amounts of data are being generated every minute every day from a huge number of devices and services across the Internet. Typical sources of such data include sensors, Internet of Things (IoT) [9] devices, and social networks. They create a paradigm shift where (near) real-time data is becoming ubiquitous in all aspects of the human life. Each of dynamic data flows delivered from these sources is referred to as a data stream.

Various application scenarios have been getting benefits from data streams [10] such as Smart Cities [11], Smart Grids [12], and Remote Health Monitoring [13]. However, the explosion of highly dynamic data on the Web together with the complexity of domain applications still far exceed the computing capability of current methods and infrastructures. One of major challenges emerged among these real scenarios is the ability to perform complex reasoning over huge volumes of heterogeneous data streams in (near) real-time [14]. Different research communities including Database, Semantic Web (SW) and Knowledge Representation & Reasoning (KRR) have been focusing on the above challenge and, as a result, the new area of so-called Stream Reasoning (SR) started to develop [15].

Several methods have been proposed for processing data streams. For example, in the SW community, existing solutions extend: (i) RDF [16] (a standard model for data interchange on the Web) to RDF streams [17] with timestamps for capturing temporal properties of data; and (ii) Simple Protocol and RDF Query Language (SPARQL) [18] (a query language for RDF data) to continuous SPARQL-based query languages (e.g., C-SPARQL language [19], CQELS-QL [20], and SPARQL$_{stream}$ [21]) for querying over streams of RDF data. In addition, various engines have been implemented and widely used in the stream processing frameworks such as C-SPARQL [22] and CQELS [20].

Majority of existing solutions are divided into two categories: Data Stream Management Systems (DSMS) [23] and Complex Event Processing (CEP) systems [24]. The former approach uses a so-called *window operators* [25] to transform data streams into timestamped relations and processes them with relational algebra-based techniques [26]. The latter approach considers observable raw data as primitive events and identifies when composite events occur with a given complex event pattern constructed from some specific operators [27]. Both approaches are able to deal with the transient nature of data streams mostly based on pattern matching techniques.

These systems, however, do not meet requirements of real-world applications [28]. The complexity of such applications requires capabilities for reasoning over incomplete, diverse, and unreliable input streams together with rich background knowledge in order to extract new knowledge which supports users in decision making. Answer to this need, the research of recent years in SR focuses on coupling reasoning techniques with reactive throughput-efficient stream processing systems [29].

1.1 Motivation & Problem Description

The concept of Stream Reasoning (SR) was introduced by Heiner Stuckenschmidt, Stefano Ceri, Emanuele Della Valle, and Frank van Harmelen in 2010 as follows:

> SR is logical reasoning in real time on gigantic and inevitably noisy data streams in order to support the decision process of extremely large numbers of concurrent users. [30]

A more detail definition from the Encyclopedia of Database Systems is:

> SR refers to inference approaches and deduction mechanisms which are concerned with providing continuous inference capabilities over dynamic data. The paradigm shift from current batch-like approaches towards timely and scalable stream reasoning leverages the natural temporal order in data streams and applies windows-based processing to complex deduction tasks that go beyond continuous query processing such as those involving preferential reasoning, constraint optimization, planning, uncertainty, non-monotonicity, non-determinism, and solution enumeration. [15]

Logical reasoning has been playing an important role in designing and building expert systems. Its techniques are powerful at representing complex domain knowledge and

solving NP-hard [31] problems. With these advances, using logical reasoning is a high potential direction to overcome limitations of stream processing solutions in dealing with complex decision making tasks [32]. However, the disadvantage of expressive reasoning is scalability in which the computational time may rise exponentially along with an increase of the domain complexity and the input size. This drawback gets worse in a dynamic environment where input data keeps coming and stressing reasoners. Therefore, as stated in the above definitions, SR aims at providing solutions to perform complex reasoning over highly changing data streams in (near) real time. This research trend is considered as "an unexplored yet high impact research area and new multidisciplinary approach that can provide the abstractions, foundations, methods, and tools required to integrate data streams, the SW, and reasoning systems" [14].

The capability of SR promisingly benefits numerous areas of the human life [10, 14, 33] such as managing traffic flows, monitoring public-health risks, discovering new drugs, etc. These real-world applications require systems being able to handle challenges of the trade off between complexity and scalability coming from data streams, complex domain knowledge, and user requirements and preferences. A recent survey in SR [28] listed 9 capabilities that a such system must have:

- (R1) handle volume - the amount of data,
- (R2) handle velocity - the speed of data processing,
- (R3) handle variety - the number of data types,
- (R4) cope with incompleteness - the behavior dealing with missing information,
- (R5) cope with noise - the behavior dealing with fault or misleading data,
- (R6) provide answers in a timely fashion - the ability to reasoning in (near) real time,
- (R7) support fined-grained information access - being able to access on smaller piece of information,
- (R8) integrate complex domain models - the ability to combine rich domain knowledge,
- (R9) capture what user wants - meeting requirements and preferences of the user.

In [10, 14], multiple open issues are presented when building a SR system that meets those above requirements. Such issues can be summarized in 3 big challenges which can capture all above requirements:

- Integration: among data streams coming from multiple sources with different data types or between data streams and knowledge bases (can be static or quasi-static). For example, the efficient traffic monitoring use case requires that streaming data collected from traffic sensors on roads, weather sensors, and social networks (e.g., traffic tweets on Tweeter) is combined with a background knowledge about a city such as street maps or city cultural event information. Solutions for this challenge can capture requirements R3, R7, and R8.

- Scalability: is commonly evaluated on two dimensions: the ability to perform higher complex reasoning tasks (the computational complexity) and the ability to process larger input (the input size). In the context of streaming environments, a SR system is required to provide answers for a user reasoning request in timely fashion. Solutions for this challenge can capture requirements R1, R2, R6, and R9.

- Expressivity: shows to what extent complex reasoning can be supported by a stream reasoner. Extracting new knowledge from different abstract levels of data streams and information domains requires different expressive levels. The transient nature of input information demands the system being able to manage uncertainty or inconsistency. In addition, users requirements and preferences may ask for very hard reasoning abilities such as finding all of optimal answers (i.e., non determinism) or making a plan under their constraints and preferences (i.e., combinatorial optimization). Solutions for this challenge can capture requirements R4, R5, R7, and R9.

It has been a decade since the notation of SR was first introduced. Numerous techniques have been proposed in this area. Different communities have focused on complementary aspects of processing data streams. From the SW realm, besides the development of data models, query models, and languages inspired by RDF and SPARQL [34–37], RDF stream processors have been extended with entailment regimes. For example, C-SPAQRL [1] and Sparkwave [38] support RDF Schema (RDFS)[2] inference, TrOWL [39] and Streaming Knowledge Bases [40] support simple Web Ontology Language (OWL)[3] reasoning. Several attempts in extending OWL 2 DL[4] and its fragments, and optimizing its reasoning process by applying incremental reasoning or parallelization techniques [41–43]. Most of proposed solutions in SW focus closer to the data side so that they are normally referred to as stream processing approaches.

[1] https://github.com/streamreasoning/CSPARQL-ReadyToGoPack
[2] https://www.w3.org/TR/rdf-schema/
[3] https://www.w3.org/OWL/
[4] https://www.w3.org/TR/owl2-overview/

In the KRR community, researchers study about reasoning over changing worlds long time ago under the label of temporal logic [44] and belief revision [45]. These approaches, however, mainly focus on low-volume and low-frequency data. Recently, studies in this field have been investigating on extending expressive reasoning formalisms to cope with streaming data such as in ASP [46–48] or in Metric Temporal Logic [49, 50]. Some studies focus on building theoretical foundations for SR. For example, authors in [51] proposed the Logic-based framework for Analyzing Reasoning over Streams (LARS) which defines a rule-based modeling language to formalize SR semantics. Proposed solutions in this community focus closer on knowledge and inferences via enabling logical reasoning in the streaming settings, so that they are normally referred to as stream reasoning approaches.

So far, SR has seen several positive outcomes addressed all mentioned challenges but they have not resolved issues completely [28]. On one hand, stream processing solutions can handle streams of data and timely produce new results, but they are limited in complex reasoning capabilities that are required to solve user's sophisticated demands such as the ability to handle defaults, common sense, preferences, recursion, and non-determinism. On the other hand, logical reasoning engines can perform such complex reasoning tasks are mostly designed to work on static or quasi-static data. Extension from these engines to work on dynamic data is not trivial because expressivity of a reasoner is known to be inversely related to its performance - the more expressive the reasoner is, the longer it takes to perform reasoning. Most of studies have focused on monotonic reasoning or restricted to some fragments of non-monotonic reasoning [52] in order to speed up processing time of the reasoner. Moreover, in the attempt to bridge the gap between RDF Stream Processing (RSP) and expressive reasoning, other research groups follow the approach to combine a RDF stream processor with a reasoner in a pipeline [40, 53, 54]. This combination is based on the principle of having a 2-tier approach where: i) a stream processor is used to filter semantic data elements, and ii) a logical reasoner is used for computationally intensive tasks. It takes advances from both stream processing solutions and expressive reasoning solutions in order to achieve a better trade-off between scalability and expressivity of a SR system. However, most of exciting engines implemented from this approach hard-code a logical reasoner as subprocess that performs repetitive reasoning to infer new knowledge from data streams and a given rule set. Therefore, they do not provide a flexible way to seamlessly integrate the stream processing and reasoning functionalities.

This thesis aims at addressing the challenge of achieving the trade-off between scalability and expressivity for a reasoner over non-noisy SW data streams. On one hand, to push the expressivity of the reasoner higher, I leverage advanced techniques from non-monotonic reasoning, namely ASP [55], in order to resolve complex reasoning tasks.

ASP with its stable model semantics is well-known as a powerful high expressive declarative programming language to represent rich knowledge structures and the ability of managing defaults, common sense, preferences, recursion, and non-determinism. These are complex reasoning tasks with the expensive computational cost. On the other hand, I study optimization techniques to scale up the reasoner in the dynamic environment. Scalability is referred to as the ability to provide answers in an acceptable time when the throughput increases and the reasoning gets computationally intensive. In summary, my thesis focuses on tackling a research question *"How to perform non-monotonic reasoning based on ASP over RDF data streams in a scalable way?"*.

1.2 Research Questions

To address the main research problem mentioned above, the following questions need to be answered:

RQ1: How to enable complex reasoning based on ASP over RDF data streams?

At the beginning of this research, I explore the combination of advances in RDF stream processing techniques and ASP to address the expressivity concern. Instead of hard-coding an ASP solver as a reasoner over RDF data streams as in [40, 53, 54], I investigate an integration between SPARQL-inspired query languages which define a setting for continuously querying RDF streams and ASP-Core-2 [56] has been standardized as an input language format which can perform one-shot reasoning over data in form of predicates. This integration allows users to describe their sophisticated requirements and preferences via a continuous reasoning request which can be processed seamlessly by an ASP-based reasoner on RDF streaming data.

RQ2: How to scale up the reasoning process over RDF data streams under stable model semantics of ASP?

While *RQ1* investigates on the expressivity of an RDF stream reasoning engine, the second research question concerns about its scalability. The strong declarative aspect of ASP can modeling and solving complex problems on domain-specific knowledge including incomplete information, defaults, and preferences, but it is very costly to compute answer sets (i.e., solutions of a reasoning task). In addition, in the streaming setting, the reasoner needs to return results faster than when new input arrives in order to maintain the stability. I break this question into two sub-questions to consider:

- **RQ2.1: Which information is relevant to the reasoning process and how it can be used to optimize the reasoning performance?**

  To scale up the RDF stream reasoner, the first sub research question aims at discovering information pieces that are relevant to the reasoning process. Two typical information sources for this investigation are user-defined continuous reasoning requests and input data streams. In this way, the characteristics of semantic structures from both input-driven and domain-driven dimensions are studied.

- **RQ2: How to use such information found in RQ2.1 efficiently to speed up the reasoning process without losing the correctness of reasoning results?**

  In this second sub research question, a method to enhance the stream reasoning process using the semantic structures defined in *RQ2.1* is studied. Moreover, in the optimization step, stable model semantics of ASP need to be taken into account for guaranteeing the accuracy of reasoning results.

1.3 Hypotheses

The research in this thesis stands on these following hypotheses:

H1. The continuous extension of the ASP language with the RDF streaming features provides a higher expressivity to capture users' sophisticated requirements and preferences.

H2. Partitioning an input window in which data is independent helps to reduce the reasoning cost of an expressive stream reasoner. Moreover, the total time when reasoning sequentially over such partitions of the input window can be smaller than the reasoning time over the whole window under the circumstance of monotonically increasing reasoning time.

H3. The semantic dependencies between input data streams play an important role in the reasoning process in terms of its performance and the correctness of results. These dependencies can be captured based on the structure of a given reasoning request and can be decomposed in such a way to enable parallel reasoning and maintain the correctness of results.

H4. Parallel reasoning over partitioned data streams when taking into account input dependencies can reduce the cost of the reasoning process and maintain the correctness of combined reasoning results.

1.4 Overview of The Proposed Approach

The goal of this thesis is to provide an efficient and effective solution to perform complex reasoning based on ASP over RDF data streams in a scalable way, i.e., providing answers for the research questions mentioned in Section 1.2. The proposed approach constitutes these following main elements:

- *A continuous ASP-based reasoning language for RDF streams.* This piece of work answers the research question *RQ1* and validates the hypothesis *H1*. A language is extended from the ASP language with features of the continuous SPARQL query language to reasoning continuously over RDF streams. This language should provide a rich declarative way to express users' complex requirements and preferences in the form of so-called a *continuous reasoning request*. Such request should be registered into an expressive reasoner and be processed continuously whenever new data arrives.

- *Input-driven dependency characterization.* This element answers the research question *RQ2.1* and validates the hypotheses *H2* and *H3*. First, the key features which potentially affect the scalability of an expressive reasoner are identified. The correlation between such features and their impact on the reasoner's performance are empirically evaluated under the assumption of independent input data (*H2*). Later, dependencies among input data are taken into account for a further investigation on their significant impact on the reasoning performance in a streaming scenario. These dependencies can be discovered based on the structure of rules defined in the continuous reasoning request. This rule set is restricted to be under the stratified negation fragment of normal ASP, which ensures uniqueness of the solution. This work extends from study about dependency graph in ASP [2, 57] to capture relationships among input data streams (*H3*).

- *Input-driven parallel reasoning.* This functionality answers the research question *RQ2.2* and validates the hypotheses *H3* and *H4* by following the data partitioning approach. The analysis of input dependencies in the previous piece of work is used to construct a plan for partitioning input data (*H3*). This plan will guide the reasoner to split input data into chunks on-the-fly and process them in parallel while maintaining the correctness of the combined results (*H4*).

1.5 Contributions

The contributions of the research in this thesis include:

1. A continuous reasoning language, namely *the C-ASP language*, which is extended from ASP with RDF streaming features, to enable ASP-based reasoning over RDF data streams. This language is able to capture users' sophisticated requirements and preferences.

2. An ASP-based stream reasoner, namely *the C-ASP reasoner*, which processes the C-ASP language and performs complex continuous reasoning over RDF streams. This reasoner outperforms the state-of-the-art stream processor C-SPARQL.

3. A formal method for analyzing dependencies among input data based on the structure of a C-ASP reasoning request. This method characterizes different relationships between two predicates appearing in the input data in form of so-called *input dependency graph*.

4. Algorithms for building the input dependency graph from the C-ASP request, constructing a partitioning plan from this graph, and guiding the parallel reasoning process in C-ASP to split input data on-the-fly and combining reasoning results.

5. Formal proof that guarantees the correctness of the approach under the stable model semantics of ASP.

6. Extension of the C-ASP reasoner with input-driven parallelization for validation and testing proposed algorithms.

7. The implementation of two scenarios as demonstrators to validate the proposed approach in the area of Smart Cities and Smart Enterprise applications.

1.6 Thesis Outline

The rest of the thesis is organized as follows:

- *Chapter 2* presents the preliminaries for this research, which involves SW, ASP, and SR. It defines the terminologies and notations used in the theoretical design phase of the approach.

- *Chapter 3* summarizes the related works regarding to research questions mentioned in Section 1.2. Firstly, the classification of existing solutions in SR is described, and followed by a discussion about achievements and limitations of current methods in each category. Secondly, the works of parallel reasoning over data streams are reviewed.

- *Chapter 4* focuses the research question *RQ1* associated with hypothesis *H1*. An extension of the ASP language with `PREFIX, FROM, STREAM`, and `WINDOW` clauses from the continuous SPARQL language, namely the C-ASP language, is presented. In addition, an implementation of a so-called C-ASP reasoner which processes the proposed continuous reasoning language is described. Finally, the chapter reports experimental results showing C-ASP performances.

- *Chapter 5* addresses the research question *RQ2.1* associated with hypotheses *H2* and *H3*. In the first part, it details the effort in enhancing the scalability of the C-ASP reasoner over RDF streams with the case of independent input data streams. In the second part, this chapter presents a formal method to analyze dependencies among input data within a window and provides algorithms to build this dependency graph.

- *Chapter 6* focuses the research question *RQ2.2* associated with hypotheses *H3* and *H4*. It describes how dependencies discovered in the previous chapter can be used to improve the performance of the C-ASP reasoner. Algorithms for constructing a partitioning plan and on-line splitting input data to enable parallel reasoning is elaborated. Proofs for the correctness of reasoning results are also provided in this chapter. At the end, it details an extension of C-ASP with this parallel approach and the evaluation of efficiency.

- *Chapter 7* validates the practicality of the C-ASP reasoner by deploying it in Smart City and Smart Enterprise applications, namely the contextual event filtering system and the IoT-enabled meeting management system. In particular, the chapter reports the descriptions of such applications, their functionalities, as well as the contribution of the C-ASP reasoner in those scenarios.

- *Chapter 8* concludes the achievements and limitations of this research and discusses the potential future research.

1.7 Publications

Contributions of this work have been published in various relevant international conferences and journals as follows:

- Pham, Thu-Le, Muhammad Intizar Ali, and Alessandra Mileo. "Enhancing the Scalability of Expressive Stream Reasoning via input-driven Parallelization." In Semantic Web Preprint: 1-17, 2018.

- Pham, Thu-Le, Alessandra Mileo, and Muhammad Intizar Ali. "Towards Scalable Non-Monotonic Stream Reasoning via Input Dependency Analysis." In Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, pp. 1553-1558. IEEE, 2017.
- Pham, Thu-Le, Stefano Germano, Alessandra Mileo, Daniel Küemper, and Muhammad Intizar Ali. "Automatic configuration of smart city applications for user-centric decision support." In Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on, pp. 360-365. IEEE, 2017.
- Ali, Muhammad Intizar, Naomi Ono, Mahedi Kaysar, Zia Ush Shamszaman, Thu-Le Pham, Feng Gao, Keith Griffin, and Alessandra Mileo. "Real-time data analytics and event detection for IoT-enabled communication systems." Web Semantics: Science, Services and Agents on the World Wide Web 42 (2017): 19-37.
- Puiu, Dan, Payam Barnaghi, Ralf Toenjes, Daniel Kümper, Muhammad Intizar Ali, Alessandra Mileo, Josiane Xavier Parreira, Marten Fischer, Sefki Kolozali, Nazli Farajidavar, Feng Gao, Thorben Iggena, Thu-Le Pham, Cosmin-Septimiu Nechifor, Daniel Puschmann, Joao Fernandes. "Citypulse: Large scale data analytics framework for smart cities." IEEE Access 4 (2016): 1086-1108.
- Germano, Stefano, Thu-Le Pham, and Alessandra Mileo. "Web stream reasoning in practice: on the expressivity vs. scalability tradeoff." In International Conference on Web Reasoning and Rule Systems, pp. 105-112. Springer, Cham, 2015.
- Pham, Thu-Le. "A Scalable Adaptive Method for Complex Reasoning Over Semantic Data Streams." In European Semantic Web Conference, pp. 751-759. Springer, Cham, 2015.For

In addition, some parts of this work are described in project deliverables as follows:

Project: Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications

- Mirko Presser, João Fernandes, Daniel Kuemper, Thorben Iggena, Marten Fischer, Payam Barnaghi, Nazli Farajidvar, Sefki Kolozali, Thu-Le Pham, Muhammad Intizar Ali, and Dan Puiu. "Report on Integration and Evaluation Results". October 2016. URL: http://www.ict-citypulse.eu/page/sites/default/files/citypulse_d6.3_report_on_integration_and_evaluation_results_final.pdf
- João Fernandes, Dan Puiu, Thorben Iggena, Daniel Kuemper, Marten Fischer, Sefki Kolozali, Nazli Farajidavar, Daniel Puschmann, Feng Gao, Thu-Le Pham,

Azadeh Bararsani, and Aneta Vulgarakis. “Smart City Demonstrator”. July 2016. URL: http://www.ict-citypulse.eu/page/sites/default/files/citypulse_d6.2_smart_city_demonstrator-final.pdf

- Dan Puiu, Daniel Kümper, Marten Fischer, Sefki Kolozali, Nazli Farajidavar, Feng Gao, Thorben Iggena, Thu-Le Pham, Daniel Puschmann, Joao Fernandes, Bogdan Serbanescu, and Cosmin Marin. “Smart City Environment User Interfaces”. February 2016. URL: http://www.ict-citypulse.eu/page/sites/default/files/citypulse_d5.3_smart_city_environment_user_interfaces_final.pdf

- Alessandra Mileo, Stefano Germano, Thu-Le Pham, Dan Puiu, Daniel Kuemper, and Muhammad Intizar Ali. “User-Centric Decision Support in Dynamic Environments”. August 2015. URL: http://www.ict-citypulse.eu/page/sites/default/files/citypulse_d5.2-user-centric_decision_support_in_dynamic_environments_final.pdf

- Alessandra Mileo, Feng Gao, Muhammad Intizar Ali, Thu-Le Pham, Maria Bermudez, and Daniel Puschmann. “Real-time Adaptive Urban Reasoning”. July 2014. URL: http://www.ict-citypulse.eu/page/sites/default/files/citypulse_d5.1_real-time_adaptive_urban_reasoning_final.pdf

Project: Enabling the Internet of Everything: a Linked Data infrastructure for networking, managing and analyzing streaming information

- Thu-Le Pham, Mahedi Kaysar, Muhammad Intizar Ali, and Alessandra Mileo. “Stream Reasoning and Knowledge Extraction”. Jan 2016.

Chapter 2

Background

This chapter provides the background information used in the research of this thesis. The chapter is started with basic concepts in SW and SW in the streaming setting. Subsequently, the underlying concepts and relevant methodologies in ASP are presented. Finally, the chapter details the development of the SR area.

2.1 Semantic Web

SW is about connecting facts rather than linking to a specific document or an application as in the traditional World Wide Web (WWW). It uses machine-accessible knowledge to abstract from web documents and applications. This thesis uses various SW techniques. In this section, the data models as well as the SW streams are presented.

2.1.1 Data Models

RDF[1] is a framework proposed by the WWW Consortium (W3C)[2] for standardizing data interchange on the Web. RDF models the SW data in the form of *triples*, each consisting of a subject, a predicate, and a object. An RDF triple encodes a statement which is a claim about the world by describing a relation (represented by the predicate) between two entities (represented by the subject and the object). Figure 2.1 visualizes a triple as a simple graph with two nodes represented for the subject and the object in ovals and a directed link represented for the predicate.

[1] https://www.w3.org/TR/rdf11-concepts/

[2] https://www.w3.org/Consortium/

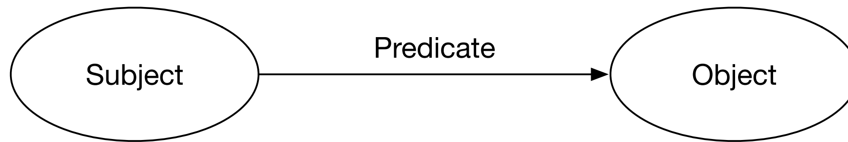


FIGURE 2.1: The graphical representation of an RDF triple

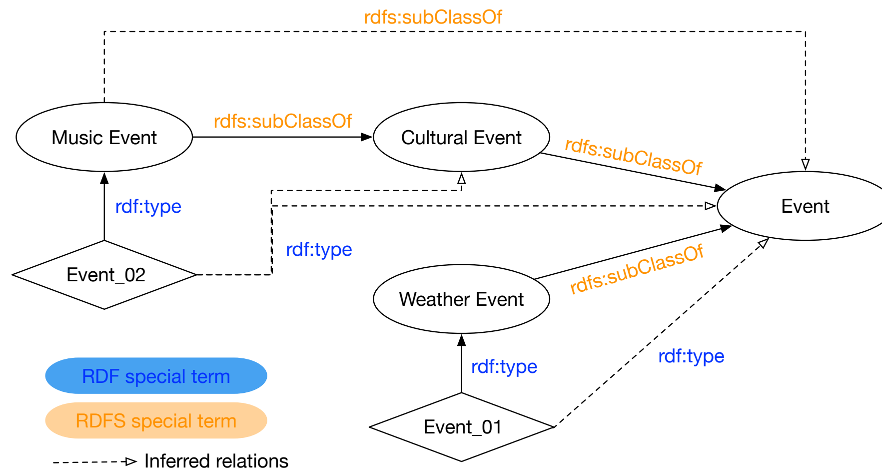


FIGURE 2.2: The example of the RDFS inference

RDFS[3] provides the data-modeling vocabulary for RDF data. It extends the basic RDF vocabulary with terms which allow describing classes (i.e., groups of related resources), property domains and ranges, taxonomies (i.e., classes and properties hierarchies), RDF containers, RDF collections, and so on. RDFS provides basic elements for the description of ontologies and enable inference, namely the RDFS entailment, to infer implicit RDF statements from explicitly specified ones. Figure 2.2 illustrates an example of the RDFS inference based on class hierarchies.

OWL[4] aims at leveraging advances of expressive and reasoning power of description logic to the SW. It extends RDF and RDFS by adding more vocabulary for representing richer types of properties and classes (e.g., symmetric properties, enumerated classes, etc) and more expressive relations among them (e.g., disjointedness, exactly one, equality, etc). However, OWL semantics is not fully compatible with RDF/RDFS semantics. To partially overcome this problem, OWL provides three increasingly expressive sub-languages: OWL Lite, OWL DL, and OWL Full.

SPARQL[5] is a SQL-like query language for retrieving and manipulating data stored in the RDF format. Moreover, with the availability of SPARQL specification for web service, it currently serves as an RDF data access protocol. The results of a SPARQL query

[3] https://www.w3.org/TR/rdf-schema/
[4] https://www.w3.org/TR/owl-features/
[5] https://www.w3.org/TR/rdf-sparql-query/

```
Select ?eventId Where {
        ?eventId rdf:type ?eventType.
        ?eventType rdfs:subClassOf :Event.
}
```

Listing 2.1: An example of a basic SPARQL query

over an RDF dataset varies depending on reasoning types supported by the SPARQL engine implementation. Listing 2.1 expresses a basic SPARQL query over the RDF graph shown in Figure 2.2. This query asks for Ids of any events available in the graph. {`Event_01`} is answer of the query with no reasoning support while {`Event_01`, `Event_02`} is result with the RDFS entailment support.

2.1.2 Semantic Web Streams

Moving from static data to streaming data, the very first notable update is in the RDF data model with temporal properties. The RDF data stream model is extended from RDF data with two main dimensions: data item (i.e., the minimal information unit in the stream) and time annotation (i.e., a set of time instants associated with each data item). The first extension, which is also the most popular, is the one in which an RDF statement is enriched with one timestamp t, e.g. [20, 21, 34, 58]. It reports that the RDF statement is valid at t. The second extension is adopted with two timestamps t_s and t_e, e.g. [36, 59, 60]. The semantics that is usually associated to these timestamps is a time interval in which the RDF statement is valid. Listing 2.2 (a) and (b) (in Turtle[6]-like syntax) shows examples for such time-annotated RDF statement. The first three elements in each row define an RDF statement while the fourth element in the square brackets identifies timestamps. Listing 2.2 (a) reports that an `event1` is detected at `dangan` street at timestamp $t = 1$ while Listing 2.2 (b) describes an `event2` is detected at `newcastle` street during the period of time from 1 to 6. The third extension proposes to use an RDF graph as an informative unit and associates it with one timestamp, e.g. [61]. An example for this extension is illustrated in Listing 2.2 (c) (in Trig[7]-like syntax) which describes an RDF graph `g1` including two RDF statements and being valid at timestamp $t = 2$.

In the following, the basic concepts in RSP are formalized and adopted from [4, 34, 59, 62].

[6] https://www.w3.org/TR/turtle/
[7] https://www.w3.org/TR/trig/

```
(a) :event1 :detectedAt :dangan [1]
(b) :event2 :detectedAt :newcastle [1, 6]
(c) :g1 {
        :event3 :detectedAt :thomas
        :event3 :detectedSource :sensors
} [2]
```

LISTING 2.2: Samples for the enrichment of the RDF data model

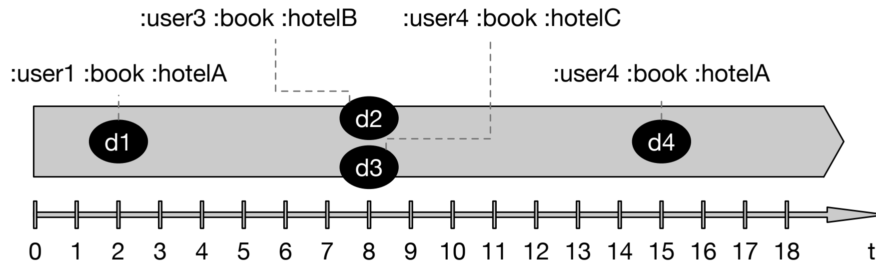


FIGURE 2.3: An RDF stream of booking hotels

Definition 2.1. A time line T is an infinite, discrete, ordered sequence of time instants $(t_1, t_2, ...)$, where $t_i \in N$. A time unit is the difference between two consecutive time instants $(t_{i+1} - t_i)$ and it is constant.

Definition 2.2. A timestamped RDF statement is a pair (d, t), where d is an RDF statement and $t \in T$ is a time instant.

Definition 2.3. An RDF data stream S is a (possibly infinite) bag (multi-set) of timestamped RDF statements in non-decreasing time order:

$$S = (d_1, t_1), (d_2, t_2), (d_3, t_3), ...$$

Example 2.1. *A travel company records the bookings from users to hotels that are available on its website in Figure 2.3. An RDF stream S of the bookings is as follows:*

$$(: user1 : book, : hotelA, 2)$$
$$(: user3 : book, : hotelB, 8)$$
$$(: user4 : book, : hotelC, 8)$$
$$(: user4 : book, : hotelA, 15)$$
$$...$$

The statements assert that ***user1*** *and* ***user4*** *booked* ***hotelA*** *at timestamps 2 and 15,* ***user3*** *booked* ***hotelB*** *and* ***user4*** *booked* ***hotelC*** *at the same time instant $t = 8$.*

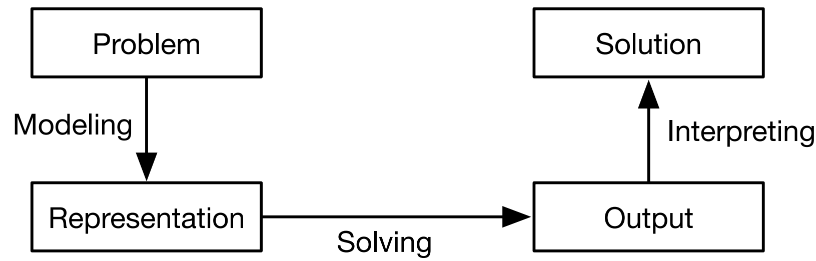


FIGURE 2.4: Declarative problem solving paradigm [1]

2.2 Answer Set Programming

ASP emerged in the 1990s from Logic Programming and Non-monotonic Reasoning as a declarative problem solving paradigm [1, 55, 63–67]. The Figure 2.4 illustrates this declarative approach. Compare to the traditional way of solving a problem by telling a computer *how to solve a problem*, the declarative paradigm tackles a problem with a different perspective by focusing on *how to state a problem* and leaving a solving process to the computer. The declarative approach models the given problem by creating its formal representation. The solving process automatically extracts an implicit state space from this representation and outputs results after exploring the state spaces with its sophisticated search algorithms. Those results are interpreted as solutions of the original problem.

The fundamental idea of ASP is to express a given problem into so-called *logic program*. The models of this logic program provides the solutions to the original problem, namely *answer set* or *stable models*, under the *stable models semantics* which introduced by Michael Gelfond and Vladimir Lifschitz in 1988 [63]. In what follows, I provide the syntax and semantics of ASP in Section 2.2.1 and Section 2.2.2, respectively. Section 2.2.3 is devoted to the ASP solving paradigm and Section 2.2.4 presents the optimization in ASP.

2.2.1 Syntax

As common in ASP, I denote uppercase letters or strings starting with an uppercase letter for *variables* while lowercase letters or strings starting with lowercase letter for *constants*, *predicates*, or *atoms*. In addition, function symbols are not considered.

Definition 2.4. A *term* is either a variable or a constant.

Definition 2.5. An *atom* is an expression $p(t_1, ..., t_n)$, where p is a *predicate* of arity n and t_1, ..., t_n are terms.

Definition 2.6. A *literal* is either a *positive literal* p or a *negative literal not* p, where p is an atom.

Definition 2.7. A *disjunctive rule* r is of the form:

$$a_1 \vee ... \vee a_n :- \; b_1, ..., b_k, \; not \; b_{k+1}, ..., \; not \; b_m$$

where $a_1, ..., a_n, b_1, ..., b_m$ are atoms and $n \geq 0, m \geq k \geq 0$.

"not" is called *negation as failure, default negation*, or *weak negation.* A rule r of the form above is normally composed by the head (i.e, the disjunction $a_1 \vee ... \vee a_n$) and the body (i.e, the conjunction $b_1, ..., b_k$, not $b_{k+1}, ...,$ not b_m). Bellow are some notations to distinguish parts of a rule r:

- the *head* of r: $H(r) = \{a_1, ..., a_n\}$
- the *positive body* of r: $B^+(r) = \{b_1, ..., b_k\}$
- the *negative body* of r: $B^-(r) = \{b_{k+1}, ..., b_m\}$
- the *body* of r: $B(r) = B^+(r) \cup B^-(r)$

A rule r is called:

- a *fact* if $B(r) = \emptyset$
- a *constraint* if $H(r) = \emptyset$
- a *normal rule* if $|H(r)| = 1$
- a *positive rule* if $B^-(r) = \emptyset$
- a *Horn rule* if it is positive and normal
- a *safe rule* if each variable in r also appears in at least one positive literal in the body of r

Recursive rules are rules where some body predicate depends, directly or transitively, on a predicate in the head. Intuitively, recursion through negation (or unstratified negation) happens when two or more predicates are mutually defined over *not* such as $\{b \leftarrow not \; a, \; a \leftarrow not \; b\}$.

Definition 2.8. An *ASP program* (or program) is a finite set of safe rules.

Example 2.2. *Consider the graph N-coloring problem with three colors red, green and blue* ($N = 3$) *and the directed graph as in Figure 2.5. An ASP program expressing this 3-coloring problem is illustrated in Listing 2.3. Line 1, 2, and 3 are facts which represent the directed graph and available colors via predicates* **node**, **edge**, *and* **col**. *Line 4 is a disjunctive rule (notation | in Clingo source code is similar to notation* $\cup$ *in the logic program). Line 5 is a constraint to eliminate unwanted candidate solutions.*

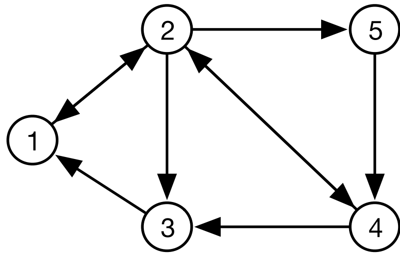


FIGURE 2.5: A directed graph

```
1 node(1).node(2).node(3).node
    (4).node(5).
2 edge(1,2).edge(2,1).edge(2,3).
    edge(2,4).edge(2,5).edge
    (3,1).edge(4,3).edge(4,2).
    edge(5,4).
3 col(red).col(blue).col(green).
4 color(X,red) | color(X,blue) |
     color(X,green) :- node(X).
5 :- edge(X,Y),col(C),color(X,C)
    ,color(Y,C).
```

LISTING 2.3: An ASP program for graph 3-coloring (with Clingo format input)

2.2.2 Semantics

In this section, I describe the semantics of an ASP program which is based on the answer set semantics originally proposed by Gelfond and Lifschitz in 1991 [68].

Definition 2.9. A term, an atom, a literal, a rule, or a program is *ground* if no variable occurs in it.

Definition 2.10. Given a program P, the *Herbrand Universe* U_P is a set of all constants occurring in P and the *Herbrand Base* B_P is a set of all possible ground atoms constructible from predicates appearing in P with constants in U_P.

Example 2.3. *Consider the program* P *as in Example 2.2. The Herbrand Universe and a snapshot of the Herbrand Base as follow:*
U_P = {*1, 2, 3, 4, 5, red, blue, green*}.
B_P = {*node(1), node(red),..., edge(1,1), edge(1,blue),..., col(1), col(green),..., color(3,4), color(5,2), color(2,green)...* }.

For any program P:

- $grnd(r)$ denotes the set of ground rules obtained by substituting variables in a rule $r \in P$ with constants in U_P in all possible ways and the ground program P.
- $grnd(P)$ denotes the ground program of P which is a collection of $grnd(r)$ for all $r \in P$ $(grnd(P) = \bigcup_{r \in P} grnd(r))$.
- An *interpretation* of P is a subset of Herbrand Base B_P.

Definition 2.11. (Satisfaction) Let a_g be a ground atom, r_g be a ground rule of a rule r ($r_g \in grnd(r)$), and P be a program. Consider an interpretation $I \subseteq B_P$, the satisfaction relationship ($\models$) between I and a_g, r_g, r, P is defined as follow:

- $I \models a_g$ iff $a_g \in I$
- $I \models not\ a_g$ iff $a_g \notin I$
- $I \models B(r_g)$ iff $B^+(r_g) \subseteq I$ and $B^+(r_g) \cap I = \emptyset$
- $I \models H(r_g)$ iff $H(r_g) \cap I \neq \emptyset$
- $I \models r_g$ iff ($I \models B(r)$ implies $I \models H(r)$)
- $I \models r$ iff $I \models r_g\ \forall r_g \in grnd(r)$
- $I \models grnd(P)$ iff $I \models r_g\ \forall r_g \in grnd(P)$
- $I \models P$ iff $I \models r\ \forall r \in P$

The answer sets of a program P are defined in two steps by using its ground program $grnd(P)$: first defines the answer sets of positive disjunctive programs, then reduces the general programs to positive ones and checks a stable condition. An intuitive answer set semantics for positive programs are *minimal models* which is defined as follow:

Definition 2.12. An interpretation $I \subseteq B_P$ is *minimal model* of a positive program P if $I \models P$ and $\nexists M \subset I$ such that $M \models P$.

Definition 2.13. The Gelfond-Lifschitz (GL) reduct P^I of a program P relative to an interpretation I is defined by:

$$P^I = \{H(r) :- B^+(r) \mid r \in P \text{ and } B^-(r) \cap I = \emptyset\}$$

The GL reduct of P reduces the general program P to its positive program by:

- deleting all ground rules $r_g \in grnd(P)$ for which $B^-(r_g) \cap I \neq \emptyset$ holds, and
- deleting all negative bodies from the remaining ground rules.

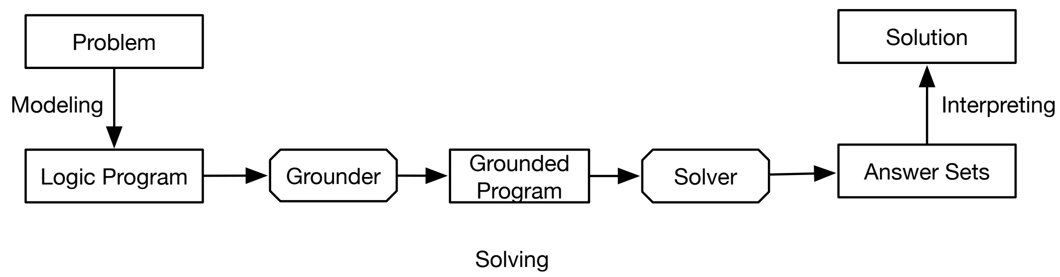


FIGURE 2.6: The ASP solving process

Definition 2.14. Given a program P and an interpretation I of P. I is an *answer set* (or *stable model*) of P if I is a minimal model of P^I.

Example 2.4. *The program in Example 2.2 has 6 answer sets (predicates* `node, edge`*, and* `col` *are removed):*

I_1 = {*color(1, blue), color(2, green), color(3, red), color(4, blue), color(5, red)*}
I_2 = {*color(1, red), color(2, green), color(3, blue), color(4, red), color(5, blue)*}
I_3 = {*color(1, green), color(2, blue), color(3, red), color(4, green), color(5, red)*}
I_4 = {*color(1, red), color(2, blue), color(3, green), color(4, red), color(5, green)*}
I_5 = {*color(1, green), color(2, red), color(3, blue), color(4, green), color(5, blue)*}
I_6 = {*color(1, blue), color(2, red), color(3, green), color(4, blue), color(5, green)*}

2.2.3 The ASP solving paradigm

Following the declarative paradigm, the ASP solving process is depicted in the Figure 2.6. This solving process includes two steps in a pipeline. In the first step (so-called the instantiation or grounding step), a grounder generates a variable-free program of the input program. In the second step (so-called the solving or model searching step), a solver computes answer sets of the grounded program. Those answer sets are interpreted as solutions of the original problem.

The grounding phase generates a ground program (i.e., the propositional program) that does not contain any variable by replacing the variables with all the constants appearing in the program. This ground program has the same answer sets as the original program and is considered as input for the solving phase. Over the past 25 years, various efficient grounders have been developed, including DLV grounder [69], Gringo [70, 71], Lparse [72, 73], GIDL [74], psgrnd [75].

The solving phase takes the output of the instantiation phase and computes answer sets. Informally, this solving step consists of two subtasks [55]: generate candidate models and

checking whether these candidate models are stable. The number of candidate models in the first subtask is potentially exponential and the procedure in the second subtask to decide if a candidate model is a stable model is NP-complete [76]. Therefore, the efficient realization of this phase is needed to enhance the performance of the whole ASP system. The availability of ASP solvers is very various such as Smodels [77], Platypus [78], DLV solver [79], nomore++ [80], GnT [81], assat [82], Cmodels [83], Wasp [84], and Clasp [85].

Instead of offering grounders and solvers individually, the ASP community has combined them into a monolithic system, e.g., Clingo [86] couples the grounder Gringo and the solver Clasp, DLV integrates the DLV grounder and DLV solver. Meanwhile, several Application Programming Interfaces (APIs) have been developed, which allow an embedding program to interact with an ASP system such as the interaction between ASP and Java is supported in DLV Wrapper [87], JASP [88], and EmbASP [89]; the combination of ontologies and reasoning modules is developed in ONTODLV API [90]; the embedding of ASP in Python and Lua programs is available in Gringo and Clasp. In addition, integrated development environments (IDEs) are also available with supportive tools for editing, debugging, testing, and visualizing such as SeaLion [91] and ASPIDE [92].

Due to the availability of a rich modeling language [93] together with efficient systems and engineering environments [94], ASP has been successfully applied in widespread applications in both research and industry fields. For example, the recent paper [95] reports these ASP applications in robotics, bioinformatics, and industry. In robotics, ASP has been used to: generate optimal plans of actions of multiple robots within a given time [96, 97], diagnose failures in the plan execution [98], rearrange multiple movable objects in a cluttered surface [99]. In bioinformatics, [100] and [101] model a biological signaling network and biochemical reactions, [102] reconstructs phylogenies, [103] studies protein structure prediction, and [104, 105] answer complex queries over biomedical ontologies with ASP. In industry, ASP has been integrated in e-tourism [106] to advice promising offers for customers, in workforce management of the Gioia Tauro seaport [107], or in intelligent phone-call routing platform, namely zLog [8], of the Exeura company.

2.2.4 Optimization in ASP

All of current competitive ASP reasoners imitate semantics as defined in Section 2.2.2 by implementing the 2-tier approach as in Figure 2.6. The grounding phase in ASP

[8]http://www.exeura.eu/en/solution/customer-profiling/

systems may be very computationally expensive since the generated ground program is probably of exponential size in regard to the input program. For example, considering the following program which contains one rule and two facts:

```
fact(1), fact(2).
newfact(X_1, ..., X_n) :- fact(X_1), ..., fact(X_n).
```

The instantiation generates 2^n ground rules, corresponding to n variables X_1, ..., X_n over two constants 1 and 2. The output of this grounding phase is the input of the model searching phase. Hence, an efficient grounder plays an important role in the performance of the whole system.

To enhance the instantiation process, substantial efforts have been focused on extending optimization techniques [108]. The most relevant ones can be recalled such as backjumping techniques [109–111] exploit the semantic and the structural information of a rule to reduce the number of useless ground rules [112]; join-ordering methods reorder the body literals of a rule during the grounding process [113]; program rewriting strategies, which inspired from query optimization techniques in relational algebra, automatically rewrite rules by rearranging their projections and selections in the execution tree [114]; dynamic magic sets, which extended from the magic set technique [115], exploit the information provided by the magic predicates in disjunctive programs [116]; and parallel techniques allows for performing concurrent instantiations [57].

Other efforts have been putting restrictions in the syntax of a logic program to improve the grounding step [55]. For instances, Lparse imposes rules on the domain restriction in which very each variable in a rule must appear in some positive domain predicate (i.e., predicates that are not defined via recursive negation or choice rules) [117]. Another restriction on rules which are more relaxed than the domain restriction can be found in DLV or Gringo, namely rule safety [1, 69]. A rule is safe if every variable in that rule occurs in some positive body literal whose predicate is not a built-in comparison predicate.

For the solving phase, different approaches and extensions have been implemented in those ASP solvers. One approach constructs sophisticated search algorithms from the Davis-Putman-Logemann-Loveland procedure which was introduced in [118], e.g., in Smodels, nomore++, and DLV. Another approach utilizes advances of SAT solvers [119], e.g., in assat and Cmodels. An extension of the Conflict-Driven Clause Learning [120] is implemented in Wasp and Clasp. Clasp is enhanced with parallel evaluation methods in [121]. Recently, there are some efforts on applying machine learning techniques to identify the best solver among existing solvers to deal with a particular input program [122, 123].

To close this section, I provide the concept of dependency graph [2, 57, 69] which is widely used in ASP to enable parallelism in the grounding step. This concept is considered as a tool to analyze the structure of non-ground answer set programs so that it helps to enable parallel instantiation algorithms in generating a much smaller ground program equivalent to a given logic program. To do so, the grounders generates instances of rules containing only atoms which can possibly be derived from a given program by taking into account the dependencies among intensional database (IDB) predicates.

According to the database terminology, a predicate occurring only in *facts* is referred to as an extensional database (EDB) predicate, all others as IDB predicates. EDB predicates are relations stored in a database, while IDB ones are relations defined by one or more rules. Thus, an IDB predicate can appear in the body or head of a rule while an EDB predicate is only in the body.

Definition 2.15. Let P be a program. The dependency graph of P is a directed graph $G_P = \langle N, E \rangle$, where N is a set of nodes and E is a set of arcs. N contains a node for each IDB predicate of P, and E contains an arc $e = (p, q)$ if there is a rule r in P such that q occurs in the head of r and p occurs in a positive literal of the body of r.

Example 2.5. *Consider the following program P, where a is an EDB predicate:*

$$p(X,Y) \vee s(Y) :- q(X), q(Y), not\ t(X,Y).$$

$$q(X) :- a(X).$$

$$p(X,Y) :- q(X), t(X,Y).$$

$$t(X,Y) :- p(X,Y), s(Y).$$

The dependency graph of P is illustrated in Figure 2.7 where a is not appeared in the graph since it is an EDB predicate.

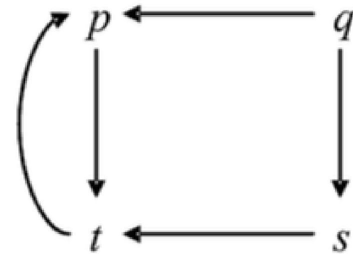


FIGURE 2.7: A example of a dependency graph [2]

2.3 Stream Reasoning

Stream reasoning (SR) emerged as a research trend provides solutions for applying logical reasoning techniques over massive data streams in the scalable way [30]. In particular, SR performs a set of logical rules, which express knowledge bases and reasoning tasks, on input streams and derives actionable knowledge. The research on SR has been rooted from different research areas: Active Database Management (ADM) [124], Data Stream Management Systems (DSMS) [4] and Complex Event Processing (CEP) [27], and logical reasoning [125]. In fact, SR requires new theoretical investigations that go beyond those research areas such as which logical language is most appropriate or how to define

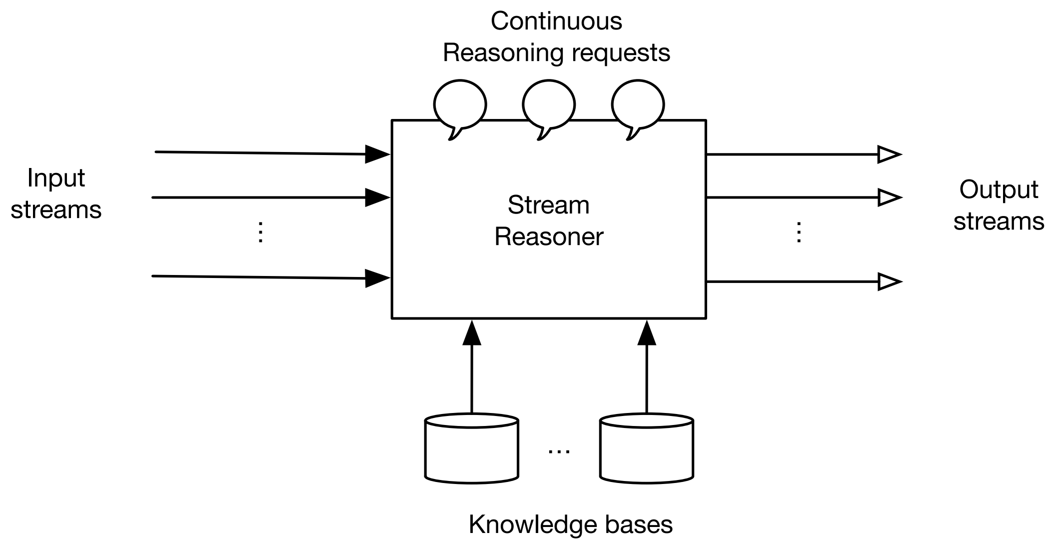


FIGURE 2.8: The conceptual architecture of a SR system

soundness and completeness for stream reasoning. Since the term SR introduced, this area has seen multiple promising results and various applications have been built on the top of them.

2.3.1 Conceptual Architecture

Figure 2.8 shows the conceptual architecture of a SR system. The stream reasoner takes multiple data streams as inputs and produces outputs as streams. Input streaming data are considered as infinite sequences of time-varying data elements [29], and once a data element has been streamed by, it can not be revised. The output streams can be stored or processed by other streaming systems. The stream reasoner also requires stored knowledge bases to provide richer answers. The knowledge bases should be static or infrequently updated. Similar to stream processing systems where users can register continuous queries, continuous reasoning requests can be registered at the stream reasoner. Those requests are evaluated continuously based on a combination of input streams and knowledge bases. The reasoning requests are expected to have higher expressivity than queries so that they can capture more sophisticated requirements of users.

2.3.2 Stream Reasoning Development

The rise of data streams has evoked new challenges about managing, processing, querying, and reasoning over infinite sequences of data with high frequency rate. In this

section, I highlight two most prominent approaches that led to the growth of the SR research area: DSMS [4] and CEP [27].

DSMS solutions have been developed in the database community in order to cope with high rates of data updates. Extending from the ADM approach [126], DSMS adds the concept of *stream* is as follows:

> A stream S is a (possibly infinite) bag (multi-set) of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in T$ is the timestamp of the element. [4]

A stream is usually unbounded but at a given timestamp, the number of elements is finite. There is no assumption on data arrival order. Size and time constraints of DSMS applications make it difficult to store and process data elements after their arrival. They require an in-flow processing model.

A conceptual architecture for DSMS is reported in Figure 2.9 which is directly taken from [3]. A DSMS system is modeled as a set of continuous queries Q, n input streams ($n \geq 1$), and four possible outputs:

- (Derived) *stream* is intermediate elements in the answer produced by operators in the query once and never changed.
- *Store* is to keep elements of the answer that may be changed or removed at a certain point in the future.
- *Scratch* is considered as working memory of the system. It keeps data which is not part of the answer but may be needed in computing the answer.
- *Throw* represents the recycle bin in which the system throws away unnecessary data.

DSMS adopts a new interaction paradigm by introducing: *continuous queries* - ones are deployed once and continue to produce results (as new stream items arrive) until removed, and *windows* - operators to limit the portion of an input stream from which elements can be selected to process [25]. This form of interaction is called the Database-Active Human-Passive model as opposite from the Human-Active Database-Passive model in the traditional database management systems [127].

One of main contributions in DSMS is the work developed by researchers in Stanford University, namely Continuous Query Language (CQL). CQL is a SQL-based declarative

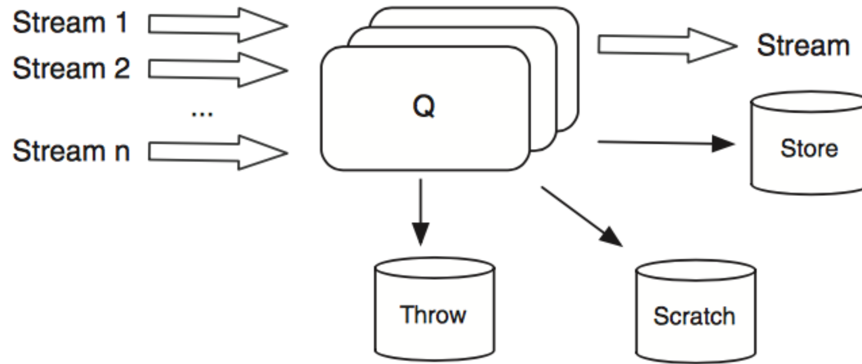


FIGURE 2.9: The conceptual architecture of DSMSs [3]

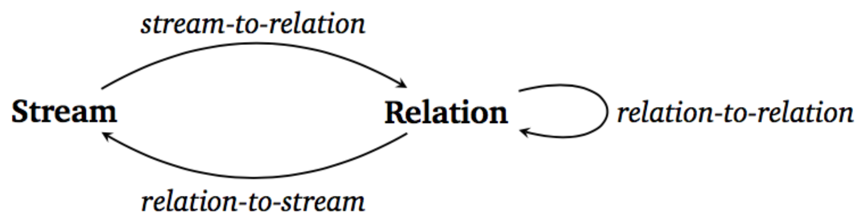


FIGURE 2.10: The CQL model [4]

language over streams. This work defines a precise abstract semantics for continuous queries based on the concepts of streams and *relations*:

> A relation R is a mapping from each time instant in T to a finite but unbounded bag of tuples belonging to the schema of R. [4]

With the formalization of streams and relations, the authors constructed a stream processing model through three classes of operators, depicted in Figure 2.10:

- *Stream-to-relation* operators produce a relation R from an input stream S. In other words, these operators extract finite bags of data elements (relations) from a potentially infinite bag of timestamped data elements (streams). To do so, they are mainly the concept of *sliding window* to take a snapshot of a stream. There are several types of sliding window proposed in the-state-of-the-art such as fixed windows [128], tumbling windows [129], value-based windows [130], etc. Below, I report two most popular sliding windows:
  - A *time-based sliding window* on a stream S takes two time intervals as input parameters: width ω (the dimension of the window) and slide β (the distance

between two consecutive windows). Its output relation at time t is:

$$R(t) = \begin{cases} \emptyset \text{ if } t < \omega - 1 \\ \{s | \langle s, t' \rangle \in S \text{ and } t' \geq t_{end} \text{ and } t' \leq t_{start}\} \text{ if otherwise} \end{cases}$$

where $t_{start} = \lfloor t/\beta \rfloor.\beta$ and $t_{end} = max\{t_{start} - \omega, 0\}$.

  - A *tuple-based sliding window* extracts a fixed number of last data items on a stream S. It also takes two parameters width ω and slide β as input. However, the width indicates that the current window should contain exactly ω data items while the slide specifies β data items are removed or added at each window.

- *Relation-to-relation* operators produce a relation R from one or more relations $R_1, ..., R_n$ $(n \geq 1)$. These operators are derived from the traditional relational algebra expressed in SQL.

- *Relation-to-stream* operators produce an output stream S from a relation R. Opposite to stream-to-relation operators, those stream out data elements in relations. In CQL, there are three types of such operators are defined: RStream (i.e., produces the computed timestamped set of relations at each step), IStream (i.e., streams out the difference between the timestamped set of relations computed at current step and previous step), and DStream (i.e., streams out the difference between the timestamped set of relations computed at previous step and current step).

CEP solutions also focus on analyzing data streams and generating insights on the current situation. While DSMS is well suited for processing streams which require aggregations or assessing occurrence in intervals, CEP has the capability to construct new pieces of information by means of temporal relations among data elements in streams. To do so, CEP associates a specific semantics to data items, so-called *events* defined as "An object that represents, encodes, or records an event, generally for the purpose of computer processing. [131]". Other definitions of events can be found in the-state-of-the-art of CEP such as an event is a significant change in the state of the world [132], it can also be a thing that did not happen at all [133], or it is a computer processable object that formed with particular attributes [27]. Categories of events are reported in three dimensions: duration, context, and complexity. Under the duration dimension, an event can happen at a time point (instantaneous) (e.g., an eye blinking) or can happen for a period of time (e.g., a football match). With the context dimension, an event can happen within a system (internal) (e.g., a reasoner fires a rule) or can happen outside a system (external) (e.g., a weather change notification). On the complexity dimension, an event can be primitive or is not considered as summarizing or denoting a set of other

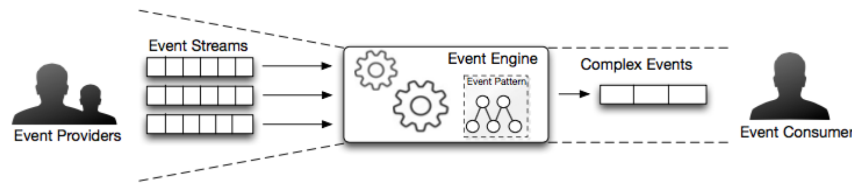


FIGURE 2.11: The CEP conceptual architecture

events (simple) (e.g., a phone call) or can be composite or is created by combining a set of other events (complex) (e.g., a stock market crash).

Figure 2.11 illustrates the conceptual architecture of CEP. A CEP system is modeled as an agent receives input events from event providers, evaluates in real-time with the event processing logic, and streams out complex events for event consumers. CEP considers observable raw data as primitive events and aims at detecting occurrences of composite events. These composite events are ones whose occurrence relies on the occurrence or absence of other events. For instance, the fact that a traffic jam on a road is detected when there is no traffic light on that road but different sensors located on that road keeps notifying a number of cars greater than 40 and average speed of cars lower than 10 km/h in last 30 minutes. To detect such complex events from data streams, an event pattern is registered at those CEP systems. It is defined as "A template specifying one or more combinations of events" [24].

Also in [24], four categories of event patterns are reported:

- Logical pattern: describes the occurrence (or non-occurrence) of relevant event types using logical operators such as conjunction, disjunction, exclusive-disjunction, and negation.
- Attribute-based pattern: describes the occurrence (or non-occurrence) of relevant event types based on constraints over event attributes.
- Dimensional pattern: describes the temporal, spatial, or spatio-temporal relations between event types.
- Aggregated pattern: describes the occurrence (or non-occurrence) of relevant event types based on the occurring rules of event instances.

An event pattern is constructed from given operators which express sequencing and ordering relationships. Examples of such operators are depicted in Figure 2.12 in which the horizontal bars express the result evaluation over such operators. The formal semantics of those operators can be found in [5].

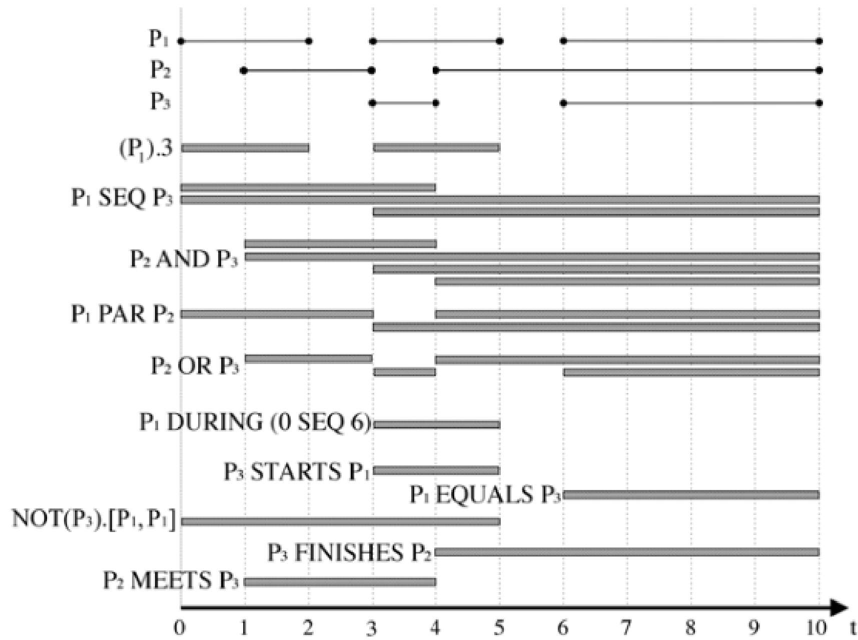


FIGURE 2.12: CEP operators [5]

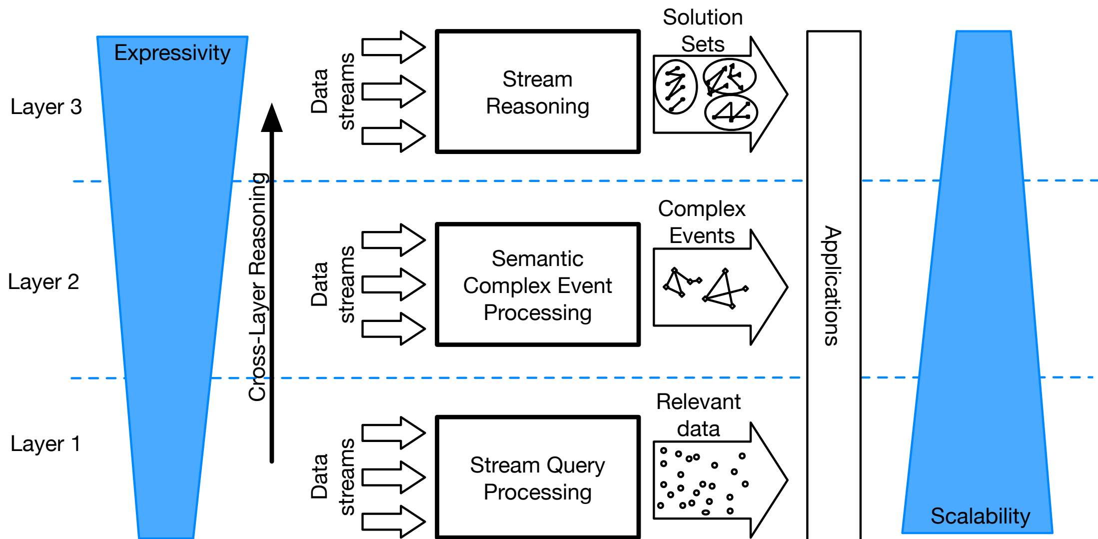


FIGURE 2.13: SR Layers [6]

2.3.3 Stream Reasoning Categories

Considering existing approaches and solutions for processing SW data, authors in [6] characterized SR into three main layers with respect to the expressivity of the reasoning tasks that they can support. Figure 2.13 shows the representation of these three layers with expressivity increase.

- *Stream Query Processing Layer.* This bottom layer includes all systems that inherit the processing model of DSMS. Their continuous queries are extended from

SPARQL to deal with semantically annotated input, namely RDF streams. Existing RSP query languages support a subset of SPARQL 1.1 features and operators to form query patterns [134] and have different underlying semantics [62]. These semantics differences rise heterogeneous consequences at the query evaluation and result production. Example stream query processing systems include: CQELS [20], C-SPARQL [19] , IMaRS [135], TrOWL [41], Streaming-SPARQL [136], Streaming Knowledge Bases [40], Sparkwave [38].

- *Semantic Complex Event Processing Layer*. This middle layer follows a different approach than the previous one. The systems in this layer inherit the processing model of CEP systems. Most of existing CEP solutions process events at a syntactical level and can not share definitions for the syntax or semantics of composite events. This isolates those solutions [137, 138] from others and unable to integrate with richer knowledge representation and reasoning [139]. To overcome these issues, the approaches in this layer mainly leverage from rule-based or Nondeterministic-Finite-Automata-based techniques [140]. In the first approach, an event pattern is described by rules and input events are considered as facts. In the second approach, an event pattern is specified as a state model and input events are used to control state transitions. Examples of CEP engines Sase+ [141], TESLA [142], Etalis [60], EP-SPARQL [36].

- *Stream Reasoning Layer*. This top layer investigates in approaches to deal with more complex reasoning tasks than previous layers and be able to infer new logical conclusions from input streams. Solutions in this layer aim at enabling the capability to deal with non-monotonicity, uncertainty, defaults, preferences, or common sense inferences. Systems in this layer mainly leverage from expressive logic reasoning paradigms such as temporal logics [143], temporal action logics [144], event calculus [145], description logic [146], or ASP. The processing of data streams with such logics has brought many positive results such as LARS [51], DyKnow [49], Ticker [147], Laser [148], P-MTL [149].

Other attempts follow the *cross-layer reasoning* approach by combining solutions from different layers, each one is responsible for processing data streams at different levels of abstract. For example, [150, 151] leverage advances from a semantic complex event processing system with a production rules system, [54] ties up a stream query processing system with an ASP reasoner. Such solutions normally combine the underlying layers in a pipeline. Each layer filters and abstracts input streams and streams out relevant data to the next layer.

2.4 Summary

In this chapter, relevant concepts and techniques in the SW, ASP, and SR are introduced as the background for the research carrying on this thesis. In particular, I presented the conceptual architecture of SR systems, highlighted two major research areas that have contributed the most in the development of SR, and its classifications based on the complexity of reasoning tasks. For SW, the foundation for modeling, representing, and querying static SW data is introduced. It is followed by the extension of SW concepts in the streaming environment. Besides, a discussion about ASP solving paradigm is presented together with its formal syntax and semantics. In this thesis, I leverage the techniques in these three research areas and integrate them to enhance the expressivity and scalability of a stream reasoner over RDF data.

Chapter 3

Related Work

Works towards SR have resulted in many different perspectives and focuses. This chapter presents related work in two sections regarding two main research questions mentioned in Chapter 1. Section 3.1 analyzes existing languages and engines in the SR area, as well as their evaluation. Section 3.2 discusses relevant techniques to optimize the reasoning process over data streams. The chapter is summarized in Section 3.3.

3.1 Stream Reasoning

This section describes existing works related to research question *RQ1* which demands a highly declarative language for continuous reasoning over RDF streams. Various approaches for modeling and processing streams have been proposed in the state-of-the-art, which rely on concepts from DSMS, CEP, and logical reasoning. First, the existing SR languages and engines are presented in Section 3.1.1 and then the collection of benchmarks for evaluating them is detailed in Section 3.1.2.

3.1.1 Languages and Engines

In the following, key representative approaches are selected to be presented for discussion and are categorized into three classes, namely SPARQL-, CEP-, Logic-based approaches. This classification is relatively driven from the characteristics of supporting languages for accessing and querying streaming data.

3.1.1.1 SPARQL-based approaches

The commonality across these approaches are the usage of SPARQL 1.1 to model queries over RDF streams. Those query languages focus on extending the syntax and semantics of SPARQL 1.1 with streaming operators. For instance, `STREAM` clause is added into the SPARQL query with window operators in order to allow a user to specify input streams and how to take a portion of them to process. Below, prominent solutions under this category are discussed.

One of the first SPARQL-based query language proposed for processing RDF streams is *Streaming-SPARQL* [136]. Its syntax is extended from SPARQL 1.1 with a set of keywords and grammar to allow the definition of time- and count-based windows over data streams. The keyword `STREAM` followed by an Internationalized Resource Identifier (IRI)[1] describes an input data stream and the keyword `RANGE` followed by a number defines the size of the window on that input stream. The semantics of Streaming-SPARQL extends the logical SPARQL algebra for stream processing on the foundation of a temporal relational algebra based on multi-sets. In this way, an input stream is not defined as a sequence but as unlimited multi-set of triples. Each element in the stream is an RDF triple annotated with two parameters t and n in order to express that the triple is valid at time t and occurs n times at that point in time.

Barbieri et al. [19, 22, 34, 152] introduced *C-SPARQL* (i.e., Continuous SPARQL) as a new language for continuous queries over streams of RDF data. This language extends from SPARQL 1.1 with new features such as RDF stream data type, aggregates, window management, and timestamps. Similar to Streaming-SPARQL, C-SPARQL provides a set of keywords and grammar to define clauses which capture such new features. It also changes the semantics of SPARQL 1.1 queries from one-time semantics to continuous semantics with the formalization of aggregates, windows, and the timestamp function. Additionally, C-SPARQL supports to query RDF streams while taking into account the background knowledge expressed in RDF datasets. This language is implemented in the C-SPARQL engine, where users can register continuous queries. Registered C-SPARQL queries are then mapped to an internal model which enables automatic decomposition and transformation processes in order to generate intermediate static and dynamic queries. After the transformation, those queries are evaluated by a suitable orchestration of Esper[2] and Jena[3] as reasoning engines. Esper executes queries over RDF streams and produces a sequence of RDF graphs over time while Jena is responsible for executing a

[1] https://www.w3.org/TR/rdf11-concepts/section-IRIs
[2] http://www.espertech.com/esper/
[3] http://jena.apache.org

standard SPARQL query against each RDF graph in the sequence and produces continuous results. C-SPARQL engine accepts one-timestamped RDF triples as input and returns results as specified in its query form.

Another SPARQL-based language to process RDF streams is $SPARQL_{stream}$ [21, 35]. Similar to C-SPARQL, an RDF stream in SPARQL$_{stream}$ is defined as a sequence of triples and each triple is annotated with one timestamp. The query language of SPARQL$_{stream}$ is inspired from C-SPARQL and SNEEql [153], but is improved to correct the types supported and the semantics of windowing operators. SPARQL$_{stream}$ focuses on solutions for streaming data mapping and querying using ontology-based approaches. This language is implemented in the ontology-based streaming data access service which is built on top of several existing technologies such as continuous data querying, ontology-based data access, and SPARQL query processing. This service has three main processes, namely query translation, query processing, and data translation. When a SPARQL$_{stream}$ query is registered, the query translation process will transform it, which expressed in terms of the ontology, into target continuous query (i.e., SNEEql) by means of proposed S_2O mapping language [35]. This target continuous query is evaluated by the query processing phase to extract the relevant data. The result of this phase is a set of tuples which then fed into the data translation process to transforms into ontology instances by using S_2O.

CQELS (Continuous Query Evaluation over Linked Streams) [20] is presented as a native and adaptive query processor for unified query processing over Linked Stream Data and Linked Data. Similar to C-SPARQL, CQELS adopts the processing model of DSMS and extends SPARQL 1.1 with windowing and relational operators to process one-timestamped RDF streams. However, there are three main differences distinguish between them. Firstly, differently from C-SPARQL engine that is implemented based on the "black box" approach which leverages the processing capabilities from other engines, CQELS is proposed based on the "white box" approach and is implemented naively. This white box approach is claimed to avoid the overhead and limitations of the use of existing engines. Secondly, while C-SPARQL supports RStream in relation-to-stream operators, CQELS supports only IStream. Thirdly, the query evaluation in C-SPARQL is periodic while in CQELS, the execution is triggered by the arrival of new triples. CQELS with its native approach enables to apply query rewriting techniques to make its query processor dynamically adapting to the changes in the input. In this way, it is claimed to improve the performance of the query execution in term of delay and complexity.

Another approach is inheriting advances from the Rete algorithm [154], which is presented in [38]. Authors introduced *Sparkwave* as an approach for continuous schema-enhanced pattern matching over RDF streams. It detects from one-timestamped RDF data streams the triple pattern conjunctions bound over joining variables but those triples must occur inside a same time window. Besides RDF streams, Sparkware takes graph pattern (i.e., a conjunction of triple patterns) as input and operates over a fixed RDF schema which limits it in supporting for static data instances. The pattern matching process is based on the Rete algorithm to associate input sources to compute entailments. To address RDF schema entailment, Sparkwave extends Rete with an additional network called ϵ-network. This network is positioned in front of the normal Rete network and is responsible for generating triples following schema entailments.

Following the similar approach as Sparkware, *INSTANS* [155] is also based on the Rete algorithm to process streams. It supports the ability to evaluate simultaneous and continuous multiple queries. Users register their multiple interconnected SPARQL 1.1 queries and rules. Then, INSTANS continuously evaluates incoming RDF data streams against the compiled set of queries and stores intermediate results into a Rete-like structure. When all the conditions are matched, the result is instantly produced.

Those mentioned approaches provide different answers at disparate moments due to the heterogeneity of their operational semantics, which render the process of understanding and comparing continuous query results. This raises a need from the RSP community to unify them into one. To address this challenge, authors in [62] proposed RSP-QL, a unifying formal model for representing and processing RDF streams, that reflects the different semantics of existing RSP engines. To do so, they extended the temporal dimension in the RDF data model in three ways: timestamped RDF graphs are RDF statements with a time annotation, RDF streams are ordered sequences of timestamped RDF graphs, and time-varying and instantaneous RDF graphs capture the evolution of RDF graph over time. An RSP-QL query is a continuous one which extended from SPARQL, whose correctness can be formally assessed and that can capture the processing model of existing systems (i.e., C-SPARQL, CQELS, and SPARQL_{stream}). This work constitutes a contribution to ongoing efforts in the SW community to provide standardized and agreed definition of extensions to RDF and SPARQL for managing data streams.

3.1.1.2 CEP-based approaches

Approaches in this direction mainly inherit the processing model of CEP systems for detecting events from data streams. They consider each raw data element in the stream

as an atomic event, and their solutions aim at extracting implicit knowledge from input streams in form of complex events.

One of the popular complex event processing engines over data streams is the Event TrAnsaction Logic Inference System (ETALIS) [5]. The engine receives atomic events which are annotated with two timestamps as input streams and produces detected complex events which indicate the changes happening in near real-time as output streams. Users can register complex event patterns in ETALIS using its declarative rule-based language, namely ETALIS Language for Event (ELE). To detect events specified in ELE at runtime, ETALIS uses the event-based backward chaining algorithm which converts queries to Prologs rules and evaluates them whenever data arrives. ETALIS supports multiple Prolog engines such as YAP [4] and SWI [5] with three consumption policies: recent (i.e., the latest input events which can be matched are selected for matching the event patterns and are ignored in the next evaluations), chronological (i.e., the earliest input event which can be matched are selected for matching event patterns and are ignored in the next evaluation), and unrestricted (i.e., all input events are selected for matching event patterns). While detecting complex events, ETALIS may need to evaluate the background knowledge on the fly and infer implicit knowledge as logical sequences from deductive rules.

Darko Anicic et al., who proposed ETALIS, extended ELE to enable the usage of ETALIS in SW applications. This extension, called *EP-SPARQL* [36, 156], allows a user to define complex event tasks for ETALIS in a SPARQL-like language. In this way, the input streams and the background knowledge are both represented in the RDF format. The input events are RDF triples annotated with two timestamps, which represent the lower and upper bound of the occurring interval (interval semantics). The knowledge base is expressed in the form of RDFS ontologies and is converted into a Prolog program at design time. EP-SPARQL queries are translated into logic expressions in ELE and are evaluated continuously by ETALIS at runtime.

Authors in [157] studied the integration of the currently available CEP features into RSP-QL. This work results a new language *RSEP-QL* whose operators aim at enabling both DSMS and CEP features. RSEP-QL models both event patterns and their evaluation semantics taking into account the presence of selection and consumption policies. In addition, it captures the behaviors of both EP-SPARQL and C-SPARQL. However, the work focuses mainly on defining semantics for the `SEQ` operator in the integration with RSP-QL while others of CEP operators (such as `DURING`, `NOT`, etc) are not considered.

[4]ttp://www.dcc.fc.up.pt/ Evsc/Yap/

[5]http://swi-prolog.org/

3.1.1.3 Logic-based approaches

Approaches in this direction mainly inherit from reasoning techniques which are available in the KRR community. These efforts aim at creating expressive SR systems which are able to scope with more complex tasks over streams rather than systems in SPARQL- and CEP-based approaches. Reasoning techniques, such as default reasoning, non-monotonicity, preference, or multiple possible solutions are important to deal with missing or incomplete data, users' preferences, or enumerating alternative solutions. Several works have been proposed in the attempt of enabling such expressive reasoning capabilities in the streaming environments.

Reasoning over changing data or program has been considered in ASP. Incremental ASP [46] uses the module theory in [158] to introduce new techniques for incremental grounding and solving. The domain description is expressed as a triple (B, P, Q) of logic programs in which B describes static knowledge, P and Q are slices that depend on a parameter t. P captures knowledge accumulating with increasing t while Q is specified for each value of t. As regards grounding, all instantiations which are derivable from static knowledge can be grounded only one while the parameter t in a rule of the program part P or Q is instantiated with a new value at each step. Regarding solving, the model computation can be executed incrementally replying on the composition of modules. Following this work, the same authors augmented the concept of the incremental logic program and its mechanism with asynchronous information, aiming at reasoning about real-time dynamic systems.

Reactive ASP [47] provides additional means to add new data online at running time via so-called online progressions of external events and inquiries. While entire event streams are made available for reasoning, inquiries act as punctual queries. At each step, external information can be incorporated to ground new rules dynamically. However, the program P and Q in reactive ASP cannot express the relevance of information relative to an interval step. To enable such a window mechanism, reactive ASP is extended with time-decaying logic program [48] in which the program Q is a sequence $\{Q_1, ..., Q_m\}$. The instantiation of each Q_i expires after a specified lifespan of n steps. Ideas from incremental ASP, reactive ASP and time-decaying logic programs have been improved continuously and are now subsumed in the current version 5 of Clingo [159].

Targeting at filling the missing theoretical underpinnings for SR which has been raised in [14], authors in [51] conceived a Logic-based framework for Analytic Reasoning over Streams (LARS). They first provided a formal model of a stream as a sequence of time-annotated formulas and a generic notion of window function as a sub-stream. The

practical usage of this window notation includes time-based windows (i.e., data selection is based on temporal constraints), tuple-based window (i.e., data selection is based on order constraints), and partition-based windows (data selection is based on a mix of order and semantic information). In addition to the usual boolean operators (i.e., and, or, implies, not), authors defined three temporal logic operators $\Box$, $\Diamond$, and @, to represent the fact that a formula holds at all times, some time, and a specific time point in a window, respectively. To account for the aspect of streaming, LARS supports window operators of the form $\boxplus$ to restricts the scope on which enclosed formula applies. LARS language includes: LARS formulas and LARS programs - set of rules similar as in Datalog but built upon LARS formulas. Additionally, LARS programs are equipped with multiple-model semantics as in ASP to deal with incomplete information and negation. Authors proved that LARS captures the CQL language (including aggregates) and ETALIS semantics. In [160], LARS is extended with a filter window to select data based on semantic information and generalized time-based window to access the future. Moreover, the computational complexity of model checking and satisfiability for both LARS formulas and programs are investigated as well as the use of LARS is explored via use cases such as cache management in content-centric networking or dynamic knowledge-based configuration of cyber-physical systems.

Fragments of LARS have been implemented recently in prototype engines called Ticker [147] and Laser [148]. Both of them explore incremental reasoning techniques on a practical fragment of LARS called plain LARS on different realization principles. Ticker supports sliding time- and tuple-based windows and can be run in two evaluation modes. The first one exploits the static encoding and calls ASP-based reasoner (i.e., Clingo) for one-shot evaluation. This mode is claimed as a choice for a stratified program (i.e., single solution) where no ambiguity arises which model to compute. The second mode carries out incrementally adapting a model based on an incremental adjustment of a program and supports use cases that have multiple potential solutions. The program update is built based on the extension of justification-based truth maintenance system [161] that also allows for removing expired ground rules. The experimental evaluation of Ticker is conducted based on two small scenarios (i.e., caching strategy and content retrieval) and reports the run-time performance of Ticker in these two modes [147].

Developed in parallel with Ticker, Laser focuses on applications which have deterministic solutions. Hence, this engine supports positive and stratified plain LARS programs. Its incremental evaluation is based on the idea of annotating formulas with two time markers, namely consideration time and horizon time. These time makers indicate time interval in which formulas hold and enable the implementation a technique similar to the semi-naive evaluation in Datalog [162] to reduce duplication derivations. Laser

is compared against C-SPARQL, CQELS, and Ticker (using Clingo mode) on single-rule programs and the cooling system use case. It is reported that Laser has been outperformed significantly [148]. However, both Ticker and Laser do not support for reasoning on SW data streams.

Other attempts towards expressive stream reasoning in SW applications follow the idea of cross-layer reasoning in Figure 2.13 such as ASR [53] and StreamRule [54]. In particular, ASR calls the DLVhex solver [163] and StreamRule uses the Clingo solver [48] as a subprocess to infer new knowledge from data streams and a given rule set. In order to support ASP solvers for reasoning about RDF data streams, a middle layer is required for transformation between data formats. For example, the StreamRule system intercepts the query results (output RDF stream) filtered by the RSP engine and translates them into ASP syntax before streaming them into the ASP reasoner Clingo. Given the data transformation overhead, the performance of the reasoning subprocess should be measured by not only the processing time of the solver but also the time required for data transformation. In addition, the ASP solver in the subprocess needs to return results faster than when the new input data arrives, in order to ensure the stability of the whole system. This requires optimization techniques that can further speed up the processing.

	C-SPARQL	CQELS	EP-SPARQL	SPARQL $_{stream}$	Laser	Ticker
Data model	RDF	RDF	RDF	RDF	Predicate	Predicate
Time model	ST	ST	Interval	ST	ST	ST
Query language	SB	SB	SB	SB	LARS	LARS
Time-based window	✓	✓		✓	✓	✓
Count-based window	✓	✓			✓	✓
Background knowledge	✓	✓	✓	✓	✓	✓
Reasoning capability	RDFS		RDFS		ASP	ASP

Table 3.1: Comparison of SR systems. (Note: ST stands for Single timestamp and SB for SPARQL-based)

To summarize, Table 3.1 lists representative SR systems and compares them according to data models, operators, and reasoning capabilities that they support. Except for Laser and Ticker which focus on predicate streams, other systems support RDF-based data models and SPARQL-based query languages. Most stream reasoners annotate a single timestamp to each data item and integrate with background knowledge which enables richer functions for reasoning. However, none of the current systems addresses the merging of large background data with streams [164]. Due to the fact that semantic reasoning is computationally expensive, some existing solutions implement only a reduced set of

reasoning entailments. For example, C-SPARQL and EP-SPARQL consider subsets of RDFS, or Laser and Ticker partially support ASP reasoning.

3.1.2 Comparative Analysis

One of open challenges in SR area is how to systematically evaluate implementations against well-defined quality criteria. Among variety of proposed engines, their results have crucial differences in operational semantics. A common benchmarking framework would not only help to assess differences and limitations of existing implementations, but also provide a basis for steering future research directions and standardization efforts. Thomas et al. [165] investigated the key issues that such systems must face and identify best practices to design a benchmark with three Key Performance Indicators (KPI) and seven commandments for designing the stress tests of system evaluation. Several benchmarking systems focus on different aspects mentioned in [165]. Bellow, I discuss a set of prominent benchmarks that are strongly related to RSP and ASP reasoning for evaluating existing engines.

3.1.2.1 RSP Benchmarks

SRBench [134] is the first proposal in this direction for functionally evaluating streaming RDF/SPARQL engines. The basic dataset of SRBench is built from the real-world dataset, namely Linked Sensor Data (LSD)[6], published by Kno.e.sis[7]. This dataset is part of the Linked Open Data Cloud[8] and contains weather sensors observations which are collected from 2002 and semantically annotated using the Semantic Sensor Network (SSN)[9] ontology. Together with weather sensors streams, SRBench uses two static datasets, GeoNames[10] and DBpedia[11], to assess the interlinkable ability of a system. Those datasets are claimed to be cautiously chosen to meet the data requirements of designing a benchmark such as relevant [166], realistic [167], semantically valid [168], and interlinkable [169]. Besides, SRBench verbally describes 17 benchmarking queries which are classified into three parts: graph pattern matching, solution modifiers, and query forms. The first class includes queries that cover the basic graph pattern matching operators such as `AND, FILTER, UNION, OPTIONAL`. The second class includes ones with projection and `DISTINCT` modifiers which modify the results of the graph pattern matching operators. The last class determines the form of the final output of a query (`SELECT,`

[6]http://wiki.knoesis.org/index.php/LinkedSensorData
[7]http://knoesis.wright.edu
[8]https://lod-cloud.net
[9]https://www.w3.org/TR/vocab-ssn/
[10]http://www.geonames.org/ontology/documentation.html
[11]https://wiki.dbpedia.org

`CONSTRUCT`, and `ASK`). Finally, the authors of this work complemented their benchmark on three systems: SPARQL$_{stream}$, CQELS (Aug.2011), and C-SPARQL (0.7.4). The conclusion drawn from that experiment is some of the queries are not supported by the existing engines and therefore cannot be executed. In short, SRBench assesses the system's ability to deal with streams, background integration and reasoning features. However, this benchmark supports very limited reasoning capability. The queries involve reasoning only over three properties `rdfs:subClassOf, rdfs:subPropertyOf` and `owl:sameAs`.

A second proposal is *LSBench* [170]. It includes a synthetic dataset on social networks and a set of 12 queries covering both stream processing and background integration. Its dataset is generated by the proposed Stream Social network data Generator (S2Gen) to simulate the stream data in social networks. S2Gen offers 3 ranges of parameters for flexibility: the generating period - the period of time in which the social activities are generated, the maximum number of posts/comments/photos for each user per week, and correlation probabilities. There are 3 social data streams (i.e., GPS stream, posts and comments stream, and photos stream) and 1 static data (i.e., user profile) in the dataset of LSBench. Authors of this benchmark argued that 3 characteristics of existing engines are critically important to be taken into account in evaluation: (i) even supported languages of all considered engines extend from SPARQL and CQL but they have some difference in semantics, (ii) the execution mechanisms of such engines are also different, and (ii) the outputs of a single test in each engine can be various due to any change in the running environment and experiment parameters. Based on those arguments, they aim at assessing not only query language features but also systematic measures of well-known quality criteria (e.g., throughput, latency) over three engines C-SPARQL, CQELS, and JTALIS (a Java wrapper for ETALIS). In addition, LSBench raises an issue about the correctness test by defining a way to compare the mismatch among outputs of those engines. However, this benchmark does not include any reasoning assessment.

Focusing on the theme of correctness, authors in [171] proposed *CSRBench* for checking the correctness of RDF stream engines, which was not targeted by SRBench and LSBench. The authors of this work showed that such engines comply with their own semantics and these may differ from each other via a simple example as an evidence. Therefore, two assumptions about correctness that had been used as default in other benchmarks (i.e., the tested systems work correctly, and the tested systems have the same operational semantics) do not hold anymore. Deepening in the understanding of the semantics of RDF continuous processing, three main dimensions which affect query results are identified: system (e.g., the initial time for window, timestamp policy, and notification policy), query (e.g., query operators, window slide, and window size),

and input stream data (e.g., data rate, window content size, and data stream distribution). Taking into account such dimensions, CSRBench extends from SRBench with parametrized queries in window size, window slide, aggregation, and join operator over different timestamped triples. In addition, this benchmark includes an oracle that generates and compares results of RDF stream processors and checks their correctness. The prototype of this proposed oracle is implemented and experimented over three representative engines C-SPARQL (0.9), CQELS (Aug 2011) and SPARQL_{stream} (1.0.5). The dataset was used for this particular experiment is a small subset of SRBench dataset which contains weather observations of hurricane Charley in the US for 3 hours. It was reported that none of these engines passes all tests. The benchmark has a step moving in the direction of correctness verification but does not add any extra reasoning-related things from SRBench.

CityBench [172] is a benchmarking suite to evaluate RDF stream processing engines within the context of Smart City applications. Different from other previous benchmarks which mainly focus on features of query languages and engines, CityBench is dedicated to assessing the behavior of such engines with taking into account dynamic features that are significant in real-time scenarios. The authors analyzed 7 dynamic requirements that can potentially affect performance, scalability and correctness of smart city applications: (1) address high data distribution, (2) scope with unpredictable data arrival rate, (3) handle the increasing number of concurrent queries, (4) deal with a large amount of background data while processing streams, (5) update the quasi-static background during query execution, (6) support on-demand discovery of data streams with taking into account quality constraints, (7) be able to switch between multiple semantically equivalent data streams during query execution. Based on that, CityBench is built with three components: datasets, a testbed infrastructure, and queries. Firstly, the datasets are leveraged from outcomes of the CityPulse project[12] which includes both real and syntactic data about the city of Aarhus, Denmark. There are 5 streaming datasets collected from sensors (i.e., vehicle traffic, parking, weather, pollution, and user location), 2 quasi-static datasets about cultural and library events in the city. Among them, pollution and user location streams are synthetic. All data is annotated with the SSN ontology. Secondly, a testbed infrastructure is designed in a configurable way which allows users to configure a variety of metrics for the evaluation. The testbed supports 6 configuration metrics: changes in input streaming rate, playback time, variable background data size, number of concurrent queries, number of streams within a single query, and selection of RDF stream engines. Thirdly, a set of queries includes 13 verbal queries in the context-aware travel/parking planner/administration. In addition, the

[12] http://www.ict-citypulse.eu/page/

flexibility of CityBench is highlighted through an experimental evaluation over two representative RDF stream engines CQELS and C-SPARQL. The experiment compares the latency, memory consumption, and completeness of these two engines and its results are reported that no existing engines can handle efficiently quasi-static data, and support adaptation in stream processing. The contribution of this benchmark in this research area is opening a way to evaluate engines under real-time circumstances. However, it does not contribute to assessing the expressivity of reasoning tasks that such engines are supporting.

While pioneering benchmarks isolate evaluate RSP engines against various dimension (i.e., functional coverage, performance, correctness), *YABench* [173] is emerged to assess those engines with all such dimensions. To do so, the YABench framework is designed with 4 components: a stream generator creates input streaming data, an oracle that evaluates the correctness of results, a set of supported engines, and a visualization tool of results. The stream generator creates RDF data streams from weather observations in LSD dataset with various parameters such as the number of simulated systems, the time interval between two measurements of a single station, the duration of the generated output stream, the seed for randomization to vary the timestamps of initial measurement of each system. The oracle of YABench is an extension of the oracle proposed in CSRBench with correctness metrics for each window. YABench currently supports two engines CQELS and C-SPARQL. And finally, YABench provides a reporting web application, named as YABench reports, which displays the performance measurements in forms of graphs. Instead of defining new benchmarking queries, YABench reuses ones from CSRBench. This benchmark builds a complete workflow for evaluating such engines from defining tests, generating suitable test data, executing tests, and finally analyzing the results. However, YABench does not consider the influence of multiple windows in one query or an engine, the combination of background knowledge, as well as the reasoning support.

3.1.2.2 Reasoning Benchmarks

So far, existing benchmarks have been proposed for evaluating RSP engines, yet none of them is available to assess the reasoning capabilities of such engines in the streaming setting. Targeting in this direction, authors in [174] proposed a benchmark, called *SLUBM*, for assessing the performance of incremental reasoning engines in stream-based experiments. It is an extension of a well-known benchmark for OWL knowledge base systems, named Lehigh University Benchmark (LUBM) [175], with a temporal semantic dimension. In particular, SLUBM uses `semester` as the solo time granularity to convert static triples in the LUBM knowledge base into timestamped triples. This `semester` is

considered as a time slice in which all facts are defined semantically validated. Hence, as the time evolves, data in the previous semester are considered expired and must be removed and replaced by newer data. The predicate classes in LUBM ontology are classified heuristically into three groups: static, near-dynamic, and dynamic. Static is a label for predicates which are considered to be true for the whole timespan, while dynamic (or near dynamic) indicates ones that change constantly for every time slice (or every window of time slices). In the data generator, the temporal predicates decide the percentage of dynamic data generated in each `semester`. This benchmark is used to conduct experiments to evaluate the loading time and query execution time of 4 different engines: Jess[13], BaseVISor [176], Pellet [177], and C-SPARQL. Among these engines, only C-SPARQL is a stream processing engine while others are designed for reasoning over static data. To be able to test these systems with dynamic data, SLUBM re-designed 6 temporal inference rules for such engines by introducing a `time-to-last` which describes the expiration time of the conclusions that derive from these rules. The experiments are deployed over 14 queries defined in LUBM over a temporal dedicate `takecourse` (approximately 10% of the generated data are dynamic). The benchmark makes a part of the LUBM static dataset available to be streamed while retaining most of the LUBM's old standards. It is a first attempt in evaluating the reasoning capabilities of stream reasoners. However, the benchmark is limited in assessing systems according to its streaming supported features as well as various complexity levels of reasoning tasks.

Benchmarks, which focus on stressing the performance of ASP solvers on problems of increasing input size and complexity, have been populated in the context of the ASP competitions over the years [178]. These competitions are biannual events since 2007 with two initial goals: collecting benchmarks and creating a fair competition environment. Its format is consolidated into two main tracks: the model and solve track, and the system track. The former track aims at integrating different scientific communities by opening for all types of solvers with declarative modeling capabilities. The participant systems are compared over a variety of problem domains defined in the benchmark. The latter track focuses on fostering language standardization. The standard input language ASP-CORE-2.0 was introduced in the fourth competition in 2013. During such competitions, the benchmark has been updating with new problem domains. Each problem defined in the benchmark is provided with alternative encodings. The input data of such problems are mostly synthetic. The ASP benchmark enables assessment of different complexity levels of reasoning problems as well as optimization techniques which have

[13]http://www.jessrules.com/

been implemented in existing engines. However, this benchmark is designed for comparing such engines in static environments, which is not suitable to use in the streaming world.

3.2 Optimization via Parallel Strategy

In SR, the trade-off between expressivity and scalability is evident. Rich frameworks which encompass advance reasoning features are normally computationally intensive. The applicability of such systems is hindered by the exponential growth in the availability of streaming data on the Web. Two prominent directions for enhancing the scalability of SR reasoners are incremental and parallel approaches. Incremental approaches reduce the reasoning time of a single reasoner by avoiding many unnecessary re-computation when new data arrives, while parallel approaches take advantages of multiple concurrent reasoners to process input data faster. With this manner, parallel approaches become popular choices for scaling up the systems nowadays when the amount of data generated increase exponentially. Therefore, in this thesis, I follow the parallel approach to optimize the expressive reasoning process over SW streams. In the following, I discuss existing works related to the research question $RQ2$ which apply parallel techniques to optimize the reasoning process.

Parallel strategies were important features of database technology in the nineties in order to speed up the execution of complex queries [179]. Compared to traditional parallel processing, parallel reasoning has some additional concerns such as complex data dependencies makes the partitioning process is not easy. Two major trends are introduced to process reasoning in parallel [180]. The first one is called data partitioning approaches in which only data is split into smaller chunks and processed by several reasoners. The second trend is rule partitioning approaches in which only rules are partitioned into different reasoners to perform the reasoning tasks. These studies have also shown that effective partitioning is not easy due to the various reasoning complexities [181].

In SW, several studies how to assign a partition of the parallel computation to a set of machines have been published. As a part of the LarKC project[14], MaRVIN (Massive RDF Versatile Inference Network) [182] is a parallel and distributed platform for calculating the full closure of large amounts of RDF data. To infer through forward chaining, MaRVIN proposes the divide-conquer-swap strategy, which extends the traditional approach of divide-and-conquer with an iterative procedure whose result converges towards completeness over time. Its algorithm can be described in three steps: (first) the input data is partitioned into several independent portions with equal sizes, (second) each

[14]http://www.larkc.eu

portion is assigned to each compute node to calculate the closure using a conventional reasoner, (third) new partitions are created over the mix of old and new data. This process is repeated until no new triples are derived. In this way, MaRVIN does not require upfront data analysis. The closure is guaranteed to obtain via its divide-conquer-swap method due to the fact that MaRVIN supports only monotonic logic reasoning.

Similarly, [183] proposes a technique for materializing the closure of an RDF graph based on Hadoop framework[15]. Hadoop implements MapReduce [184] algorithm which requires all of data is encoded as a set of a pair of the form $\langle key, value \rangle$, processes them using two functions, called map and reduce, and returns new pairs as output. This work also focuses on monotonic reasoning but applies both data and rule partitioning. The rules are evaluated during the reduce function while the triples are grouped in the map function which could lead to a duplicated derivation. Moreover, the rules need to be evaluated in certain order to avoid applying same rules multiple times.

Authors in [185] also use MapReduce, but to explore the non-monotonic reasoning in the form of logic over huge datasets. They restrict this logic to the single-argument defeasible logic. Taking into account the fact that all predicates have only one argument, the map function groups together facts with the same argument value. Rules are evaluated during the reduce step for each argument value separately. This process is claimed to not alter resulting conclusions because facts with different argument values cannot produce conflicting literals and cannot be combined to reach new conclusions. Afterwards, authors extend their approach for predicates of arbitrary arity, but under the assumption of stratification [186]. The solution is extended with an idea of ranking predicates. For example, facts and predicates supported by rules containing no negative subgoals are assigned rank 0. Predicates depending negatively only on rank 0 are assigned rank 1. The reasoning is carried on in order of ranks. For each rank, reasoning is carried on until no new fact inferred.

In [187], an approach using MapReduce framework for parallel reasoning with the well-founded semantics over massive data is proposed. This work relaxes the assumption of stratified programs in [186] and allows recursion through negation. However, the programs are required to be safe and the closure calculation is based on the consecutive computation of true and unknown literals. Adapting and incorporating the computation of joins and anti-join described in [186], authors proposed the alternating fixpoint procedure to avoid full materialization of Herbrand base for any predicate. While the works in [182, 183] focus on monotonic reasoning, [185–187] examine non-monotonic reasoning over massive data. However, these attempts do not consider the streaming setting and

[15] http://hadoop.apache.org

are required more effort be able to apply on ASP due to the complexity of its stable model semantics.

In ASP, several works about parallel techniques for the evaluation of a logic program have been proposed, focusing on both phases of the ASP computation, namely grounding and solving. The recent survey on the main techniques and approaches in the literature to improve performance through the exploitation of parallelism in the execution of ASP solvers can be found in [188]. The authors explored approaches according to two orthogonal dimensions: (first) the different levels of complexity and features of the underlying language, (second) the different levels of granularity of exploitation of parallelism.

Concerning the parallelization of the grounding phase, a first investigation is the work in [189]. Authors exploited the restrictedness and the presence of domain predicates in LPARSE programs to statically partition the program rules. The distributed implementation of the grounder LPARSE is organized as a master-slave structure in which the master partitions the program and delegates the grounding of each component to different slaves. Load balancing can be heuristically controlled by assigning weights to rules. In fact, such weight is an estimate of the number of expected ground instances of each rule. However, this work is applicable only to a subset of the program rules. Therefore, in general, this work is unable to exploit parallelism fruitfully in the case of programs with a small number of rules.

F. Calimeri et al. [2] studied the parallelization of the DLV grounder by exploring some structural properties of the input program via the defined dependency graph in order to detect subprograms that can be evaluated in parallel. The study adopts a symmetric multi-processing architecture to enable communication of concurrent threads through a share memory. [57] extends this work with parallelism in three different levels in the grounding process: components, rules, and single rule level. The first level (components level) divides the input program into subprograms (or modules), according to the strongly connected components of the dependency graph among IDB predicates of that program. The second level (rules level) allows for concurrently evaluating the rules within each subprogram. The semi-naive algorithm of Datalog is used to evaluate a fixpoint when grounding recursive rules. The third level (single rule level) partitions the extension of a single rule literal into a number of subsets. This step is especially efficient when the input program consists of few rules and two first levels have no effects on the evaluation of the program. For each level, the authors proposed different techniques to take advantage of the underlying multi-threaded system.

For the solving step which is carried out after the grounding step, the computation can be visualized as the construction of a binary search tree. Hence, the parallelism can be enabled by distributing the construction of different branches of the search tree between

different processors. This idea for exploiting parallel in the solving phase was originally presented in [190, 191]. Later, authors in [78] proposed a generic approach to distribute the searching space in order to find the answer sets, which permits exploitation of the increasing availability of clustered and/or multiprocessor machines.

Dynamic distribution of the search space is vital to achieving adequate parallel performance. Two components have a great impact in the execution is task sharing (i.e., how to move a processor to a different part of the search tree) and scheduling (i.e., how to locate which part of the subtree one processor should explore next). Authors in [192] provided an analysis of distinct strategies for these two components. [193] enriched this investigation and obtained as variations of the copying and the re-computation schemes.

Authors in [189] explored parallelism for the ASP solving phase in two dimensions. The vertical one indicates a situation where separated threads of computation are employed to explore alternative literals to add to the partial answer set. The horizontal dimension indicates the use of separate threads of computation to concurrently apply different expansion techniques to a partial answer set. The first dimension is enabled by implementation of the copy-based task sharing and re-computation-based task sharing approaches. The former allows agents to share work by exchanging a complete copy of the current answer set while the latter allows agents to exchange the list of chosen literals which have been used in the construction of an answer set. The second dimension is achieved via the lookahead technique. Author deployed this technique from the field of satisfiability testing to enrich the typical algorithms for the computation of answer sets in the parallel manner. This parallelization is obtained by partitioning the set of unexplored literals and assigning each subset to a different agent. After agents process the test on these unexplored literals, their results are merged to create a new partial answer set.

A multi-thread version of the state-of-the-art answer set solver Clasp is presented in [121]. The idea in this work is to transfer the entire functionality from the sequential to a parallel setting. When the logic program is read in Clasp , the preprocessing stages, conducted by the main thread, identify redundant variables. The outcomes of the preprocessing phase are stored in an object which can be shared among all participating solving threads. Each solving thread performs propagation by implementing conflict-driven search involving nogood (i.e., set of literals that must not jointly be assigned) learning. However, the solver may learn exponentially many nogoods. To solve this issue, the authors pursued a hybrid approach by separate distribution and integration components for carefully selecting the spread constraints.

In recent years, the use of Graphical Processing Units (GPUs) has become pervasive in general-purpose applications that are not directly related to computer graphics, but

demand massive computational power. Dovier Agostino et al. [194, 195] are interested in devising the power of GPUs to enhance the parallelization of ASP solving. The approach is built on the notion of ASP computation combined with conflict-driven analysis and learning techniques derived from those adopted in [196]. All the solving components (i.e., nogoods management, search strategy, (non-chronological) backjumping, heuristics, conflict analysis and learning, and unit propagation) are performed on the GPUs by exploiting Single-Instruction Multiple-Thread parallelism. The CPU is used only to preprocess the program and to output the results.

3.3 Summary

This chapter analyzed the state-of-the-art for continuous querying and reasoning over data streams, especially semantically annotated streams. Different approaches in SR including languages, engines, benchmarks, and optimization were discussed relative to the set of research questions defined in this thesis. This analysis supports the identification of the main gaps in the literature such as: (i) need for a higher expressive language to capture more sophisticated what users want (*RQ1*), and (ii) investigation of scalable techniques for integrating more complex reasoning capabilities in streaming setting (*RQ2*). The contribution of this thesis concentrates on the proposal of an ASP-based approach for stream reasoning which leverages RDF stream processing and parallel optimization to bridge these gaps.

Chapter 4

C-ASP: Continuous Extension of ASP for RDF Stream Reasoning

Massive amounts of data are being generated every minute every day from a huge number of devices and services across the Internet. Various application scenarios in several domains, such as IoT and Smart Cities, benefited from the ability to efficiently integrate and query data streams from these sources, in some cases providing basic reasoning functionalities [10]. In recent years, SW research has contributed to advancing the state-of-the-art in semantic stream processing, in particular through the extension of RDF, a standard model for data interchange on the Web, to represent and manipulate RDF streams [17]. As a result, extensions of SPARQL, a query language for RDF data, have been proposed to continuously query RDF data streams. These efforts have resulted in the implementation of several engines for RSP, such as C-SPARQL [19] and CQELS [20], among others. Such engines are able to deal with the transient nature of data streams mostly based on pattern matching techniques, and in some cases they have limited reasoning capabilities. For example, C-SPARQL [1] supports for RDFS entailment regimes. Despite all these efforts, reasoning capabilities are still limited and cannot support complex reasoning such as the ability to handle defaults, common sense, preferences, recursion, and non-determinism. When it comes to modern real-world applications, the ability to handle incomplete and potentially inconsistent input streams, and extract knowledge from them to support decision making is not fully supported yet [28].

Some work in this direction leverages the expressive power of logic-based non-monotonic reasoning techniques to build a SR system, relying on advances in semantic stream processing technologies for representing and processing data streams on the one hand,

[1] http://streamreasoning.org/resources/c-sparql

and non-monotonic reasoning approaches for performing complex rule-based inference on the other hand. This combination is based on the principle of having a 2-tier approach where: i) a semantic stream query processor is used to filter RDF triples, and ii) a non-monotonic reasoner is used for computationally intensive tasks over the filtered data. Since the grounding phase in rule-based inference is responsible for the size of the program to be evaluated, such a combined approach improves the scalability of complex reasoning over SW streams by reducing the input to the non-monotonic reasoner. Current expressive reasoning systems over RDF data streams, like EP-SPARQL [36], ASR [53], and StreamRule [54], support non-monotonic reasoning over data streams in different ways. For instance, EP-SPARQL extends the SPARQL querying language for event processing. EP-SPARQL queries are translated into logic expressions and are evaluated continuously by ETALIS [156] which is implemented based on SWI-Prolog[2]. ASR [53] and StreamRule [54] support more complex reasoning capabilities based on ASP by using the DLVhex solver [163] and the Clingo solver [48], respectively. These systems rely on ASP by hard-coding a subprocess that performs repetitive calls to the ASP solver to infer new knowledge from data streams and a given rule set. Therefore, they do not provide a flexible way to seamlessly integrate the stream processing and reasoning functionalities.

Recently, two new SR engines, namely Ticker [147] and Laser [148], have been introduced. These two engines implement a fragment of LARS [160]. programs are encoded as ASP rules. However, Laser restricts its expressivity to positive and stratified programs and ASP encodings of Ticker's programs use mainly normal ASP rules. As a result, they do not fully exploit the expressive power of ASP, including the ability to handle disjunction, optimization, aggregations, and preferences. Moreover, they do not support the ability to query SW streams, which we believe is the key in ensuring the scalability of such systems when handling real IoT streams.

Aiming to i) exploit full capabilities of ASP to perform reasoning over RDF streams, and ii) seamlessly provide flexible ways of combining semantic stream processing and non-monotonic reasoning, this chapter mainly addresses the first research question *RQ1* which states as follows:

> *RQ1.* How to enable complex reasoning based on ASP over RDF data streams?

The chapter tests the hypothesis *H1* that is formulated below:

[2]http://www.swi-prolog.org

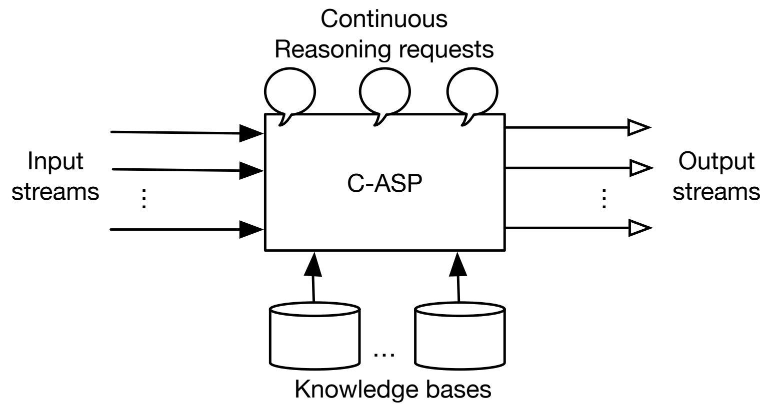


FIGURE 4.1: The C-ASP Processing Model

> *H1.* The extension of the ASP language within the RDF streaming setting provides a higher expressivity to capture users' sophisticated requirements and preferences.

To answer the research question *RQ1* and to test the hypothesis *H1*, this chapter proposes *C-ASP*, an extension of ASP to support continuous reasoning over semantic streams. The C-ASP language allows users to specify their reasoning requests which include their sophisticated reasoning requirements, input streams, and how to access those streams. Such requests are registered with the C-ASP engine and continuously executed over RDF streams by the means of windows. The chapter starts with the processing model and key definitions in Section 4.1. In Section 4.2, I define the syntax and semantics of a reasoning request in the C-ASP language as well as a set of examples in order to illustrate its expressive power. Section 4.3 provides an evaluation of the proposed approach. The chapter is summarized in Section 4.4.

4.1 The C-ASP Processing Model

The processing model of C-ASP combines RSP and ASP-based reasoning in one single framework. It defines RDF streams by means of windows and reasoning requests to be evaluated over RDF streams and RDF static datasets, as illustrated in Figure 4.1.

Similarly to other RSP engines, C-ASP takes multiple RDF data streams as inputs and produces outputs as streams. It also supports the integration with background knowledge from static RDF knowledge bases.

Definition 4.1. An RDF stream S is defined as an ordered sequence of pairs:

$$S = ..., (d_i, t_i), (d_{i+1}, t_{i+1}), ...$$

where each pair is made of an RDF triple d_i and its timestamp t_i. Timestamps are monotonically non-decreasing ($t_i \leq t_{i+1}$).

Users express their requirements and preferences in the form of *continuous reasoning requests* using the C-ASP language, an extension of the ASP language described in Section 4.2. Reasoning requests are registered with the C-ASP engine and evaluated continuously over the input streams and the knowledge bases. The continuous reasoning model of C-ASP takes as input RDF streams, evaluates a continuous reasoning request RR, and produces output streams. When evaluating RR at time t, only a portion of the input stream is considered in the evaluation of the request. It is to be noticed that the C-ASP processing model combines models developed for DSMS based on windowing, and models for CEP systems based on rules applied to one-timestamped data items to produce new events from input data.

The process can be broken down into 3 steps:

1. Windowing (from streams to relations): select subsets of the most recent elements of the input streams.
2. Evaluating (from relations to relations): perform reasoning on the finite and intermediate data portions.
3. Streaming (from relations to streams): convert the final solutions back into streams.

4.1.1 Windowing: from streams to relations

I now introduce the concept of *windows* over a stream in the windowing step.

Definition 4.2. Given a stream S as in Definition 4.1, a *window* W of S at time t can be defined as

$$W(S,t) = \{(d,t') \in S : t' \leq t\}$$

To select elements from a stream S, C-ASP supports two types of windows, namely *time-based window* and *count-based window*. Each type takes a *window parameter* $\omega \in \mathbb{N}$ to define the selection of elements from the stream.

Definition 4.3. Given a stream S as in Definition 4.1, a *time-based window* over S at time t can be defined as

$$W_T^{\omega}(S,t) = \{(d,t') \in S \;:\; t' \in [max\{0, t-\omega\}, t]\}$$

Definition 4.4. Given a stream S as in Definition 4.1, a *count-based window* over S at time t can be defined as $W_C^{\omega}(S,t) \subseteq W(S,t)$ and satisfies two following conditions:

$$|W_C^{\omega}(S,t)| = \omega, \textit{ and}$$
$$\forall (d',t') \in W_C^{\omega}(S,t), \nexists (d'',t'') \in S \setminus W_C^{\omega}(S,t) \textit{ such that } t' \leq t'' \leq t$$

In order to process the continuous stream, windows use an additional parameter β indicating the distance between two consecutive windows, referred to as *slide*. Intuitively, a *time-based sliding window* on stream S, $W_T^{\omega,\beta}(S,t)$, continuously produces a window $W_T^{\omega}(S,t)$ every β units of time. Similarly, a *count-based sliding window* on the stream S, $W_C^{\omega,\beta}(S,t)$, continuously produces a window $W_C^{\omega}(S,t)$ containing exactly ω (timestamped) triples, by removing the oldest and adding the newest β triples for each new window. Note that if the number of triples with the same timestamp removed (resp. added) makes the cardinality of the window smaller (resp. bigger) than ω then data items to be removed (resp. added) are chosen randomly.

I will illustrate how the windows work with a simple example.

Example 4.1. *A travel company records the bookings of users to hotels that are available on its website as an RDF stream $S = (d_1,2), (d_2,8), (d_3,8), (d_4,11), (d_5,15), \ldots$ as in Figure 4.2. Assume that the unit of time is minute and the stream starts producing triples at time $t = 0$.*

- *The time-based sliding window $W_T^{7,3}(S,t)$ starts producing a window at time $t = 7$ and continuously produces new windows every 3 minutes as follows:*

$$\begin{aligned} W_T^{7,3}(S,7) &= \{(d_1,2)\}, \\ W_T^{7,3}(S,10) &= \{(d_2,8),(d_3,8)\}, \\ W_T^{7,3}(S,13) &= \{(d_2,8),(d_3,8),(d_4,11)\}, \ldots \end{aligned} \tag{4.1}$$

- *The count-based sliding window $W_C^{3,1}(S,t)$ starts producing a window when $|W_C^3(S,t)| = 3$ and continuously produces a new window whenever a new triple arrives, by removing the oldest triple and adding the new one as follows:*

$$\begin{aligned} W_C^{3,1}(S,8) &= \{(d_1,2),(d_2,8),(d_3,8)\}, \\ W_C^{3,1}(S,11) &= \{(d_2,8),(d_3,8),(d_4,11)\}, \\ W_C^{3,1}(S,15) &= \{(d_2,8),(d_4,11),(d_5,15)\}^{3}, .. \end{aligned} \tag{4.2}$$

[3] $(d_3,8)$ is removed randomly.

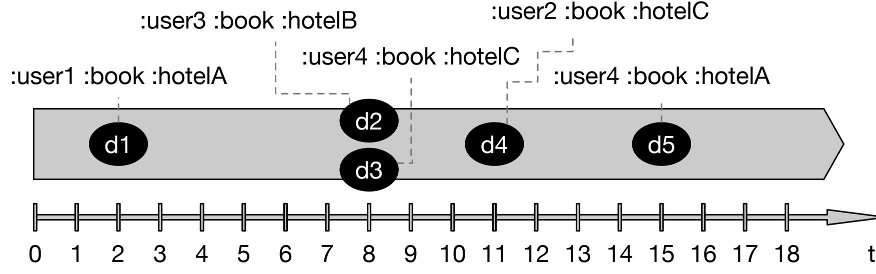


FIGURE 4.2: An RDF stream of booking hotels

4.1.2 Evaluating: from relations to relations

After selecting the relevant portion of the input stream as a set of discrete elements (the window), the second step of the C-ASP processing model evaluates the reasoning request which follows the stable model semantics of ASP. In what follows we define a C-ASP reasoning request (Definition 4.5 and its semantics in Definition 4.6 by resorting to the stable model semantics of ASP).

Definition 4.5. A C-ASP reasoning request RR is defined as

$$RR = (Pre, I, P, O)$$

where Pre defines a set of constants indicating namespace prefixes[4]; $I = I_{stream} \cup I_{kb}$, where $I_{stream} = \{(S_i, W(S_i,t)),\ i = 1..n\}$ identifies a set of RDF input streams and the windows to specify how to extract data elements from streams, and $I_{kb} = \{kb_j, j = 1..m\}$ identifies a set of RDF static datasets; P identifies a set of ASP rules (or a logic program); and $O \subset pre(P)$ identifies a selections of output data ($pre(P)$ is a set of predicate symbols appearing in P).

Intuitively, an evaluation of a reasoning request $RR = (Pre, I, P, O)$ at time t generates a solution of RR with respect to the windows generated in I at that time point.

Definition 4.6. Given a C-ASP reasoning request $RR = (Pre, I, P, O)$. A solution of RR at time t, $ans(RR, t)$ is defined as:

$$ans(RR, t) = \varphi_O(ans(\bigcup_{i=1}^{n} W(S_i, t), \bigcup_{j=1}^{m} kb_j, P))$$

$ans(RR, t)$ is computed in two steps. First, I compute an answer set of the ASP rules in P and the facts in both the static knowledge bases ($\bigcup_{j=1}^{m} kb_j$) and the windows produced

[4]As illustrated in Section 4.2, these refer to IRIs to identify RDF streams sources.

from the input streams at time t ($\bigcup_{i=1}^{n} W(S_i, t)$). Next, I select the output data items from the answer set based on the predicates defined in O as follows:

$$\varphi_O(ans(\bigcup_{i=1}^{n} W(S_i, t), \bigcup_{j=1}^{m} kb_j, P)) = \{p(.) \in ans(\bigcup_{i=1}^{n} W(S_i, t), \bigcup_{j=1}^{m} kb_j, P) : p \in O\}$$

As the nonmonotonic semantics of ASP comes with possibly multiple answer sets, C-ASP provides possibly multiple solutions for the reasoning request RR which denoted as $Ans(RR, t) = \{ans(RR, t)\}$.

4.1.3 Streaming: from relations to streams

I now need to serialize the output of the reasoning request to generate the output stream. To generate streams as output, C-ASP needs a streaming operator to assign the correct evaluation timestamp to data items derived by the evaluation step. In order to do this, I adapt the *RStream* operator defined for relational data stream processing in [4] as defined below.

Definition 4.7. Given an answer set $ans(RR, t)$ of the C-ASP reasoning request RR at time t. *RStream* is defined as follows:

$$RStream(ans(RR, t)) = \{(d, t) : d \in ans(RR, t)\}$$

Similarly, $RStream(Ans(RR, t)) = \{RStream(ans(RR, t)) : ans(RR, t) \in Ans(RR, t)\}$.

In order to generate RDF streams as output, a predicate symbol p in O has to appear in a predicate of the form $p(s, o)$. In this way, the *RStream* operator is modified as follows:

$$RStream(ans(RR, t)) = \{(< s, p, o >, t) : p(s, o) \in ans(RR, t)\}$$

4.2 Implementation: the C-ASP Language

Based on the C-ASP processing model defined Section 4.1, I now define how a C-ASP reasoning request is expressed in an extension of the ASP language with RDF streaming features, and I provide a set of examples.

4.2.1 C-ASP Reasoning Request

Consider a C-ASP reasoning request $RR = (Pre, I, P, O)$ as defined in Definition 4.6. In Figure 4.3, I provide the syntax to express each component in RR. First, to deal with the RDF data format, I use a `PrefixClause` statement which captures each element in Pre. This `PrefixClause` is adopted from the syntax for prefixes used to abbreviate IRIs[5] in SW.

```
PrefixClause → #prefix prefixName : ⟨IRI⟩;

FromStreamClause → #from stream streamIRI Window;
   Window → Time-basedWindow | Tuple-basedWindow
      Time-basedWindow → [time number TimeUnit step number TimeUnit]
         TimeUnit → d | h | m | s | ms
      Tuple-basedWindow → [count number step number]

FromClause → #from ⟨knowledge base⟩;

RuleClause → ASP rule;

OutputClause → #show predicateSymbol/number;
```

FIGURE 4.3: C-ASP syntax

The identifications of input streams and static knowledge bases in I are expressed by means of `FromStreamClause` and `FromClause`, respectively. In `FromStreamClause`, each input stream is coupled with a window (represented by `Window`) to guide the C-ASP engine on how to extract related data from the stream. In `FromClause`, static knowledge bases are specified via their paths and the C-ASP engine integrates them with input streams before performing ASP-based reasoning over them.

A rule in P follows the ASP-core2 language standard[6] and the C-ASP implementation relies on the Clingo solver[7]. However, the extension of ASP rules to deal with RDF streams introduces predicate symbols, which are obtained from converting an RDF triple $\langle s, p, o\rangle$, in form of `prefixName_p`. The predicate symbol `prefixName_p` identifies that this predicate is from RDF input streams or RDF datasets while p (without `prefixName`) is an internal predicate defined and used within the ASP rules. In this way, an input RDF triple with a timestamp $(\langle s, p, o\rangle, t)$ is automatically converted into an ASP predicate of the form `prefixName_p`(s, o, t).

[5] https://www.w3.org/TR/rdf11-concepts/#section-IRIs
[6] https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf
[7] http://potassco.sourceforge.net

```
#prefix tl : ⟨http://travel.org/⟩;

#from stream <http://travel.org/booking> [time 1h step 30m];

bookedHotel(Hotel) :- tl_booked(User,Hotel,Time);
noH(N) :- N = #count{Hotel:bookedHotel(Hotel)};

#show noH/1;
```

Listing 4.1: RR1

```
#prefix tl : ⟨http://travel.org/⟩;
#prefix ht : ⟨http://hotel.org/⟩;
#prefix ct : ⟨http://city.org/⟩;
#prefix rdf : ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩;
#prefix ex : ⟨http://example.org/⟩;

#from stream <http://travel.org/booking> [time 1h step 30m];
#from <hotelkb.rdf>;
#from <streetkb.rdf>;

bookedHotel(Hotel):-tl_booked(User,Hotel,Time);
fiveStar(Hotel):-bookedHotel(Hotel), ht_star(Hotel,Star),
                 Star = 5;
ex_fmHotel(Hotel,Street):-fiveStar(Hotel),
                          ht_located(Hotel,Street),
                          rdf_type(Street,"ct_MainStreet");

#show ex_fmHotel/2;
```

Listing 4.2: RR2

In addition, output statements identify output predicates the C-ASP engine needs to provide after reasoning. The syntax of an output statement in O is defined in `OutputClause`. The variable `number` in `OutputClause` identifies the number of arguments in `predicateSymbol`. If `number` = 2 then the C-ASP engine provides output as (timestamped) RDF streams by converting output atoms (i.e., `predicateSymbol(s,o)`) to triples (i.e., `<s,predicateSymbol,o>`) and assigning timepstamps to them with the *RStream* operator (i.e., `(<s,predicateSymbol,o>, t)`). Otherwise, C-ASP outputs (timestamped) predicate-format streams.

4.2.2 Examples of a C-ASP Reasoning Request

In what follows, I present some of the features of the C-ASP language defined above by providing examples of continuous reasoning requests. I refer to the hotel booking scenario introduced in Example 4.1.

```
#prefix tl : ⟨http://travel.org/⟩;
#prefix ht : ⟨http://hotel.org/⟩;
#prefix ct : ⟨http://city.org/⟩;
#prefix rdf : ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩;
#prefix ex : ⟨http://example.org/⟩;

#from stream <http://travel.org/booking> [time 1h step 30m];
#from stream <http://travel.org/canceling> [time 1h step 30m];
#from <hotelkb.rdf>;
#from <streetkb.rdf>;

bcHotel(Hotel):-tl_booked(User,Hotel,Time1),
                tl_canceled(User,Hotel,Time2),
                Time1<Time2;
bcStar(Hotel):-bcHotel(Hotel), ht_star(Hotel,Star),
                Star <= 3;
hasStar(Hotel):-ht_star(Hotel,Star);
bcStar(Hotel):-bcHotel(Hotel), not hasStar(Hotel,Star);
ex_bcmHotel(Hotel,Street):-bcStar(Hotel),
                           ht_located(Hotel,Street),
                           rdf_type(Street,"ct_MainStreet");

#show ex_bcmHotel/2;
```

Listing 4.3: RR3

RR1, illustrated in Listing 4.1, is a simple C-ASP reasoning request with aggregation. The request is made by the travel company in order to know how many hotels have been booked during the last hour. It will notify the company every 30 minutes.

Assume that the information of hotels and streets are stored in static RDF datasets, `hotelkb.rdf` and `streetkb.rdf` respectively. The company wants to know which 5-star hotels located on a main street have been booked in the last hour. The reasoning request *RR2* in Listing 4.2 shows the combination of static and streaming data.

To illustrate an example of combining multiple input streams, I assume that the company also records information when a user cancels a booking. The reasoning request *RR3* (Listing 4.3) notifies the company on which below-3-star hotels located on a main street have been booked and then canceled during the last hour. This request also allows the company to decide how to deal with incomplete information about hotels' stars via the negation-as-failure rule `bcStar(Hotel):- bcHotel(Hotel), not hasStar(Hotel);`.

I now showcase an example of C-ASP that can capture more sophisticated requirements via optimization statements in ASP. Imagine that the company wants to know the most expensive and highest star hotels that have been booked during the last hour. They do not want to get notification of those hotels located in a noisy area (I assume that a hotel located on a main street is noisy). Moreover, they are more interested in the most expensive hotels. *RR4* expresses such request as illustrated in Listing 4.4. This

```
#prefix tl : ⟨http://travel.org/⟩;
#prefix ht : ⟨http://hotel.org/⟩;
#prefix ct : ⟨http://city.org/⟩;
#prefix rdf : ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩;

#from stream <http://travel.org/booking> [time 1h step 30m];
#from <hotelkb.rdf>;
#from <streetkb.rdf>;

1{bookedHotel(Hotel):tl_booked(User,Hotel,Time)}1;
noisyHotel(Hotel):-bookedHotel(Hotel),
                   ht_located(Hotel,Street),
                   rdf_type(Street,"ct_MainStreet");
:- noisyHotel(Hotel);
#maximize {Y@1 : ht_star(Hotel,Star), bookedHotel((Hotel)};
#maximize {Y@2 : ht_cost(Hotel,Cost), bookedHotel(Hotel)};

#show bookedHotel/1;
```

Listing 4.4: RR4

```
PREFIX ct : <http://www.insight-centre.org/citytraffic#>
PREFIX ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX sao : <http://purl.oclc.org/NET/sao/>
PREFIX  : <http://www.insight-centre.org/dataset/SampleEventService#>

CONSTRUCT {?obId1 sao:hasValue ?v1. ?obId2 sao:hasValue ?v2.}
FROM STREAM <.../TrafficData182955> [range 3000ms step 1s]
FROM STREAM <.../TrafficData158505> [range 5000ms step 4s]
FROM <http://localhost:8080/SensorRepository.rdf>

WHERE {
?p1   a  ct:CongestionLevel.
?p2   a ct:CongestionLevel.
?obId1 ssn:observedProperty ?p1.
?obId1 sao:hasValue ?v1.
?obId1 ssn:observedBy :AarhusTrafficData182955.
?obId2 ssn:observedProperty ?p2.
?obId2 sao:hasValue ?v2.
?obId2 ssn:observedBy :AarhusTrafficData158505.
}
```

Listing 4.5: Snapshot of query Q1C

request takes advantage of the ability of ASP to handle expressive reasoning such that managing optimization statements.

4.3 Evaluation

In this section, I compare the performance of the C-ASP reasoner against one of the most mature RSP engines, C-SPARQL, with respect to two metrics, latency and memory

consumption, by using the well-known stream processing benchmark CityBench [172]. Another RSP engine that is still maintained is CQELS, but I do not consider CQELS because its processing mode does not allow certain positive rules to be expressed: both C-ASP and C-SPARQL process streaming data in batches while CQELS processes every new data item immediately and therefore cannot reason about elements appearing in the same window. Laser and Ticker are not considered because they do not support RDF streams and RDF datasets.

Latency refers to the time consumed by the engine between the input arrival and the output generation while memory consumption reflects the usage of the system memory during the execution. The experiment was conducted on a machine with 24-core Intel(R) Xeon(R) 2.40 GHz and 96G RAM. I used Java 1.8 with heap size from 5GB to 20GB for C-SPARQL and Clingo 4.5.4 for C-ASP. The empirical results are encouraging as they show that C-ASP outperforms C-SPARQL in both latency and memory consumption. I aim at showing that C-ASP has superior performance when dealing with the same expressivity as C-SPARQL with increasing stream frequency. One of possible reason for the difference in the performance of these two engines is the difference of their implementation. C-ASP delegates the reasoning task to Clingo which implements multiple optimization techniques to speed up the grounding step and the solving step (see Section 2.2.4) while C-SPARQL delegates the query evaluation to Esper and Jena. In this way, C-SPARQL limits to apply query rewriting techniques to make its query processor dynamically adapting to the changes in the input. Programming languages may contribute on this performance difference. C-ASP uses Clingo which written in C++ while C-SPARQL is written fully in Java.

To make sure that both engines return the same result format (triples) for a fair comparison, I modify the SELECT statements of C-SPARQL queries in CityBench to the CONSTRUCT statements. I translate C-SPARQL CONSTRUCT queries into C-ASP reasoning requests. In particular I use queries Q1C, Q2C, and Q10C as representative samples in terms of the number of query patterns and the presence of join operators. Listing 4.5 and 4.6 show snapshots of the C-SPARQL query Q1C and the corresponding C-ASP reasoning request R1C.

First, I evaluate the performance of the two engines with a frequency $f = 1$ (i.e., replay streams at the original rate). I stream data for 10 minutes to C-SPARQL queries Q1C, Q2C, and Q10C (respectively, R1C, R2C, and R10C for C-ASP reasoning requests) and measure the latency and memory consumption every minute. Results shown in Figures 4.4, 4.6, and 4.8 indicate that the latency of C-ASP is minimal compared to C-SPARQL for all three queries. More specifically, C-ASP performs almost 3 times (or more) faster than C-SPARQL for queries Q1C, Q2C and slightly faster for query Q10C. In addition,

```
#prefix ct : ⟨http://www.insight-centre.org/citytraffic#⟩;
#prefix ssn : ⟨http://purl.oclc.org/NET/ssnx/ssn#⟩;
#prefix sao : ⟨http://purl.oclc.org/NET/sao/⟩;
#prefix  : ⟨http://www.insight-centre.org/dataset/SampleEventService#⟩;
#prefix rdf : ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩;

#from stream  <.../TrafficData182955> [time 3s step 1s];
#from stream  <.../TrafficData158505> [time 5s step 4s];
#from <dataset/SensorRepository.rdf>;

observerBy(ObId):-  ssn_observedBy(ObId, "_TrafficData182955");
observerBy(ObId):-  ssn_observedBy(ObId, "_TrafficData158505");
result(ObId, V):-  observerBy(ObId), sao_hasValue(ObId, V)
                   ssn_observedProperty(ObId, P),
                   rdf_type(P, "ct_CongestionLevel");

#show result/2;
```

Listing 4.6: Snapshot of reasoning request R1C

it is noticeable in those figures that the memory consumption of C-ASP is less than a half of the C-SPARQL memory consumption.

I then increase the frequency of streams to $f = 2$ and re-run the experiment with the similar setting. Figures 4.5, 4.7, and 4.9 show that C-ASP still outperforms C-SPARQL. In details, the latency of C-SPARQL increases slightly from $f = 1$ to $f = 2$ for queries Q1C and Q10C, and increases sharply for query Q2C (around 2 times). In contrast, C-ASP maintains the same latency for both $f = 1$ and $f = 2$. Those figures also show that when $f = 2$, the memory consumption of both engines remains stable as in $f = 1$ and C-ASP consumes half less memory than C-SPARQL.

4.4 Summary

This chapter presents C-ASP, a ASP-based approach for performing complex reasoning over RDF streams. More precisely, C-ASP enables the continuous reasoning capability on ASP by adding: RDF streams to the data types, windowing operators to capture the most relevant portions of data from streams, (stable model semantics) entailment at window-level, and streaming operators to stream out the results. The conceptual architecture of the C-ASP reasoner is presented together with its continuous reasoning model. This model supports the C-ASP language which extended from the ASP language and RDF stream features. This language, which leverages full expressive power of ASP, allows users to express their requirements and preferences in form of C-ASP

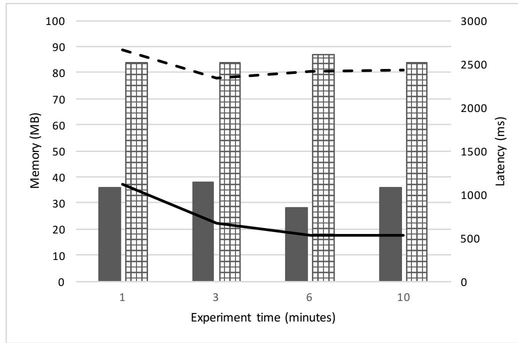


Figure 4.4: Q1C & R1C (f=1)

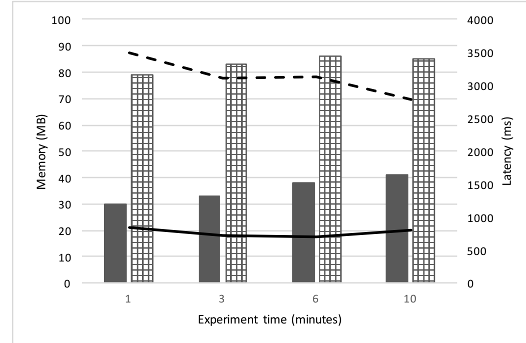


Figure 4.5: Q1C & R1C (f=2)

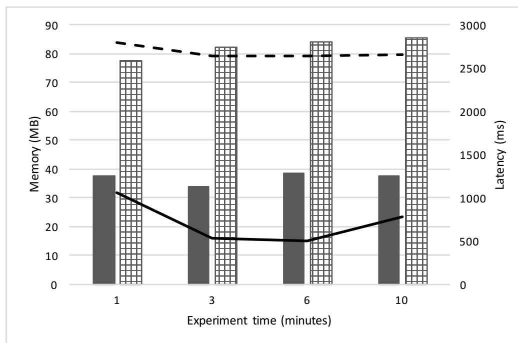


Figure 4.6: Q2C & R2C (f=1)

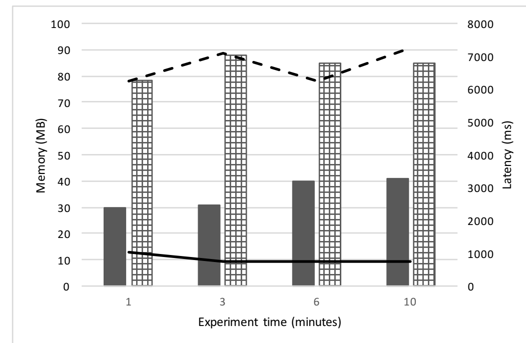


Figure 4.7: Q2C & R2C (f=2)

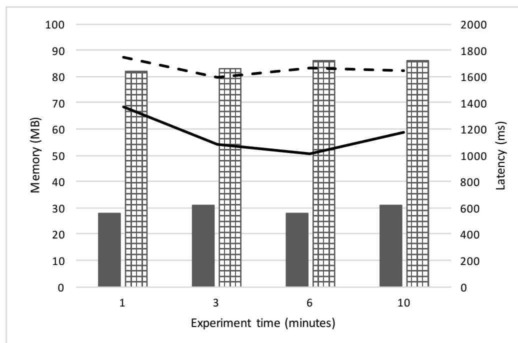


Figure 4.8: Q10C & R10C (f=1)

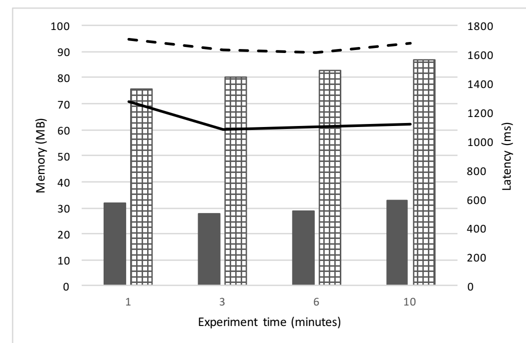
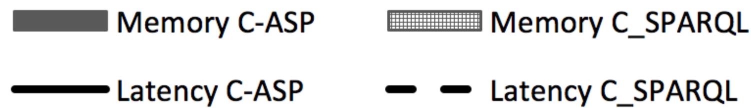


Figure 4.9: Q10C & R10C (f=2)



reasoning requests. Such requests are registered once at the C-ASP engine and continuously evaluated when data arrives. The experimental evaluation shows that the C-ASP engine outperforms the state-of-the-art RSP engine C-SPARQL.

Chapter 5

Characterizing Input-driven Dependency

The highly declarative power of ASP opens up the capabilities of the C-ASP reasoner to capture more sophisticated requirements of users and to solve more complex tasks over diverse SW streams. However, computing solutions of such reasoning tasks under stable model semantics of ASP is quite costly, especially in the dynamic environment. As defined in Chapter 1, the second research question *RQ2* in this thesis is concerned about enhancing the scalability of the proposed C-ASP stream reasoner. This question is composed of two sub-questions. In this chapter, I mainly tackle the first sub-question of *RQ2* (*RQ2.1*). The second sub-question of *RQ2* (*RQ2.2*) is answered in Chapter 6. The research question *RQ2.1* states as follows:

> *RQ2.1.* What information potentially affects the reasoning process of an expressive stream reasoner?

The chapter tests the hypothesis *H2* and the first half of the hypothesis *H3* that are formulated below:

> *H2.* Partitioning an input window in which data is independent helps to reduce the reasoning cost of an expressive stream reasoner. Moreover, the total time when reasoning sequentially over such partitions of the input window can be smaller than the reasoning time over the whole window under the circumstance of monotonically increasing reasoning time.
>
> *H3.* The semantic dependencies between input data streams plays an important role in the reasoning process in terms of its performance and the

correctness of results. These dependencies can be captured based on the structure of a given reasoning request and can be decomposed in such a way to enable parallel reasoning and maintain the correctness of results.

To answer the research question *RQ2.1* and to test the hypotheses *H2, H3*, the key features that potentially affect the scalability of the C-ASP reasoner are identified. The chapter is formed in two parts. In the first part, the correlation between key features and their impact on the reasoner's performance are empirically evaluated under the assumption of independent input data (*H2*), presented in Section 5.1. This work is in collaboration with Stefano Germano at University of Calabria and is published in [197], and also in his PhD thesis "Logic Programming in non-conventional environments"[1].

Later, this assumption is relaxed and dependencies among input data are analyzed for a further investigation on their significant impact on the reasoning performance in a streaming scenario (*H3*), which is detailed in the second part of the chapter. This second part starts with a running example introduced in Section 5.2.1 in order to illustrate the importance of dependencies among data. Section 5.2.2 presents the assumptions applied for this section and the following chapter. A formal characterization for analyzing dependencies among input data based on the structure of a given reasoning request is then proposed in Section 5.2.3. Section 5.2.4 presents algorithms which characterize different relationships between two predicates appearing in the input data in the form of so-called input dependency graph. The chapter is summarized in Section 5.3.

5.1 Reasoning over Independent Data Streams

This section provides a preliminary analysis on how the scalability of the C-ASP reasoner can be improved under the assumption that data elements in an input window are independent. This reasoner processes data streams by replying on an ASP solver in which the grounding phase and the solving phase are strongly affected by the amount of input data. Hence, the performance of the C-ASP reasoner is affected by not only the size of an input window but also the frequency of windows sending to it. In order to maintain the stability, the C-ASP reasoner needs to keep up with the new coming windows. In this way, it has to reason and return results faster than the next window arrives for preventing the time delay accumulation. Assume that the size of the input window W is fixed, and a new window is sent to the C-ASP reasoner every unit of time U, the question is "Can W be processed in less than U of time by reasoning sequentially

[1] https://www.mat.unical.it/phd/Alumni?action=AttachFile&do=get&target=GermanoAbstract.pdf

over partitions of W?". Formally, the question can be stated as: *Is there a window* W' *such that:* $|W'| < |W|$ *and* $T(W, W') \leq U$*?*

$T(W, W')$ notates the time needed by the reasoner to process W by reasoning sequentially over same-size partitions W' of W. The number of W' is $\lceil \frac{|W|}{|W'|} \rceil$ and $T(W, W') = \lceil \frac{|W|}{|W'|} \rceil \times T(W')$, in which $T(W')$ notates the time needed by the reasoner to process W' in one computation.

In the following, I answer the above question by demonstrating how the window size and the streaming frequency affects the reasoning process via an empirical evaluation on the C-ASP reasoner based on the travel planner scenario. I present below the scenario, data streams, and the reasoning request for the experiment and discuss findings.

5.1.1 Experiment Setting

Scenario. Alice is moving on a street. She wants to know real-time events that affect her travel plan to react accordingly. She needs an application built on the top of the C-ASP reasoner to notify events which are considered to be critical to her current situation. The C-ASP reasoner receives event streams that indicate changes in the real world (such as accidents, road traffic, flooding, road diversions and so on) and updates on the user's current status (such as Alice's current location and activity). With this information as input streams, the C-ASP reasoner is in charge of: (i) selecting among the list of events, which are the ones that are really relevant according to the Alice's context, and (ii) continuously ranking their level of criticality with respect to her context in order to decide whether a new path needs to be computed.

Data streams. As mentioned in the scenario, there are two input streams: a stream of events and a stream of user's context. An event is modeled with 4 features: event name, category, location, and time. For example, Listing 5.1 describes the condition of weather being strong wind at a certain time and a given location. User's context is defined by current activity, location, and time. For instance, Listing 5.2 illustrate that at a certain time, Alice is driving a car at a particular location.

Static input data. A small instance of the context ontology which describes the effect of events on certain activities is created. For example, the ontology contains the fact `<ec:Fog uc:effect ac:DrivingCar>` which indicates that any person who is driving a car is affected by foggy weather. I generate 10 categories of events such as: roadwork, obstructions, incident, sporting events, disasters, weather, traffic conditions, device status, visibility air quality, incident response status. Each type of events has more than 2 event names (e.g., traffic condition has event names: clear traffic, slow

```
@prefix sao:   <http://purl.oclc.org/NET/UNIS/sao/sao#> .
@prefix tl:    <http://purl.org/NET/c4dm/timeline.owl#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix ec:    <http://purl.oclc.org/NET/UNIS/sao/ec#> .

sao:e1    a    ec:StrongWind ;
        sao:hasCategory    ec:WeatherEvent ;
        sao:hasEventLocation   "38011736-121867224" ;
        tl:time             "2014-11-26T13:00:00"^^xsd:dateTime .
```

LISTING 5.1: An example of a weather event

```
@prefix sao:   <http://purl.oclc.org/NET/UNIS/sao/sao#> .
@prefix tl:    <http://purl.org/NET/c4dm/timeline.owl#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix uc:    <http://purl.oclc.org/NET/UNIS/sao/uc#> .
@prefix ac:    <http://purl.oclc.org/NET/UNIS/sao/ac#> .

uc:uc1   uc:user    uc:Alice ;
        uc:activity    ac:DrivingCar ;
        sao:hasUserLocation   "38011736-121867224" ;
        tl:time             "2014-11-26T12:00:00"^^xsd:dateTime .
```

LISTING 5.2: An example of user's context

traffic, and congested traffic). In addition, the route of the user is also considered as static input data.

Reasoning request. The Listing 5.3 shows a snapshot of the C-ASP reasoning request which is used for this experiment. This request contains 10 rules which have 2 negation-as-failure rules. Rule r_1 identifies events that affect the current activities of the user. Rules $r_2 - r_7$ define 'out_of_context' events which may be relevant but not be critical to the user. Rule r_2 confirms that an event is 'out_of_context' if it does not affect the current activity of the user. Rules r_3 and r_4 identify the case when the user is moving out of the path where related event is happening. Similarly, rules r_6 and r_7 filter events that are not happening on the user's path. Rule r_5 shows that the related event is 'out_of_context' when the user has passed by the event's location. Rules r_8 and r_9 compute the criticality of an event and rule r_{10} reports the event which has criticality is bigger than 0.

5.1.2 Experiment Discussion

The experiment is conducted over a machine running Debian GNU/Linux 6.0.10, containing 8-cores of 2.13 GHz processor and 64 GB RAM. I evaluate the same reasoning request with varying sizes of input window W (from 100 to 30000 events) of the city event stream (see Listing 5.3) and measured the reasoning time of the C-ASP reasoner

```
#prefix tl : ⟨http://purl.org/NET/c4dm/timeline.owl#⟩;
#prefix uc : ⟨http://purl.oclc.org/NET/UNIS/sao/uc#⟩;
#prefix ac : ⟨http://purl.oclc.org/NET/UNIS/sao/ac#⟩;
#prefix sao : ⟨http://purl.oclc.org/NET/UNIS/sao/sao#⟩;
#prefix rdf : ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩;

#from stream <http://city.org/cityevents> [count W step W ];
#from stream <http://city.org/usercontext> [count 1 step 1 ];
#from <effect.rdf>;

(r1)related(EID,UID) :- uc_effect(EN,AN), rdf_type(EID,EN),
                        uc_activity(UID,AN);
(r2)out_of_context(EID,UID) :- rdf_type(EID,EN), uc_activity(UID,AN),
                        not related(EID,UID);
(r3)wrong_location(UID) :- sao_hasUserLocation(UID,ULID),
                        not path_segment(ULID,_);
(r4)out_of_context(EID,UID):- related(EID,UID), wrong_location(UID);
(r5)out_of_context(EID,UID) :- sao_hasUserLocation(UID,ULID),
                        related(EID,UID), path_segment(ULID,UPN),
                        sao_hasEventLocation(EID,ELID),
                        path_segment(ELID,EPN), EPN-UPN<0;
(r6)wrong_location_event(EID) :- sao_hasEventLocation(EID,ELID),
                        not path_segment(ELID,_);
(r7)out_of_context(EID,UID) :- wrong_location_event(EID),
                        related(EID,UID);
(r8)weight(EID,UID,D) :- sao_hasUserLocation(UID,ULID),
                        sao_hasEventLocation(EID,ELID), related(EID,UID),
                        path_segment(ULID,UPN), path_segment(ELID,EPN),
                        EPN-UPN>=0, D = n - (EPN-UPN);
(r9)weight(EID,UID,0) :- out_of_context(EID,UID).
(r10)weight_critical(EID,UID,N) :- weight(EID,UID,N), N > 0.

#show weight_critical/3;
```

Listing 5.3: The reasoning request for notifying critical events

($T(W)$). The reasoner is triggered 20 times for each W and then the average performance time is computed. These values are plotted in Figure 5.1.

Given $U = 1\ s$, the graph shows that the C-ASP reasoner is 'stable' if every window streaming into it has the size is smaller than 17520 *events* (i.e., green line in Figure 5.1). Let denote this 'stable' window as W_s. For window sizes are bigger than $|W_s|$, the reasoner will accumulate a delay that will cause a bottleneck. Giving the reasoning time is monotonically increasing as in Figure 5.1, there are some W bigger than W_s that can be processed in less than 1 U by partitioning W into smaller chunks and process them in sequence. The easiest way to perform this split is to consider several windows of the same size.

For example, consider $|W| = 20000$ *events*, it will take 1232 ms for the reasoner process all events in W in one computation ($T(W) = 1232\ ms$). The reasoner will combine a

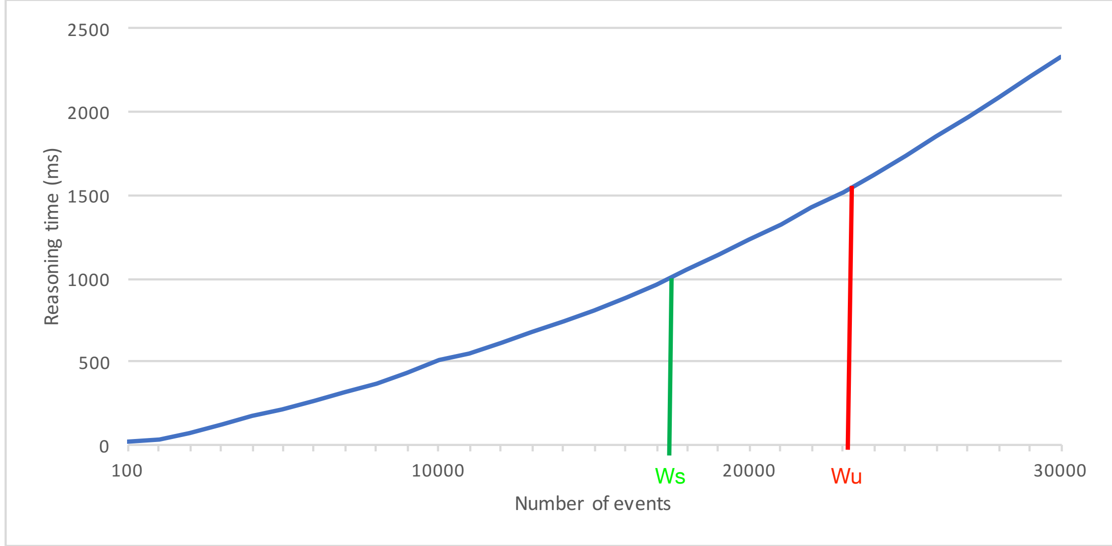


FIGURE 5.1: Reasoning time

delay in each computation and therefore will crash at some point. However, if W is split into 4 chucks of the same size $|W'| = 5000\ events$ and process sequentially, the reasoner will take $T(W, W') = \lceil \frac{20000}{5000} \rceil \times T(5000) = 4 \times 216\ ms = 864\ ms$ for processing W. So there is a proper window size ($|W'|$) such that $T(W, W') \leq U$. In other words, the reasoner can be 'stable' with bigger window sizes.

Moreover, if W is divided into windows of size $|W'| = 2000\ events$, the reasoning time for W will be $T(W, W') = \lceil \frac{20000}{2000} \rceil \times T(2000) = 10 \times 72\ ms = 720\ ms$. It shows that, in general, there could be more than one way to split the input window. For any given W, a proper value for W' such that $T(W, W') \leq U$ can be found in a trivial way just checking for each W' less than W_s and verifying that $T(W, W') = \lceil \frac{W}{W'} \rceil \times T(W') \leq U$. Increasing W up to the point where there is no W' such that $|W'| < |W_s|$ and $T(W, W') \leq U$, this point, denoted as W_u, is the upper bound of the window where the reasoner can scale up to and still maintain the stability. In this experiment, the upper bound value is 23350 *events* (i.e., red line in Figure 5.1). It means that the reasoner can be stable if the window size is less than or equal to 23350 *events*.

In summary, the experiment shows that:

$$\forall W \text{ where } |W_s| \leq |W| \leq |W_u|, \exists W' \text{ such that } T(W, W') \leq U$$

In other words, the processing time of the C-ASP reasoner over an input window can be reduced by reasoning sequentially over the window's partitions. This observation holds under the circumstance of the monotonically increasing reasoning time and independent input data (to guarantee the reasoning results).

```
(r1) very_slow_speed(X) :- average_speed(X,Y), Y<20;
(r2) many_cars(X) :- car_number(X,Y), Y>40;
(r3) traffic_jam(X) :- very_slow_speed(X), many_cars(X),
                          not traffic_light(X,red);
(r4) car_fire(X) :- car_in_smoke(C,high), car_speed(C,0),
                       car_location(C,X);
(r5) give_notification(X) :- traffic_jam(X);
(r6) give_notification(X) :- car_fire(X);
```

Listing 5.4: Sample rules for detecting events

5.2 Reasoning over Dependent Data Streams

Section 5.1 has shown that the reasoning time of the C-ASP reasoner over an input window can be reduced by reasoning parallel (or even sequential) over partitions of that window. This observation is based on the assumption that input data in the window is independent in order to guarantee the reasoning results, which is rarely a case. Moreover, partitioning data randomly in general decreases the accuracy of the final answers. To overcome this issue, the data partitioning process needs to take into account the dependencies among input data items.

5.2.1 Running Example: Traffic Management

In this section, I motivate the problem by providing an example to show the significance of the relationships among input data in the reasoning process of the C-ASP reasoner. This example is then used throughout the rest of the chapter to explain the formal definitions and algorithms which construct the input dependency analysis.

Consider the following scenario: A city manager wants to know real-time events happening in the city in order to make informed decisions on traffic management, reaction to vandalism/crime, management of traffic congestions, reduction of risks for drivers/-cyclists/pedestrians, and so on. To do that, he wants to register a continuous reasoning request into the C-ASP reasoner that integrates and reasons upon relevant semantic streams from different sources to detect events of interest. Listing 5.4 shows a rule set P of his reasoning request to detect events `traffic_jam` and `car_fire`.

Assume that the set of input predicates $inpre(P)$ (i.e., predicates appear in the input streams) as follows:

$inpre(P) = \{$`average_speed, car_number, traffic_light, car_in_smoke, car_speed, car_location`$\}$.

The C-ASP reasoner is triggered whenever a new input window W arrives. As an illustrative example, assume at time t, the input window W (in ASP format) arrives as follows:

W = {*average_speed(newcastleRoad, 10)*, *car_number(newcastleRoad, 55)*, *traffic_light(newcastleRoad,red)*, *car_in_smoke(car1, high)*, *car_speed(car1, 0)*, *car_location(car1, danganRoad)*}

This example is probably not presenting issues in terms of performance, but as the number of cars, segments, traffic lights and other events increases, the scalability of the system becomes an issue. In order to process W faster, partitioning W randomly as in Section 5.1 could generate wrong results. For example, W can be partitioned randomly into two smaller chunks, $W1$ and $W2$ as bellow:

W_1 = {*average_speed(newcastleRoad, 10), car_number(newcastleRoad, 55), car_in_smoke(car1, high)*}, and

W_2 = {*traffic_light(newcastleRoad,red), car_speed(car1, 0), car_location(car1, danganRoad)*}

Reasoning in parallel over these two input partitions produces as a result the event *traffic_jam(newcastleRoad)* and the action *give_notification(newcastleRoad)* is triggered, which is not correct. The accurate answer is the event *car_fire(danganRoad)* detected and the notification about the *danganRoad* segment. Partitioning randomly the input stream may reduce the processing time of C-ASP reasoner but also reduce the accuracy of its reasoning results in return. Therefore, the parallel reasoning process should consider the relations between input data (ground atoms) in the input window, and distribute the computation accordingly across multiple instances of the rule set. Note that this approach is different from distributing the processing by splitting the rules, and it targets instead the input predicates. How this input analysis is done will be detailed in the following sections.

5.2.2 Assumptions

While addressing research question *RQ2*, I make some assumptions on the supported the rule set and the input data streams in the reasoning request.

Regarding the logic program P in the reasoning request which is given in advance to the C-ASP reasoner, I assume that P is constructed under the stratified negation fragment

of normal ASP (i.e., the program should contain no recursion through negation). This assumption ensures uniqueness of the answer sets and contributes in guaranteeing the correctness of the parallel inference process. In this way, I make the restriction on the expressivity of the ASP semantics to achieve: (i) a better scalability via parallelism and (ii) correctness of reasoning results. Relaxing this assumption is an interesting direction for future work.

Regarding the assumption on the input streams, I assume that unrelated input data are filtered out. In other words, the C-ASP reasoner receives only data which are ground atoms of predicates appearing in the logic program P. In a formal way, $inpre(P) \subseteq pre(P)$. In addition, to analyze the dependencies among input data, note that $inpre(P)$ is known in advance.

Another assumption on the input streams which is related to different types of predicates in P. According to the database terminology, there are two types of predicates: EDB (extensional database) and IDB (intensional database). I assume that the input predicates in $inpre(P)$ can be either IDB or EDB predicates.

5.2.3 Input Dependency Analysis

In this section, I discuss the problem of analyzing the dependency of input elements in a window W for the C-ASP reasoner with respect to a stratified negation program P in a reasoning request. I first extend the *dependency graph* defined in 2.15 to capture different relationships among all predicates in P. Thereafter, I introduce the concept of input dependency graph that shows how input data items in W relate to each other with respect to the logic program P.

The concept of dependency graph has been widely used in ASP as a tool to analyze the structure of non-ground answer set programs [2, 57]. It has been efficiently used in parallel instantiation algorithms that generate a much smaller ground program equivalent to a given logic program. Note that the computation of most ASP systems follows a two-phase approach: an instantiation (or grounding) phase generates a variable-free program which is then evaluated by propositional algorithms in the solving phase. The instantiation process in ASP can be expensive from a computational viewpoint and the size of the ground program has a huge effect on the performance of the solver. To address this issue, the idea of parallel grounding has been investigated, which relies on the concept of dependency graph.

As defined in 2.15, a dependency graph G is a directed graph where nodes are IDB predicates and arcs show the relationship between a positive IDB predicate in the body

with a predicate in the head of a rule. This graph divides the input program P into sub-programs, according to the dependencies among the IDB predicates of P, and identifies which of them can be grounded in parallel.

However, in this thesis, I do not partition the logic program for the grounding process. I focus instead on partitioning the input on-the-fly and evaluating each partition in parallel with a copy of the whole program P. The reasons to follow the input partitioning approach are:

- in the context of dynamic environments, the amount of input data streaming into the reasoner normally much bigger than the number of rules defined in a registered reasoning request, and
- in the context of dynamic environments, the amount of input data at each execution varies in terms of rate and size, thus having different effects on performance.

In order to capture this aspect, I first define an *extended dependency graph* which stems from the definition in 2.15. Besides the dependencies among IDB predicates defined in the dependency graph, other relationships should be taken into account, such as between two EDB predicates, or between an IDB predicate and an EDB predicate. This graph shows different types of dependencies among predicates in P by considering:

- i) the (transitive) relation between two predicates (both IDB and EDB) in the body of a rule,
- ii) both positive and negative literals.

Definition 5.1. Let P be a logic program. The *extended dependency graph* of P, denoted as $G_P = \langle N_P, E_P \rangle$, is a graph in which:

i) N_P is a set of nodes, where each node represents a predicate in $pre(P)$.

ii) $E_P = E_{P_1} \cup E_{P_2}$, where:

  (a) E_{P_1} contains undirected edges $e_u = (p_u, q_u)$ if p_u and q_u occur in the body of a rule r in P. Moreover, $(p_u, p_u) \in E_{P_1}$ if $p_u \in B^-(r)$.

  (b) E_{P_2} contains directed edges $e_d = \langle p_d, q_d \rangle$ if q_d occurs in the head of r and p_d occurs in the body of r.

  Note that p_u, q_u, p_d, q_d are predicates that can appear in either a positive or a negative literal.

Example 5.1. *Consider the program P in Listing 5.4. The extended dependency graph G_P illustrated in Figure 5.2 represents different relations among predicates in P including directed and undirected edges.*

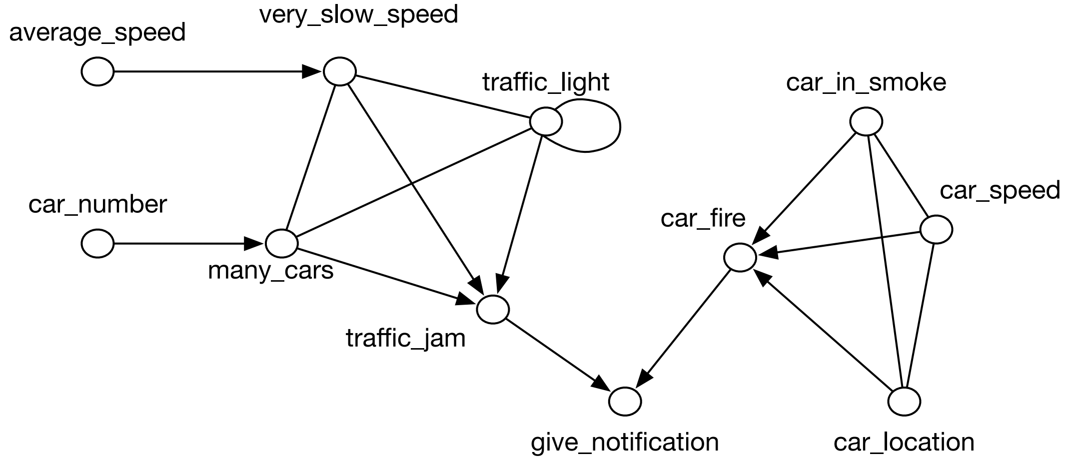


FIGURE 5.2: Extended dependency graph G_P

Based on the extended dependency graph, I introduce the concept of *input dependency graph* of P with respect to $inpre(P)$. This input dependency graph describes how predicates in $inpre(P)$ depend on each other. Below, I describe the meaning of *direct path* that is used to build the input dependency graph.

Definition 5.2. Given the extended dependency graph $G_P = \langle N_P, E_P \rangle$ of the logic program P, a directed path from node p_1 to node p_n is a sequence of nodes $p_1, p_2, ..., p_n$ such that $p_i \in N_P, i = 1..n$ and $\langle p_j, p_{j+1} \rangle \in E_{P_2}, j = 1..n-1$.

Example 5.2. *A directed path from* `average_speed` *to* `give_notification` *is a sequence of predicates:*

`average_speed, very_slow_speed, traffic_jam, give_notification.`

Definition 5.3. Let P be a logic program, $G_P = \langle N_P, E_P \rangle$ be an extended dependency graph of P, and $inpre(P)$ be a set of input predicates of P. The *input dependency graph* of P with respect to $inpre(P)$ is an undirected graph $G_P^{inpre(P)} = \langle N_P^{inpre(P)}, E_P^{inpre(P)} \rangle$, where $N_P^{inpre(P)} \subset N_P$ is a set of nodes and $E_P^{inpre(P)}$ is a set of edges. $N_P^{inpre(P)}$ contains a node for each predicate in $inpre(P)$, and $\forall p, q \in N_P^{inpre(P)}$, $(p, q) \in E_P^{inpre(P)}$ if one of the following conditions is satisfied:

i) $p \neq q$ and there is a sequence of nodes $p_1, p_2, ..., p_{n-1}, p_n$ $(n > 1, p_1 = p, p_n = q)$ such that $\exists i \in [1, n)$, $(p_i, p_{i+1}) \in E_{P_1}$ and there are two directed paths: one is from p_1 to p_i if $p_1 \neq p_i$ and the other is from p_n to p_{i+1} if $p_n \neq p_{i+1}$.

ii) $p = q$ and $((p,p) \in E_{P_1}$ or $\exists u \in N_P, (u,u) \in E_{P_1}, \langle p,u \rangle \in E_{P_2})$.

Example 5.3. *Consider the extended dependency graph G_P in Example 5.1 with the input predicates $inpre(P) = \{$`average_speed, car_number, traffic_light, car_in_smoke, car_speed, car_location`$\}$. The input dependency graph $G_P^{inpre(P)}$ is shown in Figure 5.3.*

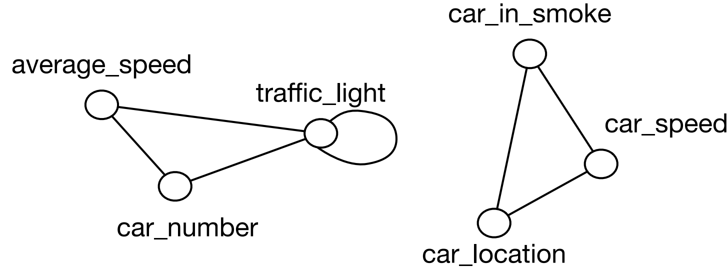


FIGURE 5.3: Input dependency graph $G_P^{inpre(P)}$

In this example, the link between `car_in_smoke` and `car_speed` is created based on the basic case of the condition (i) in Definition 5.3 in which p_1 = `car_in_smoke`, p_2 = `car_speed`, and $(p_1, p_2) \in E_{P_1}$. The link between `average_speed` and `car_number` illustrates the general case of the condition (i) where a sequence of four predicates ($n = 4$) p_1 (`average_speed`), p_2 (`very_slow_speed`), p_3 (`many_cars`), p_4 (`car_number`) satisfies that $i = 2$, $(p_2, p_3) \in E_{P_1}$ and there are two directed paths, one is from p_1 to p_2 and the other is from p_4 to p_3. The self-loop of the predicate p = `traffic_light` derives from condition (ii) where $(p,p) \in E_{P_1}$.

Definition 5.4. Let P be a logic program and $inpre(P)$ be a set of input predicates of P. Predicates $p, q \in inpre(P)$ *depend on each other* if there is an edge (p,q) in the input dependency graph $G_P^{inpre(P)}$.

The first condition in Definition 5.3 represents the dependency between two different predicates in $inpre(P)$. In this way, all ground atoms of these two predicates are considered to be dependent to each other, but there is no dependency among ground atoms of the same predicate. These are called *predicate-level* dependencies. Figure 5.4 (a) show an example of predicate-level dependencies among ground atoms (in dash boxes) of predicates `average_speed` and `car_number`. The second condition in Definition 5.3 shows a predicate dependent on itself. It indicates that all ground atoms of that predicate are considered to be dependent on each other. These are called *atom-level* dependencies. The Figure 5.4 (b) illustrates such atom-level dependencies among ground atoms of the predicate `traffic_jam`. When two different predicates depend on each other or a predicate depends on itself, it means that their ground atoms can contribute to infer a new

fact by firing a single rule or multiple rules. Therefore, all ground atoms of dependent predicates need to be processed together in order to guarantee that rules in P are fired properly and to ensure correctness of results.

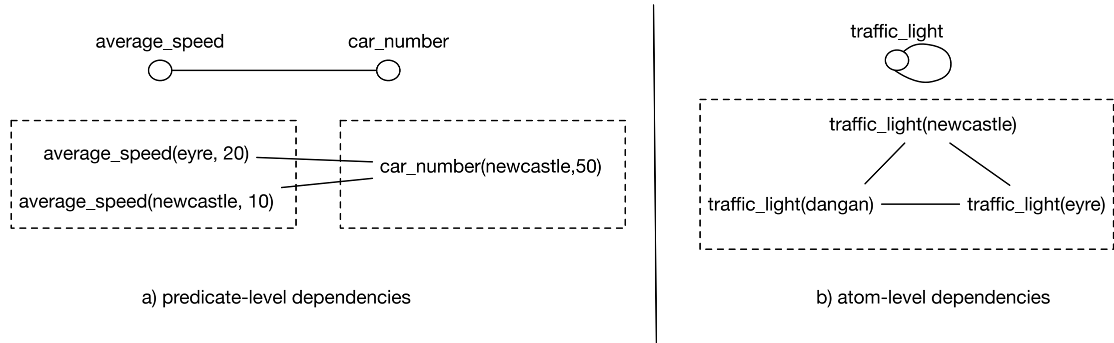


FIGURE 5.4: Types of dependencies

5.2.4 Building Input Dependency Graph

In this section, I present a set of algorithms to build an input dependency graph when a logic program and a set of input predicates are given in advance. The process is depicted in Figure 5.5. First, the given logic program in the registered reasoning request is used as input to build an extended dependency graph as defined in Definition 5.1. Then this extended dependency graph is fed to the next step together with the given input predicates to create an input dependency graph.

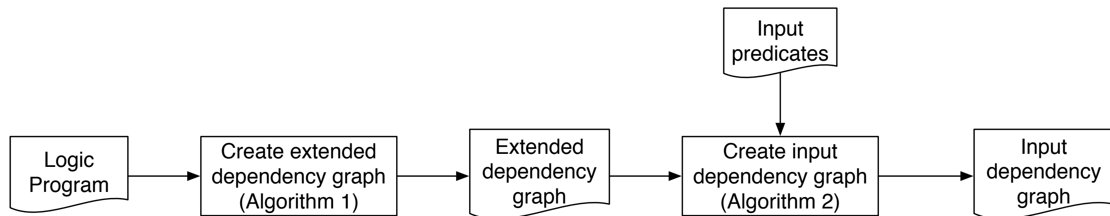


FIGURE 5.5: The process to build an input dependency graph

The function to create an extended dependency graph is shown in Algorithm 1. The algorithm processes each rule r in the given logic program P by first adding all predicates in r into N_P (a set of nodes). Line 4 uses the method $pre()$ to compute all predicates in r. Next, for each predicate in the body of r (Line 5), the algorithm creates a self-loop for the predicate if it appears in negated literal ($IsNegative()$) or it appears more than 1 time in the body of r ($IsDuplicate()$) (Lines 6 - 8). These lines correspond to condition (a) of Definition 5.1. Lines 9 - 11 create undirected edges between two different predicates (i.e., the condition (a) of Definition 5.1) while the directed edges indicate the dependencies between a predicate in the body with a predicate in the head of r are created in Lines 12 - 14 (i.e., the condition (b) of Definition 5.1).

Algorithm 1 Creating extended dependency graph

Input: a logic program P
Output: an extended dependency graph G_P

1: **procedure** EDG(P)
2: $N_P \leftarrow \{\}$, $E_{P_1} \leftarrow \{\}$, $E_{P_2} \leftarrow \{\}$
3: **for** $r \in P$ **do**
4: $N_P = N_P \cup pre(r)$
5: **for** $b \in pre(B(r))$ **do**
6: **if** $IsNegative(b) || IsDuplicate(b)$ **then**
7: $E_{E_1} = E_{P_1} \cup \{(b, b)\}$
8: **end if**
9: **for** $b' \neq b \in pre(B(r))$ **do**
10: $E_{E_1} = E_{P_1} \cup \{(b, b')\}$
11: **end for**
12: **for** $h \in pre(H(r))$ **do**
13: $E_{E_2} = E_{P_2} \cup \{\langle b, h \rangle\}$
14: **end for**
15: **end for**
16: **end for**
17: $E_P = (E_{P_1}, E_{P_2})$
18: **return** $G_P = \langle N_P, E_P \rangle$
19: **end procedure**

Algorithm 1 guarantees the creation of the extended dependency graph because all predicates in a rule and all rules in the given logic program are examined. The most inner loop (Line 12) needs to be executed for every predicates in the head of a rule. In normal ASP programs (i.e., the programs contain all normal rules), the head contains only 1 predicate so the iteration for this loop is 1. The loops in Lines 5 and 9 are executed for every predicate in the body of a rule while the outer loop (Line 3) iterates according to the number of rules in the logic program. Given n_b is the maximum number of predicates in the body among all rules in the logic program ($n_b = max(|pre(B(r))| : \forall r \in P)$), the worst-case time complexity of Algorithm 1 is $\mathcal{O}(|P|n_b^2)$ ($|P|$ is the number of rules in P).

Algorithm 2 creates an input dependency graph as defined in Definition 5.3. $N_P^{inpre(P)}$ and $E_P^{inpre(P)}$ contain vertexes and edges of the graph. At the beginning, each predicate in $inpre(P)$ is identified as a vertex (Line 2). Each vertex is checked to see if it depends on other vertexes according to the conditions in Definition 5.3. In Lines 5-9, the algorithm checks condition (i) in Definition 5.3 by calling the underlying function *CheckDependency* which is detailed in Algorithm 3. Lines 10-17 create a self-loop for a vertex if the condition (ii) in Definition 5.3 holds. First, it takes a self-loop in E_{P_1} that is related to the current vertex (Lines 10-12). Then, it creates a self-loop for a vertex if this vertex implies another self-loop vertex (Lines 13-17).

The goal of the function *CheckDependency* is to check if two separated vertexes v_1 and

Algorithm 2 Creating input dependency graph

Input: an extended dependency graph G_P and a set of input predicates $inpre(P)$
Output: an input dependency graph $G_P^{inpre(P)}$

```
1: procedure IDG(G_P, inpre(P))
2:     N_P^{inpre(P)} ← inpre(P)
3:     E_P^{inpre(P)} ← {}
4:     for v_1 ∈ N_P^{inpre(P)} do
5:         for v_2 ∈ N_P^{inpre(P)} do
6:             if CheckDependency(v_1, v_2, G_P) then
7:                 E_P^{inpre(P)} = E_P^{inpre(P)} ∪ {(v_1, v_2)}
8:             end if
9:         end for
10:        if (v_1, v_1) ∈ E_{P_1} then
11:            E_P^{inpre(P)} = E_P^{inpre(P)} ∪ {(v_1, v_1)}
12:        end if
13:        for v ∈ N_P do
14:            if (v, v) ∈ E_{P_1} & ⟨v_1, v⟩ ∈ E_{P_2} then
15:                E_P^{inpre(P)} = E_P^{inpre(P)} ∪ {(v_1, v_1)}
16:            end if
17:        end for
18:    end for
19:    return G_P^{inpre(P)} = ⟨N_P^{inpre(P)}, E_P^{inpre(P)}⟩
20: end procedure
```

v_2 depend on each other as per the condition (i) in Definition 5.3. There is a basic dependency between two predicates if there is an undirected link between them (Lines 12-13). Otherwise, the algorithm will find if there are two direct paths connected by an undirected edge between those two vertexes. This function is extended from the Breadth-First Search (BFS) algorithm to discover those paths. This algorithm will terminate at Line 13 or when all vertexes are checked.

The creation of an input dependency graph in Algorithm 2 is guaranteed because the dependencies between every two predicates in $inpre(P)$ are examined. The main operator of the algorithm is the *CheckDependency* method which check if two different predicates depend on each other according to the condition (i) in Definition 5.3. Lines 7 - 20 in Algorithm 3 is basically the BFS algorithm operating on $inpre(P)$ and a set of directed edges E_{P_2}. The outer loop of Algorithm 3 performs another similar BFS. Then, the time complexity of *CheckDependency* method is $\mathcal{O}((|inpre(P)| + |E_{P_2}|)^2)$. Algorithm 2 triggers *CheckDependency* method for every pair of predicates in $inpre(P)$. Therefore, the time complexity of Algorithm 2 is $\mathcal{O}(|inpre(P)|^2(|inpre(P)| + |E_{P_2}|)^2)$.

Algorithm 3 Check dependency between 2 vertexes

Input: two vertexes v_1, v_2 and an extended dependency graph G_P
Output: true/false

```
1: procedure CheckDependency(v1, v2, G_P)
2:     queueV1 ← [v1]
3:     queueV2 ← [v2]
4:     checked ← {}
5:     while queueV2 ≠ ∅ do
6:         tempV2 ← queueV2.remove(0)
7:         while queueV1 ≠ ∅ do
8:             tempV1 ← queueV1.remove(0)
9:             if (tempV1, tempV2) ∈ checked then
10:                continue
11:            end if
12:            if (tempV1, tempV2) ∈ E_P1 then
13:                return true
14:            else
15:                for ⟨tempV1, cV⟩ ∈ E_P2 do
16:                    Add cV into queueV1
17:                end for
18:            end if
19:            Add (tempV1, tempV2) into checked
20:        end while
21:        Add v1 into queueV1
22:        for ⟨tempV2, cV⟩ ∈ E_P2 do
23:            Add cV into queueV2
24:        end for
25:    end while
26:    return false
27: end procedure
```

5.3 Summary

In this chapter, the first section reports the correlation between the input window size, the streaming rate and the reasoning time. The experiment shows that when the reasoning time of the C-ASP reasoner over streaming data is monotonically increasing, this time can be reduced by reasoning sequentially over the input window's partitions. The data elements in the window are assumed to be independent to make sure that the reasoning results are correct. To relax this assumption, the second section is dedicated to analyze how these input data elements depend on each other. A clear characterization and formal definitions for analyzing the dependencies among input data streams are provided. At the center of the proposed analysis is the concept of an input dependency graph which expresses two types of dependencies among data: predicate-level and atom-level. The input dependency graph is built based on the extension of the dependency

graph in ASP. A set of algorithms for building this graph are provided and their complexities are discussed. Leveraging outcomes in this chapter, the next chapter describes how to use the proposed input dependency graph to enable parallel reasoning while maintaining the correctness of results. The material presented in the first part of this chapter has been published in [197] and also in the PhD thesis "Logic Programming in non-conventional environments"[2] of Stefano Germano (i.e., the co-author of the paper [197]). The work presented in the second part can be found at [198] and [199].

[2] https://www.mat.unical.it/phd/Alumni?action=AttachFile&do=get&target=GermanoAbstract.pdf

Chapter 6

Input-driven Parallel Reasoning

Chapter 5 targets on finding the information which can be used to optimize the C-ASP reasoner (*RQ2.1*). As the result, the input dependency graph is defined to capture how input data in a window depends on each other. Continuing from this investigation, this chapter tackles mainly the second sub-question of the research question *RQ2* that states the following:

> *RQ2.2.* How to use such information found in *RQ2.1* efficiently to speed up the reasoning process without losing the correctness of reasoning results?

The first part of the hypothesis $H3$ has been demonstrated in Chapter 5. This chapter focus on testing the second part of $H3$ and the hypothesis $H4$ that are formulated as follows:

> *H3.* The semantic dependencies between input data streams can be captured based on the structure of a given rule sets. These dependencies can be decomposed in such a way to maintain the correctness of the parallel reasoning results.

> *H4.* Parallel reasoning over partitioned data streams when taking into account input dependencies can reduce the reasoning cost and maintain the correctness of combined reasoning results.

To answer the research question *RQ2.2* and to test the hypotheses *H3, H4*, this chapter investigates a method to partition data streams based on the input dependency graph. This method and a proof of correctness of the results are presented in Section 6.1.

Section 6.2 describes how to use the proposed partitioning plan in guiding the parallel reasoning of the C-ASP engine. Section 6.3 conducts an evaluation on the effectiveness of this approach via experiments with different levels of expressivity of the reasoning request. The chapter is then summarized in Section 6.4.

6.1 Partitioning Plan

In this section, I investigate on how to use the input dependency graph in optimizing the C-ASP reasoner with parallelism while maintaining the correctness of reasoning results. The parallel reasoning in C-ASP is enabled via the data partitioning approach. In other words, the input data in a window is partitioned into smaller chunks. However, this partitioning method needs to take into account the dependencies among input data in order to ensure the accuracy of the final results. In this way, the input dependency graph is used to create a so-call *partitioning plan*, defined as follows:

Definition 6.1. A partitioning plan of a given input dependency graph $G_P^{inpre(P)} = \langle N_P^{inpre(P)}, E_P^{inpre(P)} \rangle$ is defined as:

$$\{N_1, ..., N_r : N_r \subseteq N_P^{inpre(P)} \text{ and } 1 \leq r \leq |N_P^{inpre(P)}|\}$$

I analyze the input dependency graph for creating the partitioning plan in two cases: unconnected and connected graph, detailed in Section 6.1.1 and 6.1.2 respectively.

6.1.1 Unconnected Input Dependency Graph

In this subsection, I consider a case where the input dependency graph is unconnected[1] to partition the input window W into smaller chunks and provide the proof for the correctness of results when reasoning parallel over these chunks. This serves as a basic case of the partitioning plan.

The input dependency graph $G_P^{inpre(P)}$ that is not connected induces naturally a subdivision of the graph into several *connected components* (or *components*). A connected component of an undirected graph is a maximal connected subgraph of the graph. For instance, $G_P^{inpre(P)}$ in Figure 5.3 is decomposed into two components which have separated sets of nodes from $inpre(P)$:

[1] An undirected graph is connected if, for every pair of vertexes, there is a path in the graph between those vertexes.

N_1 = {`average_speed`, `traffic_light`, `car_number`}, and
N_2 = {`car_in_smoke`, `car_speed`, `car_location`}.

These sets of nodes are used as a partitioning plan in the partitioning process of the parallel C-ASP reasoner (presented in Section 6.2) for splitting ground atoms in an input window on-the-fly.

Consider the input window W as in Section 5.2.1. Splitting W while taking into account the partitioning plan (N_1, N_2) produces two smaller chunks as follows:

W'_1 = {`average_speed(newcastle, 10), car_number(newcastle, 55), traffic_light(newcastle)`}
W'_2 = {`car_speed(car1, 0), car_in_smoke(car1, high) car_location(car1, dangan)`}

Parallel reasoning over these two chunks against the rule set in Listing 5.4 produces two sub-results: {} for W'_1 and {`car_fire(dangan)`, `give_notification(dangan)`} for W'_2. Union of these two sub-results provides the correct answer as reasoning over whole W.

In order to ensure the proposed approach provides all and only the expected results when the input window is split and processed in parallel, I close this subsection by providing a sketch of the correctness proof.

Proposition 1. *Given $G_P^{inpre(P)}$ that is not connected, $G_1, ..., G_n$ $(n > 1)$ are connected components of $G_P^{inpre(P)}$, and W is an input window such that $pre(W) \subseteq inpre(P)$:*

$$Ans_P(W) = \bigcup_{i=1}^{n} Ans_P(W_i)$$

where $W = \bigcup_{i=1}^{n} W_i$, and $pre(W_i)$ is the set of nodes of G_i.

Proof. I introduce some notations that are used in the proof:

- $pre(body(r))$: a set of predicates appearing in the body of rule r.
- $pre_head(r)$: a predicate appearing in the head of rule r.
- $ground(p)$: a set of ground atoms over the predicate p.

Suppose $a \in Ans_P(W)$, let consider the following cases:

- a is created by firing one rule r in P
  $\Rightarrow \forall_{p_i,p_j \in pre(body(r))}(p_i \neq p_j), (p_i, p_j) \in E_P^{inpre(P)}$
  $\Rightarrow \exists i \in [1, n] : \forall p \in pre(body(r)), ground(p) \subset W_i$
  $\Rightarrow a \in Ans_P(W_i)$
  $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$

- a is created by firing two rules r_1, r_2 in P
  $\Rightarrow pre_head(r_1) \in pre(body(r_2))$

  - If $pre(body(r_2)) = \{pre_head(r_1)\}$
    $\Rightarrow \forall p \in pre(body(r_1)),\ (pre_head(r_1), p) \notin E_P^{inpre(P)}$
    $\Rightarrow \exists W_i \neq W_j : pre(body(r_1)) \subset pre(W_i)$ and $pre_head(r_1) \in pre(W_j)$
    $\Rightarrow \forall p \in pre(body(r_1)), ground(p) \subset W_i$ and $ground(pre_head(r_1)) \subset W_j$
    $\Rightarrow a \in Ans_P(W_i)$ (by firing both r_1 and r_2) or $a \in Ans_P(W_j)$ (by firing r_2)
    $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$
  - Else
    $\Rightarrow \forall p \in pre(body(r_1)), \forall q \in pre(body(r_2)), (p, q) \in E_P^{inpre(P)}$
    $\Rightarrow \exists i = [1..n] : \forall p \in pre(body(r_1)) \cup pre(body(r_2)), ground(p) \subset W_i$
    $\Rightarrow a \in Ans_P(W_i)$
    $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$

- Similarly, when a is created by firing k rules $r_1, ..., r_k$ in P
  $\Rightarrow \exists i \in [1..n] : \forall p, q \in \bigcup_{j=1}^{k} pre(body(r_j)), (p, q) \in E_P^{inpre(P)} \rightarrow p, q \in pre(W_i)$
  $\Rightarrow \exists W_i : a \in Ans_P(W_i)$
  $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$

Suppose $a \in \bigcup_{i=1}^{n} Ans_P(W_i) \Rightarrow \exists i \in [1..n] : a \in Ans_P(W_i)$

- If a is created by firing a set of positive rules
  $\Rightarrow a \in Ans_P(W)$ because $W_i \subset W$

- If a is created by firing a set of rules (e.g., $r_1, ..., r_k$) with negation-as-failure
  $\Rightarrow \forall p \in \bigcup_{i=1}^{k} pre(body^{-}(r_j)), ground(p) \subset W_i$ and $\nexists W_j \neq W_i$: $ground(p) \subset W_j$
  $\Rightarrow a \in Ans_P(W)$.

Therefore, the proposition 1 is proved.

6.1.2 Connected Input Dependency Graph

In Section 6.1.1, I presented how a partitioning plan is created in the case of an unconnected input dependency graph. This section is dedicated to investigate to create a

partitioning plan under the circumstance of a connected input dependency graph. This type of graph is not straightforward to identify and separate connected components, which are the main factors to build the partitioning plan. Below, I describe an example to motivate the problem and illustrate the solution approach.

Consider the logic program P' which includes P in Listing 5.4 (Section 5.2.1) and the following rule:

(r_7) `traffic_jam(X) :- car_fire(X), many_cars(X).`

Assume that $inpre(P') = inpre(P) = \{$`average_speed, car_number, traffic_light, car_in_smoke, car_speed, car_location`$\}$. The input dependency graph $G_{P'}^{inpre(P')}$ is shown in Figure 6.1.

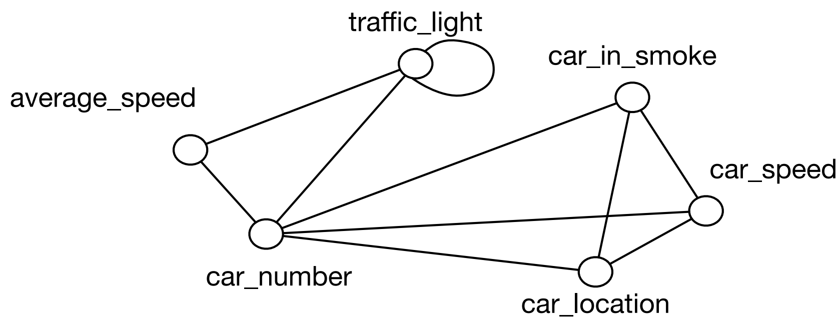


FIGURE 6.1: Input dependency graph $G_{P'}^{inpre(P')}$

This graph is connected, which means that input data of all predicates in $inpre(P')$ have to be processed together in order to maintain the correctness of reasoning results. The parallel reasoning approach cannot be applied if the input dependency graph cannot be decomposed as in this case. To cope with this issue, I introduce the decomposing process to create the partitioning plan from the connected input dependency graph by extending the Greedy algorithm introduced in PowerGraph [200]. The Greedy algorithm is a rule-based partitioning mechanism which aims to minimize vertex-cuts[2] while assigning balanced load across partitions. To do so, this algorithm requires a predefined number of partitions and chooses the partition for two endpoint vertexes of every edge in the graph by applying the following rules:

- Rule 1: if both endpoint vertexes have been previously assigned in any common partition, pick the smallest common partition.

[2] A vertex-cut set is a set of vertices of a graph which, if removed (or "cut")–together with any incident edges–disconnects the graph.

- Rule 2: if either vertex has been previously assigned partitions or both of vertexes have been previously assigned in different partitions, pick the smallest partition from the union of all assigned partitions of two vertexes.
- Rule 3: if none of vertexes has been previously assigned, pick the smallest partition overall.

I extend the Greedy algorithm by: (i) relaxing the requirement of a predefined number of partitions, and (ii) choosing the partition based on not only its size but also how connected the graph formed by vertexes in that partition is. The first extension enables the automatic decomposing process without any knowledge of the number of partitions. The second extension prioritizes assigning vertexes into a partition such that after the assignment, all vertexes in that partition can create a highly connected graph. In this way, a highly connected sub graph of the input dependency graph shall be put in the same partition and reduce the number of common vertexes among different partitions. Let call such partition is 'highly connected'. Similar to highly connected graph, a partition is considered as 'highly connected' if the degree of each vertex in that partition is bigger than or equal to $\lfloor n_p/2 \rfloor + 1$ (n_p is the number of vertexes in the partition).

The greedy-based decomposing process for partitioning the input dependency graph is presented in Algorithm 4. The algorithm takes the input dependency graph as input and provides a partitioning plan as output. PP is used to store the partitioning plan (Line 2) while D keeps the information about the degree of a vertex in each partition (Line 3). For each edge $e = (s, t)$ in the input dependency graph, the algorithm first uses the function `findPartitionIds` to find the set of partitions, ps, containing the endpoint vertex s in Line 6. Similarly, pt denotes the set of partitions containing the endpoint vertex t (Line 7). Then, this algorithm follows three rules of Greedy in Lines 8-10 (Rule 1), Lines 11-13 (Rule 2), and Lines 14-16 (Rule 3). The function `choosePartition` used in Rule 1 and 2 examines the participation of the edge e based on both the size of the partition and how much the vertexes in that partition connected. Rule 3 is modified to add new partition into the partitioning plan PP if none of vertexes has been previously assigned. After identifying the partition Id for the edge e, the algorithm adds two endpoint vertexes of e into that partition and updates the degrees of those vertexes in D.

The function `choosePartition` takes 4 input parameters: the edge e, the set of partition Ids S (i.e., $S = ps \cap pt$ (Rule 1) or $S = ps \cup pt$(Rule 2)), the partitioning plan PP, and the information of vertexes' degrees D. This function returns the partition Id as output. The output partition Id has to satisfy two conditions: smallest size and highly connected. To do so, this function first sorts the partition Ids in S in the ascending manner based

on the partition size (Line 25). Later, it chooses the smallest and highly connected partition (Lines 26-30) using the function `checkHighlyConnected`. If there is no highly connected partition, the smallest one is provided as output (Line 32).

Algorithm 4 Decomposing process

Input: input dependency graph $G_P^{inpre(P)} = \langle N_P^{inpre(P)}, E_P^{inpre(P)} \rangle$
Output: Partitioning plan

1: **procedure** DecomposeIDG($G_P^{inpre(P)}$)
2: $PP \leftarrow \{\}$
3: $D \leftarrow \{\}$
4: **for** $e = (s,t) \in E_P^{inpre(P)}$ **do**
5: **if** $s \neq t$ **then**
6: $ps \leftarrow findPartitionIds(s, PP)$
7: $pt \leftarrow findPartitionIds(s, PP)$
8: **if** $ps \cap pt \neq \emptyset$ **then**
9: $pId \leftarrow choosePartition(e, ps \cap pt, PP, D)$
10: **end if**
11: **if** $ps \cap pt = \emptyset$ && $ps \cup pt \neq \emptyset$ **then**
12: $pId \leftarrow choosePartition(e, ps \cup pt, PP, D)$
13: **end if**
14: **if** $ps = \emptyset$ && $pt = \emptyset$ **then**
15: $pId \leftarrow addNewPartition(PP)$
16: **end if**
17: Add s and v into partition pId
18: Update partition pId in PP
19: Update degrees of vertexes in the partition pId in D
20: **end if**
21: **end for**
22: return PP
23: **end procedure**
24: **procedure** choosePartition(e,S,PP,D)
25: $sortedS \leftarrow sort(S, PP)$
26: **for** $pId \in S$ **do**
27: $dpId \leftarrow D.get(pId)$
28: **if** checkHighlyConnected(e,dpId) **then**
29: return pId
30: **end if**
31: **end for**
32: return $sortedS.get(0)$
33: **end procedure**

The function `findPartitionIds` has the worse-case complexity of $\mathcal{O}(kn)$ to find the set of partitions containing a vertex (k is the number of partitions and n is the number of vertexes of the input dependency graph. The function `choosePartition` has the overall time complexity of $\mathcal{O}(k^2+n)$, in which $\mathcal{O}(k^2)$ comes from the function `sort` and $\mathcal{O}(n)$ is the complexity of the function `checkHighlyConnected`. The function `addNewPartition` has constant complexity since it simply initializes the new partition and then adds to

the partitioning plan. The main function `DecomposeIDG` traverses every edges in the input dependency graph, so its time complexity is $\mathcal{O}(m(kn+k^2+n))$ (m is the number of edges).

Example 6.1. *Consider the input dependency graph $G_{P'}^{inpre(P')}$ in Figure 6.1. Algorithm 4 creates a partitioning plan with two partitions $N_1 = \{$`traffic_light, average_speed, car_number`$\}$ and $N_2 = \{$`car_number, car_in_smoke, car_speed, car_location`$\}$ in which the vertex `car_number` appears in both partitions. These two partitions create two sub graphs of the input dependency graph as in Figure 6.2.*

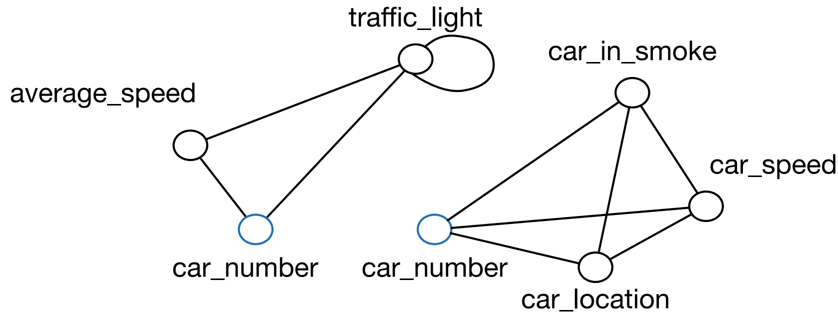


FIGURE 6.2: Output of the decomposing process for $G_{P'}^{inpre(P')}$

In order to ensure that under the circumstance of connected input dependency graph, the proposed approach provides all and only the expected results when the input is split and processed in parallel, I now provide a sketch of the correctness proof.

Proposition 2. *Given $G_P^{inpre(P)}$ that is connected and W is an input window such that $pre(W) \subseteq inpre(P)$:*

$$Ans_P(W) = \bigcup_{i=1}^{n} Ans_P(W_i)$$

where $W = \bigcup_{i=1}^{n} W_i$, and $pre(W_i)$ are computed by the Algorithm 4.

Proof. When $G_P^{inpre(P)}$ is connected, Algorithm 4 decomposes $inpre(P)$ into $pre(W_i), i = 1..n$ and the intersection of any two sets $pre(W_i)$ and $pre(W_j)$ ($pre(W_i) \neq pre(W_j)$) may be not empty. Without loosing generality, assume that $pre(W_i) \cap pre(W_j) = \{p\}, p \in inpre(P)$. All ground atoms of $p \in W$ occurs in both W_i and W_j. In this way, I do not loose the dependencies between p with other predicates in $pre(W_i)$ (or in $pre(W_j)$). Therefore, the correctness of the parallel reasoning process is maintained as proved in Proposition 1.

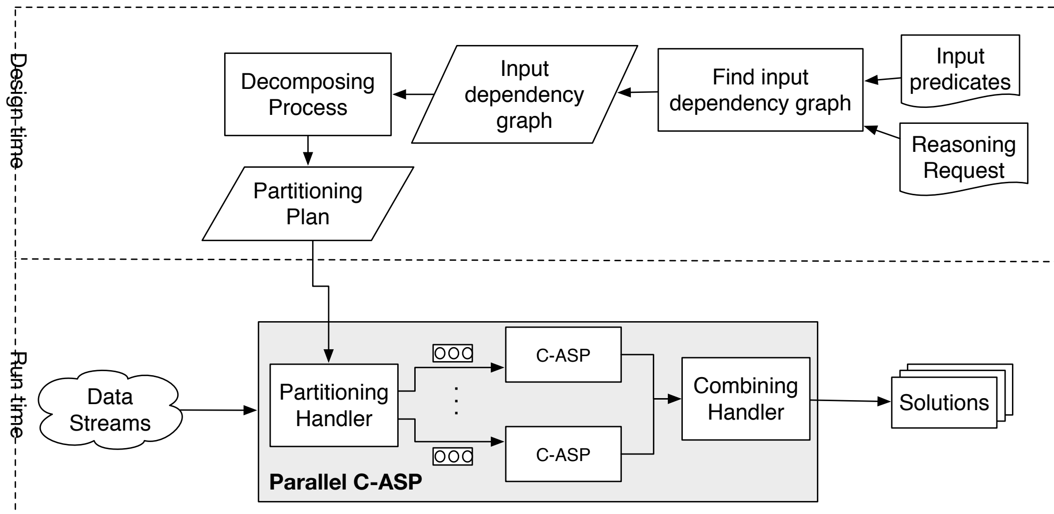


FIGURE 6.3: The Extended StreamRule

6.2 Parallel Reasoning in C-ASP

The C-ASP reasoner extended with the partitioning process is shown in Figure 6.3. The extension consists of the *partitioning handler* and the *combining handler*. The partitioning handler splits an input window W coming from streams into several sub-windows taking into account the input dependency. The combining handler combines outputs from parallel instances of the reasoner. For the realization of the partitioning process, the analysis of input dependency is made available within the framework initially at design time. To achieve this, the set of rules in the reasoning request and a set of input predicates are given in advance in order to build an input dependency graph as defined in Definition 5.3. Then the graph decomposing process described in Section 6.1 builds a partitioning plan by decomposing this graph into several components, with duplicated predicates when needed.

The partitioning handler. At run-time, the partitioning handler starts to split an input window on-the-fly by using the partitioning plan provided at design-time. Algorithm 5 shows the partitioning process. First, the `group()` method classifies items in the window by their predicates (Line 3). For each group of items, the algorithm identifies a set of partitions' Ids that groups belong to based on the partitioning plan (Line 5). Finally, it adds that group into the proper partitions corresponding to those Ids.

Grouping input data in W based on their predicate symbols in Line 3 has a time complexity of $\mathcal{O}(|W|)$. Identifying where each group should add to (Line 5) has an overall complexity $\mathcal{O}(n_p)$ (n_p is the number of partitions in the partitioning plan). Lines 6 - 8 are triggered n_g times in which n_g is the number of groups found in Line 5. That makes the overall time complexity of Lines 4 - 9 to be $\mathcal{O}(|inpre(W)|(n_p + n_g))$. Due to the

fact that the number of input data is much bigger than the number of their predicate symbols plus the number of partitions in the partitioning plan, the overall complexity of Algorithm 5 is proportional to $\mathcal{O}(|W|)$.

The combining handler. To combine the outputs from concurrent instances of the C-ASP reasoner, the combining handler unions them to form the final result as follows:

$$Ans_P(W) = \bigcup_{i=1}^{n} Ans_P(W_i)$$

Where P is a program under stratified negation, W is an input window, and W_i $(i = 1..n)$ are partitions of W provided by the partitioning handler. The correctness of the parallel reasoning process is guaranteed by Proposition 2.

Algorithm 5 Partitioning method

Input: a partitioning plan PP and an input window W
Output: sub-windows of W

```
1: procedure PARTITION(PP, W)
2:     Partitions ← [ ];
3:     G ← group(W);
4:     for g ∈ G do
5:         C ← findPartitionIds(g.predicate, PP);
6:         for c ∈ C do
7:             Add g.items into Partitions[c];
8:         end for
9:     end for
10:    return Partitions;
11: end procedure
```

6.3 Evaluation

In this section, I evaluate the performance of the proposed parallel approach with two different levels of expressivity of the reasoning request: positive recursive rules (experiment 1), and stratified negation rules (experiment 2). The first experiment is conducted on the reasoning request which contains positive recursive rules while the second one is dedicated for the request which contains stratified negation rules. In each experiment, I measure two metrics: latency and memory consumption, as in Section 4.3. The experiments were conducted on a machine with 24-core Intel(R) Xeon(R) 2.40 GHz and 96G RAM. I used Java 1.8 with heap size from 5GB to 20GB for C-SPARQL and Clingo 4.5.4 for the reasoners. For the rest of this section, I use R to refer to the C-ASP reasoner

presented in Chapter 4 (without parallelism), and PR to refer to the optimized version of C-ASP with the parallel approach that is detailed in Chapter 5 and 6.

6.3.1 Experiment 1: Recursive positive rules

For the experiment with recursive positive rules that are not supported by C_SPARQL, I compare PR against R and Jena reasoner[3] by using a widely used benchmark for reasoning systems, LUBM [175]. I select a different benchmark for this experiment (comparing to the experiment in Section 4.3) due to limitations regarding expressivity of rules in CityBench. In order to evaluate these engines, I create a reasoning request which contains a set of rules $\{r_1, ..., r_{15}\}$ as in Listing 6.1. This rule set includes 4 recursive rules over 15 rules. In terms of parallel optimization in PR, a given set of input predicates for this reasoning request is as follows:

$inpre(P)$ = {`rdf_type`, `uniben_worksFor`, `uniben_subOrganizationOf`, `uniben_teacherOf`, `uniben_teachingAssistantOf`, `uniben_takesCourse`, `uniben_publicationAuthor`, `uniben_advisor`}

The input dependency graph of this reasoning request is connected and is decomposed by Algorithm 4 in order to form a partitioning plan for PR with three subsets of $inpre(P)$ as follows:

pp_1 = {`rdf_type`, `uniben_worksFor`, `uniben_subOrganizationOf`},
pp_2 = {`uniben_takesCourse`, `uniben_teacherOf`, `uniben_advisor`, `uniben_publicationAuthor`},
pp_3 = {`uniben_takesCourse`, `uniben_teachingAssistantOf` }

I use Univ-Bench Artificial Data Generator[4] to generate and stream data to the engines. Due to the fact that the Jena reasoner does not support data stream processing, I run this experiment in two settings: static and streaming.

Static setting. In this setting, I evaluate PR, R and Jena reasoner with different sizes of input data from 5k to 100k (k=1000) triples. I trigger each engine 3 times per each input data size and take the average. Figure 6.4 and 6.5 show the effect over the latency and memory consumption with the increasing number of triples for three engines. A closer look at the results in Figure 6.4 reveals that PR outperforms R over subsequent increase from 10k to 100k (R can not process 60k and 100k triples). Compare

[3] https://jena.apache.org/documentation/inference/
[4] https://github.com/rvesse/lubm-uba

```
#input rdf_type/2 ;
#input uniben_worksFor/2 ;
#input uniben_subOrganizationOf/2 ;
#input uniben_teacherOf/2 ;
#input uniben_teachingAssistantOf/2 ;
#input uniben_takesCourse/2 ;
#input uniben_publicationAuthor/2 ;
#input uniben_advisor/2 ;


#prefix uniben : ⟨http://www.lehigh.edu/.../0401/univ-bench.owl#⟩;
#prefix rdf : ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩;

#from stream <http://lubm.org#universities> [time 3s step 2s];


(r1) rdf_type(X,"Profesor"):-rdf_type(X,"uniben_FullProfessor");
(r2) rdf_type(X,"Profesor"):-rdf_type(X,"uniben_AssociateProfessor");
(r3) rdf_type(X,"Profesor"):-rdf_type(X,"uniben_AssistantProfessor");
(r4) canBecomeDean(X,U):-rdf_type(X,"Professor"),uniben_worksFor(X,D),
                          uniben_subOrganizationOf(D,U);
(r5) canBecomeHeadOf(X,D):-uniben_worksFor(X,D);
(r6) commonResearchInterests(X,Y):- uniben_researchInterest(X,R),
                          uniben_researchInterest(Y,R);
(r7) commonPulication(X,Y):-uniben_publicationAuthor(P,X),
                          uniben_publicationAuthor(P,Y);
(r8) commonResearchInterests(X,Y):-commonPulication(X,Y);
(r9) uniben_teacherOf(Y,C):-commonResearchInterests(X,Y),
                          uniben_teacherOf(X,C);
(r10) commonResearchInterests(X,Y):- uniben_advisor(X,Z),
                          uniben_advisor(Y,Z);
(r11) canRequestRecommendationLetter(X,Z) :- uniben_advisor(X,Z);
(r12) canRequestRecommendationLetter(X,Z) :- teaches(Z,X);
(r13) teaches(X,Y):-uniben_teacherOf(X,C), uniben_takesCourse(Y,C);
(r14) teaches(X,Y):-uniben_teachingAssistantOf(X,C),
                          uniben_takesCourse(Y,C);
(r15) suggestAdvisor(X,Y):-teaches(Y,X);

#show canBecomeDean/2;
#show canBecomeHeadOf/2;
#show commonResearchInterests/2;
#show canRequestRecommendationLetter/2;
#show teaches/2;
#show suggestAdvisor/2;
```

Listing 6.1: The reasoning request with positive recursive rules inspired from LUBM

to Jena, PR is slightly slower when the input size is smaller than 30k. However, PR is considerably faster than Jena when the number of triples is bigger than 30k. When the input size increases from 60k to 100k triples, the latency of Jena increases sharply from 200 seconds to 750 seconds while PR's latency only increases slightly from 100 seconds to 200 seconds. This is an indication of the scalability of my approach over increasing size of the input. For the memory consumption, Figure 6.5 shows that all engines have increasing memory consumption issue but Jena seems to be better at memory management when increasing the number of input triples.

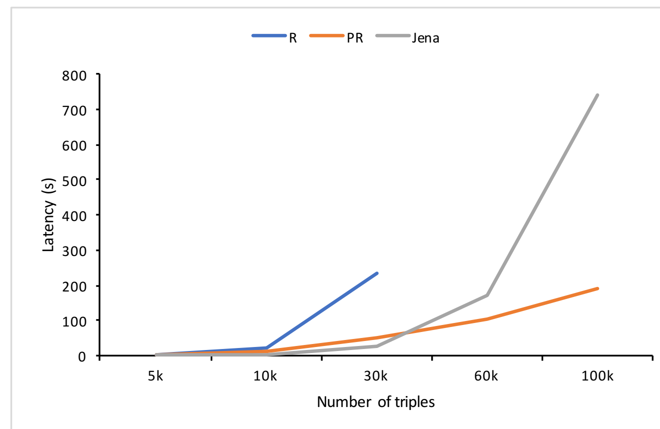


FIGURE 6.4: Latency (recursive rules with static setting)

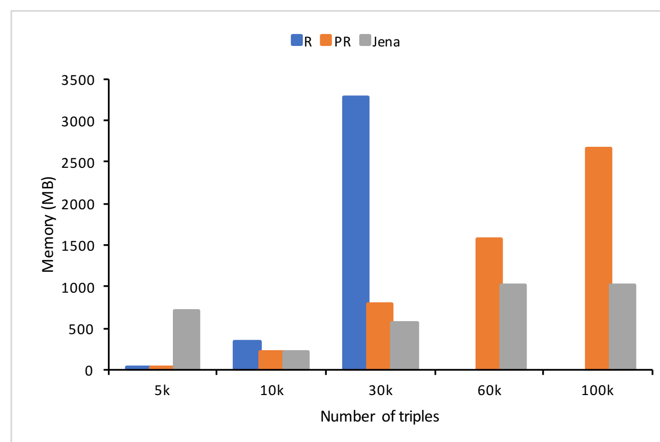


FIGURE 6.5: Memory consumption (recursive rules with static setting)

Streaming setting. In the streaming setting, I trigger PR and R by streaming triples for 10 minutes with various rates from 1k to 5k triples/second. I use the time-based window size of 3 seconds with the sliding step of 2 seconds. Figure 6.6 reports the latency observed from PR and R. It shows that PR performs as R at the streaming rate of 1k triples/second. The reason for this is that the number of input triples is small enough and the Clingo solver in the C-ASP reasoner does not suffer from exponential grounding. However, I observe the benefit of parallel optimization in PR at the streaming rates of 3k and 5k triples/second where PR performs much faster than R. In addition, the

latency of PR is more stable than the one of R during the 10-minute streaming. This means that my approach generates a smaller ground program and a smaller search space, speeding up both grounding and solving of the reasoner. For memory consumption that is illustrated in Figure 6.7, PR consumes slightly less memory than R. The figures also show that there is a considerable increase in memory consumption when streaming rate increases from 1k to 5k triples/seconds.

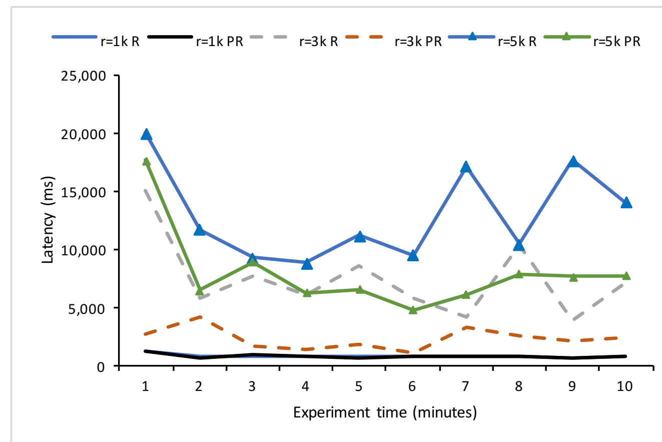


FIGURE 6.6: Latency (recursive rules with streaming setting)

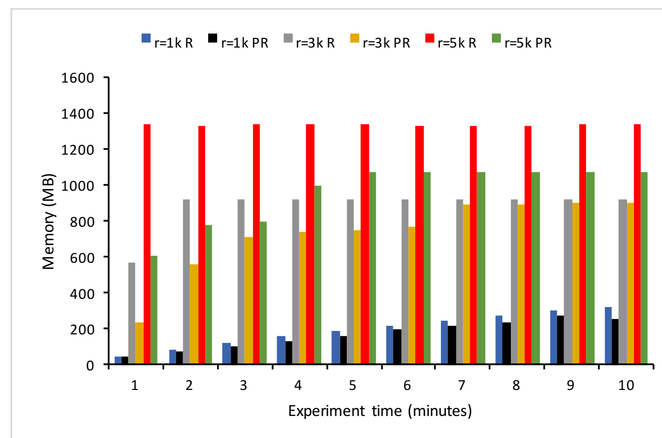


FIGURE 6.7: Memory consumption (recursive rules with streaming setting)

6.3.2 Experiment 2: Stratified negation rules

I now focus on a rule set which has stratified negations. I modify rules r_5, r_{12} and r_{15} in the reasoning request of Experiment 1 with 3 negation-as-failure atoms as in Listing 6.2. As a result, the experimental reasoning request has a rule set including 4 recursive rules and 3 negation-as-failure rules over 15 rules.

As the rules in Experiment 1 has been changed, the partitioning plan for the reasoner PR has been changed as follows:

```
(r'5)  canBecomeHeadOf(X,D):-uniben_worksFor(X,D), uniben_headOf(Z,D),
                                 not commonResearchInterests(X,Z);
(r'12) cannotRequestRecommendationLetter(X,Z):-teaches(Z,X),
                                 not uniben_advisor(X,Z);
(r'15) suggestAdvisor(X,Y):-teaches(Y,X),not uniben_advisor(X,Z);
```

Listing 6.2: Negation-as-failure rules

pp'_1 = {`uniben_takesCourse`, `uniben_advisor`, `uniben_teacherOf`, `uniben_publicationAuthor`, `uniben_teachingAssistantOf`},
pp'_2 = {`rdf_type`, `uniben_worksFor`, `uniben_subOrganizationOf`},
pp'_3 = {`uniben_advisor`, `uniben_worksFor`, `uniben_publicationAuthor`}

The number of partitions remains the same as in Experiment 1. However, the number of duplicated predicates (i.e., the predicate appears in more than two partitions) in this experiment is three (i.e., `uniben_advisor`, `uniben_publicationAuthor`, `uniben_worksFor`) while there is only one duplicated predicate in Experiment 1 (i.e., `uniben_takesCourse`).

I compare PR against R only since Jena reasoner does not support negation-as-failure. Similar to Experiment 1, I evaluate the same two engines for 10 minutes with various streaming rates from 1k to 5k triples/second. Figure 6.8 and Figure 6.9 illustrate a similar pattern in latency and memory consumption as observed in Experiment 1. PR has faster reasoning time at streaming rates 3k and 5k triples/second, but consumes slightly higher memory compared to R at 5k triples/seconds.

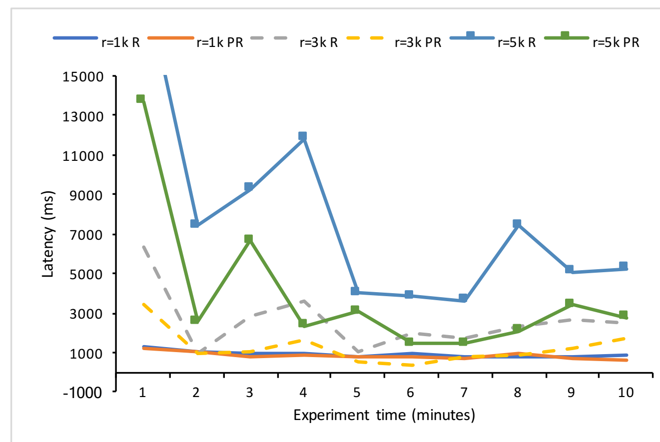


Figure 6.8: Latency (recursive and stratified negation rules)

6.4 Summary

Scalability is a key challenge for the applicability of reasoning techniques to rapidly changing information. This chapter constructs a method to parallelization of stream

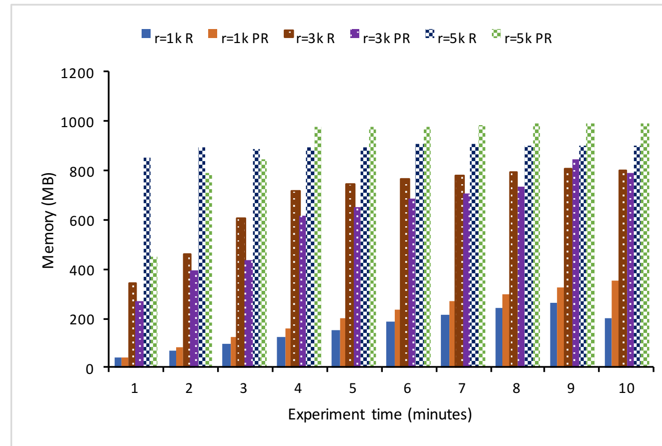


FIGURE 6.9: Memory consumption (recursive and stratified negation rules)

reasoning by input dependency analysis. The rationale for using input dependencies as guidance for parallel reasoning has been discussed. Building upon the work in Chapter 5, this chapter investigates parallel reasoning while taking into account the dependencies among input data. A greedy-based algorithm is proposed to identify a partitioning plan by decomposing such graph into subgraphs such that the number of common nodes among partitions is as small as possible.

An architecture of a parallel version of the C-ASP reasoner is defined in which the partitioning plan, realized at the designed time, will guide the reasoning process to split input data on-the-fly. Two new components are introduced, named partitioning handler and combining handler, take care of distributing input data to concurrent reasoners and combining final reasoning results, respectively. I implemented the proposed parallel approach as an extension of the C-ASP reasoner and provided a proof of correctness under the assumption that no recursion through negation is present in the rules, thus guaranteeing the uniqueness of the solution. Furthermore, I considered the different levels of expressivity that are supported by my prototype reasoner and conducted a detailed experimental evaluation by comparison with different systems based on their expressivity. This evaluation indicates that the parallel C-ASP reasoner not only has a competitive performance in comparison with existing systems but it also supports higher expressivity of reasoning tasks. This work is a demonstration that expressive reasoning is possible also in streaming environments, and it paves the way for investigating feasible solutions in this space. This study and related results have been published in [199].

Chapter 7

Use Cases and Prototypes

The validity and efficiency of the C-ASP reasoner are discussed in previous chapters (Chapters 4 - 6). This chapter validates the practicality of this reasoner by answering the following question:

> How is the C-ASP reasoner deployed in real-world scenarios and how does it meet the requirements of such applications?

The practicality of this reasoner is showcased by demonstrating its usage in two Smart City and Smart Enterprise applications: the contextual event filtering system and the IoT-enabled meeting management system. In particular, descriptions of such two applications and their functionalities are presented in order to illustrate the contribution of the C-ASP reasoner in those scenarios.

Section 7.1 discusses the use of the C-ASP reasoner in the contextual event filtering system. Section 7.2 details the use of the C-ASP reasoner for an IoT-enabled meeting management system. The chapter is summarized in Section 7.3.

7.1 Contextual Event Filtering System

This section reports on the use of the C-ASP reasoner as a key component within the CityPulse project [1]. The main objective of this project was to "*develop, build and test a distributed framework for the semantic discovery and processing of large-scale real-time IoT and relevant social data streams for knowledge extraction in a city environment*" [201]. The CityPulse framework integrates and processes large volumes of streaming

[1] http://www.ict-citypulse.eu

city data in a flexible and extensible way. Service and application creation is facilitated by open APIs that are exposed by CityPulse components. The CityPulse components are depicted in Figure 7.1 and can be divided into two main categories:

- Large-scale data stream processing modules (i.e., in the blue box): these include tools which allow the application developer to interact with heterogeneous and unreliable data sources from cities; these tools also allow discovering, summarizing and processing data streams.
- Adaptive decision support modules (i.e., in the red box): these include tools which can be used for making various recommendations based on the user context and the current status of the city.

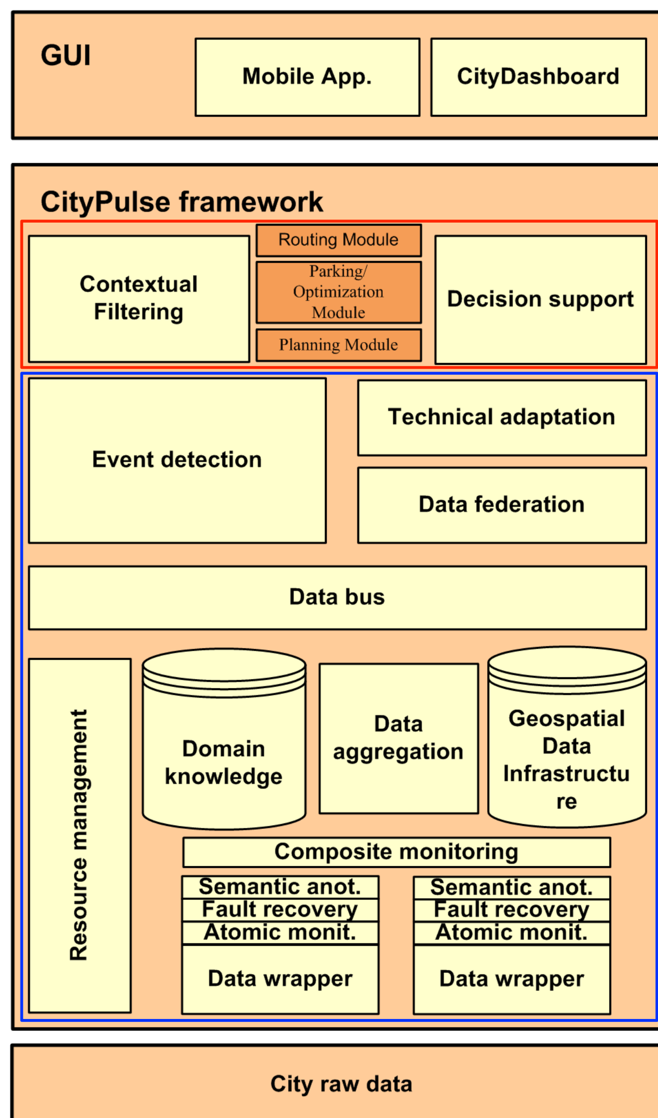


FIGURE 7.1: The components of CityPulse framework with their APIs [7]

Smart city applications in changing environments require to take into account user preferences and requirements, as well as dynamic contextual information represented by real-time events, in order to provide optimal decision support to the end user at any time. The event-driven adaptation and context-driven user-centrality of the CityPulse framework are materialized by a closed loop between the Contextual Filtering component, the user application, and the Decision Support component. The research presented in this thesis has been applied in building the Contextual Filtering component within the CityPulse framework, and in its interaction with the Decision Support component.

7.1.1 Contextual Filtering & Requirements

The main role of the Contextual Filtering component is to i) continuously identify and filter events that might affect the optimal result of the decision making task (performed by the Decision Support component) and ii) react to such changes in the real world by requesting Decision Support for the computation of new solutions when needed. This not only ensures that the selected solution provided to the user remains the best option when situations change, but it also empowers the CityPulse framework to automatically provide alternative decisions whenever the selected best decision is no longer the best for a particular situation.

The adaptive capability of identifying and reacting to unexpected events in a user-centric way relies on two aspects:

- A characterization of the user implicit and explicit context provided by the user application
- A stream of events provided by the Event Detection component.

The Contextual Filtering capability of the CityPulse framework is a good fit for the use of the C-ASP reasoner. This stems from the requirements of this component as follows:

- The context-awareness requirement is essential to easily capture sophisticated user context (including user requirements, preferences, events of interest and activities). The C-ASP reasoner meets this requirement through its expressive reasoning request language. Thus, it provides users the wide ability to express their complex context and seamlessly connect to event streams. In this way, users are able to receive their most wanted events regarding to their provided context.
- The scalability requirement is important for (near) real-time notification of a ranked list of critical events to users. The C-ASP reasoner can meet this requirement as it is efficient in terms of latency.

7.1.2 Implementation of C-ASP Reasoner for Contextual Filtering

Contextual Filtering subscribes to a subset of events among those provided by the Event Detection component. The types of events Contextual Filtering subscribes to are application-specific and are in part dependent on the domain (determined at design-time), and in part contextualized, depending on the specific reasoning task or user preferences specified by the user application (determined at run-time). For example, in the Context-aware Travel Planner application (see Section 7.1.3), traffic and weather conditions are relevant types of events that can be characterized at design time. However, when the user selects a route among those provided by Decision Support to go from a starting point A to an ending point B, only traffic and weather conditions in areas around that specific route are potentially relevant, and they can be augmented by the user interest in other types of events on the way (such as cultural or social gathering).

The occurrence of such relevant events is notified to the Contextual Filtering component by the Event Detection component, which provides additional metadata describing the event. Listing 7.1 illustrates an example of an annotated event about a traffic jam, as it is received by Contextual Filtering. Information about the user context can be gathered by Contextual Filtering in several ways: it can be either explicitly stated in the event request or in the user application (e.g. specifying events of interest), or it can be explicitly or implicitly acquired by identifying user's current activity (e.g. using the speed to detect that the user is in a car, or having the user specifying in the application what type of transportation he/she is using). With this information, Contextual Filtering is able to: (i) select, among the list of detected filtered events, the ones that are contextually relevant; and (ii) continuously rank their level of criticality to decide when an action is to be triggered.

The level of criticality of events is dynamically assessed by Contextual Filtering based on metrics such as the location-based correlation between a detected event and user's current location, or an explicit measure of how severe the event is (referred to as Event Level). The current implementation of Contextual Filtering uses a linear combination of these metrics. An application developer can configure such metrics and users can modify them in order to satisfy their own requirements. Listing 7.2 is a snapshot of the logic rules in the reasoning request used in Contextual Filtering. Rule r_1 filters out unrelated events. Rule r_2 generates sets of solutions containing one critical event each per solution, provided that the event is not expired. Rules $r_3 - r_7$ compute the criticality of an event. Based on this level of criticality, Contextual Filtering refers back to the user application that an action is required, which can be either generating a new request for Decision Support to automatically provide an alternative (better) solution, or informing the user about the critical event and let the user decide whether a new solution is needed.

```
@prefix geo:http://www.w3.org/2003/01/geo/wgs84_pos# .
@prefix sao:   <http://purl.oclc.org/NET/UNIS/sao/sao#> .
@prefix tl:    <http://purl.org/NET/c4dm/timeline.owl#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix prov:  <http://www.w3.org/ns/prov#> .
@prefix ec:    <http://purl.oclc.org/NET/UNIS/sao/ec#> .

sao:c2d69ca8-b404-4006-ad48-9317397251ab   a   ec:TrafficJam ;
        ec:hasSource       "SENSOR" ;
        sao:hasLevel       "1"^^xsd:long ;
        sao:hasLocation    [ a         geo:Instant ;
                             geo:lat   "56.17091325696965"^^xsd:double ;
                             geo:lon   "10.15728564169882"^^xsd:double
                           ] ;
        sao:hasType        ec:TransportationEvent ;
        tl:time            "2015-11-26T13:27:46.079Z"^^xsd:dateTime .
```

Listing 7.1: An example of an annotated event in Contextual Filtering

```
(r1) related_city_event(EventId):-filtering_event(Type),
             sao_hasType(EventId, Type).
(r2) 1 <= {selected_city_event(EventId) : related_city_event(EventId),
               not expired_event(EventId)} <= 1.
(r3) value(RankEleName, Value) :- selected_city_event(EventId),
               ranking_city_event_data(EventId, RankEleName, Value).
(r4) value_with_ranking_type(RankingElementName, M) :- M = Value*Int,
               value(RankingElementName,Value),
               ranking_multiplier(RankingElementName,Int).
(r5) sum(C) :- #sum{Value: value_with_ranking_type(_,Value)}
(r6) criticality(C) :- C = M/100, sum(M).
(r7) critical_city_event(EventId,C):- selected_city_event(EventId),
               criticality(C).
```

Listing 7.2: Rules for contextual filtering of events with ranking through linear combination

Two prototype applications have been collaboratively developed by the teams involved in CityPulse, namely Context-aware Parking and Travel Planner, using live data from the city of Aarhus in Denmark. The descriptions of these two applications can be found at http://www.ict-citypulse.eu/scenarios/(i.e., scenario 1 and 2). Both applications aim at providing parking/travel-planning solutions, which go beyond state of the art solutions by allowing users to provide multidimensional requirements and preferences such as air quality, traffic conditions and parking availability. In this way, a user receives parking and route recommendations based on the current situation in the city and the user context. Both applications are built as Client/Server applications. The server side hosts an instance of the CityPulse smart city framework. The client side is a mobile application on Android in both cases.

7.1.3 Context-aware Travel Planner

Tony needs to travel from home to work. When he starts the client application, the city map is displayed with options to input his start and end points for his journey. Different means of transportation are generally available to him such as walking, biking, or car as shown in Figure 7.2. In addition, Tony can also specify his constraints and preferences on the recommended routes such as fastest, shortest, or cleanest as illustrated in Figure 7.3. His request is first processed by the Decision Support component which recommends all possible optimal routes on the map (see Figure 7.4). Next, Tony needs to choose one of such routes (i.e., the blue route in Figure 7.4). When he starts the navigation, Contextual Filtering will subscribe to city events related to that route from the Event Detection component. Contextual Filtering should notify Tony any critical event happen on his route which may affect his journey, such as a traffic jam, as shown in Figure 7.5. Recalculation of his chosen route can happen depending on his decision over the reported critical event.

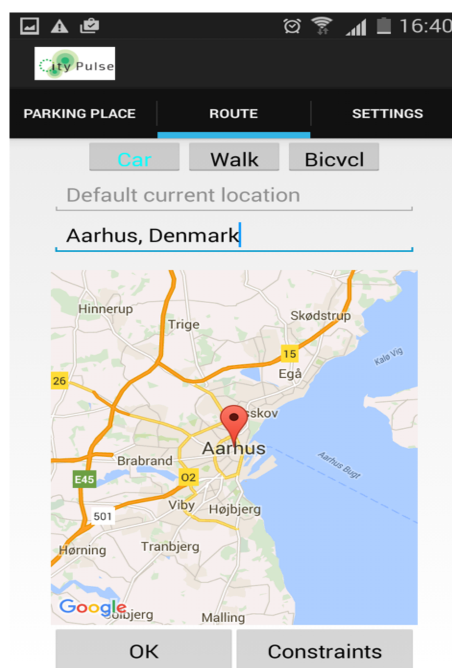


FIGURE 7.2: Route selection

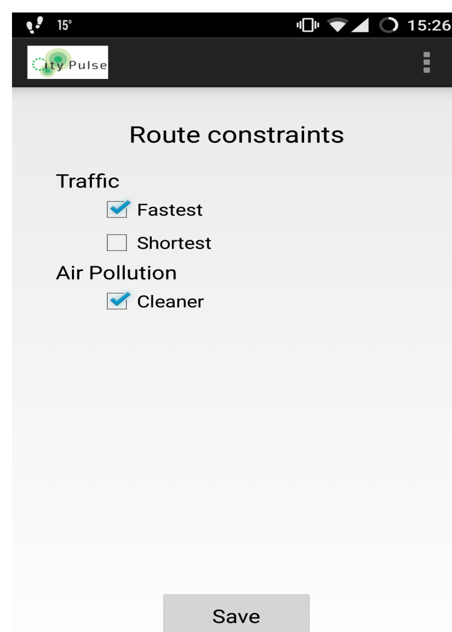


FIGURE 7.3: Route constraints

7.1.4 Context-aware Parking Planner

Mary is having a hard time finding a public parking space. The city is increasingly reducing the number of parking spaces per unit (e.g. apartments), and the difficulty of finding a parking space means Mary has to drive around to look for available parking spots. This is both annoying for her and has a negative environmental impact (increased

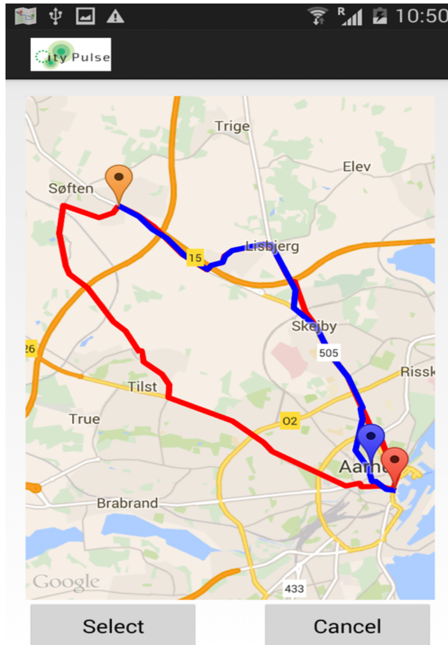


FIGURE 7.4: Optimal routes

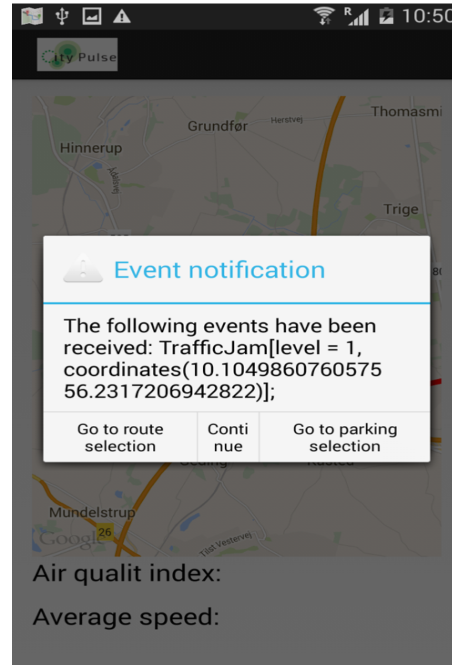


FIGURE 7.5: Ciritcal Event

pollution and noise). By using the Context-aware Parking Planner, Mary can specify an area within which to find a parking spaces, as shown in Figure 7.6. This area is a circle which can be centered by the destination point or by her current location (the blue circle on the map in Figure 7.6) and a fixed radius (i.e., 1000 meters). In addition, she can add her preferences by choosing parking spaces with the cheapest toll or shortest walking distance to the destination or both (see Figure 7.7). Decision Support provides optimal parking spaces which satisfy Mary's requirements as illustrated in Figure 7.8. When Mary chooses one of the parking spaces, Contextual Filtering continuously subscribes to parking events in that parking space and will notify her as soon as possible when that parking space is full.

7.2 IoT-enabled Meeting Management System

Enterprise communication systems are designed in such a way to maximize the efficiency of communication and collaboration within the enterprise. In this domain, mobile users have the potential to produce a lot of dynamic sensory input that can be used for the next generation of mobile enterprise collaboration, with great potentials for better user experience. The IoT-enabled Meeting Management System (IoT-MMS) is a research project developed with Cisco Systems. This project aims at enabling IoT in the smart

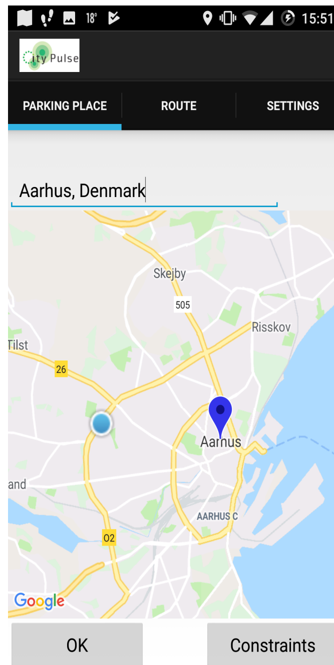


Figure 7.6: Parking selection

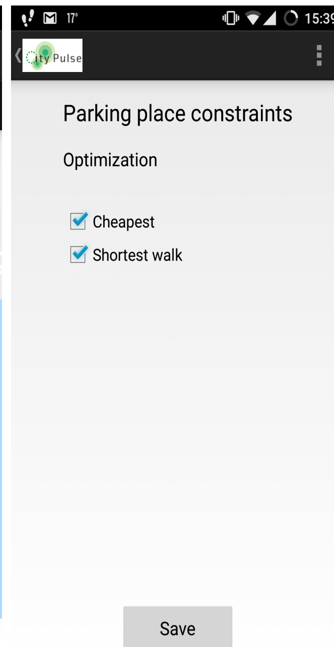


Figure 7.7: Parking constraint

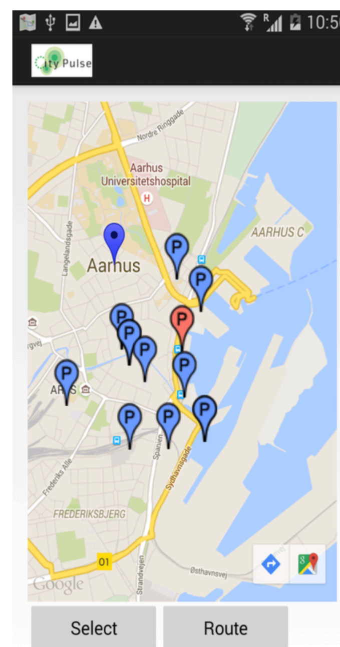


Figure 7.8: Optimal parking spaces

enterprise, through a Linked Data infrastructure for networking, managing and reasoning upon heterogeneous, distributed and continuously changing data streams. In this section, I present how the work in this thesis has been used as a part of IoT-MMS.

7.2.1 Motivating Scenario

Alice is hosting an online meeting for her company *FictionDynamic*. The meeting is planned to be held in Meeting Room B at 11:00 am. Bob and Charlie will be attending the meeting while they are on the move, thus their availability and ability to participate in the meeting in various ways is dynamically changing. IoT-MMS enables:

(i) automatic on-the-fly semantic enrichment of IoT information related to the meeting attendees,

(ii) communication of such richer information to the participants via their IoT-MMS clients through a panel showing IoT values and related user capabilities (e.g. ability to hear properly, share a screen, type, talk),

(iii) use of such rich information to improve user experience and optimize meeting management on-the-fly.

The integration of a web-based IoT-MMS with sensory input and enterprise data such as attendees details, calendars and agenda items makes it possible to characterize and manage the following aspects in a flexible and inter-operable way:

- updating (enabling or disabling) users capabilities based on IoT input (via sensors' visualization and interpretation, semantic integration and stream query processing);
- managing agenda items, including users assigned to a particular item and capability requirements for that item via declarative logic rules;
- dynamically verifying privacy-constraints on agenda items based on location and context.

Contextual information can be explicitly available as the user specifies whether he/she is in a public or private place, but they can also be detected by specific simple event detection logic via query processing in the framework. However, the ability to reason about constraints and planning (e.g. for rescheduling agenda items) needs to be handled by more expressive rules, and are handled by the stream reasoning component. In order to further illustrate the multiplicity of situations that can occur, a few instances of the motivating scenario are characterized as follows:

- **Capability-aware participation.** Charles is on the move and gets notified about a last-minute meeting to be held with a customer on a specific product with the development team. As the head of the team, Charles needs to be mainly a listener, and intervene only if needed. Information about Charles capabilities as a listener (e.g. level of attention while driving or while in a noisy area) are collected and interpreted through inertial and environmental sensors via his Android phone. Such capabilities are continuously monitored and updated in the IoT-MMS web clients for the meeting host to see so that when the customer is addressing Charles directly, he knows whether the issue should be answered right away or later. Charles is also notified when his intervention is requested and what capabilities are needed to participate (e.g. quieter area, ability to type and so on). All this information is stored and associated with a synthetic representation of the meeting minutes for future reference.
- **Agenda re-shuffling.** Alice needs automatic support to dynamically re-order certain agenda items since most of the participants are on the move. IoT-MMS continuously assesses attendees' capabilities and matches them with agenda items requirements. Some agenda items are very sensitive with respect to privacy; therefore location-based constraints are associated to those items in order to prevent a

meeting participant to disclose sensitive information in public and crowded spaces like a trains or open-spaces airport lounges.

The capabilities of the C-ASP reasoner are well suited to handle the functionalities required by IoT-MMS, for the following reasons:

- the ability to reason about constraints, planning, and preferences can be handled by the expressive power of C-ASP.
- the ability to notify in (near) real-time about user's current participation, as well as new agenda schedule for all people in the meeting, can be met by the efficiency in terms of throughput and latency of C-ASP.

7.2.2 IoT-MMS Architecture

The conceptual architecture of IoT-MMS is illustrated in Figure 7.9 and can be divided into three main layers:

- Data acquisition and semantic annotation layer: is mainly responsible for acquiring sensor data from mobile devices and performing semantic annotation of the acquired data using information model which proposed in the project.
- Stream processing and reasoning layer: is responsible for the discovery of relevant sensor streams, and performs reasoning over those streams to produce actionable knowledge.
- Application layer: represents the class of enterprise applications that can benefit from IoT intelligence.

The work in this thesis has been mainly applied in the development of the SR component in the stream processing and reasoning layer in IoT-MMS.

7.2.3 Stream Reasoning in IoT-MMS

The SR component performs complex reasoning over IoT streams produced by the stream query processing component. Its conceptual architecture is provided in Figure 7.10.

The stream reasoning functionality is realized by three main components, namely Event Detection, User Reasoner and Meeting Reasoner.

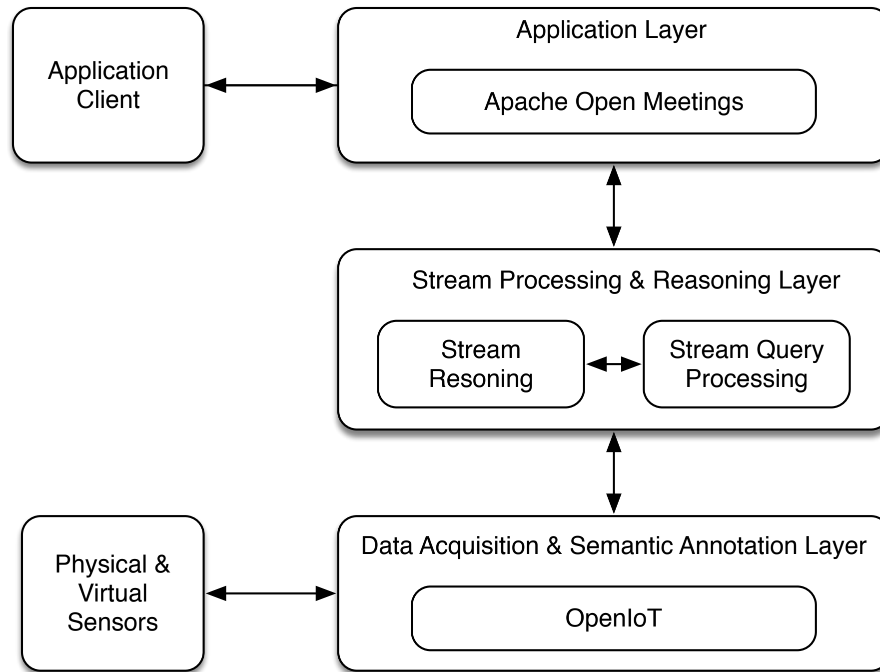


FIGURE 7.9: IoT-enabled communication system architecture [8]

Event Detection.This component integrates the dynamic sensor data streams produced by the stream query processing component (dynamic), with background knowledge about the meeting (static), such as users' calendar, meeting duration, agenda schedule and thresholds set for the different capabilities. This information is used to detect triggering events which are then used to trigger the User Reasoner or the Meeting Reasoner when needed.

In details, Event Detection detects event triggers by keeping track of sensors' data produced by the stream query processing component. It is implemented as a C-ASP reasoner where input data includes: a sensor observation stream (i.e., `sensor_observation(Sensor, SensorValue, User)`) provided by the stream query processing component, and information about discussing agenda in the meeting (i.e., `current_agenda(Agenda)`) provided by Meeting Reasoner. A snapshot of the reasoning request of Event Detection is shown in Listing 7.3. The first three rules detect changes in sensor capabilities being within or going out of a specific range while the last positive rule encodes a check on the sensitivity of items in the agenda.

User Reasoner. A separate User Reasoner component is initiated for each user when he logs into the meeting, and is stopped at logout. This component is responsible for reasoning about the dynamic status of user capabilities and their changes according to the user context. User Reasoner triggers updates on user's capabilities whenever it receives an event from the Event Detection component that relates the user capabilities with changed values in related sensors thresholds.

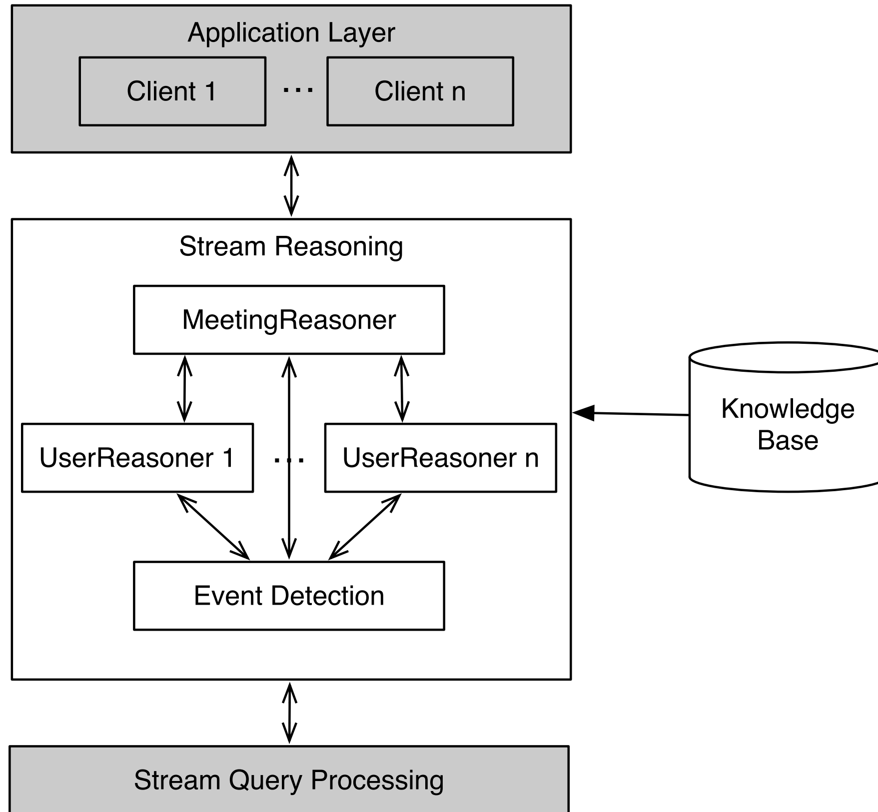


FIGURE 7.10: Stream Reasoning Layer Architecture

```
(r1)in_range(User,Sensor):-sensor_observation(Sensor,SensorValue,User)
    , sensor_threshold(Sensor, LowerBound,UpperBound), SensorValue <
    UpperBound, SensorValue > LowerBound.

(r2)in_range_event(User, Sensor):-in_range(User, Sensor),
    previous_out_range(User, Sensor).

(r3) out_range_event(User, Sensor) :- not in_range(User, Sensor),
    previous_in_range(User, Sensor).

(r4) current_agenda_event(Agenda, sensitive) :- current_agenda(Agenda)
    , agenda_property(Agenda, Property), Property == sensitive.
```

LISTING 7.3: Rules of Event Detection

For example, a change in the user position w.r.t. to the meeting (in range or out of range) is detected by Event Detection and encoded as the propositional predicates `out_range_event(User, Sensor)` or `in_range_event(User, Sensor)`, which are sent as input to User Reasoner. This triggers rules r_1 or r_2 in Listing 7.4. In a similar way, if the agenda being discussed is sensitive, Event Detection will produce an event encoded as the propositional predicate `current_agenda_event(Agenda, sensitive)` and send it to User Reasoner to verify whether the current user context triggers any meeting-related action (i.e., rule r_3 in Listing 7.4).

```
(r1) user_capability_status_event(User, Capability, on):-
    in_range_event(User, Sensor), affect(Sensor, Capability),
    user_capability_status(User, Capability, off).

(r2) user_capability_status_event(User, Capability, off):-
    out_range_event(User, Sensor), affect(Sensor, Capability),
    user_capability_status(User, Capability, on).

(r3) user_capability_status_event(User, talking, off):-
    current_agenda_event(Agenda, sensitive), user_place(User, public),
     user_capability_status(User, talking, on).
```

Listing 7.4: Rules of User Reasoner

Listing 7.4 shows a sample excerpt of rules for User Reasoner: the first two rules turn certain capabilities on or off based on detected events on range values of any of the sensors associated to the related capability; the third rule marks the talking capability as disabled if Event Detection has identified the current agenda item as sensitive.

Meeting Reasoner. This component encodes rules related to the meeting itself and normally involves more than one user, including agenda re-shuffling and meeting rescheduling, and it can be triggered either as a result of a change in user capabilities (e.g. a user is no longer able to talk therefore some agenda items might need to be swapped) or as a result of specific events detected by Event Detection (e.g. a key user has not logged in and is far from the venue, therefore, the meeting might need to be rescheduled or postponed).

Meeting Reasoner is in charge of providing a suitable reschedule on-the-fly for the agenda items based on user calendars and constraints or dependencies that might be present in the order of agenda items. As an example, when a user is not available for a particular agenda item assigned to him, Meeting Reasoner will receive an event encoded as a propositional predicate `could_not_attend(User,Item)` and reason about options to re-shuffle the agenda items. The corresponding planning rules used to re-order agenda items in this case can be found in Listing 7.5. The first rule generates all possible orders of agenda items, while the second and third rules (referred to as *consistency constraints*) check each solution to eliminate those where required users can not attend.

The required actions identified as a result of the inference performed by User Reasoner or Meeting Reasoner are then sent as notifications to the Application layer which will take care of the actuation part.

7.2.4 IoT-MMS Application Interface

In this section, I describe the implementation of the application interface of IoT-MMS.

```
(r1)1{new_agenda_order(Item,Order):order(Order)}1 :-
                    agenda_item(Item).

(r2):- agenda_speaker(Item,User), could_not_attend(User,Item),
    agenda_order(Item, Order), new_agenda_order(Item,Order).

(r3):- new_agenda_order(Item1,Order) , new_agenda_order(Item2,Order),
    Item1 != Item2.
```

LISTING 7.5: Rules of Meeting Reasoner

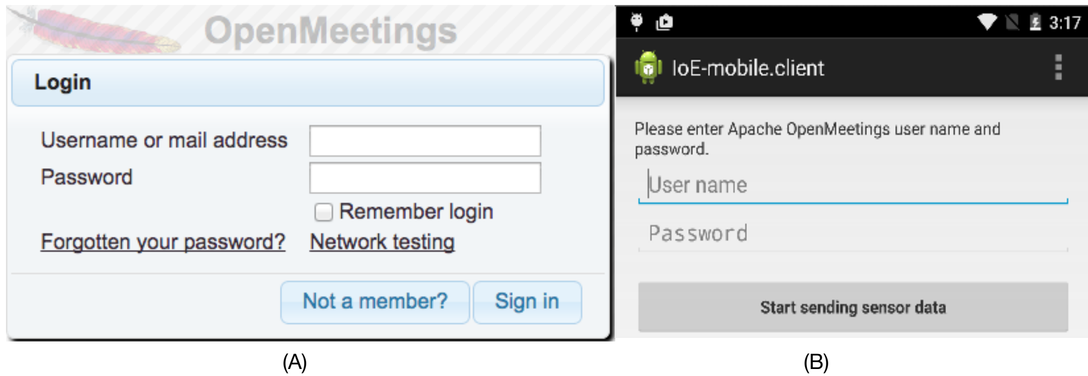


FIGURE 7.11: Application login interfaces

7.2.4.1 Android Application and User Login

This section describes the design of the Android application which is responsible for registering sensors to the OpenIoT platform and sending sensor information as continuous streams. Before entering a web conference room through the OpenMeetings (OM) application client, every user must login through the Android application using the same credentials used for OM. The Android application uses the login web services of OM for maintaining the same session for both HTTP clients. Figure 7.11(A) shows the OM login interface while Figure 7.11(B) shows the Android login interface.

After signing in, the Android application shows the list of available sensors of the mobile device. Figure 7.12 shows the list of sensors of a mobile device. After selecting the sensor input to be sent, those sensors will be registered in the OpenIoT platform and the Android application will start sending sensor data to the IoT-MMS framework.

7.2.4.2 From Meeting Creation to Notification

In describing the scenario design mentioned in Section 7.2.1, this section focuses on meeting event creation functionality, which enables to manage multiple agenda items (e.g., multiple talks, breaks) and meeting rooms description, and helps controlling the

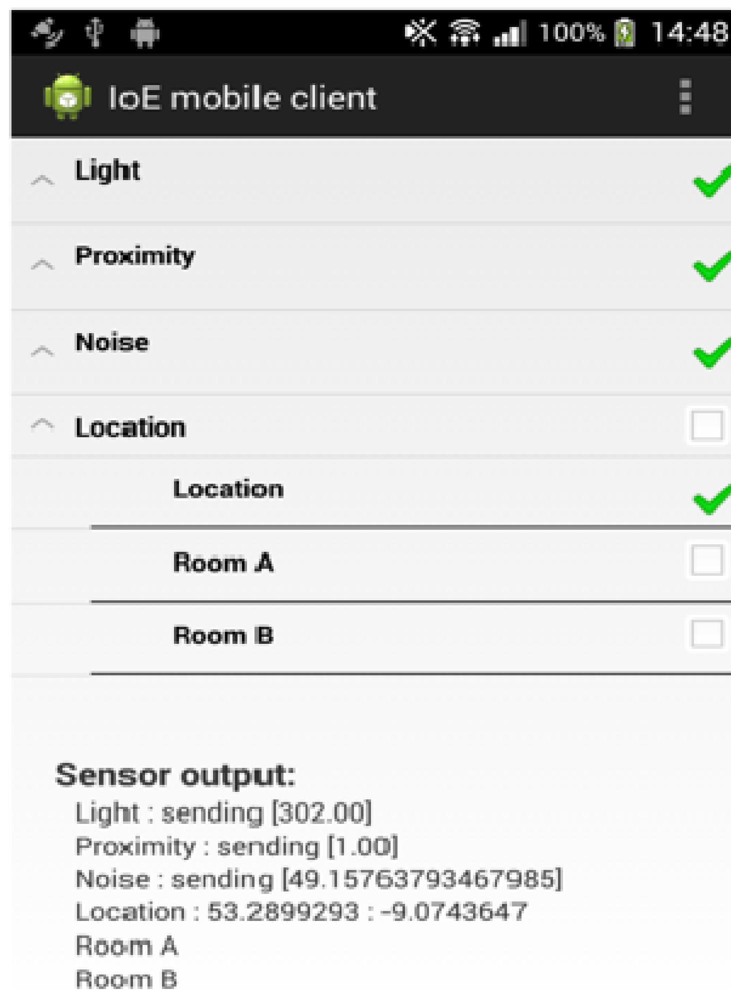


FIGURE 7.12: Available sensor list

capabilities of attendees when the meeting is running using both real-time and static information.

Creating Meeting Event: each meeting contains event information including start-time, end-time, location, list of attendees and meeting agenda, all stored in the triple store. Figure 7.13 shows that Alice has created a meeting event located in Room B and has two more attendees including Bob and Charlie. All the attendees will be notified of the event information upon creation through emails. Figure 7.14 shows how the host has set two agenda items of type *talk*, one is for 30 minutes and another one is 25 minutes. The organizer of that event sets an explicit threshold for each sensor capability (noise, light, proximity). Thresholds are configurable during the meeting based on the room condition. Figure 7.15 shows three sensor capabilities and thresholds where 80 db is for the noise sensor, 0 is for proximity and 150 lux is for the light sensor. Such thresholds can be auto-generated or set by default at meeting creation to help the meeting host in configuring the meeting.

FIGURE 7.13: Create new meeting event from OM

FIGURE 7.14: Adding agendas for meeting event

FIGURE 7.15: Setting sensor thresholds for the event

Starting Meeting Event: when a user joins a meeting in OM, each client registers as a IoT-enabled client through a web socket connection to the WebSocket server running the reasoning component. Each client sends JSON objects to the server containing user and meeting IDs. After connecting to the server, clients are ready to get IoT notification of changes in user's capabilities also as JSON object. In the meeting room interface all the IoT capabilities are represented as icons. Figure 7.16 describes the IoT capabilities

FIGURE 7.16: IoT panel in meeting room for each user

FIGURE 7.17: User's IoT Capabilities Notification

of a user which has five icons including capability of speak, listen, read, chat and user presence in the room from left to right respectively. Each icons has two states shown in Figure 7.16(A) and 7.16(B) representing enabled or disabled capabilities, and a user's presence icons indicating whether the user is physically in the room or remotely joining. When the OM application gets IoT notifications that the capability state of a user have changed, a notification message pops up and when possible and make an actuation action is executed such as muting or unmuting the microphone of that user.

IoT Notification and Action: the OM application gets real-time information from all the users who joins an OM meeting and sends IoT notification if there is any change in user's capabilities. After getting the notification, the OM server triggers an action for that user based on the notification. Different situations are illustrated in a client panel and example of pop-up messages in the following screenshot figures: Figure 7.17(A) shows that Alice has reached the meeting room and has all capabilities enabled; Figure 7.17(B) shows that there are three attendees in the room where two users are outside the room location and Bob cannot share his screen, listen or speak but he can read and type because he is in a noisy area; Figure 7.17(C) shows that Charlie is inside the meeting room and has all the capabilities enabled; and Figure 7.17(D) shows that Bob cannot read and type because he is driving.

After getting the IoT notification, the OM application notifies the host (or all the attendees if required) and perform some actions in form of pop-up warning or actuations like acting on audio/video or controlling screen sharing, chatting, white board drawing, file uploading.

7.3 Summary

This chapter provides evidence on how the C-ASP reasoner is deployed in two real-world scenarios and how it meets requirements of such scenarios. The usage of C-ASP is introduced via two systems, namely contextual event filtering and IoT-enabled meeting management system. The former aims at continuously identifying and filtering critical events that might affect the decision making of users in Smart City applications. Two prototypes of the first system are implemented for users who are traveling in Aarhus, Denmark: a travel planner and a parking planner. The latter investigates on enhancing user experiences in online meetings on-the-move by using mobile sensors. One prototype of this system is implemented which has abilities to continuously update the current status of attendees, manage agenda items, and support the organizer in re-plan the meeting on-the-fly. The prototypes implemented in the first scenario are open-source and can be downloaded at `https://github.com/CityPulse/Decision-Support-and-Contextual-Filtering`. The third-party developers can re-use those components to create new Smart City applications. Since those components (including ones in the second scenario) are implemented based on C-ASP reasoner, it requires for the developers to have a Clingo solver installed in the system and an ability to express user's requirements in C-ASP reasoning requests using C-ASP language. Related results of this chapter have been published in [7, 8, 202].

Chapter 8

Conclusion

The ability to perform complex reasoning over semantic data streams has recently become an important area of research in the SW community. Most of the existing systems for processing RDF streams extend SPARQL 1.1 language and have limitations in capturing sophisticated user requirements and dealing with complex reasoning tasks. Moreover, the exponential growth in the availability of streaming data on the Web has seriously hindered the applicability of state-of-the-art expressive reasoners, limiting their applicability to process streaming information in a scalable way. This thesis explores the trade-off between scalability and expressivity for a continuous reasoner over SW data streams and the applicability of such reasoner in Smart City and Smart Enterprise applications. On the one hand, I propose a stream reasoner in which its expressivity is enhanced from advances in both ASP and RSP research. On the other hand, I investigate on the parallel-based optimization method to scale up the proposed expressive reasoner.

8.1 Contributions

In this section, I review the research questions presented in Chapter 1 and how my contributions address them and advance the state-of-the-art.

The first question *RQ1*, stated: *How to enable complex reasoning based on ASP over RDF data streams?* My investigation in Chapter 4 resulted in the C-ASP language and the C-ASP engine.

- The C-ASP language is an ASP-based language for continuous reasoning over RDF streams. This language integrates the stable model semantics of ASP with RDF streams, windowing operators and streaming operators in order to achieve

expressive stream reasoning. C-ASP shifts the evaluation semantics of ASP from one time to continuous. In other words, C-ASP continuously produces the results of complex reasoning (solutions) as a stream.

- The C-ASP engine implements the C-ASP language which allows users to express their reasoning task in the form of a continuous reasoning request. This request is registered once into the C-ASP engine and continuously evaluated whenever new data arrives.

C-ASP seamlessly provides flexible ways of combining RSP and ASP reasoning. Moreover, it leverages the full expressive power of ASP. C-ASP represents a step forward from state-of-the-art RSP engines in capturing complex requirements and preferences of users, including optimization and common-sense reasoning. In addition, the experiments in Chapter 4 show that the implemented C-ASP engine outperforms the state-of-the-art RSP engine C-SPARQL for tasks at the same level of complexity.

Next, the thesis focuses on tackling research question *RQ2*: *How to scale up the reasoning process over RDF data streams under stable model semantics of ASP ?* Given the C-ASP engine proposed in Chapter 4, I improved the scalability of C-ASP while investigating on the two following sub-questions:

- *RQ2.1*: *Which information is relevant to the reasoning process and how can it be used to optimize the reasoning performance?* The relevant features in the streaming setting which can be used to optimize the C-ASP reasoning performance are identified in Chapter 5. The first part of this chapter reports interesting findings about reasoning sequentially over partitions of an input window: the empirical evaluation states that the reasoning time of C-ASP can be reduced when reasoning sequentially over partitions of the input window. This finding is under the assumption that the reasoning time of C-ASP is monotonically increasing. Moreover, in order to guarantee the correctness of the reasoning results, all data points in the input window are assumed to be independent. The second part of this chapter removes the independence assumption which is unrealistic in real settings, and considers different ways input data depends on each other. I proposed a new method to analyze such dependencies between input data items based on the C-ASP reasoning request registered by the user. The concept of input dependency graph is introduced to capture the dependencies among input data in a window. This input dependency graph is defined and constructed based on the extension of the notion of dependency graph in ASP. The resulting contributions have been published and presented in Web Reasoning and Rule Systems (2015) [197] and Data Engineering (2017) [198].

- *RQ2.2*: *How to use such information found in RQ2.1 efficiently to speed up the reasoning process without losing the correctness of reasoning results?* The input dependency graph proposed in Chapter 5 has been used to improve the reasoning process of the C-ASP engine as it enabled data parallelism. My investigation, under the circumstance of ASP with stratified negation, leads to the definition of the partitioning plan, described in Chapter 6. The partitioning plan is the result of decomposing the input dependency graph in such a way that all of dependencies among data items in the input window are maintained. This partitioning plan guides the C-ASP engine on how to split the input window into smaller chunks on-the-fly. In this way, C-ASP can reason in parallel over portions of the input window and reduce the reasoning time. Moreover, the final results of the parallel reasoning under the guidance of the partitioning plan is proved to be semantically correct. The contribution has been published and presented in Semantic Web (2018) [199].

Additional contributions of this thesis are presented in Chapter 7, where the practicality of the proposed C-ASP reasoner is validated in the domains of Smart City and Smart Enterprise. Two different components have been developed, namely the contextual event filtering system and the IoT-enabled meeting management system, respectively. Three prototypes of those components are implemented. This shows the evidence on how the C-ASP reasoner is deployed in real-world scenarios and how it meets requirements of such scenarios. Related results of this contribution have been published as follows: IEEE Access 4 (2016) [7], Innovations in Clouds, Internet and Networks (2017) [202], and Journal of Web Semantics (2017) [8].

8.2 Limitations

In this section I want to highlight some limitations that would be worth investigating in future research.

Concurrent evaluation. In *RQ1*, I proposed the C-ASP language and engine to capture users' sophisticated requirements and perform complex reasoning over SW data streams. The current implementation of C-ASP is limited to the case where multiple reasoning requests are registered and evaluated concurrently. For now, each registered reasoning request is processed by an instance of the C-ASP engine separately. The inability of the C-ASP engine to share resources among different concurrent reasoning requests can cause the performance bottleneck.

Granularity of input dependencies. The partitioning plan created from the input dependency graph is responsible for guiding the C-ASP engine in partitioning input data on-the-fly and reasoning in parallel. The creation of this partitioning plan currently takes place at design time and is mainly based on predicate-level dependencies. Creating on-the-fly partitioning plan would make it possible to consider extra information such as atom-level dependencies and their distribution in an input window. This may have impact on the scalability of the reasoning process but it requires to adapt the way the dependency graph is built and maintained, and it is currently not supported.

Reasoning expressivity. The reasoning time of C-ASP is reduced thanks to the optimization based on parallelism. However, this parallel reasoning is currently restricted to ASP rules with stratified negation in the reasoning request. This restriction is necessary to guarantee the correctness of the parallel reasoning results. Relaxing this assumption and allowing parallel reasoning with recursive negation or disjunctive ASP rules can push the expressivity of C-ASP and enable more complex reasoning, but careful consideration should be given on how to maintain correctness in this case.

Comparative evaluation. The experiments in this thesis do not compare the performance of C-ASP with other existing ASP-based reasoners such as Laser [148] or Ticker [147]. These reasoners do not support RDF data streams as input. In addition, their optimization follows the approach of incremental reasoning. Both incremental reasoning and parallel reasoning help to improve the reasoning process and they are not exclusive. Instead, these two approaches are complementary to each other in that the parallel approach can be applied first to partition input data and enable concurrent reasoners to run in parallel, and then the incremental approach can be applied within each reasoner instance. I have not explored this combination in this thesis.

8.3 Future Work

The results presented in this thesis open up several possibilities for future work. In the following, I discuss future directions of research on which this thesis can be extended.

The first endeavor is to investigate the reasoning process of the C-ASP engine in the presence of multiple reasoning requests. Having concurrent reasoning requests may enable synergies among common resources which are shared by those requests such as input data streams, background knowledge, or rules. The ability to share resources while evaluating concurrent reasoning requests can lead to a significant reduction in latency and increase in performance. Resource sharing and load balancing techniques in data processing (e.g., [203–205]) can be leveraged to address this challenge. However,

for an expressive stream reasoning system more complicated reasoning semantics need to be incorporated and the dynamicity of data buffers and input streams have to be considered.

The C-ASP engine supports full ASP capabilities to reason over RDF streams and static knowledge. However, when it comes to parallelism, the engine proposed and implemented in this thesis restricts the expressivity of its reasoning requests to ASP with stratified negation so that correctness of reasoning results can be maintained. The extension of C-ASP to be able to perform parallel reasoning with full ASP expressivity (i.e., recursive negation and disjunctive rules) while maintaining correctness of the reasoning results could open new opportunities in terms of application scenarios. In order to enable this, the input dependency graph needs to be updated with new types of edges to specify the recursive negation as well as disjunctive dependencies. Moreover, the combination handler also needs to be modified to ensure that the results of parallel reasoning are semantically correct.

Another direction for future investigation is to consider how different partitioning plans affect the performance of the parallel C-ASP reasoner. The current solution for creating the partitioning plan is extended from the greedy algorithm, in which the number of partitions is determined by randomly traversing edges in the input dependency graph. Different heuristics to create the partitions may have different impact on the reasoning process. The distribution of ground atoms across the different predicates could be a good information to design heuristics for creating better partitions. This could also inform the current partitioning function so that the splitting process does not rely on predicate-level analysis only.

Inspired by related work on incremental reasoning over streams with ASP, it would also be interesting to study how to combine the result of this thesis with incremental techniques to achieve better scalability for the C-ASP reasoner. Such direction would need to consider the parallel approach as a first step to reduce the amount of input data that is fetched into the stream reasoner, and then within each instance of the reasoner, consider the incremental mechanism to reduce the cost of recomputing solutions over sliding windows. In this way, efficient incremental reasoning techniques recently implemented in systems such as Laser [148] or Ticker [147] can be leveraged for a more advanced solution to expressive stream reasoning on RDF streams based on ASP.

The parallel reasoning approach in this thesis is mainly based on the analysis of dependencies between input data. This analysis considers not only the structure of a given rule set (a C-ASP reasoning request) but also the presence of data (input predicates) to create the input dependency graph. This idea can be generally applied to a broader stream reasoning field for enabling parallel reasoning. However, the method to construct

the input dependency graph needs to be customized with respect to the semantics of the supported logical reasoning.

Bibliography

[1] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3):1–238, 2012.

[2] Francesco Calimeri, Simona Perri, and Francesco Ricca. Experimenting with parallelism for the instantiation of ASP programs. *Journal of Algorithms*, 63(1):34–54, 2008.

[3] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal—The International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[5] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In *International Conference on Web Reasoning and Rule Systems*, pages 42–57. Springer, 2010.

[6] Alessandra Mileo. Web stream reasoning: From data streams to actionable knowledge. In *Reasoning Web International Summer School*, pages 75–87. Springer, 2015.

[7] Dan Puiu, Payam Barnaghi, Ralf Tönjes, Daniel Kümper, Muhammad Intizar Ali, Alessandra Mileo, Josiane Xavier Parreira, Marten Fischer, Sefki Kolozali, Nazli Farajidavar, et al. Citypulse: Large scale data analytics framework for smart cities. *IEEE Access*, 4:1086–1108, 2016.

[8] Muhammad Intizar Ali, Naomi Ono, Mahedi Kaysar, Zia Ush Shamszaman, Thu-Le Pham, Feng Gao, Keith Griffin, and Alessandra Mileo. Real-time data analytics and event detection for IoT-enabled communication systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 42:19–37, 2017.

[9] Feng Xia, Laurence T Yang, Lizhe Wang, and Alexey Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101–1102, 2012.

[10] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25:24–44, 2014.

[11] Yasir Mehmood, Farhan Ahmad, Ibrar Yaqoob, Asma Adnane, Muhammad Imran, and Sghaier Guizani. Internet-of-things-based smart cities: Recent advances and challenges. *IEEE Communications Magazine*, 55(9):16–24, 2017.

[12] Andreas Wagner, Sebastian Speiser, and Andreas Harth. Semantic web technologies for a smart energy grid: Requirements and challenges. In *In proceedings of 9th International Semantic Web Conference (ISWC2010)*, pages 33–37. Citeseer, 2010.

[13] Reza Shojanoori and Radmila Juric. Semantic remote patient monitoring system. *Telemedicine and e-Health*, 19(2):129–136, 2013.

[14] Emanuele Della Valle, Stefano Ceri, Frank Van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6), 2009.

[15] Alessandra Mileo, Minh Dao-Tran, Thomas Eiter, and Michael Fink. Stream reasoning. *Encyclopedia of Database Systems*, 2017.

[16] Graham Klyne and Jeremy J Carroll. Resource description framework (RDF): Concepts and abstract syntax. 2006.

[17] Jonas Tappolet and Abraham Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *European Semantic Web Conference*, pages 308–322. Springer, 2009.

[18] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. *W3C recommendation*, 21(10), 2013.

[19] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: a continuous query language for RDF data streams. *International Journal of Semantic Computing*, 4(01):3–25, 2010.

[20] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *The Semantic Web–ISWC 2011*, pages 370–388. Springer, 2011.

[21] Jean-Paul Calbimonte, Ho Young Jeung, Oscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *International Journal on Semantic Web and Information Systems*, 8(EPFL-ARTICLE-183971):43–63, 2012.

[22] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for C-SPARQL queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 441–452. ACM, 2010.

[23] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[24] Opher Etzion, Peter Niblett, and David C Luckham. *Event processing in action*. Manning Greenwich, 2011.

[25] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44 (3):15, 2012.

[26] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *International Workshop on Database Programming Languages*, pages 1–19. Springer, 2003.

[27] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 3–3. Springer, 2008.

[28] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, (Preprint): 1–25, 2017.

[29] Emanuele Della Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. A first step towards stream reasoning. In *Future Internet Symposium*, pages 72–81. Springer, 2008.

[30] Heiner Stuckenschmidt, Stefano Ceri, Emanuele Della Valle, and Frank Van Harmelen. Towards expressive stream reasoning. In *Dagstuhl Seminar proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[31] Jan Leeuwen. *Handbook of theoretical computer science: Algorithms and complexity*, volume 1. Elsevier, 1990.

[32] Gulay Unel and Dumitru Roman. Stream reasoning: A survey and further research directions. In *International Conference on Flexible Query Answering Systems*, pages 653–662. Springer, 2009.

[33] Emanuele Della Valle, Stefan Schlobach, Markus Krötzsch, Alessandro Bozzon, Stefano Ceri, and Ian Horrocks. Order matters! harnessing a world of orderings for reasoning over massive data. *Semantic Web*, 4(2):219–231, 2013.

[34] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *ACM SIGMOD Record*, 39(1):20–26, 2010.

[35] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pages 96–111. Springer, 2010.

[36] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[37] Özgür Lütfü Özçep, Ralf Möller, and Christian Neuenstadt. A stream-temporal query language for ontology based data access. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 183–194. Springer, 2014.

[38] Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 58–68. ACM, 2012.

[39] Yuan Ren, Jeff Z Pan, and Yuting Zhao. Towards scalable reasoning on ontology streams via syntactic approximation. *Proc. of IWOD*, 2010.

[40] Onkar Walavalkar, Anupam Joshi, Tim Finin, and Yelena Yesha. Streaming knowledge bases. In *In International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2008.

[41] Yuan Ren and Jeff Z Pan. Optimising ontology stream reasoning with truth maintenance system. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 831–836. ACM, 2011.

[42] Jacopo Urbani, Alessandro Margara, Ceriel Jacobs, Frank Van Harmelen, and Henri Bal. Dynamite: Parallel materialization of dynamic RDF data. In *International Semantic Web Conference*, pages 657–672. Springer, 2013.

[43] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of Datalog materialisation: the backward/forward algorithm. In *AAAI*, pages 1560–1568, 2015.

[44] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[45] Peter Gärdenfors. *Belief revision*, volume 29. Cambridge University Press, 2003.

[46] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental ASP solver. In *Logic Programming*, pages 190–205. Springer, 2008.

[47] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 54–66. Springer, 2011.

[48] Martin Gebser, Torsten Grote, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, and Torsten Schaub. Answer set programming for stream reasoning. *CoRR, abs/1301.1392*, 2013.

[49] Fredrik Heintz. *DyKnow: A stream-based knowledge processing middleware framework*. PhD thesis, Linköping University Electronic Press, 2009.

[50] Abdulbasit Ahmed, Alexei Lisitsa, and Clare Dixon. A misuse-based network intrusion detection system using temporal logic and stream processing. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 1–8. IEEE, 2011.

[51] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[52] Alexander Bochman. *A logical theory of nonmonotonic inference and belief change*. Springer Science & Business Media, 2013.

[53] Thang M Do, Seng W Loke, and Fei Liu. Answer set programming for stream reasoning. In *Advances in Artificial Intelligence*, pages 104–109. Springer, 2011.

[54] Alessandra Mileo, Ahmed Abdelrahman, Sean Policarpio, and Manfred Hauswirth. Streamrule: a nonmonotonic stream reasoning system for the semantic web. In *Web Reasoning and Rule Systems*, pages 247–252. Springer, 2013.

[55] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems*, pages 40–110. Springer, 2009.

[56] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: Input language format. *URL: https://www. mat. unical. it/aspcomp2013/files/ASP-CORE-2.03 b. pdf [Accessed on August 15, 2017]*, 2012.

[57] Simona Perri, Francesco Ricca, and Marco Sirianni. Parallel instantiation of ASP programs: techniques and experiments. *Theory and Practice of Logic Programming*, 13(2):253–278, 2013.

[58] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Oscar Corcho. Towards a unified language for RDF stream query processing. In *International Semantic Web Conference*, pages 353–363. Springer, 2015.

[59] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal RDF. In *European Semantic Web Conference*, pages 93–107. Springer, 2005.

[60] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. ETALIS: Rule-based reasoning in event processing. In *Reasoning in event-based distributed systems*, pages 99–124. Springer, 2011.

[61] Marco Balduini, Emanuele Della Valle, Daniele Dell'Aglio, Mikalai Tsytsarau, Themis Palpanas, and Cristian Confalonieri. Social listening of scale events using the streaming linked data framework. In *International Semantic Web Conference*, pages 1–16. Springer, 2013.

[62] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(4):17–44, 2014.

[63] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[64] Victor W Marek and Miroslaw Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm*, pages 375–398. Springer, 1999.

[65] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of mathematics and Artificial Intelligence*, 25(3-4): 241–273, 1999.

[66] Alessandro Provetti and Tran Cao Son. *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning: Papers from the 2001 AAAI Symposium, March 26-28, Stanford, California.* AAAI Press, 2001.

[67] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation—the A-Prolog perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[68] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–385, 1991.

[69] Wolfgang Faber, Nicola Leone, and Simona Perri. The intelligent grounder of DLV. In *Correct Reasoning*, pages 247–264. Springer, 2012.

[70] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 266–271. Springer, 2007.

[71] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in gringo series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 345–351. Springer, 2011.

[72] Tommi Syrjänen. Omega-restricted logic programs. In *International Conference on Logic Programming and NonMonotonic Reasoning*, pages 267–280. Springer, 2001.

[73] Tommi Syrjnen. Lparse 1.0 users manual, 2002.

[74] Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO (ID) with bounds. *arXiv preprint arXiv:1401.3840*, 2014.

[75] Deborah East, Mikhail Iakhiaev, Artur Mikitiuk, and Mirosław Truszczyński. Tools for modeling and solving search problems. *AI Communications*, 19(4):301–312, 2006.

[76] Remi Brochenin, Marco Maratea, and Yuliya Lierler. Disjunctive answer set solvers via templates. *Theory and Practice of Logic Programming*, 16(4):465–497, 2016.

[77] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[78] Jean Gressmann, Tomi Janhunen, Robert E Mercer, Torsten Schaub, Sven Thiele, and Richard Tichy. Platypus: A platform for distributed answer set solving. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 227–239. Springer, 2005.

[79] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3): 499–562, 2006.

[80] Christian Anger, Martin Gebser, Thomas Linke, André Neumann, and Torsten Schaub. The nomore++ approach to answer set solving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 95–109. Springer, 2005.

[81] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic (TOCL)*, 7(1):1–37, 2006.

[82] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[83] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345, 2006.

[84] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native asp solver based on constraint learning. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 54–66. Springer, 2013.

[85] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, volume 7, pages 386–392, 2007.

[86] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo= ASP+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

[87] Francesco Ricca. The DLV Java wrapper. In *APPIA-GULP-PRODE*, pages 263–274. Citeseer, 2003.

[88] Onofrio Febbraro, Nicola Leone, Giovanni Grasso, and Francesco Ricca. JASP: A framework for integrating answer set programming with Java. In *KR*, 2012.

[89] Davide Fuscà, Stefano Germano, Jessica Zangari, Marco Anastasio, Francesco Calimeri, and Simona Perri. A framework for easing the development of applications embedding answer set programming. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, pages 38–49. ACM, 2016.

[90] Francesco Ricca, Lorenzo Gallucci, Roman Schindlauer, Tina Dell'Armi, Giovanni Grasso, and Nicola Leone. OntoDLV: an asp-based system for enterprise ontologies. *Journal of Logic and Computation*, 19(4):643–670, 2008.

[91] Paula-Andra Busoniu, Johannes Oetsch, Joerg Puehrer, PETER SKOČOVSKÝ, and Hans Tompits. SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming*, 13 (4-5):657–673, 2013.

[92] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: Integrated development environment for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 317–330. Springer, 2011.

[93] Martin Gebser and Torsten Schaub. Modeling and language extensions. *AI Magazine*, 37(3), 2016.

[94] Yuliya Lierler, Marco Maratea, and Francesco Ricca. Systems, engineering environments, and competitions. *AI Magazine*, 37(3), 2016.

[95] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3), 2016.

[96] Esra Erdem, Erdi Aker, and Volkan Patoglu. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, 5(4):275–291, 2012.

[97] Esra Erdem, Volkan Patoglu, Zeynep G Saribatur, Peter Schüller, and Tansel Uras. Finding optimal plans for multiple teams of robots through a mediator: A logic-based approach. *Theory and Practice of Logic Programming*, 13(4-5):831–846, 2013.

[98] Esra Erdem, Volkan Patoglu, and Zeynep G Saribatur. Integrating hybrid diagnostic reasoning in plan execution monitoring for cognitive factories with multiple robots. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 2007–2013. IEEE, 2015.

[99] Giray Havur, Guchan Ozbilgin, Esra Erdem, and Volkan Patoglu. Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 445–452. IEEE, 2014.

[100] Nam Tran and Chitta Baral. Hypothesizing about signaling networks. *Journal of Applied Logic*, 7(3):253–274, 2009.

[101] Martin Gebser, Torsten Schaub, Sven Thiele, and Philippe Veber. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming*, 11(2-3):323–360, 2011.

[102] Daniel R Brooks, Esra Erdem, Selim T Erdoğan, James W Minett, and Don Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39(4):471, 2007.

[103] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental & Theoretical Artificial Intelligence*, 21 (2):79–121, 2009.

[104] Esra Erdem, Yelda Erdem, Halit Erdogan, and Umut Öztok. Finding answers and generating explanations for complex biomedical queries. In *AAAI*, 2011.

[105] Esra Erdem and Umut Oztok. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming*, 15(1):35–78, 2015.

[106] Francesco Ricca, Antonella Dimasi, Giovanni Grasso, Salvatore Maria Ielpa, Salvatore Iiritano, Marco Manna, and Nicola Leone. A logic-based system for e-tourism. *Fundamenta Informaticae*, 105(1-2):35–55, 2010.

[107] Francesco Ricca, Giovanni Grasso, Mario Alviano, Marco Manna, Vincenzino Lio, Salvatore Iiritano, and Nicola Leone. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming*, 12(3):361–381, 2012.

[108] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.

[109] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[110] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[111] Edward Tsang. *Foundations of constraint satisfaction: the classic text.* BoD–Books on Demand, 2014.

[112] Simona Perri, Francesco Scarcello, Gelsomina Catalano, and Nicola Leone. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):195, 2007.

[113] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP instantiators by join-ordering methods. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 280–294. Springer, 2001.

[114] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using database optimization techniques for nonmonotonic reasoning. 1999.

[115] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The journal of logic programming*, 10(3-4):255–299, 1991.

[116] Mario Alviano and Wolfgang Faber. Dynamic magic sets and super-coherent answer set programs. *AI Communications*, 24(2):125–145, 2011.

[117] Ilkka Niemela, Patrik Simons, and Tommi Syrjanen. Smodels: a system for answer set programming. *arXiv preprint cs/0003033*, 2000.

[118] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[119] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[120] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

[121] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Multi-threaded ASP solving with clasp. *Theory and Practice of Logic Programming*, 12(4-5):525–545, 2012.

[122] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. A portfolio solver for answer set programming: Preliminary report. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 352–357. Springer, 2011.

[123] Marco Maratea, Luca Pulina, and Francesco Ricca. A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming*, 14(6):841–868, 2014.

[124] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *ACM Sigmod Record*, volume 18, pages 215–224. ACM, 1989.

[125] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[126] Norman W Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999.

[127] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12 (2):12039, 2007.

[128] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, volume 96, page 594, 1996.

[129] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Nesime Tatbul, Stan Zdonik, and Michael Stonebraker. Monitoring streams—a new class of data management applications. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 215–226. Elsevier, 2002.

[130] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A model for sequence databases. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 232–239. IEEE, 1995.

[131] David Luckham. Event processing glossary-version 2.0, event processing technical society. *http://www. ep-ts. com/component/option, com_docman/task, doc_download/gid, 66/Itemid, 84/*, 2016.

[132] K Mani Chandy, Michel Charpentier, and Agostino Capponi. Towards a theory of events. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 180–187. ACM, 2007.

[133] S Chakravarthy and D Mishra-Snoop. An expressive event specification language for active databases. university of florida. Technical report, Technical Report UF-CIS-TR-93-007.

[134] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. SR-Bench: a streaming RDF/SPARQL benchmark. In *International Semantic Web Conference*, pages 641–657. Springer, 2012.

[135] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Extended Semantic Web Conference*, pages 1–15. Springer, 2010.

[136] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL-extending SPARQL to process data streams. In *European Semantic Web Conference*, pages 448–462. Springer, 2008.

[137] Adrian Paschke. A semantic design pattern language for complex event processing. In *AAAI Spring Symposium: Intelligent Event Processing*, pages 54–60, 2009.

[138] Robin Keskisärkkä and Eva Blomqvist. Semantic complex event processing for social media monitoring-a survey. In *Proceedings of Social Media and Linked Data for Emergency Response (SMILE) Co-located with the 10th Extended Semantic Web Conference, Montpellier, France. CEUR workshop proceedings (May 2013)*, 2013.

[139] Marc Schaaf, Stella Gatziu Grivas, Dennie Ackermann, Arne Diekmann, Arne Koschel, and Irina Astrova. Semantic complex event processing. *Recent Researches in Applied Information Science*, pages 38–43, 2012.

[140] Kia Teymourian, Malte Rohde, and Adrian Paschke. Knowledge-based processing of complex stock market events. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 594–597. ACM, 2012.

[141] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On supporting kleene closure over event streams. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1391–1393. IEEE, 2008.

[142] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 50–61. ACM, 2010.

[143] Alessandro Artale and Enrico Franconi. A survey of temporal extensions of description logics. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):171–210, 2000.

[144] Patrick Doherty and Jonas Kvarnström. Temporal action logics. *Foundations of Artificial Intelligence*, 3:709–757, 2008.

[145] Anastasios Skarlatidis, Georgios Paliouras, Alexander Artikis, and George A Vouros. Probabilistic event calculus for event recognition. *ACM Transactions on Computational Logic (TOCL)*, 16(2):11, 2015.

[146] Carsten Lutz. Description logics with concrete domains–a survey. 2003.

[147] Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A system for incremental ASP-based stream reasoning. *Theory and Practice of Logic Programming*, pages 1–20, 2017.

[148] Hamid R Bazoobandi, Harald Beck, and Jacopo Urbani. Expressive stream reasoning with Laser. In *International Semantic Web Conference*, pages 87–103. Springer, 2017.

[149] Mattias Tiger and Fredrik Heintz. Stream reasoning using temporal logic and predictive probabilistic state models. In *Temporal Representation and Reasoning (TIME), 2016 23rd International Symposium on*, pages 196–205. IEEE, 2016.

[150] Adrian Paschke and Harold Boley. Rule responder: rule-based agents for the semantic-pragmatic web. *International Journal on Artificial Intelligence Tools*, 20 (06):1043–1081, 2011.

[151] Kia Teymourian, Malte Rohde, and Adrian Paschke. Fusion of background knowledge and streams of events. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 302–313. ACM, 2012.

[152] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th international conference on World wide web*, pages 1061–1062. ACM, 2009.

[153] Christian YA Brenninkmeijer, Ixent Galpin, Alvaro AA Fernandes, and Norman W Paton. A semantics for a query language over sensors, streams and relations. In *British National Conference on Databases*, pages 87–99. Springer, 2008.

[154] CL Rete. A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17–37, 1982.

[155] Mikko Rinne, Esko Nuutila, and Seppo Törmä. INSTANS: high-performance event processing with standard RDF and SPARQL. In *Proceedings of the 2012th International Conference on Posters & Demonstrations Track-Volume 914*, pages 101–104. Citeseer, 2012.

[156] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012.

[157] Daniele Dell'Aglio, Minh Dao-Tran, Jean-Paul Calbimonte, Danh Le Phuoc, and Emanuele Della Valle. A query model to capture event pattern matching in RDF stream processing query languages. In *European Knowledge Acquisition Workshop*, pages 145–162. Springer, 2016.

[158] Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In *ECAI*, volume 6, pages 412–416, 2006.

[159] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with Clingo. *Theory and Practice of Logic Programming*, pages 1–56, 2018.

[160] Harald Beck, Minh Dao-Tran, and Thomas Eiter. LARS: A logic-based framework for analytic reasoning over streams. *Artificial Intelligence*, 261:16–70, 2018.

[161] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Answer update for rule-based stream reasoning. In *IJCAI*, pages 2741–2747, 2015.

[162] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[163] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. *DLV-hex: Dealing with semantic web under answer set programming.* In: Proc. of ISWC, 2005.

[164] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. Approximate continuous query answering over streams and dynamic linked data sets. In *International Conference on Web Engineering*, pages 307–325. Springer, 2015.

[165] Thomas Scharrenbach, Jacopo Urbani, Alessandro Margara, Emanuele Della Valle, and Abraham Bernstein. Seven commandments for benchmarking semantic flow processing systems. In *Extended Semantic Web Conference*, pages 305–319. Springer, 2013.

[166] Jim Grey. The benchmark handbook for database and transaction systems, 1993.

[167] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156. ACM, 2011.

[168] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

[169] Juan F Sequeda and Oscar Corcho. Linked stream data: A position paper. 2009.

[170] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference*, pages 300–312. Springer, 2012.

[171] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Oscar Corcho, and Emanuele Della Valle. On correctness in RDF stream processor benchmarking. In *International semantic web conference*, pages 326–342. Springer, 2013.

[172] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. CityBench: A configurable benchmark to evaluate RSP engines using smart city datasets. In *International Semantic Web Conference*, pages 374–389. Springer, 2015.

[173] Maxim Kolchin, Peter Wetz, Elmar Kiesling, and A Min Tjoa. YABench: A comprehensive framework for RDF stream processor correctness and performance assessment. In *International Conference on Web Engineering*, pages 280–298. Springer, 2016.

[174] Tu Ngoc Nguyen and Wolf Siberski. SLUBM: An extended LUBM benchmark for stream reasoning. In *OrdRing@ ISWC*, pages 43–54, 2013.

[175] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[176] Christopher J Matheus, Ken Baclawski, and Mieczyslaw M Kokar. Basevisor: A triples-based inference engine outfitted to process ruleml and r-entailment rules. In *Rules and Rule Markup Languages for the Semantic Web, Second International Conference on*, pages 67–74. IEEE, 2006.

[177] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[178] Martin Gebser, Marco Maratea, and Francesco Ricca. The sixth answer set programming competition. *Journal of Artificial Intelligence Research*, 60:41–95, 2017.

[179] Filippo Cacace, Stefano Ceri, and Maurice Houtsma. A survey of parallel execution strategies for transitive closure and logic programs. *Distributed and Parallel Databases*, 1(4):337–382, 1993.

[180] Ramakrishna Soma and Viktor K Prasanna. Parallel inferencing for OWL knowledge bases. In *37th International Conference on Parallel Processing*, pages 75–82. IEEE, 2008.

[181] Peiqiang Li, Yi Zeng, Spyros Kotoulas, Jacopo Urbani, and Ning Zhong. The quest for parallel reasoning on the semantic web. In *International Conference on Active Media Technology*, pages 430–441. Springer, 2009.

[182] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: Distributed reasoning over large-scale semantic web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):305–316, 2009.

[183] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank Harmelen. Scalable distributed reasoning using MapReduce. In *Proceedings of the 8th International Semantic Web Conference*, pages 634–649. Springer-Verlag, 2009.

[184] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[185] Ilias Tachmazidis, Grigoris Antoniou, Giorgos Flouris, and Spyros Kotoulas. Towards parallel nonmonotonic reasoning with billions of facts. In *KR*, 2012.

[186] Ilias Tachmazidis and Grigoris Antoniou. Computing the stratified semantics of logic programs over big data through mass parallelization. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 188–202. Springer, 2013.

[187] Ilias Tachmazidis, Grigoris Antoniou, and Wolfgang Faber. Efficient computation of the well-founded semantics over big data. *arXiv preprint arXiv:1405.2590*, 2014.

[188] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Parallel answer set programming. In *Handbook of Parallel Constraint Reasoning*, pages 237–282. Springer, 2018.

[189] Marcello Balduccini, Enrico Pontelli, Omar Elkhatib, and Hung Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31 (6):608–647, 2005.

[190] Enrico Pontelli and Omar El-Khatib. Exploiting vertical parallelism from answer set programs. In *Answer Set Programming*, 2001.

[191] Raphael A Finkel, Victor W Marek, Neil Moore, and Miroslaw Truszczynski. Computing stable models in parallel. In *Answer Set Programming*, 2001.

[192] Hung Viet Le and Enrico Pontelli. An investigation of sharing strategies for answer set solvers and SAT solvers. In *European Conference on Parallel Processing*, pages 750–760. Springer, 2005.

[193] Enrico Pontelli, Hung Viet Le, and Tran Cao Son. An investigation in parallel execution of answer set programs on distributed memory platforms: Task sharing and dynamic scheduling. *Computer Languages, Systems & Structures*, 36(2):158–202, 2010.

[194] Agostino Dovier, Andrea Formisano, Enrico Pontelli, and Flavio Vella. Parallel execution of the ASP computation - an investigation on GPUs. In *ICLP (Technical Communications)*, 2015.

[195] Agostino Dovier, Andrea Formisano, Enrico Pontelli, and Flavio Vella. A GPU implementation of the ASP computation. In *International Symposium on Practical Aspects of Declarative Languages*, pages 30–47. Springer, 2016.

[196] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.

[197] Stefano Germano, Thu-Le Pham, and Alessandra Mileo. Web stream reasoning in practice: on the expressivity vs. scalability tradeoff. In *Web Reasoning and Rule Systems*, pages 105–112. Springer, 2015.

[198] Thu-Le Pham, Alessandra Mileo, and Muhammad Intizar Ali. Towards scalable non-monotonic stream reasoning via input dependency analysis. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 1553–1558. IEEE, 2017.

[199] Thu-Le Pham, Muhammad Intizar Ali, and Alessandra Mileo. Enhancing the scalability of expressive stream reasoning via input-driven parallelization. *Semantic Web*, (Preprint):1–17, 2018.

[200] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[201] Ralf Tönjes, P Barnaghi, M Ali, A Mileo, M Hauswirth, F Ganz, S Ganea, B Kjærgaard, D Kuemper, Septimiu Nechifor, et al. Real time IoT stream processing and large-scale data analytics for smart city applications. In *poster session, European Conference on Networks and Communications*. sn, 2014.

[202] Thu-Le Pham, Stefano Germano, Alessandra Mileo, Daniel Küemper, and Muhammad Intizar Ali. Automatic configuration of smart city applications for user-centric decision support. In *Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on*, pages 360–365. IEEE, 2017.

[203] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, pages 245–256, 2003.

[204] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[205] Chan Le Van, Feng Gao, and Muhammad Intizar Ali. Optimizing the performance of concurrent RDF stream processing queries. In *European Semantic Web Conference*, pages 238–253. Springer, 2017.