Logic Programming in non-conventional environments

Stefano Germano

Nov 2014 – Oct 2017
Version: 2.0

UNIVERSITÀ DELLA CALABRIA

Department of Mathematics and Computer Science

Doctoral Degree in Mathematics and Computer Science

XXX Cycle – *S.S.D.* INF/01

Doctoral Thesis

Logic Programming in non-conventional environments

Stefano Germano

Coordinator	Prof. Nicola Leone
Supervisor	Prof. Giovambattista Ianni

Nov 2014 – Oct 2017

Stefano Germano
Logic Programming in non-conventional environments
Doctoral Thesis, Nov 2014 – Oct 2017
Coordinator of the Doctoral Programme: Prof. Nicola Leone
Supervisor: Prof. Giovambattista Ianni

University of Calabria
Doctoral Degree in Mathematics and Computer Science
Department of Mathematics and Computer Science
Via Pietro Bucci – I-87036
Arcavacata di Rende, CS – Italy

Abstract

Logic Programming became a very useful paradigm in many different areas and thus several languages (and solvers) have been created to support various kinds of reasoning tasks. However, in the last decades, thanks also to results in the Computational Complexity area, the weaknesses and the limits of this formalism have been revealed. Therefore, we decided to study solutions that would allow the use of the *Logic Programming* paradigm in contexts, such as *Stream Reasoning*, *Big Data* or Games' *AI*, that have very specific constraints that make the practical use of logic-based formalisms not so straightforward.

Logic Programming is best used for problems where a properly defined search space can be identified and has to be explored in full. For this reason, almost all the approaches that have been tried so far, have focused on problems where the amount of input data is not huge and is stored in a few well-defined sources, which are often completely available at the beginning of the computation.
Some interesting ideas and approaches in the mentioned fields have been already introduced, however, they are in a preliminary stage and often are tailored to a specific problem, and do not allow the users to perform "complex" reasoning on the data. In order to make the utilisation of this paradigm computationally feasible and reliable in contexts where the reasoning methods have to handle a huge amount of data that "expires" soon, i.e., become soon useless, and quickly react to them, new solutions have to be introduced.

In this Thesis, we illustrate how *Logic Programming* can play a role in such challenging scenarios. We both describe the general approaches taken, and how these solutions have been used in several application contexts, some of which were milestones of large-scale international projects. We found that combining different reasoning methods and technologies is one of the crucial methodologies to adopt in order to effectively tackle and solve these challenges. Moreover, we identify the methodological gaps that are not yet closed, and prevent the large adoption of logic-based programming techniques.

Sommario

La *Programmazione Logica* è diventata un paradigma molto utile in diverse aree e sono perciò stati creati diversi linguaggi (e risolutori) per supportare vari tipi di operazioni di ragionamento. Tuttavia negli ultimi decenni, grazie anche ai risultati ottenuti nell'area della Complessità computazionale, si sono evidenziate le debolezze e i limiti di questo formalismo. Abbiamo quindi deciso di studiare soluzioni che permettessero l'uso del paradigma della *Programmazione Logica* in contesti come *Stream Reasoning*, *Big Data* o *Intelligenza Artificiale* nei (Video-)Giochi; contesti che hanno vincoli molto specifici che rendono l'uso pratico di formalismi basati sulla logica non particolarmente semplice.

La *Programmazione Logica* è utilizzata soprattutto per i problemi in cui è possibile identificare uno spazio di ricerca ben definito e che deve essere esplorato a fondo. Per questo motivo, quasi tutti gli approcci finora sperimentati si sono concentrati su problemi in cui la quantità di dati di input non è enorme e viene memorizzata in poche fonti ben definite che sono spesso completamente disponibili all'inizio del calcolo.
Sono già state introdotte alcune idee e approcci interessanti nei settori citati, ma sono in fase preliminare e spesso sono su misura per un problema specifico, e non consentono di svolgere ragionamenti "complessi" sui dati. Per rendere l'utilizzo di questo paradigma computazionale praticabile e affidabile in contesti in cui i metodi di ragionamento devono gestire un'enorme quantità di dati che "scadono" presto, cioè diventano presto inutilizzabili, e reagiscono rapidamente ad essi, è necessario introdurre nuove soluzioni.

In questa tesi illustriamo come la *Programmazione Logica* possa giocare un ruolo importante in scenari così impegnativi. Descriviamo sia gli approcci generali adottati sia il modo in cui queste soluzioni sono state utilizzate in diversi contesti applicativi, alcuni dei quali sono stati fondamentali per grandi progetti internazionali. Abbiamo constatato che la combinazione di diversi metodi e tecnologie di ragionamento è una delle metodologie cruciali da adottare per affrontare e risolvere efficacemente queste sfide. Inoltre segnaliamo quali sono le carenze metodologiche attualmente non colmate che impediscono l'adozione su vasta scala di tecniche di programmazione basate sulla logica.

Dedicata alla mia famiglia

Meiner Familie gewidmet

Dedicada a mi familia

I gcomóradh mo mhuintir

Dédiée à ma famille

Dedicated to my family

“The world is full of obvious things which nobody by any chance ever observes.”

– Arthur Conan Doyle, The Hound of the Baskervilles

Acknowledgements

During the journey ended up in this Thesis I have actually met many people, in several places, and I must admit that I learned something from each of them. Besides, they were important for me not only from a professional point of view but also from a personal one.

However, I realised they are too many to thank them one by one for what they have done for me. Therefore, I would simply say thanks to all of you, you made this journey very interesting and enjoyable.

Nonetheless, I would especially like to thank "my team" (as my mum usually calls them) who have assisted, supported and endorsed me; without them none of what I achieved in the last years would have been possible.

Furthermore, I thankfully acknowledge the collaboration with other students and researchers in various universities, which motivated and allowed me to produce this contribution, among which the *Department of Mathematics and Computer Science (DeMaCS)*[1] at the *Università della Calabria (UNICAL)*[2], the *Knowledge-Based Systems Group (KBS)*[3] of the *Faculty of Informatics*[4] at the *Technische Universität Wien (TU Wien)*[5], the *Unit for Reasoning and Querying (URQ)*[6] of the *INSIGHT Centre for Data Analytics*[7] at the *National University of Ireland (NUI) Galway*[8], the *Database group*[9] of the *Department of Computer Science*[10] at the *University of Oxford*[11].

It is important to notice that part of the work described in this Thesis has already been acknowledged by the scientific community in international Journals and Conferences. In particular:

[1] https://www.mat.unical.it
[2] http://unical.it
[3] http://www.kr.tuwien.ac.at
[4] http://www.informatik.tuwien.ac.at
[5] https://www.tuwien.ac.at
[6] https://nuig.insight-centre.org/urq
[7] https://www.insight-centre.org
[8] http://www.nuigalway.ie
[9] http://www.cs.ox.ac.uk/isg/db
[10] http://www.cs.ox.ac.uk
[11] http://www.ox.ac.uk

⁜ D. Fuscà, S. Germano, J. Zangari, F. Calimeri and S. Perri. ‘Answer Set Programming and Declarative Problem Solving in Game AIs’. In: *[45]*. 2013, pp. 81–88. URL: http://ceur-ws.org/Vol-1107/paper9.pdf.

⁜ F. Calimeri, M. Fink, S. Germano, G. Ianni, C. Redl and A. Wimmer. ‘AngryHEX: an Artificial Player for Angry Birds Based on Declarative Knowledge Bases’. In: *[45]*. 2013, pp. 29–35. URL: http://ceur-ws.org/Vol-1107/paper10.pdf.

⁜ S. Germano, T. Pham and A. Mileo. ‘Web Stream Reasoning in Practice: On the Expressivity vs. Scalability Tradeoff’. In: *Proceedings of RR 2015*, pp. 105–112. DOI: 10.1007/978-3-319-22002-4_9.

⁜ A. Mileo, S. Germano, T.-L. Pham, D. Puiu, D. Kuemper and M. I. Ali. *User-Centric Decision Support in Dynamic Environments. CityPulse - Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications*. Report - Project Delivery. Version V1.0-Final. NUIG, SIE, UASO, 31st Aug. 2015. URL: http://cordis.europa.eu/docs/projects/cnect/5/609035/080/deliverables/001-609035CITYPULSED52renditionDownload.pdf (visited on 25th Sept. 2017).

⁜ F. Calimeri, D. Fuscà, S. Germano, S. Perri and J. Zangari. ‘Embedding ASP in mobile systems: discussion and preliminary implementations’. In: *Proceedings of ASPOCP 2015, workshop of ICLP*.

⁜ F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci and A. Wimmer. ‘Angry-HEX: An Artificial Player for Angry Birds Based on Declarative Knowledge Bases’. In: *TCIAIG* 8.2 (2016), pp. 128–139. DOI: 10.1109/TCIAIG.2015.2509600.

⁜ F. Calimeri, D. Fuscà, S. Germano, S. Perri and J. Zangari. ‘Boosting the Development of ASP-Based Applications in Mobile and General Scenarios’. In: *Proceedings of AI*IA 2016*, pp. 223–236. DOI: 10.1007/978-3-319-49130-1_17.

⁜ D. Fuscà, S. Germano, J. Zangari, M. Anastasio, F. Calimeri and S. Perri. ‘A framework for easing the development of applications embedding answer set programming’. In: *Proceedings of PPDP 2016*, pp. 38–49. DOI: 10.1145/2967973.2968594.

⁜ T. Pham, S. Germano, A. Mileo, D. Kümper and M. I. Ali. ‘Automatic configuration of smart city applications for user-centric decision support’. In: *Proceedings of ICIN 2017*, pp. 360–365. DOI: 10.1109/ICIN.2017.7899441.

⁜ S. Germano, F. Calimeri and E. Palermiti. ‘LoIDE: a web-based IDE for Logic Programming - Preliminary Technical Report’. In: *CoRR* abs/1709.05341 (2017). arXiv: 1709.05341.

⋇ F. Calimeri, D. Fuscà, S. Germano, S. Perri and J. Zangari. 'A framework for easing the development of applications embedding answer set programming'. In: *J. Exp. Theor. Artif. Intell.* (2017). Submitted.

⋇ F. Calimeri, S. Germano, E. Palermiti, K. Reale and F. Ricca. 'Environments for Developing ASP programs'. In: *KI* (2017). Submitted.

⋇ T. Eiter, S. Germano, G. Ianni, T. Kaminski, C. Redl, P. Schüller and A. Weinzierl. 'The DLVHEX System'. In: *KI* (2017). Submitted.

⋇ S. Germano, F. Calimeri and E. Palermiti. 'LoIDE: A Web-Based IDE for Logic Programming Preliminary Report'. In: *Proceedings of PADL 2018*, pp. 152–160. DOI: 10.1007/978-3-319-73305-0_10.

Part of this work has been realized in the context of the projects:

- EU FP7 ***CityPulse*** Project, under grant No.603095[12];
- EPSRC VADA Project, EP/M025268/1[13];
- Angry-HEX[14], EMBASP[15], and *L*oIDE[16].

[12]http://www.ict-citypulse.eu
[13]http://vada.org.uk
[14]https://demacs-unical.github.io/Angry-HEX
[15]https://www.mat.unical.it/calimeri/projects/embasp
[16]https://www.mat.unical.it/calimeri/projects/loide

Contents

List of Figures

List of Tables

List of Programs

Acronyms

AIBIRDS Angry Birds AI Competition.

AI Artificial Intelligence.

API Application Programming Interface.

APK Android Package Kit.

ASP Answer Set Programming.

BCQ Boolean Conjunctive Query.

BSM Behavioural State Machines.

BT Behavior Tree.

BWAPI Brood War Application Programming Interface.

C-SPARQL Continuous SPARQL.

CCN Content-Centric Networking.

CDCL Conflict-Driven Clause Learning.

CEP Complex Event Processing.

CHR Constraint Handling Rules.

CI Computational Intelligence.

CQELS Continuous Query Evaluation over Linked Stream.

CQL Continuous Query Language.

CQ Conjunctive Query.

CSS Cascading Style Sheets.

CWA Closed World Assumption.

DBMS DataBase Management System.

DB DataBase.

DDL Data Definition Language.

DLP Disjunctive Logic Programming.

DLV DATALOG WITH DISJUNCTION.

DML Data Manipulation Language.

DPLL Davis-Putnam-Logemann-Loveland.

DSMS Data Stream Management System.

DTM Datasphere Transducer Model.

EDB Extensional DataBase.

EGD Equality-Generating Dependency.

ELE ETALIS Language for Events.

EP-SPARQL Event Processing SPARQL.

FO First-Order.

FPS first-person shooter.

FSM finite-state machine.

GCO Guess/Check/Optimize.

GDI Geospatial Data Infrastructure.

GOAP Goal-Oriented Action Planning.

GPS Global Positioning System.

GUI Graphical User Interface.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

ICT Information and Communication Technology.

IDB Intensional DataBase.

IDC International Data Corporation.

IDE Integrated Development Environment.

INSTANS *Incremental eNgine for STANding Sparql.*

IQM interquartile mean.

IRI Internationalized Resource Identifier.

IoT Internet of Things.

JNI Java Native Interface.

JPA Java Persistence API.

JSON JavaScript Object Notation.

KB Knowledge Base.

KR&R Knowledge Representation and Reasoning.

KR Knowledge Representation.

LARS Logic-based framework for Analyzing Reasoning over Streams.

LSD Linked Stream Data.

LUBM Lehigh University Benchmark.

OBDA Ontology-Based Data Access.

ODBC Open DataBase Connectivity.

ORM Object-Relational Mapping.

OSS open-source software.

OS Operating System.

OWA Open World Assumption.

OWL Web Ontology Language.

PBSG physics-based simulation game.

PCWA Progressive Closing World Assumption.

PDDL Planning Domain Definition Language.

PSA Perfect Synchronization Assumption.

Potassco Potsdam Answer Set Solving Collection.

QBF Quantified Boolean Formula.

RDBMS Relational DataBase Management System.

RDF Resource Description Framework.

RFID Radio Frequency IDentification.

RSP RDF Stream Processing.

RTS real-time strategy.

SCC Strongly Connected Component.

SDP statistics for derived predicates.

SMT Satisfiability Modulo Theories.

SPARQL SPARQL Protocol and RDF Query Language.

SPE Stream Processing Engine.

SQL Structured Query Language.

SSW Semantic Sensor Web.

STREAM STanford stREamdatA Manager.

strRS Streaming RDF/SPARQL.

TGD Tuple-Generating Dependency.

TMS Truth Maintenance Systems.

UCQ Union of Conjunctive Queries.

UML Unified Modelling Language.

URL Uniform Resource Locator.

VADA Value-Added DAta *systems.*

W3C World Wide Web Consortium.

Introduction

In the early '70, there has been a big hype on *Logic Programming* and its capabilities and potentialities seemed endless: time demonstrated however that this was not entirely true. In this Thesis, we investigate different scenarios exhibiting contexts where *Logic Programming* appears to be not suitable; we show how *Logic Programming*, if properly tuned and combined with other techniques and formalisms, plays a prominent role in them, and we identify which are the gaps still to be closed to make *Logic Programming* a formalism up to its encouraging promises.

Among these challenging scenarios, we can include the so called "Web of data". The rise of the "Web of data", whose ultimate goal is to develop systems that can support trusted interactions over the network and, in particular, of the "*Semantic Web*", that refers to *W3C*'s vision of the Web of *Linked Data*, led to the need to process huge amounts of heterogeneous data and the relationships between them. This collection of interrelated datasets on the Web, often referred to as *Linked Data*, are in a standard format and are reachable and manageable by *Semantic Web* tools. *Linked Data* are empowered by technologies such as *RDF*, *SPARQL*, *OWL* and these *Semantic Web* technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data.

Nowadays many applications need to process dynamic (rapidly changing) data and especially "*data streams*", i.e., unbounded sequences of time-varying data elements, and they need to be elaborated promptly; however, there are currently only a few ways to perform complex reasoning tasks on them and only very recently a completely declarative way to perform this has been introduced.
The combination of reasoning techniques with *data streams* gives rise to *Stream Reasoning*, an unexplored, yet high impact, research area (see Chapter 2).

A further promising application field requiring advanced and complex reasoning capabilities is the so-called "*Internet of Things* (*IoT*)", an area that is having a tremendous growth and that is predicted to expand even more in the following years[1]. The *IoT* represents an evolution in which objects are capable of interacting with

[1]Gartner. http://www.gartner.com/newsroom/id/3598917

other objects; in other words, *IoT* concerns about providing knowledge and insights regarding objects (i.e., things), the physical environment, the human and social activities in these environments (as may be recorded by devices), and enabling systems to take actions based on the knowledge obtained. Clearly, most of the data coming from these "objects" are in form of "flows", and many advanced applications can benefit from the ability to process these *data streams* in real-time. In this respect, *Real-Time Smart City Applications* are a notable and important example.

At the same time, the amount of data that we are collecting in the "Web of Data" and from the *IoT* devices is enormous and, usually, the data are "complex" (consisting of both structured and unstructured data) and they contain a multitude of useful information that needs to be extracted and analysed. Commonly this massive amount of data is referred as *Big Data* and is described by the so-called 4 V's (see Chapter 3).
But it is not the amount of data that is important, it is what the users do with the data that matters and, for this reason, different kinds of analysis are needed. Indeed, to extract *insights* from this "complex" data, specific technologies and algorithms are needed: as a consequence, in the latest years, the number of logic-based approaches for reasoning over *Big Data* is continuously growing.

Another very interesting field, where reasoning techniques that can quickly and effectively handle various kind of data are needed, is *Artificial Intelligence* applied to Games. Video Games are becoming always more popular and widespread and have evolved into a mass medium[2].
AI is a central component of almost every video game and often it is implemented with *ad hoc* solutions for the specific game and hard-coded within the game. However, games are also a very attractive and challenging platform that can be used to test and improve *Artificial Intelligence* techniques. Historically, very basic *AI* techniques have been used to design and implement games' *AI*s but, lately, more advanced approaches have been integrated (see Chapter 4).

The design and development of "*Intelligent Agents*" is now becoming a trend in the IT industry[3] and they are the base of many promising strategy technologies of the future[4]. Intelligent agents are also the base of many *AI* Competitions, especially the ones related to games, and they motivate researchers to invent novel and powerful solutions. The techniques developed or improved by using games as test-bed systems are eventually applied in many other fields and have a profound impact on the (real) lives of several peoples.

[2]Entertainment Software Association. http://www.theesa.com/about-esa/industry-facts
[3]Gartner. http://www.gartner.com/newsroom/id/3143521
[4]Gartner. http://www.gartner.com/smarterwithgartner/gartners-top-10-technology-trends-2017

Despite the suitability of *Logic Programming* in Problem Solving and *Knowledge Representation and Reasoning* (which can be called the “canonical” setting), it is unclear whether this paradigm can play a role in other “non-conventional” settings like the above, in which a combination of the following might be noticed *(i)* there are particular time and size demands; *(ii)* input data is not entirely available but appears in the form of streams; or, *(iii)* input data materializes while reasoning as a consequence of previous actions, like in video-games.

Indeed, *Logic Programming* has been applied in many different areas, and it is best used for problems where a properly defined search space can be identified and has to be explored in full. For this reason, almost all the approaches tried so far have focused on problems where the amount of data is not huge, the data are all stored in few well-defined sources, and they are often completely available at the beginning of the computation. Therefore, most of the methodologies and the procedures designed and developed up to now focused on this kind of problems, for instance taking ideas and optimizations from the *DataBase* (*DB*) world. However, the scenarios described above cannot be easily addressed using these methods and they need tailored ideas and solutions that make these activities computationally feasible and reliable.

We thus decided to face the challenges of such non-conventional settings, by proposing a number of specific methods, that show which and how *Logic Programming* can play a role.

Among the conclusion drawn, we must note that current approaches are often specific for a fixed language or tool and do not take advantage of the opportunities that may result from the mixing of diverse paradigms. We strongly believe that it is important to combine different reasoning methods and technologies. Only a proper combination of them can lead to a useful “intelligent system”.

Also, we observed that it turns out that *Logic Programming* is not suitable for every reasoning/computational task, as it shows some gaps that are confirmed to be difficult to close.
Such gaps are not only performance-related but also linked to a general difficulty in leveraging *Logic Programming* in real world application: lack of tools, integration with other paradigms, scarce familiarity in the conventional programmer community, etc.

Nonetheless, we believe that a declarative approach is necessary in a world where also people that are non-professional programmers are looking for solutions to effectively automate decision-making and can be also quite useful for advanced programmers to be more productive.

This thesis is organized as follows.
In Chapter 1 we provide some preliminaries about *Logic Programming* and most of the formalisms used later on. In Chapter 2 and in Chapter 3, after some preliminaries to introduce the topics we discuss our achievements in the *Stream Reasoning* and the *Big Data* fields, respectively. In Chapter 4 we introduce the vast world of *Artificial Intelligence* in Games and we illustrate our contribution in this setting. In Chapter 5 we describe some frameworks and tools we developed in order to facilitate the adoption and the employment of *Logic Programming* and solve some classic and popular obstacles of this paradigm. The work ends with conclusions and an outlook on future work.

1 Logic Programming

“ *‘Contrariwise,’ continued Tweedledee, ‘if it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.’*

— **Lewis Carroll (Charles Lutwidge Dodgson)**
(Through the Looking-Glass)

Summary of Chapter 1

The idea of *Logic Programming* is to represent a given computational problem by means of a logic program, whose intended *models* correspond to solutions; hence, a *solver* can be used in order to actually find such solutions [206, 343, 373, 386]. The programmer, hence, can concentrate on what the problem is, and get rid of the burden of defining how to solve it; in other words, *Logic Programming* paves the way to purely declarative programming.

The scientific community started to work on *Logic Programming* in the 1970’s, as a consequence of works in automatic theorem proving and *Artificial Intelligence*, with the aim of obtaining automated deduction systems. Since then, different improvements have been made and they have been applied to several new applications scenarios.

In this chapter, we introduce the basic notions about *Logic Programming* and some of the main formalisms and solvers that are used in the following chapters.

Chapter Outline

1.1 Definition and Motivation[1]

1.1.1 What *Logic Programming* is

Logic Programming is a high level, “human-oriented” language for describing problems and problem-solving methods to computers.

It is a type of programming paradigm which is based on formal logic.
The main idea is that sentences in *First-Order* (predicate) logic can be usefully interpreted as programs.

The interpretation of logic as a programming language is based upon the interpretation of implications of the form:

$$H\ if\ B_1\ and\ \ldots\ and\ B_n$$

H is called the head and $B_1\ and\ \ldots\ and\ B_n$ is called the body.

Usually, a program written in a *Logic Programming* language is a set of sentences, called *facts* and *rules*. *Rules* are of the form of the implication mentioned above, *facts* are rules that have no body.

Some of the most common *Logic Programming* languages are *Prolog*, *Datalog* and *Answer Set Programming* (*ASP*). It is worth noticing that they are *declarative* languages, i.e. they describe relationships between variables (the logic of computation) without describing an explicit sequence of step to follow or an explicit manipulation of the computer’s internal state (the control flow).

***Prolog*[2]** It was Philippe Roussel chose the name as an abbreviation for ’PROgrammation en LOGique’ to refer to the software tool designed to implement a man-machine communication system in natural language. It can be said that *Prolog* was the offspring of a successful marriage between natural language processing and automated theorem-proving. The idea of using a natural language like French to reason and communicate directly with a computer seemed like a crazy idea, yet this was the basis of the project set up by Alain Colmerauer in the summer of ’70. The main idea was to have a tool for the syntactic and semantic analysis of natural language, by considering *First-Order* logic not only as a programming language but also as a knowledge representation language. In other words, the logical formulation was to serve not only for the different modules of the natural language

[1]Preliminary definitions adapted from [206, 272, 340, 341, 386]
[2]Preliminary definitions adapted from [157]

dialogue system but also for the data exchanged between them, including the data exchanged with the user. *Prolog* has then been applied in various problem-solving areas and has become the most well-known and the most widely used *Logic Programming* language. For more information about *Prolog* see also [99, 151, 156, 343, 451, 544].

One of the main ideas of *Logic Programming*, which is due to Kowalski, is that an algorithm consists of two disjoint components, the Logic and the Control. The Logic component which specifies what is to be done, i.e. is the statement of *what* the problem is that has to be solved. The Control component which determines how it is to be done, i.e. is the statement of *how* it is to be solved. Generally speaking, a *Logic Programming* system should provide ways for the programmer to specify each of these components. However, separating these two components brings a number of benefits, not least of which is the possibility of the programmer only having to specify the logic component of an algorithm and leaving the control to be exercised solely by the *Logic Programming* system itself. In other words, an ideal of *Logic Programming* is purely declarative programming.

1.1.2 Why *Logic Programming* is important

The purpose of programming languages is to enable the communication from man-to-machine of problems and their general means of solutions.

The first programming languages were machine languages. To communicate, the programmer had to learn the psychology of the machine and to express his problems in machine-oriented terms. Higher-level languages developed from machine languages through the provision of facilities for the expression of problems in terms closer to their original conceptualization.

Concerned with the other end of the man-to-machine communication problem, *Logic Programming* derives from efforts to formalize the properties of rational human thought. For a long time, it was studied with little interest in its potential as a language for man-machine communication. This potential has been realized by the discoveries in *Computational Logic* of '70 which have made possible the interpretations of sentences in predicate logic as programs, of derivations of computations and of proof procedures as feasible executors for logic programs.

As programming language, Logic is the language which is entirely user-oriented. It differs from high-level languages in that it possesses no features which are meaningful only in machine level terms. It differs from functional languages, based on

$\lambda - calculus$, in that it derives from the normative study of human logic, rather than from investigations into the mathematical logic of functions.

Moreover, it has been used for several ambitious programming tasks. For instance, Computational Logic, as used in *Artificial Intelligence*, is the agent's language of thought[3]. Sentences expressed in this language represent the agent's beliefs about the world as it is and its goals for the way it would like it to be. The agent uses its goals and beliefs to control its behaviour.

Furthermore, *Logic Programming* has been used successfully for *Knowledge Representation and Reasoning* (*KR&R*). If we want to design an entity (a machine or a program) capable of behaving intelligently in some environment, then we need to supply this entity with sufficient knowledge about this environment. To do that, we need an unambiguous language capable of expressing this knowledge, together with some precise and well-understood way of manipulating sets of sentences of the language which will allow us to draw inferences, answer queries, and update both the *Knowledge Base* and the desired program behaviour. A good knowledge representation language should allow construction of *elaboration tolerant* knowledge bases, i.e., bases in which small modifications of the informal body of knowledge correspond to small modifications of the formal base representing this knowledge. Around 1960, McCarthy [413, 414] proposed the use of logical formulas as a basis for a knowledge representation language of this type. It was soon suggested, however, that this tool is not always adequate [427]. This may be especially true in modelling common-sense behaviour of agents when additions to the agent's knowledge are frequent and inferences are often based on the absence of knowledge. It seems that such reasoning can be better modelled by logical languages with non-monotonic consequence relations which allow new knowledge to invalidate some of the previous conclusions.

After years of research, the theoretical properties of several flavours of *Logic Programming* are well understood, and solving technologies, as evidenced by the availability of a number of robust and efficient systems, are mature for practical applications and nowadays many different domains have increasingly employed logic formalisms. In the following sections, more details about different *Logic Programming* languages, some of which non-monotonic, are described in order to clearly define their potentialities, their weaknesses and the main differences among them. This should provide a good foundation to justify the choices made in the various scenarios described in the next chapters.

For more information about *Logic Programming* see also [87, 342, 517].

[3]In *Artificial Intelligence*, an agent is an entity, embedded in a real or artificial world, that can observe the changing world and perform actions on the world

1.2 *Datalog and Datalog*$^{\pm}$[4]

Datalog [4] is a well-known *DataBase* query language based on the based on the *Logic Programming* paradigm that uses disjunction-free logic programs, where no functions are allowed [141, 567].

The *Datalog* language has been designed and intensively studied in the *DataBase* community from the '80. The focus has been mostly concentrated on well-formalized issues, like the definition of the rule-based language and the definition of optimization methods for various types of *Datalog* rules, together with the study of their efficiency. In parallel, various experimental projects have shown the feasibility of *Datalog* programming environments.

1.2.1 Language Definition[5]

Language Syntax

Datalog is in many respects a simplified version of general *Logic Programming*. A logic program consists of a finite set of *facts* and *rules*. Facts are assertions about a relevant piece of the world, rules are sentences which allow us to deduce facts from other facts. Note that rules, in order to be general, usually contain *variables*. Both facts and rules are particular forms of *knowledge*.

In the formalism of *Datalog* both facts and rules are represented as *Horn clauses* of the general shape

$$L_0 :- L_1, \dots, L_n$$

where each L_i, is a literal of the form $p_i(t_1, \dots, t_{k_i})$ such that p_i is a *predicate symbol* and the t_j are *terms*. A term is either a *constant* or a *variable*. The left-hand side (LHS) of a *Datalog* clause is called its *head* and the right-hand side (RHS) is called its *body*. The body of a clause may be empty. Clauses with an empty body represent facts; clauses with at least one literal in the body represent rules.

We will use the following notational convention: constants and predicate symbols are strings beginning with a lower-case character; variables are strings beginning with an upper-case character. Note that for a given *Datalog* program it is always clear from the context whether a particular non-variable symbol is a constant or a predicate symbol. We require that all literals with the same predicate symbol are of

[4]Preliminary definitions adapted from [140, 195, 196]
[5]Preliminary definitions adapted from [140]

the same *arity*, i.e., that they have the same number of arguments. A literal, fact, rule, or clause which does not contain any variables is called *ground*.

Any *Datalog* program P must satisfy the following *safety* conditions:

- Each fact of P is ground.
- Each variable which occurs in the head of a rule of P must also occur in the body of the same rule.

These conditions guarantee that the set of all facts that can be derived from a *Datalog* program is finite.

Datalog and Relational Databases

In the context of general *Logic Programming* it is usually assumed that all the knowledge (facts and rules) relevant to a particular application is contained within a single logic program P (or two of them, one for the "facts" and one for the "rules", that can be eventually combined in a single one). *Datalog*, on the other hand, has been developed for applications which use a large number of facts stored in a relational *DataBase*. Therefore, we will always consider two sets of clauses: a set of ground facts, called the *Extensional DataBase* (*EDB*), physically stored in a relational *DataBase*, and a *Datalog* program P called the *Intensional DataBase* (*IDB*). The predicates occurring in the *EDB* and in P are divided into two disjoint sets: the *EDB*-predicates, which are all those occurring in the *Extensional DataBase* and the *IDB*-predicates, which occur in P but not in the *EDB*. We require that the head predicate of each clause in P be an *IDB*-predicate. *EDB*-predicates may occur in P, but only in clause bodies.

Ground facts are stored in a relational *DataBase*; we assume that each *EDB*-predicate r corresponds to exactly one relation R of our *DB* such that each fact $r(c_1, \ldots, c_n)$ of the *EDB* is stored as a tuple $\langle c_1, \ldots, c_n \rangle$ of R.

Also, the *IDB*-predicates of P can be identified with relations, called *IDB*-relations, or also derived relations, defined through the program P and the *EDB*. *IDB* relations are not stored explicitly and correspond to relational views. The materialization of these views, i.e., their effective (and efficient) computation, is the main task of a *Datalog* compiler or interpreter.

Note that a *Datalog* program can be considered as a query against an *Extensional DataBase*, producing as answer some relations. In this setting, the distinction between the two sets of clauses makes yet more sense. Usually, a *DataBase* (in our case the *EDB*) is considered as a time-varying collection of information. A query (in our case, a program P), on the other hand, is a time-invariant mapping which associates a result to each possible *DataBase* state. For this reason, we will formally define the semantics of a *Datalog* program P as a mapping from *DataBase* states to result states. The *DataBase* states are collections of *EDB*-facts and the result states are *IDB*-facts.

Usually *Datalog* programs define large *IDB*-relations. It often happens that a user is interested in a subset of these relations. To express such an additional constraint, one can specify a *goal* to a *Datalog* program. A goal is a single literal preceded by a question mark and a dash ($?-$). Goals usually serve to formulate ad-hoc queries against a view defined by a *Datalog* program.

Language Semantics

Each *Datalog* fact F can be identified with an atomic formula F^* of *First-Order* Logic. Each *Datalog* rule R of the form $L_0 :- L_1, \dots, L_n$ represents a *First-Order* formula R^* of the form $\forall X_1, \dots, \forall X_m(L_1 \wedge \cdots \wedge L_n \Rightarrow L_0)$, where $X_1, \dots, X_m$ are all the variables occurring in R. A set S of *Datalog* clauses corresponds to the conjunction S^* of all formulas C^* such that $C \in S$. The *Herbrand Base* HB is the set of all facts that we can express in the language of *Datalog*, i.e., all literals of the form $p(c_1, \dots, c_k)$ such that all c_i are constants. Furthermore, let EHB denote the extensional part of the *Herbrand Base*, i.e., all literals of HB whose predicate is an *EDB*-predicate and, accordingly, let IHB denote the set of all literals of HB whose predicate is an *IDB*-predicate. If S is a finite set of *Datalog* clauses, we denote by $cons(S)$ the set of all facts that are logical consequences of S^*.

The semantics of a *Datalog* program P can now be described as a mapping $\mathfrak{M}_P$ from EHB to IHB which to each possible *Extensional DataBase* $E \in EHB$ associates the set $\mathfrak{M}_P(E)$ of intensional "result facts" defined by $\mathfrak{M}_P(E) = cons(P \cup E) \cap IHB$. Let K and L be two literals (not necessarily ground). We say that K *subsumes* L, denoted by $K \rhd L$, if there exists a substitution Θ of variables such that $K\Theta = L$, i.e., if Θ applied to K yields L. If $K \rhd L$ we also say that L is an instance of K. When a goal "$? - G$" is given, then the semantics of the program P w.r.t. this goal is a mapping $\mathfrak{M}_{P,G}$, from EHB to IHB defined as follows

$$\forall E \subseteq EHB \; \mathfrak{M}_{P,G}(E) = \{H | H \in \mathfrak{M}_P(E) \wedge G > H\}$$

An *Interpretation* (in the context of *Datalog*) consists of an assignment of a concrete meaning to constant and predicate symbols. A *Datalog* clause can be interpreted in several different ways. A clause may be true under a certain interpretation and false under another one. If a clause C is true under a given interpretation, we say that this interpretation satisfies C.

The concept of logical consequence, in the context of *Datalog*, can be defined as follows: a fact F follows logically from a set S of clauses, iff each interpretation satisfying every clause of S also satisfies F. If F follows from S, we write $S \models F$.

Note that this definition captures quite well our intuitive understanding of logical consequence. However, since general interpretations are quite unhandy objects, we will limit ourselves to consider interpretations of a particular type, called *Herbrand Interpretations*.

A *Herbrand Interpretation* assigns to each constant symbol "itself", i.e., a lexicographic entity. Predicate symbols are assigned predicates ranging over constant symbols. Thus, two non-identical *Herbrand Interpretations* differ only in the respective interpretations of the predicate symbols. For this reason, any *Herbrand Interpretation* can be identified with a subset $\mathfrak{I}$ of the *Herbrand Base* HB. This subset contains all the ground facts which are true under the interpretation. Thus, a ground fact $p(c_1, \ldots, c_n)$ is true under the interpretation $\mathfrak{I}$ iff $p(c_1, \ldots, c_n) \in \mathfrak{I}$. A *Datalog* rule of the form $L_0 : - L_1, \ldots, L_n$ is true under $\mathfrak{I}$ iff for each substitution Θ which replaces variables by constants, whenever $L_1\Theta \in \mathfrak{I} \wedge \cdots \wedge L_n\Theta \in \mathfrak{I}$ then it also holds that $L_0\Theta \in \mathfrak{I}$. A *Herbrand Interpretation* which satisfies a clause C or a set of clauses S is called a *Herbrand Model* for C or, respectively, for S.

1.2.2 Language Extensions

The *Datalog* syntax considering in the previous section corresponds to a very restricted subset of *First-Order* logic and is often referred to as *pure Datalog*. Several extensions of pure *Datalog* have been proposed in the literature. Among them, some of the most important are built-in predicates, negation, disjunction (in the head) and "complex objects" (i.e. *function symbols*).

Built-in Predicates

Built-in predicates (or "built-ins") are expressed by special predicate symbols such as $>$, $<$, $\geq$, $\leq$ $=$, $\neq$ with a predefined meaning. These symbols can occur in the right-hand side of a *Datalog* rule; they are usually written in infix notation.

From a formal point of view, built-ins can be considered as *EDB*-predicates with a different physical realization than ordinary *EDB*-predicates: they are not explicitly stored in the *EDB* but are implemented as procedures which are evaluated during the execution of a *Datalog* program. However, built-ins correspond in most cases to *infinite* relations, and this may endanger the *safety* of *Datalog* programs.

Safety, as mentioned before, means that a *Datalog* program should always have a finite output, i.e., the intensional relations defined by a *Datalog* program must be finite. It is easy to see that safety can be guaranteed by requiring that each variable occurring as argument of a built-in predicate in a rule body must also occur in an ordinary predicate of the same rule body, or must be bound by an equality (or a sequence of equalities) to a variable of such an ordinary predicate or to a constant. Here, by "ordinary predicate", we mean a non-built-in predicate.

In a similar way, *arithmetical built-in predicates* can be used.

Negation

In pure *Datalog*, the negation sign "$\neg$" is not allowed to appear. However, by adopting the *Closed World Assumption* (*CWA*) [425, 491], we may infer negative facts from a set of pure *Datalog* clauses.

In Classical Logic unstated information does not assume a truth value: that is, when an assertion is not found as a known fact, nothing can be said about its truth value. On the other hand, in the *DataBase* realm, the facts that have neither been asserted nor inferred are considered as false. The first attitude is known as the *Open World Assumption* (*OWA*), while the second is the *Closed World Assumption* (*CWA*), and each of them is perfectly coherent with the framework in which it is assumed.

The "closed" view adopted in the *DataBase* world also has two more aspects, namely the *unique name assumption*, which states that individuals with different names are different, and the *domain closure assumption*, which comes in different flavours but basically states that there are no other individuals than those in the *DataBase*.

Note that the *CWA* is not a universally valid logical rule, but just a principle that one may or may not adopt, depending on the semantics given to a language.

In the context of *Datalog*, the *CWA* can be formulated as follows:

CWA If a fact does not logically follow from a set of *Datalog* clauses, then we conclude that the negation of this fact is true.

Negative *Datalog* facts are positive ground literals preceded by the negation sign ($\neg$).

The *CWA* applied to pure *Datalog* allows us to deduce negative facts from a set S of *Datalog* clauses. It does not, however, allow us to use these negative facts in order to deduce some further facts.

More formally, let us define *Datalog*$^\neg$ as the language whose syntax is that of *Datalog* except that negated literals are allowed in rule bodies. Accordingly, a *Datalog*$^\neg$ clause is either a positive (ground) fact or a rule where negative literals are allowed to appear in the body. For safety reasons we also require that each variable occurring in a negative literal of a rule body also occurs in a positive literal of the same rule body.

In order to define the semantics of *Datalog*$^\neg$ programs, we first generalize the notion of the *Herbrand Model* to cover negation in rule bodies.
For more information about the semantics of *Datalog*$^\neg$ see also [3, 333, 510].

Moreover, a specific policy of choosing a particular *Herbrand Model*, and thus of determining the semantics of a *Datalog*$^\neg$ program, referred to as *Stratified Datalog*$^\neg$ allows to have a natural and intuitive way define a semantics. However, it does not apply to all *Datalog*$^\neg$ programs, but only to particular sub-class, the so-called *stratified* programs.
For more information about *stratified* programs see also [28, 331, 332, 508].

Disjunction

A disjunctive logic program is a logic program where disjunction may occur in the rule heads.

Many of the remarks made for *Datalog*$^\neg$ are valid also in this case.

The usefulness of disjunctive rules for *Knowledge Representation*, *DataBase* querying, and for representing incomplete information, is generally acknowledged [54, 274].

Various semantics of DLPs have been proposed; most of them are extensions of the well-known semantics of *Logic Programming* (with and without negation) and they are commonly based on the paradigm of minimal models.
For more details see [194, 274, 387, 426, 478, 479, 590].

Complex Objects

The "objects" handled by pure *Datalog* programs correspond to the tuples of relations which in turn are made of attribute values. Each attribute value is atomic, i.e., not composed of sub-objects; thus the underlying data model consists of relations in first normal form. This model has the advantage of being both mathematically simple and easy to implement. On the other hand, several application areas require the storage and manipulation of (deeply nested) structured objects of high complexity. Such complex objects cannot be represented as atomic entities in the normalized relational model but are broken into several autonomous objects. This implies a number of severe problems of conceptual and technical nature.

For this reason, the relational model has been extended in several ways to allow the compact representation of complex objects. *Datalog* can be extended accordingly. The main features that are added to *Datalog* in order to represent and manipulate complex objects are *function symbols* as a glue for composing objects from sub-objects and set constructors for being able to build objects which are collections of other objects. Function symbols are "uninterpreted", i.e., they do not have any predefined meaning. Usually one also adds a number of *predefined* functions for manipulating sets and elements of sets to the standard vocabulary of *Datalog*. There exist several different approaches for incorporating the concept of a complex structured object into the formalism of *Datalog* [318, 350, 564].

Using complex objects in *Datalog* is not as easy as it might appear. Several problems have to be taken into consideration. First of all, the use of function symbols may endanger the safety of programs. It is undecidable whether a *Datalog* program with function symbols has a finite or an infinite result. The simplest solution is to leave the responsibility to the programmer. A similar problem is the finiteness of sets. Furthermore, not all *Datalog* programs with sets have a well-defined semantics. In particular, one should avoid self-referential set definitions. Such definitions come close to Russell's paradox. A large class of programs free of self-reference, called *admissible programs*, is defined in [85]. Note also that the test whether two terms (or literals) involving sets match is a computationally hard problem. This is a particular case of *theory unification* [545].

1.2.3 *Datalog*$^{\pm}$[6]

A natural and very peculiar extension of *Datalog* programs are the so-called *existential Datalog programs*, or *Datalog*$^{\exists}$ programs for short, that extend the pure *Datalog* allowing existentially quantified variables in rule heads. This language is highly expressive and enables easy and powerful knowledge-modelling, but the presence of existentially quantified variables makes reasoning over *Datalog*$^{\exists}$ undecidable, in the general case.

Datalog$^{\exists}$ is the base of a recently introduced family of *Datalog*-based languages, called *Datalog*$^{\pm}$, which is a new framework for tractable ontology querying. *Datalog* is extended by allowing existential quantifiers, the equality predicate, and the truth constant *false* to appear in rule heads. At the same time, the resulting language

[6]Preliminary definitions adapted from [58, 118, 120, 286, 361, 459]

is syntactically restricted, so as to achieve decidability, and in some relevant cases even tractability.

In general, the *Datalog*$^{\pm}$ family intends to collect all expressive extensions of *Datalog* which are based on *Tuple-Generating Dependencies (TGDs)* (which are *Datalog*$^{\exists}$ rules with possibly multiple atoms in rule heads), *Equality-Generating Dependencies (EGDs)* and *negative constraint*. In particular, the "plus" symbol refers to any possible combination of these extensions, while the "minus" one imposes at least decidability, since, as mentioned before, *Datalog*$^{\exists}$ alone is already undecidable.

A *Datalog*$^{\exists}$ rule r is a finite expression of the form:

$$\forall X \exists Y \, atom_{[X' \cup Y]} \leftarrow conj_{[X]}$$

where (*i)* X and Y are disjoint sets of variables (next called $\forall$-variables and $\exists$-variables, respectively); (*ii)* $X' \subseteq X$; (*iii)* $atom_{[X' \cup Y]}$ stands for an atom containing only and all the variables in $X' \cup Y$; (*iv)* $conj_{[X]}$ stands for a conjunct (a conjunction of zero, one or more atoms) containing only and all the variables in X. Constants are also allowed in r. In the following, $head(r)$ denotes $atom_{[X' \cup Y]}$, and $body(r)$ the set of atoms in $conj_{[X]}$. Universal quantifiers are usually omitted to lighten the syntax, while existential quantifiers are omitted only if Y is empty. In the second case, r coincides with a standard *Datalog* rule. If $body(r) = \emptyset$, then r is usually referred to as a *fact*. In particular, r is called *existential* or *ground* fact according to whether r contains some $\exists$-variable or not, respectively. A *Datalog*$^{\exists}$ program P is a finite set of *Datalog*$^{\exists}$ rules. We denote by $preds(P) \subseteq \Pi$ the predicate symbols occurring in P, by $data(P)$ all the atoms constituting the ground facts of P, and by $rules(P)$ all the rules of P being not ground facts.

Given a *Datalog*$^{\exists}$ program P, a *Conjunctive Query* (*CQ*) $\mathcal{Q}$ over P is a *First-Order* (*FO*) expression of the form:

$$\exists Y \, conj_{[X \cup Y]}$$

where X are its free variables, and $conj_{[X \cup Y]}$ is a conjunct containing only and all the variables in $X \cup Y$ and possibly some constants. To highlight the free variables, we write $\mathcal{Q}(X)$ instead of $\mathcal{Q}$. Query $\mathcal{Q}$ is called *Boolean Conjunctive Query* (*BCQ*) if $X = \emptyset$. Moreover, $\mathcal{Q}$ is called *atomic* if $conj$ is an atom. Finally, $atoms(\mathcal{Q})$ denotes the set of atoms in $conj$.

The main reasoning task in *Datalog*$^{\pm}$ is Query Answering under the so-called *certain-answers semantics*.

If $\mathcal{Q}(X)$ is a *Conjunctive Query* (*CQ*) or a *Union of Conjunctive Queries* (*UCQ*) with free variables X, D a *DataBase* over a domain Δ whose tuples are interpreted in the usual way as ground facts, and P a *Datalog*$^{\pm}$ program, then the answer to $\mathcal{Q}$ consists of all those tuples $\mathbf{a}$ of Δ-elements such that $D \cup P \models \mathcal{Q}(\mathbf{a})$.

A number of QA-decidable (i.e. that guarantee the decidability of Query Answering) *Datalog*$^{\pm}$ languages have been defined in the literature. They rely on four main paradigms (classes):

Weakly-acyclic [179, 213, 293, 412] based on Weakly Acyclic *TGDs* introduced in the context of data exchange, guarantees the existence of a finite universal model, which in turn implies the decidability of Query Answering.

Guarded [42, 118, 119] ensures the existence of treelike universal models, which in turn implies the decidability of query answering. A rule is Guarded if it has an atom which contains all the body variables. An important subclass of Guarded *Datalog*$^{\pm}$ is *Linear Datalog*$^{\pm}$, where rules have only one body-atom.

Sticky [121, 122, 288] guarantees the termination of backward resolution, and thus the decidability of query answering. The key idea underlying Sticky is that the body-variables which are in a join always are propagated (or "stick") to the inferred atoms. The main goal of Stickiness was the definition of a language that allows for joins in rule bodies, which are not always expressible via Guarded rules.

Shy [18, 360, 361] an easily recognizable fragment of parsimonious programs, that significantly extends both *Datalog* and *Linear-Datalog*$^{\exists}$, while preserving the same (data and combined) complexity of query answering over *Datalog*, although the addition of existential quantifiers.

It is worth noticing that there are also QA-decidable "abstract" classes of *Datalog*$^{\exists}$ programs, called *Finite-Expansion-Sets*, *Finite-Treewidth-Sets*, *Finite-Unification-Sets* and *Parsimonious-Sets*, depending on semantic properties that capture the four mentioned paradigms, respectively [361, 434].

Many other paradigms have been proposed.
Notable examples are *glut-guardedness* [348] obtained by combining *weak-acyclicity* and *guardedness*, *weak-stickiness* [123] obtained by joining *weak-acyclicity* and *stickiness*, and *tameness* [287] obtained by combining *guardedness* and *stickiness*.

In short, *Datalog*$^{\pm}$ is a rule-based formalism that combines the advantages of *Logic Programming* in *Datalog* with features for expressing ontological knowledge and advanced data modelling constraints. *Datalog*$^{\pm}$ provides a uniform framework for query answering and reasoning with incomplete data.

Note that the unique least *Herbrand Model* of a *Datalog* program and a *DataBase* is always finite, and all values appearing in it are from the *active domain* of the given *EDB*, i.e., all values that appear as arguments of *EDB* facts or that are explicitly mentioned in the *Datalog* program. For ontology querying, however, it would be desirable that a *Datalog* extension could be able to express the existence of certain values that are not necessarily from the active domain of the *EDB*. This can be achieved by allowing existentially quantified variables in rule heads [463].

The *chase* is a well-known procedure that allows of answering *CQ*s. The *chase* was introduced as a procedure for testing implication of dependencies [398], but later also employed for checking query containment [314] and query answering on incomplete data under relational dependencies [124]. Informally, the *chase* procedure is a process of repairing a *DataBase* w.r.t. a set of dependencies, so that the result of the chase satisfies the dependencies. The *chase* of a *DataBase* D in the presence of a program P is the process of iterative enforcement of all dependencies in P, until a fixpoint is reached. The result of such a process, which we also call *chase*, can be infinite and, in this case, this procedure cannot be used without modifications in decision algorithms. Nevertheless, the result of a *chase* serves as a fundamental theoretical tool for answering queries in the presence of *TGD*s [124, 213] because it is representative of all models of $D \cup P$.

1.2.4 *Datalog* and *Datalog*$^{\pm}$ solvers

In the last 30 years many *Datalog* engines have been developed, using different programming languages. Most of them are not only able to evaluate only *pure Datalog* programs but they support different extensions of the language.

Among them:

Apache Jena[7]

A well-known free and open source *Java* framework for building *Semantic Web* and *Linked Data* applications that includes *Datalog* engine.

bddbddb[8] [579]

bddbddb (short for BDD-Based Deductive *DataBase*) is an implementation of standard *Datalog*. It represents the relations using binary decision diagrams (BDDs), a data structure that can efficiently represent large relations and provide efficient set operations. This allows bddbddb to efficient represent and operate on extremely large relations.

[7]From `https://jena.apache.org`
[8]From `http://bddbddb.sourceforge.net`
[9]From `http://research.cs.wisc.edu/coral`

Coral[9] [486]

A deductive system which supports a rich declarative language, and an interface to *C++* which allows for a combination of declarative and imperative programming. The declarative query language supports general Horn clauses augmented with complex terms, set-grouping, aggregation, negation, and relations with tuples that contain (universally quantified) variables.

Inter4QL[10] [535]

An open-source command-line interpreter of the 4QL language implemented in *C++*. 4QL is a rule-based *DataBase* query language with negation allowed in premises and conclusions of rules. 4QL is founded on a four-valued semantics with truth values: true, false, inconsistent and unknown.

IRIS[11] [90]

IRIS (short for Integrated Rule Inference System) is an extensible reasoning engine for expressive rule-based languages and aims to be a framework consisting of a collection of components which cover various aspects of reasoning with formally represented knowledge.

pyDatalog[12]

Adds the *Logic Programming* paradigm to *Python*'s extensive toolbox, in a pythonic way. Logic programmers can use the extensive standard library of *Python*, and *Python* programmers can express complex algorithms quickly.

LogicBlox[13] [35]

A commercial product that aims to redesign the enterprise software stack centring it around a unified declarative programming model, based on an extended version of *Datalog*.

Soufflé[14] [317]

A translator of declarative *Datalog* programs into the *C++* language. It is used as a domain-specific language for static program analysis, over large code bases with millions of lines of code. It aims at producing high-performance *C++* code that can be compiled natively on the target machine.

XSB[15] [511]

A in-memory *Logic Programming* and Deductive *DataBase* system for Unix and Windows. The XSB system is an open-source multi-threaded *Logic Programming* system that extends *Prolog* with new semantic and operational features, mostly based on the use of Tabled *Logic Programming* or tabling.

DLV Extensively described in Section 1.3.5

[10]From http://4ql.org
[11]From https://github.com/NICTA/iris-reasoner
[12]From https://sites.google.com/site/pydatalog
[13]From http://www.logicblox.com
[14]From http://souffle-lang.org
[15]From http://xsb.sourceforge.net

$\mathcal{I}$-DLV[16] [133]

The new intelligent grounder of DLV (part of DLV2, described in Section 1.3.5). It is an *ASP* instantiator that natively supports the *ASP-Core-2* standard language. Its core instantiation mechanism is based on semi-naive *DataBase* techniques. $\mathcal{I}$-DLV is also a full-fledged deductive *DataBase* system, supporting query answering powered by the *Magic Sets technique*.

In the *Datalog*$^{\pm}$ framework various systems for different paradigms have been developed:

Nyaya [571]

A system able to treat the *First-Order* rewritable fragments of *Datalog*$^{\pm}$, that is, *linear* and *sticky Datalog*$^{\pm}$. In fact, the given set of rules and query are compiled into an *SQL* query, which is then evaluated over the *Extensional DataBase*.

DLV$^{\exists}$[17] [361]

A powerful system for answering conjunctive queries over *shy Datalog*$^{\pm}$ programs. It implements a bottom-up evaluation strategy inside the well-known *Answer Set Programming* (*ASP*) system DLV.

Alaska [337]

Similarly to Nyaya, is able to treat the *First-Order* rewritable fragments of *Datalog*$^{\pm}$, and it is based on *SQL*-rewritings (backward chaining paradigm).

IRIS$^{\pm}$ [289]

An extension of the IRIS *Datalog* engine implementing the XRewrite algorithm and its optimizations techniques. XRewrite is a query rewriting technique for linear and sticky *TGDs* that is based on backward chaining resolution.

RDFox[18] [440]

A highly scalable in-memory *RDF* triple store that supports shared memory parallel *Datalog* reasoning. It uses novel and highly-efficient parallel reasoning algorithms (FBF) for the computation and incremental update of *Datalog* materialisations.

Graal[19] [41]

A *Java* toolkit dedicated to querying knowledge bases within the framework of *Datalog*$^{\pm}$. It takes as input a DLGP[20] file and a query and answers the query using various means (saturation, query rewriting).

[16] From https://github.com/DeMaCS-UNICAL/I-DLV/wiki
[17] From https://www.mat.unical.it/kr2012
[18] From https://www.cs.ox.ac.uk/isg/tools/RDFox
[19] From http://graphik-team.github.io/graal
[20] A textual format for *Datalog*$^{+}$

1.3 *Answer Set Programming (ASP)*[21]

The need for representing and manipulating complex knowledge arising in *Artificial Intelligence* and in other emerging areas, like Knowledge Management and Information Integration, has renewed the interest in advanced logic-based formalisms for *Knowledge Representation and Reasoning* (*KR&R*), which started in the early 1980s. Among them, *Disjunctive Logic Programming* (*DLP*), which has first been considered by Minker [425] in the deductive *DataBase* context, is one of the most expressive *KR&R* formalisms.

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of the rules and negation may occur in the bodies of the rules. One of the attractions of *Disjunctive Logic Programming* is its capability of allowing the natural modelling of incomplete knowledge. *DLP* is an advanced formalism for *Knowledge Representation and Reasoning*, which is very expressive in a precise mathematical sense: it allows to express every property of finite structures that is decidable in the complexity class $\Sigma_2^P(\mathsf{NP}^{\mathsf{NP}})$. Thus, under widely believed assumptions, *DLP* is strictly more expressive than *normal* (*disjunction-free*) *Logic Programming*, whose expressiveness is limited to properties decidable in NP. Importantly, apart from enlarging the class of applications which can be encoded in the language, disjunction often allows for representing problems of lower complexity in a simpler and more natural fashion.

The most widely accepted semantics is the *Answer Sets* semantics proposed by [274] as an extension of the stable model semantics of normal logic programs [273]. According to this semantics, a disjunctive logic program may have several alternative models (but possibly none), called *Answer Sets*, each corresponding to a possible view of the world.

Answer Set Programming (*ASP*) [52] is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. As an outgrowth of research on the use of non-monotonic reasoning in *KR*, it is particularly useful in knowledge-intensive applications. *ASP* is based on the stable model (*Answer Set*) semantics of *Logic Programming* [273], which applies ideas of auto-epistemic logic [429] and default logic [490] to the analysis of negation as failure.

ASP has a close relationship to other formalisms such as ***propositional satisfiability (SAT)*** [50], *Satisfiability Modulo Theories* [432, 444], *Constraint Handling Rules* [226], *Quantified Boolean Formula* [512], *Planning Domain Definition Language* [276, 415], and many others.

[21] Preliminary definitions adapted from [127, 132, 264, 363, 374, 375]

Answer Set Programming has become a popular approach to declarative problem-solving in the field of *Knowledge Representation and Reasoning* (*KR&R*). This is mainly due to its appealing combination of a rich yet simple modelling language with high-performance solving capacities.

ASP has its roots in:

- Knowledge Representation and (Non-monotonic) Reasoning,
- *Logic Programming* (with negation),
- Databases, and
- Boolean Constraint Solving.

The fully declarative nature of *ASP* allows one to encode a large variety of problems by means of simple and elegant logic programs.

The basic idea of *ASP* is to represent a given computational problem by a logic program whose *Answer Sets* correspond to solutions, and then to use an ***ASP solver*** for finding *Answer Sets* of the program. That is, to model a given problem domain and contingent input data with a *Knowledge Base* (*KB*) composed of logic assertions, such that the logic models (*Answer Sets*[22]) of *KB* correspond to solutions of an input scenario (as shown in Section 1.3.4).

***ASP* Solving** As with traditional computer programming, the *ASP* solving process amounts to a closed loop (shown in Figure 1.1).

Its steps can be roughly classified into:

1. Modelling,
2. Grounding,
3. Solving,
4. Visualizing, and
5. Software Engineering.

See also [325].

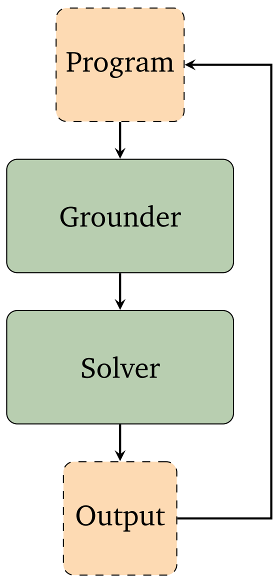


Figure 1.1.: Basic *ASP* Solving Iterative Workflow.

[22]An *ASP Knowledge Base* might have none, one or many *Answer Sets*, depending on the problem and the instance at hand

Many different extensions and variants are continuously introduced and formally analysed, and connections with other formalisms are constantly studied (examples are [69, 105, 106, 269, 299, 462, 563]), below we introduce some of them, which are used in the following chapters.

1.3.1 Language Definition[23]

It is worth recalling that a significant amount of work has been carried out by the scientific community for extending the basic language, in order to increase the expressive power and improve usability of the formalism. This has led to a variety of *ASP* "dialects", supported by a corresponding variety of *ASP* systems, that only share a portion of the basic language. Notably, the community recently agreed on the definition of a standard input language for *ASP* systems, namely *ASP-Core-2* [126], which is also the official language of the *ASP* Competition series [268]; it features most of the advanced constructs and mechanisms with a well-defined semantics that has been introduced and implemented in the latest years.

Language Syntax

The language described is disjunctive *Datalog* under the *Answer Sets* semantics [274] (which involves two kinds of negation), extended with weak constraints as specified by the *ASP Standardization Working Group* in the "ASP-Core-2 – Input Language Format" document [126].

An *ASP program* is a set of rules and weak constraints, possibly accompanied by a (single) query.[24]

A *rule* has form

$$h_1 \mid \ \ldots \ \mid h_m \leftarrow b_1, \ldots, b_n.$$

where $h_1, \ldots, h_m$ are classical atoms and $b_1, \ldots, b_n$ are literals for $m, n \geq 0$.

A *predicate atom* has form $p(t_1, \ldots, t_n)$, where p is a *predicate name*, $t_1, \ldots, t_n$ are terms and $n \geq 0$ is the arity of the predicate atom; a predicate atom $p()$ of arity 0 is likewise represented by its predicate name p without parentheses. Given a predicate atom q, q and $\neg q$ are *classical atoms*.

[23]Preliminary definitions adapted from [17, 126]

[24]Unions of conjunctive queries (and more) can be expressed by including appropriate rules in a program.

A *built-in atom* has form $t \prec u$ for terms t and u with $\prec \in \{``<", ``\leq", ``=", ``\neq", ``>", ``\geq"\}$. Built-in atoms a as well as the expressions a and $\mathbf{not}\ a$ for a classical atom a are *naf-literals*.

An *aggregate element* has form

$$t_1, \ldots, t_m : l_1, \ldots, l_n$$

where $t_1, \ldots, t_m$ are terms and $l_1, \ldots, l_n$ are naf-literals for $m \geq 0$ and $n \geq 0$.

An *aggregate atom* has form

$$\#\text{aggr}\{e_1; \ldots; e_n\} \prec u$$

where $e_1, \ldots, e_n$ are aggregate elements for $n \geq 0$, $\#\text{aggr} \in \{``\#\text{count}", ``\#\text{sum}", ``\#\text{max}", ``\#\text{min}"\}$ is an *aggregate function name*, $\prec \in \{``<", ``\leq", ``=", ``\neq", ``>", ``\geq"\}$ is an *aggregate relation* and u is a term. Given an aggregate atom a, the expressions a and $\mathbf{not}\ a$ are *aggregate literals*.

Terms are either *constants*, *variables*, *arithmetic terms* or *functional terms*. Constants can be either *symbolic constants* (strings starting with some lowercase letter), *string constants* (quoted strings) or *integers*. Variables are denoted by strings starting with some uppercase letter. An *arithmetic term* has form $-(t)$ or $(t \diamond u)$ for terms t and u with $\diamond \in \{``+", ``-", ``*", ``/"\}$; parentheses can optionally be omitted in which case standard operator precedences apply. Given a *functor* f (the *function name*) and terms $t_1, \ldots, t_n$, the expression $f(t_1, \ldots, t_n)$ is a *functional term* if $n > 0$, whereas $f()$ is a synonym for the symbolic constant f.

A *weak constraint* has form

$$:\sim b_1, \ldots, b_n. \, [w@l, t_1, \ldots, t_m]$$

where $t_1, \ldots, t_m$ are terms and $b_1, \ldots, b_n$ are literals for $m \geq 0$ and $n \geq 0$; w and l are terms standing for a *weight* and a *level*. Writing the part "$@l$" can optionally be omitted if $l = 0$; that is, a weak constraint has level 0 unless specified otherwise.

A *query* Q has form $a?$, where a is a classical atom.

A program (rule, weak constraint, query, literal, aggregate element, etc.) is *ground* if it contains no variables.

There are some syntactic shortcuts, like *Anonymous Variables*, *Choice Rules* and *Aggregate Relations*, which make writing *ASP* programs easier.

Language Semantics

The semantics of a program extends the *Answer Sets* semantics of disjunctive *Datalog* programs, originally defined in [274]. A program can be used to model a problem to be solved: the problem's solutions correspond to the *Answer Sets* of the program (which are computed by the ***ASP solver***). Therefore, a program may have no *Answer Set* (if the problem has no solution), one (if the problem has a unique solution) or several (if the problem has more than one possible solutions).

Given a program P, the *Herbrand Universe* of P, denoted by U_P, consists of all integers and (ground) terms constructible from constants and functors appearing in P. The *Herbrand Base* of P, denoted by B_P, is the set of all (ground) classical atoms that can be built by combining predicate names appearing in P with terms from U_P as arguments. A (Herbrand) *interpretation* I for P is a *consistent* subset of B_P; that is, $\{q, \neg q\} \not\subseteq I$ must hold for each predicate atom $q \in B_P$.

A *substitution* σ is a mapping from a set V of variables to the *Herbrand Universe* U_P of a given program P. For some object O (rule, weak constraint, query, literal, aggregate element, etc.), denote by $O\sigma$ the object obtained by replacing each occurrence of a variable $v \in V$ by $\sigma(v)$ in O.

A variable is *global* in a rule, weak constraint or query r if it appears outside of aggregate elements in r. A substitution from the set of global variables in r is a *global substitution for* r; a substitution from the set of variables in an aggregate element e is a (local) *substitution for* e. A global substitution σ for r (or substitution σ for e) is *well-formed* if the arithmetic evaluation, performed in the standard way, of any arithmetic sub-term $(-(t)$ or $(t \diamond u)$ with $\diamond \in \{\text{“}+\text{”}, \text{“}-\text{”}, \text{“}*\text{”}, \text{“}/\text{”}\})$ appearing outside of aggregate elements in $r\sigma$ (or appearing in $e\sigma$) is well-defined.

Given a collection $\{e_1; \ldots; e_n\}$ of aggregate elements, the *instantiation* of $\{e_1; \ldots; e_n\}$ is the following set of aggregate elements:

$$\text{inst}(\{e_1; \ldots; e_n\}) = \bigcup_{1 \leq i \leq n} \{\ e_i\sigma \mid \sigma \text{ is a well-formed substitution for } e_i\ \}$$

A *ground instance* of a rule, weak constraint or query r is obtained in two steps: (1) a well-formed global substitution σ for r is applied to r; (2) for every aggregate atom $\#\text{aggr}\{e_1; \ldots; e_n\} \prec u$ appearing in $r\sigma$, $\{e_1; \ldots; e_n\}$ is replaced by $\text{inst}(\{e_1; \ldots; e_n\})$ (where aggregate elements are syntactically separated by ";").

The *arithmetic evaluation* of a ground instance r of some rule, weak constraint or query is obtained by replacing any maximal arithmetic sub-term appearing in r by its integer value, which is calculated in the standard way.

The *ground instantiation* of a program P, denoted by $grnd(P)$, is the set of arithmetically evaluated ground instances of rules and weak constraints in P.

Given a program P and a (consistent) interpretation $I \subseteq B_P$, a rule $h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n.$ in $grnd(P)$ is *satisfied* w.r.t. I if some $h \in \{h_1, \ldots, h_m\}$ is *true* w.r.t. I when $b_1, \ldots, b_n$ are *true* w.r.t. I; I is a *model* of P if every rule in $grnd(P)$ is satisfied w.r.t. I. The *reduct* of P w.r.t. I, denoted by P^I, consists of the rules $h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n.$ in $grnd(P)$ such that $b_1, \ldots, b_n$ are *true* w.r.t. I; I is an *Answer Set* of P if I is a $\subseteq$-minimal model of P^I. In other words, an *Answer Set* I of P is a model of P such that no proper subset of I is a model of P^I.

The semantics of P is given by the collection of its *Answer Sets*, denoted by $AS(P)$.

To select optimal members of $AS(P)$, map an interpretation I for P to a set of tuples as follows:

$$\begin{aligned}\mathrm{weak}(P, I) = \{(w@l, t_1, \ldots, t_m) \mid \; & \\ & {:\sim}\, b_1, \ldots, b_n.\, [w@l, t_1, \ldots, t_m] \text{ occurs in } grnd(P) \\ & \text{and } b_1, \ldots, b_n \text{ are } \textit{true} \text{ w.r.t. } I\; \}\end{aligned}$$

For any integer l, let

$$P_l^I = \sum_{\substack{(w@l, t_1, \ldots, t_m) \,\in\, \mathrm{weak}(P,I), \\ w \text{ is an integer}}} w$$

denote the sum of integers w over tuples with $w@l$ in $\mathrm{weak}(P, I)$. Then, an *Answer Set* $I \in AS(P)$ is *dominated* by $I' \in AS(P)$ if there is some integer l such that $P_l^{I'} < P_l^I$ and $P_{l'}^{I'} = P_{l'}^I$ for all integers $l' > l$. An *Answer Set* $I \in AS(P)$ is *optimal* if there is no $I' \in AS(P)$ such that I is dominated by I'. Note that P has some (and possibly more than one) optimal *Answer Set* if $AS(P) \neq \emptyset$.

Given a program P along with a (single) query $a?$, let $Ans(a, P)$ denote the set of arithmetically evaluated ground instances a' of a such that $a' \in I$ for all $I \in AS(P)$. The set $Ans(a, P)$, which includes all arithmetically evaluated ground instances of a if $AS(P) = \emptyset$, constitutes the *answers* to $a?$. That is, query answering corresponds to cautious (or skeptical) reasoning as defined in [4].

Given a ground query $Q = q?$ of a program P, Q is true if $\forall I \in AS(P)$ q is true w.r.t. I. Otherwise, Q is false. Note that, if $AS(P) = \emptyset$, all queries are true.

Given the non-ground query $Q = q(t_1, \ldots, t_n)?$ of a program P, let $Ans(Q, P)$ be the set of all substitutions σ for Q such that $Q\sigma$ is true. The set $Ans(Q, P)$ constitutes the set of answers to Q.

1.3.2 *Knowledge Representation*

From the perspective of *KR*, a set of atoms can be thought of as a description of a complete state of knowledge: the atoms that belong to the set are known to be true, and the atoms that do not belong to the set are known to be false. A possibly incomplete state of knowledge can be described using a consistent but possibly incomplete set of literals; if an atom p does not belong to the set and its negation does not belong to the set either then it is not known whether p is true. In the context of *Logic Programming*, this idea leads to the need to distinguish between two kinds of negation: negation as failure and strong (or "classical") negation. Combining both forms of negation in the same rule allows expressing the *Closed World Assumption* – the assumption that a predicate does not hold whenever there is no evidence that it does [491]. An *ASP* program with strong negation can include the *Closed World Assumption* rules for some of its predicates and leave the other predicates in the realm of the *Open World Assumption*.

Answer Set Programming has been applied to several areas of science and technology and it is nowadays employed in a variety of applications, ranging from classical *AI* to real-world and industrial applications. In fact, main strength of *Answer Set Programming* is its wide range of applicability as a tool for *Knowledge Representation and Reasoning*. For instance, it can be used to encode in a highly declarative fashion problems in the "Deductive Databases" field (like "Reachability" and "Same Generation"), "Search" problems (like "Seating" and "Ramsey Numbers"), "Optimization Problems" (like "Maximal Cut" and "Exam Scheduling") and many others (see [17, 47, 137, 207, 253, 363, 364, 422, 445, 561]).

1.3.3 Declarative Programming Methodology

From the perspective of *Answer Set Programming*, two kinds of rules play a special role: those that generate multiple *Answer Sets* and those that can be used to eliminate some of the *Answer Sets* of a program.
This observation leads to the development of the "Guess&Check" programming methodology (that is closely related to the "Generate/Define/Test" methodology). An extension and refinement of the "Guess&Check" methodology is the *Guess/Check-/Optimize* (*GCO*) methodology, which can be described as follows.

Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem $\mathbf{P}$, a *GCO* program $\mathcal{P}$ for $\mathbf{P}$ consists of the following three main parts:

Guessing Part The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, such that *Answer Sets* of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

Checking Part The (optional) checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program filters the solution candidates in such a way that the *Answer Sets* of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the admissible solutions for the problem instance I.

Optimization Part The (optional) optimization part $\mathcal{O} \subseteq \mathcal{P}$ of the program allows expressing a quantitative cost evaluation of solutions by using weak constraints. It implicitly defines an objective function $f : AS(\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I) \to \mathbb{N}$ mapping the *Answer Sets* of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ to natural numbers. The semantics of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ optimizes f by filtering those *Answer Sets* having the minimum value; this way, the optimal (least cost) solutions are computed.

For more information about *ASP* modelling/methodologies see also [98, 257, 269].

1.3.4 *KR&R* with *ASP*: some examples[25]

In the following, we show how its fully declarative nature allows encoding a large variety of problems via simple and elegant logic programs.

[3-COL] As a first example, let us consider the well-known 3-Colourability problem, which consists of the assignment of three colours to the nodes of a graph in such a way that adjacent nodes always have different colours. This problem is known to be NP-complete.

Suppose that the nodes and the arcs are represented by a set F of facts with predicates $node$ (unary) and arc (binary), respectively. Then, the following *ASP* program allows us to determine the admissible ways of colouring the given graph.

$$\begin{array}{ll} r_1: & color(X,r) \mid color(X,y) \mid color(X,g) \ \text{:--}\ node(X). \\ r_2: & \text{:--}\ arc(X,Y), color(X,C), color(Y,C). \end{array}$$

Rule r_1 (*Guess*) above states that every node of the graph must be coloured as red or yellow or green; rule r_2 (*Check*) forbids the assignment of the same colour to any couple of adjacent nodes. The minimality of *Answer Sets* guarantees that every node is assigned only one colour. Thus, there is a one-to-one correspondence between the solutions of the 3-colouring problem for the instance at hand and the *Answer Sets* of $F \cup \{r_1, r_2\}$: the graph represented by F is 3-colourable if and only if $F \cup \{r_1, r_2\}$ has some *Answer Set*.

We have shown how it is possible to deal with a problem by means of an *ASP* program such that the instance at hand has some solution if and only if the *ASP* program as some *Answer Set*; in the following, we show an *ASP* program whose

[25]Preliminary definitions adapted from [132]

Answer Sets witness that a property does not hold, i.e., the property at hand holds if and only if the program has no *Answer Sets*.

[RAMSEY] The Ramsey Number $R(k, m)$ is the least integer n such that, no matter how we colour the arcs of the complete graph (clique) with n nodes using two colours, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique). Ramsey numbers exist for all pairs of positive integers k and m [485].

Similarly to what already described above, let F be the collection of facts for input predicates *node* (unary) and *edge* (binary), encoding a complete graph with n nodes; then, the following *ASP* program $P_{R(3,4)}$ allows to determine whether a given integer n is the Ramsey Number $R(3, 4)$, knowing that no integer smaller than n is $R(3, 4)$.

$$\begin{aligned}
r_1 :&\ \ blue(X, Y) \mid red(X, Y) \text{ :- } edge(X, Y).\\
r_2 :&\ \ \text{:- } red(X, Y), red(X, Z), red(Y, Z).\\
r_3 :&\ \ \text{:- } blue(X, Y), blue(X, Z), blue(Y, Z), blue(X, W), blue(Y, W), blue(Z, W).
\end{aligned}$$

Intuitively, the disjunctive rule r_1 guesses a colour for each edge. The first constraint r_2 eliminates the colourings containing a red complete graph (i.e., a clique) on 3 nodes; the second constraint r_3 eliminates the colourings containing a blue clique on 4 nodes. The program $P_{R(3,4)} \cup F$ has an *Answer Set* if and only if there is a colouring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an *Answer Set* for a particular n, then n is $\underline{\text{not}}$ $R(3, 4)$, that is, $n < R(3, 4)$. The smallest n, such that no *Answer Set* is found, is the Ramsey Number $R(3, 4)$.

[SUDOKU] A classic *Sudoku* puzzle consists of a tableau featuring 81 cells, or positions, arranged in a 9×9 grid, which is divided into nine sub-tableaux (regions, or blocks) containing nine positions each. Initially, a number of positions (between 17 and 35) are filled with a number picked up in the range $1 \ldots 9$. The aim of the game is to check whether every empty position can be filled with a number between 1 and 9 in such a way that each row, column and block show all digits from 1 to 9 exactly once.

Let us suppose that a set of facts F is given, representing the schema to be completed; in particular, a binary predicate pos encodes possible position coordinates; $symbol$ is a unary predicate encoding possible symbols (numbers); facts of the form $sameblock(x1, y1, x2, y2)$ state that two positions $(x1, y1)$ and $(x2, y2)$ are within the same block; facts of the form $cell(x, y, n)$ represent that a position (x, y) is filled with symbol n.

We show next an *ASP* program P_{sudoku} such that the *Answer Sets* of $P_{sudoku} \cup F$ correspond to the solutions of the *Sudoku* schema at hand; note that, in general, well-founded *Sudoku* instances have only one solution, and thus $P_{sudoku} \cup F$ will have a single *Answer Set*.

$$r_1 : \ cell(X, Y, N) \mid nocell(X, Y, N) \ :- \ pos(X), pos(Y), symbol(N).$$

$$r_2 : \ \ :- \ cell(X, Y, N), cell(X, Y, N1), N1 <> N.$$

$$r_3 : \ assigned(X, Y) :- \ cell(X, Y, N).$$
$$r_4 : \ \ :- \ pos(X), pos(Y), \textbf{not } assigned(X, Y).$$

$$r_5 : \ \ :- \ cell(X, Y1, Z), cell(X, Y2, Z), Y1 <> Y2.$$
$$r_6 : \ \ :- \ cell(X1, Y, Z), cell(X2, Y, Z), X1 <> X2.$$

$$r_7 : \ :- \ cell(X1, Y1, Z), cell(X2, Y2, Z), Y1 <> Y2, sameblock(X1, Y1, X2, Y2).$$
$$r_8 : \ :- \ cell(X1, Y1, Z), cell(X2, Y2, Z), X1 <> X2, sameblock(X1, Y1, X2, Y2).$$

Rules $r_1 - r_4$ guess the number for each cell, ensuring that each cell is filled exactly one number ($symbol$); note that the guessed values for the positions complete the extension of the predicate $cell$ for which some values have been already provided in F. Rules r_5 and r_6 check that a number does not occur more than once in the same row or column, respectively; rules r_7 and r_8, finally, ensure that two different cells in the same block do not have the same number.

1.3.5 *ASP solvers* and Language Extensions

In *Answer Set Programming*, as mentioned before, search problems are reduced to computing stable models, and *Answer Set* solvers – programs for generating stable models – are used to perform search. The search algorithms used in the design of many *Answer Set* solvers are enhancements of the *Davis-Putnam-Logemann-Loveland* (*DPLL*) procedure, and they are somewhat similar to the algorithms used in efficient SAT solvers.

The needs addressed a variety of applications fostered a thriving research within the community, causing both the enrichment and standardization of the language (as described above) and the development of other efficient solvers. Undoubtedly the success story of *ASP* has its roots in the early availability of ***ASP solvers***, beginning with the *smodels* system [528], followed by DLV [363], SAT-based ***ASP solvers***, like *assat* [379] and *cmodels* [284], and the conflict-driven learning ***ASP solver*** *clasp* [265]. Other information about *ASP* solver can be found in [13, 135, 266, 267, 311, 383, 411, 461].

In the following, we describe some of the most popular ***ASP solvers*** that we have used in the works described in this Thesis.

DataLog with Disjunction (dlv)[26]

Nowadays many systems that can evaluate disjunctive logic programs exist. One of the first was DLV a first solid, efficiency-geared implementation of a *DLP* system, became available in 1997, after 15 years of theoretical research on *DLP*. DLV system is one of the most successful and widely used *DLP* engines. After its first release, the DLV system has been significantly improved over and over in the last years, and its language has been enriched in several ways. Relevant optimization techniques have been incorporated in all modules of the DLV engine, including *DataBase* techniques for efficient instantiation and magic sets, novel techniques for answer-set checking, heuristics, back-jumping and advanced pruning operators for model generation. The DLV project has been active for more than 17 years and has led to the development and continuous enhancement of the DLV system. Recently were added parallel evaluation and evaluation in mass memory that allows to It supports a powerful language extending Disjunctive *Datalog* with many expressive constructs, including aggregates, strong and weak constraints, functions, lists, and sets.

Input data can be supplied by regular files, and also by Oracle or Objectivity *DataBase*. The DLV kernel then produces *Answer Sets* one at a time, and each time an *Answer Set* is found, “Filtering” is invoked, which performs post-processing (dependent on the active front-ends) and controls continuation or abortion of the computation. The DLV kernel consists of three major components: The “Intelligent Grounding”, “Model Generator”, and “Model Checker” modules share a principal data structure, the “Ground Program”. It is created by the Intelligent Grounding [211] using differential (and other advanced) *DataBase* techniques together with suitable data structures and used by the Model Generator and the Model Checker. The Ground Program is guaranteed to have exactly the same *Answer Sets* as the original program. For some syntactically restricted classes of programs (e.g. stratified programs), the Intelligent Grounding module already computes the corresponding *Answer Sets*. For harder problems, most of the computation is performed by the Model Generator and the Model Checker. Roughly, the former produces some “candidate” *Answer Sets* (models), the stability and minimality of which are subsequently verified by the latter. The Model Checker (MC) verifies whether the model at hand is an *Answer Set*. This task is very hard in general because checking the stability of a model is known to be $\text{co}-\text{NP}$-complete. However, MC exploits the fact that minimal model checking the hardest part can be efficiently performed for the relevant class of head-cycle-free (HCF) programs.

[26]Preliminary definitions adapted from [17, 362, 363]

A number of mechanisms have been implemented to allow DLV to interact with external systems:

- Interoperability with relational *DBMSs*: *ODBC* interface and DLV^{DB} [557]
- Interoperability with *Semantic Web* reasoners: dlvhex [199]
- Calling external (*C++*, *Python*) functions from DLV programs: dlvhex [199]
- Calling DLV from *Java* programs: *JASP* (formerly *Java Wrapper*) [214]

DLV2[27]

DLV2 is a new *ASP* system that updates DLV with modern evaluation techniques and development platforms. In particular, DLV2 combines $\mathcal{I}$-DLV [133], a fully-compliant *ASP-Core-2* grounder, with the well-assessed solver WASP [14]. These core modules are extended by application-oriented features, among them constructs such as annotations and directives that customize heuristics of the system and extend its solving capabilities.

As mentioned before, the core modules of DLV2 are:

$\mathcal{I}$-DLV the grounder **WASP** the solver

The grounder implements a bottom-up evaluation strategy based on the semi-naive algorithm and features enhanced indexing and other new techniques for increasing the system performance. Notably, the grounding module can be used as an effective deductive-*DataBase* system, as it supports a number of ad-hoc strategies for query answering.
The solver module implements a modern *CDCL* backtracking search algorithm, properly extended with custom propagation functions to handle the specific properties of *ASP* programs.

DLV2 can import relations from a *RDBMS* by means of an `#import_sql` directive. Similarly, `#export_sql` directives are used to populate specific tables with the extension of a predicate.

DLV2 supports cautious reasoning over (non)ground queries. The computation of cautious consequences is done according to *anytime* algorithms [15], so that answers are produced during the computation even in computationally complex problems. Thus, the computation can be stopped either when a sufficient number of answers have been produced or when no new answer is produced after a specified amount of time. The magic-sets technique [16] can be used to further optimize the evaluation of queries.

[27]Preliminary definitions adapted from [11]

DLV2 can be extended by means of a *Python* interface.
On the grounding side, the input program can be enriched by external atoms [125] of the form $\&\mathtt{p}(\mathtt{i}_1, \ldots, \mathtt{i}_\mathtt{n}; \mathtt{o}_1, \ldots, \mathtt{o}_\mathtt{m})$, where $\mathtt{p}$ is the name of a *Python* function, $\mathtt{i}_1, \ldots, \mathtt{i}_\mathtt{n}$ and $\mathtt{o}_1, \ldots, \mathtt{o}_\mathtt{m}$ $(n, m \geq 0)$ are input and output terms, respectively. For each instantiation $\mathtt{i}'_1, \ldots, \mathtt{i}'_\mathtt{n}$ of the input terms, function $\mathtt{p}$ is called with arguments $\mathtt{i}'_1, \ldots, \mathtt{i}'_\mathtt{n}$, and returns a set of instantiations for $\mathtt{o}_1, \ldots, \mathtt{o}_\mathtt{m}$.
On the solving side, the input program can be enriched by external propagators. Communication with the *Python* modules follows a synchronous message-passing protocol implemented by means of method calls. Basically, an external module must comply with a specific interface, whose methods are associated to events occurring during the search for an *Answer Set*, e.g. a literal is inferred as true. Whenever a specific point of the computation is reached, the corresponding event is triggered, i.e. a method of the module is called. Some methods of the interface are allowed to return values that are subsequently interpreted by the solver.
The *Python* interface also supports the definition of new heuristics [186], which are linked to input programs via *Java*-like annotations.

Within DLV2, ASP programs can be enriched by global and local annotations, where each local annotation only affects the immediate subsequent rule. The system takes advantage of annotations to customize some of its heuristics. Customizations include *body ordering* and *indexing*, two of the crucial aspects of the grounding, and *solving heuristics* to tightly link encodings with specific domain knowledge.
Concerning solving heuristics, specified via the *Python* interface, they act on a set of literals of interests, where each literal is associated with a tuple of terms.
Note that annotations do not change the semantics of input programs. For this reason, their notation starts with %, which is used for comments in *ASP-Core-2*, so that other systems can simply ignore them.

The *Potsdam Answer Set Solving Collection* (*Potassco*)[28]

The open-source project *Potassco*, gathers a variety of tools for *ASP*.

Among them we can find:

gringo a grounder that combines and extends techniques from the two major grounders namely *lparse* [556] and DLV's grounding component. A salient design principle of *gringo* is its extensibility that aims at facilitating the incorporation of additional language constructs. [270]

clasp an ***ASP solver*** that combines the high-level modelling capacities of *ASP* with state-of-the-art techniques from the area of Boolean constraint solving. Unlike existing ***ASP solvers***, *clasp* is originally designed and optimized for conflict-

[28]Preliminary definitions adapted from [256, 260]

driven *ASP* solving. Rather than applying a SAT solver to a CNF conversion, *clasp* directly incorporates suitable data structures, particularly fitting back-jumping and learning. [265]

clingo a tool that combines *gringo* and *clasp* in a monolithic system. [264]

iclingo an incremental *ASP* system built upon the libraries of *gringo* and *clasp*. Unlike the standard proceeding, *iclingo* has to operate in a "stateful way"; that is, it has to maintain its previous (grounding and solving) state for processing the current program slices. [255]

oclingo an extension of *iclingo* which adds online functionalities. [252]

There are many other tools that provide different functionalities, but they are not needed for the purposes of this thesis so there is no need to describe them here.

Standard *ASP* follows a one-shot process in computing stable models of logic programs. This view is best reflected by the input/output behaviour of monolithic *ASP* systems like DLV and *clingo*. Internally, however, both follow a fixed two-step process. First, a grounder generates a (finite) propositional representation of the input program. Then, a solver computes the stable models of the propositional program. This rigid process stays unchanged when grounding and solving with separate systems. In fact, up to version 3, *clingo* provided a mere combination of the grounder *gringo* and the solver *clasp*.

The new *clingo* 4 series offers novel high-level constructs for realizing such complex reasoning processes. This is achieved within a single integrated *ASP* grounding and solving process in order to avoid redundancies in relaunching grounder and solver programs and to benefit from the learning capacities of modern ***ASP solvers***. To this end, *clingo* 4 complements *ASP*'s declarative input language by control capacities expressed via the embedded scripting languages *Lua* and *Python*. On the declarative side, *clingo* 4 offers a new directive `#program` that allows for structuring logic programs into named and parametrizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side, viz. the scripting languages embedded via the `#script` directive. For exercising control, the latter benefit from a dedicated *clingo* library that does not only furnish grounding and solving instructions but moreover allows for continuously assembling the solver's program in combination with the directive `#external`. *clingo* 4 abolishes the need for special-purpose systems for incremental and reactive reasoning, like *iclingo* and *oclingo*, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.
Further details are provided in Section 2.3.1.

However, the *clingo* system, along with its grounding and solving components *gringo* and *clasp*, is nowadays among the most widely used tools for *Answer Set Programming* (*ASP*). As reported in [261], the new generations of the *ASP* system *clingo* (*clingo* 4 and *clingo* 5) focus heavily on this novel high-level constructs for realizing multi-shot *ASP* solving. Moreover, in *clingo* 5 they tried to integrate application- or theory-specific reasoning into *ASP* systems, by equipping them with well-defined generic interfaces facilitating the manifold integration efforts. On the grounder's side, they introduced a generic way of specifying language extensions, and they proposed an intermediate format accommodating their ground representation. On the solver's side, they introduced high-level interfaces which facilitate the integration of theory propagators dealing with these language extensions.

More details on these latest advancements are provided in [256, 259, 323, 378].

HEX *programs* and dlvhex[29]

One can see that the main benefit of the introduction of a paradigm like *ASP* consists in the possibility of describing problem domains at a high abstraction level, rather than implementing specifically tailored algorithms. The ease of modelling comes at the price of evaluation performance (nonetheless, efficient *ASP* solvers are nowadays available, see [135]). Discrete logic-based modelling paradigms are however historically weak on *a)* modelling over continuous or nearly-continuous values, and have a limited capability of *b)* dealing with probabilistic/fuzzy values.

Motivated as well by the fact that for important issues, such as *meta-reasoning* in the context of the *Semantic Web*, no adequate support is available in *ASP* and the observation that interoperability with other software is an important issue, some years ago was proposed an extension of the answer-set semantics to HEX *programs*, that are *higher-order* logic programs (which accommodate meta-reasoning through *higher-order atoms*) with *external atoms* for software interoperability. In a nutshell, HEX *programs* are an extension of *ASP* which allows the integration of external information sources, and which are particularly well-suited when some knowledge of the problem domain at hand is better modelled with means other than discrete logic.

Intuitively, a *higher-order atom* allows to quantify values over predicate names and to freely exchange predicate symbols with constant symbols, like in the rule

$$C(X) :- subClassOf(D, C), D(X).$$

[29]Preliminary definitions adapted from [191, 199–201]

An *external atom* facilitates to determine the truth value of an atom through an external source of computation. For instance, the rule

$$reached(X) :- \&reach[edge, a](X)$$

computes the predicate $reached$ taking values from the predicate $\&reach$, which computes via $\&reach[edge, a]$ all the reachable nodes in the graph $edge$ from node a, delegating this task to an external computation source (e.g., an external deduction system, an execution library, etc.).

Briefly, the Syntax of HEX *programs* can be explained specifying what are *higher-order atoms* and *external atoms*.
Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, and *external predicate names*, respectively. Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, Y_1, \ldots, Y_n$ are *terms*.
An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input* and *output* lists, respectively), and $\&g \in \mathcal{G}$ is an *external predicate name*. $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for *input* and *output* lists, respectively.

The Semantic of HEX *programs* is a generalization of the answer-set semantics [274] and uses the notion of a "reduct" as defined by [212].

By means of external atoms, different important extensions of *ASP* can be expressed in terms of HEX *programs*:

- Programs with aggregates
- Description logic programs
- Programs with monotone cardinality atoms
- Agent programs

HEX *programs* for different purposes, in which the joint availability of *higher-order* and *external atoms* is beneficial. In particular, it is well-suited as a convenient tool for a variety of tasks related to ontology languages and for *Semantic Web* applications in general, since, in contrast to other approaches, they keep decidability but do not lack the possibility of exploiting non-determinism, performing meta-reasoning, or encoding aggregates and sophisticated constructs through external atoms.

dlvhex is the prototype application for computing the models of HEX *programs*.

The system is implemented in *C++* and available as *open-source software* for all major platforms (*Linux*, *OS X*, *Windows*). Pre-compiled binaries are also provided. External sources are implemented using a plugin interface, which is currently available for *C++* and *Python*.

At the beginning, the solvers evaluated HEX *programs* by a translation to *ASP* itself, in which values of *external atoms* are guessed and verified after the ordinary *Answer Set* computation. This elegant approach does not scale with the number of external accesses in general, in particular in presence of non-determinism (which is instrumental for *ASP*).
After the developer presented a novel, native algorithm for evaluating HEX *programs* which uses learning techniques. In particular, they extended conflict-driven ASP solving techniques, which prevent the solver from running into the same conflict again, from ordinary to HEX *programs*.

They showed how to gain additional knowledge from external source evaluations and how to use it in a conflict-driven algorithm. Firstly targeting the uninformed case, i.e., when there is no extra information on external sources, and then extending this approach to the case where additional meta-information is available.

dlvhex includes many plugins (collections of related external atoms) that allow performing different tasks. Among these, there are the *Description Logic plugin*, the *dllite plugin* the *string plugin*, the *Constraint ASP plugin*, the *Action Plugin* and many others (see [583, 585]).

For more information about recent advances of dlvhex see also [202, 204, 489].

ActHEX *programs* and *Action Plugin*[30]

In general, HEX *programs* do not contemplate the possibility of changing the state of external sources. It turns out that some structural limitations of HEX *programs* prevent addressing this issue in a satisfactory way: first, external functions associated to external predicates are inherently stateless. Second, but more importantly, HEX *programs* are fully declarative: this implies that when writing a HEX *program*, it is not predictable whether and in which order an external function will be evaluated. Therefore, there were developed ActHEX *programs*.

The ActHEX formalism [70] generalizes HEX *programs* [199] introducing dedicated action atoms in rule heads. Action atoms can actually operate on and change the state of an *environment*, which can be roughly seen as an abstraction of realms outside the logic program at hand. The ActHEX framework allows to conveniently

[30]Preliminary definitions adapted from [70, 222]

design *ASP*-based applications by properly connecting logic-based decisions to actual effects thereof.

Intuitively, ActHEX *programs* extend HEX *programs* with *Action Atoms* (associated to corresponding "executable" functions). An *Action Atom* is of the form

$$\#g[Y_1, \ldots, Y_n]\{o, r\}[w : l]$$

where $\#g$ is an action predicate name, $Y_1, \ldots, Y_n$ is a list of input terms (called *input* list) of fixed length $in(\#g) = n$. Moreover, attribute $o \in \{b, c, c_p\}$ is called the *action option* that identifies an action as *brave, cautious, or preferred cautious*, while optional integer attributes r, w, and l are called *precedence*, *weight*, and *level* of $\#g$, respectively. They are optional and range over variables and positive integers. *Action Atoms* can appear only in the head of the rules.

An ActHEX *program* P is evaluated w.r.t. a fixed state (snapshot) of the *external environment* E using the following steps: (i) *Answer Sets* of P are determined w.r.t. E and the set of *best models* is a subset of the *Answer Sets* determined by an objective function; (ii) any (best) model originates a set of corresponding *execution schedules* S, i.e., a sequence of actions to execute; (iii) executing the actions of (and sequentially according to) a selected schedule S yields another (not necessarily different) state E' of the environment, called the *observed execution outcome*; finally (iv) the process may be iterated starting at (i), by considering a snapshot E'', which can be different from E' due to exogenous actions (in so-called dynamic environments). *Answer Sets* are defined similarly to HEX *programs* [199], i.e., using *Herbrand Interpretations*, the grounding of P w.r.t. the *Herbrand Universe*, and the FLP reduct; ground action atoms in rule heads are treated like ordinary atoms. We denote by $AS(P, E)$ the collection of all *Answer Sets* of P w.r.t. E. The set of *best models* of P, denoted $\mathcal{BM}(P, E)$, contains those *Answer Sets* $I \in AS(P, E)$ that minimize an objective function over weights and levels of atoms in I (equivalent to the evaluation of weak constraints in [111]). An action $a = \#g[y_1, \ldots, y_n]\{o, r\}[w : l]$ with option o and precedence r is *executable in I w.r.t. P and E* iff (i) a is brave and $a \in I$, or (ii) a is cautious and $a \in B$ for every $B \in AS(P, E)$, or (iii) a is preferred cautious and $a \in B$ for every $B \in \mathcal{BM}(P, E)$. An *execution schedule* S_I for a (best) model I is a sequence of all actions executable in I, such that for all pairs of action atoms $a, b \in I$, if $\prec (a) <\prec (b)$ then a must precede b in S_I, for $\prec (c)$ the precedence of an action atom c. Concerning the effects of actually executing actions, as well as corresponding notions of execution outcomes.

The ActHEX framework allows *a*) to express and infer a predictable order of execution for action atoms, *b*) to express soft (and hard) preferences among a set of possible action atoms, and *c*) to actually execute a set of action atoms according to a

predictable schedule. It is worth remarking that ActHEX programs do not represent an action language in a strict sense.
The main goals of the language are *1)* to provide a complementary extension to *Logic Programming* over which existing action, planning and agent languages can be grounded, and *2)* to provide a tighter and semantically sound framework for interfacing logic programs with applications of arbitrary nature.

The ActHEX framework is very versatile and can be fruitfully used in a variety of contexts like “Action languages”, “Knowledge Base Updates”, “Translation of Agent Programs”, “Web source Updates” (as shown in [70]); moreover it can also be used for logic-based games, which are an ideal test-bed (as shown in [222]).

The ***Action Plugin*** is a dlvhex *Plugin* that provides an implementation of the ActHEX language. An interface (`ActionPluginInterface`) make possible the creation of *Addons* for ***Action Plugin***.

The ***Action Plugin*** provides methods to:

- Define an External Atom;
- Registering the External Atoms;
- Define an Action Atom;
- Registering the Action Atoms;
- Define the Environment;
- Import the Action Addon;
- Define and use the Environment;
- Define and import a BestModel Selector;
- Define and import an Execution Schedule Builder;
- Control the Iteration behaviour.

For more information about ActHEX and the ***Action Plugin*** see also [584].

1.3.6 Further remarks

Several different approaches to improve evaluations of *ASP* solvers have been proposed in the last years [203, 263, 430] and many various applications have been proposed (see Section 1.3.2) showing a very active and productive research field.

For more information about *Answer Set Programming* (*ASP*) see also [12, 52, 53, 102, 103, 190, 196, 198, 258, 271, 273, 310, 373, 376, 377, 409, 443].

1.4 *Planning Domain Definition Language (PDDL)*[31]

Planning Domain Definition Language (*PDDL*) [276, 415] is a logic formalism for expressing planning tasks.
The language is inspired by the well-known *STRIPS* [221] formulations of planning problems. Its core is a simple standardization of the syntax for expressing such familiar semantics of actions, that uses pre- and post-conditions in order to describe preconditions and effects.
However, the language has been largely extended over the years, as evidenced by the planning competition series [189, 276, 388], that represented an important steer for the research in the field: the expressive power has been gradually extended to different purposes [224, 339, 593].

In this section, we briefly introduce the language.

1.4.1 Language Definition - Syntax and Semantics

Given a planning task to be performed in a particular scenario S, the main components of a *PDDL* representation are:

objects concepts of interest in S;

predicates properties of objects of interest: they can be either true or false;

initial state that specifies the starting state of S;

goal specification that describes the targets of the planning task;

actions/operators whose effects change the state of S.

A *PDDL* planning task has to be specified by two separated files: a *domain file* for predicates and actions, and a *problem file* for objects, initial state and goal specification.

A domain file must comply with the following syntax:

```
(define (domain <domain name>)
<PDDL code for predicates>
<PDDL code for first action>
[...]
<PDDL code for last action>)
```

where `<domain name>` is a string that identifies the planning domain.

[31]Preliminary definitions adapted from [132]

A problem file has the following form:

```
(define (problem <problem name>)
(:domain <domain name>)
<PDDL code for objects>
<PDDL code for initial state>
<PDDL code for goal specification>)
```

where `<problem name>` is a string that identifies the planning task, and the string `<domain name>` must match the domain name in the corresponding domain file.

This separation is an early design decision: the intent was to separate the descriptions of parametrized actions of the domain behaviours from the description of specific objects, initial conditions and goals of a problem instance. Thus, a planning problem is created by coupling a domain description with a problem description. The same domain description can be paired with many different problem descriptions, to yield different planning problems in the same domain.

An intuitive definition of each part composing a domain and a problem file is illustrated in the following, by means of some examples. For a thorough description of *PDDL*, and an extended definition of its syntax, semantics and language extensions, we refer the reader to the cited literature.

1.4.2 Planning Representation

[GRIPPER] As a first example, let us consider the following scenario, namely the *gripper* planning task[32]: a robot can move between two rooms and pick up or drop balls with either of his two arms, also called grippers. In particular, let us assume that there are four balls, and, initially, the robot is in the first room together with all balls. The goal is to place all the balls in the second room.

Intuitively, the scenario can be modelled as follows:

objects rooms, balls and robot arms;

predicates define objects properties, i.e., whether an object X is a room or a ball, whether a ball B is inside a room A, and so on;

initial state i.e., all balls and the robot are in the first room, all robot arms are empty, and so on;

goal specification all balls must be in the second room;

actions/operators the robot can move between rooms, pick up or drop a ball.

The objects of interest are:

[32] http://www.cs.toronto.edu/~sheila/2542/s14/A1/introtopddl2.pdf

- Rooms: `rooma`, `roomb`
- Balls: `ball1`, `ball2`, `ball3`, `ball4`
- Robot arms: `left`, `right`

Their descriptions can be specified in the problem file by means of the following statement:

```
(:objects rooma roomb ball1 ball2 ball3 ball4 left right)
```

As for predicates, we are interested in the following ones:

- `room(x)` – true iff x is a room
- `ball(x)` – true iff x is a ball
- `gripper(x)` – true iff x is a gripper
- `at-robby(x)` – true iff x is a room and the robot is in x
- `at-ball(x, y)` – true iff x is a ball, y is a room, and x is in y
- `free(x)` – true iff x is a gripper and x does not hold a ball
- `carry(x, y)` – true iff x is a gripper, y is a ball, and x holds y

At the beginning of the domain file, we can specify them as follows:

```
(:predicates (room ?x) (ball ?x) (gripper ?x)
(at-robby ?x) (at-ball ?x ?y) (free ?x) (carry ?x ?y))
```

The initial state defines the starting configuration:

- `room(rooma)` and `room(roomb)` are true.
- `ball(ball1)`, `ball(ball2)`, `ball(ball3)` and `ball(ball4)` are true.
- `gripper(left)`, `gripper(right)`, `free(left)` and `free(right)` are true.
- `at-ball(ball1, rooma)`, `at-ball(ball2, rooma)`, `at-ball(ball3, rooma)`, `at-ball(ball4, rooma)` and `at-robby(rooma)` are true.
- Everything else is false.

Hence, in the problem file we can add this *PDDL* representation:

```
(:init (room rooma) (room roomb) (at-robby rooma)
(ball ball1) (ball ball2) (ball ball3) (ball ball4)
(gripper left) (gripper right) (free left) (free right)
(at-ball ball1 rooma) (at-ball ball2 rooma)
(at-ball ball3 rooma) (at-ball ball4 rooma))
```

The goal specification is:

- `at-ball(ball1, roomb)`, `at-ball(ball2, roomb)`, `at-ball(ball3, roomb)`, `at-ball(ball4, roomb)` must be true.
- Everything else can be either true or false.

Thus, the following is added to the problem file:

```
(:goal (and (at-ball ball1 roomb) (at-ball ball2 roomb)
(at-ball ball3 roomb) (at-ball ball4 roomb)))
```

Finally, we need to define possible actions. For each action, we analyse its pre- and post-conditions on the scenario:

Robot move action

Description The robot can move from a room x to a room y.

Precondition `room(x)`, `room(y)` and `at-robby(x)` are true.

Effect `at-robby(y)` becomes true. `at-robby(x)` becomes false. Everything else does not change.

Pick-up action

Description The robot can pick up x in y with z.

Precondition `ball(x)`, `room(y)`, `gripper(z)`, `at-ball(x, y)`, `at-robby(y)` and `free(z)` are true.

Effect `carry(z, x)` becomes true. `at-ball(x, y)` and `free(z)` become false. Everything else does not change.

Drop operator

Description The robot can drop x in y from z.

Precondition `ball(x)`, `room(y)`, `gripper(z)`, `at-ball(x, y)`, `at-robby(y)` are true and `free(z)` is false.

Effect `carry(z, x)` and `at-ball(x, y)` become false. `free(z)` becomes true. Everything else does not change.

So, we can complete the domain file with these definitions:

```
(:action move :parameters (?x ?y)
:precondition (and (room ?x) (room ?y) (at-robby ?x))
:effect (and (at-robby ?y) (not (at-robby ?x))))

(:action pick-up :parameters (?x ?y ?z)
:precondition (and (ball ?x) (room ?y) (gripper ?z)
  (at-ball ?x ?y) (at-robby ?y) (free ?z))
:effect (and (carry ?z ?x) (not (at-ball ?x ?y)) (not (free ?z))))

(:action drop :parameters (?x ?y ?z)
:precondition (and (ball ?x) (room ?y) (gripper ?z)
  (carry ?z ?x) (at-robby ?y))
:effect (and (at-ball ?x ?y) (free ?z) (not (carry ?z ?x))))
```

[BLOCKS-WORLD] As a further example, we will consider the blocks-world planning problem [295], in which a set of blocks featuring same size and shape lies on a table in a initial configuration (possibly stacked); an agent is requested to move the blocks with the aim of arranging into a final desired configuration.

The main constraint is that only one block at a time can be moved: it may be placed either on the table or atop another block; clearly, blocks that are under another block cannot be moved.

A possible *PDDL* representation of the blocks-world scenario is:

```
 1 (define (domain BLOCKS)
 2 (:requirements :strips :typing)
 3 (:types block)
 4 (:predicates (on ?x - block ?y - block) (on-table ?x -
      block) (clear ?x - block) (hand-empty) (holding ?x -
      block))
 5
 6 (:action pick-up :parameters (?x - block)
 7 :precondition (and (clear ?x) (on-table ?x) (hand-empty))
 8 :effect (and (not (on-table ?x)) (not (clear ?x)) (not (
      hand-empty)) (holding ?x)))
 9
10 (:action put-down :parameters (?x - block)
11 :precondition (holding ?x)
12 :effect (and (not (holding ?x)) (clear ?x) (hand-empty) (
      on-table ?x)))
13
14 (:action stack :parameters (?x - block ?y - block)
15 :precondition (and (holding ?x) (clear ?y))
16 :effect (and (not (holding ?x)) (not (clear ?y)) (clear ?x
      ) (hand-empty) (on ?x ?y)))
17
18 (:action unstack :parameters (?x - block ?y - block)
19 :precondition (and (on ?x ?y) (clear ?x) (hand-empty))
20 :effect (and (holding ?x) (clear ?y) (not (clear ?x)) (not
        (hand-empty)) (not (on ?x ?y)))))
```

Listing 1.1: A representation of the blocks-world scenario in *PDDL*.

Line 1 defines the domain name, and line 2 sets some requirements on the domain definition: `:strips` specifies that the syntax used follows the most basic subset of *PDDL*, consisting of STRIPS only, while `:typing` means that the domain definition makes use of *typed variables*. This feature allows defining object and parameter types. Notably, type definitions have to be inserted before they are used. In the example the unique type used is defined at line 3, namely `block`. Once a type T is declared, it can be used to specify the type of a variable, say X: $?X - T$, as can be observed in the remaining lines.

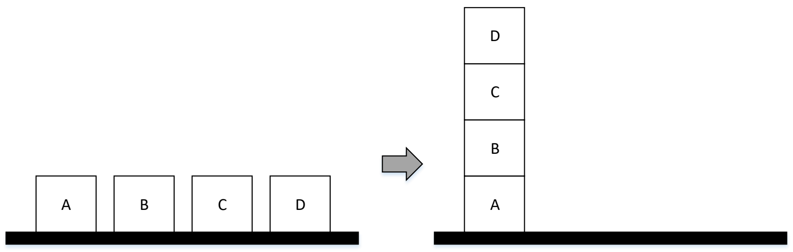


Figure 1.2.: Initial and goal configurations for the blocks-world example.

Furthermore, line 4 defines the predicates:

- `on(x,y)` where `x,y` are blocks – true iff `x` is on `y`;
- `on-table(x)` where `x` is a block – true iff `x` is on the table;
- `clear(x)` where `x` is a block – true iff `x` has not other block on it;
- `hand-empty` – true iff the agent hand is empty;
- `holding(x)` where `x` is a block – true iff the agent holds `x`.

Remaining lines define possible actions, that correspond to: *pick-up* a block, *put-down* a block, put a block on top of another (*stack* action), or remove a block which on top of another (*unstack* action).

Finally, a possible problem file, properly defining initial and goal situations and objects of interest, can be the following:

```
1 (define (problem BLOCKS-4-0)
2 (:domain BLOCKS)
3 (:objects D B A C - block)
4 (:init (hand-empty) (clear A) (clear B) (clear C) (clear D
     ) (on-table A) (on-table B) (on-table C) (on-table D))
5 (:goal (and (on D C) (on C B) (on B A))))
```

Listing 1.2: An example of a *PDDL* problem file for the blocks-world problem.

Line 3 specifies that we are considering four blocks (A, B, C, D). Figure 1.2 represents the initial state and the goal specification: line 4 states that initially the agent's hand is empty (i.e the agent is not holding any block), all blocks are on the table, and no block has another one on top of it; the goal (line 5) is that all blocks are stacked accordingly to the order: A, B, C, D.

Wrap-up

In this chapter, we briefly described the main concepts of *Logic Programming* and the reasons that make this paradigm unique. Then we mentioned some of the most well-known *Logic Programming* languages highlighting their specific characteristics.

In the next chapters we report some employments of these languages in different areas, starting from the prominent *Stream Reasoning*, and the achievement we have obtained in each of them.

2 Stream Reasoning

Logic is the beginning, not the end, of Wisdom.

— **Spock (Leonard Nimoy)**
(Star Trek VI: The Undiscovered Country)

Summary of Chapter 2

The main idea behind *Stream Reasoning* is to be able to provide continuous reasoning over "flows" of data (*data streams*).

Although this is a very recent and emerging research area, so far there is no solution that allows "complex" decision-making on top of data flows. We believe that *Logic Programming*, with its specific reasoning peculiarities, can help not only in having a formal representation of basic concepts of this field but also, combined with other techniques, can become a powerful reasoning tool for the remarkable problems of this domain.

In this chapter we first give some preliminary definitions about *Stream Reasoning* and the most interesting aspects of the topic that have been recently studied; then we present (in Sections 2.5 and 2.6) some research we conducted in this field where we integrated traditional *Stream Reasoning* techniques with logic-based ones and we studied some automation means for the query answering process. Also, we show how these ideas were used in a real project.

Chapter Outline

2.1 Definition, Motivation and Challenges

2.1.1 What *Stream Reasoning* is[1]

A common definition of *Stream Reasoning* 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. [551]

And the *data streams* mentioned in this definition are:

> unbounded sequences of time-varying data elements, they occur in a variety of modern applications. [569]

A more detailed definition, from the *Encyclopedia of Database Systems*, is:

> *Stream Reasoning* 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 toward 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. [420]

The term *Stream Reasoning* was initially proposed in [569]. In this paper, it is described as "an unexplored, yet high impact, research area" that should consist in a new multi-disciplinary approach which will provide the abstractions, foundations, methods, and tools required to integrate *data streams* and reasoning systems, thus giving an answer to many questions [178]. The idea is simple, yet pervasive. Starting from the lesson learned in the *DataBase* community (e.g., the ability to efficiently abstract and aggregate information out of multiple, high-throughput streams) a new foundational theory of *Stream Reasoning* was developed, capable to associate reasoning tasks to time windows describing data validity and to therefore to produce time-varying inferences. From these foundations, new paradigms for *Knowledge Representation* and query languages design have been derived, and

[1]Preliminary definitions adapted from [65, 177, 178, 420]

the consequent computational frameworks for *Stream Reasoning* oriented software architectures and their instrumentation have been deployed.

As mentioned before, *data streams* are unbounded sequences of time-varying data elements, that means an (almost) "continuous" flow of information. The assumption is that recent information is more relevant as they describe the current state of a dynamic system.

Streaming data is an important class of information sources. Examples of *data streams* are Web logs, feeds, click streams, sensor data, stock quotations, locations of mobile users, and so on. Streaming data is received continuously and in real-time, either implicitly ordered by arrival time, or explicitly associated with timestamps.

From these examples, it can be observed that *data streams* occur in a variety of modern applications, such as network monitoring, traffic engineering, sensor networks, *RFID* tags applications, telecom call records, financial applications, Web logs, click-streams. Specialized *Stream Database Management Systems* exist. While such systems proved to be an optimal solution for on the fly analysis of *data streams*, they cannot perform complex reasoning tasks. At the same time, while reasoners are year after year scaling up in the classical, time-invariant domain of ontological knowledge, reasoning upon rapidly changing information has been neglected or forgotten. Reasoning systems assume static knowledge and do not manage "changing worlds" – at most, one can update the ontological knowledge and then repeat the reasoning tasks.

It is worth noticing that the concept of *Stream Reasoning* is an evolution of the concept of (Real-Time) *Stream Processing*. *Stream Processing* is a term that is used widely in the literature to describe a variety of systems, from *Data Stream Management Systems* (*DSMS*s) to *Rule engines* to *Stream Processing Engines* (*SPE*s). *Stream Reasoning* emerged in the last few years as a new research area that aims at bridging the gap between *Reasoning* and *Stream Processing*. Different communities have focused on complementary aspects of processing dynamic information, referred to as *Stream Processing* when closer to the data and as *Reasoning* when closer to knowledge and event management.

Stream Processing systems mainly adopt operational and monotonic semantics at the logical core. The latter is less suited to produce results when data is missing and in particular not geared to deal with incorrect conclusions that must be retracted when more data is available. Such non-monotonic behaviour is a key aspect in expressivity that is needed for *Stream Reasoning*.

Peculiar to *Stream Processing*, and thus also to *Stream Reasoning*, are the notions of *Window* [30] and *Continuous Processing* [40]:

Window Traditional reasoning problems are based on the idea that all the information available should be taken in to account when solving the problem.
In *Stream Reasoning*, we eliminate this principle and restrict reasoning to a certain window of concern which consists of a subset of statements recently observed in the stream while previous information is ignored. This is necessary for different reasons. First of all, ignoring older statements allows us to save computing resources in terms of memory and processing time to react to important events in real time. Further, in many real-time applications, there is a silent assumption that older information becomes irrelevant at some point.

Continuous Processing Traditional reasoning approaches are based on the idea that the reasoning process has a well-defined beginning (when a request is posed to the reasoner) and end (when the result is delivered by the system).
In *Stream Reasoning*, we move from this traditional model to a continuous processing model, where requests in terms of reasoning goals are registered at the reasoner and are continuously evaluated against a *Knowledge Base* that is constantly changing.

In the latest years, due to the increasing volume of *data streams* and the crucial time requirements of many applications, different real-time approaches have been studied and specific guidelines to evaluate *Stream Processing* solutions have been developed [548], as explained in Section 2.2.1.

2.1.2 Why *Stream Reasoning* is important[2]

Will there be a traffic jam on this highway? Can we reroute travellers on the basis of the forecast? By examining the click stream from a given IP, can we discover shifts in interests of the person behind the computer? Which content on the news Web portal is attracting the most attention? Which navigation pattern would lead readers to other news related to that content? Do trends in medical records indicate any new disease spreading in a given part of the world? Where are all my friends meeting? Can we detect any intra-day correlation clusters among stock exchanges? What are the top 10 emerging topics under discussion in the blogosphere, and who is driving the discussions?

Although the information required to answer these questions is becoming increasingly available on the (Semantic) Web, there is currently no software system capable of computing the answers — indeed, no system even lets users issue such

[2]Preliminary definitions adapted from [569, 570]

queries. The reason is straightforward: answering such queries requires systems that can manage rapidly changing worlds at the semantic level.

Therefore, the study of this field is really important and very promising.

In [410] many different application scenarios are analysed, such as *(a) Semantic Sensor Web (SSW) (b) Smart Cities*, *(c)* Smart Grids, *(d)* Remote Health Monitoring, *(e)* Nanopublications, *(f)* Drug Discovery, *(g)* Abstracting and Reasoning over Ship Trajectories, *(h)* Analysis of Social Media and Mobile Applications, showing that there are really many interesting scenarios where *Stream Reasoning* can be applied.

An emblematic case is the Urban Computing [38, 72, 328, 488] (i.e., the application of pervasive computing to urban environments). The very nature of Urban Computing can be explained by means of *data streams*, representing real objects that are monitored at given locations, but reasoning about such streams can be very cost effective and problems dramatically increase when big events, involving lots of people, take place. Some years ago, due to the lack of data, solving Urban Computing problems looked like a Sci-Fi idea. Nowadays, a large amount of the required information can be made available on the Internet at almost no cost. However, current technologies are not up to the challenge of solving Urban Computing problems: this requires combining a huge amount of static knowledge about the city (i.e., urban, social and cultural knowledge) with an even larger set of *data streams* (originating in real time from heterogeneous and noisy data sources) and reasoning above the resulting time-varying knowledge. A new generation reasoner is clearly needed!

It is also worth noticing that many research groups are working on this topic and there are many research projects on this topic[3].

2.1.3 Challenges in *Stream Reasoning*[4]

The *Stream Reasoning* area is very interesting because it quite young and unexplored, and therefore many different issues still need to be solved.

The lack of a unified formal foundation for advanced reasoning with streaming data hinders the potential for expressive formalisms to be used in concrete frameworks,

[3]For instance the Distributed Heterogeneous Stream Reasoning Project of the KBS Group at the TU Wien

[4]Preliminary definitions adapted from [175, 410, 420]

Scenario	Integration		Time Manag.		Distribution		Big Data Manag.	Efficiency		Expressivity			Uncertainty Manag.	Historical Data	Quality of Service
	Different Sources	Streaming / Background Data	Data Model	Processing Model	Distrib. Sources	Distrib. Processing		High Throughput	Low Latency	Reasoning	Temporal Operators	Data Transformation			
Semantic Sensor	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Smart Cities	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Semantic Grids	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Health Monitoring	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	
Nanopublications	✓	✓			✓		✓			✓		✓	✓		
Drug Discovery	✓	✓			✓	✓	✓		✓	✓		✓	✓		
Ship Trajectories	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	
Social Media	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	

Table 2.1.: Analysis of Requirements for Application Scenarios. From [410].

and investigation on multiprocessing models to coordinate various reasoning posers in an advanced framework needs to be further investigated.

In terms of benchmarking, a direct cross comparison is only possible for few engines.

Efficient approaches to *Stream Reasoning* should also explore the interplay between statistical analysis and knowledge-driven inference methods to have both a quantitative perspective on streaming data patterns and a qualitative perspective on complex structural properties of events, context of validity, and logical correlations in decision processes.

Moreover, as shown in Table 2.1, there are various requirements for the different application scenarios presented before and most of them are not fulfilled by the available solutions at the moment. For instance, all the application fields demand some form of reasoning. However, the complexity of the reasoning task may vary significantly from application to application. It remains an open question to identify the reasoning capabilities and expressiveness required in each scenario. Similar arguments hold for data management: different applications deal with different volumes and different update rates and *Stream Reasoning* technologies are called to be applied in this large space of problems and scenarios. Because of the trade-offs discussed above, it is unknown whether it is possible (or beneficial) to develop a single solution to satisfy all of them, or if different design models, algorithms, and implementations are needed to target specific parts of this space.

Moreover, the authors of [410] presented a research agenda with a concrete description of some steps required to drive the design and implementation of future stream reasoning systems. In their analysis are present many interesting challenges from *system models* for representing *data* and *operations* on data, to aspects related to the system *implementation*, to problems that derive from the application and evaluation of the solutions and systems in the area of *Stream Reasoning*. Figure 2.1

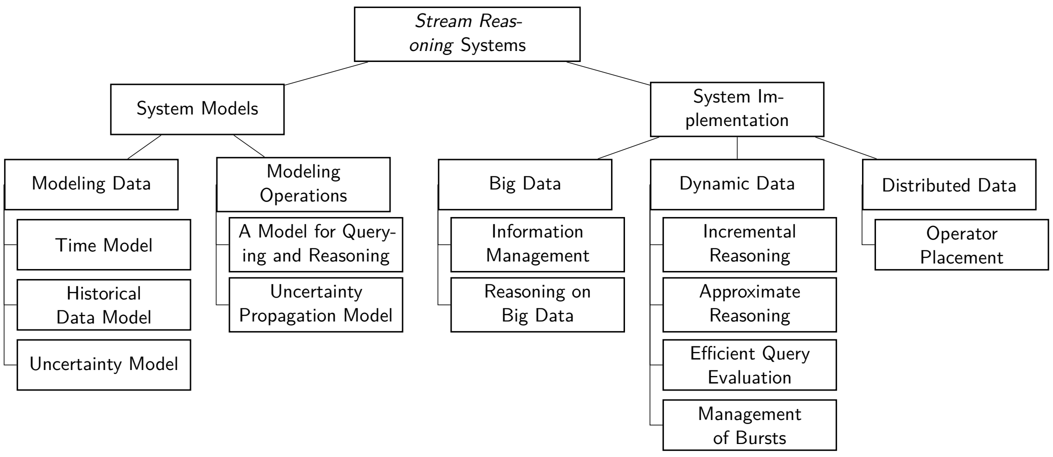


Figure 2.1.: A Research Agenda for *Stream Reasoning*. From [410].

Scenario	Integration		Time Manag.		Distribution		Big Data Manag.	Efficiency		Expressivity			Uncertainty Manag.	Historical Data	Quality of Service
	Different Sources	Streaming / Background Data	Data Model	Processing Model	Distrib. Sources	Distrib. Processing		High Throughput	Low Latency	Reasoning	Temporal Operators	Data Transformation			
Time Model		✓	✓	✓	✓			✓	✓		✓			✓	
Historical Data Model														✓	
Uncertainty Model	✓												✓		
A Model for Q&R		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	
Uncertainty Prop. Model													✓		
Information Management					✓	✓	✓	✓	✓					✓	
Reasoning on Big Data						✓	✓	✓	✓	✓					
Incremental Reasoning							✓	✓	✓	✓					
Approximate Reasoning							✓	✓	✓	✓					✓
Efficient Query Evaluation						✓	✓	✓	✓						
Management of Bursts						✓	✓	✓	✓						✓
Operator Placement					✓	✓	✓	✓	✓						

Table 2.2.: Relation between Requirements and Research Agenda. From [410].

summarizes the research agenda, while Table 2.2 shows how the topics covered by the research agenda map to the requirements described above.

However, *Stream Reasoning* research progressed and expanded its initial community to a growing number of practitioners. Very recently, authors in [175] analysed the requirements of *Stream Reasoning* w.r.t. the solutions that are currently available; their results are shown in Table 2.3. This table summarizes the current state and serves as an indication towards possible directions for future *Stream Reasoning* research.

Stream Reasoners should offer richer query languages, which include a wider set of operators to encode user needs, and the engine to evaluate them. Reasoning took a more generic connotation, and now it includes inductive reasoning techniques in addition to deductive ones. This trend will grow, combining different techniques to overcome their respective limits. Solutions need to be engineered in scalable frameworks, i.e., they must be able to integrate and reason over huge amounts of heterogeneous data while guaranteeing time requirements. And it will be import-

Requirement	Current *Stream Reasoning*
R1: Volume	★
R2: Velocity	★★★
R3: Variety	★★
R4: Incompleteness	★
R5: Noise	★
R6: Timely fashion	★★
R7: Fine-grained access	★★★
R8: Complex domains	★★
R9: What users need	★★

Table 2.3.: A review of the *Stream Reasoning* requirements w.r.t. the current state of the art (★ = not specifically treated so far, ★★ = treated but not resolved, ★★★ = universally addressed by all studies). From [175].

ant to fill the gaps between theoretical models and reality, making *Stream Reasoning* solutions robust and able to cope with issues such as noise and heterogeneity. In parallel, it will be important to identify real problems and scenarios where *Stream Reasoning* may be a solution. *Internet of Things* and *Industry 4.0* are examples of areas where to apply *Stream Reasoning* results. Moreover, it is necessary to develop benchmarking and evaluation activities, to compare and contrast the current solutions.

2.2 Data Stream Reasoning: research timeline

In this section we introduce some preliminary notions and we highlight the most prominent steps that led to the growth of this field.

2.2.1 *Data Stream Management Systems* (*DSMSs*)[5]

Data Stream Management Systems (*DSMS*s) are the first approach in Computer Science to the task of processing huge amount of real-time data of comes from the *DataBase* world. *Data Stream Management Systems* represent a vibrant area of new technology for which researchers have extended *DataBase* query languages to support continuous queries on *data streams*.

Traditional *DataBase Management Systems* are best equipped to run *one-time* queries over finite stored data sets. However, many modern applications such as network monitoring, financial analysis, manufacturing, and sensor networks require long-running, or *continuous*, queries over continuous unbounded streams of data.

Processing of *data streams* has been largely investigated in the last decades [244] and a new specialized class of *DB* systems called *Data Stream Management Systems* (*DSMS*s) have been developed. A *Data Stream Management System* is similar to a *DataBase Management System* (*DBMS*) but it is able to manage "continuous" *data streams* while the *DBMS* is, usually, able to manage only "static" *data streams*. A *DSMS* can execute *continuous queries* over data that are updated continuously, using query languages like *SQL* in *DBMS*. A *DSMS* can perform only simple reasoning tasks due to the inherent limitations of its language; one of the reasons is that researchers have extended *DataBase* query languages to support continuous queries on *data streams* but they have made little use of logic-based concepts (different from the use made in relational *DataBase*s). Moreover, in this context, not only events with a simple nature have been studied but also "complex" ones, i.e. events that rely on simpler ones and that are usually specified with the help of operators of an event algebra [601].

Data Stream Management Systems (*DSMS*s) represent a paradigm change in the *DataBase* world because they move from persistent relations to transient streams, with the innovative assumption that streams can be "*consumed*" on the fly (rather than stored forever) and from user-invoked queries to *continuous queries*, i.e., queries which are persistently monitoring streams and are able to produce their answers even in the absence of invocation. *DSMS*s can support parallel query answering

[5]Preliminary definitions adapted from [29, 569, 595]

over data originating in real time and can cope with burst of data by adapting their behaviour and gracefully degrading answer accuracy by introducing higher approximations.

Until now, *DSMS* researchers have made little use of logic-based concepts, although these provide a natural formalism and simple solutions for many of the difficult problems besetting this area. The *DataBase* query language was extended to support continuous query on *data streams* but does not have solid theoretical foundations.

In particular, in the paper [595], the author shows that *Reiter's Closed World Assumption* [491] provides a natural basis on which to study and formalize the blocking behaviour of continuous query operators, whereby concepts such as local stratification can be used to achieve a natural and efficient expression of recursive rules with non-monotonic constructs.

Data Streams can be modelled as append-only relations on which the *DSMS* is asked to support standing queries (i.e., continuous queries). As soon as tuples arrive in the input stream, the *DSMS* is expected to decide, in real time or quasi-real-time, which additional results belong to the query answer and promptly append them to the output stream. This is an incremental computation model, where no output can be taken back; therefore, the *DSMS* might have to delay returning an output tuple until it is sure that the tuple belongs to the final output — a certainty that for many queries is only reached after the *DSMS* has seen the whole input. The queries showing this behaviour, and operators causing it, are called *blocking*, and have been characterized in [39] as follows: *A blocking query operator is one that is unable to produce the first tuple of the output until it has seen the entire input*.

Clearly, blocking query operators are incompatible with the computation model of *DSMS* and should be disallowed, whereas all non-blocking queries should instead be allowed. However, many queries and operators, including essential ones such as union, fall in-between and are only partially blocking; currently, we lack simple rules to decide when, and to which extent, partially blocking operators should be allowed and how they should be treated.

The main previous results on blocking queries proved that non-monotonic query operators are blocking, whereas monotonic operators are non-blocking [296, 356]. Given that negation and traditional aggregates are non-monotonic, most current *DSMS* simply disallow them in queries, although this exclusion causes major losses in expressive power [356].

A key assumption is that operators are order-preserving. Thus, each operator takes tuples from the front of its input queue and add the tuple(s) it produces, if any, to the tail of its output buffer. Thus, buffers might delay but not alter the functions computed by simply feeding the output of one operator directly into the input of the next.

The semantics of query Q on a stream is defined by the cumulative answer that Q has returned until time τ.

For example, last occurrence of code red: $?last(T, red)$.

$$last(T, Z) \leftarrow msg(T, Z), \neg next(T, Z).$$
$$next(T, Z) \leftarrow msg(T1, Z), T1 > T.$$

This is obviously a blocking query, inasmuch as we do not have the information needed to decide whether the current red-alert message is actually the final one, while messages are still arriving. Only when the data stream ends, we can make such an inference: to answer this query correctly, we will have to wait till the input stream has completed its arrival, and then we can use the standard *CWA* to entail the negation that allows us to answer our query. But the standard *CWA* assumption will not help us to conclude that our query is non-blocking. In [595] the author exploits the timestamp ordering of the *data streams* to define a *Progressive Closing World Assumption* (*PCWA*) that can be used in the task. In this definition are also included traditional *DataBase* facts and rules, since these might also be used in continuous queries.

Definition (*Progressive Closing World Assumption* (*PCWA*))**.**
Consider a world consisting of one timestamped-ordered stream and DataBase facts. Once a fact $stream(T, ...)$ is observed in the input stream, the PCWA allows us to assume $\neg stream(T1, ...)$, provided that $T1 < T$, and $stream(T1, ...)$ is not entailed by our fact base augmented with the $stream$ facts having timestamp $\leq T$.
Therefore, PCWA for a single data stream revises the standard CWA of deductive DataBases with the provision that the world is, in fact, expanding according to its timestamps.

In the same paper, the author proposes also a new language, logic-based that allows reasoning on *data streams* called *Streamlog*.

Streamlog[6]

Streamlog ([595]) is basically *Datalog* with modified well-formedness rules for negation to guarantee simple declarative semantics and efficient execution.

[6]Preliminary definitions adapted from [595]

In *Streamlog*, base predicates, derived predicates, and the query goal are all time-stamped in their first arguments. These will be called *temporal*, to distinguish them from non-timestamped *DataBase* facts and predicates that might also be used in the programs.
The same safety criteria used in *Datalog* can be used in *Streamlog*. Furthermore, in *Streamlog* the time-stamp variables are made safe by equality chains equating their values to the timestamps in the base stream predicates. Therefore, even if $T1$ is safe, expressions such as $T2 = f(T1)$ or $T2 = T1 + 1$ cannot be used to deduce the safety of $T2$. Only equality can be used for time-stamp arguments.
These are obvious syntactic rules that will avoid blocking behaviour in the temporal rules of safe *Streamlog* programs.

Strictly Sequential

A rule is said to be *Strictly sequential* when the time-stamp of its head is $>$ than every time-stamp in the body of the rule. A predicate is *strictly sequential* when all the rules defining it are *strictly sequential*.

Sequential

A rule is said to be *sequential* when it satisfies the following three conditions:

1. the timestamp of its head is equal to the timestamp of some positive goal,
2. the timestamp of its head is $>$ or $\geq$ than the timestamps of the remaining goals,
3. its negated goals are strictly sequential or have a time-stamp that is $<$ than the timestamp of the head.

A program is said to be *sequential* when all its rules are *sequential* or *strictly sequential*.

Stratified *Datalog* programs have a syntactic structure that is easy for a compiler to recognize and turn into an efficient implementation [596]. In fact, the unique stable model of these programs, called the perfect model, can be computed efficiently using a stratified iterated fixpoint [596]. Unfortunately, stratified programs do not allow negation or aggregates in recursive rules, and therefore, are not conducive to efficient expression of algorithms such as shortest path. A lot of previous research was devoted to overcoming this limitation. In particular, there is a class of programs called locally stratified programs that have a unique stable model, called perfect model. Unfortunately, the stratification for a locally stratified program can only be verified against its instantiated version. But the simple notion of *sequential* programs for *Streamlog* avoids the non-monotonicity problems that have hamstrung *Datalog* and frustrated generations of researchers.

Sequential programs are locally stratified by their time-stamp values. To prove this the author constructs the *bistate* equivalent of the program.

A much-studied *DSMS* problem is how to best ensure that binary query operators, such as unions or joins, generate outputs sorted by increasing time-stamp values [43, 44, 316]. This problem has been extensively studied, but only at the implementation level [43, 44, 316]. At the logical level, the problem can be solved but users want to write the simple rules and let the system take care of time-skews. Therefore, in *Streamlog*, the users are allowed to work under the *Perfect Synchronization Assumption* (*PSA*), whereby the *data streams* of interest are perfectly synchronized.

Construct the bistate equivalent of the rules and obtain a stratified program, whereby the original program is locally stratified and the efficient execution techniques previously discussed remain valid. Therefore, we can relax the definition of *Strictly Sequential* rules as follows:

Strictly Sequential

> A rule is said to be *Strictly sequential* when the time-stamp of the head of the rule is $>$ than the time-stamp of each recursive goal and $\leq$ the timestamps of the non-recursive goals.

In the same work, the author shows that following properties hold:
If P is a Sequential *Program then: (i) P is locally stratified, and (ii) the unique stable model of P can be computed by repeating the iterated fixpoint of its bistate version for each time-stamp value.*

The results presented in the work [595] are still preliminary, however, they show that logic can bring sound theoretical foundations and superior expressive power to *DSMS* languages which, currently, are dreadfully lacking in both. Moreover, it is clear that *Streamlog* obtains the greater level of expressive power that negation (and aggregates) in recursive rules entail by guaranteeing that simple sequentiality conditions hold between the timestamped predicates in the rules.

Continuous Query Languages (*CQLs*)[7]

Some years before the work [595], researchers at Stanford worked on the development of a general-purpose *Data Stream Management System* (*DSMS*) for processing continuous queries over multiple continuous *data streams* and stored relations. The project, called *STanford stREamdatA Manager* (*STREAM*), was designed to investigate data management and query processing and to build a general-purpose prototype able handle high-volume and bursty *data streams* with large numbers of complex continuous queries.

[7]Preliminary definitions adapted from [29–32]

Moreover, they introduced the *Continuous Query Language* (*CQL*), an expressive *SQL*-based declarative language for registering continuous queries against streams and stored relations. This was the language supported by the *STREAM* prototype.

For simple continuous queries over streams, it can be sufficient to use a relational query language such as *SQL*, replacing references to relations with references to streams, and streaming new tuples in the result. However, as continuous queries grow more complex, e.g., with the addition of aggregation, subqueries, windowing constructs, and joins of streams and relations, the semantics of a conventional relational language applied to these queries quickly becomes unclear. To address this problem, the authors have defined a formal abstract semantics for continuous queries, and they have designed the concrete declarative query language *Continuous Query Language* (*CQL*) that implements the abstract semantics.

Abstract Semantics The abstract semantics is based on two data types, *streams* and *relations*, which are defined using a discrete, ordered time domain $\mathcal{T}$.

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 $\tau \in \mathcal{T}$ is the timestamp of the element (the logical arrival time of tuple s on stream S). There are two classes of streams: *base streams*, which are the source *data streams* that arrive at the *DSMS*, and *derived streams*, which are intermediate streams produced by operators in a query.

A relation R is a mapping from each time instant in $\mathcal{T}$ to a finite but unbounded bag of tuples belonging to the schema of R. A relation R defines an unordered of tuples at any time instant $\tau \in \mathcal{T}$, denoted $R(\tau)$. Note that in the standard relational model a relation is simply a set (or bag) of tuples, with no notion of time as far as the semantics of relational query languages are concerned. The bag of tuples in a relation at a given point in time $R(\tau)$ denotes an *instantaneous relation*. As for the streams, there are two classes of relations: *base relation* for input relations and *derived relation* for relations produced by query operators.

The abstract semantics, as shown in Figure 2.2, uses three classes of operators over streams and relations:

- A *stream-to-relation* operator takes a stream S as input and produces a relation R as output with the same schema as S.
- A *relation-to-relation* operator takes one or more relations $R_1, \ldots, R_n$ as input and produces a relation R as output.
- A *relation-to-stream* operator takes a relation R as input and produces a stream S as output with the same schema as R

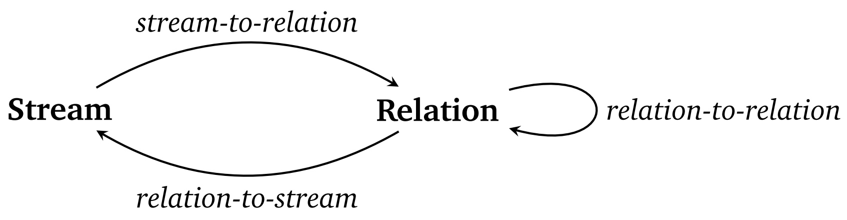


Figure 2.2.: Data types and operator classes in abstract semantics.

Stream-to-stream operators are absent, they are composed of operators of the above three classes. These three classes are "black box" components of the abstract semantics: the semantics does not depend on the exact operators in these classes, but only on generic properties of each class.

A continuous query Q is a tree of operators belonging to the above classes. The inputs of Q are the streams and relations that are input to the leaf operators, and the output of Q is the output of the root operator. The output is either a stream or a relation, depending on the class of the root operator.

At time τ, any operator of Q logically depends on its inputs up to τ. The behaviour of query Q is derived from the behaviour of its operators in the usual inductive fashion.

Concrete Language Syntactically, *CQL* is a relatively minor extension to *SQL*.

Stream-to-Relation Operators in *CQL* The stream-to-relation operators in *CQL* are based on the concept of a sliding window [39] over a stream, and are expressed using a window specification language derived from *SQL*-99:

- A *time-based* sliding window on a stream S takes a time interval ω as a parameter and produces a relation R. At time τ, $R(\tau)$ contains all tuples of S with timestamps between $\tau - \omega$ and τ. It is specified by following S with "[Range ω]". As a special case, "[Now]" denotes the window with $\omega = 0$.
- A *tuple-based* sliding window on a stream S takes an integer $N > 0$ as a parameter and produces a relation R. At time τ, $R(\tau)$ contains the N tuples of S with the largest timestamps $\leq \tau$. It is specified by following S with "[Rows N]". As a special case, "[Rows Unbounded]" denotes the append-only window "[Rows ∞]".
- A *partitioned* sliding window on a stream S takes an integer N and a set of attributes $\{A_1, \ldots, A_k\}$ of S as parameters, and is specified by following S with "[Partition By $\mathtt{A_1, \ldots, A_k}$ Rows N]". It logically partitions S into different substreams based on equality of attributes $A_1, \ldots, A_k$, computes a tuple-based

sliding window of size N independently on each sub-stream, then takes the union of these windows to produce the output relation.

Relation-to-Relation Operators in *CQL* *CQL* uses *SQL* constructs to express its relation-to-relation operators, and much of the data manipulation in a typical *CQL* query is performed using these constructs, exploiting the rich expressive power of *SQL*.

Relation-to-Stream Operators in *CQL* *CQL* has three relation-to-stream operators: *Istream*, *Dstream*, and *Rstream*.

Istream (for "insert stream") applied to a relation R contains $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$, i.e., whenever s is inserted into R at time τ.

Dstream (for "delete stream") applied to a relation R contains $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau - 1) - R(\tau)$, i.e., whenever s is deleted from R at time τ.

Rstream (for "relation stream") applied to a relation R contains the $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau)$, i.e., every current tuple in R is streamed at every time instant.

When a continuous query specified in *CQL* is registered with the *STREAM* system, a *query plan* is compiled from it. Query plans are composed of *operators*, which perform the actual processing, *queues*, which buffer tuples (or references to tuples) as they move between operators, and *synopses*, which store operator state.

The authors identified many useful concrete operators, defined strategies to generate, optimize and execute a query plan, and determined approaches to optimize the management of the synopses.

Many other research groups worked on *Data Stream Management System* since the beginning of this century. Other most notably are the academic prototypes are *TelegraphCQ* [144, 145, 347], *Aurora/Borealis* [1, 2, 142] and *PIPES* [346]. Moreover, some commercial systems have been proposed like *StreamBase* [549], *Truviso* [225] and extensions of almost all commercial *DBMS* (*MySQL*, *PostgreSQL*, *DB2*, etc.).

They have many similarities but there is no streaming *SQL* standard. Even if in the following we will see that the *CQL* approach has been adopted by most of the *RDF* and the logic-based systems.

Requirements of Real-Time *Stream Processing*[8]

From the previous sections, it is evident that applications that require real-time processing of high-volume data streams are pushing the limits of traditional data processing infrastructures.
Furthermore, several technologies have emerged-including off-the-shelf *Stream Processing* engines-specifically to address the challenges of processing high-volume, real-time data without requiring the use of custom code. At the same time, some existing software technologies, such as main memory *DSMS*s and *Rule Engines*, are also being "repurposed" by marketing departments to address these applications.

For this reason, some researchers ([548]) in the same years tried to identify some requirements that a system software should meet to excel in a variety of real-time *Stream Processing* applications. Their goal was to provide high-level guidance to information technologists so that they will know what to look for when evaluation alternative *Stream Processing* solutions.
They identified 8 rules:

Rule 1: ***Keep the Data Moving***

The first requirement for a real-time Stream Processing system is to process messages "in-stream", without any requirement to store them to perform any operation or sequence of operations. Ideally, the system should also use an active (i.e., non-polling) processing model.

Rule 2: ***Query using SQL on Streams (StreamSQL)***

The second requirement is to support a high-level "StreamSQL" language with built-in extensible stream-oriented primitives and operators.

Rule 3: ***Handle Stream Imperfections (Delayed, Missing, Out-of-Order Data)***

The third requirement is to have built-in mechanisms to provide resiliency against stream "imperfections", including missing and out-of-order data, which are commonly present in real-world data streams.

Rule 4: ***Generate Predictable Outcomes***

The fourth requirement is that a Stream Processing engine must guarantee predictable and repeatable outcomes.

Rule 5: ***Integrate Stored and Streaming Data***

The fifth requirement is that a Stream Processing system should have the capability to efficiently store, access, and modify state information, and combine it with live streaming data. For seamless integration, the system should use a uniform language when dealing with either type of data.

[8]Preliminary definitions adapted from [548]

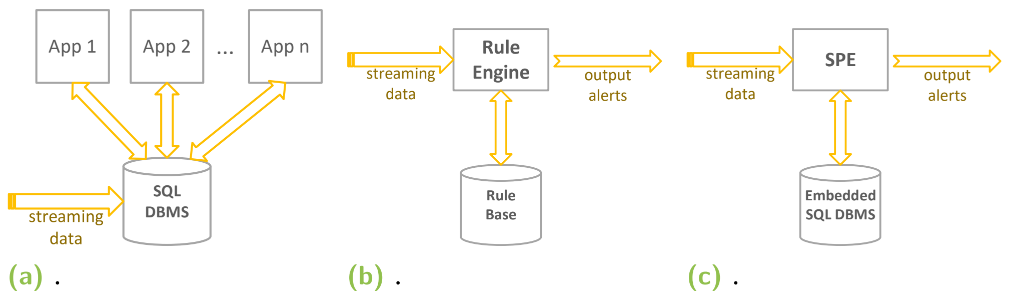


Figure 2.3.: Basic architectures of (a) a *DBMS*, (b) a *Rule Engine*, and (c) a *Stream Processing Engine*. Adapted from [548].

Rule 6: ***Guarantee Data Safety and Availability***

The sixth requirement is to ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failures.

Rule 7: ***Partition and Scale Applications Automatically***

The seventh requirement is that a Stream Processing system must be able to distribute its processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent.

Rule 8: ***Process and Respond Instantaneously***

The eighth requirement is that a Stream Processing system must have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

In the same work, the authors identified also three different software system technologies that, at that time (2005) could potentially be applied to solve high-volume low-latency streaming problems: *DBMSs*, *Rule Engines*, and *Stream Processing Engines*. Their basic architectures are shown in Figure 2.3.

Moreover, they evaluated these systems on the basis of the requirements presented earlier, and they summarized the results of their evaluation in a table (reported in Table 2.4). Each entry in the table contains one of four values:

Yes The architecture naturally supports the feature.

No The architecture does not support the feature.

Possible The architecture can support the feature. One should check with a vendor for compliance.

Difficult The architecture can support the feature, but it is difficult due to the non-trivial modifications needed. One should check with the vendor for compliance.

	DBMS	**Rule Engine**	*SPE*
Keep the data moving	No	Yes	Yes
***SQL* on streams**	No	No	Yes
Handle stream imperfections	Difficult	Possible	Possible
Predictable outcome	Difficult	Possible	Possible
High availability	Possible	Possible	Possible
Stored and streamed data	No	No	Yes
Distribution and scalability	Possible	Possible	Possible
Instantaneous response	Possible	Possible	Possible

Table 2.4.: The capabilities of various systems software.

2.2.2 *Complex Event Processing* (*CEP*)[9]

An increasing number of distributed applications requires processing continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to *complex* queries. The concepts of timeliness and flow processing are crucial for justifying the need for a new class of systems.

These requirements have led to the development of a number of systems specifically designed to process information as a flow (or a set of flows) according to a set of pre-deployed processing rules. Despite having a common goal, these systems differ in a wide range of aspects, including architecture, data models, rule languages, and processing mechanisms. In part, this is due to the fact that they were the result of the research efforts of different communities, each one bringing its own view of the problem and its background to the definition of a solution, not to mention its own vocabulary [71]. After several years of research and development, we can say that two models emerged and are competing today: the *data stream processing* model [39] and the *complex event processing* model [393].

Conversely, to *DSMS*s, the *Complex Event Processing* model views flowing information items as notifications of events happening in the external world, which have to be filtered and combined to understand what is happening in terms of higher-level events. Accordingly, the focus of this model is on detecting occurrences of particular patterns of (low-level) events that represent the higher-level events whose occurrence has to be notified to the interested parties. The contributions to this model come from different communities, including distributed information systems, business process automation, control systems, network monitoring, sensor networks, and middleware, in general. The origins of this approach may be traced back to the publish-subscribe domain [210]. Indeed, while traditional publish-subscribe systems consider each event separately from the others and filter them (based on their

[9]Preliminary definitions adapted from [160]

topic or content) to decide if they are relevant for subscribers, *Complex Event Processing* (*CEP*) systems extend this functionality by increasing the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple, related events.

For more information about *Complex Event Processing* see also [160, 209, 392].

2.2.3 Research in the *Semantic Web* community[10]

The use of the Internet as a major source of information has created new challenges for computer science and has led to significant innovation in areas such as *DataBases*, information retrieval and semantic technologies. Currently, we are facing another major change in the way information is provided. Traditionally information used to be mostly static with changes being the exception rather than the rule. Nowadays, more and more dynamic information, which used to be hidden inside dedicated systems, is getting available to decision makers.

The Web is highly dynamic: new information is constantly added, and existing information is continuously changed or removed. Large volumes of data are produced and made available on the Web by online newspapers, blogs, social networks, etc., not to mention data coming from sensors for environmental monitoring, weather forecast, traffic management, and domain-specific information, like stock prices. In these scenarios, information changes at a very high rate, so that we can identify a *stream of data* on which we are called to operate with high efficiency.

As mentioned before, this leads to the development of *DSMS*s and *CEP* that effectively deal with the transient nature of *data streams*, providing low delay processing even in the presence of large volumes of input data generated at a high rate. All these systems are based on data models, like for example the well-known relational model, which allow only a predefined set of operations on streams with a fixed structure. This allows the implementation of ad-hoc optimizations to improve the processing. However, the Web provides streams of data that are extremely heterogeneous, both at a structural and at a semantical level. For example, a Twitter stream is radically different from a stream delivered from a news channel, not only because they are stored using different formats, but also because they contain different types of information. Furthermore, the ability of operating on-the-fly on several of these streams simultaneously would allow the implementation of real-time services that can select, integrate, aggregate, and process data as it becomes available, for example, to provide updated answers to complex queries or to detect situations

[10]Preliminary definitions adapted from [60, 61, 174, 177, 410, 551, 569, 570]

of interests, to automatically update the information provided by a website or application.

Moreover, processing of *data streams* has been largely investigated and rapidly changing data can be analysed on the fly by specialized *Data Stream Management Systems*. Reasoners are year after year scaling up in the classical, time-invariant domain of ontological knowledge and reasoning upon rapidly changing information has been neglected or forgotten. However, *Data Stream Management Systems* cannot perform complex reasoning tasks, and they lack a protocol to publish widely and to provide access to the rapidly changing data. Reasoners, on the other hand, can perform such complex reasoning tasks, and the *Semantic Web* is providing the tools and methods to publish data widely on the Web. These technologies, however, do not really manage changing worlds: accessing and reasoning with rapidly changing information have been neglected or forgotten by their development communities.

The state of the art in reasoning over changing worlds is based on temporal logic and belief revision; these are heavyweight tools, suitable for data that changes in low volumes at low frequency. The challenge is to make the transition from hand-crafted systems to automatic reasoning over *data streams* of similar magnitudes.

The combination of reasoning techniques with *data streams* gives rise to *Stream Reasoning* [570], which is a new multi-disciplinary approach that can provide the abstractions, foundations, methods, and tools required to integrate *data streams*, the *Semantic Web*, and reasoning systems.
Central to the notion of *Stream Reasoning* is a paradigmatic change from persistent *Knowledge Bases* and user-invoked reasoning tasks to transient streams and continuous reasoning tasks.

Obviously *Stream Reasoning* making sense only if it is in real time, of multiple, heterogeneous, gigantic and inevitably noisy *data streams*, in order to support the decision process of extremely large numbers of concurrent user.

A first step toward *Stream Reasoning* has been to combine the power of existing *Data Stream Management Systems* and the *Semantic Web* [569].
However, different models, languages, and systems have been proposed in the last years to handle streams on the Web, combining *Semantic Web* technologies with *Complex Event Processing* (*CEP*) and *Data Stream Management System* (*DSMS*) features. These languages and systems, commonly labelled under the *RDF Stream Processing* (*RSP*) name, are solutions that extend *SPARQL* with *Stream Processing* features, based on either the *CEP* or *DSMS* paradigm.

Combining *CEP* and *DSMS* features in a unique model is a step towards filling the gap between *RDF Stream Processing* (*RSP*) and *Stream Processing* engines available on the non-semantically-aware systems on the market (e.g., Oracle Event Processor, ESPER, IBM InfoSphere Streams) [160]. There are indeed several motivations behind combining *DSMS* and *CEP*. It is clearly possible to mix different *DSMS* and *CEP* languages to achieve the desired tasks, but there are drawbacks, e.g., the need to learn multiple languages, the limited possibility for query optimizations, the potential higher amount of resources.

Actually, as mentioned in [177], the nature of streams requires a paradigmatic change (first arose in *DB* community):

from persistent data
: to be stored and queried on demand (a.k.a. one time semantics)

to transient data
: to be consumed on the fly by continuous queries (a.k.a. continuous semantics)

In [570] the authors systematically analysed the problems and have divided *Stream Reasoning* research into five areas:

- Theory for *Stream Reasoning*
- Logic Language for *Stream Reasoning*
- Stream Data Management for the *Semantic Web*
- *Stream Reasoning* for the *Semantic Web*
- Engineering and Implementations

On a general level, currently there are solutions for reasoning about static knowledge and solutions for handling streaming data. Therefore, a basic requirement for a stream-reasoning system is to integrate these two aspects in a common approach that can perform reasoning on semantic streams. *Stream Reasoner* takes several streams of rapidly changing information and several static sources of background knowledge as input.

We have a lot of data of many different types, but in order to have efficient *Stream Reasoners*, some issues need to be solved.
These are the issues identified in [570]:

- Lack of Theory for *Stream Reasoning*
- Heterogeneous Formats and Access Protocols
- Semantic Modelling
  Semantic Modelling of *data streams* involves several difficulties:
  - Window dependencies

    - Time dependencies
    - Relationships between summarization and inference
    - Merging with static information sources
    - Learning from stream
- Scale
- Continuous Processing
- Real-Time Constraints
- Parallelization and Distribution

The same authors suggested also the following quality criteria that can be used to test *Stream Reasoners*:

- number of *data streams* handled simultaneously;
- update speed of the *data streams* (for example, in assertions per second);
- number of subscribed queries handled in parallel;
- number of query subscribers that must be notified;
- time between event occurrence and notification of all subscribers.

Another important aspect is that knowledge and data can change over time.
In [569] the authors consider knowledge as invariable during the observation period, only data can change, and they classify the data according to the frequency they are expected to change, in these categories:

1. **Invariable** data that do not change in the observation period, e.g. the names and lengths of the roads.
2. **Periodically changing** data, for which a temporal law describing their evolution is present in the Invariable knowledge; they are further classified as:
    a) *Probabilistic* data, e.g. the fact that a traffic jam is present in the west side of Milan due to bad weather or due to a soccer match is taking place in San Siro stadium;
    b) *Pure periodic* data, e.g.the fact that every night at 10 PM Milan west-side overpass road closes.
3. **Event-driven** changing data that got updated as a consequence of some external event not described in the knowledge, which are further characterized by the mean time between changes:
    a) Fast, as an example consider the intensity of traffic (as monitored by sensors) for each street in a city;
    b) Medium, as an example consider roads closed for accidents or congestion due to traffic;

c) Slow, as an example consider roads closed for scheduled works.

As reported also in [569], the authors developed a pluggable algorithm, in the context of the LarKC European Research Project[11] [216], which ideally includes five steps (shown in Figure 2.4) to be iterated until a good enough answer is found:

1. ***retrieve*** relevant resource/content/context,
2. ***select*** relevant problems/methods/data,
3. ***abstract*** by extracting information, calculating statistics and transforming to logic,
4. ***reason*** upon the aggregated knowledge,
5. ***decide*** if a new iteration is needed.

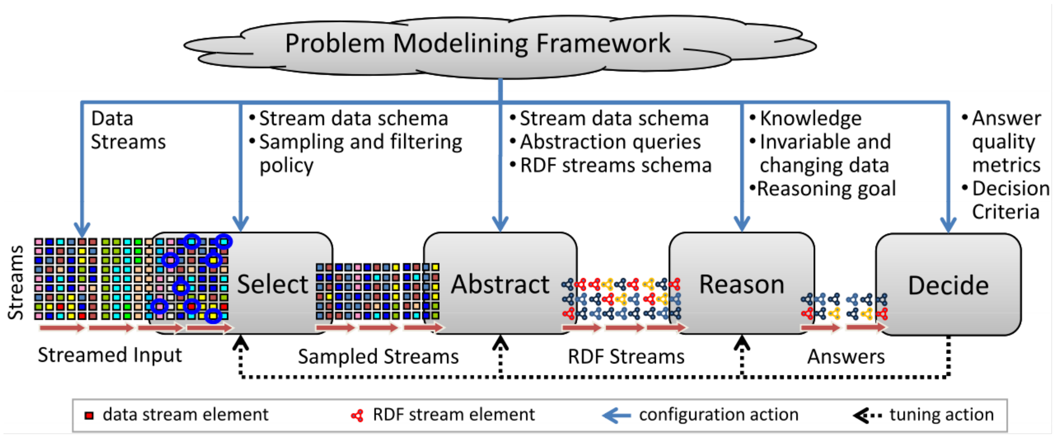


Figure 2.4.: Conceptual System Architecture. From [569].

This is a very interesting approach and this architecture should be used by every *Stream Reasoner*.

Furthermore, in the *Semantic Web* world, were also developed new notions to represent streams (*RDF streams*) and process them (solvers like *C-SPARQL*, *CQELS*, etc.) to support the pragmatic change from persistent data to transient data [60, 61].

In [60] is introduced the notion of *RDF streams* as the natural extension of the *RDF* data model and then the *SPARQL* language is to query *RDF streams*.

RDF streams are new data formats set at the confluence of conventional *data streams* and of conventional atoms usually injected into reasoners. An *RDF stream*, similar to *RDF* graphs, is identified by using an *IRI* (a locator of the actual streaming data source), but instead of being a static collection of triples a stream is a sequence of *RDF* triples that are continuously produced and annotated with a time-stamp.

[11] `http://www.larkc.org`

Timestamps can be considered as annotations of *RDF* triples, and are monotonically non-decreasing.

Moreover, in [569], the authors presented two alternative formats for *RDF streams*:

- A *RDF molecules stream* is an unbounded bag of pairs $\langle \rho, \tau \rangle$, where ρ is a *RDF* molecule [182] and τ is the timestamp that denotes the logical arrival time of *RDF molecule* ρ on the stream;
- A *RDF statements stream* is a special case of *RDF molecules stream* in which ρ is an *RDF* statement instead of a *RDF molecule*.

These researchers presented, in addition, two frameworks related to them:

- the *RDF Molecules Stream Reasoning Framework*
- the *RDF Statements Stream Reasoning Framework*

We refer the reader to their articles for more information.

In [61] is presented a technique for *Stream Reasoning* that incrementally maintains a materialization of ontological entailments in the presence of streaming information (that is an extension of the algorithm developed in [573]). Maintenance of a materialization when facts change, i.e., facts are added or removed from the *Knowledge Base*, is a well-studied problem but the approach of these researchers is innovative because adding expiration time information to each *RDF* statement, they shown that it is possible to compute a new complete and correct materialization by (a) dropping explicit statements and entailments that are no longer valid, and (b) evaluating a maintenance program that propagates insertions of explicit *RDF* statements as changes to the stored implicit entailments.

In the context of processing *RDF streams*, many different engines have been developed. These *RSP* engines can be broadly divided into two groups. Approaches inspired by *DSMS* exploit sliding window mechanisms to capture a recent and finite portion of the input data, enabling their processing through *SPARQL* operators [297] in an atemporal fashion. And *RSP*s influenced by *CEP* reactively process the input streams to identify relevant events and sequences of them. We call the former *semantic stream processing* systems, i.e. systems that inherit the processing model of *DSMS*s, but consider semantically annotated input, namely *RDF* triples, and define continuous queries by extending *SPARQL*. We call the latter *semantic event processing* systems, i.e. systems that pose their roots in a processing paradigm that is more similar to that of *CEP* systems, and offer operators for (temporal) pattern detection as the main building blocks for computation.

In the following we briefly describe are some of the most popular *semantic stream processing* systems:

Streaming-SPARQL [92]

Streaming-SPARQL is an extension of *SPARQL* designed for processing streams of *RDF* data. However, the main contributions of this work are theoretical. In particular, the authors mainly focus on the specification of its semantics using temporal relational algebra and provide an algorithm to automatically transform *SPARQL* queries into this newly extended algebra.

C-SPARQL [59, 60, 62, 63]

Continuous SPARQL (*C-SPARQL*) is among the first contributions in the area of *Stream Reasoning* and is often cited as a reference in the field. It is a new language for continuous queries over streams of *RDF* data with the declared goal of bridging the gap between the world of *Stream Processing* systems, and in particular *DSMS*s, and *SPARQL*. *C-SPARQL* is an extension of *SPARQL* to support continuous queries, registered and continuously executed over *RDF data streams*, considering windows of such streams. Supporting streams in *RDF* format guarantees interoperability and opens up important applications, in which reasoners can deal with knowledge that evolves over time. The distinguishing features of *C-SPARQL* are *(i)* the support for timestamped *RDF* triples, *(ii)* the support for continuous queries over streams, and *(iii)* the definition of ad-hoc, explicit operators for performing data aggregation, which is seen as a feature of primary importance for streaming applications (abandoned after the introduction of *SPARQL* 1.1 aggregates). The results of *C-SPARQL* queries are continuously updated as new (streaming) data enters the system.

SPARQL$_{\text{stream}}$ [116, 117]

SPARQL$_{\text{stream}}$ is another syntactic extension of *SPARQL* to enable queries over *RDF* streams and introduces S_2O, an extension to R_2O [67] for expressing mappings from streaming sources to an ontology. As *C-SPARQL*, is based on the idea of using *RDF* streams, i.e., *RDF* triples annotated with timestamps. In *SPARQL*$_{\text{stream}}$ these streams are virtual, relying on the original *data streams* for generating the query results, while *C-SPARQL* natively manages the *RDF* stream triples in its data model. Both include time windows for transforming infinite streams of data into bounded sequences to which other standard operators can be applied. Moreover, *SPARQL*$_{\text{stream}}$ considers time windows in the past (upper bound different to the current time) and adheres to the *SPARQL* 1.1 definition for aggregates.

CQELS [468]

CQELS is a native and adaptive query processor for unified query processing over *Linked Stream Data* and *Linked Data* [91, 522]. Similar to *C-SPARQL*, *CQELS* adopts the processing model of *DSMS*s, providing windowing and relational operators together with ad-hoc operators for generating new streams from the computed results. Differently from *C-SPARQL*, *CQELS* offers a processing model in which query evaluation is not periodic, but triggered by the arrival of new triples. The distinctive difference of this solution w.r.t. *C-SPARQL* is in the processing engine, which strictly integrates the evaluation of background and streaming data, without delegating them to external components. This makes possible to apply query rewriting techniques and optimizations well studied in the field of relational *DataBase*s. In contrast to the existing systems, *CQELS* uses a "white box" approach and implements the required query operators natively to avoid the overhead and limitations of closed system regimes. It provides a flexible query execution framework with the query processor dynamically adapting to the changes in the input data. During query execution, it continuously reorders operators according to some heuristics to achieve improved query execution in terms of delay and complexity. Moreover, external disk access on large *Linked Data* collections is reduced with the use of data encoding and caching of intermediate query results.

It is worth noticing that there are many more *semantic stream processing* systems, such as [49, 61, 493, 574].

In the following, we briefly describe some of the most popular *semantic event processing* systems:

***EP-SPARQL* (*ETALIS*)** [20–22, 25]

EP-SPARQL is a unified language for event processing and reasoning. Similarly to *C-SPARQL*, *EP-SPARQL* provides windowing operators (both count and time-based) for isolating portions of the input streams which will be processed by the system. However, differently from *C-SPARQL*, the main building blocks of the *EP-SPARQL* language are represented by a set of logical and temporal (sequence) operators that can be combined to express complex patterns of information items. Another notable difference is in the data model and consists in the way time is associated to *RDF* triples; while *C-SPARQL* associates one timestamp to each triple, representing a single point in time (point semantics), *EP-SPARQL* adopts two timestamps, which represent the lower and upper bound of the occurring interval (interval semantics). This reflects on output triples, whose occurrence intervals are computed from the input elements that contributed to their generation. The idea is promising and makes easy to write complex patterns involving content and time constraints on the input *RDF* triples. However, the approach used for writing patterns has some limitations: for example, when a pattern is satisfied by different sets of elements in the input stream, users do not have any operator for deciding which ones to select. As far as implementation is concerned, *EP-SPARQL* queries are translated in logic expressions in the *ETALIS Language for Events*

(*ELE*) [24] and computed at run-time using the event-based backward chaining algorithm of the *ETALIS* [23] engine, which converts queries to *Prolog* rules and evaluates them.

Sparkwave [336]
: *Sparkwave* is a system designed for high performance on-the-fly reasoning over *RDF data streams*. It trades complexity for performance: in particular, *Sparkwave* poses severe limitations to the size of the background knowledge, which must fit into the main memory of a single machine; moreover, it operates over a pre-loaded *RDF* schema and provides limited reasoning functionalities. *Sparkwave* implements a variant of the *RETE* algorithm [223], in which a pre-processing phase is used to materialize derived information before performing pattern matching.

INSTANS [504–506]
: INSTANS is an incremental engine for near-real-time processing of complex, layered, heterogeneous events. Based on the *RETE* algorithm [223], INSTANS performs continuous evaluation of incoming *RDF* data against multiple *SPARQL* queries. Intermediate results are stored into a β-node network. When all the conditions of a query are matched, the result is instantly available.

These and other systems have been deeply analysed in [410] and classified according to the application requirements described at the beginning of this chapter. More information about *RDF* Stream Processors implementations can be found in the *W3C RSP Community Group* wiki webpage about *RDF Stream Processors Implementation*[12].

Moreover, in the latest years, there have been other proposals based on different approaches, like [368, 492, 529] or the ones described in the next section. Nonetheless, these systems also lack complex reasoning capabilities, and the theory behind them is often not well-defined.

Regarding the query model of *RSP* languages, it is similar to the one of *CQL* shown in Figure 2.2. As shown in Figure 2.5, there are three classes of operators over *RDF* streams and *RDF* graphs:

S2R Stream to bounded *RDF*, which inherits the idea of *stream-to-relation* operators in *CQL* which produce a relation from a stream

R2R Bounded *RDF* to *RDF*, which inherits the idea of *relation-to-relation* operators in *CQL* which produce a relation from one or more other relations

R2S Bounded *RDF* to Stream, which inherits the idea of *relation-to-stream* operators in *CQL* which produce a stream from a relation

[12]https://www.w3.org/community/rsp/wiki/RDF_Stream_Processors_Implementation

$$\begin{tikzcd}
\textbf{\textit{RDF} Stream} \arrow[r, bend left, "\textit{S2R operators}"] & \textbf{\textit{RDF} Mappings} \arrow[l, bend left, "\textit{R2S operators}"] \arrow[loop right, "\textit{SPARQL operators}"]
\end{tikzcd}$$

Figure 2.5.: *CQL* extension for *RDF data streams*.

In these *RSP* operators the **R** denotes finite *RDF* graphs or mappings, as opposed to unbounded sequences of *RDF* graphs, i.e. streams. In addition to those operators (which can be thought as part of a *RSP Data Manipulation Language* (*DML*) in *SQL* terms), there is also the need for a *Data Definition Language* (*DDL*) to register a stream, register continuous queries, etc. Of all known *RSP* languages, only *C-SPARQL* has *DDL* primitives, but they are limited to query registration.[13]

There is so far no *RSP* language that can combine both paradigms (the one derived from *DSMS* and the one derived from *CEP*) under a clearly defined semantics, leaving a gap for those use cases that require this query expressivity. However, some initial attempts exist. In *C-SPARQL*, one can access the timestamp of a statement and specify limited forms of temporal conditions. *CQELS* recently proposed to integrate sequencing and path navigation [163], although it does not include typical selection mechanisms of *CEP* [160].

More information about *RSP* Query Features and Semantics can be found in the *W3C RSP Community Group* wiki web-pages about *RSP* Query Features[14], RSP Query Semantics[15] and Example of RSP-QL query[16].

Other details about querying *data streams* will be provided in the next section.

In the following we discuss some approaches proposed to evaluate and compare *RSP* systems.

Benchmarking *Stream Reasoning* Systems[17]

As mentioned before, one of the open challenges is benchmarking the existing *RSP* engines. The variety of implementations that have been proposed so far, result crucial differences in operational semantics. Even though all of these approaches try to solve similar challenges, they differ in various important aspects; among others,

[13]From https://www.w3.org/community/rsp/wiki/RSP_Query_Features
[14]https://www.w3.org/community/rsp/wiki/RSP_Query_Features
[15]https://www.w3.org/community/rsp/wiki/RSP_Query_Semantics
[16]https://www.w3.org/community/rsp/wiki/Example_of_RSP-QL_query
[17]Preliminary definitions adapted from [9, 173, 334, 335, 441, 469, 599]

they employ different underlying systems, query rewriting mechanisms, execution strategies and query semantics. A common benchmarking framework would help to assess differences and limitations of these existing implementations, but also provide a basis for steering future research directions and standardization efforts.

The *W3C RSP Community Group*[18] proposed two kinds of tests:

Soak testing addresses the system performance under the expected production load over a continuous period of time

Stress testing checks the response of the system under heavy loads

Moreover, they proposed also some metrics in order to evaluate those engines:

- Memory consumption
- Query execution time
- Query/Data throughput
- CPU usage
- Correctness of results
- Size of *Knowledge Base*
- *Reasoning*
- *Caching*

In addition, they have specified which (input) parameters should be used for Query and Data:

Query
- Number of joins
- Type of join
- Implies reasoning
- Number of streams
- Aggregation functions
- Selectivity
- Window size/slide

Data
- Variety of data (structure, values)

  Stream
  - Number of triples / graph
  - Input rate

  Background data
  - Location (local vs remote)
  - Size of the data:
    - storable in primary memory
    - storable in secondary memory

These *features* form the foundation for the creation of effective benchmarks.[19]

Several benchmarking systems that focus on different features of the *RSP* systems have been proposed so far.

First of all, it is worth mentioning the *Linear Road Benchmark* [33], which however, is designed for relational (traditional) *Stream Processing* systems and thus not suitable to evaluate graph-based queries on *LSD* processing engines. As originally designed to evaluate traditional *DSMSs*, the benchmark is based on the relational

[18]http://www.w3.org/community/rsp

[19]From https://www.w3.org/community/rsp/wiki/RSP_Benchmarking

data model, so it does not capture the properties of *RDF* graph data. Moreover, Linear Road does not consider interlinking the benchmark data set with other data sets; neither does it address reasoning. The benchmark simulates a traffic management scenario where multiple cars are moving on multiple lanes and on multiple different roads. The system to be tested is responsible to monitor the position of each car, and continuously calculates and reports to each car the tolls it needs to pay and whether there is an accident that might affect it. In addition, the system needs to continuously maintain historical data, as it is accumulated, and report to each car the account balance and the daily expenditure. Linear Road is a highly challenging and complicated benchmark due to the complexity of the many requirements. It stresses the system and tests various aspects of its functionality, e.g., window-based queries, aggregations, various kinds of complex join queries; theta joins, self-joins, etc. It also requires the ability to evaluate not only continuous queries on the stream data, but also historical queries on past data. The system should be able to store and later query intermediate results.

Then two complementary benchmarks have been proposed for the evaluation and continuous improvement of *RSP* engines: *LSBench* [469] and *SRBench* [599].
LSBench is mainly focused on understanding the throughput of existing *RDF* Stream processors and checking correctness by comparing the results of different processors and quantifying the mismatch among them.
SRBench is mainly focused on understanding coverage for *SPARQL* constructs.

The authors of *LSBench* observed that the following evaluation-related characteristics of these engines are critically important:

- The difference in semantics has to be respected, as the engines introduce their own languages based on *SPARQL* and similar features from *CQL*;
- The execution mechanisms are also different. *C-SPARQL* uses periodical execution, i.e., the system is scheduled to execute periodically (time-driven) independent of the arrival of data and its incoming rate. On the other hand, *CQELS* and *ETALIS* follow the eager execution strategy, i.e., the execution is triggered as soon as data is fed to the system (data-driven). Based on opposite philosophies, the two strategies have a large impact on the difference of output results.
- For a single engine, any change in the running environment and experiment parameters can lead to different outputs for a single test.

All these characteristics make a meaningful comparison of stream engines a non-trivial task. To address this problem, they proposed methods and a framework to facilitate such meaningful comparisons of *LSD* processing engines w.r.t. various aspects. Their major contribution is a framework coming with several customizable tools for simulating realistic data, running engines, and analysing the output.

Using a social network scenario, the benchmark uncovered conceptual and technical differences between *CQELS*, *C-SPARQL*, and *ETALIS*. Furthermore, it highlighted performance differences between these engines and included limited functionality and correctness tests. Because *LSBench* does not include means to determine the correct output, however, it does not provide absolute correctness figures to *RSP* engine developers. The benchmark is also not customizable for engines' varying execution strategies.

SRBench (*Streaming RDF/SPARQL* (*strRS*) Benchmark) aims at assessing the abilities of *strRS* engines in dealing with important features from both *DSMS*s and *Semantic Web* research areas combined in one real-world application scenario. That is, how well can a system cope with a broad range of different query types in which *Semantic Web* technologies, including querying, interlinking, sharing and reasoning, are applied on highly dynamic streaming *RDF* data. The benchmark can help both researchers and users to compare *strRS* engines in a pervasive application scenario in our daily life.

SRBench defines a set of queries that cover *RSP*-specific aspects, such as ontology-based reasoning or the application of static background knowledge to streaming data. The authors conduct a functional evaluation of the *RSP* engines *C-SPARQL*, *CQELS*, and *SPARQL*$_{\text{stream}}$ and conclude that the capabilities of these engines are still fairly limited. Due to the focus on functional aspects, *SRBench* does not recognize differences in the operational semantics of the benchmarked systems. To validate the query results, they propose correctness metrics such as precision and recall.

However, *LSBench* and *SRBench* do not consider the different operational semantics of the benchmarked systems in order to assess the correctness of query evaluation results. Therefore later *CSRBench* [173], an extension of *SRBench* to address correctness verification, has been proposed. The main motivations of the authors were that while these two evaluation efforts provide relevant contributions to the state of the art, one common limitation is that they do not consider checking the output produced by *RDF* stream processors. *SRBench* defines only functional tests in order to verify the query language features supported by the engines, while *LSBench* does not verify the correctness of the answers, but limits the analysis of correctness to the number of outputs. In sum, both benchmarks make two assumptions: *1*) the tested systems work correctly, and *2*) the tested systems have the same operational semantics. However, these assumptions do not always hold for all *RSP* engines, and hence these benchmarks may supply misleading information about them. In fact, *RDF* stream processors do not always adhere to their operational semantics, as

Feature	Operator	*C-SPARQL*	*CQELS*	*SPARQL*$_{\text{stream}}$
Report strategy	*S2R*	Window close and Non-empty content	Content-change	Window close and Non-empty content
Tick	*S2R*	Tuple-driven	Tuple-driven	Tuple-driven
Output operator	*R2S*	Rstream	Istream	Rstream, Istream and Dstream
Empty relation notification	*R2S*	Yes	No	No
Time unit		seconds	hundreds milliseconds	hundreds milliseconds

Table 2.5.: Classification of the *RDF* stream processors.

shown in [172]. Furthermore, even when *RDF* stream engines comply with their own semantics, these may differ from each other and therefore produce different but correct results. This means that it is considerably more difficult to compare these engines than those that process static *SPARQL* queries. Not only are correct answers determined by the input stream and the query, applying a given *SPARQL* extended algebra, but also by the operational semantics of each system.

Therefore, *CSRBench* (Correctness checking Benchmark for *Streaming RDF/SPARQL*) focuses on the correctness of stream query results. *CSRBench* evaluates *RSP* engines' compliance to their respective operational semantics using an oracle that determines the validity (i.e., correct or incorrect) of the query results. It thereby complements functional (*SRBench*) and performance (*LSBench*) evaluations so it takes first steps towards validating *RSP* engines but lacks comprehensive correctness evaluations over time. The authors find that none of the tested engines passes all tests and provide a detailed report on why certain engines fail at specific queries.

In the same paper, the authors proposed a characterization of the operational semantics of *RDF* stream processors. They defined a common model to capture the different behaviours of the systems taking into account two existing and well-known work of the data streaming world: *CQL* [32] and *SECRET* [96, 181], a framework to characterise and analyse the operational semantics of the window operators. They adapted these two models to be applied to *RDF* stream engines, defining a model that can be used to assess the correctness of the systems. The systems are implemented in different ways, but their operational semantics can be explained by the model they defined. These descriptions are important not only to foresee how the systems have to work (and consequently to compute the expected correct results) but also to highlight the differences between them. In Table 2.5 is reported the summary of their classification of *RDF* stream processors.

In the same years, also the well-known benchmark for reasoning over static datasets *Lehigh University Benchmark* (*LUBM*) have been extended to make it work for stream-based experiments. This new benchmarking system, called "SLUBM" [441], was conceived in order to preserve the semantics of the *LUBM* ontology while adding a time dimension (of semester unit) to the *KB*; this allows retaining most of the *LUBM*'s old standards.

Another interesting benchmark, related to the ***CityPulse*** Project described in Section 2.4.1, is the *CityBench Benchmarking Suite* [9]. The main motivation of this work was that available benchmarks for the evaluation of *RDF Stream Processing* (*RSP*) solutions are either synthetic or mostly based on static data dumps of considerable size that cannot be characterised and broken down, and so the need to benchmark *RSP* systems moving away from pre-configured static test-bed towards a dynamic and configurable infrastructure. Few of the existing *RSP* engines have been evaluated using offline benchmarks, but none of them has been tested based on features that are significant in real-time scenarios. There is a need for a systematic evaluation in a dynamic setting, where the environment in which data is being produced and the requirements of applications using it are dynamically changing, thus affecting key evaluation metrics.

Therefore, the author first identified a set of dynamic requirements of smart applications which must be met by *RSP* engines and then designed benchmarks based on such requirements, using real-time datasets gathered from sensors deployed within a real City, providing a testing environment together with a set of queries classified into different categories for evaluation of selected application scenarios. The Challenges of *Smart City Applications*, which can be useful to evaluate *RSP* systems, that they identified are:

- Data Distribution
- Unpredictable Data Arrival Rate
- Number of Concurrent Queries
- Integration with Background Data
- Handling Quasi-static Background Data
- On-demand Discovery of *data streams*
- Adaptation in *Stream Processing*

For more details about these Challenges and the respective Requirements for *RSP* system, see [9].

Recently a new benchmark framework for *RSP* engines, called *YABench* (Yet Another *RDF Stream Processing* Benchmark) [335], has been proposed in order to assess both correctness and performance of *RSP* engines. The main motivation of the authors was that existing benchmarks tackle particular aspects such as functional coverage, result correctness, or performance but none of them assesses *RSP* engine behaviour comprehensively w.r.t. all these dimensions.

YABench extends the concept of correctness checking and provides a flexible and comprehensive tool-set to analyse and evaluate *RSP* engine behaviour. It is highly configurable and provides quantifiable and reproducible results on correctness and performance characteristics. *YABench* provides means for the definition of test scenarios, generates reproducible test *data streams*, performs evaluation runs, and provides analyses of the results. It provides full reproducibility and emphasizes visual presentation of results to foster an understanding of engines' individual characteristics, including correctness under varying input loads, window sizes, and window frequencies.

YABench overcomes the limitations of *LSBench* by introducing a configurable oracle that allows emulating the behaviour of different engines. This is an essential requirement due to the fact that currently available engines do not agree on common operational semantics. Hence, the oracle represents a means to create reproducible results based on configurable operational semantics allowing comparing results from different engines along different dimensions such as performance and correctness. *YABench* extends the validation of *SRBench* implementing its same metrics but on a per-window basis and thereby makes it possible to quantify engines' retrieval performance on the most granular level. *YABench* extends the idea of oracle-based validation of *CSRBench* using more comprehensive correctness metrics (i.e., precision and recall) for each window. Moreover, they relate these correctnesses metrics directly to performance metrics such as delay in query result delivery or memory consumption and CPU utilization. Thereby, *YABench* provides insights into throughput and scalability and provides a comprehensive tool-set to investigate *RSP* engine characteristics, including both performance and correctness. In addition, their modular architecture also allows researchers to easily exchange the *RSP* engines, stream generators and continuous queries used in the benchmark.

In the latest years, another interesting framework on this topic has been introduced. It is called *LARS* and it is detailed described in depth in Section 2.3.2.

More information about benchmarks of *RSP* systems can be found on the *W3C RSP Community Group* wiki webpage about *RSP* Benchmarking[20].

2.2.4 Further remarks

For more information about Data Stream Reasoning see also [36, 64, 175, 420, 547].

[20]https://www.w3.org/community/rsp/wiki/RSP_Benchmarking

2.3 *Stream Reasoning* and *Logic Programming*

As mentioned before, the advancements in Internet and Sensor technology has created new challenges triggered also by the emergence of continuous *data streams*, like web-logs, mobile locations, or traffic data.
While existing *Data Stream Management Systems* allow for high-throughput *Stream Processing*, they lack complex reasoning capacities [570].

Many solutions have been developed also in the *Logic Programming* community. In the following, we focus mostly on *ASP* because it is the formalism we have used in our research works in this area.

2.3.1 In the *Answer Set Programming* community[21]

Answer Set Programming faces a growing range of increasingly complex applications. Many real-world applications, like planning or model checking, comprise parameters reflecting solution sizes. However, in the propositional setting of *ASP*, such problems can only be dealt with in a bounded way by considering, in turn, one problem instance after another, gradually increasing the bound on the solution size.

In the latest years, also the *ASP* community has been concerned with the problems related to the *Stream Reasoning* and researchers tried to find an approach to knowledge-intense *Stream Reasoning*, based on *Answer Set Programming* as a prime tool for *Knowledge Representation and Reasoning*. It is a big innovation for the *ASP* world because before *ASP* is usually used to solve "offline" problems (which means that entire problem is known a priori and can be solved without additional information).

However, the sheer amount and continuous flow of information produced by *data streams* precludes the direct application of *ASP*, simply because it is designed for singular reasoning from all available information. Unlike this, *Stream Reasoning*, instead, "restricts processing to a certain window of concern, focusing on a subset of recent statements in the stream, while ignoring previous statements" [64].

Solving such a problem with traditional *ASP* systems, like DLV or *clingo*, requires relaunching the system upon the arrival of each character. Although each time only the last two readings need to be taken into account, neither of the following ways to utilize standard *ASP* systems is satisfactory from a *KR&R* viewpoint: *(a)* one may

[21]Preliminary definitions adapted from [184, 249–252, 255, 259, 294]

add further rules to explicitly identify outdated readings (in order not to reason about them) among the whole data; *(b)* an external component may filter readings and pass only the most recent ones on to the *ASP* system.

To accommodate this in *ASP*, some years ago researchers at the University of Potsdam developed new techniques that allow us to formulate problem encodings dealing with emerging as well as expiring data in a seamless way. These researchers proposed first an incremental approach to both grounding and solving in *ASP*, with the goal of avoiding redundancy by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem. Then they proposed a reactive approach to *ASP* that allows us to implement real-time dynamic systems running online in changing environments and incorporating online *data streams*. We present their approach in the next paragraphs.

Background

To explain their approach we have first to introduce some background notions.

We define a (*parametrized*) *domain description* as a triple $(B,\ P,\ Q)$ of logic programs, among which P and Q contain a (single) parameter k ranging over the natural numbers. In view of this, we sometimes denote P and Q by $P[k]$ and $Q[k]$. The base program B is meant to describe static knowledge, independent of parameter k. The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k. The goal is then to decide whether the program

$$R[k/i] = B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$$

has an *Answer Set* for some (minimum) integer $i \geq 1$.

In order to provide a clear interface between program slices and to guarantee their compositionality, the researchers build upon the concept of *module* developed in [453].

Definition (Modular Logic Programs (definition from [453])).
We define a *logic program module* similarly to Gaifman and Shapiro [235], but consider the case of normal logic programs instead of positive (disjunctive) logic programs.
A triple $\mathbb{P} = (P, I, O)$ is a (propositional logic program) module, if

1. P is a finite set of rules of the form $h \leftarrow B^+, \sim B^-$;
2. I and O are sets of propositional atoms such that $I \cap O = \emptyset$;
3. $Head(P) \cap I = \emptyset$.

The *Herbrand Base* of module $\mathbb{P}$, $Hb(\mathbb{P})$, is the set of atoms appearing in P combined with $I \cup O$. Intuitively the set I defines the *input* of a module and the set O is the *output*. The input and output atoms are considered visible, i.e. the visible *Herbrand Base* of module $\mathbb{P}$ is $Hb_v(\mathbb{P}) = I \cup O$. Notice that I and O can also contain atoms not appearing in P, similarly to the possibility of having additional atoms in the *Herbrand Bases* of normal logic programs. All other atoms are hidden, i.e. $Hb_h(\mathbb{P}) = Hb(\mathbb{P}) \setminus Hb_v(\mathbb{P})$.

A *domain description* $(B, P[k], Q[k])$ is modular, if the modules

$$\mathbb{P}_i = \mathbb{P}_{i-1} \sqcup \mathbb{P}[i](O(\mathbb{P}_{i-1})) \quad \text{and} \quad \mathbb{Q}_i = \mathbb{P}_i \sqcup \mathbb{Q}[i](O(\mathbb{P}_i))$$

are defined for $i \geq 1$, where $\mathbb{P}_0 = \mathbb{B}(\emptyset)$.

A domain description $(B, P[k], Q[k])$ is *bound*, if, for all $i \geq 1$

$$atom(grd(B)) \subseteq head(grd(B)) \quad \text{and} \quad atom(grd(P[i])) \subseteq head(grd(B \bigcup_{1 \leq j \leq i} P[j]))$$

Incremental *ASP*

The researchers at the University of Potsdam propose to compute *Answer Sets* in an incremental fashion, starting from $R[1]$ but then gradually dealing with the program slices $P[i]$ and $Q[i]$ rather than the entire program $R[i]$. However, B and the previously processed slices $P[j]$ and $Q[j]$, $1 \leq j < i$, must be taken into account when dealing with $P[i]$ and $Q[i]$: while the rules in $P[j]$ are accumulated, the ones in $Q[j]$ must be discarded. For accomplishing this, an *ASP* system has to operate in a "stateful way". That is, it has to maintain its previous state for processing the current program slices. In this way, all components, B, $P[j]$ and $Q[i]$ are dealt with only once and duplicated work is avoided when increasing i.

Given that an *ASP* system is composed of a grounder and a solver, this incremental approach has the following specific advantages over the standard approach. As regards grounding, it reduces efforts by avoiding reproducing previous ground rules. Regarding solving, it reduces redundancy, in particular, if a learning *ASP* solver is used, given that previously gathered information on heuristics, conflicts, or loops, respectively, remains available and can thus be continuously exploited.
The researchers provide some empirical evidence using the incremental *ASP* system *iclingo*.

As mentioned above, the computation of *Answer Sets* consists of two phases: a *grounding* phase aiming at a compact ground instantiation of the original program and a *solving* phase computing the *Answer Sets* of the obtained ground program.

The incremental approach is based on the idea that the grounder, as well as the solver, are implemented in a stateful way.
As regards grounding, at each step i, the goal is to produce only ground rules stemming from program slices $P[i]$ and $Q[i]$, without re-producing previous ground rules. The ground program slices are then gradually passed to the solver that accumulates all ground rules from $P[j]$, for $1 \leq j \leq i$, while discarding the rules from $Q[j]$, if $j < i$.
Given a program P over A and $I \subseteq grd(\mathcal{A})$, we define an (*incremental*) *grounder* as a partial function $\texttt{ground}$: $(P, I) \mapsto (P', O)$, where P' is a program over $grd(\mathcal{A})$ and $O \subseteq grd(\mathcal{A})$. Thereby, P' stands for the ground program obtained from P, where the input atoms I provide domain information used to instantiate non-ground atoms in the rules of P. The output atoms in O essentially correspond to $head(P')$. Their main use is to carry state information, as O can serve as input to subsequent grounding steps.
A grounder ground is adequate, if for every program P over $\mathcal{A}$ and $I \subseteq \mathcal{A}$ such that is defined $\texttt{ground}(P, I) = (P', O)$, the following holds:

1. $(P \cup \{\{a\} \leftarrow \mid a \in I\}) \equiv (P' \cup \{\{a\} \leftarrow \mid a \in I\})$,
2. $\bigcup_{X \in AS(P \cup \{\{a\} \leftarrow \mid a \in I\})} (X \setminus I) \subseteq O \subseteq head(grd(P)|_Y)$ where $Y = I \cup head(grd(P))$,
3. for every $r' \subseteq P'$, there is some $r \in grd(P)$ such that $head(r) = head(r')$ and $body(r)^+ \setminus (I \cup O) \subseteq body(r')^+$.

The first condition expresses that P and P', each augmented with any combination of input atoms in I, must be equivalent. The second condition stipulates that all non-input atoms belonging to some *Answer Set* X of $(P \cup \{\{a\} \leftarrow \mid a \in I\})$ are contained in O. In addition, O must not exceed the head atoms of $grd(P)|_{I \cup head(grd(P))}$ in order to suitably restrict subsequently produced ground rules, using O as an input. Finally, the third condition forbids the introduction of rules that cannot be obtained from $grd(P)$ via permissible simplifications. Clearly, an adequate grounder may apply *Answer Set* preserving simplifications to compact its output.

We assume that atoms not occurring as the head of any rule are eliminated; even if such an atom becomes derivable later on when another program is added, it can thus not interact with the rules already present.
The reason for this design decision is that, although operating in an open environment, the possible addition of information or program slices, respectively, should not force the solver to continuously rebuild its existent data structures.

This approach, comprising incremental grounding and solving, matches exactly the semantics of (programs induced by) separated modular domain descriptions.

Reactive *ASP*

In order to capture dynamic systems, the researchers at the University of Potsdam take advantage of incremental logic programs [255], described before.

In this approach, reasoning is driven by successively arriving events. No matter when a request arrives, its logical time step is aligned with the ones used in the incremental program.

Grounding and solving in view of possible yet unknown future events constitutes a major technical challenge. For guaranteeing redundancy-freeness, the continuous integration of new program parts has to be accomplished without reprocessing previously treated programs. Also, simplifications related to events must be suspended until they become decided. Once this is settled, the approach leaves room for various application scenarios.

An *online progression* represents a stream of events and inquiries. While entire event streams are made available for reasoning, inquiries act as punctual queries.
We define an *online progression* $(E_i[e_i], F_i[f_i])_{i\geq 1}$ as a sequence of pairs of logic programs E_i, F_i with associated positive integers e_i, f_i.
An online progression is asynchronous in distinguishing stream positions like i from (logical) timestamps. Hence, each event E_i and inquiry F_i includes a particular time stamp e_i or f_i, respectively, indicated by writing $E_i[e_i]$ and $F_i[f_i]$. Such timestamps are essential for synchronization with parameters in the underlying (incremental) logic programs. Note that different events and/or inquiries may refer to the same time stamp.

Let $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ be a finite online progression and $(B, P[t], Q[t])$ be an incremental logic program. We define:

1. the k-expanded logic program of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ w.r.t. $(B, P[t], Q[t])$ as

$$R_{j,k} = B \cup \bigcup_{1\leq i\leq k} P[t/i] \cup Q[t/k] \cup \bigcup_{1\leq i\leq j} E_i[e_i] \cup F_j[f_j]$$

   for each k such that $1 \leq e_1, \ldots, e_j, f_j \leq k$, and
2. a reactive *Answer Set* of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ w.r.t. $(B, P[t], Q[t])$ as an *Answer Set* of a k-expanded logic program $R_{j,k}$ of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ for a (minimum) $k \geq 1$.

The incremental program constitutes the offline counterpart of an online progression; it is meant to provide a general (schematic) description of an underlying dynamic system. The parameter k represents a valid horizon accommodating all occurring events and inquiries. Thus, it is bound from below by the timestamps occurring in the online progression. The goal is then to find a (minimum) horizon k such that $R_{j,k}$ has an *Answer Set*, often in view of satisfying the global query $Q[t/k]$. In addition, inquiries, specific to each j, can be used for guiding *Answer Set* search. Unlike this, the whole stream $(E_i[e_i])_{1\leq i\leq j}$ of events is taken into account. Observe that the number j of events is independent of the horizon k. Finally, it is important to note that the above definition of an expanded program is static because its parameters are fixed.

Unlike offline incremental *ASP* [255], its online counterpart deals with external knowledge acquired asynchronously. When constructing a ground module, one can thus no longer expect all of its atoms to be defined by the (ground) rules inspected so far. Rather, atoms may be defined by an online progression later on.To accommodate this, potential additions need to be reflected and exempted from program simplifications, as usually applied w.r.t. (yet) undefined atoms. To this end, we assume in the following each (non-ground) program P to come along with some set of explicit ground input atoms referred to by I_P. Such atoms provide "hooks" for online progressions to later incorporate new knowledge into an existing program part.

Note that there are other concepts (like *Modular Online Progression*, *Compositionality*, *Mutually Revisability*, *Instantiation*) related to *online progression* but they are not needed for the purposes of this work, therefore, they will not be described.
For more information see [252].

They implemented a prototypical reactive ***ASP solver*** called *oclingo* which extends *iclingo* with online functionalities.
To this end, *oclingo* acts as a server listening on a port, configurable via its – port option upon start-up.
Unlike *iclingo*, which terminates after computing an *Answer Set* of the incremental logic program it is run on, *oclingo* waits for client requests.
To issue such requests, the researchers implemented a separate controller program that sends online progressions to *oclingo* and displays *Answer Sets* received in return.

Three parts are distinguished via the declarations '`#base.`', '`#cumulative t.`' and '`#volatile t.`' where `t` serves as the parameter. Of particular interest is the declaration preceded by '`#external`', delineating the input to the cumulative part provided by future online progressions.

The application-oriented features of *oclingo* also include declarations '`#forget t.`' in external knowledge to signal that yet undefined input atoms, declared at a step smaller or equal to `t` are no longer exempted from simplifications, so that they can be falsified irretrievably by the solver in order to compact its internal representation of accumulated incremental program slices.

Furthermore, *oclingo* supports an asynchronous reception of input. If new input arrives before solving is finished, the running solving process is aborted, and the solver is relaunched w.r.t. the new external knowledge.

Stream Reasoning with *ASP*

Later the researchers at the University of Potsdam proposed an *ASP*-based approach to *Stream Reasoning* based on the sliding window model. The idea is (i) to read an "offline" encoding just once and (ii) to keep only the n last entries of an "online" data stream.

They accomplish this by extending the previous approach to reactive *ASP* [252] by means for dealing with time-decaying program parts. While standard *ASP* solving deals with one problem instance at a time, they face continuously changing instances. They address this by proposing novel language constructs that allow for specifying and reasoning with time-decaying logic programs in an effective way. Moreover, they develop modelling techniques that are robust enough to handle changing data without continuous reprocessing or increasing memory demands.

Stream data often stays in a sliding window for several steps before it can (and should) be discarded so that it fits neither into the cumulative part P nor the query Q in a natural way. In order to address this shortcoming, they introduced the concept of *time-decaying logic programs*.

To provide a formal account of *time-decaying logic programs*, subject to emerging and expiring constituents, the researchers rely on *module theory* [453] for capturing the continuous composition and decomposition of program parts. To this end, they further extend the incremental and reactive module theory developed in [252, 255]. They also introduced directives for specifying the respective modules, leading to an extension of the pre-existing language of *oclingo*.

A *time-decaying logic program* Q^l is a logic program Q annotated with a *life span* $l \in N \cup \{\infty\}$; when $l = \infty$, we often write just Q. The life span allows for steering the expiration of non-persistent program parts, also called *transients*. To support this flexibility in practice, the *oclingo* language was augmented with new directives of the form:

```
      #volatile t [: l].
```

While `t` indicates the name written for the incremental parameter t in a (schematic) program $Q[t]$, the additional integer `l` gives the life span l of $Q^l[t]$. If `l` is omitted, as in the prior *oclingo* language, it is taken as 1, thus leading to $Q^1[t]$.

A *time-decaying incremental logic program* is a triple of the form

$$(B, P[t], \{Q_1^{l_1}[t], \ldots, Q_m^{l_m}[t]\})$$

in which $B, P[t], Q_1^{l_1}[t], \ldots, Q_m^{l_m}[t]$ are time-decaying logic programs. Such an incremental program serves as "offline" encoding of an underlying dynamic system. While ordinary incremental logic programs $(B, P[t], Q[t])$ specialize the decaying case to $(B, P[t], \{Q^1[t]\})$, the life spans $l_1, \ldots, l_m$ can diverge from 1 and one another.

A *time-decaying online progression*, representing a stream of lasting and transient program parts, is a sequence

$$(E_i[e_i], \{F_{1_i}^{l_{1_i}}, \ldots, F_{m_i}^{l_{m_i}}\}[f_i])_{i \geq 1}$$

of pairs in which $E_i, F_{1_i}^{l_{1_i}}, \ldots, F_{m_i}^{l_{m_i}}$ are time-decaying logic programs and e_i, f_i are positive integers. The latter represent minimum values assumed for the incremental parameter t in an associated "offline" (incremental) logic program.

In order to generalize the previous setting, beyond '`#volatile`' directives, they extended *oclingo*'s (external) controller component to additionally support the following:

```
    #volatile : l.
```

As with (transient) incremental logic program parts, `l` gives the life span l of a transient F^l.

The value of f_i (and e_i) is given in a '`#step` i.' directive, expressing that an underlying incremental program must have progressed to the position i of a reading in the stream.

The possibility of associating stream data with a life span (not fixed to 1) is essential to provide "automatic" reasoning support for sliding windows. If this possibility were unavailable, either the whole window contents would need to be provided as transient online input at each step, thus "replaying" part of the data when windows overlap, or rules referring to persistently added data would have to be deactivated once the data "expires".

The possibility of integrating recent additions without exhaustively reprocessing the entire collection of (non-expired) data and rules requires incrementally gathered program parts to be "compositional". This condition can be expressed in terms of modules [453].

A set A of atoms is an *Answer Set* of a module $\mathbb{P}$ if A is a (standard) *Answer Set* of $P(\mathbb{P}) \cup \{\{a\} \leftarrow \mid a \in I(\mathbb{P}) \cap A\}$; denote the set of all *Answer Sets* of $\mathbb{P}$ by $AS(\mathbb{P})$. For two modules $\mathbb{P}$ and $\mathbb{Q}$, the *composition* of their *Answer Sets* is $AS(\mathbb{P}) \bowtie AS(\mathbb{Q}) = \{A_{\mathbb{P}} \cup A_{\mathbb{Q}} \mid A_{\mathbb{P}} \in AS(\mathbb{P}), A_{\mathbb{Q}} \in AS(\mathbb{Q}), A_{\mathbb{P}} \cap (I(\mathbb{Q}) \cup O(\mathbb{Q})) = A_{\mathbb{Q}} \cap (I(\mathbb{P}) \cup O(\mathbb{P}))\}$. The module theorem in [453] shows that the semantics of $\mathbb{P}$ and $\mathbb{Q}$ is *compositional* if their join is defined, i.e. if $\mathbb{P} \sqcup \mathbb{Q}$ is *well-defined*, then $AS(\mathbb{P} \sqcup \mathbb{Q}) = AS(\mathbb{P}) \bowtie AS(\mathbb{Q})$. In *ASP* solving, compositionality eases adding new rules to a program, as it boils down to combining (without revising) the constraints characterizing *Answer Sets*.

The solving component of *oclingo* exploits this to successively integrate rules without large overhead; in particular, strongly connected components are only calculated locally once a new program part is added. As a consequence, the compliance of models computed by *oclingo* with *Answer Sets* of $P(\mathbb{P}) \cup P(\mathbb{Q})$ relies on $P(\mathbb{P}) \sqcup P(\mathbb{Q})$ to be well-defined. Otherwise, *oclingo*'s lightweight incremental processing cannot guarantee meaningful outcomes, and relaunching an *ASP* system from scratch would be required instead.

For turning programs into modules, we associate a (non-ground) program and a set of (ground) input atoms with a module imposing certain restrictions on the induced ground program.
To this end, for a ground program P and a set X of ground atoms, we define $P|_X$ as $\{\ h \leftarrow a_1, \ldots,\ a_m,\ not\ a'_{m+1}, \ldots,\ not\ a'_{n'} \mid h \leftarrow a_1, \ldots,\ a_m,\ not\ a_{m+1}, \ldots,\ not\ a_n \in P,\ a_1, \ldots,\ a_m \subseteq X,\ a'_{m+1}, \ldots, a'_{n'} = a_{m+1}, \ldots,\ a_n \cap X\ \}$.

In this work, the authors identified also a way to associate (non-ground) programs with (ground) modules and they formalized the definition of *modular time-decaying online progression*. For more information about this see [249, 250].

Novelties in *clingo* 4

As mentioned in Section 1.3.5, *clingo* 4 fully supersedes its special-purpose predecessors *iclingo* and *oclingo*. Briefly, in *iclingo* a program is partitioned into a *base* part, describing static knowledge independent of the step parameter `t`, a *cumulative* part, capturing knowledge accumulating with increasing `t`, and a *volatile* part specific for each value of `t`. These parts were delineated in *iclingo* by the directives `#base`, `#cumulative t`, and `#volatile t`. In *clingo* 4, all three directives are captured by `#program` declarations along with `#external` for volatile rules. Similar considerations can also be made for *oclingo*. Note that the `#external` directive here is a generalization of the one mentioned before.

Another innovative feature of *clingo* 4 is its incremental optimization. This allows for adapting objective functions along the evolution of a program at hand and can be really useful in the *Stream Reasoning* context.

Answer Set Programming* for *Stream Reasoning

In the same year the researchers at the University of Potsdam proposed their "Reactive ASP Solver", other researchers published a paper with three aims: *i*) to introduce a prototype of dlvhex *Stream Reasoning*, *ii*) to formalize *ASP* for building *Stream Reasoning* systems, and *iii*) to further apply *Semantic Web* techniques (*OWL*) for sensor-based applications.

This work is mainly interesting because the researchers at the La Trobe University defined a conceptual model that formalizes *ASP*-based *Stream Reasoning*.

According to them, a *Stream Reasoning* system has three main components, which are *a*) a sensor system, *b*) a *Data Stream Management System* (*DSMS*), and *c*) a *Stream Reasoner*.

Here we briefly introduce the notation used in the following paragraphs:

- dr: is the time period between the starting time and the finishing time of a reasoning process which always terminates.
- ds: is the time period between the starting time and the finishing time of a sensor taking a data sample (usually very small).
- Δs: is the time period between the two start times of taking two consecutive data samples of a sensor. The sample rate f_s is: $f_s = 1/\Delta s$.
- Δr: is the time period between the two start times of two consecutive reasoning processes of the reasoner. The reasoning rate f_r is: $f_r = 1/\Delta r$.

There are two communication strategies between the *DSMS* and the *Stream Reasoner*: *push* and *pull*. In the *pull* method, when the reasoner needs sensor data sample(s), it sends a query to the *DSMS* which will perform the query and return the data sample(s) to the reasoner. In the *push* method, the reasoner registers with the *DSMS* the sensor name from which it wants to have the data sample. The *DSMS* returns to the reasoner the data sample whenever it is available. They use the pull method in their prototype to discover the maximum reasoning speed of the reasoner when continuously running as fast as possible.

In the following, we introduce the formalization of the *data streams* provided to the *Stream Reasoner*. The time when a sample is taken is assumed to be very close to the time when that sample is available for reasoning, otherwise, the reasoner will give its result with a consistent delay.

Data Stream Data stream DS is a sequence of sensor data samples d_i ordered by timestamps. $DS = \{(t_1, d_{t_1}), (t_2, d_{t_2}), \ldots, (t_i, d_{t_i}), \ldots\}$ where d_{t_i} is the sensor data sample taken at time t_i, and $t_1 < t_2 < \ldots < t_i < \ldots$.

Data Window A data window available at time t, W_t, is a finite subsequence of a data stream DS and has the latest data sample taken at time t. The number of data samples, $|W_t|$, of this subset is the size of the window.
For $W_t \subseteq DS$, and $t_s = t : W_t = \{(t_1, d_{t_1}), (t_2, d_{t_2}), \ldots, (t_s, d_{t_s})\}$ where W_t is data window at time t, $s = |W_t|$ is the size of the window, $t_1 < t_2 < \cdots < t_s$, t_s is the time when the latest sample of the data window is taken and $d_{t_i}(1 \leq i \leq s)$ is the sensor data sample taken at time t_i.
The data window can also be defined by a time period, for example, a data window that includes all data samples taken in the last 10 seconds.

Window Slide Samples Window slide samples l is the number of samples counted from the latest sample (inclusive) of one data window to the latest sample (exclusive) of the next data window.

Window Slide Time Given two continuous data windows W_{t_1} at time t_1 and W_{t_2} at time t_2 ($t_2 \geq t_1$), the time period between t_1 and t_2 is called window slide time Δw, therefore $\Delta w = t_2 - t_1$.
The window slide time can be calculated with the formula: $\Delta w = l * \Delta s$. When we use the term "window slide", it means window slide samples or window slide time depending on context.

Data Window Stream Given a data stream DS, a data window stream WS is a sequence of data windows W in time order. $WS = \{(t_1, W_{t_1}), \ldots, (t_i, W_{t_i}), \ldots\}$ where W_{t_i} is a data window at time t_i, $t_1 < t_2 < \ldots < t_i < \ldots$, and $W_{t_i} \subseteq DS$.

Here we introduce a formalization of the *Stream Reasoner* of a system model that has one data stream and one reasoner. This is easily extensible to models that have: one data stream providing data for multiple reasoners, one reasoner using data from multiple *data streams*, and many reasoners using data from multiple *data streams*.

Data Window Reasoner An *ASP*-based data window reasoner AWR is a function that maps every data window $W \subseteq DS$ to a set SA of *Answer Sets*. $AWR : W_S \rightarrow 2^{\Sigma}$ where AWR denotes an *ASP*-based data window reasoner, W_S is the set of all data windows from data stream DS, Σ denotes the set of all possible *Answer Sets* S for any input, and 2^{Σ} is the power set of Σ.

The reasoner AWR has input data window W and gives a set SA of *Answer Sets*: $AWR(W) = SA$, where $SA = S_1, S_2, \dots, S_n$, and $S_i \in \Sigma, (1 \leq i \leq n)$.

For the implementation of their system, the researchers used a *UNIX shell script* to trigger the reasoner continuously, therefore the *Operating System* has to repeatedly load dlvhex, run it, and then unload it. This is resource consuming and can reduce reasoning speed.
However, since the new version of the ***Action Plugin*** allows to iterate the process of evaluation/execution of *Action Atoms* [222], this system could be improved using this new feature and the "*Environment*" that allows storing information from one iteration to another.

2.3.2 The *LARS* framework[22]

In the latest years researchers of the Vienna University of Technology started to observe that the recent rise of smart applications has drawn interest to logical reasoning over *data streams* and different query languages and *Stream Processing/Reasoning* engines were proposed; however, due to a lack of theoretical foundations, the expressivity and semantics of these diverse approaches were only informally discussed.

The emergence of sensors, networks, and mobile devices has generated a trend towards *pushing* rather than *pulling* of data in information processing. As mentioned above, in *Stream Processing*, studied by the *DataBase* community, input tuples dynamically arrive at systems in form of possibly infinite streams. To deal with unboundedness of data, the systems typically apply *window operators* to obtain snapshots of recent data. The user runs *continuous queries* on the latter that are triggered either periodically or by events, e.g., by the arrival of new input. And the *Continuous Query Language* (*CQL*) for *Stream Processing* has a clear operational semantics. However, in the *Stream Reasoning* area, different research communities have contributed to various aspects of this topic, leaving several challenges to overcome. First, these predominantly practical approaches often define semantics only informally, which makes them hard to predict and hard to compare. Second, advanced reasoning features are missing, e.g., non-monotonicity, non-determinism or model generation. According techniques have been studied almost exclusively on static data.

Therefore, they introduced the *Logic-based framework for Analyzing Reasoning over Streams* (*LARS*), which provides a rule-based formalism with different means to refer to or abstract from time, including a novel window operator, i.e., a flex-

[22]Preliminary definitions adapted from [81–83]

ible mechanism to change the view on streaming data. Moreover, *LARS* features a model-based semantics, and it offers besides monotonic also non-monotonic semantics that can be seen as an extension of *Answer Set Programming* (*ASP*) for *Stream Reasoning*. Hence, introducing a common ground to express various semantic concepts of different *Stream Processing/Reasoning* formalisms and engines, which can now be formally characterized in a common language, and thus be compared analytically.

These researchers formally defined the concept of **Stream** and the concept of **Windows function**, with the most common types of windows (*Time-based*, *Tuple-based* and *Partition-based*).

Moreover, as mentioned before, they defined the *LARS* Logical Framework, presenting a logic with different means for time reference and time abstraction and, on top of it, introducing a rule language with a model-based, non-monotonic semantics. In order to do this, they introduced the *Windows operator* and defined the syntax and the semantics for Formulas, specifying a Structure and defining the Entailment relation for them. Furthermore, they defined a rule language, with semantics similar to *Answer Set Programming*, for *Stream Reasoning*, formalizing the concepts of Rule, Program and *Answer Stream*.

For more information about the formal definitions and the semantics of these concepts, we refer the reader to [81–83].

In the same papers, they studied the complexity of some reasoning tasks in *LARS*. Let α be a formula, P a program, W a set of window functions evaluable in polynomial time, and let $B \subseteq \mathcal{A}$ be a set of atoms. We say that a stream $S = (T, \upsilon)$ *is over* $\mathcal{A}' \subseteq \mathcal{A}$, if $\upsilon(t) \setminus \mathcal{A}' = \emptyset$ for all $t \in T$.
The reasoning tasks considered are:

***Model checking* (MC)**

Given $M = \langle T, \upsilon, W, B\rangle$ and a time point t, check whether

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

***Satisfiability* (SAT)**

For decidability, we assume that relevant atoms are confined to (polynomial) $\mathcal{A}' \subseteq \mathcal{A}$. The reasoning tasks are:

- Given W, B, a timeline T and a time point t, is there an evaluation function υ on T such that $M, S, t \Vdash \alpha$, where $M = \langle T, \upsilon, W, B\rangle$ and $S = (T, \upsilon)$ *is over* $\mathcal{A}'$?

	α/α^-	P/P^-
MC	PSPACE/P	PSPACE/co − NP
SAT	PSPACE/NP	PSPACE/Σ_2^P

Table 2.6.: Reasoning in ground *LARS* (completeness results). From [82].

- Given W, B and a data stream D, does there exist an answer stream of P for D over $\mathcal{A}'$ (relative to W and B) at t?

In Table 2.6 is shown a summary of the complexity of reasoning in ground *LARS*, where α^-, P^- are formulas, respectively programs, with nesting of window operators bounded by a constant. Note that the problems refer to the more general notion of entailment but (hardness) results carry over to satisfaction.
More detailed complexity analysis are provided in following works (such as the ones mentioned below).

In addition, they demonstrated how the semantics of *CQL* can be expressed in *LARS*, and they studied the relation of *LARS* and *ETALIS*.

LARS is a starting point for intriguing research issues. Informally or operationally specified semantics of various state-of-the-art *Stream Processing/Reasoning* engines such as *CQELS*, *C-SPARQL*, and *SPARQL*$_{\text{stream}}$ may now be formalized, studied and compared rigorously in a common language. For practical concerns, tractable and efficient fragments of *LARS* are of interest; related to this are operational characterizations of its semantics.

In the following years, the same researchers extended and applied the work on the *Logic-based framework for Analyzing Reasoning over Streams* (*LARS*) in many different ways, briefly summarized in the list below:

- Presenting a generic algorithm for incremental Answer update of logic programs for *Stream Reasoning* with *ASP*-like semantics (*LARS* programs), based on *stream stratification* and an extension of *Truth Maintenance Systems* [503] techniques by temporal data management. [79]
- Establishing important steps towards formally comparing two *RSP* semantics implemented in two well-known engines, namely *C-SPARQL* and *CQELS*, by proposing translations to capture the languages and execution modes of the engines, and discussing how to formalize a notion of agreement between the two semantics as well as a condition for it to hold. [161, 162]
- Defining different notions of equivalence between *LARS* programs and giving semantic characterizations in terms of models, and then characterizing

the computational complexity of deciding the considered equivalence relations. [80]

- Proposing a generic architecture for generating/gathering streaming data, for evaluating different Stream Processors/Reasoners and for running those evaluations. [431]
- Using *LARS* to design novel techniques towards intelligent administration of *Content-Centric Networking* (*CCN*) [308] routers, developing an approach that allows for autonomous switching between existing strategies in response to changing content request patterns and obtaining flexible router configuration at runtime which allows for faster experimentation and may thus help to advance the further development of *CCN*. [77, 78]
- Developing a new *Stream Reasoner*, called *Laser*, that supports a pragmatic, non-trivial fragment of the logic *LARS*, that implements a novel evaluation procedure which annotates formulae to avoid the re-computation of duplicates at multiple time points. This procedure, combined with a judicious implementation of the *LARS* operators, is responsible for significantly better run times than the ones of other state-of-the-art systems. [76]
- Introducing the notion of *Tick Streams* to formally represent the sequential steps of a fully incremental *Stream Reasoning* system and developing a prototypical engine for well-defined logical reasoning over streaming data, called *Ticker*, based on a practical fragment of *LARS*. *Ticker* has two reasoning strategies: one utilizes *clingo* with a static *ASP* encoding, the other uses Truth Maintenance techniques [187] to adjust models based on the incremental encoding. [84]

2.3.3 Further remarks

It is worth noticing that a prominent exploration of the applicability of *ASP* for the *Semantic Web* is *StreamRule* [419], that is described in more details in Section 2.5.

Similar investigations have also been carried out in other logic-based formalisms, we do not describe them here and we refer the reader to the literature on the topic.

For more information about incremental update of logic programs, reactive reasoning, applications of *Logic Programming* in online/dynamic environments and, more broadly, *Logic Programming* and *Stream Reasoning* see also [26, 100, 104, 174, 176, 192, 254, 261, 423].

2.4 *Stream Reasoning* and *Smart City Applications*: a case study - The ***CityPulse*** project

In this section and in the following ones we first introduced some interesting case studies and then we report about some results obtained using *Stream Reasoning* techniques in the context of *Smart City Applications*.
The investigations reported in the following sections have been conducted during the visit in the *Reasoning and Querying Unit* (lead by *Dr. Alessandra Mileo*) of the *INSIGHT Centre for Data Analytics*[23] (formerly *DERI*) at the *National University of Ireland (NUI) Galway*[24].

2.4.1 CityPulse Real-Time *IoT Stream Processing* and Large-scale Data Analytics for *Smart City Applications*[25]

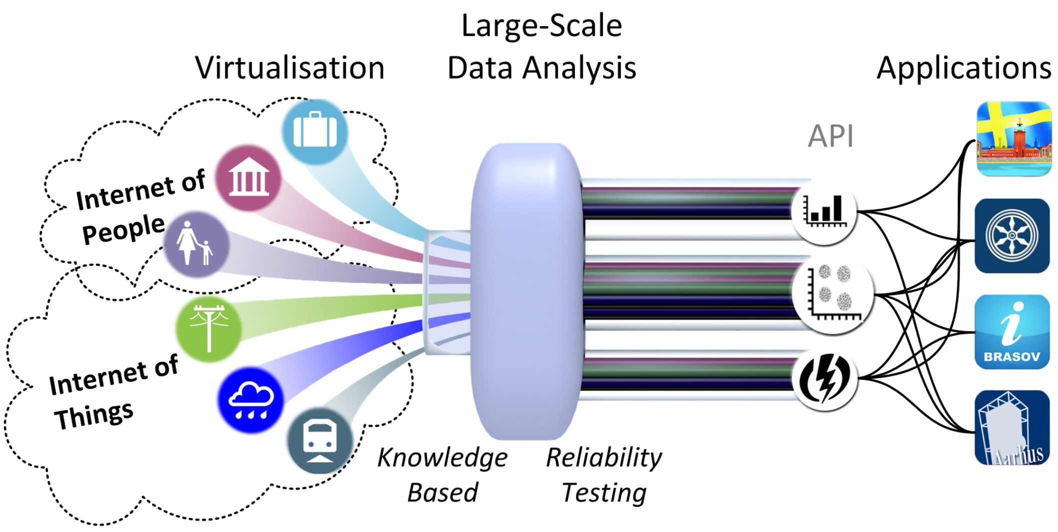


Figure 2.6.: Integrated Approach of ***CityPulse***. From [158].

An increasing number of cities have started to introduce new *Information and Communication Technology* (*ICT*) enabled services with the objective of addressing sustainability as well as improving the operational efficiency of services and infrastructure. In addition, there is an increasing interest in providing novel or enhanced service offerings and improved experiences for citizens and businesses.
The "*Smart Cities*" are evolving into a larger ecosystem or ecosystems that were previously disconnected. More and more applications and services in these ecosystems are going online. The city council is the pivotal facilitator in making this online ecosystem of ecosystems become a reality.

[23] https://www.insight-centre.org
[24] http://www.nuigalway.ie
[25] Preliminary definitions adapted from [480], and ***CityPulse*** website and documentation

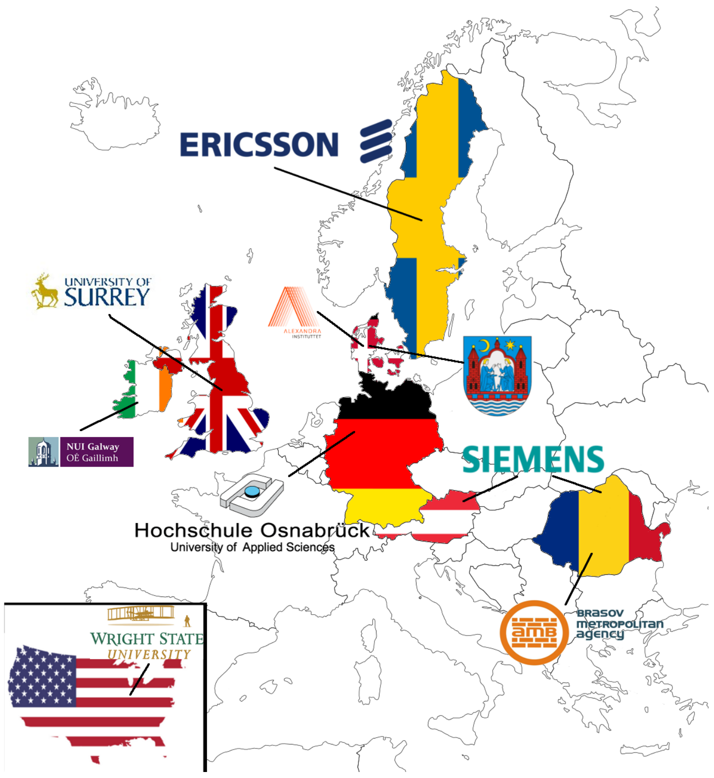


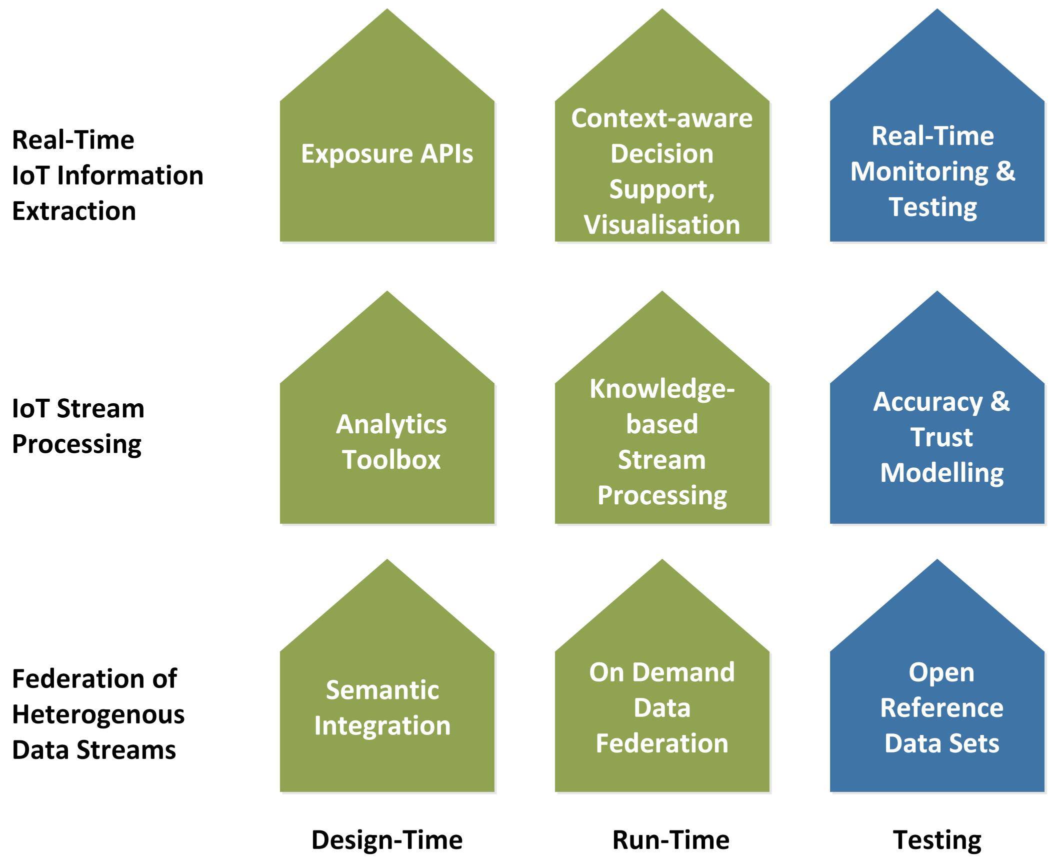
Figure 2.7.: ***CityPulse*** Partners. From [158].

The ***CityPulse***[26] project aims to develop innovative *Smart City Applications* by using an integrated approach to the *Internet of Things* and the Internet of People. It should facilitate the creation and provision of reliable real-time *Smart City Applications* by bringing together the two disciplines of knowledge-based computing and reliability testing (Figure 2.6).

The ***CityPulse*** framework supports *Smart City* service creation by means of a distributed system for semantic discovery, data analytics, and interpretation of large-scale (near-)real-time *Internet of Things* data and social media *data streams*. To goal is to break away from silo applications and enable cross-domain data integration. The ***CityPulse*** framework integrates multimodal, mixed quality, uncertain and incomplete data to create reliable, dependable information and continuously adapts data processing techniques to meet the quality of information requirements from end users. Different from existing solutions that mainly offer unified views of the data, the ***CityPulse*** framework is also equipped with powerful data analytics modules that perform intelligent data aggregation, event detection, quality assessment, contextual filtering, and *Decision Support*.

CityPulse is an international project, made by a consortium composed of 10 institutions from 8 different countries and 2 different continents, as shown in Figure 2.7.

[26]EU FP7 ***CityPulse*** Project under grant No.603095. http://www.ict-citypulse.eu



Real-Time IoT Information Extraction
Exposure APIs
Context-aware Decision Support, Visualisation
Real-Time Monitoring & Testing
IoT Stream Processing
Analytics Toolbox
Knowledge-based Stream Processing
Accuracy & Trust Modelling
Federation of Heterogenous Data Streams
Semantic Integration
On Demand Data Federation
Open Reference Data Sets
Design-Time
Run-Time
Testing

Figure 2.8.: Processing Steps during different Life-cycle stages. From [158].

The project is partially funded by the **European Commission's 7th Framework Programme (FP7)** under the contract number: *609035*.

Framework description

Smart City data is *Big Data*. It is multi-modal and varies in quality and format and representation form. The data needs to be processed, aggregated and higher-level abstractions need to be created from the data to make it suitable for the event processing, knowledge extractions and event processing applications that enable intelligent applications and services for *Smart City* platforms. Data needs to be integrated from various domains and the resulting knowledge exposed to various domains in a federated fashion.

CityPulse provides large-scale *Stream Processing* solutions to interlink data from *Internet of Things* and relevant Social Networks and to extract real-time information for the sustainable and *Smart City Applications*. The main objective of the project is 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.

The ***CityPulse*** framework is organized in three consecutive iteratively applied processing layers, covering federation of heterogeneous *data streams*, large-scale *IoT Stream Processing*, and real-time information processing and knowledge extraction. To achieve reliability, ***CityPulse*** integrates knowledge-based methods with reliability monitoring and testing at all stages of the data *Stream Processing* and interpretation. ***CityPulse*** provides solutions for the different life-cycle stages of data processing and utilization, supporting application development, i.e. design-time, application provision, i.e. run-time and obviously also the testing (see Figure 2.8).

Architecture of the *CityPulse* Framework

CityPulse provides novel approaches to support the seamless integration of dynamic *IoT*-enabled *data streams*, Internet of People data (i.e. relevant social media streams), and introduces knowledge-enabled intelligent methods for big *IoT* data analytics and *Smart City* services and processes.

An architectural overview of the approach which addresses the project's key issues and outlines the work packages is shown in Figure 2.9.

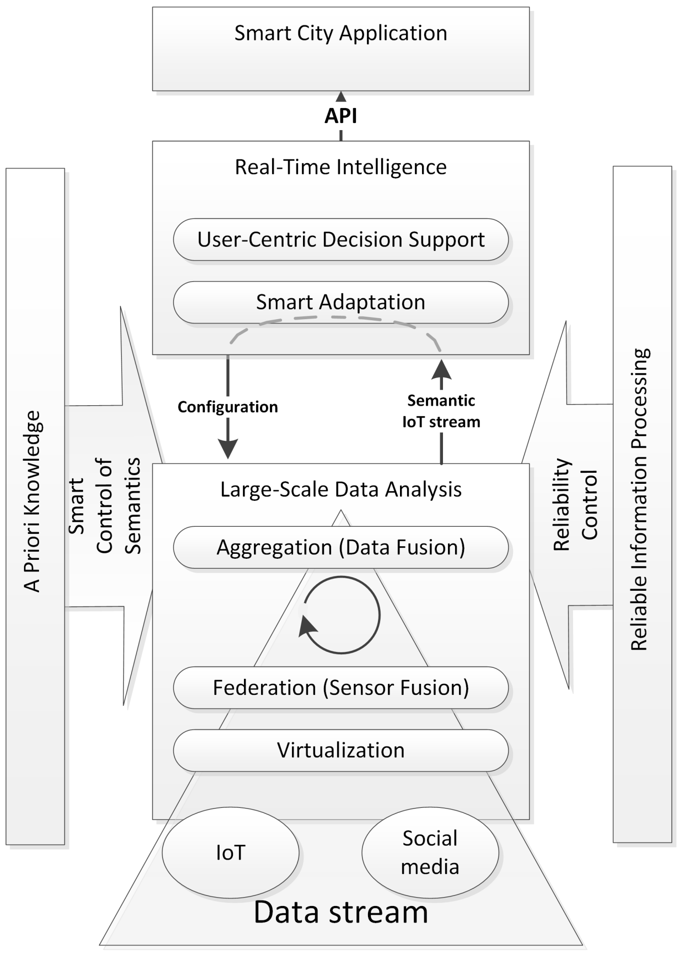


Figure 2.9.: ***CityPulse*** Framework Architecture Overview. From [158].

Briefly, the key issues addressed in ***CityPulse*** are:

Visualization Semantic annotation of heterogeneous data for automated discovery and knowledge-based processing

- Heterogeneous data sources
- Overcome silo architectures and provide common abstract interface
- Assigning semantic annotations to *data streams*

Federation On demand integration of heterogeneous Cyber-Physical-Social sources

- Sensor-fusion
- Combines heterogeneous *data streams* to one unified view

Aggregation Large-scale data analytics

- Data-fusion
- Reduce amount of data
  - Clustering
  - Summarization
  - Filtering
  - Pattern recognition

Smart Adaptation Real-Time interpretation and data analytics control

- Higher level information processing
  - Interpretation of semantic data
  - Transforming lower level dynamic information to higher level abstractions
- Enables adaptation of the data processing pipeline

User-centric *Decision Support* Context-aware customized *IoT* information extraction

- Goal: provide optimal configuration of *Smart City Applications*
- Social and context analysis
  - Matchmaking and discovery mechanisms
  - Match data according to users preferences and context

Reliable Information Processing Testing and monitoring accuracy and trust

- Challenge: Dynamic environments, changes and prone to errors
- Reliable data processing requires accuracy and trust
- Cope with malfunctions, disappearing sensors, conflicting data, ...
  - monitoring of streams (runtime)
  - testing of applications (design-time)

Smart City Applications *API* for rapid prototyping

- Cities challenge:
  - Rapidly growing digital economy requires new applications and information systems
- Provide an *API* for faster prototyping and access to ***CityPulse*** framework and information

For more information about ***CityPulse*** see also [480].

2.5 *Web Stream Reasoning* in Practice: on the Expressivity vs. Scalability tradeoff[27]

Web Stream Reasoning has emerged as a research field that explores advances in *Semantic Web* technologies for representing and processing *data streams* on one hand, and emerging approaches to perform complex rule-based inference over dynamic and changing environments on the other hand. Advances in the Internet and Sensor technologies converging to the *Internet of Things* (*IoT*) have also contributed to the creation of a plethora of new applications that require processing and make sense of web *data streams* in a scalable way.

As mentioned before, in the *Semantic Web* and *Linked Data* realm, technologies such as *RDF*, *OWL*, *SPARQL* have been recently extended to provide mechanisms for processing semantic *data streams* [63, 116, 468]. However, a variety of real-world applications in the *IoT* space require reasoning capabilities that can handle incomplete, diverse and unreliable input and extract actionable knowledge from it. Non-monotonic *Stream Reasoning* techniques for the (Semantic) Web have potential impact on tackling them.

Semantic technologies for handling *data streams* can not exhibit complex reasoning capabilities such as the ability to manage defaults, common-sense, preferences, recursion, and non-determinism. Conversely, logic-based non-monotonic reasoners can perform such tasks but are suitable for data that changes in low volumes at low frequency.

To reach the goal of combining the advantages of these two approaches in the last years a few works have been proposed; some tried to develop extensions of *ASP* [52] in order to deal with dynamic data [82, 250, 251], others tried to combine semantic stream query processing and non-monotonic reasoning [184, 419]. The *StreamRule* framework [419] is an example which provides a baseline for exploring the applicability of complex reasoning on *Semantic Web Streams*.

The conceptual idea behind *StreamRule* is to process *data streams* at different levels of abstraction and granularity, in such a way to guarantee that the amount of relevant data is filtered (and therefore reduced in size) as the complexity of the reasoning increases.[28] This has in principle a high potential in making complex reas-

[27]From **S. Germano**, T. Pham and A. Mileo. 'Web Stream Reasoning in Practice: On the Expressivity vs. Scalability Tradeoff'. In: *Proceedings of RR 2015*, pp. 105–112. DOI: 10.1007/978-3-319-22002-4_9.

[28]Note that in *ASP*, the expressivity of the language is strictly related to the computational complexity, therefore we refer to expressivity and (computational) complexity interchangeably.

oning on semantic streams feasible and scalable. However, the one-directional processing pipeline in *StreamRule* from query evaluation to non-monotonic reasoning is a strong limitation in exploring the expressivity versus scalability trade-off: the dynamic nature of web streams and their changing rate, quality and relevance makes it impossible to specify at design time what is the correct throughput and reasoning complexity the system can support and what window size and time-decay model is most suitable.

The main goal of this work is to provide a preliminary analysis on how we can improve the scalability of expressive *Stream Reasoning* for the *Semantic Web* combining continuous query processing and *Answer Set Programming* (*ASP*). We rely on the *StreamRule* [419] system as an instance of such an approach for implementation and testing, and we aim at providing general insights that hold for any *ASP*-based *Stream Reasoning* system.

The main idea we present in this work relies on concepts that can help make the *StreamRule* processing pipeline bi-directional or adaptive so that the expressivity versus scalability trade-off can be optimized in changing environments.

We start our investigation by identifying which the key features can potentially affect the expressivity versus scalability trade-off in a 2-tier *Web Stream Reasoning* system like *StreamRule*. The correlation between such features and their impact on scalability are then empirically evaluated by our practical analysis of performance and correlation between streaming rate, window size, properties of the input streams and complexity of the reasoning. Finally, some hints for discussion are presented based on our empirical results.

2.5.1 Core concepts for analysis

The key contribution of this work is to report on initial investigation on how to perform complex reasoning on web *data streams* maintaining scalability. We refer to scalability as to the ability to provide answers in an acceptable time with increasing input size and when the reasoning gets computationally intensive. We will introduce some key concepts that can later guide the design of heuristics for systems like *StreamRule* (which we will consider as a reference model in the remainder of this chapter), where query processing and non-monotonic reasoning features are adapted to continuously improve the expressivity versus scalability trade-off in changing environments. The conceptual architecture of *StreamRule* is based on a 2-tier approach to *Web Stream Reasoning*, shown in Figure 2.10 where query processing (first tier) is used to filter semantic data elements, while non-monotonic reasoning (second tier) is used for computationally intensive tasks.

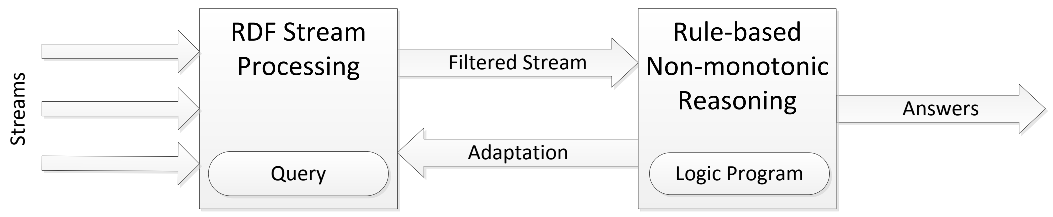


Figure 2.10.: 2-tier approach to *Web Stream Reasoning*.

We define the following concepts and notation:

Unit of Time (U) The unit of time is the time interval to which collected inputs are sent to the system (we will assume this as fixed in our analysis).

Reasoning complexity (C) We refer to the reasoning complexity as the computational complexity required to perform a given reasoning task involving a set of *ASP* rules. As mentioned earlier in this work, the computational complexity is strictly related to the language expressivity in *ASP*; in fact, more expressive language constructs in *ASP* correspond to higher computational complexity. The type of rules used within the *ASP* program affects grounding (which also affects memory consumption) and solving (which is related to computational complexity), and therefore has an impact on scalability. For simplicity, we assume in our analysis that the reasoning complexity is fixed (based on the rules in the program). However, this aspect deserves a more formal characterization to be able to use the reasoning complexity as a feature to design adaptive heuristics for optimization; we plan to investigate this in future work.

Streaming size (S) The streaming size is the number of input elements sent to the reasoning component every Unit of Time.

Window size (W) The (tuple based) window size[29] is the size of the input the reasoning component processes per computation.

Reasoning time (R_t) The reasoning time is considered as the time needed by the non-monotonic reasoner (second tier only) to compute a solution.[30]

$T(N)$ is the time needed by the reasoner to process N input elements.

$T_\omega(S, W)$ is the time needed by the reasoner to process a streaming size S dividing (and processing) it into windows of size W. The number of windows (and therefore the number of computations needed) is $\lceil \frac{S}{W} \rceil$. Formally $T_\omega(S, W) = \lceil \frac{S}{W} \rceil \times T(W)$.

[29]In this work we only consider non-overlapping windows. For overlapping windows, the formula $T_\omega(S, W)$ should hold also when duplicating events in overlapping parts.

[30]Note that this is different from the total processing time, which includes the time required for query processing (first tier). In this work, we mainly focus on the reasoning time only, relying on the extensive evaluation of query processing engines for the query processing time, as in [468].

S_u is the number of elements that can be processed by the reasoner in one unit of time.[31] Formally $S_u = N$ s.t. $T(N) = 1$.

S_l is the maximum number of elements that can be processed within one unit of time using a proper windows size W.

The question summarizing our problem is as follows: *Given a fixed streaming size S with fixed complexity C and unit of time U, find a window size W such that*

$$T_\omega(S, W) \leq U \tag{Q1}$$

Finding this window size and being able to adapt it to changing streaming rates would reduce the bottleneck between the two tiers since it will ensure that the non-monotonic reasoner can keep up with the results produced by the query processing engine without the cumulative delay experienced in *StreamRule*. Previous experiments in [419] showed that the current implementation of *StreamRule* with *CQELS* as query processor and *clingo* as *ASP* reasoner encounters a bottleneck when the non-monotonic reasoner returns results after the next input arrives from the stream query processing component, thus cumulating a delay that makes the system not scalable. Making the process bi-directional requires to dynamically provide answers to Q1.

We can observe that if $T(N)$ is monotonically increasing, we have that

$$\forall S' \; where \; S_u \leq S' \leq S_l, \; \exists W' \; s.t. \; T_\omega(S', W') \leq U$$

This is our case, as illustrated in our empirical evaluation.

2.5.2 Experiments

In this section, we present the scenario, dataset, rule-set, and platform we used for the empirical evaluation of our trade-off analysis and discuss our findings.

Scenario

Consider a user moving on a path. She wants to know real-time events that affect her travel plan to react accordingly. The *Stream Reasoning* system receives events as *Linked Stream Data* 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 user's location and activity).

[31]Note that this is different from the streaming size.

With this information as input stream, the *Web Stream Reasoning* system is in charge of *i*) selecting among the list of events, which are the ones that are really relevant according to the user's context, and *ii*) continuously ranking their level of criticality w.r.t. the user task and context[32], in order to decide whether a new path needs to be computed.

The query processing component filters out events which are unrelated to the user, e.g. events are not on the user's path, thus limiting the input size for the non-monotonic reasoner. The reasoner receives as input filtered events and an instance of the context ontology related to the activities and status of the user, provides ranked critical events as output. The event includes 4 attributes: type, value, time, and location. For example:

```
event(weather, strong-wind, 2014-11-26T13:00:00, 38011736-121867224)
```

describes the condition of weather being strong wind at a certain time and a given location.

Dataset

For our experiment, we generate traffic events based on 10 types 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 values, e.g. traffic condition has values: good, slow, congested. In addition to that, we create a small instance of the context ontology which describes the effect of events on certain activities.

Rule-set

The *ASP* rule set we used for this experiment includes 10 rules which have 2 negated atoms (using negation-as-failure). Since the complexity of the reasoning is fixed and related to a specific program in our setup, we do not quantify the complexity in this initial investigation.

Platform

We used the state-of-the-art *ASP* reasoner *clingo* 4.3.0 and *Java* 1.7. The experiments were conducted over a machine running Debian GNU/Linux 6.0.10, containing 8-cores of 2.13 GHz processor and 64 GB RAM.

[32]In the current implementation we evaluate criticality mainly based on how close an event is to the user location, and how fast is the user moving. In future work we plan to extend this contextual characterization to consider not only location but also other features such as the user transportation type, user's health condition etc.

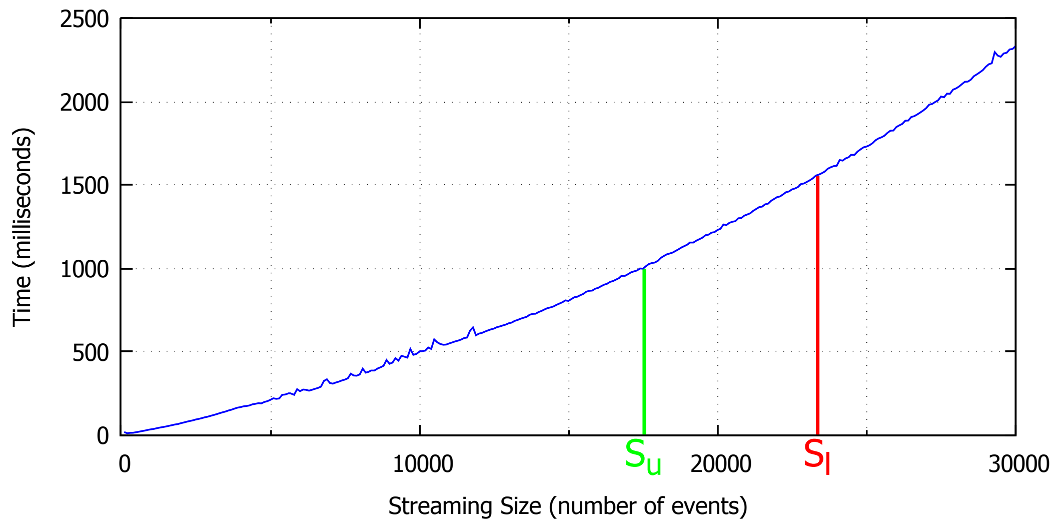


Figure 2.11.: Reasoning time.

Empirical Results

We evaluated the same *ASP* program with varying input size S (from 100 to 30000 events) and measured the reasoning time of the system ($T(S)$). We trigger the reasoner 20 times for each S and then we computed the *interquartile mean* (*IQM*) to smooth results. These values are plotted in Figure 2.11.

Given $U = 1\ s$, the graph shows $S_u = 17520\ events$. In other words, for this particular case (and fixed Rule-set and Platform), the *Stream Reasoning* system will be "stable" if the streaming size of the *ASP* reasoner is smaller than $17520\ events$. For streaming size bigger than $17520\ events$, the system will cumulate a delay that will cause a bottleneck. Giving our function is monotonically increasing, there are some streaming sizes bigger than S_u that can be processed in less than $1\ U$. We then investigated the idea of dividing events in windows, assuming we can find a split such that the correctness of the result will not be affected[33].

The easiest way to perform this split is to consider several windows of the same size. For example, consider $S = 20000\ events$, it will take $1232\ ms$ for the reasoner process all $20000\ events$ in one computation ($T(20000) = 1232\ ms$). The whole system will combine a delay in each computation and therefore will crash at some point. However, if we use the window size $W = 5000\ events$ ($T(5000) = 216\ ms$), the reasoner will take $T_\omega(20000, 5000) = \lceil\frac{20000}{5000}\rceil \times T(5000) = 4 \times 216\ ms = 864\ ms$ for processing S, using 4 computations. So we have found a proper window size (W) such that $T_\omega(S, W) \leq U$; in other words we have found a split for which the system remains stable.

[33]Algorithms to perform such splits are under investigation and will be the subject of future work.

Moreover, if we divide S into windows of size $W = 2000\ events$, the reasoning time for S will be $T_\omega(20000, 2000) = 720\ ms$, so, also in this case, T_ω is less than or equal to $1\ U$. Therefore, in general, there may be more than one way to split the events.

For any given S, a proper value[34] for W such that $T_\omega(S, W) \leq U$ can be found in a trivial way just checking for each streaming size (S') less than S_u the time required ($T(S')$) and verifying that $T_\omega(S, S') = \lceil \frac{S}{S'} \rceil \times T(S') \leq U$; when we find such S', we can put $W = S'$.

Running this algorithm increasing S up to the point where we cannot find any streaming size S' less than S_u such that $T_\omega(S, S') \leq U$, we can compute the value of S_l. Based on this we found that for this experiment the value of S_l is $23350\ events$. It means that the system can scale if the streaming size is less than or equal to $23350\ events$.

We can also apply these algorithms to the trend line function that fit the data in order to have a more precise result. In our experiment we found an Order 2 polynomial trend line which fit very well our data with an R-squared value of 0.99979 and we have used this to find the values of S_u and S_l.

2.5.3 Discussion.

Based on our experiments, we observe that:

- Given a unit of time and a particular *ASP* program, we can find an optimal window size for a given streaming size for reducing the processing time of the system.
- This conclusion holds if there is no dependency between input events for the reasoning component[35].

We are currently investigating how to generalize our empirical results to set the basis for designing adaptive heuristics for *Web Stream Reasoning*. An improvement to this approach, for some scenarios, is to find the value of W that minimize $T_\omega(S, W)$. A key aspect we are also considering is to provide a formal characterization that helps to relax the assumption of independence between input events, in order to determine how to find an optimal number of window for a given streaming rate and a given *ASP* program. Since we started our empirical evaluation based on a given *ASP* program, another interesting direction will be to investigate more in-depth how the complexity of the reasoning affects our analysis.

[34]Note that our goal is not to find the minimum, we just want to find one split.
[35]Note that this assumption needs to be formally characterized and more investigation is ongoing in this direction.

2.6 Automatic Configuration of *Smart City Applications* for User-Centric *Decision Support*[36]

Smart City Applications require *Internet of Things* (*IoT*) discovery and matchmaking techniques dedicated to dynamicity handling. Information taken into account during the matchmaking process originates from diverse data sources including *data streams*, city services, the user's social context, situational awareness (e.g., user location), preferences and application configurations. The exponential growth in the availability of information from numerous data sources raises several difficulties in implementing, sustaining, and optimizing operations and interactions among different city departments and services [437]. There is a strong need for *Smart City Application* tools which support easy development of "smart applications".

The state-of-the-art for *Smart City Frameworks* has major focus on existing *Smart City* platforms and the existing works are mainly in four key areas: (i) data acquisition (ii) semantic interoperability, (iii) real-time data analysis and event detection, and (iv) *Smart City Application* development support. Among the existing frameworks such as PLAY [552], iCore [281], and STAR-CITY [357], ***CityPulse*** [480] is the only framework supporting all four previously mentioned features. As mentioned before, in addition to data acquisition and semantic interoperability, the ***CityPulse*** framework provides a complete set of domain-independent real-time data analytics tools such as data federation, data aggregation, event detection, quality analysis and *Decision Support*. The application development is supported by a set of *APIs* provided by ***CityPulse***. In this work, we focus on the decision-making process, which is designed and implemented within ***CityPulse*** framework.

The *Decision Support* component in ***CityPulse*** supports the complex reasoning capabilities that are required in various *Smart City Applications* such as non-monotonic, non-deterministic, and recursive reasoning. This component represents higher-level intelligence, strongly connects to user application layer, and acts as a flexible interplay between user-centric factors and dynamic aspects of the changing environment within the city. Factors such as user interests and reputation requirements are also considered in the *Decision Support* process. The exploitation of such factors along with richer user profiles has a great potential for providing more personalised *De-*

[36]From A. Mileo, **S. Germano**, T.-L. Pham, D. Puiu, D. Kuemper and M. I. Ali. *User-Centric Decision Support in Dynamic Environments. CityPulse - Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications*. Report - Project Delivery. Version V1.0-Final. NUIG, SIE, UASO, 31st Aug. 2015. URL: http://cordis.europa.eu/docs/projects/cnect/5/609035/080/deliverables/001-609035CITYPULSED52renditionDownload.pdf (visited on 25th Sept. 2017).
And T. Pham, **S. Germano**, A. Mileo, D. Kümper and M. I. Ali. 'Automatic configuration of smart city applications for user-centric decision support'. In: *Proceedings of ICIN 2017*, pp. 360–365. DOI: 10.1109/ICIN.2017.7899441.

cision Support, and greatly improve user experience when interacting with *Smart City Applications*. Although elicitation and usage of user profiles are optional, motivated by their potential we included explicit aspects of user profiles in the *Decision Support* process which are automatically encoded and used to configure the way the *Decision Support* process works. These aspects include not only user location but also user preferences and constraints on the solutions provided, as well as dynamic correlations between contextual activities and their dependencies with city events for a particular user in specific application scenarios.

The ability to continuously characterize the correlation between city events and user activities is used to dynamically filter events that are relevant for a particular user at a specific time so that *Decision Support* can be instructed to find new solutions whenever needed. This functionality has been specified and implemented in a component called Contextual Filtering [480]. In the proposed characterisation, events, user activities and their dependencies are modelled using *Linked Data* and open vocabularies in order to provide a lightweight, interoperable and well-established foundation for *Decision Support*.

In the following we focus on user-centricity and describe how user requirements in terms of preferences and constraints can be explicitly specified and mapped into a representation that is independent of the specific application.

2.6.1 User-Centric *Decision Support*

The *Decision Support* component is responsible for higher-level intelligence, which can utilize user contextual information, background knowledge, and real-time events to deduce intelligent conclusion in real-time. This component is also capable of acquiring the analysing additional information sources related to user contextual patterns, users' application usage behaviour, and self-defined preferences (while using a *Smart City Application*) to provide optimal configuration for *Smart City Applications* and enable these applications to generate reactive application logic within deployed *Smart City Applications*. Figure 2.12 represents a general information flow and interactions of *Decision Support* with other external components.

As an initial step to start the information processing, *Decision Support* receives following input: (i) *a reasoning request* from the application interface which includes user related **Functional Parameters** and **Non-Functional Parameters**, **Functional Constraints**, and **Functional Preferences**, (ii) *background knowledge*, which is domain dependent information available for reasoning and application logic strictly tied to the given scenario, and (iii) *external information sources*, which is any relevant information collected through external components required for that specific

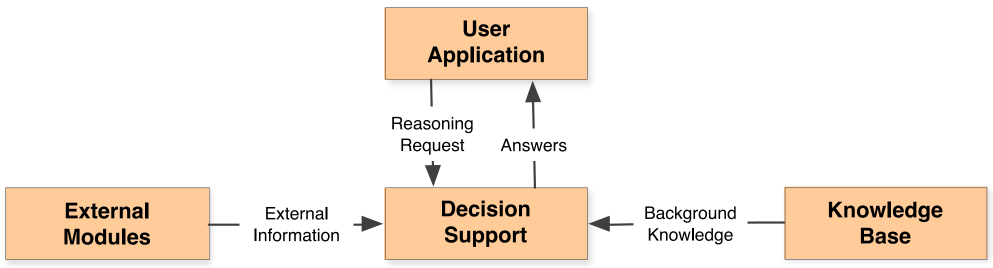


Figure 2.12.: *Decision Support* I/O.

scenario. After processing all related input information as mentioned above, *Decision Support* generates a set of scenario-driven solutions, which are guaranteed to be optimal and satisfying all requirement and preferences specified by the individual user.

In real-world scenarios, the reasoning module has to deal with issues related to incomplete and contradictory information, diverse and unreliable input data, and most importantly user defined constrained and preferences that are not explicitly input by the end user. In order to better support the provision of optimal decisions, the reasoning module must have the ability to expressively deduce information from the information collected through internal and external modules additional to the user-defined input. We achieved this expressivity within *Decision Support* by opting to use a declarative non-monotonic logic reasoning approach based on *Answer Set Programming* (*ASP*). All input information of *Decision Support* is mapped into *ASP*-format rules. *Decision Support* combines mapped rules with already existent domain-dependent rules, to design an application logic for the provision of optimal solutions to the users. Figure 2.13 depicts a system sequence diagram of *Decision Support* to showcase all interactions and processing steps involved in this component. In what follows, we briefly elaborate each and every information processing step of the *Decision Support* component.

- *Request Handler* receives a *ReasoningRequest* as an input from the application containing user preferences and requirements. Whenever a new reasoning request arrives, a new instance of *Decision Support* is initiated. This ReasoningRequest is interpreted as the *InterpretedRequest* and used to initiate the *DS Manager*.
- *Request Re-writer* automatically generates the logic rules required for the specific reasoning request that is received from DS Manager, the detailed process of automated mapping and rules generated is presented in Section 2.6.2. After receiving the rules from Request Rewriter, the DS Manager asks *CoreEngine* to perform the reasoning by sending InterpretedRequest and Rules as parameters.

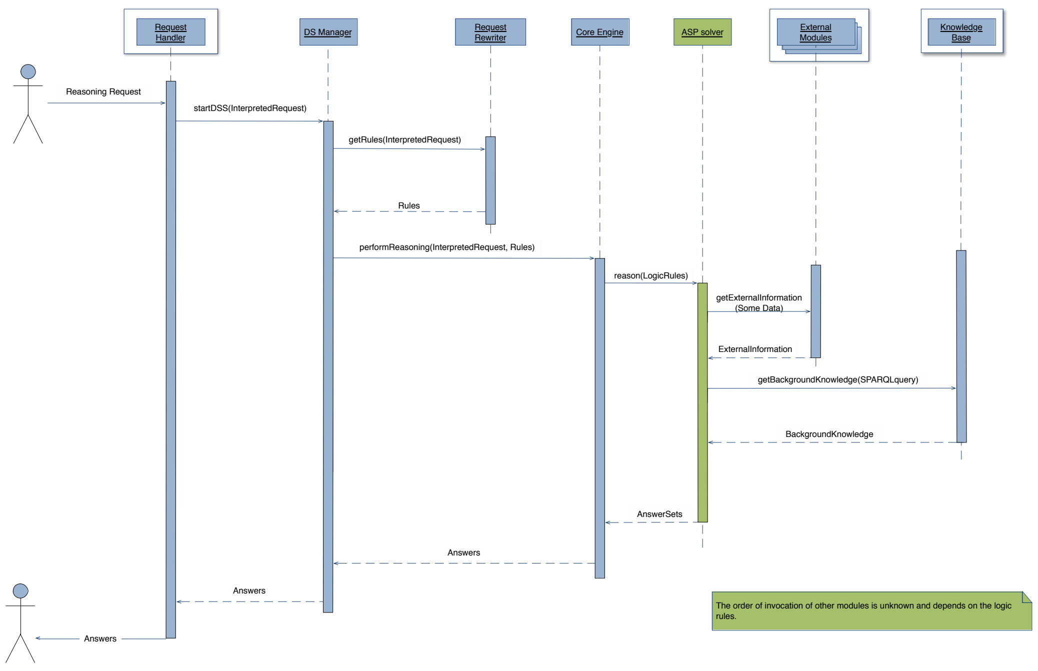

Request Handler
DS Manager
Request Rewriter
Core Engine
ASP solver
External Modules
Knowledge Base
Reasoning Request
startDSS(InterpretedRequest)
getRules(InterpretedRequest)
Rules
performReasoning(InterpretedRequest, Rules)
reason(LogicRules)
getExternalInformation (Some Data)
ExternalInformation
getBackgroundKnowledge(SPARQLquery)
BackgroundKnowledge
AnswerSets
Answers
Answers
Answers
The order of invocation of other modules is unknown and depends on the logic rules.

Figure 2.13.: *Decision Support* sequence diagram.

- *CoreEngine* is a component that executes the *ASP* solver (in our current implementation, we use Clingo4 [260] as the *ASP* solver) using the EMBASP framework[37] which is able to invoke the *ASP* solver and to collect *Answer Sets* as plain *Java* objects (explained in details in Section 5.2). The *ASP* solver starts by collecting:
  - *ExternalInformation* refers to additional information collected through external modules, which can vary depending on the scenario. For example, in the *Travel Planner* scenario (see Section 2.6.3), the possible routes from starting point to ending point or the latest city events can be considered as *ExternalInformation*. *Decision Support* directly interacts with external modules within the *ASP* program, using "external atoms". As mentioned in Section 1.3.5, by using them, the *ASP* reasoner is able to invoke the external modules interactively, only on need basis and can also re-use derived answers for other reasoning tasks. This feature offers a very powerful ability on-demand composition of reasoning tasks to provide solutions.
  - *BackgroundKnowledge* is static information containing important facts and rules related to a particular domain and stored internally for reasoning tasks. For example, in the Parking Space scenario (see Section 2.6.3), the locations of the parking spaces are part of background knowledge.

[37]https://www.mat.unical.it/calimeri/projects/embasp

- The *ASP* Solver combines *ExternalInformation*, *BackgroundKnowedgle*, and *Rules* to compute the optimal answers (in the form of *Answer Sets*) satisfying users' defined constraints and preferences.
- EMBASP *Framework* processes *Answer Sets* and the optimal selected *Answers* (as objects) are sent to the CoreEngine and then back to the user application for visualization.

The actual deduction process for generating solutions to the *Decision Support* task required by the application is performed by combining background knowledge, external information, user preferences & constraints and scenario-dependent rules. The fully declarative nature of the *ASP* framework used in the implementation of the *Decision Support* component capabilities makes it possible to combine these rules and knowledge facts in a straightforward way, and enables full exploitation of the expressive power of *ASP* inference for constraint checking and preference-based deduction. This very same declarative feature is likely to simplify the extension of *Decision Support* provided within the ***CityPulse*** framework and to reuse for other *Decision Support* tasks.

2.6.2 User-Centric *Decision Support* Request: Specification and Mapping

Preference-driven and constraint-based reasoning provide a powerful mechanism where user-centricity is a key feature and enables to find an optimal match between the needs, preferences of citizens, available *data streams* and city services. This section describes the specification of a reasoning request. We focus on user Preferences and Constraints and their automatic mapping into declarative deduction rules. These rules are then used in the *Decision Support* process for solution optimization.

In what follows we will detail each element of the Reasoning Request and define the automatic mapping or translation into deduction rules used by the *Decision Support* component. Such translation is defined in a general way so that independently of the **Functional Details** defined as strings and values, an automatic declarative rule-based specification can be obtained, which is seamlessly combined with the rules in the *Decision Support* module used by a specific application.

As illustrated in Figure 2.14, the Reasoning Request consists of:

- **Type (***T***)** determines the reasoning task required by the application. This is used directly by *Decision Support* to perform the correct task using the reasoning engine and needs to be identified among a set of available options at

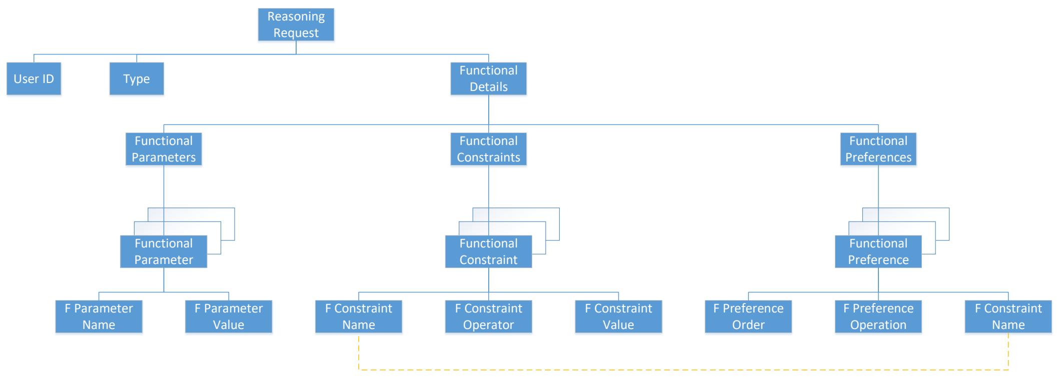


Figure 2.14.: Representation of Reasoning Request.

design time by the application developer. Such options have been defined for the implemented scenarios. Customization and extension of available types will be possible via *APIs*.

- **User Reference** is an identifier of the user that made the request. Such reference is meant to be a unique identifier related to user credentials that will be used in the final integration activities in order to manage user logins and instances of the ***CityPulse*** framework in different cities.
- **Functional Details** represent the actual criteria for the reasoning task required by the user (i.e. constraints and preferences for solution optimization). **Functional Details** include **Functional Parameters**, **Functional Constraints**, and **Functional Preferences**.

We shall focus now on the specification of each of the aspects included in **Functional Details**, and illustrate how they are automatically mapped and translated into logical deduction rules.

Functional Parameters

A **Functional Parameter** defines a mandatory information for the Reasoning Request (for instance the "ending point" in a *Travel Planner* scenario). A set of **Functional Parameters** (Π) is composed by a finite set (of cardinality n_Π) of individual **Functional Parameter** ($\pi^i; \pi^i \in \Pi \; \forall i \in [1; n_\Pi]$). Each **Functional Parameter** (π^i) is composed of:

- **Functional Parameter Name** (N_{π^i})
- **Functional Parameter Value** (V_{π^i})

i.e. $\pi^i = \langle N_{\pi^i}, V_{\pi^i} \rangle$.

The **Functional Parameter Name** (N_{π^i}) is a string taken from a fixed set of strings ($\Theta_{T,N_{\pi^i}}$) and the **Functional Parameter Value** (V_{π^i}) is specific for each scenario.

The set of **Functional Parameters** (Π) is translated as the concatenation of the translations of each **Functional Parameter** (π^i) that composes it. Each **Functional Parameter** ($\pi^i = \langle N_{\pi^i}, V_{\pi^i} \rangle$) is translated as:

$$parameter(N_{\pi^i}, V_{\pi^i}).$$

The **Functional Parameter Value** can be a single value or a set of possible values. When the **Functional Parameter Value** is a set (e.g. expressed as enumeration of possible values), it is translated into several of the above facts, one for each item in the set.

Functional Constraints

A **Functional Constraint** defines a numerical restriction about a specific aspect of the Reasoning Request. This restriction is “strict” and needs to be fulfilled by each of the answers (otherwise referred to as solutions) offered to the user. A set of **Functional Constraints** (Γ) is composed by a finite set (of cardinality n_Γ) of individual **Functional Constraint** ($\gamma^i; \gamma^i \in \Gamma \; \forall i \in [1; n_\Gamma]$).

Each **Functional Constraint** (γ^i) is composed of:

- **Functional Constraint Name** (N_{γ^i})
- **Functional Constraint Operator** (O_{γ^i})
- **Functional Constraint Value** (V_γ)

i.e. $\gamma^i = \langle N_{\gamma^i}, O_{\gamma^i}, V_\gamma \rangle$.

The **Functional Constraint Name** is a string taken from a fixed set of strings ($\Theta_{T,N_{\gamma^i}}$) and the **Functional Constraint Operator** is an arithmetic operator taken from a fixed set ($\Theta_{O_{\gamma^i}} = \{=, \neq, >, <, \geq, \leq\}$). For each **Functional Constraint Operator** (O_{γ^i}) we denote with $\overline{O_{\gamma^i}}$ its complementary operator. The **Functional Constraint Value** V_γ is an integer number.

The **Functional Constraints** (Γ) is translated as the concatenation of the translations of each **Functional Constraint** (γ^i) that composes it. Each **Functional Constraint** is translated as:

- The “real” constraint:
$$\leftarrow violatedC(N_{\gamma^i}).$$
- A rule to derive if it is violated:
$$violatedC(N_{\gamma^i}) \leftarrow valueOf(N_{\gamma^i}, AV), AV\overline{O_{\gamma^i}}V_{\gamma^i}.$$

where AV is an *ASP* variable.

Functional Preferences

A **Functional Preference** defines a "soft" constraint or priority among the verification of specific aspect of the Reasoning Request. This restriction is "weak" and should be optimized by the *Decision Support* component in order to provide the optimal or most preferred answers to the user.

A set of **Functional Preferences** (Ω) is composed by a finite set (of cardinality n_Ω) of individual **Functional Preference** ($\omega^i; \omega^i \in \Omega \ \forall i \in [1; n_\Omega]$). Each **Functional Preference** (ω^i) is composed of:

- **Functional Preference Order** (O_{ω^i}).
- **Functional Preference Operation** (Opt_{ω^i})
- **Functional Constraint Name** (N_{γ^i}), as defined before.

i.e. $\omega^i = \langle O_{\omega^i}, Opt_{\omega^i}, N_{\gamma^i} \rangle$.

The **Functional Preference Order** (O_{ω^i}) is an integer $\in [1; n_\Omega]$ and the **Functional Preference Operation** (Opt_{ω^i}) is an optimization operator taken from a fixed set ($\Theta_{Opt_{\omega^i}} = \{minimize, maximize\}$).

The **Functional Preferences** (Ω) is translated as the concatenation of the translations of each **Functional Preference** (ω^i) that composes it. Each **Functional Preference** is translated as:

$$\#Opt_{\omega^i}\{AV@O_{\omega^i} : valueOf(N_{\gamma^i}, AV)\}.$$

To allow more flexibility in the logic program each **Functional Preference** (ω^i) could be also translated as (in addition to the previous translation):

$$preference(O_{\omega^i}, Opt_{\omega^i}, N_{\gamma^i}).$$

2.6.3 Use-case Scenarios

In order to demonstrate how *Decision Support* can be used to develop applications for *Smart Cities* and citizens, we have implemented two context-aware use-cases using the live data from the city of Aarhus, Denmark: a *Travel Planner* app and a *Parking Planner* app. In this section, we present the Reasoning Request, logic rules automatically generated from the Reasoning Request, scenario-dependent rules, and

External Modules used in the *Decision Support* process. The implementation of the *Decision Support* for these two scenarios is available at

https://github.com/CityPulse/Decision-Support-and-Contextual-Filtering

Travel Planner Scenario

Tony needs to travel from home to work. Different means of transportation are generally available to him and include walking, biking, car, and public transport. Transportation can be optimized to Tony's preferred travel time, convenience, total cost, environmental impacts, and personal health. Factors that impact this optimization include the conditions of the different transportation modes, including but not limited to road, weather, maintenance works, traffic intensity, people density, pollution, air quality, irregularities in traffic schedules, road tolls, seating availability, accidents, availability of city bikes. Tony will be presented with his ideal route and will be able to select each leg of the journey based on concurrent and projected aggregated conditions. Recalculation of his chosen route(s) can happen if conditions or preferences change, and the provided solution will adapt to any detour of own choice.

In order to provide optimal travel-planning solutions to Tony, *Decision Support* allows him to provide his multi-dimensional requirements and preferences such as air quality, traffic conditions, etc. The Reasoning Request for this scenario has the following main fields:

Type indicating what *Decision Support* module of the *Smart City Framework* is to be used for this application ("TRAVEL-PLANNER" in this case).

User Reference indicating the unique "User ID".

Functional Details specifying possible values of user's requirements and preferences. Tables 2.7, 2.8, and 2.9 show concrete possible values of **Functional Parameters**, **Functional Constraints**, and **Functional Preferences** respectively.

Table 2.7.: Example of **Functional Parameters** for the *Travel Planner* scenario.

Functional Parameters	Name	Value Type	Value
Starting Point	STARTING_POINT	Coordinate	56.17888121 10.15399361
Ending Point	ENDING_POINT	Coordinate	56.15183187 10.15450859
Starting Time/Date	STARTING_DATETIME	Date	2017-01-10T18:25:43.511Z
Transportation Type	TRANSPORTATION_TYPE	Enum	{CAR, WALK, BICYCLE}

The concrete reasoning request is automatically mapped into *ASP* rules (see example rules 1-8 in Listing 2.1) in which: **Functional Parameters** are translated as simple logic facts (rules 1-2); **Functional Constraints** (rules 3-4) are translated as strong constraints, which reduce the solution space by eliminating answers that are

Table 2.8.: Example of **Functional Constraints** for the *Travel Planner* scenario.

Functional Constraints	Name	Operator	Value Type	Value
Travel time less than X	TRAVEL_TIME	$\leqslant$	Duration	15
Distance less than X	DISTANCE	$\leqslant$	Number	1000
Pollution amount less than X	POLLUTION	$\leqslant$	Number	13,5

Table 2.9.: Example of **Functional Preferences** for the *Travel Planner* scenario.

Functional Preferences	Order	Operation	Name
Travel time	1	MINIMIZE	TRAVEL_TIME
Distance	2	MINIMIZE	DISTANCE
Pollution	3	MINIMIZE	POLLUTION

violating any of those constraints. **Functional Preferences** are translated as optimize statements (rules 5-8), which rank the solutions to provide only those that are qualitatively better w.r.t. the optimization statements used. Those rules are combined with the specific scenario-driven rules for the *Travel Planner Decision Support* module (rules 9-13) for reasoning.[38] The *Decision Support* component collects all possible routes from the *Geospatial Data Infrastructure* (*GDI*) component [480] as well as the last snapshot of values of relevant functional properties for those routes which can be produced dynamically by the Data Federation component [241, 242, 480] or retrieved from the *Knowledge Base* (rules 10-12).

```
 1 parameter(‘‘ENDING_POINT’’,‘‘10.1591864 56.1481156’’).
 2 parameter(‘‘STARTING_POINT’’,‘‘10.116919 56.226144’’).
 3 :- violatedConstraint(‘‘POLLUTION’’).
 4 violatedConstraint(‘‘POLLUTION’’) :- valueOf(‘‘POLLUTION’’,
       AV), 135 < AV.
 5 #minimize{AV@1 : valueOf(‘‘TIME’’, AV)}.
 6 preference(1,‘‘MINIMIZE’’,‘‘TIME’’).
 7 #minimize{AV@2 : valueOf(‘‘DISTANCE’’, AV)}
 8 preference(2,‘‘MINIMIZE’’,‘‘DISTANCE’’).
 9 inputGetRoutes(SP, EP, V, 5) :- parameter(‘‘STARTING_POINT’
     ’, SP), parameter(‘‘ENDING_POINT’’, EP), routeCostMode(V
     ).
10 route(@getRoutes(SP, EP, V, N)) :- inputGetRoutes(SP, EP, V
     , N).
11 routeData(@getRoutesData(SP, EP, V, N)) :- inputGetRoutes(
     SP, EP, V, N).
12 maxPollution(@getMaxPollution(RouteID)) :- selected(RouteID
     ).
13 1 <= {selected(RouteID) : route((RouteID, _, _))} <= 1.
```

Listing 2.1: A snapshot of logic rules for *Travel Planner* scenario.

The External Modules used for this scenario are:

[38] A full set of rules is available at https://github.com/CityPulse/Decision-Support-and-Contextual-Filtering/tree/master/res/dss

GDI which enables calculation of different distance measures and allows enhanced information interpolation to increase reliability. Furthermore, an enhanced routing system enables multidimensional weighting on path, e.g., depending on distance, duration, pollution, events or combined metrics. Thereby, it is possible to avoid certain areas or block partial routes for specific applications.

Data Federation which is responsible for processing the application request for *IoT* streams and automatically discover the most relevant *data streams* after catering for individual requirements and preferences for a particular user request. It is also responsible for automatically integrating heterogeneous *data streams* and perform *Complex Event Processing* over the integrated stream.

Both the *GDI* and the Data Federation components are part of the ***CityPulse*** framework. Their implementation are available at

`https://github.com/CityPulse`

Parking Planner Scenario

Frank 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 Frank has to drive around for a long time looking for parking spots. This is very time-consuming for him and results in negative environmental impact (pollution, noise). By using multiple input sources of information the application can provide Frank with a certain degree of probability of finding a parking spot in different locations, thus reducing the driving time (and related CO2 emissions). By knowing the number of cars on the road at any time, the application can help Frank avoiding congested hot spots by being rerouted towards different paths to even out the distribution.

Decision Support aims to provide optimal available parking slots nearby Frank's point of interest while taking into account his constraints and preferences. The Reasoning Request for this scenario has the following main fields:

Type indicating what *Decision Support* module is to be used for this application ("PARKING-SPACE" in this case).

User Reference indicating the unique "User ID".

Functional Details specifying possible values of user's requirements and preferences. Tables 2.10, 2.11, and 2.12 show concrete possible values of **Functional Parameters**, **Functional Constraints**, and **Functional Preferences** respectively.

Similar to the *Travel Planner* scenario, the concrete reasoning request is automatically mapped into *ASP* rules (see example rules 1-8 in Listing 2.2), and combined

Table 2.10.: Example of **Functional Parameters** for the *Parking Planner* scenario.

Functional Parameters	Name	Value Type	Value
Starting Point	STARTING_POINT	Coordinate	56.17888121 10.15399361
Point Of Interest	POINT_OF_INTEREST	Coordinate	56.15183187 10.15450859
Starting Time/Date	STARTING_DATETIME	Date	2017-01-10T18:25:43.511Z
Distance Range	DISTANCE_RANGE	Number	1000
Time Of Stay	TIME_OF_STAY	Duration	100

Table 2.11.: Example of **Functional Constraints** for the *Parking Planner* scenario.

Functional Constraints	Name	Operator	Value Type	Value
Walking distance less than X	DISTANCE	$\leqslant$	Number	1000
Cost less than X	COST	$\leqslant$	Number	50

with the specific scenario-driven rules for the Parking *Decision Support* module (rules 9-13). The *Decision Support* component collects all possible parking slots with their cost from the *Knowledge Base* (these parking slots are in 'DISTANCE_-RANGE' which is checked by resorting to the *GDI* component) as well as the last snapshot of availability of parking slots which can be produced dynamically by the Data Federation component (rules 9-11). Similarly to the *Travel Planner* scenario, the External Modules used for this scenario are *GDI* and Data Federation.

```
1  parameter(‘‘DISTANCE_RANGE’’,1000).
2  parameter(‘‘POINT_OF_INTEREST’’,‘‘10.116919 56.226144’’).
3  :- violatedConstraint(‘‘COST’’).
4  violatedConstraint(‘‘COST’’) :- valueOf(‘‘COST’’, AV), 100
       < AV.
5  preference(1,‘‘MINIMIZE’’,‘‘DISTANCE’’).
6  #minimize{AV@1 : valueOf(‘‘DISTANCE’’, AV)}.
7  preference(2,‘‘MINIMIZE’’,‘‘COST’’).
8  #minimize{AV@2 : valueOf(‘‘COST’’, AV)}.
9  parkingSpace(@getParkingSpaces(POI, DR)) :- parameter(‘‘
       POINT_OF_INTEREST’’, POI), parameter(‘‘DISTANCE_RANGE’’,
        DR).
10 availability(@getAvailability(ParkingID)) :- selected(
       ParkingID).
11 totalCost(@getTotalCost(ParkingID, ToS)) :- selected(
       ParkingID), parameter(‘‘TIME_OF_STAY’’, ToS).
12 1 <= {selected(ParkingID) : parkingSpace((ParkingID,
       Position,Distance))} <= 1.
13 distance(Distance) :- selected(ParkingID), parkingSpace((
       ParkingID,Pos,Distance)).
```

Listing 2.2: A snapshot of logic rules for *Parking Planner* scenario.

Table 2.12.: Example of **Functional Preferences** for the *Parking Planner* scenario.

Functional Preferences	Order	Operation	Name
Walking Distance	1	MINIMIZE	DISTANCE
Cost	2	MINIMIZE	COST

2.6.4 Discussion

In this work, we described how we designed and implemented a user-centric declarative *Decision Support* component by leveraging the expressivity and fully declarative nature of *ASP*. To achieve this, we define a representation method that allows a user to specify constraints and preferences, and we propose an automatic mapping to convert user requests into logical rules. In order to demonstrate the efficiency in term of re-usability and declarativity of this approach, we showcase the implementation of the *Decision Support* component for two *Smart City Applications*: the *Travel Planner* and the *Parking Planner* applications. Our rule-based *Decision Support* component can be used in various application scenarios by: (i) identifying values of the parameters to be constrained or optimized in the Reasoning Request, (ii) describing the domain-specific rules for the decision task (or using existing reasoning modules available in the ***CityPulse*** Framework), (iii) plugging in the proper External Modules to compute subtasks when needed for scalability. As a result of our proposal, we can achieve user-centricity in the *Decision Support* process in order to provide optimal solutions that better target user needs.

Wrap-up

In this chapter we reported about an approach to perform complex reasoning on web *data streams* maintaining scalability and we defined some key concepts that allow computing the processing time of *RSP* systems. In addition, we formalized the specification of User-Centric *Decision Support* requests and we described how to map them to *Logic Programming*.
The results of these investigations have been applied to the ***CityPulse*** Project.
It is worth noticing that some of the investigations reported in this chapter have been conducted during the visit in the *Reasoning and Querying Unit* (lead by *Dr. Alessandra Mileo*) of the *INSIGHT Centre for Data Analytics*[39] (formerly *DERI*[40]) at the *National University of Ireland (NUI) Galway*[41].

As noted in this chapter the amount of data along with the speed of the data are important factors. They are central aspects in the *Big Data* field, in the next chapter we report about this related area, highlighting some analysis we performed on specific topics, such as *Data Wrangling*.

[39] https://www.insight-centre.org
[40] http://www.deri.ie
[41] http://www.nuigalway.ie

Big Data

3

To clarify, add detail.

— **Edward R. Tufte**
(Envisioning Information)

Summary of Chapter 3

Answering “complex” queries over large amount of data requires a perfect balance between expressivity and efficiency. However, the commonly used solutions are very fast and scalable but, as such, they cannot handle advanced reasoning tasks.

Logic Programming can help, thanks to the wide variety of languages that are available, although not all of them are good for all reasoning tasks or for the same category of programs. Therefore, reliable techniques should be developed, in order to estimate how difficult an activity is and which is the most appropriate formalism and/or evaluation engine to tackle it.

In this chapter, after a brief introduction on the well-known topic of *Big Data*, we focus (in Sections 3.3 and 3.4) on specific problems we worked on, concerning the use of logic-based techniques when a huge quantity of data is involved, and we discuss the optimizations that can be implemented to allow a more efficient management of resources.

Chapter Outline

3.1 Definition, Motivation and Challenges

3.1.1 What *Big Data* is and why it is important[1]

Over the past 20 years, data has increased on a large scale in various fields. According to a report from *International Data Corporation* (*IDC*), in 2011, the overall created and copied data volume in the world was 1.8ZB ($\approx 10^{21}$ B), which increased by nearly nine times within five years [240]. This figure will double at least every other two years in the near future.

Under the explosive increase of global data, the term of *Big Data* is mainly used to describe enormous datasets. Compared with traditional datasets, *Big Data* typically includes masses of unstructured data that need more real-time analysis. In addition, *Big Data* also brings about new opportunities for discovering new values, helps us to gain an in-depth understanding of the hidden values, and also incurs new challenges, e.g., how to effectively organize and manage such datasets.

Recently, industries become interested in the high potential of *Big Data*, and many government agencies announced major plans to accelerate *Big Data* research and applications. In addition, issues on *Big Data* are often covered in public media and two premier scientific journals, Nature and Science, also opened special columns to discuss the challenges and impacts of *Big Data*[2].

Big Data is an abstract concept. Apart from masses of data, it also has some other features, which determine the difference between itself and “massive data” or “very big data”.

At present, although the importance of *Big Data* has been generally recognized, people still have different opinions on its definition. In general, *Big Data* shall mean the datasets that could not be perceived, acquired, managed, and processed by traditional IT and software/hardware tools within a tolerable time. Because of different concerns, scientific and technological enterprises, research scholars, data analysts, and technical practitioners have different definitions of *Big Data*.

In 2011, an *IDC* report defined *Big Data* technologies as

> A <u>new</u> generation of technologies and architectures, designed to <u>economically</u> extract <u>value</u> from very large <u>volumes</u> of a wide <u>variety</u> of data, by enabling high-<u>velocity</u> capture, discovery, and/or analysis. [240]

[1]Preliminary definitions adapted from [147, 351]
[2]Nature Special Big Data and Science Special Online Collection: Dealing with Data

With this definition, characteristics of *Big Data* may be summarized as four *V*s, i.e., *Volume* (great volume, i.e., the amount, size, and scale of the data), *Variety* (various modalities, i.e., the structural variation of a dataset and of the data types that it contains as well as the variety in what it represents, its semantic interpretation and its sources), *Velocity* (rapid generation and analysis, i.e., the speed at which data are generated as well as the rate at which they must be analysed), and *Value* (huge value but very low density). Commonly *Value* is defined as the desired outcome of *Big Data* processing [327] and not as defining characteristics of *Big Data* itself and it is substituted by *Veracity*, which refers not only to the reliability of the data forming a dataset, but also, as IBM has described, to the inherent unreliability of data sources [238]. Such 4'Vs definition was widely recognized since it highlights the meaning and necessity of *Big Data*, i.e., exploring the huge hidden values. This definition indicates the most critical problem in *Big Data*, which is how to discover values from datasets with an enormous scale, various types, and rapid generation.

O'Reilly experts identified in [397] the following reasons why *Big Data* matters:

- The world is increasingly awash in sensors that create more data – both explicit sensors like point-of-sales scanners and *RFID* tags, and implicit sensors like cell phones with *GPS* and search activity.
- Harnessing both explicit and implicit human contribution leads to far more profound and powerful insights than traditional data analysis alone.
- Competitive advantage comes from capturing data more quickly, and building systems to respond automatically to that data.
- The practice of sensing, processing, and responding is arguably the hallmark of living things. We are now starting to build computers that work the same way.
- As our aggregate behaviour is measured and monitored, it becomes feedback that improves the overall intelligence of the system.
- With more data becoming publicly available, from the Web, from public data sharing sites, from increasingly transparent government sources, from science organizations, from data analysis contests, and so on, there are more opportunities for mashing data together and open sourcing analysis. Bringing disparate data sources together can provide context and deeper insights than what is available from the data in any one organization.

These issues pose many important scientific questions on which a high number of researchers is currently working on.

Due to the popularity of the topic, we do not describe the details here.
However, it is worth noticing that also the *Logic Programming* community worked

on *Big Data* analysis and management, even if they often referred under different names, for instance [94].
It is also important to highlight that the *Big Data* topic is strictly connected with *Stream Reasoning* and *Internet of Things* (*IoT*).

3.1.2 Challenges in *Big Data*[3]

The sharply increasing data deluge in the *Big Data* era brings huge challenges on data acquisition, storage, management and analysis. At each step, there is work to be done and there are challenges with *Big Data*.

The first step is data acquisition. Some data sources, such as sensor networks, can produce staggering amounts of raw data. Much of this data is of no interest, and it can be filtered and compressed by orders of magnitude. One challenge is to define these filters in such a way that they do not discard useful information. The second big challenge is to automatically generate the right metadata to describe what data is recorded and how it is recorded and measured. This metadata is likely to be crucial to downstream analysis.

Frequently, the information collected will not be in a format ready for analysis. The second step is an information extraction process that pulls out the required information from the underlying sources and expresses it in a structured form suitable for analysis. A news report will get reduced to a concrete structure, such as a set of tuples, or even a single class label, to facilitate analysis. Furthermore, we are used to thinking of *Big Data* as always telling us the truth, but this is actually far from reality. We have to deal with erroneous data: some news reports are inaccurate.

Data analysis is considerably more challenging than simply locating, identifying, understanding, and citing data. For effective large-scale analysis, all of this has to happen in a completely automated manner. This requires differences in data structure and semantics to be expressed in forms that are computer understandable, and then robotically resolvable. Even for simpler analyses that depend on only one data set, there remains an important question of suitable data storage design. Usually, there will be many alternative ways in which to store the same information. Certain designs will have advantages over others for certain purposes, and, possibly, drawbacks for other purposes.

Mining requires integrated, cleaned, trustworthy, and efficiently accessible data, declarative query and mining interfaces, scalable mining algorithms, and *Big Data* computing environments. A problem with current *Big Data* analysis is the lack of co-

[3]Preliminary definitions adapted from [94, 147, 351, 352]

ordination between *DataBase* systems, which host the data and provide *SQL* querying, with analytics packages that perform various forms of non-*SQL* processing, such as data mining and statistical analyses. Today's analysts are impeded by a tedious process of exporting data from the *DataBase*, performing a non-*SQL* process and bringing the data back.

Having the ability to analyse *Big Data* is of limited value if users cannot understand the analysis. Ultimately, a decision-maker, provided with the result of the analysis, has to interpret these results. Usually, this involves examining all the assumptions made and retracing the analysis. Furthermore, as described above, there are many possible sources of error: computer systems can have bugs, models almost always have assumptions, and results can be based on erroneous data. For all of these reasons, users will try to understand, and verify, the results produced by the computer. The computer system must make it easy for her to do so by providing supplementary information that explains how each result was derived, and based on precisely what inputs.

In short, there is a multi-step pipeline required to extract value from data. Heterogeneity, incompleteness, scale, timeliness, privacy and process complexity give rise to challenges at all phases of the pipeline. Furthermore, this pipeline is not a simple linear flow – rather there are frequent loops back as downstream steps suggest changes to upstream steps.

However, the challenges do not come only from this pipeline of steps but also from real-world constraints and requirements; in [147] are listed the following obstacles in the development of *Big Data* applications (taken from [6, 146, 352]):

Data representation

Many datasets have certain levels of heterogeneity in type, structure, semantics, organization, granularity, and accessibility. Data representation aims to make data more meaningful for computer analysis and user interpretation. Nevertheless, an improper data representation will reduce the value of the original data and may even obstruct effective data analysis. Efficient data representation shall reflect data structure, class, and type, as well as integrated technologies, so as to enable efficient operations on different datasets.

Redundancy reduction and data compression

Generally, there is a high level of redundancy in datasets. Redundancy reduction and data compression are effective to reduce the indirect cost of the entire system on the premise that the potential values of the data are not affected.

Data life-cycle management

Compared with the relatively slow advances of storage systems, pervasive sensing and computing are generating data at unprecedented rates and scales. We are confronted with a lot of pressing challenges, one of which is that the current storage system could not support such massive data. Generally speaking, values hidden in *Big Data* depend on data freshness. Therefore, a data importance principle related to the analytical value should be developed to decide which data shall be stored and which data shall be discarded.

Analytical mechanism

The analytical system of *Big Data* shall process masses of heterogeneous data within a limited time. However, traditional *RDBMS*s are strictly designed with a lack of scalability and expandability, which could not meet the performance requirements. Non-relational *DataBases* have shown their unique advantages in the processing of unstructured data and started to become mainstream in *Big Data* analysis. Even so, there are still some problems of non-relational *DataBases* in their performance and particular applications. More research is needed on the in-memory *DataBase* and sample data based on approximate analysis.

Data confidentiality

Most *Big Data* service providers or owners at present could not effectively maintain and analyse such huge datasets because of their limited capacity. They must rely on professionals or tools to analyse such data, which increase the potential safety risks. Therefore, analysis of *Big Data* may be delivered to a third party for processing only when proper preventive measures are taken to protect such sensitive data, to ensure its safety.

Energy management

The energy consumption of mainframe computing systems has drawn much attention from both economy and environment perspectives. With the increase of data volume and analytical demands, the processing, storage, and transmission of *Big Data* will inevitably consume more and more electric energy. Therefore, system-level power consumption control and management mechanism shall be established for *Big Data* while the expandability and accessibility are ensured.

Expendability and scalability

The analytical system of *Big Data* must support present and future datasets. The analytical algorithm must be able to process increasingly expanding and more complex datasets.

Cooperation

Analysis of *Big Data* is an interdisciplinary research, which requires experts in different fields cooperate to harvest the potential of *Big Data*. A comprehensive *Big Data* network architecture must be established to help scientists and engineers in various fields access different kinds of data and fully utilize their expertise, so as to cooperate to complete the analytical objectives.

Furthermore, each of the *Big Data* dimensions (the Vs) has specific challenges that should be addressed:

Volume

- Processing Performance
- Curse of Modularity
- Class Imbalance
- Curse of Dimensionality
- Feature Engineering
- Non-Linearity
- Bonferonni's Principle
- Variance and Bias

Velocity

- Data Availability
- Real-Time Processing/Streaming
- Concept Drift
- Independent and Identically Distributed Random Variables

Variety

- Data Locality
- Data Heterogeneity
- Dirty and Noisy Data

Veracity

- Data Provenance
- Data Uncertainty
- Dirty and Noisy Data

Besides, most the approaches that are currently available, such as the basic *MapReduce* platforms, lacks essential features like iteration (or equivalently, recursion) and, more importantly, "complex" query answering. In this context, *Logic Programming* and its *KR&R* capabilities can be exploited in order to extend and improve the methodologies and the frameworks on the market.

3.1.3 Further remarks

For more information about *Big Data* see also [365].

3.2 Query Answering over *Big Data*: a case study - the VADA project

In this and in the next section we first introduce some interesting case studies, and then we report about some results obtained using logic-based techniques for *Big Data* in the context of *Data Wrangling*.
The investigations reported in the following section have been conducted during the visit at the *Department of Computer Science* (lead by *Professor Georg Gottlob F.R.S.*) at the *University of Oxford*[4].

3.2.1 VADA: *value-Added DAta systems*[5]

VADA brings together three UK research groups with proven international leadership in *DataBases* and information systems, yet with complementary areas of specialism, to develop principles, techniques and architectures for adding value to data. The applicants have outstanding track records against many criteria and include 2 Fellows of the Royal Society (FRS), 4 Fellows of the ACM (FACM), 2 recipients of Royal Society Wolfson Research Merit Awards, 3 Fellows of the Royal Society of Edinburgh (FRSE) and a winner of the BCS Roger Needham Award; their current grant portfolio exceeds £9 million (£3.2M at Edinburgh, £3.9M at Oxford, £2.4M at Manchester) from national funding bodies, EU and industry; and they have 130 papers with over 100 citations. The principal investigator is *Prof. Georg Gottlob* from Oxford, and the lead CIs at Edinburgh and Manchester are *Prof. Leonid Libkin* and *Prof. Norman Paton*, respectively.

The VADA consortium consists of the three leading *DataBase* research groups in the UK and 11 non-academic partners from 4 different countries (shown, respectively, in Figure 3.1 and in Figure 3.2). All teams contribute an outstanding and unique mix of complementary expertise and skills, from foundations to applications, that is well suited for this programme.

Research Programme description

Data is everywhere, generated by increasing numbers of applications, devices and users, with few or no guarantees on the format, semantics, and quality. The economic potential of data-driven innovation is enormous, estimated to reach as much as £40B in 2017, by the Centre for Economics and Business Research. To realise this potential, and to provide meaningful data analyses, data scientists must first spend

[4]http://www.ox.ac.uk
[5]Preliminary definitions adapted from [338], and VADA website and documentation

Figure 3.1.: Universities of the VADA consortium.

Figure 3.2.: Project partners of the VADA consortium.

a significant portion of their time (estimated as 50% to 80%) on "*Data Wrangling*" – the process of collection, reorganising, and cleaning data.

This heavy toll is due to what is referred as the four *V*'s of *Big Data*: *Volume* – the scale of the data, *Velocity* – speed of change, *Variety* – different forms of data, and *Veracity* – uncertainty of data. There is an urgent need to provide data scientists with a new generation of tools that will unlock the potential of data assets and significantly reduce the *Data Wrangling* component. As many traditional tools are no longer applicable in the 4 V's environment, a radical paradigm shift is required. The proposal aims at achieving this paradigm shift by adding value to data, by handling data management tasks in an environment that is fully aware of data and user contexts, and by closely integrating key data management tasks in a way not yet attempted, but desperately needed by many innovative companies in today's data-driven economy.

The VADA[6] research programme will define principles and solutions for V*alue-*A*dded* DA*ta systems*, which support users in discovering, extracting, integrating, accessing and interpreting the data of relevance to their questions. In so doing, it uses the context of the user, e.g., requirements in terms of the trade-off between completeness and correctness, and the data context, e.g., its availability, cost, provenance and quality. The user context characterises not only what data is relevant, but also the properties it must exhibit to be fit for purpose. Adding value to data then involves the best effort provision of data to users, along with comprehensive information on the quality and origin of the data provided. Users can provide feedback on

[6]EPSRC Project EP/M025268/1. `http://vada.org.uk`

the results obtained, enabling changes to all data management tasks, and thus a continuous improvement in the user experience.

Establishing the principles behind VA*lue-Added* DA*ta systems* requires a revolutionary approach to data management, informed by interlinked research in data extraction, data integration, data quality, provenance, query answering, and reasoning. This will enable each of these areas to benefit from synergies with the others. Research has developed focused results within such sub-disciplines; VADA develops these specialisms in ways that both transform the techniques within the sub-disciplines and enable the development of architectures that bring them together to add value to data.

The commercial importance of the research area has been widely recognised. The VADA programme brings together university researchers with commercial partners who are in desperate need of a new generation of data management tools. They will be contributing to the programme by funding research staff and students, providing substantial amounts of staff time for research collaborations, supporting internships, hosting visitors, contributing challenging real-life case studies, sharing experiences, and participating in technical meetings. These partners are both developers of data management technologies (LogicBlox, Microsoft, Neo) and data user organisations in healthcare (The Christie), e-commerce (LambdaTek, PricePanda), finance (AllianceBernstein), social networks (Facebook), security (Horus), smart cities (FutureEverything), and telecommunications (Huawei).

The VADA Architecture

The VA*lue-Added* DA*ta systems* (VADA) architecture is illustrated in Figure 3.3. Briefly, the key components of the VADA project are:

Transducers

The *Transducers* are a collection of components that represent the functionalities within the wrangling process; a *transducer* [7] is a software component with input and output dependencies defined as *Datalog* queries over the *Knowledge Base* and/or the state of the transducer. The input dependencies, for example, may initiate the evaluation of a transducer when information becomes available on which it can act. For example, a *mapping generation* transducer may start to evaluate when matches have been created between source and target schemas, or a *data fusion* transducer may start to evaluate when duplicates have been detected. The architecture is not tied to a specific or fixed set of transducers.

[7] The notion of *transducer* is inspired by earlier work on *relational transducers* [5], although the languages used are not formally equivalent.

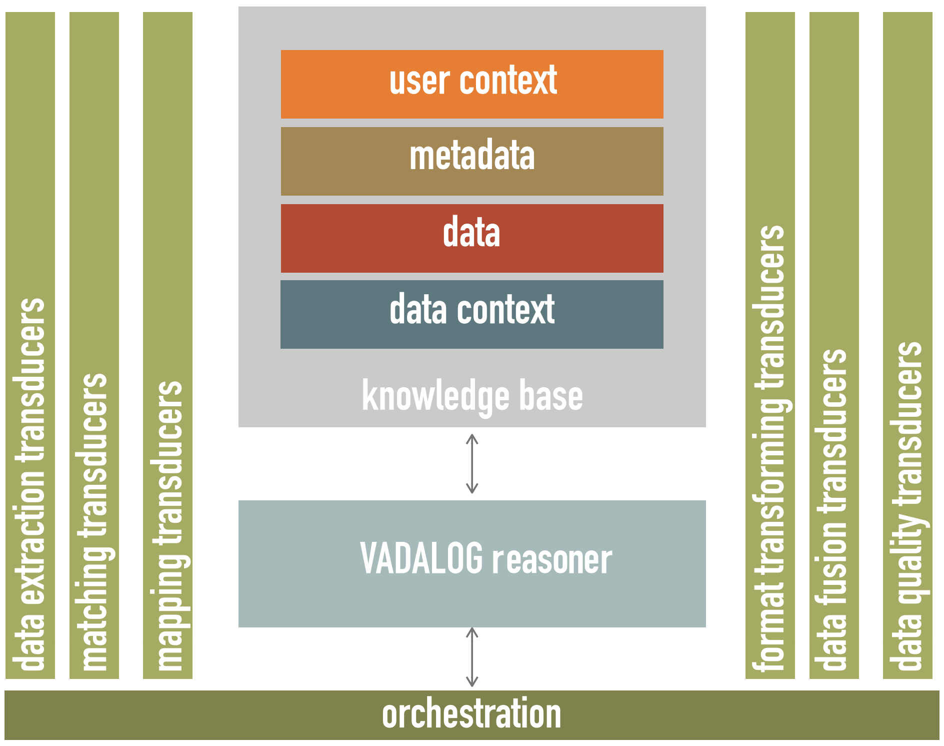


Figure 3.3.: The VADA Architecture. From "Data Integration: a system-based view – Dr. Alvaro A. A. Fernandes".

Knowledge Base

The *Knowledge Base* is a repository for representing the data of relevance to the *Data Wrangling* process; this includes information about the requirements of the user (*user context*), the application domain (*data context*), and both data and metadata created and used by the transducers that participate in the wrangling process.

VADALOG Reasoner

The VADALOG *Reasoner* supports reasoning over the *Knowledge Base* using VADALOG [231], a new member of the *Datalog*$^{\pm}$ family of languages [119]; VADALOG plays several roles in the architecture, including specifying transducer dependencies, coordinating the orchestration of the transducers, and representing schema mappings.

3.3 Feature-based Engine Selection for VADALOG program

In the context of *Data Wrangling*, and specifically for VADALOG programs, we investigated about the introduction of a multi-engine approach that we present next.

Data Wrangling has been recognised as a recurring feature of *Big Data* life cycles.[8] It has been defined as:

> a process of iterative data exploration and transformation that enables analysis. [324]

Data Wrangling is a complex task that requires powerful approaches in order to be tackled effectively. As highlighted by several earlier works [75, 227–229] various forms of reasoning are required in order to fulfil all the requirements of the different *Data Wrangling* tasks and ontological reasoning is a key aspect in this context.

In [230] the authors analysed some problems and opportunities in the *Big Data* context for the *Data Wrangling* process. Among them, they highlight the need of the support for *Knowledge Representation and Reasoning* solutions able to deal with the diverse and uncertain working data that is of relevance to the *Data Wrangling* process and some fundamental problems related with data querying, such as intractability, scalability indexing, approximations, etc.

As mentioned before, to add value to data in ways that accommodate diverse user and data contexts, the VADA project must support iterative improvement within each component on the basis of insights from users, external sources and other components. In order achieve this goal a common logical framework, known as VADALOG, for uniformly expressing most rules and reasoning mechanisms used by the different components have been developed. The main component responsible for this is the VADALOG *Reasoner* where the reasoning over the *Knowledge Base* is provided using the VADALOG language.[9]

[8]From [230]
[9]From the VADA proposal

3.3.1 Preliminaries

The VADALOG language[10]

The need to share data and knowledge between components of a *Data Wrangling* system makes it clear that a common language is needed to express such knowledge, and enable reasoning over it. The authors in [231] expressed their vision of a *lingua franca for Data Wrangling*. Such a language should provide a uniform way to address the different needs in the *Data Wrangling* process:

- expressing knowledge in a shared knowledge-base
- reasoning about data and transformation of data within the components
- specifying the workflow between the components

A key challenge to such a language is large volume of data on the one hand, and requirements for highly expressive reasoning on the other hand. Clearly, meeting both requirements at the same time is hard. Yet there is a full spectrum of possibilities in between:

- small volume of data: complex reasoning
- large volume of data: simple processing
- very large volume of data: parallel processing

The challenge of offering both expressiveness and scalability in a single system poses particular design challenges to a language for *Data Wrangling*. A single monolithic – highly expressive – language is not enough to meet the requirement of scalability in the presence of *Big Data*.

The language should address these features by providing a uniform view of data (independent of its source) while supporting the components by being suited to data extraction, integration and exchange. At the same time, it must allow for efficient processing and scalability when confronted with *Big Data*. One of the best-established languages in the data management community for knowledge-based reasoning is *Datalog*. Over the years, it has been studied in great detail and extended in various ways [66], see Section 1.2 for more details. The authors of [231] recognized also that the family of languages often called *Datalog*$^{\pm}$ seeks to add to *Datalog*'s expressive power, yet not by sacrificing efficiency and scalability and therefore they decided to use it as base for the VADALOG language. They have not provided a specific syntax or semantics of the VADALOG language but they relied on the one of *Datalog*$^{\pm}$.

[10]Preliminary definitions adapted from [231]

VADALOG Design Principles In the same paper, the authors identified the following design principles for their proposed language:

Solid Foundation: *Datalog*

The language is based on *Datalog*, extended by features that are well-known in the data management community: in particular existential quantification (as in *TGDs*, existential rules or *Datalog*$^{\pm}$) as well as numerous other features motivated by the theoretical and practical needs of *Data Wrangling*. Having *Datalog* at its foundation gives VADALOG a well-understood core that has been the topic of research for many years now.

Family of Languages

One single, all-encompassing language cannot meet at the same time the goal of being highly expressive as well as having low computational complexity. VADALOG, therefore, consists of *profiles* of the language (in the same meaning as the profiles of languages such as *OWL*), each providing a specific subset suited to a particular purpose. This allows simple computations to be efficiently and scalably executed over *Big Data*, while at the same time allowing complex reasoning over a smaller *Knowledge Base*.

Combining Strengths

There exist a number of powerful knowledge-based systems, *DataBase* systems, and systems that are able to deal with *Big Data* that often combine the expertise of large groups of researchers and engineers. The language is thus designed with the intent of making use of systems specifically suited for particular tasks. For example, if a VADALOG programme is formulated in a profile of the language that is particularly suited to an existing knowledge-based system, then this system is used as a backend-engine. This design choice does not come for free – many interesting research challenges have to be addressed to deal with multiple engines working in a unified system.

Handling Volume

Volume may be coped with by using an engine that is suited for handling huge amounts of data. Yet this is not always the most efficient approach. The language contains as "first-class citizens" the support for partitioning the data into *dataspheres* which may be parametrized using domain-dependent or data-dependent parameters. This language-design principle of being able to handle volume by allowing clever partitioning of the data, combined with using engines that can handle big amounts of data when necessary, allows VADALOG to efficiently deal with huge amounts of data.

Modularity

Reasoning and transformation tasks are organized into self-contained modules – based on the concept of a data *transducer*, which receives data from different dataspheres and produces data in different dataspheres. Such transducers modularly encapsulate their dependencies (which dataspheres they require to be present) and their guards (what conditions must be met to be executed). Defining all of these parts of the transducer is done using VADALOG in a single, maintainable module for each such transducer.

Dynamic Orchestration

Defining single modules – transducers – is only one part of a *Data Wrangling* system. A key part of such a system is that all its components are able to share data and knowledge between them and, importantly, react to such knowledge by dynamically selecting which next steps to take. For example, as a result of quality analysis, the system may choose to redo data extraction with different background knowledge, or adding a data source. A key part of VADALOG is thus a profile for specifying such *transducer networks* that dynamically orchestrate components of the *Data Wrangling* system.

Extensibility

A rule-based language based on *Datalog* is clearly suited for knowledge-based reasoning tasks. Yet, while a *Data Wrangling* system has reasoning-intensive tasks, it also has tasks which are better suited to be implemented in other languages. To harness components implemented in other languages, VADALOG allows extensibility at a number of levels: at the transducer level, which gives the components wide-ranging freedom in how to approach its task (such as an existing component analysing data quality); at the level of *actions*, which are middle-scale tasks (such as navigating web pages), and at the level of *external functions*, which can add small-scale functionality not supported in the language (such as non-supported string or number functions).

The VADALOG *Reasoner* component[11]

VADALOG and its reasoning facilities must address a wide range of desiderata, including the ability to represent and reason about data extractions, data integration views, and data exchange between components, while accommodating uncertain knowledge and user preferences. As mentioned before, VADALOG will have unprecedented functionality, specifically addressing the *Knowledge Representation and Reasoning* requirements of data-intensive applications.

Knowledge-based reasoning is the common foundation for the VADA system – a component that manages data and context information, about data or user, and reasons over these to derive new insights. The main idea is to create a uniform framework for data and rule-based *Knowledge Representation* in VADA, used for information exchange between its components, as well as for intelligent decision-making, e.g., to select relevant sources based on user and data context and on knowledge about the

[11]Preliminary definitions adapted from the VADA proposal

sources and the type of data. Conclusions drawn via automated reasoning will then cause the framework to appropriate, modify data and update the *Knowledge Base*. This framework will be designed around the VADALOG reasoning language.

Necessity of a uniform formalism for data representation and reasoning. Knowledge-based reasoning provides a strong foundation for managing diverse types of data, including data and user contexts. It also serves as a common base for deriving additional information from existing data via rules. This facility is available to VADA's user, as well as to other VADA components, that will all make extensive use of this facility. Many parts of the joint data and *Knowledge Base*, e.g., the data dictionary, will be shared among several components. Given that many components produce or manage data and use knowledge for reasoning, VADA requires a uniform formalism for data and *Knowledge Representation*, complemented by an appropriate reasoning mechanism. Different applications may require different storage models. In particular, data and rules may be divided over main memory, standard relational *DataBase*s, *Data Warehouses*, *Cloud Storage*, and so on. The uniform formalism for data and *Knowledge Representation* we strive for in this work package shall be independent of the storage model. However, the concrete reasoning mechanisms and query-answering algorithms that are based on this formalism will have to take different possible forms of storage into account.

Desiderata. The following are the main desiderata for the data representation and reasoning component in VADA:

(D1) It should provide a single, uniform view of factual data, rule-based knowledge, data, and user context, whether stored in an internal data or *Knowledge Base*, or in an existing external *DataBase* or data-warehouse, or extracted from the Web or other sources, or added via reasoning.

(D2) It should be suited for data integration (including view management and querying [359, 395], as well as discovery of schema mappings [291]) and data exchange between components.

(D3) It should be suited for rules used to reason about websites and the data appearing on them, as needed for data extraction, including provenance and confidence of the data to be extracted.

(D4) It should provide means for updating data and, where necessary, rules.

(D5) It should be suited for advanced logical reasoning tasks, such as closed-world reasoning (e.g., if it is not known that a data item is corrupted, assume it is correct), and transitive closure (e.g., recover the origin of a data item by following a chain of local provenance pairs backward), as well as ontological reasoning and *Ontology-Based Data Access* (*OBDA*), that enriches a *DataBase* by an ontology, thus providing a high-level conceptual context for the data [138]. All of these are essential to effectively derive conclusions from existing data.

(D6) It should be suited for uncertain knowledge, as well as inconsistencies that naturally result when data is extracted and integrated from many sources, including web-sources, and provide approximate and probabilistic reasoning to handle such data.

(D7) It should be suited for user preferences and other user contexts, which will allow for personalized query answering, where answers are tailored to the needs of specific users.

(D8) It should lend itself to an efficient and scalable inference mechanism for executing the rules and reasoning on top of data and knowledge.

Query Answering. An important part of the Reasoning component is Query Answering. QA under ontologies is becoming increasingly important, and thus our techniques will be tightly integrated with VADALOG-expressible ontologies in order to reason about the quality of approximate and imprecise answers to queries.

The very notion of query answering is undergoing a fundamental transformation. We are used to having a fixed data set, and a standard data management tool with a clear and well-defined semantics of query answers. This vision is no longer true though for two reasons. One is the sheer amount of data. It is simply impossible to get precise answers to queries – in many cases, we cannot even feasibly scan the data. The other is imprecision and the proliferation of imperfect information, due to both extracting from multiple sources (often not controlled by us) and integrating; despite our best efforts to clean the data, those factors will still remain. We thus need to deliver a new way of providing query answers that takes into account both the data context (amount of data, imperfection) and the user context (the notion of acceptable query answers) and delivers fast algorithms for providing meaningful and helpful answers to users' queries.

In providing query answering facilities for VADA, there are two distinct challenges, corresponding to two of the V's – Volume and Veracity. These depend on where the need for querying occurs in relation to other tasks.

- To extract or integrate data, we have to deal with massive data sets, often not under our control. The key issues there are handling the *scale of data* and the ability to produce useful *approximate answers*.
- To query data that has already been extracted from different sources, the main challenge is handling *uncertainty and imprecision*. Despite our best efforts to clean it and resolve inconsistencies in the integration process, uncertainties are bound to persist given the heterogeneous nature of the data we start with.

We describe next the approach we are designing and developing.

3.3.2 A multi-engine approach

As explained in Section 1.2.3 the *Datalog*$^{\pm}$ family contains several classes of QA-decidable languages, each of them with its own strengths and weaknesses. Moreover, in Section 1.2.4, we described various *Datalog*$^{\pm}$ solvers and we highlighted that each of them is able to treat a specific fragment of *Datalog*$^{\pm}$ and it is optimized for it. Furthermore, as described in the previous section, the VADALOG language was conceived in order to not be restricted to a specific class of *Datalog*$^{\pm}$ languages and actually can use different fragments and have several profiles depending on the data and the user contexts.[12]

Given all these considerations, and in accordance with the idea established in the VADA Project proposal of *"selecting the most suitable pool of execution back-ends to be used to answer* VADALOG *queries and translating them into the suitable concrete query languages depending on the back-end"*, we decided to design a multi-engine approach that could satisfy all the requirement of the VADALOG Reasoner mentioned above.

The main idea is to identify, collect and integrate into a modular and scalable framework various systems that can solve all sorts of VADALOG programs. Since programs in the VADALOG language can include a wide variety of operators and therefore their complexity can differ greatly, the range of solvers we can use to perform Query Answering over them is really broad, from those specifically tailored to perform the *chase* on *TGD*s and/or *EGD*s to those that can be adapted with specific techniques (such as Skolemization) to support *Datalog*$^{\pm}$ programs or part of them.

In addition, we want to develop an *AI*-based strategy to select the best solver to choose among the available ones based on the properties and the features of the Data (*Dataspheres*) and of the Program (*Transducer*), and on the characteristics of the available engines.

[12]It is worth mentioning that at the moment of writing the VADA research group is also focusing on a specific fragment as logical core of the VADALOG language. [86]

Programs and Data Features

Many different studies have been carried out, in the *Logic Programming* community, in order to identify "features" of logic programs and often these are used to build multi-engine approaches. In [257, 262, 303, 400–406, 449], we identified the following features that could be useful in a multi-engine approach: (here divided in some groups adapted from [449])

Problem size features [13]

- Number of Rules r
- Number of Atoms a
- Ratios r/a, $(r/a)^2$, $(r/a)^3$
- Ratios reciprocal a/r, $(a/r)^2$, $(a/r)^3$
- Number of Functions f
- Number of Variables v
- Number of Constraints c
- c / v
- Number of Equivalences

Balance features [14]

- Ratio of positive and negative atoms in each body
- Ratio of positive and negative occurrences of each variable
- Fraction of normal rules
- Fraction of constraints
- Fraction of unary, binary and ternary rules

"Proximity to horn" features [15]

- Fraction of horn rules
- Number of occurrences in a Horn clause for each variable

Peculiar features

- Head sizes
- Strongly Connected Components (SCC)
- Head-Cycle Free [88] components
- Presence of queries
- Stratification properties

Graph Features

- Variable nodes degree statistics (mean, variation coefficient, min, max, entropy)
- Rule nodes degree statistics (mean, variation coefficient, min, max, entropy)

Moreover, different operations could be "encoded" in the rules of the Program, such as *(a)* Selection, *(b)* Union, *(c)* Intersection, *(d)* Projection, *(e)* Join, *(f)* Cross Product, and *(g)* Negation.

And from the (labelled) dependency graph of the Program, other peculiar properties can be analysed, for instance *(a)* stratification (acyclic graph, recursion), *(b)* vertex data (adjacent vertices, isolated vertices, etc.), *(c)* structure of the graph (is a tree?, is linear?, etc.), and *(d)* strongly connected components.

[13]This type of features can be considered to give an idea of what is the size of the ground program.
[14]This type of features can help to understand what is the "structure" of the analysed program.
[15]These features can give an indication on "how much" a program is close to be horn: this can be helpful, since some solvers may take advantage from this setting (e.g., minimum or no impact of completion [149] when applied).

Furthermore, there are different ways on how the Data could be structured. Some examples are a tree structure, a list (a degenerate tree) structure or a graph structure; for each of them, many properties might be useful, such as Connectivity, Components, Rank, Treewidth [290], etc. Although, the identification of these "structures" is often very expensive from a computational point of view.

We are now analysing all these features in order to identify which of them may be useful for our multi-engine approach. Moreover, we are investigating which ones we can estimate and how we can do it; because they should be cheap to compute, in order to be useful, but at the same time powerful enough to give insights on the choice of the engine.

Methodology

In addition to the *features* described above, and guided by the analysis of real logic programs developed in the earlier works mentioned before, we identified four methods that can be used to guide the solver selection.

These methods take as input *i*) the Data (actually, since the amount of data is usually huge, in the execution phase they take as input some "properties" or "statistical information" of the data), *ii*) the Program, *iii*) the Engines (their characteristics). The output of each of them should be the "*best*" engine to run. The "*best* engine is defined by two properties *i*) being able to execute the logic program encoded in the Transducer (remember that not all engines have the same expressivity) and *ii*) being able to do so in the least possible amount of time (given a fixed memory bound) w.r.t. the other engines.

In the following, we describe the methods that we identified.

Order based

Find a (partial) order among the engines based on some "high-level features" of the input Data and Program.

Example. Given a "size" feature on the *Data* and a "complexity" on the *Program*, we could have, for instance, the following cases:

- "**Large**" *Data* and "**Simple**" *Program*
- "**Small**" *Data* and "**Complex**" *Program*

If we consider two engines, a *SQL*-based one and an *ASP*-based one, we could have the following order:

- in case 3.3.2, $\mathbb{E}_{SQL} \lessdot \mathbb{E}_{ASP}$
- in case 3.3.2, $\mathbb{E}_{ASP} \lessdot \mathbb{E}_{SQL}$

where $\mathbb{E}_A \lessdot \mathbb{E}_B$ if $\mathbb{E}_A$ is "better" than $\mathbb{E}_B$.

In this example we assume that the best engine is expressive enough to support the given Program, otherwise, the method should rank it in a lower position or exclude it from the ordered list of engines.

Score based

Choose the engine based on the values of the score given by its characteristics.

In order to do this we have to map each characteristic (C_i) to the engines (E) that possess it, defining a score for each of them, and then we associate, through a weight, each rule R of the Program to the C of the E it is good for.

Example. In the following, a schematic version of this method.

$$\begin{tikzcd}
E_1 \arrow[r, leftrightarrow] \arrow[dr, leftrightarrow] & C_1 & R_1 : \ldots \arrow[l, "w_1"'] \\
\cdots & C_2 & R_2 : \ldots \arrow[ul, "w_2"] \\
 & & \cdots \\
E_n \arrow[dr, leftrightarrow] & \cdots & R_n : \ldots \arrow[dl, "w_n"] \\
\cdots & C_m & \cdots
\end{tikzcd}$$

Label based

Identify labels that are "meaningful" for the different types of rules. Then assign one (or more) label(s) to each rule of the Program and use these labels to decide the engine to run.

Example. Given, for instance, the following labels for specific types of rules:

TYPE	LABEL
$A(X,Y)->C(X)$ – (projection)	SIMPLE
$A(X,Y)->B(X,Y)$ – (renaming)	SIMPLE
$A(X,Y),B(Y,Z)->B(X,Z)$ – (join)	COMPLEX

We can use them to classify the rules of the Program in order to identify which engine is the most appropriate one for the input at hand.

Moreover, if we have information about the sizes of the predicates in the body we can try to predict the size of the predicate in the head and this can give helpful information for the selection of the engine. This topic is extensively analysed in Section 3.4.

Induction based

Collect and create examples of Data and Programs, and use Machine Learning techniques to analyse them, according to the "features" mentioned above.

Furthermore, we can try to identify the minimum set of features needed to be able to select the best engine.

This method is similar to the approach of most of the multi-engine systems developed in the *Logic Programming* field.

We are evaluating all these methods in order to identify the most appropriate ones for our needs.

3.3.3 Component Description

The VADALOG *Reasoner* component, as shown in Figure 3.3, it is the core of the VADA framework and should be integrated with all the other components (i.e., with the *Transducers* and the *Knowledge Base*). Therefore, its design and development were guided by modularity and scalability concepts as well as by clean and precise interfaces that can be easily accessed by other components.

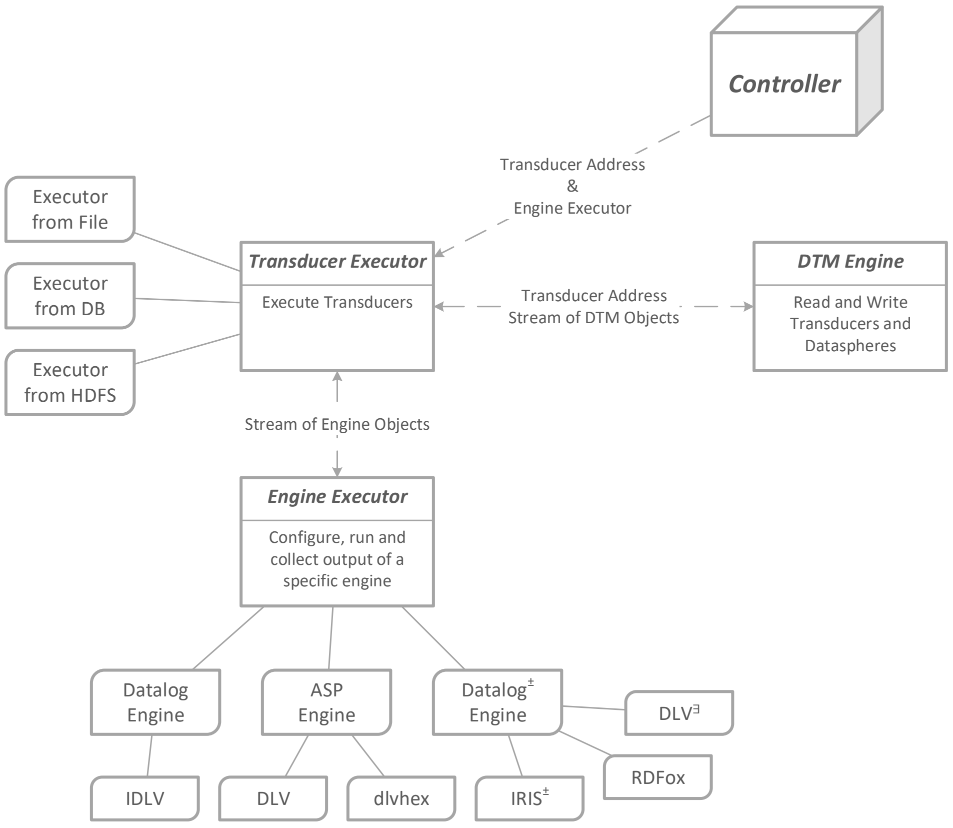


Figure 3.4.: Sketch of the VADALOG *Reasoner* component.

The central idea of our architecture, as shown in Figure 3.4 is quite simple. There is a *Controller* that specifies which Transducer have to be executed and which Engine will be responsible to do it. Then the *Transducer Executor* collects all the information needed using the *DTM Engine*; it is worth noticing that this component does not need to retrieve all the data, it may provide a handler, a stream or it might simply provide metadata (like location and credential to access the data) if the selected

Engine is able to directly access them[16]. And finally the *Transducer Executor* run the specified Engine through the *Engine Executor*, it retrieves the results from it and store them using the *DTM Engine*.
In Figure 3.5 is shown a specific execution of this flow where both the Transducer and the Datasphere are retrieved.

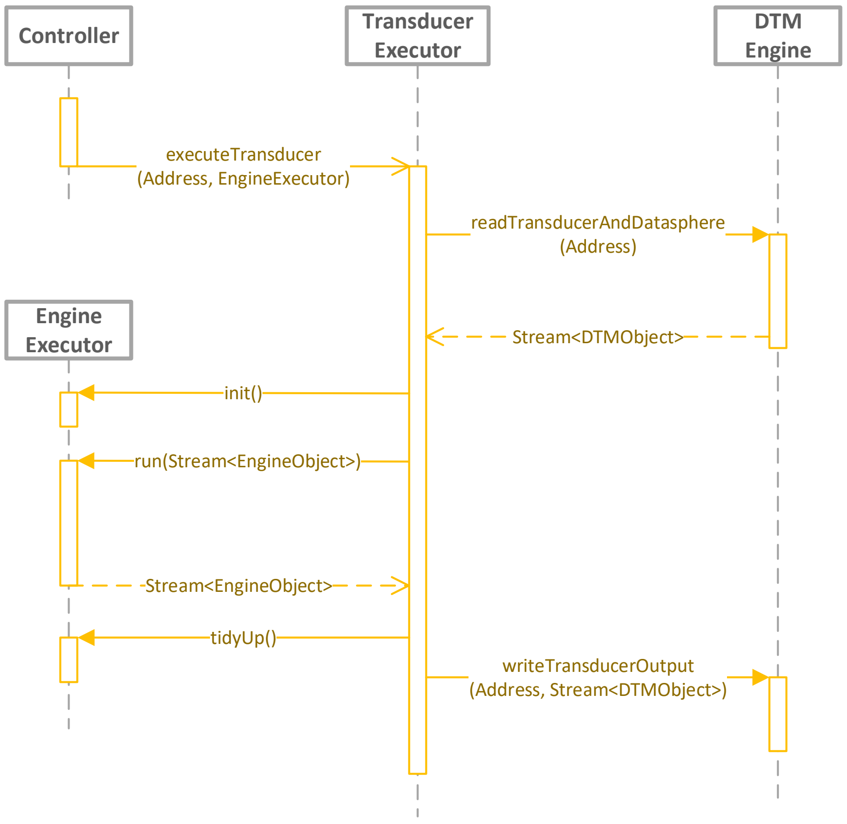

Controller
Transducer Executor
DTM Engine
executeTransducer (Address, EngineExecutor)
readTransducerAndDatasphere (Address)
Engine Executor
Stream<DTMObject>
init()
run(Stream<EngineObject>)
Stream<EngineObject>
tidyUp()
writeTransducerOutput (Address, Stream<DTMObject>)

Figure 3.5.: Sequence Diagram of a Transducer execution.

In order to be more general and fully compatible with other components, we decided that all the messages and all the *API* of this component have to use only flow of logic programs (mentioned in Figure 3.4 as *DTM* and Engine Objects) and/or addresses of Transducers.

In addition, it is worth noticing that *Transducer Executor* and *Engine Executor* are just interfaces and can be easily extended in order to introduce new functionalities and new engines. For instance, different *Transducer Executors* able to handle various kind of storages may be introduced and tested to identify the most appropriate ones.

However, the possibility to easily integrate new engines is a key feature in order to develop a scalable and adaptable multi-engine system. Using a double hierarchy,

[16]For instance, some solvers are able to directly access data stored into a *DataBase*, therefore if part of the data is stored into a *DB*, the *Transducer Executor* might simply provide the information needed to retrieve them.

we introduced different types of engines and several engines for each type. In Figure 3.4 are mentioned some categories and engines that we are integrating. We decided to integrate *Datalog* engines, such as $\mathcal{I}$-DLV, *ASP* engines, such as DLV and dlvhex, *Datalog*$^{\pm}$ engines, such as IRIS$^{\pm}$, RDFox and DLV$^{\exists}$. Moreover, we also integrated some preliminary engines that were developed at the beginning of the VADA project (a *Prolog*-based engine and a *SQL*-based one). As shown in [89], many systems can execute *TGDs* and/or *EGDs*, therefore we are planning to add even more systems with different characteristics to extend the range of choices for our engine selection.

Thanks to this modular architecture, we can easily test the solvers and the methods described in previous sections in order to identify how the *Controller* should behave to be successful.

3.3.4 Related Work and Discussion

Algorithm selection is a very old problem in computer science [502]. In the last years, multi-engine approaches became quite popular also in the *Logic Programming* community [152, 262, 300, 303, 320, 321, 380, 399, 400, 404, 406, 407, 452, 481, 520, 527, 587–589, 600] thanks to the availability of several solvers for each language that often use very different techniques to compute the models of a program. These approaches are often based on the identification of peculiar features of the logic program, the collections of some solvers for the specific language and on the use of Machine Learning techniques in order to select among the available solvers the “most promising” one, i.e., the one that should perform better than others, usually w.r.t. time and memory consumption, on the input program. The main reasons to use this approach are that very often the time and the memory required in order to solve a specific logic program strictly depend on the algorithm (and the heuristics) used by the solver and, in many cases, on a specific input program distinct engines have performances that are order of magnitude different.

As described in Section 3.3.2 we are analysing different methods and instead of limiting the multi-engine approach to Machine Learning algorithm on top of easy-to-compute features, we are trying to identify if also other kinds of methods, perhaps more “declarative”, can be applied to the selection of the engine. In this context many different techniques can be useful, for instance the *Rete Match Algorithm* [223] can be used to find all the matches of the features of the Program to the available engines.

These methods will be evaluated in the context of VADALOG programs and applied to *Data Wrangling* problems such the ones of the VADA project.

3.4 Time/Size estimation of Logic Programs evaluation – a refined approach

In this section we report our preliminary ideas and results on a size/time estimation of logic programs and its application in the context of multi-engine approaches.

Almost always the results of a reasoning process are needed as soon as possible and this is one of the motivations behind the huge amount of research made in the last decades in the area of *Artificial Intelligence* related to problem-solving. However, in many cases, the results of a reasoning task are useful only for a limited period of time, after which they became useless. This is certainly true in the *Stream Reasoning* context and definitely also in the context of (video-)games or, in general, "intelligent agents". Nevertheless, when dealing with huge amount of data, this problem becomes more important, since the time needed to produce meaningful and helpful results could be enormous.

Given these premises the question

"How much time will need my reasoning task in order to be completed?"

arises spontaneously.

We started to study this problem in the context of *Logic Programming*, specifically, we started to investigate the time required for a Logic Program to be evaluated. This problem is quite interesting and has been studied from various perspectives in different contexts (see Section 3.4.1), but estimating directly the time is quite hard mainly because the execution time is often strictly related to the specific optimizations and heuristic of the solver at hand.

One might think of determining the complexity class of the specific instance and, from it, try to derivate an estimate of the time, but this might not be very reliable because problems in the same complexity class can have really different execution time.

However, several investigations in the context of multi-engine systems pursue, in a direct or in an implicit way, exactly this goal. The main idea in multi-engine systems is to find, among the available ones, a proper (combination of) engine(s) able to perform better than any other. These systems usually use inductive approaches and are often very effective if the number of solvers is not large. Anyway, as explained in the next sections, they are mostly used on ground programs where a lot of information about the program are already available and, more important in this context, they do not have a precise idea of the time needed by the selected solver neither of how much the execution time of a program is different from another one.

Instead of estimating directly the time, another idea might be to estimate the size of each *IDB* predicate given the body literals of the rules in which it appears, i.e., roughly the number of rules that should be instantiated by as grounder, and this should approximately give the number of "operations" of the engine and therefore an appraisement of the time needed to evaluate the program. Also this problem has been largely investigated in the *Logic Programming* community because it is useful in many contexts, for instance to parallelize the execution of a solver [466], and, before, it was deeply studied also in the *DataBase* community [567] in order to identify the so-called "query size" that it is extremely useful in query optimization and, more in general, in query answering. More details on these studies are provided in Section 3.4.1.

In addition to estimating the time needed, it might be useful to identify the "time relation" among two programs, i.e., if one program can be evaluated quicker than another one and, if possible, also (the order of magnitude of) their gap. This is helpful in many different situations, for instance, if, as mentioned at the beginning, the results of the reasoning task are useless after a specified period of time and we have two different programs (P_1 and P_2) that can be executed. Assume that P_1 provides the exact result to the problem at hand and P_2, instead, provides an acceptable estimate. If we know (or we can estimate) that the execution time of P_1 is too big (it is orders of magnitude out of the time-limit we have) and we can estimate that the P_2 should take less time (perhaps orders of magnitude less than P_1) and thus it could probably be completed before the time-limit; we can decide (even in an automatic way) to use directly P_2 without wasting time and resources with P_1.

In the next section, we describe some basic notions and specific previous work in this context.

3.4.1 Preliminaries and Related Work[17]

This topic has been extensively studied and there are many works in the literature about it, especially about the estimation of the size. In the following we focus mainly on this problem.

[17]Preliminary definitions adapted from [305, 370, 371, 474]

Starting from the '80s in the *DataBase* community, several researchers tried to estimate the size of queries. Such estimating algorithms are essential to optimize queries (to select query execution plans) and avoid infeasible computations.

Lots of methods have been proposed in the literature, which differ in their complexity, cost, and accuracy (as usual, accuracy comes at the expenses of time and space requirements); these are just few examples [110, 546, 554]. Some of the firsts algorithms, which have also dealt with recursive queries were based on sampling [239, 381, 382]. These methods work in the *DataBase* context but do not work in the *Logic Programming* one because the "extension" of all the predicates are not available, to have then we should perform the "instantiation".

In the same context were proposed other approaches based on "*histograms*", and they became quickly very popular and influenced many consequential works [109, 110, 180, 305–307, 382, 473, 474, 536]. In the following we provide some details about them.

Consider a relation R with n numeric attributes X_i $(i = 1 \ldots n)$.
The *domain* $\mathcal{D}_i$ of attribute X_i is the set of all possible values of X_i and the (finite) *value set* $\mathcal{V}_i \subseteq \mathcal{D}$ of attribute X_i is the set of values of X_i that are actually present in R. Let $\mathcal{V}_i = \{v_i(k) : 1 \leq k \leq \mathcal{D}_i\}$, where $v_i(k) < v_i(j)$ when $k < j$.
The *spread* $s_i(k)$ of $v_i(k)$ is defined as $s_i(k) = v_i(k+1) - v_i(k)$, for $1 \leq k < \mathcal{D}_i$.[18]
The *frequency* $f_i(k)$ of $v_i(k)$ is the number of tuples in R with $X_i = v_i(k)$.
The *area* $a_i(k)$ of $v_i(k)$ is defined as $a_i(k) = f_i(k) \times s_i(k)$.

The *data distribution* of X_i is the set of pairs $\mathcal{T}_i = \{(v_i(1), f_i(1)), \ldots, (v_i(\mathcal{D}_i), f_i(\mathcal{D}_i))\}$.

The *joint frequency* $f(k_1, \ldots, k_n)$ of the value combination $\langle v_1(k_1), \ldots, v_n(k_n)\rangle$ is the number of tuples in R that contain $v_i(k_i)$ in attribute X_i, for all i.
The *joint data distribution* $\mathcal{T}1, \ldots, n$ of $X_1, \ldots, X_n$ is the entire set of (value combination, joint frequency) pairs.

In the sequel, for 1-dimensional cases, we use the above symbols without the subscript i.

Data distributions are very useful in *DataBase* systems but are usually too large to be stored accurately, so *histograms* come into play as an approximation mechanism.

A *histogram* on an attribute X is constructed by partitioning the *data distribution* of X into β (≥ 1) mutually disjoint subsets called *buckets* and approximating the

[18]We take $s_i(\mathcal{D}_i) = 1$.

frequencies and *values* in each *bucket* in some common fashion. This definition leaves several degrees of freedom in designing specific *histogram classes* as there are several possible choices for each of the following (mostly orthogonal) aspects of *histograms*:

Partition Rule:

This is further analysed into the following characteristics:

Partition Class

This indicates if there are any restrictions on the *buckets*. Of great importance is the *serial* class, which requires that *buckets* are non-overlapping w.r.t. some parameter (the next characteristic), and its subclass *end-biased*, which requires at most one non-singleton *bucket*.

Sort Parameter

This is a parameter whose value for each element in the *data distribution* is derived from the corresponding attribute *value* and *frequencies*. All *serial histograms* require that the sort parameter values in each *bucket* form a contiguous range. Attributes *value* (*V*), *frequency* (*F*), and *area* (*A*) are examples of sort parameters that have been discussed in the literature.

Source Parameter

This captures the property of the *data distribution* that is the most critical in an estimation problem and is used in conjunction with the next characteristic in identifying a unique partitioning. *Spread* (*S*), *frequency* (*F*), and *area* (*A*) are the most commonly used source parameters.

Partition Constraint

This is a mathematical constraint on the source parameter that uniquely identifies a single *histogram* within its partition class. Several partition constraints have been proposed so far, e.g., *equi-sum*, *v-optimal*, *maxdiff*, and *compressed*. Many of the more successful ones try to avoid grouping vastly different source parameter values into a bucket.

We use $p(s, u)$ to denote a *serial histogram class* with partition constraint p, sort parameter s, and source parameter u.

Construction Algorithm:

Given a particular partition rule, this is the algorithm that constructs *histograms* that satisfy the rule. It is often the case that, for the same *histogram class*, there are several construction algorithms with different efficiency.

Value Approximation:

This captures how *attribute values* are approximated within a *bucket*, which is independent of the partition rule of a *histogram*. The most common alternatives are the *continuous value assumption* and the *uniform spread assumption*; both assume values uniformly placed in the range covered by the *bucket*, with

SORT PARAMETER	SOURCE PARAMETER			
	SPREAD (S)	**FREQUENCY (F)**	**AREA (A)**	**CUM. FREQ (C)**
VALUE (V)	EQUI-SUM	EQUI-SUM V-OPTIMAL MAX-DIFF COMPRESSED	V-OPTIMAL MAX-DIFF COMPRESSED	SPLINE-BASED V-OPTIMAL
FREQUENCY (F)		V-OPTIMAL MAX-DIFF		
AREA (A)			V-OPTIMAL MAX-DIFF	

Table 3.1.: Histogram Taxonomy. From [474].

the former ignoring the number of these *values* and the later recording that number inside the *bucket*.

Frequency Approximation:

This captures how *frequencies* are approximated within a *bucket*. The dominant approach is making the *uniform distribution assumption*, where the *frequencies* of all elements in the *bucket* are assumed to be the same and equal to the average of the actual *frequencies*.

Error Guarantees:

These are upper bounds on the errors of the estimates a *histogram* generates, which are provided based on information that the *histogram* maintains.

A multi-dimensional *histogram* on a set of attributes is constructed by partitioning the *joint data distribution* of the attributes. They have the exact same characteristics as 1-dimensional *histograms*, except that the partition rule needs to be more intricate and cannot always be clearly analysed into the four other characteristics as before, e.g., there is no real sort parameter in this case, as there can be no ordering in multiple dimensions [473].

Several types of *histograms* have been obtained by specifying new choices for *histogram* "aspects" described above. Table 3.1 provides an overview of them. Note that all locations in the table correspond to valid *histograms*. All *histograms* make the *uniform spread* and the *uniform frequency* assumptions when approximating the *data distribution* within a *bucket*.

We discuss next the *maxdiff* type, which places *bucket* boundaries between adjacent source-parameter values (in sort-parameter order) whose difference is among the largest. As mentioned before, the goal of all the most successful partition constraints is to avoid grouping attribute values with vastly different source parameter values into a *bucket*. The *maxdiff histograms* try to achieve this goal by inserting *bucket* boundaries between adjacent source values (in sort parameter order) that differ by large amounts. These histograms can be efficiently constructed by first

Histogram	Time Taken (msec)	
	Space = 160b	Space = 400b
Compressed	5.9	9.3
Equi-sum	6.2	10.9
MaxDiff	7.0	12.8
V-optimal-end-biased	7.2	10.9
Spline Based	20.3	41.7
V-optimal	42.9	67.0
Equi-Depth: by P^2	4992	10524

Table 3.2.: Construction cost for various histograms. From [474].

Histogram	Error (%)
Trivial	60.84
Equi-depth: P^2	17.87
V-optimal (A,A)	15.28
V-optimal (V,C)	14.62
Equi-width	14.01
V-optimal (F,F)	13.40
V-optimal-end-biased (A,A)	12.84
V-optimal-end-biased (F,F)	11.67
Equi-depth: Precise	10.92
Spline-based (V,C)	10.55
Compressed (V,A)	3.76
Compressed (V,F)	3.45
Maxdiff (V,F)	3.26
V-Optimal (V,F)	3.26
Maxdiff (V,A)	0.77
V-Optimal (V,A)	0.77

Table 3.3.: Errors due to histograms. From [474].

computing the differences between adjacent source parameters, and then placing the bucket boundaries where the $\beta - 1$ highest differences occur.

Table 3.2 illustrates the difference in the construction costs of various *histograms*. It contains actual timings collected from running the corresponding algorithms on a SUN-SPARC, for varying amounts of space allocated to the *histograms*.[19] The specific timings are for *histograms* with V and A as the sort and source parameters, respectively, but all other combinations of sort and source parameters produce quite similar results. Table 3.3, instead, shows (in decreasing order) the errors generated by the entire set of *histograms* on some experimental results proposed in [474].

As can be seen from Table 3.2, the construction cost is negligible for most of the *histograms* when sampling techniques are used. As indicated in Table 3.3, a clear separation was observed throughout the experiments between a set of effective *histograms* and a set of poor *histograms*. Although the relative performance of *histograms* in the lower set varies between experiments, and on some occasions *histograms* from the upper set were competitive, the results in Table 3.3 are quite characteristic overall.

Approaches based on *histograms* could be used in the context of *Logic Programming*, although they are less effective since the extension of a predicate does not only depend on *EDB* but can also depend on *IDB* and this increases the error of the estimate.

In the last years, *Logic Programming* community started to gain interest in this topic. Researchers at the Stony Brook University developed an algorithm, called *SDP*, for

[19]These timings do not include the time taken to scan the relation and compute the sample.

estimating *Datalog* query sizes efficiently by calculating statistical dependency for both base and derived predicates [370]. Then they extended it to general rules and n-ary predicates and they made it also able to handle negation and mutual recursions as well as other operations [371].

Starting from the work on *histograms*, they first defined the notions described above using a logic-based nomenclature and then proposed a new concept, called "*Dependency Matrix*", which extend the notion of a *histogram* and can be viewed as two-dimensional *histograms*. We describe in the following these concepts for n-ary predicates.

Consider a n-ary predicate $p(x_1, \ldots, x_n)$.
If p is a base predicate, then its associated set of facts will be denoted by $factset(p)$.
The *value sequence*, $\bar{v}_d$ $(1 \leq d \leq n)$, is the *sorted* sequence of x_d-values that are present in $factset(p)$, and $\bar{v}_d^i$ is the i-th value of $\bar{v}_d$.
The *frequency*, $\bar{f}_d^i$, of $\bar{v}_d^i$ is the number of facts in $factset(p)$ with $x_d = \bar{v}_d^i$.
We use $\bar{v}^i$ to denote the i-th element of a sequence $\bar{v}$, "$\bullet$" to denote sequence concatenation, and $\| \cdots \|$ to denote the length of a sequence or the cardinality of a set.
Since we are dealing with discrete values in finite relations, all argument values can be assumed to be integers. Without loss of generality, we adopt this assumption in the sequel, for simplicity.

Given a *fact-set* for an n-ary predicate $p(x_1, \ldots, x_n)$, the *data distribution*, $\mathcal{T}_d$, for x_d is the sequence of value-frequency pairs $[(\bar{v}_d^1, \bar{f}_d^1), \ldots, (\bar{v}_d^m, \bar{f}_d^m)]$ where $m = \| \bar{v}_d \|$.

As mentioned before, *data distribution* is the basis for size estimation in all cost-based query optimizers, but this information is normally too large to store and use efficiently. One key step of all size estimation algorithms is to partition data distributions into distribution segments and summarize these segments in such a way that they can be approximated efficiently both in time and space. Therefore, *Dependency Matrices* were proposed in [370, 371] as one such summarization technique which keeps argument dependency information. There, *data distributions* are partitioned using the popular *maxdiff* rule. However, their size estimation algorithms do not depend on the choice of any particular partition rule and we can simply assume that distributions are partitioned according to "some" partition rule.

Consider a distribution $\mathcal{T}$ and its partition segments $\mathcal{T}^1, \ldots, \mathcal{T}^n$, each $\mathcal{T}^i$ has three parameters: *floor*, *ceiling*, *size* which are the minimal argument value, the maximal argument value, and the number of argument values contained in $\mathcal{T}^i$.
The set of values that are contained in $\mathcal{T}^i$ is approximated as $vals(\mathcal{T}^i) = \{v \in integers \mid \mathcal{T}^i.floor \leq v \leq \mathcal{T}^i.ceiling\}$. These three parameters constitute the summary of a partition segment and they are stored by *dependency matrices*.

$\mathcal{F}$	1	2	3	4	5	6	7	8
2		1						
3							1	1
4				1				
5					1		1	1
6						1		
7					1	1		
8	1		1					

The *fact-matrix* $\mathcal{F}$

$\mathcal{F}$	1	2	3	4	5	6	7	8
2		1						
3							1	1
4				1				
5					1		1	1
6						1		
7					1	1		
8	1		1					

The segmented $\mathcal{F}$

		(1,1,1)	(2,4,3)	(5,8,4)
		1	2	3
(2,4,3)	1		2	2
(5,5,1)	2			3
(6,8,3)	3	1	1	3

The *dependency matrix*

Table 3.4.: A *Dependency Matrix*. From [371].

More formally, let $p(x_1, \dots, x_n)$ be an n-ary predicate and let $\mathcal{T}_i$ $(1 \leq i \leq n)$ be the distribution for x_i. Suppose each $\mathcal{T}_i$ is partitioned into β_i segments $\mathcal{T}_i^{j_i}$, with $1 \leq j_i \leq {}_i$. The *dependency matrix* for p, denoted $\mathbf{M}\langle p \rangle$, is a matrix whose $(j_1, \dots, j_n)$-th element, $\mathbf{M}\langle p \rangle(j_1, \dots, j_n)$, is

$$\| \, \{p(x_1, \dots, x_n) \in factset(p) \mid \wedge_{1 \leq i \leq n} \, x_i \in vals(\mathcal{T}_i^{j_i})\} \, \|$$

In addition, the j_i-th coordinate on its i-th axis, denoted $\mathbf{M}\langle p \rangle_i^{j_i}$, is associated with three parameters: *floor*, *ceiling*, and *size* whose values are the same as the corresponding values associated with the distribution segment $\mathcal{T}_i^{j_i}$.

We often use $\mathbf{M}\langle p \rangle_i$ to denote the i-th axis of a matrix $\mathbf{M}\langle p \rangle$ and $\mathbf{M}\langle p \rangle_i^{j_i}$ to denote the j_i-th coordinate on $\mathbf{M}\langle p \rangle_i$.
From previous definition, we know that the matrix element $\mathbf{M}\langle p \rangle(j_1, \dots, j_n)$ summarizes $vals(\mathcal{T}_1^{j_1}) \times \cdots \times vals(\mathcal{T}_n^{j_n})$, and stores the number of $(x_1, \dots, x_n)$-values that are in the fact-set and summarized by this matrix element.
Given a $\beta_1 \times \cdots \times \beta_n$ *dependency matrix* $\mathbf{M}\langle p \rangle$, the size estimate of p, $size(p)$ or $size(\mathbf{M}\langle p \rangle)$, can be computed as the sum of all dependency matrix elements, i.e., $size(p) = \sum_{i_1, \dots, i_n} \mathbf{M}\langle p \rangle(i_1, \dots, i_n)$. Note that if p is a base predicate then $size(p)$ is its actual size. Therefore, one could estimate the size of a predicate by computing its *dependency matrix*.

In Table 3.4 is shown an example of *Dependency Matrix* for a predicate $p(x_1, x_2)$ with the following fact-set $factset(p)$:

$$p(2,2).\ p(3,7).\ p(3,8).\ p(4,4).\ p(5,5).\ p(5,7).$$

$$p(5,8).\ p(6,6).\ p(7,5).\ p(7,6).\ p(8,1).\ p(8,3).$$

The *dependency matrices* for the base predicates can be computed according to some partition rule, such as the *maxdiff* rule and definitions provided above. In order to compute *dependency matrices* for derived predicates, they defined a size estimation algorithm, called *statistics for derived predicates* (*SDP*). *SDP* computes *dependency* *matrices* of derived predicates by an abstract evaluation of their defining rules where rule bodies are replaced with algebraic expressions over the *dependency matrices* that correspond to the body predicates. Recursive rules are evaluated iteratively until approximate fixed points are reached.

Consider a $\beta_1^p \times \cdots \times \beta_m^p$ *dependency matrix* $\mathbf{M}\langle p\rangle$, a $\beta_1^q \times \cdots \times \beta_m^q$ *dependency matrix* $\mathbf{M}\langle q\rangle$, and two integers $1 \leq d_p \leq m$ and $1 \leq d_q \leq n$. We say that the d_p-th axis of $\mathbf{M}\langle p\rangle$, $\mathbf{M}\langle p\rangle_{d_p}$, and the d_q-th axis of $\mathbf{M}\langle q\rangle$, $\mathbf{M}\langle q\rangle_{d_q}$, are *aligned* if $\beta_{d_p} = \beta_{d_q}$, $\mathbf{M}\langle p\rangle^i_{d_p}.floor = \mathbf{M}\langle q\rangle^i_{d_q}.floor$, and $\mathbf{M}\langle p\rangle^i_{d_p}.ceiling = \mathbf{M}\langle q\rangle^i_{d_q}.ceiling$ for all $1 \leq i \leq \beta_{d_p}$. That is, $\mathbf{M}\langle p\rangle_{d_p}$ and $\mathbf{M}\langle q\rangle_{d_q}$ are *aligned* if all their coordinates have the same *floor* and *ceiling* parameters. For any pair of *dependency matrices* $\mathbf{M}\langle p\rangle$ and $\mathbf{M}\langle q\rangle$, and a pair of axis indexes d_p and d_q , we can always make $\mathbf{M}\langle p\rangle_{d_p}$ and $\mathbf{M}\langle q\rangle_{d_q}$ aligned by *refining* the associated partition segments.

Let $\mathbf{M}\langle p\rangle$ and $\mathbf{M}\langle q\rangle$ both be $\beta_1 \times \cdots \times \beta_n$ matrices. It follows directly from the definitions that $\mathbf{M}\langle p\rangle(j_1, \ldots, j_n)$ and $\mathbf{M}\langle q\rangle(j_1, \ldots, j_n)$ summarize the same values for all $j_1, \ldots, j_n$ if and only if $\mathbf{M}\langle p\rangle_d$ and $\mathbf{M}\langle q\rangle_d$ are *aligned* for all $d = 1, \ldots, n$. Since, as noted above, matrices can always be *aligned*, it follows that any pair of matrices can be *refined* so that they summarize exactly the same values.

The researchers at the Stony Brook University described also the algorithms for computing the *Dependency Matrices* for the following operations:

- Selection
- Union
- Intersection
- Projection
- Join
- Cross Product
- Negation

And, as mentioned before, also for *recursive predicates*.

They also presented an algorithm for computing size estimates for all predicates in a bottom-up fashion using the algebra over matrices defined earlier. This algorithm is based on the visit of all the trees in the *predicate dependency graph*'s *condensation* of a *Knowledge Base*. The algorithm resembles the usual naive bottom-up procedure for evaluating Horn rules except that here there is an abstract computation over the size estimation algebra.

For more details about the details of these algorithms, we refer the reader to the papers of the author of the *SDP* algorithm [369–371].

Operation	Complexity
construct $\mathbf{M}\langle p\rangle$	$O(n \times (ds \times \lg(ds) + fs))$
$project(\mathbf{M}\langle r\rangle, in\_args, out\_args)$	$O(\beta^n)$
$union(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(n \times \beta^n)$
$intersect(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(n \times \beta^n)$
$minus(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(n \times \beta^n)$
$join(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(\beta^{m+n-k})$
$product(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(\beta^{m+n})$

Table 3.5.: Complexity of *SDP* Operations. From [371].

In the same papers, they also presented an analysis of the complexity of these algorithms. They are summarized in Table 3.5. These results assume that all predicates are n-ary, fact-set sizes of base predicates are bounded by fs, domains of the arguments sizes are bounded by ds, and *dependency matrices* are all n-dimensional sparse matrices of sizes $\beta \times \cdots \times \beta$.

Furthermore, several multi-engine approaches, which deal with such estimations, have been proposed in the *Logic Programming* community. They are described in Section 3.3 therefore we do not describe them here. We just mention that the following we use the ME-ASP solver.

Lastly, it is worth noticing that several other approaches, based on different techniques, have been proposed to analyse and derive statistics (such as the time required) of logic programs [168–170, 416–418, 472], as future work we intend to provide a detailed comparison with them.

3.4.2 The idea

In order to dig into the problem and starting to analyse it, we have chosen a specific formalism, namely *Answer Set Programming* (*ASP*), and a precise application scenario, the discovery of the best engine in a multi-engine approach starting from the original program.

Therefore, in the following we mainly refer to *ASP* programs, but these concepts can be extended to other *Logic Programming* languages.

In order to define the general idea behind this work, we provide some useful notation: (derived from the definitions in [211])

P is a finite set of safe *rules*

$F(P)$ is the set of all *facts* in P

R is the set $P \setminus F(P)$ (the *rules* that are not *facts*)

$Ground(P)$ is the set of all *Ground Instances* of its *rules* over U_P (the *Herbrand Universe* of P)

$AS(P)$ is the set of all *Answer Sets* for P

In Figure 3.6 is reported a schematic visualization of the solving process of an *ASP* engine with the notation provided above in order to clarify it.

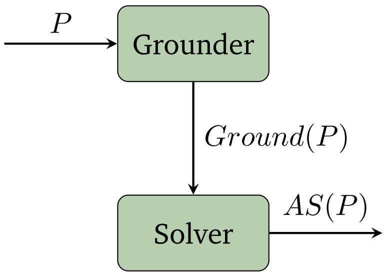


Figure 3.6.: *ASP* solving process schematization using the provided notation.

Moreover, we provide the following notation for the features:

$\mathcal{F}_P$ is the set of features of P

$\mathcal{F}_{F(P)}$ is the set of features of $F(P)$

$\mathcal{F}_R$ is the set of features of R

$\mathcal{F}_{Ground(P)}$ is the set of features of $Ground(P)$

The main idea is to design and develop algorithms (and then a system) that given a logic program P, extracts some features from $F(P)$ and from R, called respectively $\mathcal{F}_{F(P)}$ and $\mathcal{F}_R$ in order to be able to estimate (with a certain degree of precision δ) the time required by the *ASP* solver to evaluate the program (i.e., to find the $AS(P)$).

As described at the beginning, this idea could be exploited in different ways and for several purposes.

In this work, instead of considering the "explicit program and data features" as described in Section 3.3.2 and as in the different works about multi-engine approaches mentioned in Section 3.4.1, we consider the "features" as the estimate of $Ground(P)$, i.e., the size of the all the predicates in the program and the rules produced by the grounder.

Therefore, instead of deriving directly the time, we target the size. This approach is more powerful because, although for an initial investigation we focus on multi-engine systems, these derived data can then be used for different purposes *(a)* to predict the time needed by a solver, *(b)* to order different programs based on their expected size, *(c)* to suggest how some rules of a program can be modified in order to improve solving performances without modifying the meaning of the program, and many more.

In order to achieve this goal, i.e. to estimate $Ground(P)$, we started from the idea of the researchers at the Stony Brook University described in Section 3.4.1

and properly expanded it in order to improve the performance and allow their technique to deal with non-monotonicity, i.e., disjunction (in the head) or non-stratified negation (or their propagation).

Actually, we are not interested in all the details of $Ground(P)$, but only on how many rules of each type we have in $Ground(P)$. As "type", for this preliminary stage, we consider the number of atoms in the body and in the head (that roughly represents the number of operations that have to be performed) because it has been identified as one of the most important features in the multi-engine approaches described before. Therefore, the $\mathcal{F}_P$ are composed of the number of rules for each size (eventually with the *Dependency Matrices* of each predicate, that could be considered as "raw data").

In the *ASP* context, as well as in other logic-based languages, some operations are provided during the grounding phase (i.e. the instantiation) while other during the solving phase (i.e. the model generation and the model checking). The Grounder, given the rule (r_1)

$$r_1: \;\; a(X) \;{:}\!-\; b(X), c(X).$$

is able to derive all the ground values of a in the *Answer Sets* of the program, if the values of b and c are defined. Therefore, in this case, this rule is not evaluated in the solving phase, because it is discarded by the Grounder.

However, in rules disjunction (in the head) or non-stratified negation, like

$$r_2: \;\; d(X) \mid e(X) \;{:}\!-\; b(X), c(X).$$

the Grounder is not able to define the values of d or e (therefore they are called "*undefined*") and thus the Grounder produces the ground version of the whole rule r_2. Moreover, all the rules that have *undefined* atoms in the body (like d and e), cannot be simplified by the Grounder and increase the number of rules that the solver should deal with.

The Grounder also provides further simplifications, for instance, given a rule like

$$r_3: \;\; f \;{:}\!-\; b, \textbf{not}\; g.$$

it perform these kinds of simplifications:

- remove b if it is known to be true and
- remove the entire rule if g is known to be true

Because we want to evaluate the number of rules of each type produced by the grounder and the number of atoms they contain, we should be able to estimate these simplifications.

In order to do this we keep 2 *Dependency Matrices* for each predicate:

- The S (*Simplified*, i.e., True) *Dependency Matrix*
- The U (*Undefined*, i.e., Potentially True) *Dependency Matrix*

And we compute the size of a predicate as the sum of all dependency matrix elements and the size of a rule, using the following method.
Given the rule :

$$r_4: \quad a(X) :- b(X), c(X).$$

Let $\mathbf{M}_S\langle\psi\rangle$ and $\mathbf{M}_U\langle\psi\rangle$ be, respectively, the "*simplified*" and the "*undefined*" *dependencies matrices* for the predicate ψ, we compute the size of the rule r_4 and the predicate a in the following way:

- Performing the join between $\mathbf{M}_S\langle b\rangle$ and $\mathbf{M}_S\langle c\rangle$
  We obtain the rules of size 0 (the facts) and we update $\mathbf{M}_S\langle a\rangle$
- Performing the join between $\mathbf{M}_S\langle b\rangle$ and $\mathbf{M}_U\langle c\rangle$ and the join between $\mathbf{M}_U\langle b\rangle$ and $\mathbf{M}_S\langle c\rangle$
  We obtain the rules of size 1 (with either b or c in the body) and we update $\mathbf{M}_U\langle a\rangle$
- Performing the join between $\mathbf{M}_U\langle b\rangle$ and $\mathbf{M}_U\langle c\rangle$
  We obtain the rules of size 2 (with both b and c in the body) and we update $\mathbf{M}_U\langle a\rangle$

Clearly the "*undefined*" *dependencies matrices* ($\mathbf{M}_U\langle\psi\rangle$) could be empty if b and c are derived from monotonic rules.

This technique can easily be extended to other operations and also to rules with more atoms in the body.

Next we describe our first prototypical system implementing this approach.

3.4.3 System Description

In order to compute the dependency matrices described in the previous sections and to estimate the number of rules of $Ground(P)$, we built a system, called *ASPtimator*, based on 3 modules. The basic components of *ASPtimator*, as shown in Figure 3.7, are:

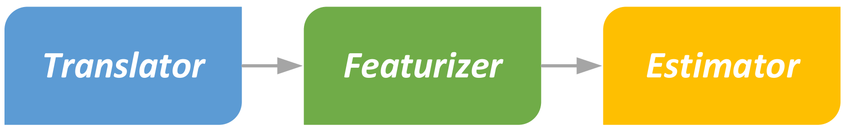


Figure 3.7.: Basic components of *ASPtimator*.

the *Translator*

It parses the logic program, finds the *Strongly Connected Components*, the *condensation* of the *dependency graph* and the *SDP* operations that need to be performed and converts them to a *Middle File Format*

the *Featurizer*

It computes the *Dependency Matrices* for *EDB* and *IDB* and identify the number of rules of each type

the *Estimator*

It uses the data of the *Featurizer* to perform the predictions (time, solver, order, etc.)

We do not describe all the technical details here because they are not of interest in this context. However, it is worth noticing that *(a)* the *Translator* module has been implemented in *Java* using the DLV *Wrapper* and it writes its output as a *JSON* file; *(b)* the *Featurizer* module has been implemented in *Python* and it is able to read the *Middle File Format*, to the *Dependency Matrices* for the *EDB*, to execute the *SDP* operations in order to derive the *Dependency Matrices* for the *IDB* predicates and to compute the number of rules for each type (i.e. the pair of number atoms in the head, number of atoms in the body);[20] and *(c)* the *Estimator* module is conceived to be generic enough to be able to perform different "predictions" but it has not been implemented yet, for first tests we are planning to use the one of ME-ASP.

The *Middle File Format* contains the *SCC*s of the program with all the *SDP* operations that need to be performed and has the following structure:

- A list of *SCC*, each of which contains a subprogram with a list of rules
  Each rule contains a type and
  if the rule is a fact it contains also:
  - the predicate name
  - predicate arity
  - terms values

  otherwise it contains also:
  - predicates in the head
  - predicates in the body

[20]It performs also some basic statistics about the elapsed time for each operation and the accuracy of the estimation produced (if a ground-truth is provided)

- operations

  Each operation contains a list of *SDP* operations with the following data:
  - the *SDP* operation name
  - the arguments of this *SDP* operation
  - a new name for the resulting predicate of this operation (if not given, all the predicate names of the rule are considered)

This format allows to define all the information needed to the *Featurizer* in order to perform the *SDP* algorithm described before. Having the *Translator* module and the *Featurizer* as two separate and independent components allows to easily adapt this approach to other logic-based formalisms.

3.4.4 Discussion

We started our preliminary tests using the native approach of the researchers at the Stony Brook University but, in our case, the sizes predicted are often very far from the real ones. In detail, the results of *Selection*, *Union* and *Cross Product* are quite good (as also reported in their paper) and for some examples of *Projection* and *Join* the errors decrease quite sharply, although they remain significantly high; however, in other cases, the predicted values are orders of magnitude far from the real ones.

We noticed that a great source of error is the first part of the algorithm where the parameters (*floor*, *ceiling* and *size*) of the two Dependency Matrices have to be "aligned". Therefore, we are investigating more on the *alignment* and *refinement* algorithms mentioned in their works and explained in the Ph.D. Dissertation of one of the authors.

Moreover, we discovered that the Containment Assumption [21] can play a crucial role in the estimation of the predicate size, as reported also in the literature [148, 521, 554]. Also the concept of "Modularity" of logic programs [197, 453] can be useful in this context.

At the moment we are continuing these investigations, and we are working on the development and the testing of the *ASPtimator* system. We are planning to validate our approach and then extend it to more features and other types of rules, as well as other logic-based formalism.

[21] *When joining two tables, the set of values in the join column with the smaller column cardinality is a subset of the set of values in the join column with the larger column cardinality.* From [554]

Wrap-up

In this chapter we reported about some investigations on the application of multi-engine approaches in the context of ontological reasoning applied to *Data Wrangling* scenarios. In addition, we introduced preliminary ideas on the characterization of logic programs and their usage in multi-engine approaches.
It is worth noticing that some of the investigations reported in this chapter have been conducted during the visit at the *Department of Computer Science* (lead by *Professor Georg Gottlob F.R.S.*) at the *University of Oxford*[22], working on the VADA Project.

Nonetheless, similar questions might also arise in contexts that at first glance may appear quite far and different, such as video games. In the next chapter, we report about some experiment performed, using *Logic Programming*, in the context of *Artificial Intelligence* applied to Games and "Intelligent Agents".

[22] http://www.ox.ac.uk

4 Logic and *AI* in Games

Thus we have on stage two men, each of whom knows nothing of what he believes the other knows, and to deceive each other reciprocally both speak in allusions, each of the two hoping (in vain) that the other holds the key to his puzzle.

— **Umberto Eco**
(The Island of the Day Before)

Summary of Chapter 4

AI is a very popular topic nowadays and has been applied to a large number of different fields, and new areas of application are continuously identified.

One of the fields that has given (and has taken) more to *AI* and where it is currently applied the most is Games. In this field, many new techniques are constantly proposed: among these, also different logic-based methods have been specially developed and optimized for games. We think that *Logic Programming*, combined with other *AI* formalisms and approaches, is an excellent paradigm to be used in the creation of *intelligent agents*, as shown in this chapter, and it is likely to become a prominent approach in the future.

In this chapter we first give some preliminary information about (video) games, *Artificial Intelligence* (*AI*) and the different use of *AI* techniques in games; then we present (in Sections 4.4 and 4.5) some projects and experiments we were involved in, where we applied logic-based and *Artificial Intelligence* techniques in the field, and we describe how this synergy has proved to be successful.

Chapter Outline

4.1 Definition, Motivation and Challenges

4.1.1 *AI* in Games: an outline[1]

Artificial Intelligence (*AI*) has seen immense progress in recent years. It is both a thriving research field featuring an increasing number of important research areas and a core technology for an increasing number of application areas. In addition to algorithmic innovations, the rapid progress in *AI* is often attributed to increasing computational power due to hardware advancements. *AI* advances have enabled a better understanding of images and speech, emotion detection, self-driving cars, web searching, *AI*-assisted creative design, and game-playing, among many other tasks; for some of them, machines have reached human-level status or beyond.

The relationship between *Artificial Intelligence* (*AI*) and games is well-established since long time and games have been helping *AI* research progress because games pose interesting and complex problems for *AI*. But it is not only *AI* that is advanced through games; games have also advanced through *AI* research because, in the broadest sense, most games incorporate some form of *Artificial Intelligence*. This relationship has been quite fruitful as witnessed by the enormous number of scientific publications and books produced about this topic [10, 97, 112, 143, 154, 155, 353, 424, 439, 482–484, 499, 519, 562, 591, 592] and all the companies and research groups that work in this area. Apart from industrial and academics research groups that have and are working on project related to this topic, it is worth mentioning that all the major IT company have nowadays research divisions, like IBM Research, Microsoft Research, Google DeepMind and Facebook AI Research, that are actively working in project related with *AI* and games.

Moreover, many research organizations and communities have created Conferences about *AI* and Games, like the *IEEE Conference on Computational Intelligence and Games (CIG)*[2], the *AAAI Conference Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*[3], the *International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational and Serious Games (CGAMES)*[4], and recently [326, 391] also some important Journals have been introduced, such as the *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*[5].

[1]Preliminary definitions adapted from [586, 592]
[2]`http://www.ieee-cig.org`
[3]`http://www.aaai.org/Library/AIIDE`
[4]`http://www.cgames.org`
[5]`http://cis.ieee.org/ieee-transactions-on-computational-intelligence-and-ai-in-games.html`

Furthermore, many competitions involving *AI* in games have been proposed by private or companies but also by research groups.
In Appendix A are described some of the most popular ones.

Furthermore, various challenges between *Intelligent Agents* (bots) created by popular company and the best professional players of specific games have had a great journalistic success and those bots had great achievements. Some examples:

- *IBM Watson* in the *Jeopardy!*® game[6]
- *Google DeepMind AlphaGo* in the Go game[7]
- *Alibaba* in the *StarCraft*® game[8]
- *OpenAI* in the *Dota 2*™ game[9]

Clearly *Intelligent Agents* are a central topic in the context of *Artificial Intelligence* and Games. Find a definition for "agent" is not easy, authors in [586] distinguished two general usages of the term:

A Weak Notion of Agency

Perhaps the most general way in which the term agent is used is to denote a hardware or (more usually) software-based computer system that enjoys the following properties:

autonomy agents operate without the direct intervention of humans or others, and they have some kind of control over their actions and their internal state [139];

social ability agents interact with other agents (and possibly humans) via some kind of *agent-communication language* [275];

reactivity agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;

pro-activeness agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by *taking the initiative*.

A Stronger Notion of Agency

Some researchers – particularly those working in *AI* – generally mean an "agent" to be a computer system that, in addition to having the properties identified above, is either conceptualised or implemented using concepts that

[6] [107, 108, 218–220, 366, 558] and http://researcher.watson.ibm.com/researcher/view_group_pubs.php?grp=2099
[7] [525, 526] and https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html
[8] [465] and https://github.com/alibaba/gym-starcraft
[9] https://blog.openai.com/dota-2

are more usually applied to humans. For example, it is quite common in *AI* to characterise an agent using mentalistic notions, such as knowledge, belief, intention, and obligation [523]. Some *AI* researchers have gone further and considered emotional agents [73, 74]. Another way of giving agents human-like attributes is to represent them visually, perhaps by using a cartoon-like graphical icon or an animated face [396, p. 36].

Various other attributes are sometimes discussed in the context of agency, like:

mobility is the ability of an agent to move around an electronic network [580];

veracity is the assumption that an agent will not knowingly communicate false information [236, pp. 159-164];

benevolence is the assumption that agents do not have conflicting goals, and that every agent will therefore always try to do what is asked of it [507, p. 91];

rationality is (crudely) the assumption that an agent will act in order to achieve its goals, and will not act in such a way as to prevent its goals being achievedat least insofar as its beliefs permit [236, pp. 49-54].

Other attributes of agency are formally defined in [285].

As noted in [586], Intelligent Agents (and their issues) can be analysed from different perspectives:

Agent theory is concerned with the question of what an agent is, and the use of mathematical formalisms for representing and reasoning about the properties of agents.

Agent architectures can be thought of as software engineering models of agents; researchers in this area are primarily concerned with the problem of designing software or hardware systems that will satisfy the properties specified by agent theorists.

Agent languages are software systems for programming and experimenting with agents; these languages may embody principles proposed by theorists.

In the following, we will refer mainly to logic-based Intelligent Agents, based on the "stronger" notion presented above, and we will describe them from all the different areas mentioned before.

Logic-based formalisms have been exploited in this context, especially in the last decade, in order to improve the existing approaches and to demonstrate the powerful capabilities of *KR&R* of this paradigm. This topic will be discussed in details in Section 4.2.

4.1.2 *AI* in Games: why it is important

> In which area of human life is *Artificial Intelligence* (*AI*) currently applied the most?
> The answer, by a large margin, is Computer Games.
> This is essentially the only big area in which people deal with behaviour generated by *AI* on a regular basis.[10]

Clearly, *AI* research and video games are a mutually beneficial combination. On the one hand, *AI* technology can provide solutions to an increasing demand to add realistic, intelligent behaviour to the virtual creatures that populate a game world. On the other hand, as game environments become more complex and realistic, they offer a range of excellent test-beds for fundamental *AI* research.

There are a number of reasons why games offer the ideal domain for the study of *Artificial Intelligence*. In [592] the authors identified the following:

- Games are Hard and Interesting Problems
- Games have a Rich Human Computer Interaction
- Games are Popular
- There Are Challenges for All *AI* Areas
- Games Best Realize Long-Term Goals of *AI*

Moreover, logic-based approaches, can gain several valuable and useful suggestion when applied to such a broad and varied domain, for instance on how to improve the languages and the techniques in order to make them more affordable for people without a deeper knowledge of logic or more "productive", i.e. to develop specific extensions that make easier and faster some reasoning tasks.

4.1.3 *AI* in Games: challenges[11]

The Video Game industry is continuously growing and it has taken the lead in entertainment industry. This industry is increasingly adopting the techniques and recommendations academia offers, especially in the *AI* area. Moreover, it is successfully using many of the advances obtained by academia, although there are many non-tackled challenges in this sense.

[10]From `https://www.microsoft.com/en-us/research/project/video-games-and-artificial-intelligence/`
[11]Preliminary definitions adapted from [355, 390]

Research in *Artificial Intelligence* may take advantage of the wide variety of problems that video-games offer, such as adversarial planning, real-time reactive behaviours and planning, and decision-making under uncertainty.

Playing games with *AI* is still a very open field. For many different games, we just have simple heuristics and we do not know which techniques, or combination of techniques, are more appropriate to solve them. The improvements in this field, as pointed out at the beginning of this chapter, are continuous and very important in many different areas (Computer Vision, reasoning techniques, content generation, Learning, agents intelligent behaviours, etc.) but they are not fully solved yet.

The use of *Logic Programming* adds more challenges to the field due to its inherent proprieties. For instance, speed and memory consumptions are often an issue in games and these are also weak points of logic-based approaches; the use of incremental and reactive techniques could be useful in this case but they are still in a very early stage of development.

4.2 Logic for games' *AI*[12]

Historically *Artificial Intelligence* (*AI*) has been associated with logic-based or symbolic methods such as Reasoning, *Knowledge Representation* and Planning[13].

Many different logic-based approaches and techniques have been applied to games and used to develop their *AI*s.

A well-known example of a branch of *AI* that has been extensively used to realize games' *AI*s is Automated Planning [280]. The field of automated planning has studied planning on the level of symbolic representations for decades. Typically, a language based on first-order logic is used to represent events, states and actions, and tree search methods are applied to find paths from the current state to an end state. In this area a lot of different techniques have been invented, one of the most popular (used in many famous video games) is *Goal-Oriented Action Planning (GOAP)*[14], which refers to a simplified STRIPS-like planning architecture specifically designed for real-time control of autonomous character behaviour in games. Many different papers have been published on the topic, for more information see also [68, 455–458, 578].

Planning, as well as other *AI* techniques, has also been used for automatic content generation, such as level contents or solutions, for more information also [301, 433, 531, 532, 534, 562]. Authors of [533] were even able to build a Logical Game Engine, called LUDOCORE, that is able to provide a higher-level language to describe games and centralized solutions to tedious or error-prone programming tasks, achieving not only a concise representation of a game's mechanics but also the ability to automatically generate interesting gameplay traces that meet meaningful constraints.

It is worth noting that many other logic formalisms can explicitly represent an agent's knowledge; one of the most widely known is *Prolog*, that has been widely employed in games *AI*s [312, 329, 555, 572].

[12]Preliminary definitions adapted from [533, 592]

[13]While *Computational Intelligence* (*CI*) has been associated with biologically-inspired or statistical methods such as neural networks (including what is now known as deep learning) and evolutionary computation.

[14]`http://alumni.media.mit.edu/~jorkin/goap.html`

4.2.1 In the *Answer Set Programming* community[15]

As mentioned in the previous chapters, *Answer Set Programming* became widely used in *AI* and is recognized as a powerful tool for *Knowledge Representation and Reasoning*, especially for its high expressiveness and the ability to deal also with incomplete knowledge. The fully declarative nature of *ASP* allows one to encode a large variety of problems by means of simple and elegant logic programs. The semantics of *ASP* associates a program with none, one, or many *Answer Sets*, each one corresponding one-to-one to the solutions of the problem at hand. For these reasons many researchers, in the latest years, applied *ASP* in the games domain.

One of the first application was in the context of interactive gaming environment. The Qsmodels project [460] aimed to demonstrate the viability of using *ASP* in the environment of the *Quake 3 Arena* (Q3A) adopting the high-level agent architecture described in [55] that consists in the loop: "Observe - Select Goal - Plan - Execute". Q3A is a *first-person shooter*: the player's goal is to kill enemies using weapons and upgrades found inside the game field (normally a labyrinth). The human-like enemies found within Q3A are called BOTs. Like in the most computer games, Q3A bots behave according to the rules of a *finite-state machine* (*FSM*) defined by expert game programmers.

The Qsmodels architecture consists of two layers executed concurrently: a high level, responsible for mid-term and long-term planning, and a (low level) in charge of plan execution and emergency state reactions.
The authors highlighted that their architecture has several advantages over the traditional schema for the *AI* part of games. It is easier to develop and keeps the *AI* at higher level of abstraction. The easiness in development is reached by keeping the planning rules separated from the *world model description* rules so that they can be written even by *AI* beginners.

Another early application was in the domain of *cognitive agents*, i.e. such with explicit representation of their mental attitudes such as beliefs, goals, obligations and alike. In [447] the author focus on implementation of agent's belief base as an *ASP Knowledge Base* (*KB*).

An agent using an *ASP* module in its belief base and at the same time embodied in an environment requires a programming framework which (i) allows an easy integration of heterogeneous *Knowledge Bases* treated on a par, and (ii) provides a flexible programming language for encoding of agents behaviours. No single *Knowl-*

[15]Preliminary definitions adapted from [234, 447, 448, 460, 530]

edge Representation technology offers a range of capabilities and features required for different application domains and environments agents operate in.

Therefore, the authors developed a modular agent programming language, called *Jazzyk* [448], based on the programming framework of *Behavioural State Machines* (*BSM*) [446]. *BSM* framework, and thus also *Jazzyk*, draws a strict distinction between the *knowledge representational layer* and a *behavioural layer* of an agent program. To exploit strengths of various *KR* technologies, the *knowledge representational layer* is kept abstract and open, so that it is possible to plug-in different heterogeneous *KR* modules as agent's *Knowledge Base*. The main focus of *BSM* computational model is the highest level of control of an agent: its *behaviours*. Hence, it supports a high degree of modularity w.r.t. employed *KR* technologies, and at the same time provides a clear and concise semantics.

The *Jazzyk* modular agent programming language was then used to implement *Jazzbot*, a virtual agent embodied in a simulated 3D environment of a *first-person shooter* computer game *Nexuiz*. *Jazzbot* provides a test-bed for investigation of applications of non-monotonic reasoning techniques, *ASP* in particular, on a realistic, yet affordable agent system. *Jazzbot* is a goal-driven agent. It features four *KR* modules representing *belief base*, *goal base*, and an interface to its *virtual body* in a *Nexuiz* environment respectively. While the goal base consists of a single *KB* realized as an *ASP* logic program, the belief base is composed of two modules: an *ASP Logic Programming* one and a *Ruby* module.
The use of logic-based techniques is clearly beneficial for agent development, even if in this work the authors believe that they are better suited for modelling static aspects of the environment rather than for agents' behaviours.

Recently, in [530] was proposed an extension to the traditional Japanese logic puzzle *Sudoku*, named *Proofdoku*, developed within the practice of *AI-based game design [205]*. The *Proofdoku* project aims to uncover new player experiences unreachable without the affordance of *Artificial Intelligence* (*AI*) systems as well as to understand how the context of a deployed game pushes back on those systems. In *Proofdoku*, the player works under the traditional rules of *Sudoku* with one key twist: they must explain their reasoning (and it must be valid). A small *AI* system, built using the technology of *Answer Set Programming* (*ASP*) and co-developed with the design of the game, plays several roles during gameplay including checking the validity of player arguments and computing hints for the various phases of play. Through interaction with *Proofdoku*, players can learn and generalize new deductive inference patterns for *Sudoku* (including those unknown to the designers).

The author specified that the *Proofdoku* project was initiated to answer specific questions about applying *ASP*:

- What practical engineering concerns arise when deploying an *ASP*-backed gameplay experience?
- Which aspects of live play might be enabled or enhanced using *ASP*?

These questions are answered in the paper in necessarily game-specific ways. However, the concerns that arose in design, development, and deployment touch on much broader issues: maintaining responsiveness of the *AI* system for players; the level of project-specific engineering; software licensing; and monetary costs associated with centralization.

Some years ago we also used the famous *Guess/Check/Optimize* (*GCO*) methodology to design and implement the *AI*s of some popular games (*Connect Four* and *Reversi*) in order to show some advantages of declarative programming frameworks (in particular *Answer Set Programming*) against imperative (algorithmic) approaches while dealing with *KR&R*: solid theoretical bases, no need for algorithm design or coding, explicit (and thus easily modifiable/upgradeable) knowledge representation, declarative specifications which are already executable, very fast prototyping, quick error detection, modularity. We implemented "classic" strategies, typically difficult to implement when dealing with the imperative programming, in a rather simple and intuitive way. Moreover, we had the chance to test the *AI* without the need for rebuilding the application each time we made an update, thus observing "on the fly" the impact of changes: this constitutes one of the most interesting features granted by the explicit *Knowledge Representation*. In addition, we developed different versions of the *AI*s, in order to show how easy is to refine the quality or to generate different strategies or "styles"; and these include also non-winning human-like behaviours.

4.3 *Angry Birds* and the *Angry Birds AI Competition*[16]

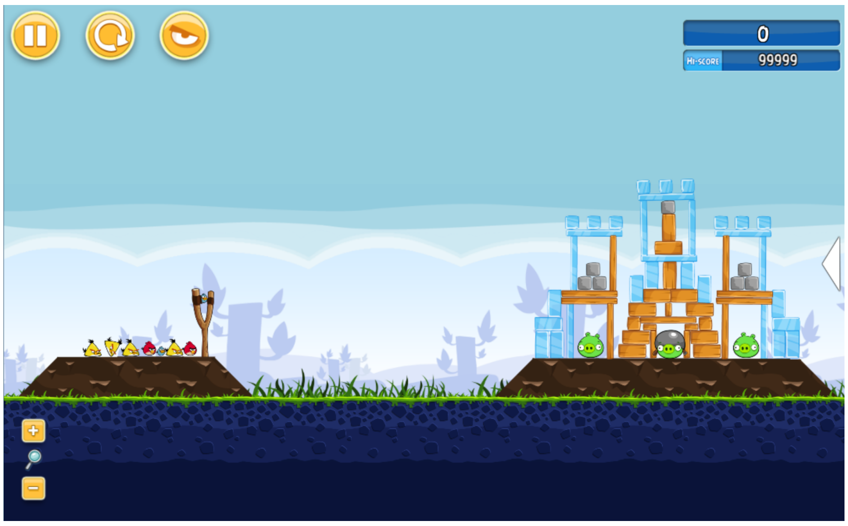


Figure 4.1.: A screenshot of the *Angry Birds* ™ game. Courtesy of Rovio Entertainment Corporation.®.

4.3.1 *Angry Birds*

Angry Birds is one of the most popular games of all times. It has a simple gameplay and one simple task: destroy all pigs of a given level by throwing different "angry" birds at them using a slingshot (see Figure 4.1). The pigs are protected by a structure composed of blocks of different materials with different physical properties such as mass, friction, or density. The actions a player can perform are specified by (1) the release coordinate $\langle x, y \rangle$ and (2) the tap time $\langle t \rangle$ after release when the bird's optional special power is activated. A game level is solved if executing a selected sequence of actions $\langle x, y, t \rangle$ leads to a game state that satisfies certain victory conditions.

In *Angry Birds*, all green pigs need to be destroyed in order to solve a given game level. The score is given to the player according to the number of destroyed pigs and objects plus a bonus for each spared bird. The pigs are sheltered by complex structures made of objects of different materials (wood, ice, stone, etc.) and shapes, mostly but not exclusively rectangular. After a player's shot, the scenario evolves complying with laws of physics, with the crash of object structures and a generally complex interaction of subsequent falls.

[16]Preliminary definitions adapted from [494–498, 568]

Different birds have different behaviours and special powers, and while the player knows the order in which birds will appear on the slingshot, the player cannot manipulate this order.

Interestingly, in this game, one can find many of the challenges that physics-based games present to the *AI* community. These are mostly related to the need of dealing with uncertainty in several respects, such as predicting the unknown consequences of a possible move/shot, or estimating the advantages of a choice w.r.t. possible alternative moves, or planning over multiple moves where any intermediate move is subject to failure or unexpected outcome. In turn, the above technical challenges require the effective resolution of other important issues, like the identification of objects via artificial vision, the combination of simulation and decision-making, the modelling of game knowledge, and the correct execution of planned moves.

Actually, *Angry Birds* is an example of the *physics-based simulation game* (*PBSG*) category, i.e. a video game where the game world simulates real-world physics (Newtonian physics).

The game world in these games is typically completely parametrized, i.e., all physics parameters such as mass, friction, density of objects, gravity, as well as all object types and their properties and location are known internally.

*Physics-based simulation game*s have been around since the beginning of video games. Even some of the very first games on commercial game consoles fall under this category. Such games consist of objects, liquids, or other entities that behave according to the laws of physics and they often use an underlying physics simulator that computes the correct physical behaviour. These games look and feel very realistic as all actions a player performs have outcomes that are more or less consistent with what one would expect to happen in the real world. What this requires is that all physical properties of all game entities and the game world, such as mass, density, friction, gravity, metric, angles, or locations, are exactly known to the game. Then each action and each movement can be exactly and deterministically computed by the physics simulator.

Implementing the physics of these games is quite a standard task, and the biggest advance over the years has been the more and more realistic and sophisticated graphics. *Physics-based simulation games* (*PBSG*s), such as *Angry Birds*, *Cut the Rope*, *Gears*, or *Feed Me Oil*, form a very popular game category, particularly through the rise of touchscreen devices that allow easy manipulation of the game world and easy execution of actions by the players. Interaction with the game via touching is particularly suitable for physics games as it feels like interacting with real objects. These games are easy to play as the possible moves are simple.

The game world in these games simulates Newtonian physics using a game internal physics engine such as Box2D[17] (used by *Angry Birds*) that knows all physics parameters and spatial configurations at all times, which makes the physics of the gameplay look very realistic.

*Physics-based simulation game*s and *Artificial Intelligence* have always had a close and fruitful relationship. This is particularly useful for physics simulation games since other entities in the game world should behave intelligently; they should behave like they are controlled by other human players. What makes these games particularly hard and challenging for *AI* is that the number of possible moves can be very large and effectively infinite, and that the consequences of moves are unknown in advance. The large number of moves is due to the effect of the exact location and/or timing of moves, where small changes may result in differences in the outcome of the physics simulation. Often, noise is added to the simulator to make these games more challenging. Without actually simulating a move, its outcome is very hard to predict. It becomes even harder if the exact physical properties of the objects or the behaviour of the simulator are unknown for the *AI* beforehand and have to be learned through observation. The task of an *AI* agent is not only to predict the outcome of an individual move but to identify a move that achieves the desired outcome.

A recent research trend in *Artificial Intelligence* is to build systems or agents that can play *physics-based simulation game*s as good as or better than human players.
This is a very different problem from traditional Game *AI* and most probably a much harder one. The main difference is that for Game *AI*, all physical parameters and the complete information of the game world are known to the *AI*. What is unknown is the behaviour of the human player who could be an opponent or a partner, or who could be ignored, depending on the game. In this case, the *AI* knows only as much about the game worlds as it can see. Therefore, computer vision should be employed to detect objects and tell the *AI* where they are and what they are. While this gives us uncertainty about what and where the objects are, another major problem is that the outcome of actions is unknown. Simulating the effects of an action is easy when all physical parameters are known, but if they are not exactly known then a simulation does not produce accurate results and one has to find other ways of predicting the outcome of actions. Humans are very good at predicting physics thanks to a lot of practice and experience in interacting with the real world. For *AI*, this is still a very difficult problem that needs to be solved in order to build *AI* that can successfully interact with the real world.

[17]`http://box2d.org`

This "human way" of playing makes it very hard for computers to play well, particularly compared to games like *chess* that are difficult for humans, but easy for computers to play. The main difficulty, as mentioned above, of *PBSG*s like *Angry Birds* is related to the problem that the consequences of physical actions are not known in advance without simulating or executing them. This is partly due to the fact that the exact physics parameters are not available, which makes exact calculations impossible. But even if they were available, it would be necessary to simulate a potentially infinite number of possible actions, as every tiny change in release coordinate or tap time can have a different outcome. Knowing the outcome of an action is important for selecting a good action, and particularly for selecting a good sequence of actions.

This way of solving problems is very similar to what humans have to do every day when interacting with the physical world and what humans are very good at; that is, physical properties of entities are unknown, most information is obtained visually, the action space is continuous, and the exact outcome of actions is not known in advance.

The capabilities required for accurately estimating consequences of physical actions using only visual input or other forms of perception are essential for the future of *AI*. Any major *AI* system that will be deployed in the real world and that physically interacts with the world must be aware of the consequences of its physical actions and must select its actions based on potential consequences. This is necessary for guaranteeing that there will not be any unintended consequences of its actions, that nothing will get damaged and no one will get hurt. If this cannot be guaranteed, it is unlikely society will accept having these *AI* systems living among them, as they will be perceived as potentially dangerous and threatening. *Angry Birds* and other *PBSG*s provide a simplified and controlled environment for developing and testing these capabilities. It allows *AI* researchers to integrate methods from different fields of *AI*, such as Computer Vision, Machine Learning, *Knowledge Representation and Reasoning*, Heuristic Search, Reasoning under Uncertainty, and *AI* Planning that are required to achieve this.

4.3.2 The *Angry Birds AI Competition* (*AIBIRDS*)

The *Angry Birds AI Competition* (*AIBIRDS*) was initiated in 2012 by some researchers of the Australian National University and is held in collocation with some of the major *AI* conferences, such as the *European Conference on Artificial Intelligence* in 2014 and the *International Joint Conference on Artificial Intelligence* in 2013, 2015, 2016 and 2017.

The task at the Competition is to play a set of *Angry Birds* levels within a given time limit, typically about 3 minutes per level on average. Levels can be played and replayed in any order. However, the levels are new and have not been seen by the participants during the development and training of the agents.

In the *AIBIRDS* Competition, we play *Angry Birds* using the web version, publicly available at [389]. The Competition server interfaces with the website using a Chrome™ browser extension which allows us to take screenshots of the live game and to execute different actions using simulated mouse operations. Participating agents run on a client computer and can only interact with the server via a fixed communication protocol. This allows agents to request screenshots and to submit actions and other commands which the server then executes on the live game, such as obtaining the current reference scores for each level. Therefore, the only information participants obtain are sequences of screenshots of the live game. Hence, *AI* agents have exactly the same information available as human players. In particular, they do not know the exact location and other parameters of objects or the game world.

In order to make it easier for participants to build their own agents, the organizers provide basic game-playing software and also encourage all participants to open-source their agents. The framework provided by organizers only requires a Chrome™ browser and a *Java* ™ environment and can be installed on most popular operating systems. What is provided to participants is a computer *Vision Module* that detects known (=hard-coded) objects and gives an approximation of the objects boundary, their location and type. In addition, a trajectory planning module is provided which allows agents to specify which point they want to hit with a bird and if they want to shoot with a high or a low trajectory. This module then returns the approximate release point that hits the given target point. Since this depends on the scale of the game world, which can be different for every level, the trajectory planning module automatically adjusts trajectories in subsequent shots. The framework also includes an interface to the official *Angry Birds* game that can take screenshots and execute mouse actions. In order to demonstrate the use of these modules, a sample agent called the *Naïve Agent* is provided, which selects a random pig as the next target, and selects a random trajectory and tap point depending on the bird type. Having a bit of randomness seems beneficial as it avoids being trapped in unsuccessful strategies. It can also help to make a lucky shot. Interestingly, the *Naïve Agent* was the winner in 2012 and still outperforms about one-third of the agents in the current benchmark.

The agents are ranked according to their combined high scores over all solved levels and after several rounds of elimination a winner is determined. There is a qualification round where we select the best agents for the final rounds, followed by group stages of four agents where the two best agents of each group progress to the next round until only two agents are left. They then compete in a grand finale to determine the champion. At the end of every Competition, there is the *Human vs Machine Challenge* where is tested whether the best *AI* agents are already better than humans (=typically conference participants). The performance of agents is clearly improving every year. In 2013, agents were clearly better than beginners, while in 2014 the best agent was already better than two-thirds of the human players.

In order to achieve the goal of the Competition, i.e., to foster the development an *AI* agent that can play any previously unseen level as good as or better than the best human players, we need to efficiently solve a number of problems in an environment that behaves according to the laws of physics:

- detect and classify known and unknown objects
- learn properties of (unknown) objects and the game world
- predict the outcome of actions
- select good actions in a given situation
- plan a successful action sequence
- plan the order in which game levels are played

As mentioned before, these problems can be covered by different areas of *AI* such as Computer Vision, Machine Learning, *Knowledge Representation and Reasoning*, Planning, Heuristic Search, and Reasoning under Uncertainty, but due to the physical nature of the game world and the unknown outcome of actions, these are largely open problems. Progress and contributions in each of these areas will improve the performance of an agent, but in order to reach the goal, we need to jointly develop solutions to these problems across different *AI* areas. What makes research on *PBSG* such as *Angry Birds* so important, is that the same problems need to be solved by *AI* systems that can successfully interact with the physical world. Humans have these capabilities and are using them constantly, *AI* is a long way away in this respect. The *Angry Birds AI Competition* and other *PBSG*s offer a platform to develop these capabilities in a simplified and controlled environment. They allow *AI* researchers to focus on the core problems without distractions that are present in the real world. Approaches that work in these domains would constitute an important stepping stone for developing intelligent agents that perform well in other real-world tasks. Successfully integrating methods from these areas is indeed one of the great challenges of *AI*.

Competition Setting

Game Environment For each agent participating in the Competition, a unique corresponding *Angry Birds* game instance runs on a game server, while the agent itself is executed on a client computer. The Competition machinery supports *Java*, *C/C++* and *Python* agents, and can run the artificial players under either Windows or Linux. Each agent is allowed a total of 100MB of local disk space on the client computer, for its convenience, including the space required for the agent code. Client computers have no access to the internet, and can only communicate with the game server by means of a specific communication *API*. No communication with other agents is possible, and each agent can only access files in its own directory.

The communication with the game server allows each agent to obtain screenshots of the current game state to submit actions and other commands. The actual game is played over the Google Chrome (browser) version of *Angry Birds*, in SD (low-resolution) mode, and all screenshots have a resolution of 840×480 pixels.

Game Objects The objects an agent might have to deal with correspond to all block and bird types, background, terrain, etc. occurring in the first 21 levels of the "Poached Eggs" level set available at `chrome.angrybirds.com`. In addition, the Competition levels may include white birds, black birds, so-called TNT boxes, triangular blocks and hollow blocks (triangle and squares).

Intuitively, a proper knowledge about each game component is crucial, in order to develop a decently performing agent. In particular, the special features of each bird type are an essential aspect of the game: one might say that the whole reasoning process depends on this.

Once a bird is shot, the player can perform an additional "tap" anytime afterwards, provided that it is performed before the bird touches any other object. This action causes different events according to each bird type: blue birds generate multiple little blue birds; yellow birds accelerate and become very good at breaking wood; white birds drop an explosive egg while accelerating towards the sky; black birds explode making great damage, and so on. "Tapping" at the appropriate moment in time can make shot outcomes vary greatly.

Game Levels and Competition Setting The levels used in the Competition are not known in advance to the participants and are not present in the original version of the game; throughout the Competition, each game level can be accessed, played and re-played in arbitrary order by the agent. Participants have a total time budget to solve the Competition levels corresponding to a few minutes per game level, on average. As an example, as reported by the official Competition rules, for 10 levels there is a maximum allowed time of 30 minutes. Once the overall time limit is reached, the connection of agents with the game server is terminated, then the agents have up to two minutes to store any needed information and then stop running.

The *Naïve Agent* is launched on all qualification game levels in advance; it does not participate in the Competition, but its high scores are recorded and intended to provide participants with a reference baseline.

Some strategy is needed when the agent takes part in multiple rounds. In particular, each participating agent must be able to distinguish between qualification round 1, qualification round 2, and the Finals.
Qualifications are run in two rounds, both on the same level set. During the first qualification round, agents can obtain, besides their own scores, the per level high score obtained by the *Naïve Agent*; during the second qualification round, agents can obtain the overall per level high score obtained in the first round (among all participants). Agents can, for example, program the strategy by determining the game levels where they can obtain the highest improvements between the two rounds. Agents cannot be modified between round 1 and round 2.
The highest scoring agents after qualification round 2 participate in the finals, where they are divided into groups. During finals, any agent can query the current group high score for each game level (but not the high scores of other groups). Grand finals are played on groups of two agents only.

It is clear that time plays a fundamental role in the agent performance and that being just able at shooting birds is definitely not enough to be successful in such a Competition. Many choices must be taken at the strategic level, i.e. taking into account both game rules and Competition settings, altogether with the time limits, the scores from the sample agent and the competitors, and the specific capabilities of the agent itself.

It is also worth noting that, since the agent is supposed to play each level more than once with the aim of improving the previous scores, some kind of learning mechanisms, and especially memory capabilities, are welcome.

Participants[18]

The *Angry Birds AI Competition* attracted interest from participants all over the world. Most are academic participants and students, some from research institutes and even *AI* amateurs. Over 40 teams from 17 countries have participated so far

[18]Preliminary definitions adapted from [95, 319, 358, 471, 575]

and a multitude of *AI* approaches have been tried.
In the following we briefly describe, in alphabetical order, some of the more interesting teams.

The *Beau-Rivage* team [319] (Champion of 2013), from Switzerland, used Machine-Learning techniques, referring to the Multi-Armed Bandit (MAB) problem. They observed that the game can be interpreted as search tree with a very shallow depth d, but a large branching factor. More importantly, without the actual physics engine, it is difficult to estimate the consequence of the actions. Therefore, it is more reasonable to model the outcome (score) of a given strategy-level pair as a distribution. Playing a level with a certain static strategy would then be analogous to sampling from the corresponding distribution. The problem of optimizing sampling from different distributions is known as the MAB problem.

The *DataLab Birds* team [95] (Champion of 2014 and 2015), from the Czech Republic, used Qualitative Structural Analysis, using a simple planning agent that decides which strategy to play for the current move considering the environment blocks configuration, reachable targets, possible trajectories, the bird currently on the sling and the birds available on the stage. They developed 4 different Strategies: the Dynamite strategy, the Building strategy, the Destroy as many pigs as possible strategy and the Round blocks strategy. In the Building strategy, to destroy the physical structure formed by connected blocks, the agent identifies and targets the structure's weak part, which is determined by spatial relations between the building blocks. Moreover, they added specific logic for tapping time, white bird and trajectories estimation.

The *IHSEV* team [470, 471], from France, used Advanced Simulation, proposing a generic framework based on theory of mind, which allows an agent to reason and perform actions using multiple simulations of automatically created or externally added models of the perceived environment.

The *Impact Vactor* team [575], from Poland, used Qualitative Reasoning, assigning each object which is reachable by a shot a numerical value expressing the scale of damage it does once hit. The playing program is then supposed to shoot at the object with the highest value. Interestingly they defined formally the Quantitative and the Qualitative representations of the gameplay scene, providing formal definitions of *influence* (which they further divided into two categories – *horizontal impact* and *vertical impact*), *stability* and *connection points*. Whereas the latter two can be evaluated regardless of the "type" of shot (i.e. if the trajectory is *high* or *low*), the former requires an assumption about the shot and it is counted by means of iterative analysis involving two central concepts: *propagation of force* (in the case of horizontal impact) and *centre of rotation* (in the case of vertical impact). Moreover, in the value estimation, they considered the type of the bird that will be launched, the type of the shooting trajectory (high or low parabola) and a block which is a direct target of the shot.

The *Plan A+* team [358], from South Korea, used Heuristics, proposing multiple strategies with different strategic approaches. The agent selects some targets and for each time to shoot, it generates multiple trajectories (maximum two for each target) and counts the number of objects (stone, wood, and ice) to be broken by the bird on the sling. Then it uses different scoring method to select the best trajectory.

Also, other researchers in [464, 575, 597, 598] used spatial reasoning or qualitative physics to estimate the outcome of a shot. As they say, observed one approach to solve the game is by analysing the structure and identifying its strength and weaknesses. This can then be used to decide where to hit the structure with the birds. On balance, this technique seems quite effective and promising.

Many more approaches have been proposed and many more *AI* techniques have been tested; the AKBABA team [515, 516], from Germany, used search and simulation to find appropriate parameters for launching birds, the *AngyBER* team [565, 566], from Greece, improved it proposing a Bayesian ensemble regression framework, the *AngryBNU* team [594], from China, used deep reinforcement learning, the *s-birds (Avengers/Returns)* team [164–166], from India, used a hybrid approach based on rote learning, the *SEABirds* team [450], from Germany, used an Analytic Hierarchy Process (AHP) with background knowledge and heuristics, the *TeamWisc* team [438], from the USA, used the Weighted Majority Algorithm (WMA) and Naïve Bayesian Networks to learn how to judge possible shots, the *UFAngryBirdsC* team [34], from Brazil, used a training/execution mode based on the ExpectMiniMax algorithm, authors in [188] integrated deliberately planning and acting with refinement methods. Unfortunately, we cannot analyse all of them in detail here.

A lot of research has been carried out in the latest year around the *Angry Birds* topic, about Level Generation [217, 313, 540, 541], Representation and Reasoning [245–248] and even Computational Complexity [543]. This demonstrates how much the field is active, how researchers are interested in the topic and how it could be useful and productive to work in this area thanks to the different contributions and influences that can be obtained.

4.4 Angry-HEX: An Artificial Player for *Angry Birds* Based on Declarative *Knowledge Bases*[19]

In this section, we present Angry-HEX, an artificial player for *Angry Birds*, based on Declarative *Knowledge Bases*, that participated in many editions (from 2013 to 2017) of the *Angry Birds AI Competition*, which recently inspired a number of research contributions [128, 438, 566, 598]. The agent features a combination of traditional imperative programming and declarative programming that allows us to achieve high flexibility in strategy design and knowledge modelling. In particular, we make use of *Artificial Intelligence* (*AI*) and HEX *programs* [199], which, as mentioned in Section 1.3.5, are a proper extension of *ASP* programs towards integration of external computation and knowledge sources; *Knowledge Bases*, written in *ASP*, drive both decisions about which target to hit for each shot (tactic gameplay), and which level should be played and how (strategic gameplay). The usage of *Answer Set Programming* has several benefits. First of all, *ASP Knowledge Bases* (*KBs*) are much more flexible and easier to change than a procedural algorithm. In particular, thanks to the availability of constructs such as aggregate atoms and weak constraints, reasoning strategies can be easily encoded as a set of few rules, while the development of dedicated algorithms is time-consuming and error-prone. Therefore, conducting experiments with different strategies in order to optimize our agent is much easier. Furthermore, as mentioned in Section 1.3, *ASP* is a general purpose declarative language in which temporal and spatial reasoning can be embedded as shown for instance in [283, 367]; also, *ASP* can be used in the planning domain, for modelling action languages (see e.g. the seminal work [375]), and probabilistic reasoning [56].

The advantages above come however at the price of a lower scalability when large input datasets have to be dealt with. In the context of the *Angry Birds* game, it is also necessary to deal with data coming from (approximately) continuous domains: Indeed, physics simulation libraries, using floating-point operations, are needed; such data cannot be efficiently dealt with in a natively discrete, logic-based framework such as *ASP*. In this respect, HEX *programs*, an extension of *ASP*, are well-suited as they allow for encapsulating external sources of information: on the one hand, numeric computations and part of the spatial processing tasks are computed on what can be called "the physics side" and can be embedded as external sources; on the other hand, actual reasoning strategies can be specified declaratively on a declarat-

[19]From F. Calimeri, M. Fink, **S. Germano**, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci and A. Wimmer. 'Angry-HEX: An Artificial Player for Angry Birds Based on Declarative Knowledge Bases'. In: *TCIAIG* 8.2 (2016), pp. 128–139. DOI: 10.1109/TCIAIG.2015.2509600.
And F. Calimeri, M. Fink, **S. Germano**, G. Ianni, C. Redl and A. Wimmer. 'AngryHEX: an Artificial Player for Angry Birds Based on Declarative Knowledge Bases'. In: *[45]*. 2013, pp. 29–35. URL: http://ceur-ws.org/Vol-1107/paper10.pdf.

ive “reasoning side”. This way, numeric processing is hidden in the external sources, since returned elaborated results are limited to those aspects which are relevant for the declarative part; this can be encoded by a fairly limited set of discrete logic assertions.

The main contributions of this work can be summarized as follows. Towards modelling dynamic (situation-dependent) and static knowledge in a *physics-based simulation game*, we propose a hybrid model in which logic-based, declarative knowledge modelling is combined with traditional programming modules whose purpose is acting on the game and extracting discrete knowledge from the game itself. In our architecture, the decision support side is implemented in an extension of *Answer Set Programming* (*ASP*) integrated with external sources of computation modelled by means of traditional imperative programming. We contextualized this approach to the *Angry Birds* Game, and propose an agent which participated in the *Angry Birds AI Competition* (*AIBIRDS*) Series. Also, we analysed the performance of the proposed agent in several respects.

We point out next the advantages of the herein proposed approach.

- It is possible to deal with the respective limitations of both declarative modelling and traditional programming and gain instead from respective benefits; indeed, on the one hand, *Logic Programming* is extremely handy and performance efficient when dealing with discrete domains, but it has limited ability to cope with nearly-continuous domains, and at the price of unacceptable performance. On the other hand, ad-hoc programmed modules lack flexibility but allow efficient processing of nearly-continuous knowledge (ballistic and geometric formulas, artificial vision, etc.).
- The introduction of declarative logic-based knowledge bases allows combining statements concerning common-sense knowledge of the game (e.g., in the context of *Angry Birds*, “Blue birds are good on ice blocks”) with objective knowledge (e.g. “an ice block w pixels wide is currently at coordinates (x, y)”) when performing decision-making; it permits also to focus attention on declarative descriptions of the game knowledge and of the goals to be achieved, rather than on how to implement underlying evaluation algorithms. This allows fast prototyping, and consequently much greater efficiency in the usage of developer time. For instance, both Strategy and Tactics behaviours can be easily refined and/or redefined by quickly changing logic assertions.
- Benchmarks show that logic-based approaches, and particularly, *ASP*-based approaches, if properly combined with procedural facilities, can be employed in applications having requirements near to real-time, making the gap between the performance of current logic-based solutions and requirements of pure real-time applications much narrower.

- The approach generalizes to a wide range of applications which share a number of aspects with the *Angry Birds* setting, such as automated room cleaning, semi-interactive turn-based games, planning in slowly evolving environments, such as robot and space probes etc. One can adapt our approach to such or similar scenarios, by adding a *process* step to the traditional *observe-think-act* cycle [344], thus obtaining a sort of *observe-process-think-act* cycle, and properly implementing the four stages. More in detail, the observation and process phases can be implemented by hard-wiring sensors and processing their inputs in a procedural environment, providing a focused set of external sources of knowledge to the think phase; the process phase plays the role of discretizing and reducing information, so that the think phase, carried out by a logic-based system, is efficiently performed, still keeping the benefits of declarative knowledge modelling; the act phase is straightforward and re-wires decisions to actual actions. A deployment of an approach similar to present work can be found in [222].

In the following, we overview the agent architecture and outline specific design choices introduced for dealing with physics and uncertainty; then, we comment experimental results in terms of time and score performance. Finally, we draw conclusions and discuss open issues and future development.

4.4.1 The Angry-HEX Agent

Since logic-based reasoning is not specifically tailored to reasoning with non-discrete domains, it is particularly challenging to deal with physics-based simulations. This technical challenge can be coped with a hybrid system. Hence, we propose a double-sided architecture, in which a "decision-making" side and a "simulation side" can be identified. The decision support side is realized using a logic-based *Knowledge Base*, while the simulation side is out-sourced to specialized library code.

In particular, in the Angry-HEX agent the decision-making process is carried out by computing the *Answer Sets* of a number of HEX *programs*. Namely, the program P_{Tact} models the knowledge of the game within a single level, i.e. tactical aspects; and P_{Strat} models the strategical knowledge required when deciding which level is convenient to be played. When decisions have to be made, both P_{Tact} and P_{Strat} are coupled with respective sets of logical assertions A_{Tact} and A_{Strat}, where A_{Tact} describes the situation in the currently played level, and A_{Strat} describes the overall game status (scores, etc.). Respectively, each *Answer Set* of $P_{Tact} \cup A_{Tact}$ describes a possible target object to be hit with a bird, while the *Answer Sets* of $P_{Strat} \cup A_{Strat}$ describe which is the next level to be played.

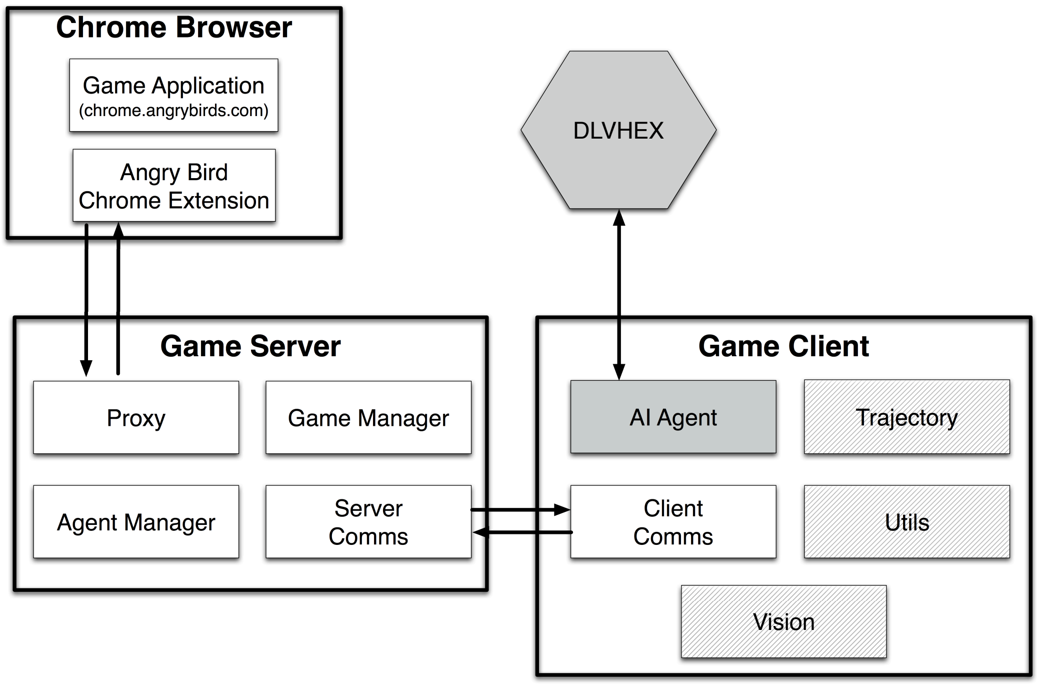


Figure 4.2.: The Framework Architecture. Slanted lines rectangles represent parts of the framework modified by our team, while grey ones represent the modules entirely developed by our team. Remaining modules were provided by the Competition organizers.

It is worth mentioning that we did not make use of Machine Learning techniques or other means for automatically obtaining knowledge of the game. Tactics and Strategy have been modelled based on our own experience and experiments with the game. Our contribution is indeed focused on the ease of knowledge modelling rather than automated learning, which can be subject of future research.

We describe next the main components of the Angry-HEX agent.

Framework Architecture

The Framework architecture consists of several components as shown in Fig. 4.2. The *Angry Birds Extension* works on top of the Google Chrome™ browser, and allows interacting with the game by offering a number of functionalities, such as capturing the game window and executing actions (e.g., clicking, zooming). The *Vision Module* segments images and recognizes the minimum bounding rectangles of essential objects, their orientation, shape and type. Objects include birds of various colours (Red, Blue, Yellow, Black and White), pigs of different sizes, the Slingshot, and bricks made of several materials (Wood, Ice and Stone) and shapes.

The *Trajectory Module* estimates the parabolic trajectory that a bird would follow, given a particular release point of the slingshot. The *AI Agent* stub is supposed to include the *Artificial Intelligence* programmed by participants of the Competition, therefore it is the core module implementing the decision-making process. The *Game Server* interacts with the *Angry Birds Extension* via the *Proxy* module, which can handle commands like `CLICK` (left click of the mouse), `DRAG` (drag the cursor from one place to another), `MOUSEWHEEL` (scroll the mouse wheel), and `SCREENSHOT` (capture the current game window). There are many categories of messages (Configuration messages, Query messages, In-Game action messages and Level selection messages); the *Server/Client Communication Port* receives messages from agents and sends back feedback after the server executed the actions asked by them.

Our agent uses all these framework utilities in order to obtain information and, in general, to play the game levels.

Other Improvements to the Framework Architecture

The framework utilities allow an agent to gather the information needed to play the game levels; hence, we enhanced some modules of the base architecture in order to fit our needs. These parts are reported in Figure 4.2 as boxes filled with slanted lines; in the following, we discuss such improvements.

- Concerning the *Vision Module*, we added the possibility of recognizing the orientation of blocks and the level terrain; even though these features were later implemented in the Framework Architecture by the Competition organizers, until very recently, we preferred to stick to our version, for what some particular vision tasks are concerned.
- In the Trajectory Module, we added thickness to trajectories. A parabola is attributed a *thickness* value proportional to the size of a bird: this feature is helpful in order to exclude actually unreachable targets from the set of possible hits, because of narrow passages and sharp edges in objects' structures. Also, while the original Trajectory Module is capable of aiming at objects' centroids only, we can aim at several points taken on the left and on the top face of objects (recall that birds are always shot from the left-hand side of the screen). This has a two-fold benefit: first, objects that have their centroid hidden by other objects are not necessarily out of a possible hit, for instance, an object can have its top face clearly reachable while its centroid point is not; second, we can better choose the most convenient among a set of hitting points. For instance, higher points on the left face of an object are preferable because of an expectedly greater domino effect.
- Waiting for the outcome of a shot can be a time-consuming task: we added a quick-shot modality in which the next shot is performed after a fixed time, although there might still be slightly moving objects.

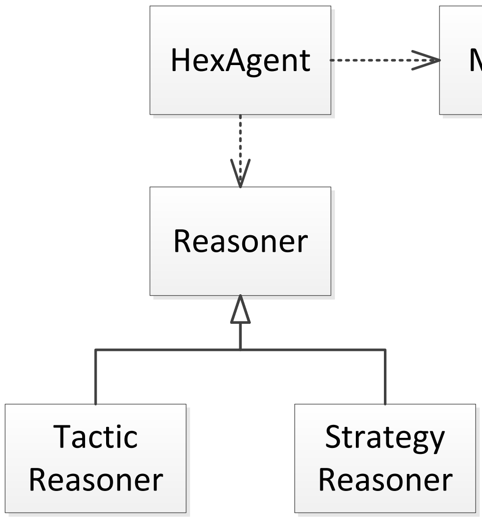


Figure 4.3.: An overview of the Angry-HEX Agent Architecture.

Our Agent

As already said, the core component of the framework Game Client is the *AI* agent. Figure 4.3 shows an overview of the Angry-HEX agent: the bot is composed of two main modules, the **Reasoner** and the **Memory**.

The Memory module provides the agent with some learning mechanisms; in its current version, its first goal is to avoid that Angry-HEX replays the same level in the same way twice (for instance, by selecting a different initial target at the beginning of a level). Such an approach results to be quite effective, since changing the order of objects to be shot (even just the first one) results in completely different outcomes in terms of level evolution, and hence in future targeting choices and possibly improved scores for the same level.
The Reasoner module is in charge of deciding which action to perform. This module features two different intelligence layers: the **Tactics** layer, which plans shots and steers all decisions about "how" to play a level, and the **Strategy** layer, which establishes in what order the levels have to be faced; this layer decides also whether it is worth replaying, not necessarily in a consecutive attempt, the same level more than once.

Tactics layer. The Tactics layer is declaratively implemented using the dlvhex solver, which computes optimal shots on the basis of the information about the current scene and the knowledge modelled within the HEX *program* P_{Tact}.

In particular, the Tactics layer accesses and produces the following information.

- *Input data*: scene information encoded as a set of logic assertions A_{Tact} (position, size and orientation of pigs, ice, wood and stone blocks, slingshot, etc. as obtained by the *Vision Module*); a *Knowledge Base* P_{Tact} encoding knowledge about the gameplay. It is worth noting that physics simulation results and several other pieces of information are accessed within P_{Tact} via the so-called external atom construct.

- *Output data*: *Answer Sets* of $P_{Tact} \cup A_{Tact}$ which contain a dedicated predicate *target* describing the object which has been chosen as a target and some information about the required shot, like the type of trajectory (high or low) and the hitting point (several points on the left and top face of the target can be aimed at).

We describe next the knowledge modelled by P_{Tact}.

A shootable target T is defined as an object for which it exists a direct and unobstructed trajectory from the slingshot to T.
For each shootable target T, we define a measure of the estimated damage that can occur on all other objects if T is hit. The specific behaviour of each bird type is taken into account (e.g. yellow birds are very effective on wood, etc.). Also, the estimated damage takes into account the probability of specific events. The higher the estimated damage function value, the better the target.

Targets are ranked by taking first those which maximize the estimated damage to pigs; estimated damage to other objects is taken into account, on a lower priority basis, only for targets which tie in the estimated damage for pigs.

The *optimal Answer Set*, containing, among its logical consequences the optimal target T_{opt}, is the *Answer Set* maximizing the damage function (see Section 4.4.2). T_{opt} is then passed to the Trajectory Module. This latter module computes the actual ballistic data needed in order to hit T_{opt} (see Section 4.4.1 for more details).

Next, we show some typical assertions used in Angry-HEX; for the sake of simplicity, we report a properly simplified version of the rules, even though the general idea is respected. In the following, with variable names of type $T[_i]$ and $O[_i]$ we refer to trajectory types and objects, respectively.

$$target(O,T) \mid nontgt(O,T) \leftarrow shootable(O,T). \quad \Big\} \textbf{Guess}$$

$$\left.\begin{array}{l} \leftarrow \ target(O_1,\_),\ target(O_2,\_),\ O_1 \neq O_2. \\ \leftarrow \ target(\_,T_1),\ target(\_,T_2),\ T_1 \neq T_2. \\ target\_exists \ \leftarrow \ target(\_,\_). \\ \leftarrow \ not\ target\_exists. \end{array}\right\} \textbf{Check}$$

Intuitively, the first rule expresses that each shootable object can be possibly aimed, while the constraints (the “check” part) ensure that exactly one target is chosen.

Strategy layer. Upon completion of a level, the Strategy layer decides which level should be played next. Like for the Tactics layer we pursued a declarative *ASP* approach for the implementation of the Strategy layer. This module is modelled by means of an *ASP* program, and the next level to be played is conveniently extracted from its logical consequences. This approach significantly improves the management of this layer w.r.t. the previous version of the Angry-HEX agent, where the Strategy was hard-wired in *Java*. The *ASP* program P_{Strat} contains appropriate modelling of the following guidelines on the choice of the next level (here order reflects priority).

1. Play each level once.
2. Play levels for which the gap between our agent's score and the current best score is maximal (up to a limited number of attempts k).
3. Play levels where Angry-HEX outperforms all other agents, but its score minimally differs from the second best result (up to a number of attempts k').
4. If none of the above rules is applicable, play a random level.

The program P_{Strat} has several pieces of input data available, reflecting the history of the game w.r.t. the played levels and scores achieved. Moreover, for each level, the Strategy layer keeps track of previously selected target objects and, as mentioned, ensures the avoidance of repetition of the same opening shot on a particular level, thus allowing multiple approaches at solving the same level.

For example, the encoding of the first guideline in the *ASP* environment is:

$$
\begin{aligned}
r_1: \quad & \mathit{chooselevel}(1) \leftarrow \mathit{timeslevelplayed}(1,0), \mathit{myscore}(1,0). \\
r_2: \quad & \mathit{chooselevel}(X) \leftarrow \mathit{timeslevelplayed}(X,0), \mathit{myscore}(X,0), \\
& \qquad \mathit{timeslevelplayed}(Y,Z), \#\mathit{succ}(Y,X), Z \geq 1.
\end{aligned}
$$

Rule r_1 schedules Level 1 at the beginning of the game. The rule r_2 states that if a Level X has not been yet played (represented by predicates $\mathit{timeslevelplayed}(X,0)$ and $\mathit{myscore}(X,0)$), X comes next after Y (predicate $\#\mathit{succ}(Y,X)$), and Y has been played more than once ($\mathit{timeslevelplayed}(Y,Z)$, $Z \geq 1$), then we choose the level X as the next one to be scheduled ($\mathit{chooselevel}(X)$). For instance, if the facts $\mathit{timeslevelplayed}(4,0)$, $\mathit{myscore}(4,0)$, $\mathit{timeslevelplayed}(3,1)$, are available due to the rules described above, the fact $\mathit{chooselevel}(4)$ will be deducted, and then Level 4 will be scheduled.

4.4.2 Reasoning with Physics-Based Simulation

The "simulation side" allows accessing and manipulating information typically not tailored to being dealt with a logic-based decision support system; in particular, we employ external atoms constructs to perform physics simulations and spatial preprocessing that help us to decide where to shoot. This way, the actual physics simulation and the numeric computations are hidden in the external atoms. The external atoms summarize the results in a discrete form.

Given a current level state, external processing is used for several tasks such as: (i) determine if an object is *stable* (i.e., prone to an easy fall); (ii) determine whether an object B will fall when object A falls due to a structural collapse, or if it can be *pushed* (i.e., A can make B fall by domino effect); (iii) determine which objects intersect with a given trajectory of a bird, and in which sequence; (iv) determine if an object O is *shootable* (i.e., there exist a trajectory with O as the first intersecting object); (v) find the best trajectory for a White Bird (white birds have a peculiar behaviour and require a special treatment).

In the following, we present a more detailed description of the simulation information we used. The data coming from the simulation side is fed into the decision-making side as input assertions and by means of external atoms. Input assertions approximately encode statical information which is available a priori, like the current position of objects. External atoms elaborate and produce information triggered by the decision-making side, like running a physics simulation and providing its outcome in terms of the number of falling objects, etc.

Input assertions. The input assertions, in the form of logical facts, encode information about the position of objects in a scene and data needed for trajectory prediction, such as:

$birdType(BT)$. The type of bird that is currently on the slingshot.

$slingshot(X, Y, W, H)$. The size and position of the slingshot, used for trajectory prediction.

$velocity(X)$. The current velocity scale. This is a value used internally by the trajectory prediction module.

$object(O, M, X, Y, W, H, A)$. There is one assertion of this kind for each object in the scene. Objects are enumerated for unique identification with ID O. Material M can be any of $ice, wood, stone, pig, ground$. Location of centroid (X, Y) of the rotated rectangle denotes the position of the object, width W and height H denote its size, and angle A denotes the rotation of the object at hand.

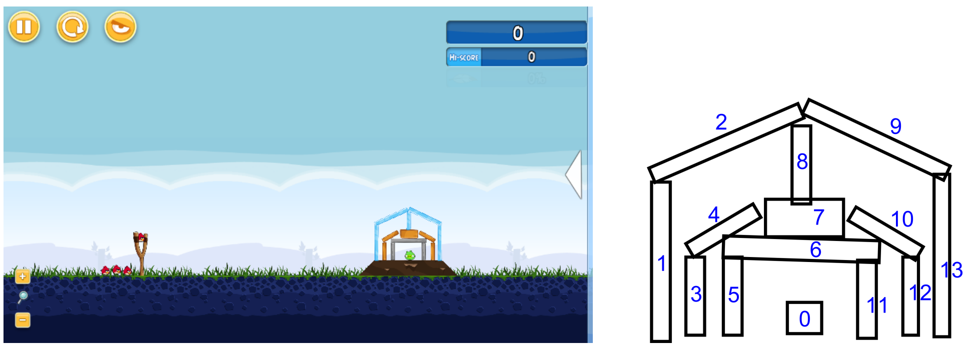


Figure 4.4.: An example of a level (#4 from the Theme One of the "Poached Eggs" set) and the corresponding reconstruction made by our external atoms using Box2D.

External Atoms. The following information sources available to the HEX *program* are implemented as external atoms.

All external atoms are implemented using the physics software Box2D, which has been chosen for being a well documented, supported and regularly updated 2D physics engine. Box2D is widely used for game development and indeed the same *Angry Birds* game uses the library.

All objects ($objs$) are added to a 2D World that we will call W. A simulation is then started on W, and the world is allowed to settle (as shown in Fig. 4.4). Usually, there are small gaps between the objects, because of vision inaccuracies. Gaps need to be explicitly dealt with, since it is desirable that an object should still be detected as resting on the other object even if there is a one-pixel gap: we, therefore, let the physics software proceed up to equilibrium (i.e. until all the objects are at rest). After every time step of the simulation we set all directional and angular velocities to zero, to avoid objects gaining speed and crashing buildings, and in order to keep the settled world as close to the original as possible. Let W^* be the 2D scene computed from W as above. In the following, we assume that the mentioned external atoms implicitly operate on W^*.

$\&on\_top\_all[objs](O_u, O_l)$

Allows to browse the set of couples of objects O_u, O_l for which O_u lies on top of object O_l. This information is used assuming that if O_l does not exist, O_u would likely fall. This is determined by checking whether object O_u exerts a force on O_l , that is oriented downwards. If so, we assume that O_u rests on O_l. In order to improve performance, a graph of object dependencies is calculated on the first call to this atom and, subsequently, cached answers are served.

$\&next[D, T_O, T_j, V, S_x, S_y, S_w, S_h, objs](I, O)$

For a bird trajectory aimed at object D, of type T_j, $\&next$ allows inspecting which objects are intersected by such a trajectory. T_j can either be $high$ or low (see Figure 4.5). V, S_x, S_y, S_w, S_h are helper variables required by the traject-

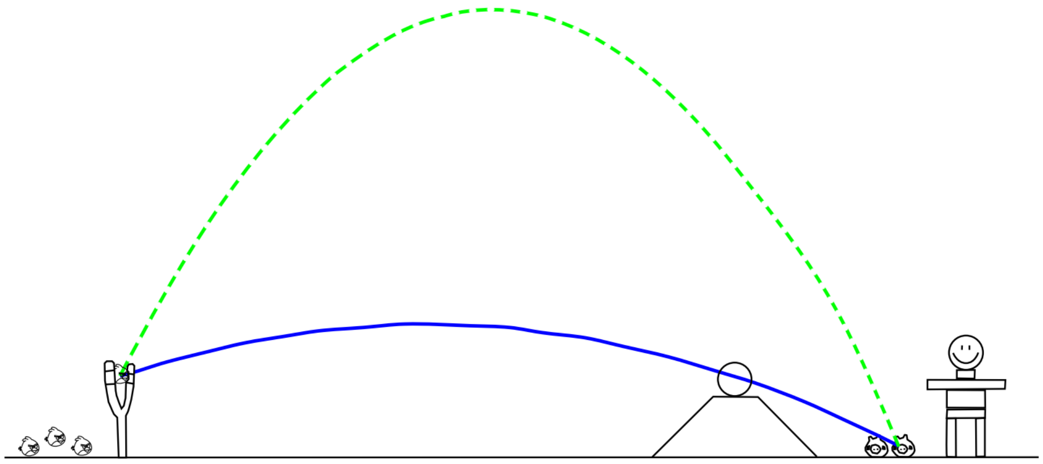


Figure 4.5.: An example of low and high trajectories (solid and dashed line respectively).

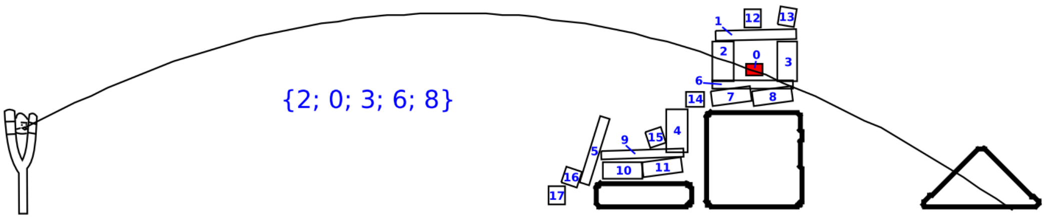


Figure 4.6.: An example of the output from the &next atom.

ory prediction module. V is the velocity scale, available from the $velocity(\dots)$ atom and S_x, S_y, S_w, S_h are the slingshot position and dimension values, available from the atom $slingshot(\dots)$. I is the position of the object O in the sequence of objects that would be hit in the trajectory T_j (e.g. in Figure 4.6 the object $\#2$ has position 1 and so on). The offset T_O allows choosing from a set possible hitting points on the exposed faces of O.

$\&shootable[O, T_j, V, S_x, S_y, S_w, S_h, B, objs](O, S, U)$
This statement is true if object O is shootable with trajectory type T_j, i.e. if there exists a parabola whose O is the first intersected object from left to right. Most of the terms have the same meaning of ones in the $next$ atom. B identifies the bird type, which could be one of red, yellow, blue, black, white, for the thickness of a bird is used when determining shootability. S and U are the best offsets positions over O faces for the given trajectory and the given bird type.

$\&firstbelow[P, objs](O)$
Denotes that the object with ID O is directly below the object P, with no items in between. We calculate this similarly to the $\&next$ atom, but instead of a trajectory, we use a straight upward ray from O to P. The statement holds only if P is the first intersected object by such a ray.

$\&stability[W, H](S)$

Denotes the stability S of an object given its parameters W, H ($width$ and $height$ respectively). S is the ratio $(width/height) \times 50$ rounded to the nearest integer or 100 if it is greater than 100.

$\&canpush[objs](O_A, O_B)$

For each object O_A, selects all the objects O_B that can be pushed by O_A by a left-to-right domino fall. The canpush relation between an object O_A an O_B is computed by geometrically checking whether O_A, if rotated of 90 degrees rightwards, would overlap the O_B extension.

$\&clearsky[O, objs]$

This atom is specific for white birds and identifies whether the object O can be hit by the egg of a white bird. That is, whether there is enough space above O to let a White Bird to vertically release its egg on the object.

$\&bestwhite[O, T_j, V, S_x, S_y, S_w, S_h, objs](Y)$

Again, with specific focus on White Birds behaviour, this atom returns the best height Y above the object O where to shoot, with trajectory type T_j, in order to achieve a good shot. A shoot with a White Bird on object O is considered optimal if it actually hits O and maximizes the damage effects of the "departure at 45 degrees" of the white bird in order to hit other objects. The other terms have the same meaning as in the $\&next$ atom.

The following are some examples of logic rules featuring external atoms; pushDamage intuitively describes the likelihood of damage when an object Obj_B is "pushed" by an adjacent object Obj_A:

$$\begin{aligned} pushDamage(Obj_B, P_A, P_B) \leftarrow\ & pushDamage(Obj_A, \_, P_A), P_A > 0, \\ & \&canpush[ngobject](Obj_A, Obj_B), \\ & pushability(Obj_B, Pu_B), P = P_A * Pu_B/100. \end{aligned} \tag{4.1}$$

$\&canpush$ works as described above, and allows determining whether Obj_A can make Obj_B fall by means of a *domino effect*. It is worth noticing that $\&canpush$, as well as other atoms, uses geometric computations in a continuous space, however, the values returned are discrete, in this particular case the result of the atom evaluation corresponds to the truth of a set of propositional atoms. P_B is a damage value estimate expressed as an integer value ranging from 0 to 100, and obtained as the product between the push damage P_A of Obj_A and the *pushability* Pu_B of Obj_B, normalized in the integer range $0, \ldots, 100$. The *pushability* value for an object is defined relying on empirical knowledge of the game, and defines how much an object can be *pushed* in terms of its shape, stability and material (e.g. long rods are easily pushable, etc.).

Another example follows.

$$eggShootable(Obj, X) \leftarrow \&clearsky[Obj, objects](), \quad ngobject(Obj, \_, X, \_, \_, \_, \_). \tag{4.2}$$

The above rule checks if an object that is not the scene ground surface ($ngobject$) can be hit by the egg released by a White Bird. Again, like in rule 4.1, the computation in the continuous space is entirely performed by the external atom.

The estimated damage function

The aim of the reasoning engine is to find the "best" object to shot, the most appropriate tap time and "how" the object should be hit (i.e., where to aim – to the centre of the object, to a long or a short side, etc.). In order to identify the best target object, we attribute to each possible target a score Sc_1 based on the sum of damage likelihood for each pig and TNT box in the scene, and a score Sc_2 based on the sum of damages of other objects. We select the target that maximizes Sc_1, or, in case of a tie, we maximize Sc_2.

In turn, per each object, we attribute several damage type quotas. In general, all the damage types are computed in terms of causal event chains, in which damage is linearly weighted by the likelihood of the event causing the damage. The likelihood of an event is in turn obtained by the product of fixed empirical values combined with the likelihood of previous events in the causality chain. Damage types are described next.

direct damage: the damage an object takes when hit by a bird. Direct damage is calculated by examining the sequence of objects that intersect the assumed trajectory using the $\&next$ atom. This type of damage depends on the intrinsic *damage probability* P of each object and on the *energy loss* E of the bird (the farther an object is in the intersected object list the lesser its direct damage value). The following is an example of a rule to compute the direct damage:

$$directDamage(Obj, P, E) \leftarrow target(Inner, Tr), next(Obj, 0, Outer, T, \_), objectType(Obj, T), birdType(Bird), damageProbability(Bird, T, P), energyLoss(Bird, T, E). \tag{4.3}$$

The $next$ atom summarizes the external $\&next$ by means of rules like the following:

$$next(X, Y, Z, T, C) \leftarrow shootable(X, T, C, \_), slingshot(Sx, Sy, Sw, Sh), \&next[X, C, T, V, Sx, Sy, Sw, Sh, objects](Y, Z), velocity(V), T \neq egg. \tag{4.4}$$

In the above, the truth of an assertion $next(X, Y, Z, T, C)$ can be read as "X is the Y-th object in the trajectory T aiming at object Z, with horizontal shift C from the object's centroid".

push damage: the damage an object undergoes when pushed by another object. It is calculated by building the already mentioned chain of "pushes" and depends on the intrinsic *pushability* of an object and on the values coming from the external atom $\&canpush$. An example illustrating the role of $\&canpush$ is provided in rule (4.1) above.

fall damage/fall desirability: the damage an object undergoes when another object is destroyed below it. It is calculated according to the values of the $\&on\_top\_all$ external atom, and it depends on the intrinsic *material fall importance* P_N of the object and on all other kinds of damages (i.e. a fall damage chain can be started by a direct or push damage chain). For instance, in order to compute the damage of a "falling" object Obj one can specify the following rule:

$$\begin{aligned} &fallDamage(Obj, P) \leftarrow pushDamage(RemovedObj, \_, P_R), P_R \geq 50, \\ &\qquad \&on\_top\_all[objects](Obj, RemovedObj), objectType(Obj, T), \\ &\qquad materialFallImportance(T, P_N), P = P_R * P_N / 100. \end{aligned} \tag{4.5}$$

Modelling empirical knowledge

A good portion of empirical knowledge of the gameplay is encoded in terms of logical assertions. As opposed to hard-wiring this information into traditional code, this allows better flexibility and easier fine-tuning and troubleshooting.

One type of such empirical knowledge comes into play when static object damage values are combined in order to form causal chains. Causal chains terminate using an energy loss estimate; energy losses take into account the residual energy available when the effects of a shot propagate from an object to another. Also, we model the attitude of a particular bird type towards destroying different material types. For instance, the following assertions encode the damage probability of an object depending on the object material and on the type of bird hitting the object at hand. They correspond to intuitive statements like "Blue birds are very good on ice blocks":

$$\begin{aligned} &damageProbability(blue, wood, 10).\\ &damageProbability(yellow, wood, 100).\\ &damageProbability(blue, ice, 100).\\ &damageProbability(yellow, ice, 10).\end{aligned}$$

Other empirical knowledge includes the following:

damage probability for each couple *(Bird_Type, Material)* we encode the damage an object made of *Material* receives when hit by *Bird_Type*.

energy loss for each couple *(Bird_Type, Material)*, we encode the reduction of energy a bird of *Bird_Type* experiences when it destroys an object made of *Material*.

pushability it denotes how "easy" is an object made of a given material to be pushed, when other conditions and features (i.e., shape) are fixed. For instance, stones react less to pushes than wood.

material fall importance the damage an object made of a given material can cause when it falls, under other conditions equal, like size. For instance, stones are assumed to have greater density.

A second group of empirical gameplay information comes into play when dealing with trajectory prediction. The Trajectory prediction module, given in input some target coordinates, considers several "high" (aiming at a vertical fall to an object) and several "low" trajectories (aiming at a horizontal hit on the left-hand face of an object) but returns only two of both categories. Many candidate trajectories are discarded because of obstructions before the target point. This discretization is done in order to reduce the space of possibilities which the reasoning module has to take decisions on. This approximation can be considered acceptable, and we indeed did not experiment appreciable differences in the effects of two different parabolas of the same category.

Trajectory prediction is treated differently depending on the type of bird that is on the slingshot at reasoning time. For what red, yellow, blue and black birds are concerned, we use a normal "parabolic" trajectory. This trajectory aims at the objects' left face if a *low* trajectory is selected while the top face is aimed at if a *high* trajectory is chosen. Three hitting points are possible for each face, for a total of 12 possible trajectories per object.

As for tapping time, we tap yellow birds relatively close to the target object, so to maximize their acceleration effect without compromising the parabolic trajectory; blue birds are tapped a bit earlier in order to maximize their "spread" effect, while black birds are tapped right on target so to maximize their explosion effect.

The white bird receives special treatment: we first try to identify which objects can be hit by the vertical fall of the *egg* that a white bird can release with a tap. We then choose the best point where the bird should release the egg itself, in terms of side effects, since the white bird continues its trajectory after laying its egg, thus creating more damage chains.

4.4.3 Results/Benchmarks

In this section we discuss experiments and performance.

Competition outcomes and Third party benchmarks[20]

Angry-HEX performed quite well in all the editions of the *Angry Birds AI Competition*, from 2013 to 2017, even if we never ranked first. Our team participated also in 2012[21], but with two preliminary agents, that were also largely different from the herein presented Angry-HEX agent. Our agent reached the *Semi Final* in 2013[22] (being the best one during all previous rounds), the *Quarter Finals* in 2014[23], the *Grand Final* in 2015[24] (achieving the second place[25]), the *Quarter Finals* in 2016[26] (due to a bug caused our agent to kept solving the same levels it had already solved rather than trying to solve levels it had not solved yet), the *Semi Final* in 2017[27] (achieving the third place).

The Organizing Committee performed also some benchmarks over the participating agents in the same settings of the Competition, in order to allow participants to easily compare the performance of their agents with others. The benchmarks were run on the first 21 levels of the freely available Poached Eggs levels, and each agent had a time budget of 63 minutes, corresponding to an average of 3 minutes per level. They stored them in some tables, where bold numbers represent high-scores of each level among all participants.

In the after-show Benchmarks of 2013[28], Angry-HEX performed better than all other participants. Our score was very good in many levels (reaching the 3 stars score), even if only at one level we had the best score among all the participants. It is worth noticing that the winning team of 2013 (*Beau Rivage*) arrived only forth in these Benchmarks and that the four best team of the Competition are in the first four positions of the Benchmarks ranking (even if in a completely inverted order).

[20]The Competition Results as well as the Benchmark Results are fully reported in Appendix B
[21]`http://ai2012.web.cse.unsw.edu.au/abc.html`
[22]`http://aibirds.org/past-competitions/2013-competition/results.html`
[23]`http://aibirds.org/past-competitions/2014-competition/results.html`
[24]`http://aibirds.org/past-competitions/2015-competition/results.html`
[25]Interestingly, our agent of 2015 was the same as the one of 2014 but, as the organizers wrote:
“*The **Grand Final** between DataLab Birds and the **runner up AngryHex from Italy and Austria** was actually the most exciting AIBIRDS match ever. The leaderboard changed almost every minute, sometimes every few seconds. Even though the levels were very difficult, both AI agents did a great job in solving them, some with incredible shots. Datalab Birds solved all 8 levels, while AngryHex solved only 7 out of 8 levels, but with higher scores. Both agents were solving levels again and again in order to improve their overall score. Fortunately, both teams were not attending personally, otherwise, there would have surely been one or two heart attacks among the team members.*”
[26]`http://aibirds.org/past-competitions/2016-competition/competition-results.html`
[27]`http://aibirds.org/angry-birds-ai-competition/competition-results.html`
[28]`http://aibirds.org/past-competitions/2013-competition/benchmarks.html`

It is also important to notice that no agent was able to perform better than others in all the levels, and also that best scores of each level was done mostly by the agents in the centre/bottom part of the ranking.

In the after-show Benchmarks of 2014[29], Angry-HEX performed better than most of the other participants that had outperformed it in the Quarter Finals[30]. In some levels we performed similarly to 2013 (usually with a slightly lower score); however, in some other, we performed much better. Similarly to what happened in the previous edition, in 2014 no agent was able to perform better than others in all the levels, and also this year the best scores of each level was done mostly by the agents in the centre/bottom part of the classification. The fact that the scores in each level are typically distributed over many participants that are listed below Angry-HEX in rankings might hint that the strategies implemented in other agents are tailored to specific types of levels, but generalize less, or worse, than the one of Angry-HEX.

Moreover, in 2014 the organizers tested also the agents (of 2014) on the second Theme of Poached Eggs (the following 21 levels). This benchmark is quite interesting because these levels are more difficult than the previous ones and they contain also some objects that are not allowed in the Competition and therefore the *Vision Module* is not able to recognize. Angry-HEX lost a position w.r.t. the results in the first 21 levels but it was still better than other agents that had outperformed it in the Competition (for instance of the *IHSEV* agent that ranked fourth in the Competition). It is also quite surprising how well the *Naïve Agent* performed on these levels, it ranked right after Angry-HEX, with a Total Score almost three times as high as the last classified agent.

In the after-show Benchmarks of 2016[31] that contained also the agents of the 2015 Competition (as well as all the previous ones), we got curious results. The Angry-HEX version of 2015 (that was exactly the same of 2014), had exactly the same score in some levels but in a couple of them got a much lower score w.r.t the previous version and we ranked much lower. The Angry-HEX version of 2016, however, performed quite better than the previous years in some levels and slightly worse in others but it was not able to solve two levels and for this reason, its rank was worse[32]. It is worth noticing that the winner of 2016 (*BamBirds*, the *AIBIRDS* Champion at the time) only ranked #13, but did solve all 21 levels (as all the version of Angry-HEX did apart from the one of 2016 that missed a couple of them). As also the organizers observed, it is clear that solving more levels is better than having

[29] `http://aibirds.org/past-competitions/2014-competition/benchmarks.html`

[30] Teams with names in bold participated in the 2014 Competition, the others in the 2013 Competition

[31] `http://aibirds.org/benchmarks.html` (this *URL* will probably change in the next years, following a similar structure to the other links to Benchmark Results)

[32] It is worth noticing that if we had solved them, even with the lowest score we had among all the years, we would have exceeded the score of 2013

high scores and that doing well in the Benchmarks does not necessarily translate to success in the Competition, where all levels are unknown.

A proper deeper analysis of the levels where Angry-HEX was not the best in these years could be a fruitful approach for defining special cases in its Tactics. It is worth to notice that in the latest Benchmark, in all the first 21 levels of Poached Eggs, at least one agent reached a 3-star score, that represent a very good score to achieve for a human player. However, to date, the "Man vs Machine Challenge", in which each year human players compete with the winners of the *Angry Birds AI Competitions*, was always won by human players.

Eventually, it is important to notice that the differences between our results in these years are very little, and are mostly due to a large restructuring of the agent code, which has been performed in 2014 and in 2016 in order to make it more extensible, flexible and easy to install. Therefore, we expect to take advantage of this work in the forthcoming Competition editions.

Tests on time performance

The Tactics layer is repeatedly prompted for deciding the chosen target on the current scene: reducing reasoning times is crucial in order to better exploit the allowed time and improving scores. We recall that the core of the Tactics layer is an evaluation machinery carried over the logical *Knowledge Base* P_{Tact}, coupled with a set of assertions A_{Tact} that describe the current scene; hence, both the size of A_{Tact} and P_{Tact} affects its performance. Another important performance factor is the number and duration of calls to external libraries.

The size of A_{Tact} is directly proportional to the number of objects in the current scene: there is one logical statement for each object in the scene, plus a constant number of facts encoding the slingshot position, the current bird type, and the scale factor of the scene. P_{Tact}, instead, is a fixed *Knowledge Base* featuring about three hundred statements, made both of rules and facts that encode the domain knowledge. For what calls to external libraries are concerned, these were optimized with the aim of reducing the number of possibly redundant computations.[33]

Due to the unavailability to the public of the levels used in the official *Angry Birds AI Competition*, our comparative studies of actual performance are limited to the publicly known first 21 levels of the "Poached Eggs" level set,[34] though they do not explicitly stress reasoning capabilities of artificial agents. The experiments were

[33]Furthermore, the burden of repeated identical calls to external sources was mitigated by caching.
[34]"Poached Eggs" is the first level set available in the publicly downloadable version of the game, which reflects the Competition setting in qualitative terms (type of materials, shape of objects and type of birds available), and are customarily used by the organizers in the after-show benchmarks

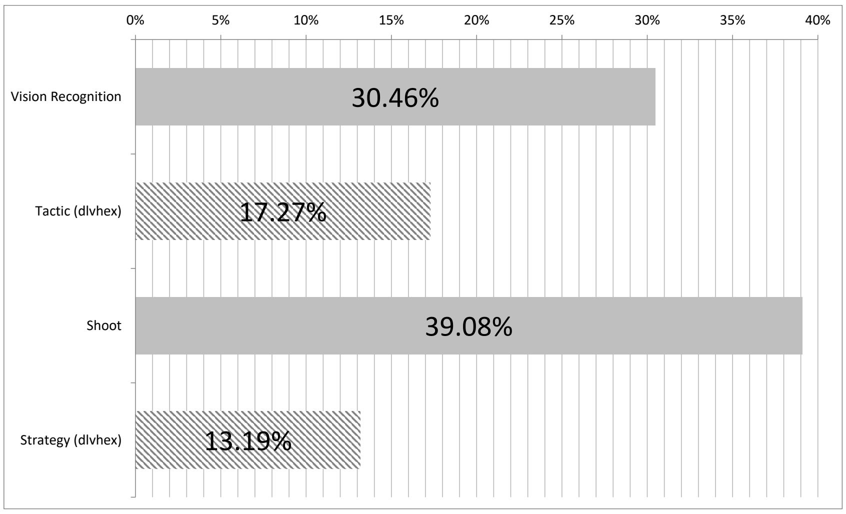


Figure 4.7.: The average percentage of time spent in each "main task" by our agent. Solid bars account for tasks common to all the agents (i.e. tasks performed by using functionalities provided by the organizer's framework); slanted lines bars account for specific tasks of our agent, performed using our reasoning modules.

conducted on a Virtual Machine running Ubuntu 14.04.2 LTS, containing 2-cores of a 2.29 GHz Quad-core Processor and 3 GB of RAM, running standalone in its hypervisor. First, we noted that the reasoning time was a small fraction of the overall time spent for each shot; indeed, most of the time is spent by animations and the simulation of the shot itself, as shown in Figure 4.7.

Figure 4.8, instead, depicts the number of objects in the scene of a level against the time spent by the reasoning tasks within the Tactics layer, on that level. It is worth noting that the number of objects in a level is not a direct measure of "how hard" is a level from a player perspective: this depends on the materials, the shapes, the (relative) positions, the way pigs are sheltered, and so on. The number of objects in a scene is however proportional to the size of data given in input to the Tactics layer. We found some direct proportionality between time and the number of objects in a scene, but, interestingly, the system seems to scale fairly well w.r.t. the latter. The fraction of reasoning time dedicated to external atom calls did not show a clear trend: it averaged around 61% of the total reasoning time, with low and high peaks of 30% and 90% respectively.

We also focused in measuring the performance of the Tactics layer when changing Tactics *Knowledge Bases*. In order to compare the impact on time performance, when changing Tactics, we measured the reasoning time of three, incrementally better in terms of gameplay, different reasoning *Knowledge Bases*. The cactus plot in Figure 4.9 shows the performance of the Tactics layer (i.e. the time taken to take a decision given an input scene already processed by the *Vision Module*), for all the runs needed for completing the Poached Eggs levels (54 shots in total); in order to

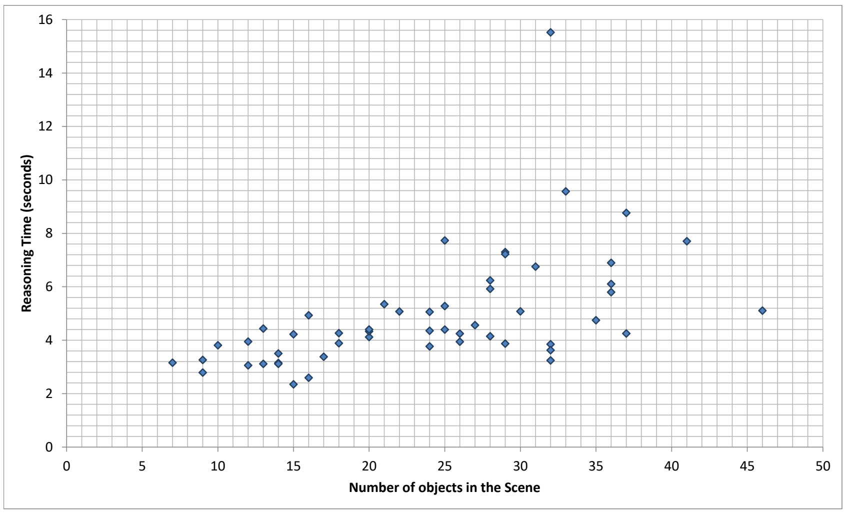


Figure 4.8.: The time spent by the Tactics layer w.r.t. the number of objects in the scene.

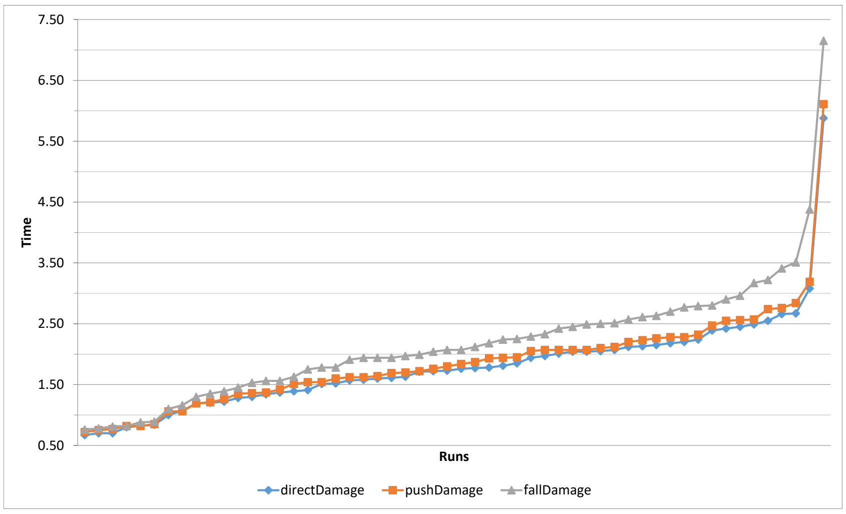


Figure 4.9.: Reasoning times for different tactics *Knowledge Bases*, measured on the first 21 levels of “Poached Eggs”.

give a clearer glimpse at the resulting trends the data series were sorted by execution time. We experimented with three different tactics: *directDamage* (a *Knowledge Base* of 133 logic assertions), in which only estimated damage by direct hits is maximized; *pushDamage* (143 assertions), in which we add the potential damage of domino effect, and *fallDamage* (155 assertions), which corresponds to the Tactics participating to the Competition, in which we add also damage due by the vertical fall of objects. It is easy to see that the time differences are not substantial, especially if compared to the time cost of other evaluation quotas (vision, gameplay, etc.) with *fallDamage* clearly asking for a somewhat larger price in terms of time consumption. We can conclude that, when designing *Knowledge Bases*, a fairly large degree of freedom in designing sophisticated tactics can be enjoyed without being worried by a loss in time performance.

External calls were a key-point for efficiency: the strategy of outsourcing several tasks outside the decision-making core proved to be useful. As a matter of fact, the performance of one earlier attempt of implementing an *ASP*-only agent capable of playing a real-time game was far from satisfactory [460], given the fairly limited efficiency of *ASP* solvers; we believe that implementing this *AI* player using *ASP* (with no external calls and/or a "process" stage), if feasible at all, would not have fit with the real-time requirements of the game.

4.4.4 Discussion

The *Angry Birds* game, and hence the study, design and implementation of this work, led to face several challenges for *Knowledge Representation and Reasoning*, and *Artificial Intelligence* in general; eventually, we can make some considerations. First of all, it looks clear that, in order to accomplish complex jobs/tasks, a monolithic approach should be discarded in favour of more diversified ones, consisting of convenient integrations of various, if not many, methods and technologies. In this respect, any bundle of *KR&R* formalism/system of use, besides expressive power, suitable syntax/semantics and good efficiency/performance, must feature proper means for easing such integration at all levels, from within the language to the actual implementation.

The setting of the *Angry Birds* game has two particular aspects: first, *Angry Birds* can be seen as a game which lies somehow between a real-time game and a turn-based one, allowing a fairly large time window for deciding the next shot; second, at each step, the Tactics layer must explore a search tree whose size is reasonably small and polynomially proportional to the number of objects in the current scene.

The above setting can be lifted to a variety of other contexts in which *(a)* there is an acceptable time window available for reasoning, and *(b)* the set of actions that can be taken by the agent at hand is fairly small, do not underly an exponentially larger search space, and no look-ahead of arbitrary depth on future scenarios is required (or convenient) to be performed. This generalized context covers e.g. planning in slowly evolving environments (e.g. robot and space probes), automated room cleaning, semi-interactive turn-based games, etc.

In this respect, the Angry-HEX agent constitutes a good proof-of-concept showing how *ASP*, if properly deployed in a hybrid architecture and extended with procedural aids, not only qualifies as an effective tool for implementing near real-time solutions while enjoying the advantages of declarative approaches but witnesses that the realizability of real-time applications is much closer.

The work carried out by the scientific community in the latest years and the effective use of *ASP* in real-world and industry-level applications [364], suggest *Answer Set Programming* as a powerful tool in such scenarios; and the present work confirms this idea. Moreover, performances do not constitute a big issue, as discussed in Section 4.4.3.

The Angry-HEX agent has been released as *open-source software* (*OSS*), under the *GNU Affero General Public License*[35], and it is publicly available at

`https://github.com/DeMaCS-UNICAL/Angry-HEX`

Moreover we provided a webpage[36], with useful information about our agent.

It is worth noticing that other people have already taken advantage of the possibility of freely study, use, distribute and even improve the Angry-HEX project.
In this respect, recently, the authors of [159] proposed an agent, called *DualHEX*, which is based on the Angry-HEX agent and extends it considering not only the current bird while selecting a target but also the next one. To do this it compares the damage probability of an object for the current bird and the next one, if the next bird can cause more damage on the target than the current one, the target is discarded. The authors believe that this small improvement will minimize the *Answer Sets* such that it gets closer to people choices and, thus, allow the agent to achieve good scores.[37]

As for the original goal of Angry-HEX, even though from the benchmarks and the results of the competitions our approach seems quite effective and general, we further identified several aspects in which Angry-HEX can be improved. Most importantly, we aim at introducing the planning of multiple shots based on the order of birds that must be shot (it is worth remembering that the type and number of birds, as well as the order the player have to shoot them, is given at the beginning of each level); we think that this might prove to be useful especially when dealing with complex levels. A more accurate study of the interaction between objects, and a more detailed implementation of the different shapes of the objects are also under consideration. Clearly, we aim to not affect much reasoning time when introducing these improvements.

[35]`https://github.com/DeMaCS-UNICAL/Angry-HEX/blob/master/LICENSE`
[36]`https://demacs-unical.github.io/Angry-HEX`
[37]From [159]

4.5 Other Game's AIs experiments

In this section, we present some experiments related to Games and *Artificial Intelligence* conducted in order to investigate further ideas and strategies in alternative environments.

4.5.1 A full native *ASP*-based *Android* App: *GuessAndCheckers*[38]

GuessAndCheckers is a native mobile application that works as a helper for players of a "live" game of the Italian checkers (i.e., with a physical board and pieces). By means of the device camera, a picture of the board is taken, and the information about the current status of the game is translated into facts that, thanks to an *ASP*-based *Artificial Intelligence* module, make the app suggest a move. The app is well-suited to assess applicability of *ASP* in the mobile context; indeed, while integrating well-established *Android*™ technologies[39], thanks to *ASP*, it features a fully-declarative approach that eases the development, the improvement of different strategies and also experimenting with combinations thereof.

GuessAndCheckers is a native mobile application which has been designed and implemented by means of the specialization of EMBASP for DLV on *Android* discussed in Section 5.2. It works as a helper for users that play "live" games of the Italian checkers (i.e., by means of physical board and pieces); it has been initially developed for educational purposes. The app, that runs on *Android*, can help a player at any time; by means of the device camera a picture of the physical board is taken: the information about the current status of the game is properly inferred from the picture by means of a *Vision Module* which relies on OpenCV[40], an open source computer vision and machine learning software.

A proper *ASP* representation of the state of the board is created, and eventually, an *ASP*-based reasoning module suggests the move. It is worth to remember that the declarative approach to *KR&R* supported by *ASP* it is significantly different from a "classic" algorithmic approach. Basically, besides solid theoretical bases that lead to declarative specifications which are already executable, there is no need for algorithm design or coding, and easy means for explicitly encoding the knowledge of an expert of the domain. We refer the reader to the material available online [131],

[38]From F. Calimeri, D. Fuscà, **S. Germano**, S. Perri and J. Zangari. 'Boosting the Development of ASP-Based Applications in Mobile and General Scenarios'. In: *Proceedings of AI*IA 2016*, pp. 223–236. DOI: 10.1007/978-3-319-49130-1_17.
[39]https://developer.android.com
[40]http://opencv.org

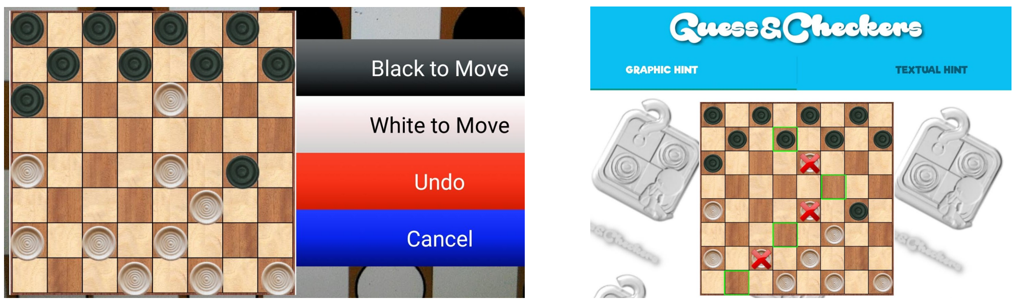


Figure 4.10.: Screenshots from the *GuessAndCheckers* app: recognized board and suggested move for the black.

including full encodings, for more information; in the following, we will illustrate how we mixed them in order to obtain the artificial prompter. The goal was to avoid ad-hoc behaviours for each situation and develop a strategy which turns to be "good" in general. The strategy has been tuned via careful studies and a significant amount of experimental activities. It consists of several rules that can be defined as well-known by a human expert of the domain; they have been evaluated and then filtered out from a large set of hints, thus constituting a real *Knowledge Base* of good practices for real checkers players.

The fully-declarative approach of *ASP* made easy to define and implement such *Knowledge Base*; furthermore, it was possible to develop and improve several different strategies, also experimenting with many combinations thereof.

The reasoning module consists of a Manager, developed in *Java*, that does not have decision-making powers; instead, it is in charge of setting up different *ASP* logic programs (i.e. builds and selects the *ASP* program to be executed at a given time) and makes use of EMBASP to communicate with DLV. At first, an *ASP* program (the *Capturing Program*) is invoked to check if captures are possible; if a possible capturing move is found, the *Capturing Program* provides the cells of the "jump(s)", the kind of captured pieces and the order of the capture; another logic program is then invoked to select the best move (note that each piece can capture many opponent pieces in one step). A graph-based representation of the board is used, along with a set of rules of the Italian checkers:

a) capture the maximum number of pieces
(observing the rule *"If a player is faced with the prospect of choosing which captures to make, the first and foremost rule to obey is to capture the greatest number of pieces"*);
b) capture with a king
(observing the rule *"If a player may capture an equal number of pieces with either a man or king, he must do so with the king"*);

c) if more than one capture are still available, choose the one having the lowest quantity of men pieces
(observing the rule *"If a player may capture an equal number of pieces with a king, in which one or more options contain a number of kings, he must capture the greatest number of kings possible"*)

d) finally, if the previous constraints have not succeeded in filtering out only one capturing, select those captures where a king occurs first
(observing the rule *"If a player may capture an equal number of pieces (each series containing a king) with a king, he must capture wherever the king occurs first"*)

On the other hand, if no captures are possible, a different logic program (the *All Moves Program*) is invoked to find all the legal moves for a player (each *Answer Set*, here, represents a legal move). When more than one move can be performed (i.e., there are no mandatory moves), another logic program (*Best Game Configuration Program*) is invoked. Unlike the other programs, that only implement mandatory actions, the *Best Game Configuration Program* implements the actual strategy of our checkers prompter. The logic rules perform a reasoning task over the chequerboard configurations, and the best one is chosen by means of *weak constraints*.

The strategy of the *Best Game Configuration Program* is mainly based on the well-known strategic rule: *"pieces must move in a compact formation, strongly connected to its own king-row with well-guarded higher pawns"*. More in detail, our reasoner guarantees that the user can move its pawns without overbalancing its formation. Pawns attack moderately, trying to remain as compact as possible, preserving at the same time a good defence of its own king row: technically, a *fully guarded back rank* (roughly speaking, prevent the opponent from getting kings unless she sacrifices some pieces).

The availability of DLV brought to *Android* via EMBASP allowed us to take great advantage from the declarative *KR&R* capabilities of *ASP*; indeed, as already mentioned, we experimented with different logic programs, in order to find a version which plays "smartly", yet spending an acceptable (from the point of view of a typical human player) time while reasoning. We think of the design of the *AI* as the most interesting part of the app; we have been able to "implement" some classic strategies, a task that is typically not straightforward when dealing with the imperative programming, in a rather simple and intuitive way. Moreover, we had the chance to test the *AI* without the need for rebuilding the application each time we made an update, thus observing "on the fly" the impact of changes: this constitutes one of the most interesting features granted by the explicit *Knowledge Representation*.

Even if we know that the English draughts (Checkers) game was (weakly) solved a few years ago [513], i.e. from the standard starting position, perfect play by both sides leads to a draw. But this game has roughly 500 billion billion possible positions (5×10^{20}). The task of solving the game, determining the final result in a game with no mistakes made by either player, is daunting. Since 1989, almost continuously, dozens of computers have been working on solving checkers, applying state-of-the-art *Artificial Intelligence* techniques to the proving process. This is the most challenging popular game to be solved to date, roughly one million times as complex as Connect Four.[41]

Therefore, we were not interested in solving exactly the game or in building a prompter able to suggest the perfect move to the user but in demonstrating how *Logic Programming* can be effectively used to define some well-known *AI* strategies and how easily they can be implemented, extended and tested using the EMBASP framework. For this reason, we first implemented a basic strategy based on the Italian Checkers Federations good practices, and then we improved it, adding look-ahead and finding the best combinations of empirical rules that can lead to "good suggestions". Thanks to the declarative nature of our approach, the modifications of our strategy and the integration of different ideas was straightforward.

We performed an extensive set of tests in order to find the right balance between the "quality" of the suggestions and the time required to compute them, being able to reach a good compromise.

GuessAndCheckers was part of a bachelor thesis work and its source code, along with the *APK* and the benchmark results, are available online at
https://www.mat.unical.it/calimeri/projects/embasp/#Applications

4.5.2 *UniCraft*: a modular StarCraft agent[42]

UniCraft is an artificial agent (a bot) that is able to play the popular *real-time strategy* (*RTS*) game *StarCraft: Brood War*[43]. The aim of the project was to explore different *AI* techniques in the context of *RTS* games.

Real-time strategy (*RTS*) games represent a genre of video games in which players must manage economic tasks like gathering resources or building new bases, increase their military power by researching new technologies and training units, and lead them into battle against their opponent(s). *RTS* games serve as an in-

[41]From [513]
[42]Preliminary definitions adapted from [539]
[43]http://eu.blizzard.com/en-gb/games/sc

teresting domain for *Artificial Intelligence* (*AI*) research, since they represent well-defined complex adversarial systems [113] which pose a number of interesting *AI* challenges in the areas of planning, learning, spatial/temporal reasoning, domain knowledge exploitation, task decomposition and dealing with uncertainty [454]. One of the main sources of uncertainty in *RTS* games is their adversarial nature. Since players do not possess an exact knowledge about the actions that their opponent will execute, they need to build a reasonable representation of possible alternatives and their likelihood. Various problems related to uncertainty and opponent behaviour prediction in *RTS* games have already been addressed in recent years [454].

StarCraft is a *real-time strategy* video game developed by Blizzard Entertainment in 1998 that still has great global following. *StarCraft: Brood War* is an expansion set for *StarCraft* released in the same year (from this moment, every time *StarCraft* will be mentioned, we will refer to such expansion). In *StarCraft*, there are three playable races: *Terran*, *Zerg* and *Protoss*. Each race has unique properties and abilities. There are different game modes. We mainly considered the 1vs1 mode in which a player (human or bot) fights against another player (human or bot). The aim of the game is the expansion of the own colony and the destruction of the opponent one.

As mentioned above, there are many competitions related to the *StarCraft* game and a huge quantity of research papers have been written out about this game[44]. Moreover, a free and open source framework, the *Brood War Application Programming Interface* (*BWAPI*)[45], to interact with the game has been developed during the years by many people, and also some universities were involved in it[46]. *BWAPI* is used in many different competitions[47] and allows users to create *AI* agents that play the game. *BWAPI* only reveals the visible parts of the game state to *AI* modules by default. Information on units that have gone back into the fog of war is denied to the *AI*. This enables programmers to write competitive non-cheating *AI*s that must plan and operate under partial information conditions. *BWAPI* also denies user input by default, ensuring the user cannot take control of game units while the *AI* is playing.[48]

During the *UniCraft* project we explored some popular *AI* techniques and performed many experiments in order to understand well how they can interact and we envisaged the idea of combining *Logic Programming* with these techniques. In order to be able to plug and combine different techniques we first developed a general *Uni-*

[44] https://github.com/bwapi/bwapi/wiki/Academics#papersdissertationsreportsprojectsetc
[45] http://bwapi.github.io
[46] https://github.com/bwapi/bwapi/wiki/Academics#universities-involved-with-bwapi
[47] http://bwapi.github.io/#competition
[48] From http://bwapi.github.io

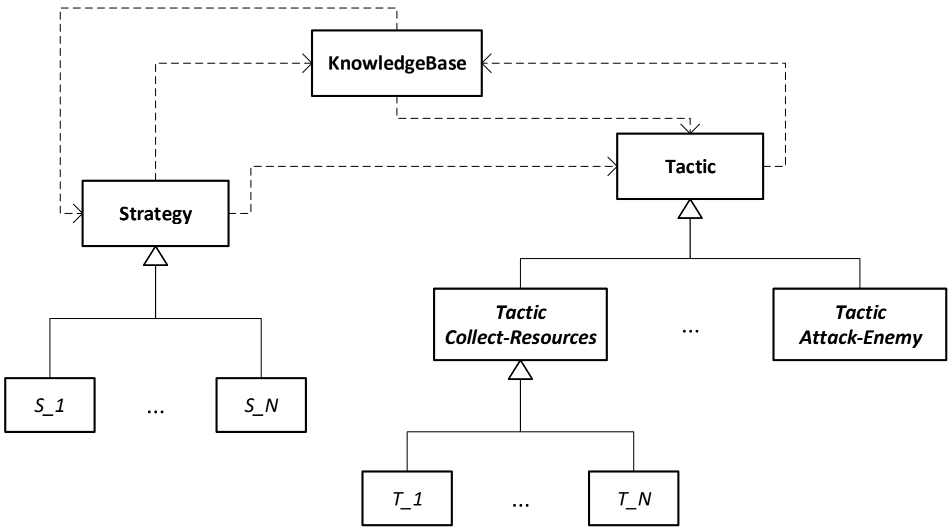


Figure 4.11.: Schematic Architecture of the *UniCraft* bot.

Craft bot, structuring it in a modular way using an architecture based on a similar idea to the one of Angry-HEX.

We divided the agent into a *Tactical* part and a *Strategic* part and both of them have access to a *Knowledge Base*. The *Strategic* part is the brain of *UniCraft*. It decides what to do in every game frame. The *Tactical* part decides how to perform a specific task (like collecting resources, constructing buildings, creating or updating units, etc.). The *Knowledge Base* stores some game information that *BWAPI* does not provide (like the exact number of a specific kind of unit, etc.).

They are built in such a way they are independently and they can be easily extended, so it is very straightforward to add new behaviours, as can be grasped by the Architecture of the *UniCraft* bot, shown in schematic form in Figure 4.11.

We examined some *AI* techniques that are popular and successful in the video games' context, like *Behavior Tree* (*BT*) and *Goal-Oriented Action Planning* (*GOAP*).

As mentioned in [153], a *Behavior Tree* (*BT*) is a way to structure the switching between different tasks[49] in an autonomous agent, such as a robot or a virtual entity in a computer game. *BT*s are a very efficient way of creating complex systems that are both modular and reactive. These properties are crucial in many applications, which has led to the spread of *BT* from computer game programming to many branches of *AI* and Robotics. They have been extensively used in high-profile video games such as Halo, Bioshock and Spore.

[49]assuming that an activity can somehow be broken down into reusable sub-activities called tasks sometimes also denoted actions or control modes

As mentioned in [458], *Goal-Oriented Action Planning* (*GOAP*) is a decision-making architecture that takes the next step and allows characters to decide not only what to do, but how to do it. A character that formulates his own plan to satisfy his goals exhibits less repetitive, predictable behaviour, and can adapt his actions to custom fit his current situation. In addition, the structured nature of a *GOAP* architecture facilitates authoring, maintaining, and re-using behaviours. Many popular games architectures are based on *GOAP*, such as F.E.A.R., Fallout 3 and Deus Ex: Human Revolution.

We compared and implemented these techniques and we performed some tests in order to evaluate our bot.

UniCraft was part of a bachelor thesis work and its source code, along with the implementations of some techniques used and some examples of the *AI* modules developed, are available online at
https://github.com/ayfrank/UniCraft

4.5.3 Discussion

These experiments gave us many insights on the relation between *AI* and games and allowed us to learn that often the combination of different techniques can lead to very interesting achievements.

We saw how a fully-declarative approach can be useful to define different strategies and easily utilize them, and how a well-defined structure can help to test multiple techniques in a straightforward way. This allows to have prototypes quickly but also to try various ideas and to compare and combine them effortlessly.

Moreover, they show that frameworks are often useful in order to simplify the development of *AI*-based solutions. The development of such a base working platform for intelligent agents is one of the goals of the Angry-HEX project, in particular the creation of a modular architecture, based on the popular Tactics/Strategy division of responsibilities, which should allow anyone who would like to try to develop (or improve) an agent to do it by simply integrating their own component.
Further information on the development of specific solutions in order to facilitate the adoption and the usage of *Logic Programming* are provided in Chapter 5.

Wrap-up

In this chapter we illustrated how *Logic Programming* combined with other *AI* techniques can be very effective in the Intelligent Agents and Games field. In order to prove this, we illustrated some experiences in this field along with their results that confirm how the *Knowledge Representation and Reasoning* capabilities of logic-based formalism can be exploited in this context.
It is worth noticing that some of the investigations reported in this chapter have been conducted inside the international joint project Angry-HEX with researchers from Università della Calabria (UNICAL[50]), Technische Universität Wien (TUWIEN[51]), Marmara University (MARMARA[52]), and Max Planck Institut für Informatik (MPI[53]).

On the basis of the information provided in this and in the previous chapters, it is evident that proper tools solutions to support the development of logic-based products are needed. In the next chapter we report some proposals we made in order to help this.

[50]http://www.mat.unical.it
[51]http://www.kr.tuwien.ac.at
[52]http://www.knowlp.com
[53]http://www.mpi-inf.mpg.de/departments/databases-and-information-systems

5 Streamlining the use of *Logic Programming*

It's the stuff you leave out that matters. So constantly look for things to remove, simplify, and streamline. Be a curator. Stick to what's truly essential. Pare things down until you're left with only the most important stuff.

— **Jason Fried**
(ReWork)

Summary of Chapter 5

There are many reasons why, for some tasks and applications, *Logic Programming* should be preferred over other paradigms [54, 343, 385, 386, 413], however often is quite difficult to integrate it into a project (that usually uses also other formalisms) or to set-up everything needed in order to effectively run logic solvers, especially in emerging platforms, such as smartphones.

We are convinced that only if proper (i.e. reliable and easy-to-use) frameworks and tools are available, a formalism will be successful and employed by the users. Therefore, we decided to investigate new approaches that could make easier and more productive the use of the *Logic Programming* paradigm.

In this chapter, after a very brief introduction on the traditional problem of use and diffusion of *Logic Programming*, we present (in Sections 5.2 and 5.3) two projects we worked on, namely EMBASP and *Lo*IDE, which have as their purpose, among others, the simplification and the spreading of logic-based formalisms.

Chapter Outline

5.1 Motivation and Challenges

5.1.1 Streamlining the use of *Logic Programming*: why it is important[1]

A programming paradigm, in order to be successful, has to be complemented by appropriate "solutions and tools" that ensure interoperability, facilitate development and allow for low prototyping and production time. Those functionalities could seem to play only an ancillary role, but, in practice, they are useful in order to productively use a specific paradigm. Moreover, the distinctive characteristics and requirements of each programming paradigm, most of the times, do not allow the use of the tools and solutions from other paradigms because often they have completely different approaches and they need tailored solutions; for this reason, their design and development are not straightforward.

The availability of efficient solvers makes *Logic Programming* a valuable tool for many computationally intensive real-world applications. Effective large-scale software engineering requires infrastructure that includes advanced editors, debuggers, etc. These tools are usually collected in *Integrated Development Environments* (*IDEs*) that ease the accomplishment of various programming tasks by both novice and skilled software developers.

Logic Programming languages are not full general-purpose languages; thus, logic programs are eventually embedded in software components developed in different imperative/object-oriented programming languages. Current *IDEs* for *Logic Programming* provide clear advantages for developers but are not enough to enable assisted development of full-fledged industry-level applications (as examples [292, 501]).

Moreover, the development of *Application Programming Interfaces* (*APIs*), which offer methods for interacting with a *Logic Programming* system from an embedding program, is a necessary step in accommodating the use of logic-based solutions within large software frameworks common in the modern, highly technological world.

[1]Preliminary definitions adapted from [372]

5.1.2 Challenges in streamlining the use of *Logic Programming*

The number of specific "solutions and tools" developed for the *Logic Programming* paradigm is not very large, therefore many different design and engineering issues still need to be faced.

For instance, some *Logic Programming* languages do not have a standard output format and most of the logic solvers have different interfaces (there are no standard *API* like the *ODBC* standard of *DataBase Management Systems*), therefore it is not easy to work with different systems or to build applications that can work with different engines and easily switch from one to the other.

Moreover, unlike other paradigms, is not even clear what features development tool for *Logic Programming* should have, especially in the case of cloud solutions. The choice of the data-interchange language or the communication protocol is not obvious; for instance, it should be considered that often the amount of time required by the solver is much higher than the communication time but sometimes we want to process huge amount of data and therefore more efficient and scalable solutions are needed.

Furthermore, the unique features of this paradigm, such as declarativity, do not allow to easily adapt existing approaches developed for other programming paradigms.

5.2 EmbASP a general framework for embedding *Logic Programming* in complex systems[2]

EMBASP is a framework for the integration of logic formalisms in external systems for generic applications; it consists of a general and abstract architecture, implementable in a programming language of choice, that easily allows for proper specializations to different platforms and solvers.

Available solvers for *Logic Programming* feature different usability, ranging from a complete “black-box” approach to more “white-box” oriented proposals, and as long as the number of applications grows, the need for proper tools and interoperability mechanisms arises, for easing the development of logic-based applications in real-world contexts. Indeed, instead of from-scratch implementations of the sophisticated solving techniques featured by such solvers, or ex-novo suitable mechanisms for enabling their external executions, proper ready-to-use and reliable tools for embedding those efficient systems would greatly ease the development of general applications relying on *Logic Programming* for reasoning tasks.

In its earlier versions [130, 233], EMBASP was mainly tailored on *ASP*; hereafter, we show how the framework has been generalized for enabling the embedding of further logic formalisms, far beyond those similar to *ASP*, such as *Datalog*. Indeed, thanks to the general and abstract principles that guided the design of EMBASP, we also easily extended it supporting the *PDDL* planning language.

Furthermore, we developed an actual *Java* implementation that has been specialized in order to permit the embedding of *ASP* and *PDDL* logic modules into Desktop and mobile applications, making use of a wide range of solvers.
In particular, we produced six specialized libraries allowing the embedding of the *ASP* solvers DLV, *clingo* and DLV2 and the *PDDL Solver.Planning.Domains* [436] solver on Desktop platform. Moreover, DLV and *Solver.Planning.Domains* are also available on *Android*.

[2]From F. Calimeri, D. Fuscà, **S. Germano**, S. Perri and J. Zangari. ‘Boosting the Development of ASP-Based Applications in Mobile and General Scenarios’. In: *Proceedings of AI*IA 2016*, pp. 223–236. DOI: 10.1007/978-3-319-49130-1_17.
And F. Calimeri, D. Fuscà, **S. Germano**, S. Perri and J. Zangari. ‘Embedding ASP in mobile systems: discussion and preliminary implementations’. In: *Proceedings of ASPOCP 2015, workshop of ICLP*.
And F. Calimeri, D. Fuscà, **S. Germano**, S. Perri and J. Zangari. ‘A framework for easing the development of applications embedding answer set programming’. In: *J. Exp. Theor. Artif. Intell.* (2017). Submitted.
And D. Fuscà, **S. Germano**, J. Zangari, M. Anastasio, F. Calimeri and S. Perri. ‘A framework for easing the development of applications embedding answer set programming’. In: *Proceedings of PPDP 2016*, pp. 38–49. DOI: 10.1145/2967973.2968594.

Recently we provided also a *Python* implementation of the EMBASP framework that contains exactly the same functionalities of the *Java* one, apart from the specializations for *Android* due to restriction of the platform[3]. This clearly shows the extensibility of our approach, further discussions on this can be found in Section 5.2.5. In the following, we will discuss mainly the *Java* implementation, since both implementations contain the same functionalities and they are organized with the same structure.

The framework features explicit mechanisms for translations between strings and objects in the programming language at hand, directly employable within applications. This gives developers the possibility to work separately on logic-based modules and on applications that make use of them and keeps things simple when developing complex applications. Let us think, for instance, of a scenario in which different stakeholders are involved, such as *Android/Java* developers and *KR&R* experts. Both figures can take advantage of the fact that the *Knowledge Base* and the reasoning modules can be designed and developed independently from the rest of the *Java*-based application.

The EMBASP project and its history are extensively described in [232]. Here we mostly focus on new extensions and new implementations of the Framework.

5.2.1 The Framework

In this section we present the EMBASP framework, focusing on the support for two logic formalisms, namely *ASP* and *PDDL*. We start by describing the abstract architecture, then we provide details about the aforementioned specializations, and propose an actual *Java* implementation.

The general architecture of EMBASP is depicted in Figure 5.1: it is intended as an abstract framework to be implemented in some object-oriented programming language. In addition, this architecture can be specialized for different platforms, logic languages and solvers. In the figure, white blocks depict the abstract components, while dark-grey ones represent the modules more closely related to the proposed specializations. According to its abstract nature, Figure 5.1 reports general dependencies among the main modules. Notably, each concrete implementation might require specific dependencies among the inner components of each module, as can be observed in Figure 5.2, which is related to a concrete *Java* implementation and will be discussed hereafter.

[3]*Android* applications can be written only in *Java* (with some parts in *C/C++*), and since a couple of months also in *Kotlin*

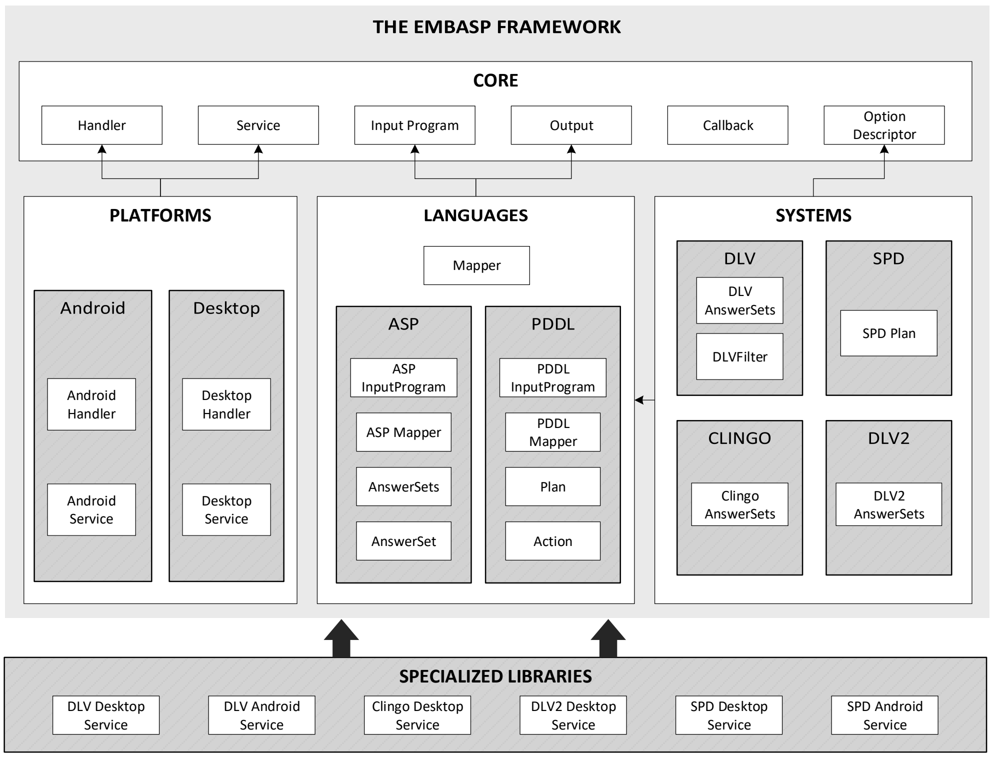


Figure 5.1.: A visual overview of EMBASP: the abstract *Framework*, and the proposed *Specialized Libraries.*. Darker blocks are related to the proposed specializations.

It is worth noting that the framework design is intended to ease and guide the generation of suitable libraries for the use of specific logic-based systems on particular platforms; resulting applications manage such systems as “black boxes”. Even though issues might arise from users demanding for a more interactive white-box usage, this allows keeping a clean design that grants an intuitive usage and an architecture which is general and easily adaptable to different platforms, reasoners and languages. The resulting libraries can hence be used in order to effectively embed logic reasoning modules, handled by the logic system(s) at hand, within any kind of application developed for the targeted platforms. In addition, as already discussed above, the framework is meant to give developers the possibility to work separately on logic-based modules and on the applications that make use of them, thus keeping things simple when developing complex applications. Additional specific advantages/disadvantages might arise depending on the programming language chosen for deploying libraries and on the target platform; special features, indeed, can make implementation, and in turn extensions and usage, easier or more difficult, to different extents.

Abstract Architecture

The framework architecture has been designed by means of four modules: *Core*, *Platforms*, *Languages*, and *Systems*, whose indented behaviour is described next. In the following, we denote with "solver" a logic-based system that is meant to be used by external application with the help of EMBASP.

Core Module The *Core* module defines the basic components of the *Framework*.
The *Handler* component mediates the communication between the *Framework* and the user who can provide it with the input program(s) via the component *Input Program*, along with any desired solver's option(s) via the component *Option Descriptor*.
A *Service* component is meant for managing the chosen solver executions.
Two different execution modes can be made available: synchronous or asynchronous. While in the synchronous mode any call to the execution of the solver is *blocking* (i.e., the caller waits until the reasoning task is completed), in asynchronous mode the call is non-blocking: a *Callback* component notifies the caller once the reasoning task is completed. The result of the execution (i.e., the output of the logic system) is handled by the *Output* component, in both modes.

Platforms Module The *Platforms* module is meant for containing what is platform-dependent; in particular, the *Handler* and *Service* components from the *Core* module that should be adapted according to the platform at hand, since they take care of practically launching solvers.

***Languages* Module** The *Languages* module defines specific facilities for each supported logic language.
The generic *Mapper* component is conceived as a utility for managing input and output via objects, if the programming language at hand permits it.
The sub-module *ASP* comprises components such as *ASPInputProgram* that adapts *Input Program* to the *ASP* case, while *AnswerSet* and *AnswerSets* represent the *Output* for *ASP*. Moreover, the *ASPMapper* allows the management of *ASP* input facts and *Answer Sets* via objects.
Similarly, the sub-module *PDDL* includes *PDDLInputProgram*, *Action*, *Plan* and *PDDLMapper*.

Systems Module The *Systems* module defines what is system-dependent; in particular, the *Input Program*, *Output* and *Option Descriptor* components from the *Core* module should be adapted in order to effectively interact with the solver at hand.

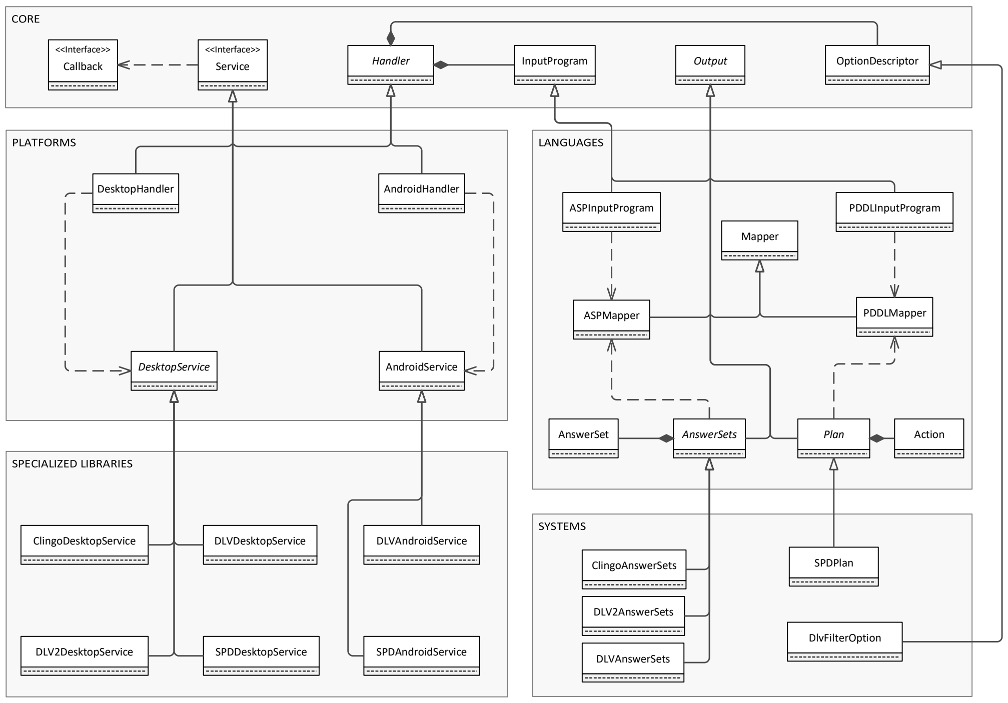


Figure 5.2.: Simplified class diagram of the provided *Java* implementation of EMBASP.

Implementing EMBASP

In the following, we propose a *Java*[4] implementation of the architecture described above. Besides the implementation of the framework itself, proper specialized libraries have been implemented.

In particular, for *ASP*, we implemented the main modules by means of classes or interfaces, and four specialized libraries that permit the use of DLV (ver. 12-17-2012) on *Android*[5] and the use of *clingo* on Desktop: different versions of *clingo* for several desktop *OS's* are already available online [264, 476]. In addition, it has been embedded the recently released DLV2, a completely renewed version of DLV, that combines the *ASP* grounder $\mathcal{I}$-DLV and the WASP solver; at the time of writing, DLV2 binaries are available for the Linux *OS*.

As for *PDDL*, we implemented a specialized library allowing the usage of the lightweight cloud *PDDL* solver, *Solver.Planning.Domains*, for any desktop platform and for the *Android* mobile one.

Figure 5.2 provides some details about classes and interfaces of the implementation. For the sake of presentation, we do not report the complete *UML* [577] class dia-

[4] https://www.oracle.com/java
[5] http://developer.android.com

gram, which is quite involved; rather, we illustrate a simplified version. Although methods inside classes have been omitted to further improve readability, adopted connectors follow *UML* syntax. In order to better outline correspondences with the abstract architecture of Figure 5.1, classes belonging to a module have been grouped together.

Core module implementation

Each component in the *Core* module has been implemented by means of a homonymous class or interface. In particular, the `Handler` class collects `InputProgram` and `OptionDescriptor` objects communicated by the user.

For what the asynchronous mode is concerned, the class `Service` depends on the interface `Callback`, since once the reasoning service has terminated, the result of the computation is returned via a class `Callback`.

Platforms module implementation

In order to support a new platform, the *Handler* and *Service* components must be adapted.

As for the *Android* platform, we developed an `AndroidHandler` that handles the execution of an `AndroidService`, which provides facilities to manage the execution of an *ASP* reasoner on the *Android* platform.

Similarly, for the desktop platform we developed a `DesktopHandler` that handles the execution of a `DesktopService`, which generalizes the usage of an *ASP* reasoner on the desktop platform, allowing both synchronous and asynchronous execution modes.

While both synchronous and asynchronous modes are provided in the desktop setting, we stick to the asynchronous one on *Android*: indeed, mobile users are familiar with apps featuring constantly reactive graphic interfaces, and according to this native asynchronous execution policy, we want to discourage a blocking execution.

Languages module implementation

This module includes specific classes for the management of input and output to *ASP* and *PDDL* solvers.

The *Mapper* component of the *Languages* module is implemented via a `Mapper` class, that allows translating input and output into *Java* objects. Such translations are guided by *Java* Annotations[6], a form of metadata that marks *Java* code and provide

[6]`https://docs.oracle.com/javase/tutorial/java/annotations`

information that is not part of the program itself: they have no direct effect on the operation of the code they annotate. They have a number of uses, such as directions to the compiler, compile-time and deployment-time processing, or runtime processing. For more details, we refer the reader to the *Java* documentation.

In our setting, we make use of such feature so that it is possible to translate input and output into strings and vice-versa via two custom annotations, defined according to the following syntax:

`@Id(string_name)` the target must be a class;

`@Param(integer_position)` the target must be a field of a class annotated via `@Id`.

In particular, for *ASP* `@Id` represents the predicate name of a ground atom that can appear as input (fact) or output (within the returned *Answer Set* (s)), while fields annotated with `@Param` define the terms and their positions in such atom. For *PDDL*, `@Id` identifies the name of an action appearing in the output plan, and fields annotated with `@Param` represent the action parameters and their positions. The mapping support is fully enabled over input and output for *ASP*, while input it is not fully supported yet for *PDDL*, and will be completed with next releases.

By means of the *Java* Reflection mechanisms, annotations are examined at runtime, and taken into account to properly define the translation.

The user has to register all its annotated classes to the `Mapper`, although classes involved in input translation are automatically detected. If the classes intended to translate are not annotated or not correctly annotated, an exception is raised. Other problems might occur if once that the solver output is returned, the user asks for a translation into objects of not annotated classes: in this case, a warning is raised and the request is ignored.

The `Mapper` in *Python*. The only notable difference between the *Java* and the *Python* implementations is in the `Mapper` component due to the absence in *Python* of a mechanism similar to the *Java* Annotations. For this reason, using a mechanism similar to the one described in Section 5.2.5, we implemented a `Predicate` abstract class to help the mapping of the objects.[7]
To make possible the translation of facts into strings and vice-versa, the abstract class `Predicate` have to be extended including the following information:

`predicateName="string_name"` a class field that contains the predicate name (in the *ASP* case) or the action name (in the *PDDL* case) to map;

[7]It is worth noticing that it is not something "extra" of the *Python* version, also in the *Java* version we have to declare and implement the two Annotations.

`[("class_field_name_1", int), ("class_field_name_2"), ...]` a list for the constructor that must contain, for each class field, one tuple with the field name, sorted by the position of the terms they represent, and optionally the keyword `int` if the field represents an integer number.

Thanks to the structure of the `Predicate` class, this information is passed to the `Mapper` class, to correctly perform the translation mechanism. If the classes intended to translate do not contain the required information, an exception is raised.

Notably, such feature is meant to give developers the possibility to work separately on the logic-based modules and on the *Java/Python* side. The mapper acts like a middleware that enables the communication among the modules and eases the burden of developers by means of an explicit, ready-made mapping between *Java/Python* objects and the logic modules.

Further insights about this feature are illustrated in the next section by means of some EMBASP use cases.

In addition to the `Mapper`, the Languages module features two sub-modules which are more strictly related to *ASP* and *PDDL*, respectively.

`ASPInputProgram` extends `InputProgram` to the *ASP* case. In addition, since the "result" of an *ASP* solver execution consists of *Answer Sets*, the *Output* class has been extended by the `AnswerSets` class that is composed by a set of `AnswerSet` objects. Moreover, the *ASP* sub-module features an `ASPMapper` class, that acts like a translator, providing proper means for a two-way translation between strings recognizable by the *ASP* solver at hand and *Java* objects directly employable within the application. The `ASPMapper` is intended for translating *ASP* input and output from and to objects: thus has a dependency from `ASPInputProgram` and `AnswerSets` classes.

In the *PDDL* sub-module, `PDDLInputProgram` extends `InputProgram` to the *PDDL* case. It is worth to notice that *PDDL* solvers typically require a clear distinction between the *domain* and the *problem* definitions. Hence, when specifying a new `PDDLInputProgram` the user has to clarify whether it is intended to be the *domain* or the *problem* part: it is not allowed to provide both parts in the same `PDDLInputProgram`. Moreover, `Action` and `Plan` represent the output of a *PDDL* solver: in particular, a `Plan` corresponds to a list of `Action` objects. The `PDDLMapper` implements the *Mapper* facilities, and depends on `Plan`, as it allows translating *PDDL* outputs in *Java* objects.

Systems Module Implementation

The classes `DLVAnswerSets`, `ClingoAnswerSets`, `DLV2AnswerSets` and `SPDPlan` implement specific extensions of the `AnswerSets` or `Plan` classes, in charge of manipulating the output of the respective solvers.

Moreover, this module can contain classes extending `OptionDescriptor` to implement specific options of the solver at hand. For instance, the class `DLVFilter` is a utility class representing the filter option of DLV.

Specializing the Framework

We implemented six libraries derived from EMBASP, allowing the embedding of *ASP* and *PDDL* reasoning modules. In the *ASP* case, these computations are handled by DLV (ver. 12-17-2012) from within *Android* and Desktop apps, and by DLV2 and *clingo* inside standalone Desktop applications; in the *PDDL* case, the reasoning can be performed both on *Android* and Desktop platforms by means of the *Solver.Planning.Domains* solver.

The class `DLVAndroidService` is in charge of offering *ASP* reasoning on *Android*, while the `DLVDesktopService`, `ClingoDesktopService` and `DLV2DesktopService` classes offer the same support on the Desktop platform.

`DLVAndroidService` is a specific version of `AndroidService` for the execution of DLV on *Android*. It is worth noting that DLV was not available for *Android*; furthermore, it is natively implemented in *C++*, while the standard development process on *Android* is based on *Java*. To this end, DLV has been on purpose rebuilt using the NDK (Native Development Kit)[8], and has been linked to the *Java* code using the *JNI*[9]. This grants the access to the *APIs* provided by the *Android* NDK and in turn accedes to the DLV exposed functionalities directly from the *Java* code of an *Android* application.

`DLVDesktopService`, `ClingoDesktopService` and `DLV2DesktopService` are specific versions tailored for the DLV, *clingo* and DLV2 reasoners, respectively, on the desktop platform; they extend the `DesktopService` with proper functions needed to invoke the embedded solver(s).

`SPDDesktopService` and `SPDAndroidService` are in charge of executing the solver called *Solver.Planning.Domains*. In particular, the `SPDDesktopService` class permits synchronous and asynchronous executions on every Desktop *OS*; while the respective `SPDAndroidService` class allows just asynchronous executions on the

[8]`https://developer.android.com/tools/sdk/ndk`
[9]`http://docs.oracle.com/javase/8/docs/technotes/guides/jni`

Android platform. Notably, differently from embedded solvers (i.e DLV), due to the fact that the solver is invoked in a cloud-based mode via *HTTP* requests, in both classes it is employed without any platform-specific adaptation. Besides the fact that on *Android* is only implemented the asynchronous mode, the main difference between the two classes relies on the way input files are managed. In particular, `SPDAndroidService` requires that all files are in the *Android Resources folder*, that within *Android* applications, represent the standard way to store images, sounds, files and resources in general[10].

5.2.2 Using EMBASP for Embedding Logic Formalisms

In the following we show how to employ the aforementioned specialized *Java* libraries generated via EMBASP for making use of *ASP* and *PDDL* from within actual *Java* applications.

We first describe the development of an *Android* application based on *ASP*, for solving *Sudoku* puzzles; then, we present a Desktop application relying on *PDDL* for solving the blocks-world planning problem. For the sake of simplicity, both examples will focus on the code related to the EMBASP usage; nonetheless, the complete code is available online at

https://www.mat.unical.it/calimeri/projects/embasp

It is worth noting that, although the following example applications make use of one formalism via one solver, EMBASP allows also to deploy applications that rely on multiple logic formalisms and multiple solvers at once. One can think, for instance, of a scenario where an application may choose to use a *PDDL* planning module for some tasks, and an *ASP* reasoning module for others: these tasks might be executed independently, or they might be interleaved or interact in different ways, i.e, the output of a task could become the input of another one. Notably, thanks to the annotation-guided mapping, the logic-based aspects can be separated from the *Java* coding: the programmer does not even necessarily need to be aware of the logic formalisms employed.

Eventually, we report some considerations about EMBASP implementation in different programming languages alternative to *Java*.

Embedding *ASP*

Imagine that a user-designed (or has been given) a proper logic program P to solve a *Sudoku* puzzle, and also that she has been provided with an initial schema which

[10] http://developer.android.com/guide/topics/resources

is meant to be solved. We assume that the initial schema is well-formed, i.e., the complete schema solution exists and is unique. For instance, P can correspond to the logic program presented in Section 1.3.4, so that, coupled with a set of facts F representing the given initial schema, allows to obtain the only admissible solution (i.e., a single *Answer Set*). It is worth remembering that, in case of less usual *Sudoku* schemata featuring multiple solutions, the *ASP* program features multiple *Answer Sets*, one-to-one corresponding to such solutions.

By means of the annotation-guided mapping, the initial schema can be expressed in forms of *Java* objects. To this extent, we define the class `Cell`, aimed at representing a single cell of the *Sudoku* schema, as follows:

```
@Id("cell")
public class Cell {

@Param(0)
private int row;

@Param(1)
private int column;

@Param(2)
private int value;

[...]

}
```

Listing 5.1: Definition of the annotated `Cell` class.

It is worth noticing how the class has been annotated by two custom annotations, as introduced above. Thanks to these annotations the `ASPMapper` will be able to map `Cell` objects into strings properly recognizable from the *ASP* solver as logic facts of the form $cell(Row, Column, Value)$.

At this point, we can create an `Android Activity Component`[11], and start deploying our *Sudoku* application:

```
public class MainActivity extends AppCompatActivity {

  [...]
  private Handler handler;

  @Override
  protected void onCreate(Bundle bundle) {
    handler = new AndroidHandler(getApplicationContext(),
                                 DLVAndroidService.class);
    [...]
  }

  public void onClick(final View view){
```

[11]`https://developer.android.com/reference/android/app/Activity.html`

```
14        [...]
15        startReasoning();
16      }
17
18      public void startReasoning() {
19        InputProgram inputProgram = new ASPInputProgram();
20
21        for (int i = 0; i < 9; i++)
22          for (int j = 0; j < 9; j++)
23            try {
24              if(sudokuMatrix[i][j]!=0) {
25                inputProgram.addObjectInput(new Cell(i, j, sudokuMatrix[i][j
                      ]));
26              }
27            } catch (Exception e) {
28              // Handle Exception
29            }
30
31        handler.addProgram(inputProgram);
32
33        String sudokuEncoding = getEncodingFromResources();
34        handler.addProgram(new ASPInputProgram(sudokuEncoding));
35
36        Callback callback = new MyCallback();
37        handler.startAsync(callback);
38      }
39    }
```

Listing 5.2: An example of an *Android* Activity for the *Sudoku* problem.

The class contains a `Handler` instance as field, that is initialized when the `Activity` is created as an `AndroidHandler`. Required parameters include the `Android Context` (an *Android* utility, needed to start an *Android Service Component*) and the type of `AndroidService` to use – in our case, a `DLVAndroidService`. In addition, in order to represent an initial *Sudoku* schema, the class features a matrix of integers as another field where position (i, j) contains the value of cell (i, j) in the initial schema; cells initially empty are represented by positions containing zero.

The method `startReasoning` is in charge of actually managing the reasoning: in our case, it is invoked in response to a "click" event that is generated when the user asks for the solution. Lines 19–31 create an `InputProgram` object that is filled with `Cell` objects representing the initial schema, which is then served to the handler; lines 33–34 provide it with the *Sudoku* encoding. It could be loaded, as it is common on *Android* apps, by means of a utility function that retrieves it from the *Android Resources folder*.

At this point, the reasoning process can start; since for *Android* we provide only the asynchronous execution mode, a callback object is in charge of fetching the output when the *ASP* system has done (lines 36–37).

Eventually, once the computation is over, from within the callback function the output can be retrieved directly in form of *Java* objects. For instance, in our case an inner class `MyCallback` implements the interface `Callback`:

```java
private class MyCallback implements Callback {

  @Override
  public void callback(Output o) {
    if(!(o instanceof AnswerSets))
      return;
    AnswerSets answerSets = (AnswerSets)o;
    if(answerSets.getAnswersets().isEmpty())
      return;
    AnswerSet as = answerSets.getAnswersets().get(0);
    try {
      for(Object obj:as.getAtoms()) {
        Cell cell = (Cell) obj;
        sudokuMatrix[cell.getRow()][cell.getColumn()] = cell.getValue();
      }
    } catch (Exception e) {
      // Handle Exception
    }
    displaySolution();
  }

}
```

Listing 5.3: An example of how to implement the `Callback` interface.

Embedding *PDDL*

Let us consider the blocks-world problem definition, as reported in Section 1.4.2, and suppose that we want to deploy a Desktop blocks-world application.

We will make use of the annotation-guided mapping, in order to retrieve the actions constituting a *PDDL* plan via *Java* objects. To this purpose, the following classes are intended to represent possible actions that a blocks-world solution plan can feature:

```java
@Id("pick-up")
public class PickUp {

  @Param(0)
  private String block;

  [...]

}
```

Listing 5.4: Definition of the annotated `PickUp` class.

```java
@Id("put-down")
public class PutDown {

  @Param(0)
  private String block;

  [...]

}
```

Listing 5.5: Definition of the annotated `PutDown` class.

```
@Id("stack")
public class Stack {

  @Param(0)
  private String block1;

  @Param(1)
  private String block2;

  [...]

}
```

Listing 5.6: Definition of the annotated Stack class.

```
@Id("unstack")
public class Unstack {

  @Param(0)
  private String block1;

  @Param(1)
  private String block2;

  [...]

}
```

Listing 5.7: Definition of the annotated Unstack class.

At this point, supposing that we are given two files defining the blocks-world domain and a problem instance, we can start deploying our application:

```
public class Blocksworld{

  private static String domainFileName = "domain.pddl";
  private static String problemFileName = "p01.pddl";

  public static void main(String[] args) {
    Handler handler = new DesktopHandler(new SPDDesktopService());

    final InputProgram inputProgramDomain = new PDDLInputProgram(
        PDDLProgramType.DOMAIN);
    inputProgramDomain.addFilesPath(domainFileName);

    final InputProgram inputProgramProblem = new PDDLInputProgram(
        PDDLProgramType.PROBLEM);
    inputProgramProblem.addFilesPath(problemFileName);

    handler.addProgram(inputProgramDomain);
    handler.addProgram(inputProgramProblem);

    try {
      PDDLMapper.getInstance().registerClass(PickUp.class);
      PDDLMapper.getInstance().registerClass(PutDown.class);
      PDDLMapper.getInstance().registerClass(Stack.class);
      PDDLMapper.getInstance().registerClass(Unstack.class);

      Plan plan = (Plan)(handler.startSync());

      for (final Object obj : plan.getActionsObjects())
        //Manage objects as needed

    } catch (Exception e) {
      // Handle Exception
    }
  }
}
```

Listing 5.8: An example of a *Java* application for the Blocksworld problem.

Line 7 creates a `DesktopHandler` object, to which a `SPDDesktopService` object is given; indeed, we are deploying a Desktop application making use of the *PDDL* solver *Solver.Planning.Domains*. Lines 9–16 set-up the input to the solver; since *PDDL* requires separate definitions for domain and problem, two `PDDLInputProgram` are created and then given to the handler. Lines 19–22 inform the `PDDLMapper` about what classes are intended to map the output actions. Eventually, line 24 synchronously invokes the solver, and then retrieves the output; at lines 26–27 the output actions can be managed accordingly to the user's desiderata.

5.2.3 *ASP*-based Applications: some Examples in the Educational Setting

In this section we describe some *ASP*-based applications developed by means of EMBASP for educational purposes, and, in particular, in the context of a university course that covers *ASP* topics; it is worth noting that such applications have been developed by some of the course attendants, i.e., undergraduate students. The educational aspect here is two-folded. The most relevant is the engagement of university (under)graduate students in *ASP* capabilities, in order to make them able to take advantage from it when solving problem and designing solutions, in the broadest sense. Furthermore, *ASP* looks well-fitted for the use in the development of educational/training software, as, for instance, the *DLVEdu* app introduced below; a deeper study of such aspects, however, is out of the scope of the present work.

In the following, we first briefly introduce three applications; then, in order to further clarify the EMBASP use, especially in the mobile setting, we describe the *DLVfit Android* App more in detail.

GuessAndCheckers

GuessAndCheckers is a native mobile application that works as a helper for users that play "live" games of the (Italian) checkers (i.e., by means of physical board and pieces).
It is extensively described in Section 4.5.1.

DLVEdu

DLVEdu is an educational *Android* App for children, that integrates well-established mobile technologies, such as voice or drawn text recognition, with the modelling capabilities of *ASP*. In particular, it is able to guide the child throughout the learning tasks, by proposing a series of educational games, and developing a personalized educational path. The games are divided into four macro-areas: Logic, Numeric-

Mathematical, Memory, and Verbal Language. The usage of *ASP* allows the application to adapt to the game experiences fulfilled by the user, her formative gap, and the obtained improvements.

The application continuously profiles the user by recording mistakes and successes, and dynamically builds and updates a customized educational path through the different games.

The application features a "Parent Area", that allows parents to monitor child's achievements and to express some preferences, such as desired express directions in order to grant/forbid access to some games or educational areas.

Connect4

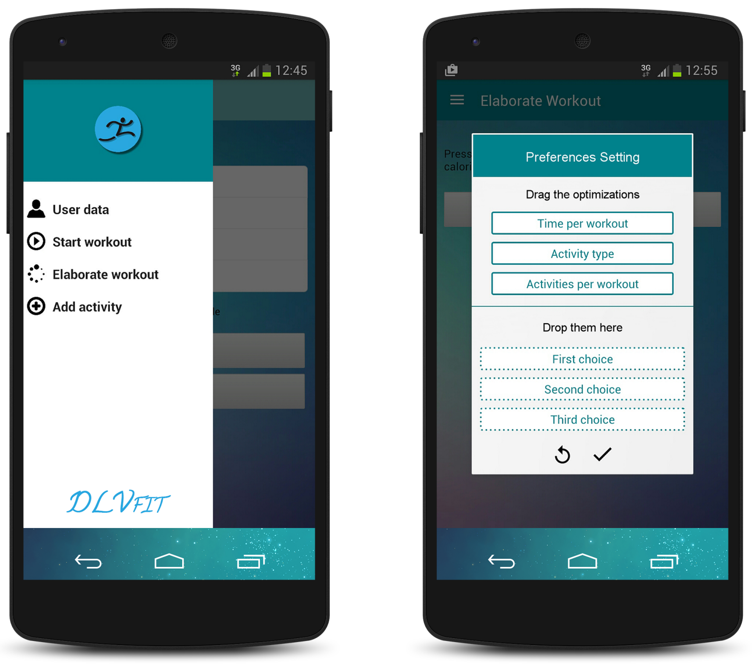
The popular turn-based *Connect Four* game is played on a vertical 7*6 rectangular board, where two opponents drop their disks with the aim of creating a line of four, either horizontally, vertically, or diagonally.

The *Connect4* application allows a user to play the game (also known as *Four-in-a-Row*) against an *ASP*-based artificial player. Notably, the declarative nature of *ASP*, its expressive power, and the possibility to compose programs by selecting proper rules, allowed designing and implementing different *AI*s, ranging from the most powerful one, that implements advanced techniques for the perfect play, to the simplest one, that relies on some classical heuristic strategies. Furthermore, by using EMBASP, two different versions of the same app have been built: one for *Android*, making use of DLV, and one for *Java*-enabled desktop platforms, making use of *clingo*.

DLVfit

The *DLVfit Android* App was the first application making use of the framework; it was conceived as a proof of concept, in order to show the framework features and capabilities. To our knowledge, it is also the first mobile app natively running an *ASP* solver.

DLVfit is a health app that aims at suggesting the owner of a mobile device the "best" way to achieve some fitness goals. The app lets the user express her own goals and preferences in a very customizable way along many combinable dimensions: calories to burn, time to spend, differentiation over several physical activities, time constraints, etc. Then, it monitors her actual activity throughout the day and, upon request, it computes one or more plans meant, if accomplished, to make her meet the aforementioned goals the way she would have preferred.



(a) (b)

Figure 5.3.: Screenshots from the *DLVfit* app: main menu (a) and list of optimizations (b).

More in detail, the app constantly detects the current user activity (running, walking, cycling, etc.) and (at a customizable frequency) stores some information (activity type, timestamps, calories burned up to the present time, etc.). Activity detection is performed by means of the *Google Activity Recognition APIs*[12], a de-facto standard on *Android*, thus relying on these for the accuracy of detection. As already mentioned, the user might ask, at any time, for a suggestion about a plan for the rest of the day; the reasoning module hence prepares a (set of) proper workout plans complying with the very personal goals and preferences previously expressed.

The user interacts with the app via a standard graphical interface; the reasoning module is actually in charge of building a proper *ASP* program, which is in turn fed to DLV via EMBASP. Such program matches the classical "Guess/Check/Optimize" paradigm introduced in Section 1.3.3, thus resulting easy to understand, enrich and customize:

- the "guess" part chooses how much time to spend on each exercise;
- the "check" part forces the resulting plan to be admissible: burning the remaining amount of desired calories, do not exceed the time constraints, etc.;

[12]https://developer.android.com/reference/com/google/android/gms/location/ActivityRecognition.html

- the "optimize" part, eventually, expresses preferences: minimize total time spent exercising, number of activities to perform, maximize the number of different activity types, avoid activities around a given time of the day, etc.

The logic program used takes as "input" (i.e., a set of facts as instances of proper predicates):

$calories\_burnt\_per\_activity(A, C)$
: the calories burnt (C), in each unit of time, per each Activity (A);

$remaining\_calories\_to\_burn(R)$
: the remaining calories to burn in the rest of the current day;

$how\_long(A, D)$
: the amount of time that can be spent for each activity A (in order to reach the goal of burn all the remaining calories);

$max\_time(T)$
: the duration of the workout (max: the remaining time to the end of day);

$surplus(C)$
: the maximum surplus of calories to burn with the suggested workouts;

$optimize(O, W, P)$
: the specific optimization operation(s) that the user wants to perform; each direction is assigned a weight (W) and a preference order (P).

An example of the basic input concepts described above is the following:

```
calories_burnt_per_activity("ON_BICYCLE", 5).
calories_burnt_per_activity("WALKING", 2).
calories_burnt_per_activity("RUNNING", 11).

remaining_calories_to_burn(200).

how_long("ON_BICYCLE", 10).
how_long("ON_BICYCLE", 20).
how_long("WALKING", 10).
how_long("WALKING", 20).
how_long("RUNNING", 10).
how_long("RUNNING", 20).

max_time(20).

surplus(100).
```

Listing 5.9: An example of input for the *DLVfit* logic code.

In this example the activities that can be performed (`"ON_BICYCLE"`, `"WALKING"` and `"RUNNING"`) are specified along with the calories they allow to burn per unit of time; then, the amount of time spent for each activity is reported. Moreover, there are pieces of information about the calories that remain to burn in the current day (at least 200, and up to 300 due to the `surplus`) and the maximum time that the user wants to spend on the workouts (20).

Custom optimization preferences are typically represented as follows:

```
optimize("RUNNING", 1, 3).
optimize("ON_BICYCLE", 3, 3).
optimize("WALKING", 2, 3).

optimize(time,0,2).

optimize(activities, 0, 1).
```

Listing 5.10: An example of custom optimization preferences for *DLVfit*.

Solutions, in this context, are actually workouts suggestions to the user.
The `optimize` predicate is of arity 3, and the third argument is supposed to express the “importance” of the statement (the higher the number, the more the importance).
In this example, the *ASP* code models that: (i) the user wants (preference level: 3) to maximize the number of favourite activities to perform, and provides an order (`"RUNNING"` first, then `"WALKING"` and finally `"ON_BICYCLE"`); (ii) if more than one admissible workout is found featuring the same favourite activities, she wants to minimize the total time spent exercising (preference level: 2); also, (iii) if there are workouts that have the same favourite activities and the same time, she wants to minimize the total number of activities (preference level: 1).

The logic program is able to find the combinations of activities that should be performed in order to burn the remaining calories. Obviously, this goal can be achieved, in general, in many different ways, each of them modelled by a different *Answer Set*. Part of the rules of the program that we used are reported hereafter; full program is available online.

```
%%%%%% Guess Part %%%%%%
activity_to_do(A, HL) | not_activity_to_do(A, HL) :-
    how_long(A, HL).

%%%%%% Check Part %%%%%%
:- activity_to_do(A, HL1), activity_to_do(A, HL2), HL1 !=
    HL2.

:- remaining_calories_to_burn(RC),
total_calories_activity_to_do(CB), RC > CB.

:- remaining_calories_to_burn(RC),
    total_calories_activity_to_do(CB), CB > RCsurplus,
    RCsurplus = RC + surplus.
```

```

:- max_time(MTS), MTS < TS, total_time_activity_to_do(TS).

%%%%%% Optimize Part %%%%%%
:~ optimize(A, W, P), activity_to_do(A, _). [W:P]

:~ optimize(time, _, P), activity_to_do(_, HL). [HL:P]

:~ optimize(activities, _, P), #int(HM), #count{A, HL :
    activity_to_do(A, HL)} = HM. [HM:P]
```

Listing 5.11: A simplified version of the *DLVfit* logic program.

The `Guess Part` chooses how much time to spend on each exercise. The `Check Part` checks that each activity selected has one specific amount of time, it ensures that all the remaining calories are burnt and that not more calories than the remaining (with the surplus) are burnt and it ensures to not exceed the maximum time that the user wants to spend on the workouts. The `Optimize Part` makes use of *weak constraints*[111, 126]: in case the user specified preferences about activities, tries to select the favourite ones; in case she specified preferences about the time spent exercising, tries to minimize it; if she specified preferences about the number of different activities, tries to minimize it.

There is a wide range of customization possibilities in this setting: thanks to the modelling capabilities and the declarative nature of *ASP*, adding new features to *DLVfit*, such as new exercises or new kind of preferences, is straightforward and sums up to adding a few lines to the logic program. It is also worth noting that the *ASP* program is dynamically built, thus providing the developer (and, in turn, the final user) with great customization and flexibility capabilities. Indeed, we plan to actually take advantage from this in the future versions of the prototype, contemplating a higher number of rules and sub-programs to be dynamically fed to DLV.

5.2.4 Related Works

The problem of embedding *ASP* reasoning modules into external systems and/or externally controlling an *ASP* system has been already investigated in the literature; to our knowledge, the more widespread solutions are the DLV *Java Wrapper* [500], *JDLV* [214], and the scripting facilities featured by *clingo* 4 [260], which allow, to different extents, the interaction and the control of *ASP* solvers from external applications.

In *clingo* 4, the scripting languages *Lua* and *Python* enable a form of control over the computational tasks of the embedded solver *clingo*, with the main purpose of supporting also dynamic and incremental reasoning; on the other hand, EMBASP, similarly to the *Java Wrapper* and *JDLV*, acts like a versatile "wrapper" wherewith the developers can interact with the solver. However, differently from the *Java Wrapper*, EMBASP features a *Mapper* that, in the *Java* implementation, makes use of annotations, a form of metadata that can be examined at runtime, thus allowing an easy mapping of input/output to *Java* Objects; and differently from *JDLV*, that uses *JPA* annotations for defining how *Java* classes map to relations similarly to *ORM* frameworks, EMBASP straightforwardly uses custom annotations, almost effortless to define, to deal with the mapping.

In addition, it allows building applications that can run different solvers, and different instances, at the same time; none of the mentioned systems exposes this feature.

A new work [487], more recent w.r.t. the earlier versions of EMBASP, introduces a formal language for defining mappings of input/output of an *ASP* program in form of objects intended to be handled by some programming language. These statements are embedded directly within the *ASP* code, and dedicated libraries are in charge of interfacing the *ASP* program with the selected object-oriented language. The proposed approach is independent of the concrete object-oriented language adopted, and a *Python* library, namely *PY-ASPIO*, has been provided as a reference implementation. Similarly to EMBASP, mapping focuses on input/output of the *ASP* programs, and is independent of the programming language adopted. However, differently from EMBASP, where the mapping is guided by custom annotations on the programming language code, in this approach the mapping annotations are part of the *ASP* code; this implies a more tight connection between the logic-based aspects and the object-oriented ones. Moreover, while this work focuses on *ASP*, EMBASP is not limited to a specific logic formalisms.

Several ways of taking advantage of *ASP* capabilities have been explored, and, interestingly, not all of them require to natively port an *ASP* system on the device of use. In particular, it is possible to let the reasoning tasks take place somewhere else, and use internet connections in order to communicate between the "reasoning service" and the actual application, according to a cloud computing paradigm, to some extent. Thanks to such mechanisms, mobile apps relying on *ASP* reasoning have already been introduced: in [185] a prototype system is presented, called *HealthyLife*, which makes use of *ASP*-based *Stream Reasoning* (ASR) in a mobile health app that has some point of contacts with *DLVfit*. The focus of *HealthyLife* is primarily to detect users daily activities and try to deal with ambiguities when recognizing situations, while *DLVfit* delegates this task to *Android Recognition API*: its primary goal is to experiment with the usage of *ASP* on mobile devices. In this respect, although the computational power of a dedicated server is not comparable to the one of a mobile device, it would be interesting to see whether *HealthyLife* could benefit from the embedding of DLV and EMBASP within it.

Both the approaches are interesting, and each of them has pro and cons depending on the scenario it is facing. The cloud-based approach grants great computational power to low-end devices, without the need for actually porting a system to the final user's device, and completely preventing any performance issue. However, in order this to take place, there is first the need for a proper application hosting, which requires non-negligible efforts both from the design and the economic points of view; furthermore, a steady internet connection might be a strong constraint, especially when the communication between the end user's device and the cloud infrastructure requires a large bandwidth. On the other hand, a native-based approach might involve significant efforts for the actual porting of pieces of software on the target device, which, in turn, might lead to performance or power consumption issues; and even if performance issues might not appear as always crucial, given the computational power which is available even on mobile devices, power consumption is sometimes decisive. Notably, the main idea behind this work is to embed an *ASP* solver directly in a mobile context, however, this possibility is not hindered by the framework. In fact, due to the structure of the middleware layer (SOLVER HANDLER), it is possible to hide the details of the solver invocation, so that it can also be carried out using a cloud/server solution.

Nevertheless, in our showcase scenario, *DLVfit* shows that the development of applications that natively runs *ASP*-based reasoning tasks on mobile devices does not necessarily suffer from the discussed drawbacks. Indeed, DLV is invoked only on demand, i.e., whenever the user wants to check possible alternatives about how to spend the rest of her day; for the whole rest of the time, no solver is running or waiting, thus preventing both performance and battery drain.

On the *PDDL* side, to our knowledge, there still is a lack of interoperability tools. A contribution in this direction is ABLE [537], an agent building toolkit providing a domain-independent planning and execution environment. The approach is very different from EMBASP, since it is focused on *PDDL* and offers advanced functionalities specific for agents performing planning tasks.

Concerning generic logic-embedding tools, some connections can be found with Tweety [559, 560], an open source framework for experimenting with logical aspects of artificial intelligence; it consists of a set of *Java* libraries that allow making use of several *Knowledge Representation* systems supporting different logic formalisms, ranging from classical logics, over logic programming and computational models for argumentation, to probabilistic modelling approaches, including *ASP*. Tweety and EMBASP cover a wide range of applications, and the use is very similar: at the bottom line, both provide libraries to incorporate proper calls to external declarative systems from within "traditional" applications. Currently, Tweety implementation is already very rich, covering a wide range of *KR* formalisms, yet looking less general, as the more abstract level is conceived as a coherent structure of *Java* libraries; also, it currently misses the mobile focus. EMBASP originally was mainly focused on fostering the use of *ASP* in the widest range of contexts, as evidenced by the specialization for the mobile setting; nevertheless, the framework core is very abstract, and has been conceived in order to create libraries for different programming languages, platforms and formalisms.

5.2.5 Design, Implementation and Usability

In the latest years the worldwide commercial, consumer and industrial scenario significantly changed; the growing popularity of "smart"/wearable devices and the *Internet of Things* (*IoT*) forced the whole *ICT* industry to radically evolve. Nevertheless, there is still a lack of tools for taking advantage of the knowledge representation capabilities of logic formalisms in this context. The specialization of EMBASP for DLV on *Android* has been an attempt to ease the development of mobile applications natively using logic-based reasoners; to our knowledge, it represented actually the first attempt reported in literature for *ASP*. As already mentioned, indeed, the preliminary version of EMBASP was originally explicitly tailored to the mobile scenario [130, 131].

Afterwards, however, the framework has been extended [233] for fostering the usage of *ASP* within real-world and industrial contexts, where it gained popularity; the framework has been made more abstract, and independent from the running platform.

The current version of EMBASP, herein presented, further enhances its abstract nature and generalizes it by opening also the logic language side, thus potentially supporting any kind of formalism; contextually, the generalized framework has been used in order to enrich the actual implementation with the support to *PDDL*. It is worth noting that *PDDL* and *ASP* are intrinsically very different, from the syntax, semantics, and knowledge representation sides; this makes the flexibility of EMBASP evident. Furthermore, new actual libraries were generated: comprehensively, the EMBASP implementations currently features the embedding of the *ASP* solvers DLV, *clingo* and DLV2, and the *PDDL* cloud solver *Solver.Planning.Domains* into Desktop applications, and the embedding of DLV and *Solver.Planning.Domains* into *Android* apps; and solvers are invoked in different modes: *Solver.Planning.Domains* is invoked via remote connections, while the others are effectively embedded, and binaries are natively executed at will by the resulting applications.

The gradual evolution and extension of EMBASP has been marked by a series of applications making use of it, developed in the context of university courses that cover *ASP* topics; authors were some of the course attendants (i.e., undergraduate students). This helped at collecting feedbacks and improving framework usability, and also proved how gradually usage became more general and easier. A showcase of the most interesting apps developed is reported in previous works [131, 233]; most of them consist of *Android* applications taking advantage from the explicit and declarative *KR&R* capabilities of *ASP*: students experimented with different logic tasks, having the chance to test the *AI*s without the need for rebuilding the application each time an update was made, thus observing the impact of changes "on the fly".

Further considerations deserve to be made about different implementation approaches, such as the choice of the programming language. Indeed, since its first preliminary implementation the programming language of choice was *Java*. Besides the fact that it represents a very popular, solid and reliable programming language, the choice was also motivated by the intention to foster the use of logic formalisms in new scenarios; in particular in the mobile one, where *Android* is by far the most widespread mobile platform, and its development and deployment models heavily rely on *Java*. However, the abstract architecture of EMBASP can be made concrete by means of other object-oriented programming languages.

Most of the components in the proposed *Java* implementation have been accomplished thanks to features that are typical of any object-oriented language, such as *inheritance* and *polymorphism*. The unique exception is represented by the *Mapper* components, implemented by means of *Java* peculiar features, such as *annotations* and *reflection*. In case of other languages that feature similar constructs, such as `C#`[13], the approach can resemble the herein presented *Java* implementation.

With different languages that lack such features, the mapping mechanism can still be implemented with a simulation via inheritance and polymorphism and applying typical Software Engineering patterns [237]. As a matter of example, one possible implementation can be accomplished using the *Prototype design pattern*, that results well-suited to our purposes, as it allows to "specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype" [237]. Such pattern can be the key to simulate the dynamical loading of

[13]Microsoft Developer Network, MSDN: *C# Attributes* (`https://msdn.microsoft.com/en-us/library/mt653979`), *C# Reflection* (`https://msdn.microsoft.com/en-us/library/mt656691`)

classes in languages that do not support it natively, as it happens with *C++*. Indeed, the run-time environment can make use of it in order to automatically create an instance of each class when it is loaded, and then register the instance with a prototype manager – in our case, represented by the *Mapper* components. All classes that in *Java* (or similar languages) would make use of reflection and annotations, can be defined by extending a properly defined `Prototype` class and then specify how to map input and output objects. Moreover, a `Mapper` class would still be needed, with a behaviour quite similar to the *Java* case.

The *Python* implementation of EMBASP is indeed the proof that it is possible to implement our Abstract Framework even in languages that do not support constructs similar to the *Java Annotations*.

5.2.6 Discussion

EMBASP is a framework for embedding logic-based reasoners into external systems. The fully abstract architecture makes the framework general enough to be adapted to a wide range of scenarios; indeed, it can support different logic-based formalisms, can be implemented in any programming language, grounded to different platforms, and can make use of different solvers.

We presented an actual *Java* implementation of the framework supporting the two declarative formalisms *ASP* and *PDDL*; this allowed deploying six specialized libraries for embedding different *ASP/PDDL* systems into both *Java*-based Desktop and native *Android* applications. We also briefly discussed the *Python* implementation of the framework and the main difference w.r.t the *Java* one.

The framework has been tested within some university courses featuring *ASP* topics, for implementing a set of applications, ranging from *AI*-based games to educative apps; it proved to be an effective set of tools and interoperability mechanisms able to ease the development of *ASP*-based applications, in both educational and real-world contexts. Eventually, we illustrated the use of the libraries by means of some running examples, and described the design and implementation of the framework and its improved usability as witnessed by complete applications developed in academic contexts.

Framework, documentation, applications and further details are freely available online at

`https://www.mat.unical.it/calimeri/projects/embasp`

5.3 LoIDE a web-based *IDE* for *Logic Programming*[14]

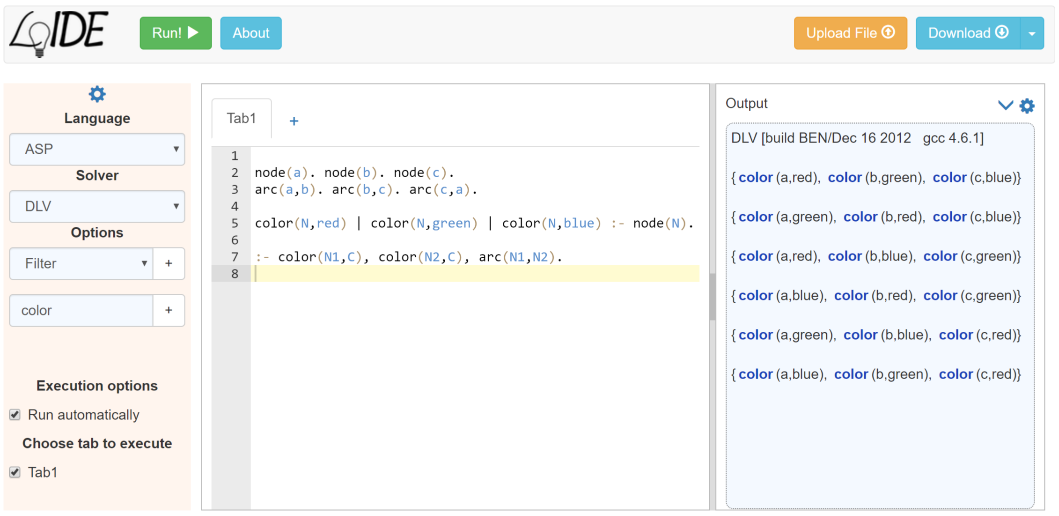
In the latest years, declarative paradigms and approaches to solving problems increasingly crossed the border of academia and have been going beyond theoretical studies in order to get into the real world. This is especially the case for logic-based formalisms; indeed, after years of theoretical results, the availability of solid and reliable systems made viable the implementation of effective logic-based solutions, even in the industrial context.

Along with such improvements in solver technology, the lack of suitable engineering tools for developing programs started to be properly addressed; for instance, one might think of the work carried out by the *Answer Set Programming* (*ASP*) community, that explicitly addressed issues like writing, debugging and testing *Answer Set* programs as well as embedding them into external, traditionally-developed systems. See Section 1.3.2 for more information.

At the same time, scenarios of computing significantly changed as well, now heavily relying on network connections and tools; in this context, the web-application paradigm, granting accessibility independently from the device in use, even from mobile, become very popular. Many existing desktop applications have been "ported" to the web, and many new have been created specifically according to this paradigm. Moreover, the *JavaScript* language became a real cross-platform language, due to availability on all types of devices, ranging from servers to *Internet of Things* (*IoT*); it has been improved with many interesting features that made it an ideal language for developing not only small scripts but also fully-fledged applications; and, fortunately, cloud-computing technologies significantly eased development, deploying and use of such applications.

In this scenario, many tools for software development have been released as web-applications, from simple text editors to *Integrated Development Environments* (*IDEs*). As a result, code editors for many different programming languages are available to be used via a web browser, like Codeanywhere, JSFiddle, Cloud9, codepad, JS Bin, CSSDesk, CodePen, repl.it, CodeBunk, Codenvy, and more.
Even developers of classic and famous development environments, like the Eclipse Foundation and Microsoft, have released cloud-based environments, respectively

[14]From **S. Germano**, F. Calimeri and E. Palermiti. 'LoIDE: A Web-Based IDE for Logic Programming Preliminary Report'. In: *Proceedings of PADL 2018*, pp. 152–160. DOI: 10.1007/978-3-319-73305-0_10.
And **S. Germano**, F. Calimeri and E. Palermiti. 'LoIDE: a web-based IDE for Logic Programming - Preliminary Technical Report'. In: *CoRR* abs/1709.05341 (2017). arXiv: 1709.05341.



Language: ASP
Solver: DLV
Options: Filter, color
Execution options
☑ Run automatically
Choose tab to execute
☑ Tab1

```
node(a). node(b). node(c).
arc(a,b). arc(b,c). arc(c,a).

color(N,red) | color(N,green) | color(N,blue) :- node(N).

:- color(N1,C), color(N2,C), arc(N1,N2).
```

```
DLV [build BEN/Dec 16 2012   gcc 4.6.1]

{color(a,red), color(b,green), color(c,blue)}

{color(a,green), color(b,red), color(c,blue)}

{color(a,red), color(b,blue), color(c,green)}

{color(a,blue), color(b,red), color(c,green)}

{color(a,green), color(b,blue), color(c,red)}

{color(a,blue), color(b,green), color(c,red)}
```

Figure 5.4.: An *ASP* program addressing a toy instance of the 3-colorability problem and the corresponding run performed by the DLV system via *Lo*IDE.

Eclipse Che (joint with Orion and Eclipse Dirigible) and Visual Studio Team Services (formerly Visual Studio Online), that include also powerful *IDE*s.

Editors for *Logic Programming* are no exception: several have been built, from very simple playgrounds like the LogiQL REQPL to more complex and complete editors like the IDP Web-IDE [167], SWISH [581] and the PDDL Editor [435].

In addition, specific editors for *ASP* programs like Clingo in the Browser, dlvhex Online Demo and Answer Set Programming for the Semantic Web - Tutorial have been proposed; however, these are quite "simplistic", and at an early stage of development and, furthermore, it is worth noticing that each web-editor for *Logic Programming* that has been introduced to date is intended for a specific language, or even for a specific solver. This raises some issues about interoperability and limits the usage of these tools.

*Lo*IDE is a web-based *IDE* for *Logic Programming* that explicitly addresses interoperability and flexibility, supporting multiple formalisms and solvers.

5.3.1 The *Lo*IDE project

The main goal of the *Lo*IDE project is the release of a modular and extensible web-*IDE* for *Logic Programming* using modern technologies and languages.

The *Lo*IDE *IDE* will provide advanced features specifically tailored for *Logic Programming*; it has been conceived in order to be extended over time and will include as many logic-based languages and solvers as possible.

A further goal of the project is to provide a web-service with a common set of *API*s for different logic-based languages; at the time of writing, this is still at an early stage of development.

*L*oIDE is provided as *open-source software* (*OSS*) and it is publicly available at

`https://github.com/DeMaCS-UNICAL/LoIDE`

Moreover, we released it as *Free Software* under the MIT License[15], with the explicit aim of helping the community of researchers and developers, that are free to study, use, distribute and even improve the project.
A prototypical running demo is available at

`https://www.mat.unical.it/calimeri/projects/loide`

Features of the *IDE*

The *L*oIDE *IDE* provides all the basic features of text and code editing that can be of use for *Logic Programming*.

We started from basic features available in *Ace*[16], a *JavaScript* embeddable code editor that constituted the base for *L*oIDE (see Section 5.3.2). Among them the most relevant, we mention here:

***indentation*:** automatically indent and outdent the code;

***document size*:** handles huge documents;

***key bindings*:** fully customizable key bindings (including vim and Emacs modes);

***search and replace*:** search and replace text via regular expressions;

***brace matching*:** highlight matching parentheses;

***mouse gestures*:** drag and drop text using the mouse;

***advanced cursors management*:** multiple cursors and selections;

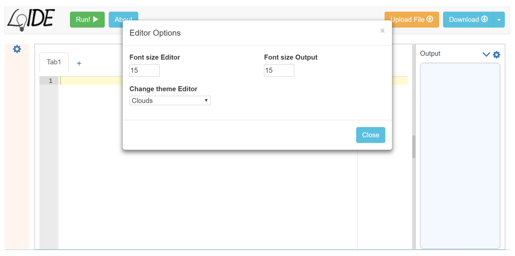
***clipboard management*:** cut, copy, and paste functionality;

***themes*:** over 20 themes available (standard `.tmtheme` files can be imported).

We extended such basic functionalities in order to properly meet the specific requirements of Logic Programs development.

Syntax highlighting. *Ace* supports syntax highlighting, already covering 110 languages; unfortunately, the logic-based languages we were interested in are not included. Relying on the specifications for cross-browser syntax highlighting, we

[15] `https://github.com/DeMaCS-UNICAL/LoIDE/blob/master/LICENSE`
[16] `https://ace.c9.io`



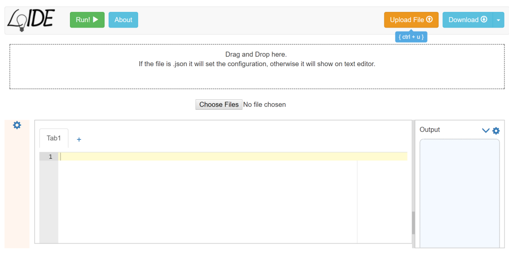
Editor Options

Font size Editor: 15

Font size Output: 15

Change theme Editor: Clouds

(a) The editor appearance options.



Drag and Drop here.
If the file is .json it will set the configuration, otherwise it will show on text editor.

No file chosen

(b) The File Import functionality.

Figure 5.5.: *L*oIDE screenshots.

introduced a basic support for *ASP* programs, and plan to include other languages as soon as the support for their specific solvers will be added to *L*oIDE.

Editor layout and appearance. The user can customize layout and appearance of the "Input" and the "Output" fields of the *IDE*. The user can change theme and fonts of each part of the interface, independently (as shown in Figure 5.5a). Moreover, size and position of the two fields are customizable as well. It is worth noticing that the all the options are automatically saved in the Web Storage[17], in order to make the persistent also across different the current browser user sessions.

Output highlighting. One of the most annoying aspects of developing and testing logic programs in practice is the need for checking output in test cases: since output is often constituted of a (possibly long) list of instances of many predicates, it can be quite tricky. Most *ASP* solvers allow filtering predicates, but this does not solve the problem and it is not a very flexible solution. *L*oIDE features an ad-hoc output highlighting: when the user selects an element of the output (for instance, a predicate name), all the elements with the same "name" will be automatically highlighted. The user can dynamically play with such highlighting and, as a consequence, the analysis of the results might be dramatically simplified.[18]

Keyboard shortcuts. *L*oIDE supports several many keyboard shortcuts. We properly extended the typical code-editors shortcuts provided by *Ace*[19] for a) Line operations, b) Selection, c) Multi-cursors, d) Go-to, e) Find/Replace, f) Undo/Redo, g) Indentation, h) Comments and i) Word/Character variations; adding specific shortcuts to Save and Load files and to Run the logic program.

Furthermore, we implemented other custom features around the *Ace*-based stem.

[17] http://www.w3.org/TR/webstorage
[18] More features for improving comprehension of the results will be added, such as different forms of visualization
[19] https://github.com/ajaxorg/ace/wiki/Default-Keyboard-Shortcuts

Multiple file support. When dealing with real-world problems in practice, logic programs often results to be split in more than one file (think, for instance, of the obvious separation between problem specification and problem instances). *L*oIDE explicitly supports multiple files management: the user can create and manage many different tabs, and also selectively decide which one has to be composed into the actual program to run.

Options. Settings of *L*oIDE can be customized, along with the behaviour of the underlying systems of use. The user can select the logic language of choice; for each language, the solver to be used to run the program can be set as well. Moreover, specific options can be selected for each solver, with predefined typical settings available for the most common. There are also more general options; for instance, the user can ask to automatically run the program at the end of each statement, so that the output dynamically changes as the user is crafting the program; this increases the interaction with the system. Furthermore, it might significantly ease the development of non-trivial programs, and be of great help in educational settings (think about a *Logic Programming* class, for instance).

Import and Export files. The content of the editor, all options and outputs can be downloaded locally to the device of use as *JSON* files, and later restored, possibly over a different device (also by means of Drag-and-Drop, if the device supports it, as shown in Figure 5.5b). Such feature is crucial for practically provide the user with a working environment which is virtually immaterial and free from specific physical workstations. Of course, also the logic program being edited can be saved.

5.3.2 Implementation

We provide next some insights on the design and implementation of *L*oIDE.

General Architecture

The system architecture, relying on a typical client-server framework, is draft in Figure 5.6.

The back-end (or *server-side* component) consists of the main *L*oIDE Web-Server, developed using the *JavaScript* runtime environment *Node.js*®[20], which exposes *API* that can be employed by a client.

[20] https://nodejs.org

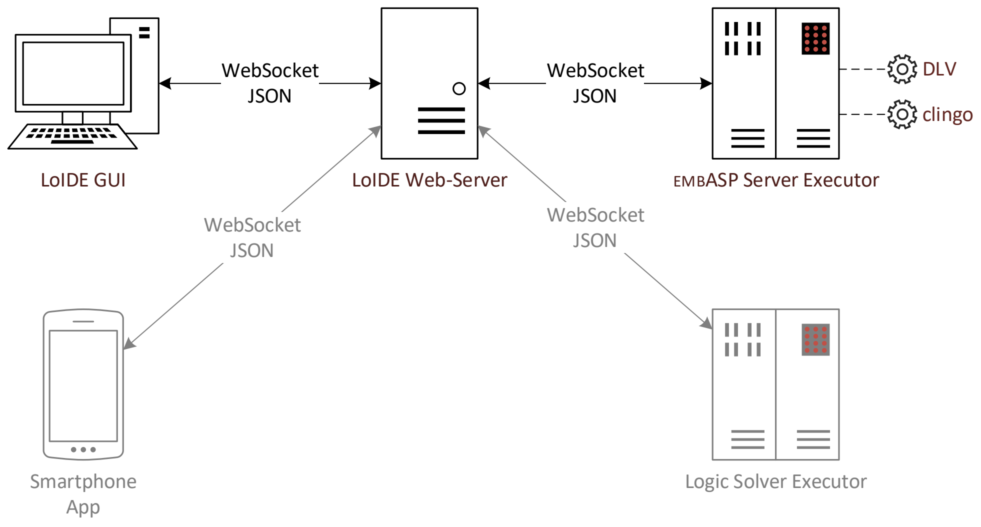


Figure 5.6.: Architecture of the *Lo*IDE project.

The front-end (or *client-side* component) consists of the *Graphical User Interface* (*GUI*) of the *IDE*, developed using modern web technologies such as *HTML5*, *CSS3* and the *JavaScript* programming language.

The execution of the logic solvers is not performed directly from within the main *Lo*IDE web-server; a dedicated component, the ***EMBASP Server Executor***, is in charge of this, instead. This choice is due to the aim of keeping the system modular and extensible; indeed, such modularity allows to easily support different "executors" and ease the management of additions, upgrades, and security issues.

All the components communicate using the *WebSocket*[21] communication protocol and the *JSON*[22] data-interchange format.

Thanks to the architecture of the project and the use of standard and common technologies between all the components, modules can be added or modified in a straightforward way while maintaining the scalability of the whole architecture as can be seen in the lower part of Figure 5.6.

Back-end – The *Lo*IDE Web-Server

The *Lo*IDE Web-Server has been developed using Node.js.

[21] `https://tools.ietf.org/html/rfc6455`
[22] `http://www.json.org`

In order to effectively use *WebSockets*, we relied on the socket.io[23] package to enables real-time bidirectional event-based communication between the client and the server; the package provided us with means for enabling several useful features, such as Reliability, Auto-reconnection support and Disconnection detection.

***LoIDE APIs*.** As mentioned before, one of the aims of the *Lo*IDE project is to develop a set of (Web) *APIs* for easily and efficiently controlling different solvers over different *Logic Programming* languages, using the *WebSocket* protocol and the *JSON* format. Specifications and implementation are at the first stage; currently, a call type is available, that given the description of the `language`, the `solver`, the list of `options` and the `program`, executes the solver over the program and returns either the `output` of the solver or any `error` messages. See *Lo*IDE *API* documentation[24] for further details.

Front-end – The *Lo*IDE *GUI*

The front-end uses modern standard web technologies (*HTML*5, *CSS*3, *JavaScript*); hence, *Lo*IDE is compatible with virtually any device currently available. We used some popular frameworks and libraries with the aim of improving user experience and making the *IDE* robust and powerful.

***Ace*[25]**

Ace is a *JavaScript* embeddable code editor. It matches the features and performance of native editors such as Sublime, Vim and TextMate and it can be easily embedded in any web page and *JavaScript* application. *Ace* is maintained as the primary editor for Cloud9 IDE and is the successor of the Mozilla Skywriter (Bespin) project. *Ace* is a community project and its source code is hosted on *GitHub* and released under the BSD license.

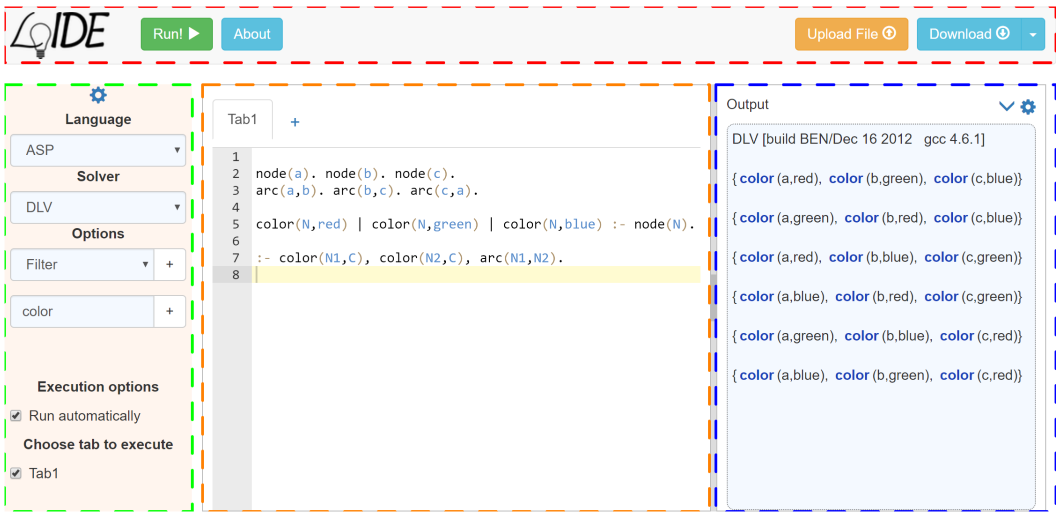
Bootstrap[26]

Bootstrap is the most popular front-end component library framework for developing responsive, mobile-first projects on the web. Bootstrap is an open source toolkit for developing with *HTML*, *CSS*, and *JavaScript* and its source code is hosted on *GitHub* and released under the MIT license.

jQuery and its UI Layout plugin[27]

jQuery is a fast, small, and feature-rich *JavaScript* library. It makes things like *HTML* document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use *API* that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has

[23]https://socket.io
[24]https://github.com/DeMaCS-UNICAL/LoIDE/wiki/APIs
[25]From https://ace.c9.io
[26]From https://getbootstrap.com
[27]From https://jquery.com



Language: ASP
Solver: DLV
Options: Filter
color
Execution options
☑ Run automatically
Choose tab to execute
☑ Tab1

```

node(a). node(b). node(c).
arc(a,b). arc(b,c). arc(c,a).

color(N,red) | color(N,green) | color(N,blue) :- node(N).

:- color(N1,C), color(N2,C), arc(N1,N2).

```

```
DLV [build BEN/Dec 16 2012   gcc 4.6.1]

{color(a,red), color(b,green), color(c,blue)}

{color(a,green), color(b,red), color(c,blue)}

{color(a,red), color(b,blue), color(c,green)}

{color(a,blue), color(b,red), color(c,green)}

{color(a,green), color(b,blue), color(c,red)}

{color(a,blue), color(b,green), color(c,red)}
```

Figure 5.7.: The main components of the web-based *GUI*.

changed the way that millions of people write *JavaScript*.

The jQuery UI Layout plugin[28] allows creating advanced UI layouts with sizeable, collapsible, nested panels and tons of options. It integrates with and enhances other UI widgets, like tabs, accordions and dialogs, to create rich interfaces.

bimap[29]

BiMap is a powerful, flexible and efficient *JavaScript* bidirectional map implementation. Enables fast insertion, search and retrieval of various kinds of data. A BiMap is like a two-sided *JavaScript* object with equally immediate access to both the keys and the values.

keymaster.js[30]

Keymaster is a simple micro-library for defining and dispatching keyboard shortcuts in web applications.

The web-based *GUI* is divided into 4 different parts (highlighted in Figure 5.7).

At the top the *navigation bar* is shown, highlighted in red in Figure 5.7, it contains proper *Run*, *Upload* and *Download* buttons.

In the middle of the interface, highlighted in orange in Figure 5.7, the *code editor* contains the editing tabs holding the program(s) to execute.

At the right side, highlighted in blue in Figure 5.7, the *output panel* dynamically shows the output of the computations and the link to the editor's layout options.

[28] http://plugins.jquery.com/layout
[29] From https://github.com/alethes/bimap
[30] From https://github.com/madrobby/keymaster

At the left side, highlighted in green in Figure 5.7, the *IDE options panel* contains all the options described in Section 5.3.1. This panel can be automatically toggled in order to save space for the main editor.

The layout is built using the Responsive Web Design (RWD) approach, i.e., it automatically adapts to the viewing environment and offers the possibility to be viewed on different devices with almost the same User Experience.

5.3.3 The EMBASP *Server Executor*

In order to decouple the web-requests management from logic-programming solvers execution, we developed **EMBASP *Server Executor*** as a completely different component; it is even implemented in a different programming language.

EMBASP *Server Executor* is a *Java* server application that is able to execute *ASP* programs with different solvers. It has the usual structure of a *Java* web-app with the following modules: (i) Control, (ii) Model, (iii) Service, and (iv) Resources. We do not discuss them in details, as the names are self-explanatory.

EMBASP *Server Executor* runs on top of Apache Tomcat®[31] and it exposes a set of *APIs* that can be used to invoke the solvers.

In order to execute the desired solver, it makes use of EMBASP[32] [233]. EMBASP is a framework for the integration (embedding) of *Logic Programming* in external systems for generic applications; it helps developers in designing and implementing complex reasoning tasks by means of solvers on different platforms.

Similarly to *Lo*IDE, **EMBASP *Server Executor*** is provided as *open-source software* (*OSS*) and it is publicly available at

https://github.com/DeMaCS-UNICAL/EmbASPServerExecutor

Moreover, it is likewise released as *Free Software* under the MIT License[33].

5.3.4 Related Works

The work herein presented is naturally comparable to other *Logic Programming IDEs* and other web-based editors.

[31] http://tomcat.apache.org
[32] https://www.mat.unical.it/calimeri/projects/embasp
[33] https://github.com/DeMaCS-UNICAL/EmbASPServerExecutor/blob/master/LICENSE

Several stand-alone, “native” editors and *IDE*s have been proposed for *Logic Programming* over different platforms; we refer the reader to the ample literature on the topic [114, 215, 345, 550, 553, 582]; moreover, many web-based editors have been recently introduced [167, 408, 435, 581].

All tools and environments share the same core of basic features, many of them are quite stable, some are already well-known. *L*oIDE, similarly to most web-based editors, has currently fewer features w.r.t. the “native” ones; however, even if quite young, it is stable effectively gives access to *Logic Programming* without the need for installing or downloading anything, from almost any platform connected to the Internet; furthermore, it could even run locally on any device featuring the Node.js runtime, with a few additional configuration steps.

Some tools (like SWISH, for instance) rely on platforms that provide functionalities via specific *API*s over *HTTP*; it is worth noting that this is not the case of *L*oIDE. Indeed, it started from *Answer Set Programming*, for which no such platforms were available, and makes use of the EMB**ASP** ***Server Executor***, implemented on purpose, that makes the project also more general and extensible.

All mentioned editors have peculiar, sometimes very interesting features; however, each one is tailored to a specific language and tightly coupled with some specific solver(s) in the back-end. On the other hand, the aim of *L*oIDE is to have a robust platform that seamlessly integrates different languages and different solvers. We do believe that this approach is more general, and could foster the use of *Logic Programming* in many contexts, especially in practical context and in education, also fruitfully promoting exchanges among the various communities in the *Logic Programming* area.

5.3.5 Future work

Even if already equipped with relevant features that make it effectively usable in practice, the project is still at an early stage of development; hence, we have already identified many future works and improvements.

We planned to extend the editing capabilities by a) language-based syntax checking, b) snippets and linting support, c) auto-complete/intellisense, d) plugins/extensions support, e) sharing of the programs in the cloud (for instance supporting Dropbox[34] and Gist[35] *API*s).

[34] `https://www.dropbox.com`
[35] `https://gist.github.com`

We are also planning to explicitly support the check for the “type” of file for each tab; indeed, it is a common practice in some logic languages to have different definitions in different files (think about the “domain text” and the “problem text” of *PDDL*).

Currently, *Lo*IDE sends the results back to the client when the solver finishes the job, given that the framework used by the EMBASP ***Server Executor*** (Section 5.3.3) does not handle “output streams”; however, the *Lo*IDE Web-Server easily allows serving back the results as soon as they are produced by the solver. An important improvement will allow such mechanism in order to save bandwidth and speed up the reception of the results from the solvers.

We plan to improve the deployment using container technologies (for instance Docker) and cloud-computing services (like *Amazon Web Services*, *Microsoft Azure* or *Google Cloud Platform*) in order to make it easier and more robust.

Advanced features like visualization techniques, such as [19, 150, 330, 354], will also further simplify output comprehension, and the possibility to save files in the user account or over cloud services will allow also pave the way to teamwork and collaborative editing.

We are working to support more *executors* (web-services), logic-based *languages* and *solvers* (engines), in order to increase the audience of the project; the addition of an interactive tutorial could also allow the users to become more familiar *Lo*IDE and with declarative programming.

Wrap-up

In this chapter we presented a comprehensive framework for the integration of logic-based formalisms in external systems for generic applications and a web-based *IDE* for *Logic Programming* meant to modular and flexible in order to support various languages and solvers.
It is worth noticing that some of the investigations reported in this chapter have been conducted inside the joint projects EMBASP and *Lo*IDE with other researchers of the Department of Computer Science (DeMaCS[36]) at the “Università della Calabria”.

In the next chapter we conclude with some considerations and overall insights.

[36]http://www.mat.unical.it

Conclusions

Throughout all the chapters of this Thesis, we have shown how *Logic Programming* can be effectively and successfully used in several different fields by developing custom solutions for the specific requirements of each of them.

We illustrated how the *Stream Reasoning* challenges of efficiently handling real-time *data streams* can be addressed using an adequate combination of logic-based reasoning technique and how this approach has been successfully applied to a large Project that involved *IoT* and *Smart City Applications*, taking into account the desideratum of the users and enabling an automatic configuration of these *Decision Support* tasks.

Then we described how some techniques, such as the “multi-engine” approach, can be useful to manage and reason on top of big amount of data and how *Logic Programming* can help to simplify and generalise these operations.

Also, we demonstrated how logic-based approaches are powerful enough to be used in many *AI* applications, such as video games, that require high-level and complex reasoning, and we proved their flexibility in a variety of problems.

Additionally, in order to productively use *Logic Programming* in as many scenarios as possible, we pointed out how the use of logic-based languages can be made easier and more intuitive, also for non-experts, and that could make these formalisms more accessible, thus fostering large spread.

There are though some gaps that are not closed yet, as specifically discussed in respective chapters. Nonetheless, this does not prevent the proposed solutions from being extended to other contexts and combined to make them more effective. As an example, the addition of a *process* step to the traditional *observe-think-act* cycle, shown in a previous chapter, can be adapted to other applications scenarios in the context of intelligent agents.

However, combining and adapting *Logic Programming* languages, engines/solvers, tools and techniques is often an extremely challenging task for several reasons.

First of all, the beauty of the declarativity and the logical foundations of these languages are paid in terms of efficiency. Even if nowadays many powerful solvers exist, they have to solve "complex" (from a Computational Complexity point of view) problems and often require huge amounts of computational resources. These "problems" make them much less efficient than tailored approaches and, often, unfeasible for real-world problems.

Furthermore, languages often have completely different approaches and, within the same language, solvers have very disparate interfaces and linguistic specificities that make hard to execute the same logic program with various engines. Moreover, they sometimes need programs to be written in a specific way in order to be executed efficiently. Even if some standardization efforts have been carried out, especially for certain formalisms, there are still many discrepancies.

In addition, the communication among distinct logic-based languages is not straightforward, and it is much harder, almost impossible, if we consider other programming paradigms. Furthermore, the lack of development and debugging tools make extremely troublesome the production and the testing of solution based on *Logic Programming*. Also, the execution of a specific logic-based solver might be problematic due to the considerable difficulties in installing and configuring (some of) them and the peculiarities of the solvers' languages.

Besides, characteristic requirements of modern *AI* applications, such as parallel or distributed computation, are almost never available in logic-based solutions and this makes difficult the development of such applications, or the inclusion in frameworks that are inherently designed for parallel execution. Additionally, most of the logic-based formalisms do not support specific types of values, such as floating-point numbers typical of graphic and simulation applications, or specific "data structures" that are commonly found in real-world data, such as sensors frequently used in *IoT* application.

It is worth noting that, as mentioned before, the aim of our work it is not to encourage the use *Logic Programming* everywhere, and we do not believe that *Logic Programming* is suitable for every reasoning/computational task. For this reason, we focused on the areas mentioned above; because we believe they contain some specific tasks that can be fruitfully tackled using *Logic Programming*. We strongly believe that is important to properly combine various reasoning methods and technologies. To illustrate this, we described some preliminary approaches that we have envisioned, designed and developed.

These results could (and should) be further improved by new investigations on well-defined problems in the areas mentioned in this Thesis, where logic-based approaches have been marginally considered. For instance, (i) in the *Stream Reasoning* area, as shown also in the preliminaries of the dedicated chapter, a lot of research has been carried out in order to have more efficient and scalable ways of processing *data streams* and in order to have a common and formal foundation in order to better define the reasoning tasks, and we believe that, as shown by the recent advancements in the area, logic can play a crucial role in this context; (ii) in the *AI* and games area, which is continuously growing and that is demanding always new theoretical and practical solutions, *Logic Programming* can be used for many different tasks and its well-known *KR&R* capabilities can be exploited in order to successfully develop Intelligent Agents.
Moreover, we strongly believe that making more "popular" and accessible the *Logic Programming* paradigm, as well as other declarative languages, can allow a wider discussion on the possibilities and the limits of this paradigm, and this will result in the development of more innovative and complete solutions that will significantly increase the number of users. This should trigger a virtuous cycle of innovation that will help the entire community to grow and improve.

A Popular *Artificial Intelligence* (*AI*) Competitions

Some of the most popular *AI* Competitions are:[1]

Angry Birds AI Competition[2]

The task of the *Angry Birds AI Competition* is to develop an intelligent *Angry Birds* playing agent that is able to successfully play the game autonomously and without human intervention. The long-term goal is to build *AI* agents that can play new levels better than the best human players. This may require analysing the structure of the objects and to infer how to shoot the birds in order to destroy the pigs and to score most points.

Angry Birds Level Generation Competition[3]

The goal of the *Angry Birds Level Generation Competition* is to build computer programs that can automatically create fun and challenging game levels (as shown in Figure A.1). The difficulty of this competition compared to similar competitions is that the generated levels must be stable under gravity, robust in the sense that a single action should not destroy large parts of the generated structure, and most importantly, the levels should be fun to play and challenging, that is, difficult but solvable.

This Competition evaluates each level generator based on the overall fun or enjoyment factor of the levels it creates. Aside from the main prize for "most enjoyable levels", two additional prizes for "most creative levels" and "most challenging levels" are also awarded.

Dota2 Bot Competition[4]

The *Dota2 Bot Competition* is about writing controllers (bots) for the popular team-based MOBA Dota2. The focus is particularly the collaboration between agents. The competition uses the original game with a custom-made mod, that allows external processes to control the player's hero. Each bot controls one hero and has to coordinate with its teammates – without any backchannel communication. No other communication besides the chat wheel or map pings is permitted.

[1]From http://www.cig2017.com/competitions-cig-2017
[2]From https://aibirds.org/angry-birds-ai-competition.html
[3]From https://aibirds.org/other-events/level-generation-competition.html
[4]From https://github.com/lightbringer/dota2ai
[5]From http://www.gvgai.net

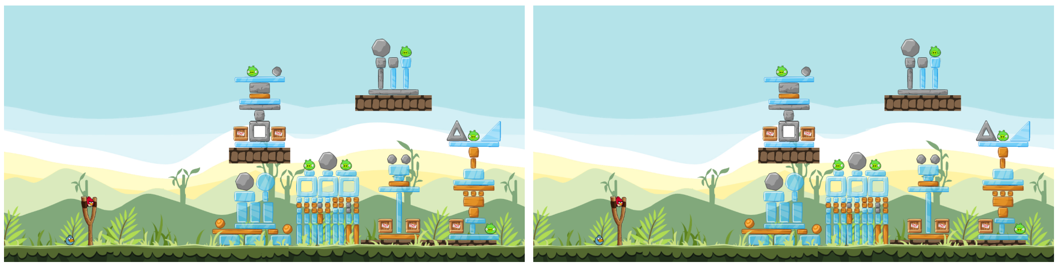


Figure A.1.: Generated levels in the *Angry Birds Level Generation Competition*. From [542].

General Video Game AI Competition[5]

The *GVG-AI Competition* explores the problem of creating controllers for general video game playing. The key questions in this Competition are:
How would you create a single agent that is able to play any game it is given?
Could you program an agent that is able to play a wide variety of games, without knowing which games are to be played?
Can you create an automatic level generation that designs levels for any game is given?

Malmo Collaborative AI Challenge[6]

The *Malmo Collaborative AI Challenge* is designed to encourage research relating to various problems in Collaborative *AI*. The Challenge takes the form of a collaborative mini-game in which players need to work together to achieve a common goal. Challenge participants are invited to develop and train collaborative *AI* solutions that learn to achieve high scores across a range of partners. This Challenge uses Project Malmo[7] [315], an *AI* experimentation and research platform that is built on top of the popular Minecraft™game.

RoboCupSoccer – Simulation League[8]

The *RoboCupSoccer – Simulation League* focus on *AI* and team strategy.
Two teams of eleven autonomous and independently moving software players (called agents) play soccer in a virtual soccer stadium represented by a central server. This server knows everything about the game, i.e. the current position of all players and the ball, the physics and so on. The game further relies on the communication between the server and each agent. On the one hand, each player receives relative and noisy input of his virtual sensors (visual, acoustic and physical) and may, on the other hand, perform some basic commands (like dashing, turning or kicking) in order to influence its environment.
There are 2 sub-leagues: 2D and 3D (as shown in Figure A.2).

[6]From `https://www.microsoft.com/en-us/research/academic-program/collaborative-ai-challenge`
[7]`https://www.microsoft.com/en-us/research/project/project-malmo`
[8]From `http://www.robocup.org/leagues/23`
[9]From `http://game.engineering.nyu.edu/showdown-ai-competition`

(a) The 2D sub-league.

(b) The 3D sub-league.

Figure A.2.: Screenshots of the RoboCupSoccer Simulation League. From http://www.robocup.org.

Showdown AI Competition[9]

The *Showdown AI Competition* is a game-based *AI* competition built around a clone of the popular game Pokémon™. This game is a turn-based team battle, where the objective is to defeat an opponent team using clever combinations of creatures and their abilities. The gameplay is reminiscent of computer role-playing game battles and collectable card games. Characteristics, such as the combination of turn-based gameplay and partial observability, are unusual in current game-based *AI* competitions and therefore offers a fresh challenge.

StarCraft AI Competition[10]

IEEE CIG StarCraft Competitions have seen quite some progress in the development and evolution of new StarCraft® bots. For the evolution of the bots, participants used various approaches for making *AI* bots and it has fertilized game *AI* and methods such as HMM, Bayesian model, CBR, Potential fields, and reinforcement learning. However, it is still quite challenging to develop *AI* for the game because it should handle a number of units and buildings while considering resource management and high-level tactics. The purpose of this competition is developing *RTS*s game *AI* and solving challenging issues on *RTS*s game *AI* such as uncertainty, real-time process, managing units.

Note that StarCraft® is a very popular Game for *AI* Competition, therefore many of them are organized by universities worldwide (for instance the *AIIDE Starcraft AI Competition* or the *Student StarCraft AI Tournament & Ladder*).

Visual Doom AI Competition[11]

The *Visual Doom AI Competition* is about writing a controller (bot) that plays Doom from pixels. The screen buffer accessed in real-time is the only information the agent can base its decision on. The winner of the Competition is determined by a multiplayer death-match tournament. Although the participants are allowed to use any technique to develop a controller, the design and efficiency of the framework allow and encourage the use machine learning methods such as reinforcement deep learning.

[10]From http://cilab.sejong.ac.kr/sc_competition
[11]From http://vizdoom.cs.put.edu.pl

B

Benchmark and Competition Results - *Angry Birds AI Competition* (*AIBIRDS*)

Benchmark Results

2013[1]

Rank	Team	Total score	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10	Level 11	Level 12	Level 13	Level 14	Level 15	Level 16	Level 17	Level 18	Level 19	Level 20	Level 21
1	AngryHex	974670	30390	54160	41890	28000	64440	24990	36900	24860	49570	50570	53510	57750	42010	58190	46550	63430	46820	50020	**38460**	46970	65190
2	WISC	963160	29030	52180	41950	27850	65810	24640	25520	55810	33680	39170	50000	54980	32450	65060	49200	63430	**54750**	44740	37370	47980	67560
3	Angry Concepts	954030	29410	52250	41320	28160	64160	15660	24630	47170	48670	48290	56130	**58600**	**50360**	58050	43340	55000	45990	44030	37880	43730	61200
4	Beau Rivage	952390	29760	43160	40500	28680	61000	33540	45780	49150	33110	42200	57420	55600	35190	62770	47270	55890	49880	38820	29780	37020	**75870**
5	HungryBirds	951440	**31210**	53860	42040	20980	65700	28430	40580	27020	50710	53070	56260	55410	30130	59780	41040	55310	43530	46770	32690	**56050**	60870
6	Luabab	894840	29600	52180	40760	28030	66100	16860	32180	48330	41840	54110	45580	53040	41110	65640	28910	65670	39550	37680	16820	36970	53880
7	Dan	893370	29210	42760	41610	27990	63840	25700	45990	23390	45930	49570	38570	54990	32270	57550	47280	63000	42770	48290	22040	36910	53710
8	Black Forest Cuckoos	874650	29880	52390	41600	28980	65380	0	32380	56310	23050	52360	54360	57750	29770	57560	**55300**	59940	52920	41410	36940	46370	0
9	IHSEV	861830	29360	52780	41240	36810	51860	33790	30130	57570	24310	29860	**59070**	40960	42840	**65640**	30700	51730	47940	47320	30940	0	56980
10	Naïve Agent	858730	29510	52230	40620	20680	55160	16070	21590	25730	35490	32600	46760	54070	49470	50590	46430	55210	48140	49430	37920	36790	54240
11	Sniper	785520	29760	42720	40500	18620	**65850**	35490	28780	48170	26960	35540	51150	49190	26180	**65640**	0	56530	37980	40460	30390	55610	0
12	Wanderer	774190	30840	**60400**	41900	**36770**	63550	26910	22070	**57780**	**51480**	**68740**	30620	42320	20050	58200	25520	38960	37560	35220	25300	0	0
13	s-birds	754190	30080	26240	**42240**	19050	38160	**36180**	**49120**	38340	41670	54940	57950	47460	22550	63560	42990	65750	38790	39120	0	0	0
14	Lambdaers	705010	30970	52390	42070	19580	63750	26400	45880	57600	49370	61820	44120	38390	29350	54340	44040	0	44940	0	0	0	0
15	A.Wang	680640	22400	43320	31910	28080	48440	24770	23960	27580	42590	57610	47410	46980	27490	**65640**	36160	**66550**	39750	0	0	0	0
16	Akbaba	641080	28640	35540	41910	19120	47470	0	23880	45380	36960	56000	48460	52310	0	59420	0	47230	44260	**54500**	0	0	0
17	ObjectS	569000	29670	43200	40620	11720	**65850**	15570	21310	37170	22610	51250	57930	45200	26570	**65640**	34690	0	0	0	0	0	0
18	AngryPKU	476100	18420	33280	24070	10120	35650	26450	0	27840	24660	30160	31740	45870	0	45640	32790	55890	0	0	33520	0	0
19	JulyPlayer	352340	29360	52520	35360	20690	46300	17400	21810	45370	48670	34860	0	0	0	0	0	0	0	0	0	0	0
20	FEI2	211870	28360	34160	30470	27650	64710	26520	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	High scores	1134920	31210	60400	42240	36770	65850	36180	49120	57780	51480	68740	59070	58600	50360	65640	55300	66550	54750	54500	38460	56050	75870
	2012 High scores	1080390	33340	60880	42880	38160	67630	32550	31240	48270	51570	58530	60420	63440	36940	65790	48200	66180	55150	57320	38720	51440	71740
	3 stars score	1042000	32000	60000	41000	28000	64000	35000	45000	50000	50000	55000	54000	45000	47000	70000	41000	64000	53000	48000	35000	50000	75000

Table B.1.: Benchmark Results 2013 - *AIBIRDS*.

2013–2014[2]

Rank	Team	Total Score	Level 1-1	Level 1-2	Level 1-3	Level 1-4	Level 1-5	Level 1-6	Level 1-7	Level 1-8	Level 1-9	Level 1-10	Level 1-11	Level 1-12	Level 1-13	Level 1-14	Level 1-15	Level 1-16	Level 1-17	Level 1-18	Level 1-19	Level 1-20	Level 1-21
1	**PlanA+**	1002380	30480	**62370**	40620	29000	69440	**36970**	32020	47320	26440	56830	47240	58210	34010	**65640**	54910	57530	51190	52120	39440	45980	64620
2	**DataLab Birds**	981120	31620	52000	41890	19790	70320	15700	45720	43190	50420	56790	50650	53420	32010	55640	46450	57380	48570	45730	35470	54680	73680
3	AngryHex (2013)	974670	30390	54160	41890	28000	64440	24990	36900	24860	49570	50570	53510	57750	42010	58190	46550	63430	46820	50020	38460	46970	65190
4	WISC	963160	29030	52180	41950	27850	65810	24640	25520	55810	33680	39170	50000	54980	32450	65060	49200	63430	**54750**	44740	37370	47980	67560
5	**AngryHex (2014)**	960320	**32660**	52580	41910	19690	68090	25180	24680	43330	42740	54890	53570	54860	41200	57150	41100	61470	50260	48050	36780	39010	71120
6	Angry Concepts	954030	29410	52250	41320	28160	64160	15660	24630	47170	48670	48290	56130	58600	**50360**	58050	43340	55000	45990	44030	37880	43730	61200
7	Beau Rivage	952390	29760	43160	40500	28680	61000	33540	45780	49150	33110	42200	57420	55600	35190	62770	47270	55890	49880	38820	29780	37020	**75870**
8	HungryBirds	951440	31210	53860	42040	20980	65700	28430	40580	27020	50710	53070	56260	55410	30130	59780	41040	55310	43530	46770	32690	**56050**	60870
9	**AngryBER**	935330	28510	43420	41910	19390	62880	35610	31980	45050	48670	51980	51550	54720	42510	45640	47090	50000	47740	44110	36450	35990	70130
10	**RMIT RedBacks**	933120	32220	53860	30480	19380	**70350**	33470	45740	29180	33320	59190	45250	56910	42300	45640	41540	**66570**	39370	48530	30460	40760	68600
11	Luabab	894840	29600	52180	40760	28030	66100	16860	32180	48330	41840	54110	45580	53040	41110	**65640**	28910	65670	39550	37680	16820	36970	53880
12	Dan	893370	29210	42760	41610	27990	63840	25700	45990	23390	45930	49570	38570	54990	32270	57550	47280	63000	42770	48290	22040	36910	53710
13	**IHSEV (2014)**	891590	29520	53070	41440	21860	56470	33470	29920	36630	40620	32300	31570	**61070**	41330	59250	34830	54160	47830	42840	**40100**	40470	62840
14	**Impact Vactor**	886990	29710	60480	42040	19700	65440	33870	36880	25720	34770	46870	57490	55230	49620	0	45860	58870	47070	48210	25040	37650	66470
15	Black Forest Cuckoos	874650	29880	52390	41600	28980	65380	0	32380	56310	23050	52360	54360	57750	29770	57560	**55300**	59940	52920	41410	36940	46370	0
16	IHSEV (2013)	861830	29360	52780	41240	**36810**	51860	33790	30130	57570	24310	29860	**59070**	40960	42840	**65640**	30700	51730	47940	47320	30940	0	56980
17	Naïve Agent (2013)	858730	29510	52230	40620	20680	55160	16070	21590	25730	35490	32600	46760	54070	49470	50590	46430	55210	48140	49430	37920	36790	54240
18	Sniper	785520	29760	42720	40500	18620	65850	35490	28780	48170	26960	35540	51150	49190	26180	65640	0	56530	37980	40460	30390	55610	0
19	Wanderer	774190	30840	60400	41900	36770	63550	26910	22070	**57780**	**51480**	**68740**	30620	42320	20050	58200	25520	38960	37560	35220	25300	0	0
20	**Naïve Agent (2014)**	756680	29760	43250	40180	10590	62490	14980	22150	35840	36050	52570	39310	48660	30000	45640	44190	52300	39530	39590	29460	40140	0
21	s-birds	754190	30080	26240	**42240**	19050	38160	36180	**49120**	38340	41670	54940	57950	47460	22550	63560	42990	65750	38790	39120	0	0	0
22	**S-birds Avengers**	706110	28520	51770	35380	27950	60920	26000	0	48010	49050	0	48320	45150	22130	53760	41240	0	45000	45750	38200	38960	0
23	Lambdaers	705010	30970	52390	42070	19580	63750	26400	45880	57600	49370	61820	44120	38390	29350	54340	44040	0	44940	0	0	0	0
24	A.Wang	680640	22400	43320	31910	28080	48440	24770	23960	27580	42590	57610	47410	46980	27490	**65640**	36160	66550	39750	0	0	0	0
25	Akbaba	641080	28640	35540	41910	19120	47470	0	23880	45380	36960	56000	48460	52310	0	59420	0	47230	44260	**54500**	0	0	0
26	**AngryDragons**	620390	28120	33440	30430	28820	65650	0	0	29340	25590	0	39580	51340	30840	55640	33920	53800	45660	38390	29830	0	0
27	ObjectS	569000	29670	43200	40620	11720	65850	15570	21310	37170	22610	51250	57930	45200	26570	**65640**	34690	0	0	0	0	0	0
28	AngryPKU	476100	18420	33280	24070	10120	35650	26450	0	27840	24660	30160	31740	45870	0	45640	32790	55890	0	0	33520	0	0
29	JulyPlayer	352340	29360	52520	35360	20690	46300	17400	21810	45370	48670	34860	0	0	0	0	0	0	0	0	0	0	0
30	FEI2	211870	28360	34160	30470	27650	64710	26520	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	High Scores	1146160	32660	62370	42240	36810	70350	36970	49120	57780	51480	68740	59070	61070	50360	65640	55300	66570	54750	54500	38460	56050	75870
	2013 Highscores	1134920	31210	60400	42240	36810	65850	36180	49120	57780	51480	68740	59070	58600	50360	65640	55300	66550	54750	54500	38460	56050	75870
	3 stars	1042000	32000	60000	41000	28000	64000	35000	45000	50000	50000	55000	54000	45000	47000	70000	41000	64000	53000	48000	35000	50000	75000

Table B.2.: Benchmark Results 2013–2014 overall - *AIBIRDS*.

[1] http://aibirds.org/past-competitions/2013-competition/benchmarks.html
[2] https://aibirds.org/past-competitions/2014-competition/benchmarks.html

2013–2014–2016[3]

Rank	Team	Total Score	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9	Level 10	Level 11	Level 12	Level 13	Level 14	Level 15	Level 16	Level 17	Level 18	Level 19	Level 20	Level 21
1	DataLab (2016)	1018230	29890	52180	41910	27960	58210	25120	45820	44830	51080	52570	56420	54750	41510	64940	38740	63840	50620	46610	38300	53170	79760
2	PlanA+ (2014)	1002380	30480	**62370**	40620	29000	69440	**36970**	32020	47320	26440	56830	47240	58210	34010	65640	54910	57530	51190	52120	39440	45980	64620
3	DataLab (2014)	981120	31620	52000	41890	19790	70320	15700	45720	43190	50420	56790	50650	53420	32010	55640	46450	57380	48570	45730	35470	54680	73680
4	HeartyTian	979050	28590	52120	41910	20630	67110	16960	45820	37810	49320	52660	56420	53220	49180	58200	31470	61450	51010	44990	37550	53360	69270
5	AngryHex (2013)	974670	30390	54160	41890	28000	64440	24990	36900	24860	49570	50570	53510	57750	42010	58190	46550	63430	46820	50020	38460	46970	65190
6	Datalab (2015)	973300	31430	52120	41910	27970	63660	34480	46290	54830	41490	49690	35080	51150	48130	58200	46530	58930	44870	44520	37380	42640	62000
7	WISC	963160	29030	52180	41950	27850	65810	24640	25520	55810	33680	39170	50000	54980	32450	65060	49200	63430	**54750**	44740	37370	47980	67560
8	AngryHex (2014)	960320	**32660**	52580	41910	19690	68090	25180	24680	43330	42740	54890	53570	54860	41200	57150	41100	61470	50260	48050	36780	39010	71120
9	Angry Concepts	954030	29410	52250	41320	28160	64160	15660	24630	47170	48670	48290	56130	58600	**50360**	58050	43340	55000	45990	44030	37880	43730	61200
10	Beau Rivage	952390	29760	43160	40500	28680	61000	33540	45780	49150	33110	42200	57420	55600	35190	62770	47270	55890	49880	38820	29780	37020	75870
11	HungryBirds	951440	31210	53860	42040	20980	65700	28430	40580	27020	50710	53070	56260	55410	30130	59780	41040	55310	43530	46770	32690	**56050**	60870
12	UFC	947680	31940	61680	**43480**	36810	66880	36080	45880	**59120**	49790	42630	**59280**	58190	48090	65640	47090	62230	45410	**56860**	30600	0	0
13	BamBirds	942790	30680	51420	42530	28600	65630	25260	23560	33220	50310	40570	45210	55940	36860	57740	50740	56420	49940	49740	32890	48250	67280
14	SeaBirds	942726	29540	53840	33490	29160	65620	17660	30590	58170	34970	56390	58250	53500	49050	55640	40606	61880	42290	40530	23110	39210	69230
15	AngryHex (2015)	940630	**32660**	52580	41910	21590	62940	33900	30110	46030	22960	63860	53570	54860	32220	53520	47120	55710	50260	42070	37040	44730	60990
16	AngryBER	935330	28510	43420	41910	19390	62880	35610	31980	45050	48670	51980	51550	54720	42510	45640	47090	50000	47740	44110	36450	35990	70130
17	RedBacks	933120	32220	53860	30480	19380	**70350**	33470	45740	29180	33320	59190	45250	56910	42300	45640	41540	**66570**	39370	48530	30460	40760	68600
18	PlanA+ (2015)	929060	30940	51450	40620	28100	48740	36690	22680	0	40970	62060	56610	57850	21260	65640	54910	56440	47540	52520	37550	35090	**81400**
19	AngryHex (2016)	906160	31520	52320	41870	29080	68410	35360	0	47860	46850	62670	44080	56280	49820	**73960**	46760	55640	51250	40840	28480	43110	0
20	Luabab	894840	29600	52180	40760	28030	66100	16860	32180	48330	41840	54110	45580	53040	41110	65640	28910	65670	39550	37680	16820	36970	53880
21	Dan	893370	29210	42760	41610	27990	63840	25700	45990	23390	45930	49570	38570	54990	32270	57550	47280	63000	42770	48290	22040	36910	53710
22	IHSEV (2014)	891590	29520	53070	41440	21860	56470	33470	29920	36630	40620	32300	31570	61070	41330	59250	34830	54160	47830	42840	**40100**	40470	62840
23	IHSEV (2015)	889090	28810	61390	41820	28160	63880	33820	45740	25770	33230	64270	55220	46830	48050	60100	42320	63760	46870	43380	20730	34940	0
24	Impact Vactor	886990	29710	60480	42040	19700	65440	33870	36880	25720	34770	46870	57490	55230	49620	0	45860	58870	47070	48210	25040	37650	66470
25	Tori	885450	32130	52700	41870	19410	66910	15780	21800	47820	33870	32880	40750	**62290**	33560	69970	45410	65760	48830	39360	0	46670	67680
26	Black Forest Cuckoos	874650	29880	52390	41600	28980	65380	0	32380	56310	23050	52360	54360	57750	29770	57560	**55300**	59940	52920	41410	36940	46370	0
27	IHSEV (2013)	861830	29360	52780	41240	**36810**	51860	33790	30130	57570	24310	29860	59070	40960	42840	65640	30700	51730	47940	47320	30940	0	56980
28	Naïve Agent (2013)	858730	29510	52230	40620	20680	55160	16070	21590	25730	35490	32600	46760	54070	49470	50590	46430	55210	48140	49430	37920	36790	54240
29	Naïve Agent (2016)	855370	29800	52610	40260	19000	54660	33830	23440	57840	25900	40750	57220	57240	29740	58340	49050	60050	47970	41590	38760	37320	0
30	Naïve Agent (2015)	838590	29760	34230	40850	13440	57640	24590	24590	43360	50710	59970	52950	55720	23070	57940	48710	63780	39280	46370	35850	35780	0
31	IHSEV (2016)	812060	28830	61390	42210	11290	67510	33820	20640	34300	40850	49980	39690	59360	34870	57860	51400	62980	41350	39240	34490	0	0
32	Sniper	785520	29760	42720	40500	18620	65850	35490	28780	48170	26960	35540	51150	49190	26180	65640	0	56530	37980	40460	30390	55610	0
33	Wanderer	774190	30840	60400	41900	36770	63550	26910	22070	57780	**51480**	**68740**	30620	42320	20050	58200	25520	38960	37560	35220	25300	0	0
34	Naïve Agent (2014)	756680	29760	43250	40180	10590	62490	14980	22150	35840	36050	52570	39310	48660	30000	45640	44190	52300	39530	39590	29460	40140	0
35	s-birds	754190	30080	26240	42240	19050	38160	36180	**49120**	38340	41670	54940	57950	47460	22550	63560	42990	65750	38790	39120	0	0	0
36	S-birds Avengers	706110	28520	51770	35380	27950	60920	26000	0	48010	49050	0	48320	45150	22130	53760	41240	0	45000	45750	38200	38960	0
37	Lambdaers	705010	30970	52390	42070	19580	63750	26400	45880	57600	49370	61820	44120	38390	29350	54340	44040	0	44940	0	0	0	0
38	Adil	683380	29950	51960	42820	0	69790	36150	31340	54890	48970	0	0	60670	0	71430	32910	61770	51480	0	39250	0	0
39	A.Wang	680640	22400	43320	31910	28080	48440	24770	23960	27580	42590	57610	47410	46980	27490	65640	36160	66550	39750	0	0	0	0
40	Akbaba	641080	28640	35540	41910	19120	47470	0	23880	45380	36960	56000	48460	52310	0	59420	0	47230	44260	54500	0	0	0
41	AngryDragons	620390	28120	33440	30430	28820	65650	0	0	29340	25590	0	39580	51340	30840	55640	33920	53800	45660	38390	29830	0	0
42	sBirds-returns	595530	28290	33890	41910	10230	56220	17860	22420	47820	51390	0	30560	56790	28830	0	0	46260	46910	0	31280	44870	0
43	ObjectS	569000	29670	43200	40620	11720	65850	15570	21310	37170	22610	51250	57930	45200	26570	65640	34690	0	0	0	0	0	0
44	AngryPKU	476100	18420	33280	24070	10120	35650	26450	0	27840	24660	30160	31740	45870	0	45640	32790	55890	0	0	33520	0	0
45	sBirds (2016)	461450	29760	43330	40850	19540	55070	15980	0	0	41180	47610	52040	47690	29950	0	38450	0	0	0	0	0	0
46	JulyPlayer	352340	29360	52520	35360	20690	46300	17400	21810	45370	48670	34860	0	0	0	0	0	0	0	0	0	0	0
47	FEI2	211870	28360	34160	30470	27650	64710	26520	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	3 stars	1042000	32000	60000	41000	28000	64000	35000	45000	50000	50000	55000	54000	45000	47000	70000	41000	64000	53000	48000	35000	50000	75000
	High Scores	1168020	32660	62370	43480	36810	70350	36970	49120	59120	51480	68740	59280	62290	50360	73960	55300	66570	54750	56860	40100	56050	81400

Table B.3.: Benchmark Results 2013–2014–2016 overall - *AIBIRDS*.

Competition Results

2013[4]

Qualification Rounds		
1	**Angry-HEX**	584600
2	Luabab	570530
3	Beau-Rivage	537380
4	Angry Concepts	532310
5	Naive Agent	506070
6	Hungry Birds	474810
7	Team Wisc	452110
8	IHSEV	435950
9	A. Wang	416530
10	Dan	396400
11	s-birds	375480
12	LAMBDAers	318320
13	Akbaba	294460
14	OBJECT-S	292770
15	Black Forest Cuckoos	286800
16	The Snipers	278420
17	AngryPKUers	278230
18	JulyPlayer	257060
19	Wanderer	216630
20	FEI2	140660

Quarter Final 1		
1	**Angry-HEX**	283970
2	Beau Rivage	179730
3	Naive	93090
4	IHSEV	85560

Quarter Final 2		
1	Team Wisc	209170
2	Angry Concepts	207300
3	NLuabab	190310
4	Hungry Birds	151480

Semi Final		
1	Angry Concepts	510740
2	Beau Rivage	331490
3	Team Wisc	286080
4	**Angry-HEX**	200830

Grand Final		
1	Beau Rivage	91140
2	Angry Concepts	0

Third Place Final		
1	Team Wisc	228560
2	**Angry-HEX**	75790

Table B.4.: 2013 Competition Results - *AIBIRDS*.

[3] `http://aibirds.org/benchmarks.html` (this *URL* will probably change in the next years, following a similar structure to the other links to Benchmark Results)

[4] `https://aibirds.org/past-competitions/2013-competition/results.html`

2014[5]

Qualification Rounds		
1	DataLab Birds	423280
2	PlanA+	372810
3	s-Birds Avengers	361770
4	Angry Dragons	317300
5	Naïve	302710
6	Impact Vactor	298390
7	**Angry-HEX**	294170
8	IHSEV	292380
9	Angry BER	253820
10	BeauRivage	238080
11	RMIT Redbacks	188890
12	Auto Lilienthal	0

Quarter Final 1		
1	DataLab Birds	346260
2	Angry BER	224860
3	Impact Vactor	173710
4	s-birds Avengers	167860

Quarter Final 2		
1	PlanA+	360920
2	IHSEV	277530
3	**Angry-HEX**	129610
4	Angry Dragons	78970

Semi Final		
1	DataLab Birds	232790
2	Angry BER	206680
3	PlanA+	206620
4	IHSEV	93100

Grand Final		
1	DataLab Birds	406340
2	Angry BER	243880

Table B.5.: 2014 Competition Results - *AIBIRDS*.

2015[6]

Qualification Rounds		
1	IHSEV	405230
2	**Angry-HEX**	341800
3	Datalab Birds	335320
4	PlanA+	308130
5	S-Birds Returns	212120
6	UFAngryBirdsC	195040
7	Adil	41360
8	Tori	129480

Quarter Final 1		
1	IHSEV	278820
2	Tori	250750
3	PlanA+	147130
4	s-birds Returns	104140

Quarter Final 2		
1	DataLab Birds	543590
2	**Angry-HEX**	352800
3	UFAngryBirdsC	101240
4	Angry Adil	0

Semi Final		
1	DataLab Birds	256080
2	**Angry-HEX**	247010
3	Tori	227700
4	IHSEV	199970

Grand Final		
1	DataLab Birds	529610
2	**Angry-HEX**	512220

Table B.6.: 2015 Competition Results - *AIBIRDS*.

2016[7]

Results after Qualification		
1	HeartyTian	576650
2	**Angry-HEX**	553720
3	Datalab Birds	550810
4	SEABirds	403190
5	S-Birds	372890
6	Naive Agent	293020
7	IHSEV	290770
8	BamBirds	60000

Quarter Final 1		
1	SEABirds	328570
2	BamBirds	280390
3	HeartyTian	252100
4	s-Birds	182970

Quarter Final 2		
1	IHSEV	470940
2	DataLab Birds	327490
3	Naive Agent	232880
4	**Angry-HEX**	231300

Semi Final		
1	IHSEV	562820
2	BamBirds	406200
3	DataLab Birds	371100
4	SEABirds	293410

Grand Final		
1	BamBirds	451250
2	IHSEV	288720

Table B.7.: 2016 Competition Results - *AIBIRDS*.

[5] https://aibirds.org/past-competitions/2014-competition/results.html
[6] https://aibirds.org/past-competitions/2015-competition/results.html
[7] http://aibirds.org/past-competitions/2016-competition/competition-results.html

2017[8]

Quarter Final 1

1	IHSEV	261,600
2	S-Birds	147,120
3	Condor	94,600

Quarter Final 2

1	**Angry-HEX**	242,980
2	Eagle's Wing	175,510
3	Vale Fina 007	106,930

Quarter Final 3

1	PlanA+	172,410
2	DataLab Birds	97,100
3	BamBirds	89,830
4	AngryBNU	0

Quarter Final Ranking

1	IHSEV	261,600
2	**Angry-HEX**	242,980
3	Eagle's Wing	175,510
4	PlanA+	172,410
5	S-Birds	147,120
6	Vale Fina 007	106,930
7	DataLab Birds	97,100
8	Condor	94,600
9	BamBirds	89,830
10	AngryBNU	0

Semi Final

1	IHSEV	415,890
2	Eagle's Wing	350,900
3	**Angry-HEX**	238,040
4	PlanA+	225,780

Grand Final

1	Eagle's Wing	355,700
2	IHSEV	275,110

Table B.8.: 2017 Competition Results - *AIBIRDS*.

[8] http://aibirds.org/angry-birds-ai-competition/competition-results.html (this *URL* will probably change in the next years, following a similar structure to the other links to Competition Results)

Bibliography

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan and S. B. Zdonik. 'Aurora: A Data Stream Management System'. In: *[298]*. 2003, p. 666. DOI: 10.1145/872757.872855 (cit. on p. 62).

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul and S. B. Zdonik. 'Aurora: a new model and architecture for data stream management'. In: *VLDB J.* 12.2 (2003), pp. 120–139. DOI: 10.1007/s00778-003-0095-z (cit. on p. 62).

[3] S. Abiteboul and R. Hull. 'Data Functions, Datalog and Negation (Extended Abstract)'. In: *Proceedings of ACM SIGMOD 1988*, pp. 143–153. DOI: 10.1145/50202.50218 (cit. on p. 14).

[4] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995 (cit. on pp. 9, 26).

[5] S. Abiteboul, V. Vianu, B. S. Fordham and Y. Yesha. 'Relational Transducers for Electronic Commerce'. In: *JCSS* 61.2 (2000), pp. 236–269. DOI: 10.1006/jcss.2000.1708 (cit. on p. 132).

[6] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, J. Gehrke, L. Haas, A. Halevy, J. Han, H. V. Jagadish, A. Labrinidis, S. Madden, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, C. Ross Kenneth and Shahabi, D. Suciu, S. Vaithyanathan and J. Widom. *Challenges and Opportunities with Big Data:* A white paper prepared for the Computing Community Consortium committee of the Computing Research Association, 2011. URL: http://cra.org/ccc/resources/ccc-led-whitepapers/ (cit. on p. 127).

[7] H. Alani, L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty and K. Janowicz, eds. *Proceedings of ISWC 2013*. Vol. 8219. LNCS. Springer, 21st–25th Oct. 2013. ISBN: 978-3-642-41337-7. DOI: 10.1007/978-3-642-41338-4 (cit. on pp. 268, 275).

[8] J. J. Alferes and J. A. Leite, eds. *Proceedings of JELIA 2004*. (Lisbon, Portugal). Vol. 3229. LNCS. Springer, 27th–30th Sept. 2004. ISBN: 3-540-23242-7 (cit. on pp. 277, 292).

[9] M. I. Ali, F. Gao and A. Mileo. 'CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets'. In: *[37]*. 2015, pp. 374–389. DOI: 10.1007/978-3-319-25010-6_25 (cit. on pp. 75, 80).

[10] L. V. Allis. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen, 1994 (cit. on p. 164).

[11] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri and J. Zangari. 'The ASP System DLV2'. In: *[46]*. 2017, pp. 215–221. DOI: 10.1007/978-3-319-61660-5_19 (cit. on p. 32).

[12] M. Alviano, F. Calimeri, W. Faber, N. Leone and S. Perri. 'Unfounded Sets and Well-Founded Semantics of Answer Set Programs with Aggregates'. In: *JAIR* 42 (2011), pp. 487–527. DOI: 10.1613/jair.3432 (cit. on p. 39).

[13] M. Alviano, C. Dodaro, W. Faber, N. Leone and F. Ricca. 'WASP: A Native ASP Solver Based on Constraint Learning'. In: *[115]*. 2013, pp. 54–66. DOI: 10.1007/978-3-642-40564-8_6 (cit. on p. 30).

[14] M. Alviano, C. Dodaro, N. Leone and F. Ricca. 'Advances in WASP'. In: *[136]*. 2015, pp. 40–54. DOI: 10.1007/978-3-319-23264-5_5 (cit. on p. 32).

[15] M. Alviano, C. Dodaro and F. Ricca. 'Anytime Computation of Cautious Consequences in Answer Set Programming'. In: *TPLP* 14.4-5 (2014), pp. 755–770. DOI: 10.1017/S1471068414000325 (cit. on p. 32).

[16] M. Alviano, W. Faber, G. Greco and N. Leone. 'Magic Sets for disjunctive Datalog programs'. In: *Artif. Intell.* 187 (2012), pp. 156–192. DOI: 10.1016/j.artint.2012.04.008 (cit. on p. 32).

[17] M. Alviano, W. Faber, N. Leone, S. Perri, G. Pfeifer and G. Terracina. 'The Disjunctive Datalog System DLV'. In: *[428]*. 2010, pp. 282–301. DOI: 10.1007/978-3-642-24206-9_17 (cit. on pp. 23, 27, 31).

[18] M. Alviano, N. Leone, M. Manna, G. Terracina and P. Veltri. 'Magic-Sets for Datalog with Existential Quantifiers'. In: *[66]*. 2012, pp. 31–43. DOI: 10.1007/978-3-642-32925-8_5 (cit. on p. 17).

[19] T. Ambroz, G. Charwat, A. Jusits, J. P. Wallner and S. Woltran. 'ARVis: Visualizing Relations between Answer Sets'. In: *[115]*. 2013, pp. 73–78. DOI: 10.1007/978-3-642-40564-8_8 (cit. on p. 251).

[20] D. Anicic. 'Event Processing and Stream Reasoning with ETALIS'. PhD thesis. Karlsruhe Institute of Technology, 2012. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025973 (cit. on p. 73).

[21] D. Anicic, P. Fodor, S. Rudolph and N. Stojanovic. 'EP-SPARQL: a unified language for event processing and stream reasoning'. In: *Proceedings of WWW 2011*, pp. 635–644. DOI: 10.1145/1963405.1963495 (cit. on p. 73).

[22] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic and R. Studer. 'A Rule-Based Language for Complex Event Processing and Reasoning'. In: *[302]*. 2010, pp. 42–57. DOI: 10.1007/978-3-642-15918-3_5 (cit. on p. 73).

[23] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic and R. Studer. 'Etalis: Rule-based reasoning in event processing'. In: *Reasoning in event-based distributed systems* 347 (2011), pp. 99–124 (cit. on p. 74).

[24] D. Anicic, S. Rudolph, P. Fodor and N. Stojanovic. 'Real-Time Complex Event Recognition and Reasoning-a Logic Programming Approach'. In: *Applied Artificial Intelligence* 26.1-2 (2012), pp. 6–57. DOI: 10.1080/08839514.2012.636616 (cit. on p. 74).

[25] D. Anicic, S. Rudolph, P. Fodor and N. Stojanovic. 'Stream reasoning and complex event processing in ETALIS'. In: *SWJ* 3.4 (2012), pp. 397–407. DOI: 10.3233/SW-2011-0053 (cit. on p. 73).

[26] G. Antoniou, S. Batsakis and I. Tachmazidis. 'Large-Scale Reasoning with (Semantic) Data'. In: *Proceedings of WIMS 2014*, 1:1–1:3. DOI: 10.1145/2611040.2611041 (cit. on p. 96).

[27] P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao and R. T. Snodgrass, eds. *Proceedings of VLDB 2001*. (Roma, Italy). Morgan Kaufmann, 11th–14th Sept. 2001. ISBN: 1-55860-804-4 (cit. on pp. 269, 297).

[28] K. R. Apt, H. A. Blair and A. Walker. 'Towards a Theory of Declarative Knowledge'. In: *Foundations of Deductive Databases and Logic Programming*, pp. 89–148 (cit. on p. 14).

[29] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava and J. Widom. 'STREAM: The Stanford Data Stream Management System'. In: *[243]*. 2016, pp. 317–336. DOI: 10.1007/978-3-540-28608-0_16 (cit. on pp. 55, 59).

[30] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma and J. Widom. 'STREAM: The Stanford Stream Data Manager'. In: *IEEE Data Eng. Bull.* 26.1 (2003), pp. 19–26. URL: http://sites.computer.org/debull/A03mar/paper.ps (cit. on pp. 50, 59).

[31] A. Arasu, S. Babu and J. Widom. 'CQL: A Language for Continuous Queries over Streams and Relations'. In: *Proceedings of DBPL 2003*, pp. 1–19. DOI: 10.1007/978-3-540-24607-7_1 (cit. on p. 59).

[32] A. Arasu, S. Babu and J. Widom. 'The CQL continuous query language: semantic foundations and query execution'. In: *VLDB J.* 15.2 (2006), pp. 121–142. DOI: 10.1007/s00778-004-0147-z (cit. on pp. 59, 79).

[33] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker and R. Tibbetts. 'Linear Road: A Stream Data Management Benchmark'. In: *Proceedings of VLDB 2004*, pp. 480–491. URL: http://www.vldb.org/conf/2004/RS12P1.PDF (cit. on p. 76).

[34] J. O. de Araujo and F. O. de França. 'UFAngryBirdsC Agent'. 2015. URL: https://aibirds.org/2015-teams/UFAngryBirdsC.pdf (cit. on p. 182).

[35] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen and G. Washburn. 'Design and Implementation of the LogicBlox System'. In: *Proceedings of ACM SIGMOD 2015*, pp. 1371–1382. DOI: 10.1145/2723372.2742796 (cit. on p. 19).

[36] W. G. Aref. 'Window-based Query Processing'. In: *[384]*. 2009, pp. 3533–3538. DOI: 10.1007/978-0-387-39940-9_468 (cit. on p. 81).

[37] M. Arenas, Ó. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. T. Groth, M. Dumontier, J. Heflin, K. Thirunarayan and S. Staab, eds. *Proceedings of ISWC 2015, Part II*. (Bethlehem, PA, USA). Vol. 9367. LNCS. Springer, 11th–15th Oct. 2015. ISBN: 978-3-319-25009-0. DOI: 10.1007/978-3-319-25010-6 (cit. on pp. 265, 291).

[38] M. Arikawa, S. Konomi and K. Ohnishi. 'Navitime: Supporting Pedestrian Navigation in the Real World'. In: *IEEE Pervasive Comput.* 6.3 (2007), pp. 21–29. DOI: 10.1109/MPRV.2007.61 (cit. on p. 51).

[39] B. Babcock, S. Babu, M. Datar, R. Motwani and J. Widom. 'Models and Issues in Data Stream Systems'. In: *[475]*. 2002, pp. 1–16. DOI: 10.1145/543613.543615 (cit. on pp. 56, 61, 65).

[40] S. Babu and J. Widom. 'Continuous Queries over Data Streams'. In: *SIGMOD Record* 30.3 (2001), pp. 109–120. DOI: 10.1145/603867.603884 (cit. on p. 50).

[41] J. Baget, M. Leclère, M. Mugnier, S. Rocher and C. Sipieter. 'Graal: A Toolkit for Query Answering with Existential Rules'. In: *Proceedings of RuleML 2015*, pp. 328–344. DOI: 10.1007/978-3-319-21542-6_21 (cit. on p. 20).

[42] J. Baget, M. Mugnier, S. Rudolph and M. Thomazo. 'Walking the Complexity Lines for Generalized Guarded Existential Rules'. In: *[576]*. 2011, pp. 712–717. DOI: 10.5591/978-1-57735-516-8/IJCAI11-126 (cit. on p. 17).

[43] Y. Bai, H. Thakkar, H. Wang and C. Zaniolo. 'Optimizing Timestamp Management in Data Stream Management Systems'. In: *Proceedings of ICDE 2007*, pp. 1334–1338. DOI: 10.1109/ICDE.2007.369005 (cit. on p. 59).

[44] Y. Bai, H. Thakkar, H. Wang and C. Zaniolo. 'Time-Stamp Management and Query Execution in Data Stream Management Systems'. In: *IEEE Internet Comput.* 12.6 (2008), pp. 13–21. DOI: 10.1109/MIC.2008.133 (cit. on p. 59).

[45] M. Baldoni, F. Chesani, P. Mello and M. Montali, eds. *Proceedings of PAI co-located with AI*IA 2013*. (Turin, Italy). Vol. 1107. CEUR-WS. CEUR-WS.org, 5th Dec. 2013. URL: http://ceur-ws.org/Vol-1107 (cit. on pp. xii, 183, 273, 279).

[46] M. Balduccini and T. Janhunen, eds. *Proceedings of LPNMR 2017*. (Espoo, Finland). Vol. 10377. LNCS. Springer, 3rd–6th July 2017. ISBN: 978-3-319-61659-9. DOI: 10.1007/978-3-319-61660-5 (cit. on pp. 265, 271).

[47] M. Balduccini, D. Magazzeni and M. Maratea. 'PDDL+ Planning via Constraint Answer Set Programming'. In: *CoRR* abs/1609.00030 (2016). URL: http://arxiv.org/abs/1609.00030 (cit. on p. 27).

[48] M. Balduccini and T. C. Son, eds. *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*. Vol. 6565. LNCS. Springer, 2011. ISBN: 978-3-642-20831-7. DOI: 10.1007/978-3-642-20832-4 (cit. on pp. 280, 282).

[49] M. Balduini, E. D. Valle, D. Dell'Aglio, M. Tsytsarau, T. Palpanas and C. Confalonieri. 'Social Listening of City Scale Events Using the Streaming Linked Data Framework'. In: *[7]*. 2013, pp. 1–16. DOI: 10.1007/978-3-642-41338-4_1 (cit. on p. 73).

[50] T. Balyo, M. J. H. Heule and M. Järvisalo. 'SAT Competition 2016: Recent Developments'. In: *Proceedings of AAAI 2017*, pp. 5061–5063. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14977 (cit. on p. 21).

[51] M. G. de la Banda and E. Pontelli, eds. *Proceedings of ICLP 2008*. (Udine, Italy). Vol. 5366. LNCS. Springer, 9th–13th Dec. 2008. ISBN: 978-3-540-89981-5. DOI: 10.1007/978-3-540-89982-2 (cit. on pp. 274, 280).

[52] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. New York, NY, USA: Cambridge University Press, 2003. ISBN: 0521818028 (cit. on pp. 21, 39, 102).

[53] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2010. ISBN: 978-0-521-14775-0. URL: http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/knowledge-representation-reasoning-and-declarative-problem-solving (cit. on p. 39).

[54] C. Baral and M. Gelfond. 'Logic Programming and Knowledge Representation'. In: *J. Log. Program.* 19/20 (1994), pp. 73–148. DOI: 10.1016/0743-1066(94)90025-6 (cit. on pp. 14, 213).

[55] C. Baral, M. Gelfond and A. Provetti. 'Representing Actions: Laws, Observations and Hypotheses'. In: *J. Log. Program.* 31.1-3 (1997), pp. 201–243. DOI: 10.1016/S0743-1066(96)00141-0 (cit. on p. 170).

[56] C. Baral, M. Gelfond and J. N. Rushton. 'Probabilistic reasoning with answer sets'. In: *TPLP* 9.1 (2009), pp. 57–144. DOI: 10.1017/S1471068408003645 (cit. on p. 183).

[57] C. Baral, G. D. Giacomo and T. Eiter, eds. *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press. ISBN: 978-1-57735-657-8. URL: http://www.aaai.org/Library/KR/kr14contents.php (cit. on pp. 282, 298, 300).

[58] V. Bárány, B. ten Cate, B. Kimelfeld, D. Olteanu and Z. Vagena. 'Declarative Statistical Modeling with Datalog'. In: *CoRR* abs/1412.2221 (2014). URL: http://arxiv.org/abs/1412.2221 (cit. on p. 15).

[59] D. F. Barbieri, D. Braga, S. Ceri and M. Grossniklaus. 'An execution environment for C-SPARQL queries'. In: *Proceedings of EDBT 2010*. ACM, pp. 441–452. DOI: 10.1145/1739041.1739095 (cit. on p. 72).

[60] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle and M. Grossniklaus. 'C-SPARQL: SPARQL for continuous querying'. In: *Proceedings of WWW 2009*, pp. 1061–1062. DOI: 10.1145/1526709.1526856 (cit. on pp. 66, 70, 72).

[61] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle and M. Grossniklaus. 'Incremental Reasoning on Streams and Rich Background Knowledge'. In: *Proceedings of ESWC 2010, Part I*, pp. 1–15. DOI: 10.1007/978-3-642-13486-9_1 (cit. on pp. 66, 70, 71, 73).

[62] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle and M. Grossniklaus. 'C-SPARQL: a Continuous Query Language for RDF Data Streams'. In: *Int. J. Semantic Computing* 4.1 (2010), pp. 3–25. DOI: 10.1142/S1793351X10000936 (cit. on p. 72).

[63] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle and M. Grossniklaus. 'Querying RDF streams with C-SPARQL'. In: *SIGMOD Record* 39.1 (2010), pp. 20–26. DOI: 10.1145/1860702.1860705 (cit. on pp. 72, 102).

[64] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, Y. Huang, V. Tresp, A. Rettinger and H. Wermser. 'Deductive and Inductive Stream Reasoning for Semantic Social Media Analytics'. In: *IEEE Intell. Syst.* 25.6 (2010), pp. 32–41. ISSN: 1541-1672. DOI: 10.1109/MIS.2010.142 (cit. on pp. 81, 82).

[65] D. Barbieri, D. Braga, S. Ceri, E. Della Valle and M. Grossniklaus. 'Stream reasoning: Where we got so far'. In: *Proceedings of NeFoRS 2010* (cit. on p. 48).

[66] P. Barceló and R. Pichler, eds. *Proceedings of International Workshop, Datalog 2.0.* (Vienna, Austria). Vol. 7494. LNCS. Springer, 11th–13th Sept. 2012. ISBN: 978-3-642-32924-1. DOI: 10.1007/978-3-642-32925-8 (cit. on pp. 135, 266, 300).

[67] J. Barrasa Rodrguez, Ó. Corcho and A. Gómez-Pérez. 'R2O, an Extensible and Semantically based Database-to-Ontology Mapping Language'. In: *Proceedings of SWDB 2004*, pp. 1069–1070 (cit. on p. 72).

[68] O. Bartheye and E. Jacopin. 'Connecting pddl-based off the shelf planners to an arcade game'. In: *Proceedings of AIG at ECAI 2008*. Vol. 8 (cit. on p. 169).

[69] S. Baselice, P. A. Bonatti and M. Gelfond. 'Towards an Integration of Answer Set and Constraint Solving'. In: *Proceedings of ICLP 2005*, pp. 52–66. DOI: 10.1007/11562931_7 (cit. on p. 23).

[70] S. Basol, O. Erdem, M. Fink and G. Ianni. 'HEX Programs with Action Atoms'. In: *Proceedings of ICLP 2010*, pp. 24–33. DOI: 10.4230/LIPIcs.ICLP.2010.24 (cit. on pp. 37, 39).

[71] T. Bass. 'Mythbusters: event stream processing versus complex event processing'. In: *Proceedings of DEBS 2007*, p. 1. DOI: 10.1145/1266894.1266896 (cit. on p. 65).

[72] A. Bassoli, J. Brewer, K. Martin, P. Dourish and S. D. Mainwaring. 'Underground Aesthetics: Rethinking Urban Computing'. In: *IEEE Pervasive Comput.* 6.3 (2007), pp. 39–45. DOI: 10.1109/MPRV.2007.68 (cit. on p. 51).

[73] J. Bates. 'The Role of Emotion in Believable Agents'. In: *Commun. ACM* 37.7 (1994), pp. 122–125. DOI: 10.1145/176789.176803 (cit. on p. 166).

[74] J. Bates, A. B. Loyall and W. S. Reilly. 'An Architecture for Action, Emotion, and Social Behavior'. In: *Proceedings of MAAMAW 1992*, pp. 55–68. DOI: 10.1007/3-540-58266-5_4 (cit. on p. 166).

[75] R. Baumgartner, S. Flesca and G. Gottlob. 'Visual Web Information Extraction with Lixto'. In: *[27]*. 2001, pp. 119–128. URL: http://www.vldb.org/conf/2001/P119.pdf (cit. on p. 134).

[76] H. R. Bazoobandi, H. Beck and J. Urbani. 'Expressive Stream Reasoning with Laser'. In: *Proceedings of ISWC 2017, Part I*, pp. 87–103. DOI: 10.1007/978-3-319-68288-4_6 (cit. on p. 96).

[77] H. Beck, B. Bierbaumer, M. Dao-Tran, T. Eiter, H. Hellwagner and K. Schekotihin. 'Rule-based Stream Reasoning for Intelligent Administration of Content-Centric Networks'. In: *Proceedings of JELIA 2016*, pp. 522–528. DOI: 10.1007/978-3-319-48758-8_34 (cit. on p. 96).

[78] H. Beck, B. Bierbaumer, M. Dao-Tran, T. Eiter, H. Hellwagner and K. Schekotihin. 'Stream reasoning-based control of caching strategies in CCN routers'. In: *Proceedings of ICC 2017*, pp. 1–6. DOI: 10.1109/ICC.2017.7996762 (cit. on p. 96).

[79] H. Beck, M. Dao-Tran and T. Eiter. 'Answer Update for Rule-Based Stream Reasoning'. In: *Proceedings of IJCAI 2015*, pp. 2741–2747. URL: http://ijcai.org/Abstract/15/388 (cit. on p. 95).

[80] H. Beck, M. Dao-Tran and T. Eiter. 'Equivalent Stream Reasoning Programs'. In: *[322]*. 2016, pp. 929–935. URL: http://www.ijcai.org/Abstract/16/136 (cit. on p. 96).

[81] H. Beck, M. Dao-Tran, T. Eiter and M. Fink. 'Towards a Logic-Based Framework for Analyzing Stream Reasoning'. In: *Proceedings of OrdRing 2014*, pp. 11–22. URL: http://ceur-ws.org/Vol-1303/paper_3.pdf (cit. on pp. 93, 94).

[82] H. Beck, M. Dao-Tran, T. Eiter and M. Fink. 'LARS: A Logic-Based Framework for Analyzing Reasoning over Streams'. In: *[93]*. 2015, pp. 1431–1438. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9657 (cit. on pp. 93–95, 102).

[83] H. Beck, M. Dao-Tran, T. Eiter and M. Fink. 'Towards Ideal Semantics for Analyzing Stream Reasoning'. In: *CoRR* abs/1505.05365 (2015). URL: http://arxiv.org/abs/1505.05365 (cit. on pp. 93, 94).

[84] H. Beck, T. Eiter and C. F. Beckmann. 'Ticker: A system for incremental ASP-based stream reasoning'. In: *TPLP* 17.5-6 (2017), pp. 744–763. DOI: 10.1017/S1471068417000370 (cit. on p. 96).

[85] C. Beeri and R. Ramakrishnan. 'On the Power of Magic'. In: *J. Log. Program.* 10.3&4 (1991), pp. 255–299. DOI: 10.1016/0743-1066(91)90038-Q (cit. on p. 15).

[86] L. Bellomarini, G. Gottlob, A. Pieris and E. Sallinger. 'Swift Logic for Big Data and Knowledge Graphs'. In: *[524]*. 2017, pp. 2–10. DOI: 10.24963/ijcai.2017/1 (cit. on p. 140).

[87] M. Ben-Ari. *Mathematical Logic for Computer Science*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 0-13-564139-X (cit. on p. 8).

[88] R. Ben-Eliyahu and R. Dechter. 'Propositional Semantics for Disjunctive Logic Programs'. In: *AMAI* 12.1-2 (1994), pp. 53–87. DOI: 10.1007/BF01530761 (cit. on p. 141).

[89] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro and E. Tsamoura. 'Benchmarking the Chase'. In: *Proceedings of PODS 2017*, pp. 37–52. DOI: 10.1145/3034786.3034796 (cit. on p. 146).

[90] B. Bishop and F. Fischer. 'IRIS - Integrated Rule Inference System'. In: *Proceedings of ARea 2008* (cit. on p. 19).

[91] C. Bizer, T. Heath and T. Berners-Lee. 'Linked Data - The Story So Far'. In: *IJSWIS* 5.3 (2009), pp. 1–22. DOI: 10.4018/jswis.2009081901 (cit. on p. 72).

[92] A. Bolles, M. Grawunder and J. Jacobi. 'Streaming SPARQL - Extending SPARQL to Process Data Streams'. In: *Proceedings of ESWC 2008*, pp. 448–462. DOI: 10.1007/978-3-540-68234-9_34 (cit. on p. 72).

[93] B. Bonet and S. Koenig, eds. *Proceedings of AAAI 2015*. (Austin, Texas, USA). AAAI Press, 25th–30th Jan. 2015. ISBN: 978-1-57735-698-1. URL: http://www.aaai.org/Library/AAAI/aaai15contents.php (cit. on pp. 270, 294).

[94] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer and R. Ramakrishnan. 'Declarative Systems for Large-Scale Machine Learning'. In: *IEEE Data Eng. Bull.* 35.2 (2012), pp. 24–32. URL: http://sites.computer.org/debull/A12june/declare.pdf (cit. on p. 126).

[95] T. Borovika, R. petlík and K. Ryme. 'DataLab Birds Angry Birds AI'. 2014. URL: https://aibirds.org/2014-papers/datalab-birds.pdf (cit. on pp. 180, 181).

[96] I. Botan, R. Derakhshan, N. Dindar, L. M. Haas, R. J. Miller and N. Tatbul. 'SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems'. In: *PVLDB* 3.1 (2010), pp. 232–243. URL: http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R20.pdf (cit. on p. 79).

[97] D. M. Bourg and G. Seemann. *AI for Game Developers*. O'Reilly Media, Inc., 2004. ISBN: 0596005555 (cit. on p. 164).

[98] M. Brain, O. Cliffe and M. De Vos. 'A pragmatic programmer's guide to answer set programming'. In: *Proceedings of SEA 2009*, pp. 49–63 (cit. on p. 28).

[99] I. Bratko. *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley, 2012. ISBN: 978-0-3214-1746-6 (cit. on p. 7).

[100] G. Brewka. 'Towards Reactive Multi-Context Systems'. In: *[115]*. 2013, pp. 1–10. DOI: 10.1007/978-3-642-40564-8_1 (cit. on p. 96).

[101] G. Brewka, T. Eiter and S. A. McIlraith, eds. *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press. ISBN: 978-1-57735-560-1 (cit. on pp. 278, 280, 286, 287).

[102] G. Brewka, T. Eiter and M. Truszczynski. 'Answer set programming at a glance'. In: *Commun. ACM* 54.12 (2011), pp. 92–103. DOI: 10.1145/2043174.2043195 (cit. on p. 39).

[103] G. Brewka, T. Eiter and M. Truszczynski. 'Answer Set Programming: An Introduction to the Special Issue'. In: *AI Magazine* 37.3 (2016), pp. 5–6. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2669 (cit. on p. 39).

[104] G. Brewka, S. Ellmauthaler and J. Pührer. 'Multi-Context Systems for Reactive Reasoning in Dynamic Environments'. In: *[514]*. 2014, pp. 159–164. DOI: 10.3233/978-1-61499-419-0-159 (cit. on p. 96).

[105] A. Brik. 'Extensions of Answer Set Programming'. PhD thesis. University of California, San Diego, USA, 2012. URL: http://www.escholarship.org/uc/item/9v1981f6 (cit. on p. 23).

[106] A. Brik and J. B. Remmel. 'Action Language Hybrid AL'. In: *[46]*. 2017, pp. 322–335. DOI: 10.1007/978-3-319-61660-5_29 (cit. on p. 23).

[107] E. Brown. 'Watson: The Jeopardy! Challenge and beyond'. In: *Proceedings of ICCI*CC 2013*, p. 2. DOI: 10.1109/ICCI-CC.2013.6622216 (cit. on p. 165).

[108] E. W. Brown. 'Watson: the Jeopardy! challenge and beyond'. In: *Proceedings of SIGIR 2012*, p. 1020. DOI: 10.1145/2348283.2348446 (cit. on p. 165).

[109] N. Bruno and S. Chaudhuri. 'Exploiting statistics on query expressions for optimization'. In: *Proceedings of ACM SIGMOD 2002*, pp. 263–274. DOI: 10.1145/564691.564722 (cit. on p. 149).

[110] F. Buccafurri, F. Furfaro and D. Saccà. 'Estimating Range Queries Using Aggregate Data with Integrity Constraints: A Probabilistic Approach'. In: *Proceedings of ICDT 2001*, pp. 390–404. DOI: 10.1007/3-540-44503-X_25 (cit. on p. 149).

[111] F. Buccafurri, N. Leone and P. Rullo. 'Strong and Weak Constraints in Disjunctive Datalog'. In: *[183]*. 1997, pp. 2–17. DOI: 10.1007/3-540-63255-7_2 (cit. on pp. 38, 235).

[112] M. Buckland and M. Collins. *AI Techniques for Game Programming*. Premier Press, 2002. ISBN: 9781931841085 (cit. on p. 164).

[113] M. Buro. 'Call for AI research in RTS games'. In: *Proceedings of CGAI 2004*, pp. 139–142 (cit. on p. 209).

[114] P. Busoniu, J. Oetsch, J. Pührer, P. Skocovsky and H. Tompits. 'SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support'. In: *TPLP* 13.4-5 (2013), pp. 657–673 (cit. on p. 250).

[115] P. Cabalar and T. C. Son, eds. *Proceedings of LPNMR 2013*. (Corunna, Spain). Vol. 8148. LNCS. Springer, 15th–19th Sept. 2013. ISBN: 978-3-642-40563-1. DOI: 10.1007/978-3-642-40564-8 (cit. on pp. 266, 271, 278).

[116] J. Calbimonte, Ó. Corcho and A. J. G. Gray. 'Enabling Ontology-Based Access to Streaming Data Sources'. In: *Proceedings of ISWC 2010*, pp. 96–111. DOI: 10.1007/978-3-642-17746-0_7 (cit. on pp. 72, 102).

[117] J. Calbimonte, H. Jeung, Ó. Corcho and K. Aberer. 'Enabling Query Technologies for the Semantic Sensor Web'. In: *IJSWIS* 8.1 (2012), pp. 43–63. DOI: 10.4018/jswis.2012010103 (cit. on p. 72).

[118] A. Calì, G. Gottlob and M. Kifer. 'Taming the Infinite Chase: Query Answering under Expressive Relational Constraints'. In: *JAIR* 48 (2013), pp. 115–174. DOI: 10.1613/jair.3873 (cit. on pp. 15, 17).

[119] A. Calì, G. Gottlob and T. Lukasiewicz. 'A general Datalog-based framework for tractable query answering over ontologies'. In: *J. Web Sem.* 14 (2012), pp. 57–83. DOI: 10.1016/j.websem.2012.03.001 (cit. on pp. 17, 133).

[120] A. Calì, G. Gottlob, T. Lukasiewicz, B. Marnette and A. Pieris. 'Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications'. In: *Proceedings of LICS 2010*, pp. 228–242. DOI: 10.1109/LICS.2010.27 (cit. on p. 15).

[121] A. Calì, G. Gottlob and A. Pieris. 'Advanced Processing for Ontological Queries'. In: *PVLDB* 3.1 (2010), pp. 554–565. URL: http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R49.pdf (cit. on p. 17).

[122] A. Calì, G. Gottlob and A. Pieris. 'Query Answering under Non-guarded Rules in Datalog+/-'. In: *[302]*. 2010, pp. 1–17. DOI: 10.1007/978-3-642-15918-3_1 (cit. on p. 17).

[123] A. Calì, G. Gottlob and A. Pieris. 'Towards more expressive ontology languages: The query answering problem'. In: *Artif. Intell.* 193 (2012), pp. 87–128. DOI: 10.1016/j.artint.2012.08.002 (cit. on p. 17).

[124] A. Calì, D. Lembo and R. Rosati. 'On the decidability and complexity of query answering over inconsistent and incomplete databases'. In: *Proceedings of PODS 2003*, pp. 260–271. DOI: 10.1145/773153.773179 (cit. on p. 18).

[125] F. Calimeri, S. Cozza and G. Ianni. 'External sources of knowledge and value invention in logic programming'. In: *AMAI* 50.3-4 (2007), pp. 333–361. DOI: 10.1007/s10472-007-9076-z (cit. on p. 33).

[126] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca and T. Schaub. *ASP-Core-2: Input language format*. Technical Report. ASP Standardization Working Group, 2013. URL: https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf (cit. on pp. 23, 235).

[127] F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci and A. Wimmer. 'Angry-HEX: An Artificial Player for Angry Birds Based on Declarative Knowledge Bases'. In: *TCIAIG* 8.2 (2016), pp. 128–139. DOI: 10.1109/TCIAIG.2015.2509600 (cit. on pp. xii, 21, 183).

[128] F. Calimeri, M. Fink, S. Germano, G. Ianni, C. Redl and A. Wimmer. 'AngryHEX: an Artificial Player for Angry Birds Based on Declarative Knowledge Bases'. In: *[45]*. 2013, pp. 29–35. URL: http://ceur-ws.org/Vol-1107/paper10.pdf (cit. on pp. xii, 183).

[129] F. Calimeri, D. Fuscà, S. Germano, S. Perri and J. Zangari. 'Boosting the Development of ASP-Based Applications in Mobile and General Scenarios'. In: *Proceedings of AI*IA 2016*, pp. 223–236. DOI: 10.1007/978-3-319-49130-1_17 (cit. on pp. xii, 205, 216).

[130] F. Calimeri, D. Fuscà, S. Germano, S. Perri and J. Zangari. 'Embedding ASP in mobile systems: discussion and preliminary implementations'. In: *Proceedings of ASPOCP 2015, workshop of ICLP* (cit. on pp. xii, 216, 238).

[131] F. Calimeri, D. Fuscà, S. Germano, S. Perri and J. Zangari. EMBASP. 2015–2017. URL: https://www.mat.unical.it/calimeri/projects/embasp (visited on 25th Sept. 2017) (cit. on pp. 205, 238, 239).

[132] F. Calimeri, D. Fuscà, S. Germano, S. Perri and J. Zangari. 'A framework for easing the development of applications embedding answer set programming'. In: *J. Exp. Theor. Artif. Intell.* (2017). Submitted (cit. on pp. xiii, 21, 28, 40, 216).

[133] F. Calimeri, D. Fuscà, S. Perri and J. Zangari. 'I-DLV: The new intelligent grounder of DLV'. In: *Intelligenza Artificiale* 11.1 (2017), pp. 5–20. DOI: 10.3233/IA-170104 (cit. on pp. 20, 32).

[134] F. Calimeri, S. Germano, E. Palermiti, K. Reale and F. Ricca. 'Environments for Developing ASP programs'. In: *KI* (2017). Submitted (cit. on p. xiii).

[135] F. Calimeri, G. Ianni and F. Ricca. 'The third open answer set programming competition'. In: *TPLP* 14.1 (2014), pp. 117–135. DOI: 10.1017/S1471068412000105 (cit. on pp. 30, 35).

[136] F. Calimeri, G. Ianni and M. Truszczynski, eds. *Proceedings of LPNMR 2015*. (Lexington, KY, USA). Vol. 9345. LNCS. Springer, 27th–30th Sept. 2015. ISBN: 978-3-319-23263-8. DOI: 10.1007/978-3-319-23264-5 (cit. on pp. 266, 280, 289).

[137] F. Calimeri and F. Ricca. *On the Application of the Answer Set Programming System DLV in Industry: a Report from the Field*. ALP Newsletter. Mar. 2012 (cit. on p. 27).

[138] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi and D. F. Savo. 'The MASTRO system for ontology-based data access'. In: *SWJ* 2.1 (2011), pp. 43–53. DOI: 10.3233/SW-2011-0029 (cit. on p. 139).

[139] C. Castelfranchi. 'Guarantees for Autonomy in Cognitive Agent Architecture'. In: *Proceedings of ATAL 1994*, pp. 56–70. DOI: 10.1007/3-540-58855-8_3 (cit. on p. 165).

[140] S. Ceri, G. Gottlob and L. Tanca. 'What you Always Wanted to Know About Datalog (And Never Dared to Ask)'. In: *TKDE* 1.1 (1989), pp. 146–166. DOI: 10.1109/69.43410 (cit. on p. 9).

[141] S. Ceri, G. Gottlob and L. Tanca. *Logic Programming and Databases*. Surveys in computer science. Springer, 1990. ISBN: 3-540-51728-6. URL: http://www.worldcat.org/oclc/20595273 (cit. on p. 9).

[142] U. Çetintemel, D. J. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing and S. Zdonik. 'The Aurora and Borealis Stream Processing Engines'. In: *[243]*. 2016, pp. 337–359. DOI: 10.1007/978-3-540-28608-0_17 (cit. on p. 62).

[143] A. J. Champandard. *AI Game Development*. New Riders Games, 2003. ISBN: 1592730043 (cit. on p. 164).

[144] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss and M. A. Shah. 'TelegraphCQ: Continuous Dataflow Processing for an Uncertain World'. In: *Proceedings of CIDR 2003*. URL: http://www-db.cs.wisc.edu/cidr/cidr2003/program/p24.pdf (cit. on p. 62).

[145] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss and M. A. Shah. 'TelegraphCQ: Continuous Dataflow Processing'. In: *[298]*. 2003, p. 668. DOI: 10.1145/872757.872857 (cit. on p. 62).

[146] S. Chaudhuri, U. Dayal and V. R. Narasayya. 'An overview of business intelligence technology'. In: *Commun. ACM* 54.8 (2011), pp. 88–98. DOI: 10.1145/1978542.1978562 (cit. on p. 127).

[147] M. Chen, S. Mao and Y. Liu. 'Big Data: A Survey'. In: *MONET* 19.2 (2014), pp. 171–209. DOI: 10.1007/s11036-013-0489-0 (cit. on pp. 124, 126, 127).

[148] S. Christodoulakis. 'Implications of Certain Assumptions in Database Performance Evaluation'. In: *TODS* 9.2 (1984), pp. 163–186. DOI: 10.1145/329.318578 (cit. on p. 161).

[149] K. L. Clark. 'Negation as Failure'. In: *Proceedings of Symposium on Logic and Data Bases 1977*, pp. 293–322 (cit. on p. 141).

[150] O. Cliffe, M. D. Vos, M. Brain and J. A. Padget. 'ASPVIZ: Declarative Visualisation and Animation Using Answer Set Programming'. In: *[51]*. 2008, pp. 724–728. DOI: 10.1007/978-3-540-89982-2_65 (cit. on p. 251).

[151] W. F. Clocksin and C. S. Mellish. *Programming in Prolog (4. ed.)* Springer, 1994. ISBN: 978-3-540-58350-9 (cit. on p. 7).

[152] M. Collautti, Y. Malitsky, D. Mehta and B. O'Sullivan. 'SNNAP: Solver-Based Nearest Neighbor for Algorithm Portfolios'. In: *Proceedings of ECML PKDD 2013, Part III*, pp. 435–450. DOI: 10.1007/978-3-642-40994-3_28 (cit. on p. 146).

[153] M. Colledanchise and P. Ögren. 'Behavior Trees in Robotics and AI: An Introduction'. In: *CoRR* abs/1709.00084 (2017). arXiv: 1709.00084 (cit. on p. 210).

[154] M. Collins. *Advance AI Techniques for Game Programming (Game Programming)*. Pearson Professional Education, 2001. ISBN: 0761536248 (cit. on p. 164).

[155] M. Collins. *Advanced Ai Game Development (Wordware Game Developer's Library)*. Plano, TX, USA: Wordware Publishing Inc., 2003. ISBN: 1556229623 (cit. on p. 164).

[156] A. Colmerauer. 'An Introduction to Prolog III'. In: *Commun. ACM* 33.7 (1990), pp. 69–90. DOI: 10.1145/79204.79210 (cit. on p. 7).

[157] A. Colmerauer and P. Roussel. 'The Birth of Prolog'. In: *Proceedings of HOPL-II 1993*, pp. 37–52. DOI: 10.1145/154766.155362 (cit. on p. 6).

[158] C. Consortium. *Final Report. CityPulse - Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications*. Report - Project Delivery. Version V1.0-Final. AA, AI, ERIC, NUIG, SAGO, SIE, UASO, WSU, UniS, Nov. 2016. URL: https://cordis.europa.eu/docs/projects/cnect/5/609035/080/deliverables/001-609035CityPulseD14FinalReportAres20172763775.pdf (visited on 25th Sept. 2017) (cit. on pp. 97–100).

[159] A. O. da Costa, I. L. Rocha and S. Elena. 'DualHEX: an extension of the AngryHEX Artificial Player for AngryBirds'. 2016 (cit. on p. 204).

[160] G. Cugola and A. Margara. 'Processing flows of information: From data stream to complex event processing'. In: *CSUR* 44.3 (2012), 15:1–15:62. DOI: 10.1145/2187671.2187677 (cit. on pp. 65, 66, 68, 75).

[161] M. Dao-Tran, H. Beck and T. Eiter. 'Contrasting RDF Stream Processing Semantics'. In: *Proceedings of JIST 2015*, pp. 289–298. DOI: 10.1007/978-3-319-31676-5_21 (cit. on p. 95).

[162] M. Dao-Tran, H. Beck and T. Eiter. 'Towards Comparing RDF Stream Processing Semantics'. In: *[442]*. 2015, pp. 15–27. URL: http://ceur-ws.org/Vol-1447/paper2.pdf (cit. on p. 95).

[163] M. Dao-Tran and D. L. Phuoc. 'Towards Enriching CQELS with Complex Event Processing and Path Navigation'. In: *[442]*. 2015, pp. 2–14. URL: http://ceur-ws.org/Vol-1447/paper1.pdf (cit. on p. 75).

[164] S. Dasgupta, S. Kothari, S. Talluri and S. Motiani. 's-birds: A Heuristic Engine based Learner Bot for the Angry Bird Problem'. 2013. URL: http://aibirds.org/2013-Papers/Team-Descriptions/sbirds.pdf (cit. on p. 182).

[165] S. Dasgupta, V. Modi, S. Vaghela, H. Kanakia and D. Shah. 's-birds Avengers: A Dynamic Heuristic Engine based Bot for the Angry Bird Problem'. 2014. URL: https://aibirds.org/2014-papers/s-birds-avengers.pdf (cit. on p. 182).

[166] S. Dasgupta, S. Vaghela, V. Modi and H. Kanakia. 's-Birds Avengers: A Dynamic Heuristic Engine-Based Agent for the Angry Birds Problem'. In: *TCIAIG* 8.2 (2016), pp. 140–151. DOI: 10.1109/TCIAIG.2016.2553244 (cit. on p. 182).

[167] I. Dasseville and G. Janssens. 'A web-based IDE for IDP'. In: *CoRR* abs/1511.00920 (2015) (cit. on pp. 242, 250).

[168] S. K. Debray and N. Lin. 'Cost Analysis of Logic Programs'. In: *TOPLAS* 15.5 (1993), pp. 826–875. DOI: 10.1145/161468.161472 (cit. on p. 156).

[169] S. K. Debray, N. Lin and M. V. Hermenegildo. 'Task Granularity Analysis in Logic Programs'. In: *Proceedings of PLDI 1990*, pp. 174–188. DOI: 10.1145/93542.93564 (cit. on p. 156).

[170] S. K. Debray, P. López-Garca, M. V. Hermenegildo and N. Lin. 'Lower Bound Cost Estimation for Logic Programs'. In: *Proceedings of 1997 International Symposium on Logic Programming*, pp. 291–305 (cit. on p. 156).

[171] J. P. Delgrande and W. Faber, eds. *Proceedings of LPNMR 2011*. (Vancouver, Canada). Vol. 6645. LNCS. Springer, 16th–19th May 2011. ISBN: 978-3-642-20894-2 (cit. on pp. 278, 280, 281).

[172] D. Dell'Aglio, M. Balduini and E. Della Valle. 'On the need to include functional testing in rdf stream engine benchmarks'. In: *BeRSys* (2013) (cit. on p. 79).

[173] D. Dell'Aglio, J. Calbimonte, M. Balduini, Ó. Corcho and E. D. Valle. 'On Correctness in RDF Stream Processor Benchmarking'. In: *[7]*. 2013, pp. 326–342. DOI: 10.1007/978-3-642-41338-4_21 (cit. on pp. 75, 78).

[174] D. Dell'Aglio, M. Dao-Tran, J. Calbimonte, D. L. Phuoc and E. D. Valle. 'A Query Model to Capture Event Pattern Matching in RDF Stream Processing Query Languages'. In: *Proceedings of EKAW 2016*, pp. 145–162. DOI: 10.1007/978-3-319-49004-5_10 (cit. on pp. 66, 96).

[175] D. Dell'Aglio, E. Della Valle, F. van Harmelen and A. Bernstein. 'Stream reasoning: A survey and outlook'. In: *Data Sci.* Preprint (2017), pp. 1–24 (cit. on pp. 51, 53, 54, 81).

[176] D. Dell'Aglio, E. D. Valle, J. Calbimonte and Ó. Corcho. 'RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems'. In: *IJSWIS* 10.4 (2014), pp. 17–44. DOI: 10.4018/ijswis.2014100102 (cit. on p. 96).

[177] E. Della Valle. 'Tutorial on Stream Reasoning for Linked Data at ISWC 2013'. Sydney, Australia, 21st Oct. 2013. URL: http://streamreasoning.org/sr4ld2013 (cit. on pp. 48, 66, 68).

[178] E. Della Valle, S. Ceri, D. Braga, I. Celino, D. Fensel, F. van Harmelen and G. Unel. 'Research Chapters in the area of Stream Reasoning'. In: *Proceedings of SR 2009*. (Heraklion, Crete, Greece), pp. 1–9 (cit. on p. 48).

[179] A. Deutsch, A. Nash and J. B. Remmel. 'The chase revisited'. In: *Proceedings of PODS 2008*, pp. 149–158. DOI: 10.1145/1376916.1376938 (cit. on p. 17).

[180] Z. Dhouioui, W. Labbadi and J. Akaichi. 'A New Algorithm for Accurate Histogram Construction'. In: *Proceedings of IMMM 2012* (cit. on p. 149).

[181] N. Dindar, N. Tatbul, R. J. Miller, L. M. Haas and I. Botan. 'Modeling the execution semantics of stream processing engines with SECRET'. In: *VLDB J.* 22.4 (2013), pp. 421–446. DOI: 10.1007/s00778-012-0297-3 (cit. on p. 79).

[182] L. Ding, T. Finin, Y. Peng, P. P. Da Silva and D. L. McGuinness. 'Tracking rdf graph provenance using rdf molecules'. In: *Proceedings of the ISWC 2005 (Poster)*, p. 42 (cit. on p. 71).

[183] J. Dix, U. Furbach and A. Nerode, eds. *Proceedings of LPNMR 1997*. (Dagstuhl Castle, Germany). Vol. 1265. LNCS. Springer, 28th–31st July 1997. ISBN: 3-540-63255-7. DOI: 10.1007/3-540-63255-7 (cit. on pp. 271, 277).

[184] T. M. Do, S. W. Loke and F. Liu. 'Answer Set Programming for Stream Reasoning'. In: *Proceedings of Canadian AI 2011*, pp. 104–109. DOI: 10.1007/978-3-642-21043-3_13 (cit. on pp. 82, 102).

[185] T. M. Do, S. W. Loke and F. Liu. 'HealthyLife: An Activity Recognition System with Smartphone Using Logic-Based Stream Reasoning'. In: *Proceedings of MobiQuitous 2012*, pp. 188–199. DOI: 10.1007/978-3-642-40238-8_16 (cit. on p. 236).

[186] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca and K. Schekotihin. 'Combining Answer Set Programming and domain heuristics for solving hard industrial problems (Application Paper)'. In: *TPLP* 16.5-6 (2016), pp. 653–669. DOI: 10.1017/S1471068416000284 (cit. on p. 33).

[187] J. Doyle. 'A Truth Maintenance System'. In: *Artif. Intell.* 12.3 (1979), pp. 231–272. DOI: 10.1016/0004-3702(79)90008-0 (cit. on p. 96).

[188] R. Du, Z. Gao and Z. Xu. *Deliberately Planning and Acting for Angry Birds with Refinement Methods*. 2015. URL: http://duruofei.com/Research/angrybird (visited on 25th Sept. 2017) (cit. on p. 182).

[189] S. Edelkamp and J. Hoffmann. *PDDL2. 2: The language for the classical part of the 4th international planning competition*. Tech. rep. Technical Report 195, University of Freiburg, 2004 (cit. on p. 40).

[190] T. Eiter, W. Faber, N. Leone and G. Pfeifer. 'Declarative problem-solving using the DLV system'. In: *Logic-based artificial intelligence* 597 (2000), pp. 79–103 (cit. on p. 39).

[191] T. Eiter, M. Fink, T. Krennwallner and C. Redl. 'Conflict-driven ASP solving with external sources'. In: *TPLP* 12.4-5 (2012), pp. 659–679. DOI: 10.1017/S1471068412000233 (cit. on p. 35).

[192] T. Eiter, M. Fink, G. Sabbatini and H. Tompits. 'A Framework for Declarative Update Specifications in Logic Programs'. In: *Proceedings of IJCAI 2001*, pp. 649–654 (cit. on p. 96).

[193] T. Eiter, S. Germano, G. Ianni, T. Kaminski, C. Redl, P. Schüller and A. Weinzierl. 'The DLVHEX System'. In: *KI* (2017). Submitted (cit. on p. xiii).

[194] T. Eiter and G. Gottlob. 'Complexity Aspects of Various Semantics for Disjunctive Databases'. In: *Proceedings of PODS 1993*, pp. 158–167. DOI: 10.1145/153850.153864 (cit. on p. 14).

[195] T. Eiter, G. Gottlob and H. Mannila. 'Adding Disjunction to Datalog'. In: *Proceedings of PODS 1994*, pp. 267–278. DOI: 10.1145/182591.182639 (cit. on p. 9).

[196] T. Eiter, G. Gottlob and H. Mannila. 'Disjunctive Datalog'. In: *TODS* 22.3 (1997), pp. 364–418. ISSN: 0362-5915. DOI: 10.1145/261124.261126 (cit. on pp. 9, 39).

[197] T. Eiter, G. Gottlob and H. Veith. 'Modular Logic Programming and Generalized Quantifiers'. In: *[183]*. 1997, pp. 290–309. DOI: 10.1007/3-540-63255-7_22 (cit. on p. 161).

[198] T. Eiter, G. Ianni and T. Krennwallner. 'Answer Set Programming: A Primer'. In: *Tutorial Lectures of RW 2009*, pp. 40–110. DOI: 10.1007/978-3-642-03754-2_2 (cit. on p. 39).

[199] T. Eiter, G. Ianni, R. Schindlauer and H. Tompits. 'A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming'. In: *Proceedings of IJCAI 2005*, pp. 90–96. URL: http://ijcai.org/Proceedings/05/Papers/1353.pdf (cit. on pp. 32, 35, 37, 38, 183).

[200] T. Eiter, G. Ianni, R. Schindlauer and H. Tompits. 'dlvhex: A Prover for Semantic-Web Reasoning under the Answer-Set Semantics'. In: *Proceedings of WI 2006*, pp. 1073–1074. DOI: 10.1109/WI.2006.64 (cit. on p. 35).

[201] T. Eiter, G. Ianni, R. Schindlauer and H. Tompits. 'Towards Efficient Evaluation of HEX Programs'. In: *Proceedings of the Workshop on Nonmonotonic Reasoning*. 2006, pp. 40–46 (cit. on p. 35).

[202] T. Eiter, T. Kaminski, C. Redl, P. Schüller and A. Weinzierl. 'Answer Set Programming with External Source Access'. In: *[304]*. 2017, pp. 204–275. DOI: 10.1007/978-3-319-61033-7_7 (cit. on p. 37).

[203] T. Eiter, T. Kaminski and A. Weinzierl. 'Lazy-Grounding for Answer Set Programs with External Source Access'. In: *[524]*. 2017, pp. 1015–1022. DOI: 10.24963/ijcai.2017/141 (cit. on p. 39).

[204] T. Eiter, C. Redl and P. Schüller. 'Problem Solving Using the HEX Family'. In: *Proceedings of Computational Models of Rationality. Essays dedicated to Gabriele Kern-Isberner on the occasion of her 60th birthday*, pp. 150–174 (cit. on p. 37).

[205] M. P. Eladhari, A. Sullivan, G. Smith and J. McCoy. *AI-Based game design: Enabling new playable experiences*. Tech. rep. Technical Report, UCSC-SOE-11, 2011 (cit. on p. 171).

[206] M. H. van Emden and R. A. Kowalski. 'The Semantics of Predicate Logic as a Programming Language'. In: *J. ACM* 23.4 (1976), pp. 733–742. DOI: 10.1145/321978.321991 (cit. on pp. 5, 6).

[207] E. Erdem, M. Gelfond and N. Leone. 'Applications of Answer Set Programming'. In: *AI Magazine* 37.3 (2016), pp. 53–68. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2678 (cit. on p. 27).

[208] E. Erdem, J. Lee, Y. Lierler and D. Pearce, eds. *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Vol. 7265. LNCS. Springer, 2012. ISBN: 978-3-642-30742-3. DOI: 10.1007/978-3-642-30743-0 (cit. on pp. 277, 298).

[209] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Company, 2010. ISBN: 978-1-935182-21-4. URL: http://www.manning.com/etzion/ (cit. on p. 66).

[210] P. T. Eugster, P. Felber, R. Guerraoui and A. Kermarrec. 'The many faces of publish/subscribe'. In: *ACM Comput. Surv.* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078 (cit. on p. 65).

[211] W. Faber, N. Leone and S. Perri. 'The Intelligent Grounder of DLV'. In: *[208]*. 2012, pp. 247–264. DOI: 10.1007/978-3-642-30743-0_17 (cit. on pp. 31, 156).

[212] W. Faber, N. Leone and G. Pfeifer. 'Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity'. In: *[8]*. 2004, pp. 200–212. DOI: 10.1007/978-3-540-30227-8_19 (cit. on p. 36).

[213] R. Fagin, P. G. Kolaitis, R. J. Miller and L. Popa. 'Data exchange: semantics and query answering'. In: *Theor. Comput. Sci.* 336.1 (2005), pp. 89–124. DOI: 10.1016/j.tcs.2004.10.033 (cit. on pp. 17, 18).

[214] O. Febbraro, N. Leone, G. Grasso and F. Ricca. 'JASP: A Framework for Integrating Answer Set Programming with Java'. In: *[101]*. 2012. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4520 (cit. on pp. 32, 235).

[215] O. Febbraro, K. Reale and F. Ricca. 'ASPIDE: Integrated Development Environment for Answer Set Programming'. In: *[171]*. 2011, pp. 317–330 (cit. on p. 250).

[216] D. Fensel, F. van Harmelen, B. Andersson, P. Brennan, H. Cunningham, E. D. Valle, F. Fischer, Z. Huang, A. Kiryakov, T. K. Lee, L. Schooler, V. Tresp, S. Wesner, M. J. Witbrock and N. Zhong. 'Towards LarKC: A Platform for Web-Scale Reasoning'. In: *Proceedings of ICSC 2008*, pp. 524–529. DOI: 10.1109/ICSC.2008.41 (cit. on p. 70).

[217] L. Ferreira and C. F. M. Toledo. 'A search-based approach for generating Angry Birds levels'. In: *[538]*. 2014, pp. 1–8. DOI: 10.1109/CIG.2014.6932912 (cit. on p. 182).

[218] D. A. Ferrucci. 'Build Watson: an overview of DeepQA for the Jeopardy! challenge'. In: *Proceedings of PACT 2010*, pp. 1–2. DOI: 10.1145/1854273.1854275 (cit. on p. 165).

[219] D. A. Ferrucci. 'Introduction to "This is Watson"'. In: *IBM Journal of Research and Development* 56.3 (2012), p. 1. DOI: 10.1147/JRD.2012.2184356 (cit. on p. 165).

[220] D. A. Ferrucci, E. W. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. M. Prager, N. Schlaefer and C. A. Welty. 'Building Watson: An Overview of the DeepQA Project'. In: *AI Magazine* 31.3 (2010), pp. 59–79. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2303 (cit. on p. 165).

[221] R. Fikes and N. J. Nilsson. 'STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving'. In: *Artif. Intell.* 2.3/4 (1971), pp. 189–208. DOI: 10.1016/0004-3702(71)90010-5 (cit. on p. 40).

[222] M. Fink, S. Germano, G. Ianni, C. Redl and P. Schüller. 'ActHEX: Implementing HEX Programs with Action Atoms'. In: *[115]*. 2013, pp. 317–322. DOI: 10.1007/978-3-642-40564-8_31 (cit. on pp. 37, 39, 93, 185).

[223] C. Forgy. 'Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem'. In: *Artif. Intell.* 19.1 (1982), pp. 17–37. DOI: 10.1016/0004-3702(82)90020-0 (cit. on pp. 74, 146).

[224] M. Fox and D. Long. 'PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains'. In: *JAIR* 20 (2003), pp. 61–124. DOI: 10.1613/jair.1129 (cit. on p. 40).

[225] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky and N. Thombre. 'Continuous Analytics: Rethinking Query Processing in a Network-Effect World'. In: *Proceedings of CIDR 2009*. URL: http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_122.pdf (cit. on p. 62).

[226] T. W. Frühwirth. *Constraint Handling Rules*. 1st. New York, NY, USA: Cambridge University Press, 2009. ISBN: 9780521877763 (cit. on p. 21).

[227] T. Furche, G. Gottlob, G. Grasso, O. Gunes, X. Guo, A. Kravchenko, G. Orsi, C. Schallhart, A. J. Sellers and C. Wang. 'DIADEM: domain-centric, intelligent, automated data extraction methodology'. In: *Proceedings of WWW 2012*, pp. 267–270. DOI: 10.1145/2187980.2188025 (cit. on p. 134).

[228] T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi and C. Schallhart. 'The ontological key: automatically understanding and integrating forms to access the deep Web'. In: *VLDB J.* 22.5 (2013), pp. 615–640. DOI: 10.1007/s00778-013-0323-0 (cit. on p. 134).

[229] T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, C. Schallhart and C. Wang. 'DIADEM: Thousands of Websites to a Single Database'. In: *PVLDB* 7.14 (2014), pp. 1845–1856. URL: http://www.vldb.org/pvldb/vol7/p1845-furche.pdf (cit. on p. 134).

[230] T. Furche, G. Gottlob, L. Libkin, G. Orsi and N. W. Paton. 'Data Wrangling for Big Data: Challenges and Opportunities'. In: *Proceedings of EDBT 2016*, pp. 473–478. DOI: 10.5441/002/edbt.2016.44 (cit. on p. 134).

[231] T. Furche, G. Gottlob, B. Neumayr and E. Sallinger. 'Data Wrangling for Big Data: Towards a Lingua Franca for Data Wrangling'. In: *Proceedings of AMW 2016*. URL: http://ceur-ws.org/Vol-1644/paper20.pdf (cit. on pp. 133, 135).

[232] D. Fuscà. 'Tools and Techniques for Easing the Application of Answer Set Programming'. PhD thesis. Dipartimento di Matematica e Informatica - Università della Calabria, 2017 (cit. on p. 217).

[233] D. Fuscà, S. Germano, J. Zangari, M. Anastasio, F. Calimeri and S. Perri. 'A framework for easing the development of applications embedding answer set programming'. In: *Proceedings of PPDP 2016*, pp. 38–49. DOI: 10.1145/2967973.2968594 (cit. on pp. xii, 216, 238, 239, 249).

[234] D. Fuscà, S. Germano, J. Zangari, F. Calimeri and S. Perri. 'Answer Set Programming and Declarative Problem Solving in Game AIs'. In: *[45]*. 2013, pp. 81–88. URL: http://ceur-ws.org/Vol-1107/paper9.pdf (cit. on pp. xii, 170).

[235] H. Gaifman and E. Y. Shapiro. 'Fully Abstract Compositional Semantics for Logic Programs'. In: *Proceedings of POPL 1989*, pp. 134–142. DOI: 10.1145/75277.75289 (cit. on p. 83).

[236] J. R. Galliers. 'A theoretical framework for computer models of cooperative dialogue, acknowledging multi-agent conflict'. PhD thesis. Open University, 1988 (cit. on p. 166).

[237] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (cit. on p. 239).

[238] A. Gandomi and M. Haider. 'Beyond the hype: Big data concepts, methods, and analytics'. In: *Int J. Information Management* 35.2 (2015), pp. 137–144. DOI: 10.1016/j.ijinfomgt.2014.10.007 (cit. on p. 125).

[239] S. Ganguly, P. B. Gibbons, Y. Matias and A. Silberschatz. 'Bifocal Sampling for Skew-Resistant Join Size Estimation'. In: *[309]*. 1996, pp. 271–281. DOI: 10.1145/233269.233340 (cit. on p. 149).

[240] J. Gantz and D. Reinsel. 'Extracting value from chaos'. In: *IDC iView* 1142.2011 (2011), pp. 1–12 (cit. on p. 124).

[241] F. Gao, M. I. Ali, E. Curry and A. Mileo. 'QoS-aware adaptation for complex event service'. In: *Proceedings of SAC 2016*. ACM, pp. 1597–1604. DOI: 10.1145/2851613.2851806 (cit. on p. 118).

[242] F. Gao, E. Curry, M. I. Ali, S. Bhiri and A. Mileo. 'QoS-Aware Complex Event Service Composition and Optimization Using Genetic Algorithms'. In: *Proceedings of ICSOC 2014*. Springer, pp. 386–393. DOI: 10.1007/978-3-662-45391-9_28 (cit. on p. 118).

[243] M. N. Garofalakis, J. Gehrke and R. Rastogi, eds. *Data Stream Management - Processing High-Speed Data Streams*. DCSA. Springer, 2016. ISBN: 978-3-540-28607-3. DOI: 10.1007/978-3-540-28608-0 (cit. on pp. 267, 273).

[244] M. Garofalakis, J. Gehrke and R. Rastogi. *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 3540286071 (cit. on p. 55).

[245] X. Ge, J. H. Lee, J. Renz and P. Zhang. 'Hole in One: Using Qualitative Reasoning for Solving Hard Physical Puzzle Problems'. In: *Proceedings of ECAI 2016 and PAIS 2016*, pp. 1762–1763. DOI: 10.3233/978-1-61499-672-9-1762 (cit. on p. 182).

[246] X. Ge and J. Renz. 'Tracking Perceptually Indistinguishable Objects Using Spatial Reasoning'. In: *Proceedings of PRICAI 2014*, pp. 600–613. DOI: 10.1007/978-3-319-13560-1_48 (cit. on p. 182).

[247] X. Ge and J. Renz. 'Representation and Reasoning about General Solid Rectangles'. In: *[509]*. 2013, pp. 905–911. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6880 (cit. on p. 182).

[248] X. Ge, J. Renz and P. Zhang. 'Visual Detection of Unknown Objects in Video Games Using Qualitative Stability Analysis'. In: *TCIAIG* 8.2 (2016), pp. 166–177. DOI: 10.1109/TCIAIG.2015.2506741 (cit. on p. 182).

[249] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu and T. Schaub. 'Stream reasoning with answer set programming: Extended version'. Available at oclingo website. 2012 (cit. on pp. 82, 90).

[250] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu and T. Schaub. 'Stream Reasoning with Answer Set Programming: Preliminary Report'. In: *[101]*. 2012. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4504 (cit. on pp. 82, 90, 102).

[251] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu and T. Schaub. 'Answer Set Programming for Stream Reasoning'. In: *CoRR* abs/1301.1392 (2013). URL: http://arxiv.org/abs/1301.1392 (cit. on pp. 82, 102).

[252] M. Gebser, T. Grote, R. Kaminski and T. Schaub. 'Reactive Answer Set Programming'. In: *[171]*. 2011, pp. 54–66. DOI: 10.1007/978-3-642-20895-9_7 (cit. on pp. 34, 82, 87, 88).

[253] M. Gebser, T. Guyet, R. Quiniou, J. Romero and T. Schaub. 'Knowledge-Based Sequence Mining with ASP'. In: *[322]*. 2016, pp. 1497–1504. URL: http://www.ijcai.org/Abstract/16/215 (cit. on p. 27).

[254] M. Gebser, T. Janhunen, H. Jost, R. Kaminski and T. Schaub. 'ASP Solving for Expanding Universes'. In: *[136]*. 2015, pp. 354–367. DOI: 10.1007/978-3-319-23264-5_30 (cit. on p. 96).

[255] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub and S. Thiele. 'Engineering an Incremental ASP Solver'. In: *[51]*. 2008, pp. 190–205. DOI: 10.1007/978-3-540-89982-2_23 (cit. on pp. 34, 82, 86–88).

[256] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub and P. Wanko. 'Theory Solving Made Easy with Clingo 5'. In: *Proceedings of ICLP 2016*, 2:1–2:15. DOI: 10.4230/OASIcs.ICLP.2016.2 (cit. on pp. 33, 35).

[257] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. 'Challenges in Answer Set Solving'. In: *[48]*. 2011, pp. 74–90. DOI: 10.1007/978-3-642-20832-4_6 (cit. on pp. 28, 141).

[258] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. DOI: 10.2200/S00457ED1V01Y201211AIM019 (cit. on p. 39).

[259] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. *Clingo = ASP + Control: Extended Report*. 2014 (cit. on pp. 35, 82).

[260] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. 'Clingo = ASP + Control: Preliminary Report'. In: *CoRR* abs/1405.3694 (2014). URL: http://arxiv.org/abs/1405.3694 (cit. on pp. 33, 112, 235).

[261] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. 'Multi-shot ASP solving with clingo'. In: *CoRR* abs/1705.09811 (2017). URL: http://arxiv.org/abs/1705.09811 (cit. on pp. 35, 96).

[262] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. T. Schneider and S. Ziller. 'A Portfolio Solver for Answer Set Programming: Preliminary Report'. In: [171]. 2011, pp. 352–357. DOI: 10.1007/978-3-642-20895-9_40 (cit. on pp. 141, 146).

[263] M. Gebser, R. Kaminski and T. Schaub. 'Complex optimization in answer set programming'. In: *TPLP* 11.4-5 (2011), pp. 821–839. DOI: 10.1017/S1471068411000329 (cit. on p. 39).

[264] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub and M. T. Schneider. 'Potassco: The Potsdam Answer Set Solving Collection'. In: *AI Commun.* 24.2 (2011), pp. 107–124. DOI: 10.3233/AIC-2011-0491 (cit. on pp. 21, 34, 220).

[265] M. Gebser, B. Kaufmann, A. Neumann and T. Schaub. 'Conflict-Driven Answer Set Solving'. In: *Proceedings of IJCAI 2007*, p. 386. URL: http://ijcai.org/Proceedings/07/Papers/060.pdf (cit. on pp. 30, 34).

[266] M. Gebser, B. Kaufmann and T. Schaub. 'Conflict-driven answer set solving: From theory to practice'. In: *Artif. Intell.* 187 (2012), pp. 52–89. DOI: 10.1016/j.artint.2012.04.001 (cit. on p. 30).

[267] M. Gebser, B. Kaufmann and T. Schaub. 'Advanced Conflict-Driven Disjunctive Answer Set Solving'. In: [509]. 2013, pp. 912–918. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6835 (cit. on p. 30).

[268] M. Gebser, M. Maratea and F. Ricca. 'The Sixth Answer Set Programming Competition'. In: *JAIR* 60 (2017), pp. 41–95. DOI: 10.1613/jair.5373 (cit. on p. 23).

[269] M. Gebser and T. Schaub. 'Modeling and Language Extensions'. In: *AI Magazine* 37.3 (2016), pp. 33–44. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2673 (cit. on pp. 23, 28).

[270] M. Gebser, T. Schaub and S. Thiele. 'GrinGo : A New Grounder for Answer Set Programming'. In: *Proceedings of LPNMR 2007*, pp. 266–271. DOI: 10.1007/978-3-540-72200-7_24 (cit. on p. 33).

[271] M. Gelfond. 'Answer Sets'. In: *Handbook of Knowledge Representation*. 2008, pp. 285–316. DOI: 10.1016/S1574-6526(07)03007-6 (cit. on p. 39).

[272] M. Gelfond and N. Leone. 'Logic programming and knowledge representation - The A-Prolog perspective'. In: *Artif. Intell.* 138.1-2 (2002), pp. 3–38. DOI: 10.1016/S0004-3702(02)00207-2 (cit. on p. 6).

[273] M. Gelfond and V. Lifschitz. 'The Stable Model Semantics for Logic Programming'. In: *Proceedings of ICLP 1988*, pp. 1070–1080 (cit. on pp. 21, 39).

[274] M. Gelfond and V. Lifschitz. 'Classical Negation in Logic Programs and Disjunctive Databases'. English. In: *New Generation Comput.* 9.3/4 (1991), pp. 365–386. ISSN: 0288-3635. DOI: 10.1007/BF03037169 (cit. on pp. 14, 21, 23, 25, 36).

[275] M. R. Genesereth and S. P. Ketchpel. 'Software Agents'. In: *Commun. ACM* 37.7 (1994), pp. 48–53. DOI: 10.1145/176789.176794 (cit. on p. 165).

[276] A. Gerevini, P. Haslum, D. Long, A. Saetti and Y. Dimopoulos. 'Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners'. In: *Artif. Intell.* 173.5-6 (2009), pp. 619–668. DOI: 10.1016/j.artint.2008.10.012 (cit. on pp. 21, 40).

[277] S. Germano, F. Calimeri and E. Palermiti. 'LoIDE: A Web-Based IDE for Logic Programming Preliminary Report'. In: *Proceedings of PADL 2018*, pp. 152–160. DOI: 10.1007/978-3-319-73305-0_10 (cit. on pp. xiii, 241).

[278] S. Germano, F. Calimeri and E. Palermiti. 'LoIDE: a web-based IDE for Logic Programming - Preliminary Technical Report'. In: *CoRR* abs/1709.05341 (2017). arXiv: 1709.05341 (cit. on pp. xii, 241).

[279] S. Germano, T. Pham and A. Mileo. 'Web Stream Reasoning in Practice: On the Expressivity vs. Scalability Tradeoff'. In: *Proceedings of RR 2015*, pp. 105–112. DOI: 10.1007/978-3-319-22002-4_9 (cit. on pp. xii, 102).

[280] M. Ghallab, D. S. Nau and P. Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN: 978-1-55860-856-6 (cit. on p. 169).

[281] R. Giaffreda. 'iCore: A Cognitive Management Framework for the Internet of Things'. In: *Proceedings of FIA 2013*. Springer, pp. 350–352. DOI: 10.1007/978-3-642-38082-2_31 (cit. on p. 109).

[282] M. L. Ginsberg, ed. *Readings in Nonmonotonic Reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987. ISBN: 0-934613-45-1 (cit. on p. 294).

[283] L. Giordano, A. Martelli and D. T. Dupré. 'Reasoning about actions with Temporal Answer Sets'. In: *TPLP* 13.2 (2013), pp. 201–225. DOI: 10.1017/S1471068411000639 (cit. on p. 183).

[284] E. Giunchiglia, Y. Lierler and M. Maratea. 'Answer Set Programming Based on Propositional Satisfiability'. In: *J. Autom. Reasoning* 36.4 (2006), pp. 345–377. ISSN: 0168-7433. DOI: 10.1007/s10817-006-9033-2 (cit. on p. 30).

[285] R. Goodwin. 'Formalizing Properties of Agents'. In: *J. Log. Comput.* 5.6 (1995), pp. 763–781. DOI: 10.1093/logcom/5.6.763 (cit. on p. 166).

[286] G. Gottlob, T. Lukasiewicz and A. Pieris. 'Datalog+/-: Questions and Answers'. In: *[57]*. 2014. URL: http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7965 (cit. on p. 15).

[287] G. Gottlob, M. Manna and A. Pieris. 'Combining decidability paradigms for existential rules'. In: *TPLP* 13.4-5 (2013), pp. 877–892. DOI: 10.1017/S1471068413000550 (cit. on p. 17).

[288] G. Gottlob, G. Orsi and A. Pieris. 'Ontological queries: Rewriting and optimization'. In: *Proceedings of ICDE 2011*, pp. 2–13. DOI: 10.1109/ICDE.2011.5767965 (cit. on p. 17).

[289] G. Gottlob, G. Orsi and A. Pieris. 'Query Rewriting and Optimization for Ontological Databases'. In: *TODS* 39.3 (2014), 25:1–25:46. DOI: 10.1145/2638546 (cit. on p. 20).

[290] G. Gottlob, R. Pichler and F. Wei. 'Bounded treewidth as a key to tractability of knowledge representation and reasoning'. In: *Artif. Intell.* 174.1 (2010), pp. 105–132. DOI: 10.1016/j.artint.2009.10.003 (cit. on p. 142).

[291] G. Gottlob and P. Senellart. 'Schema mapping discovery from data instances'. In: *J. ACM* 57.2 (2010), 6:1–6:37. DOI: 10.1145/1667053.1667055 (cit. on p. 138).

[292] G. Grasso, N. Leone, M. Manna and F. Ricca. 'ASP at Work: Spin-off and Applications of the DLV System'. In: *[48]*. 2011, pp. 432–451. DOI: 10.1007/978-3-642-20832-4_27 (cit. on p. 214).

[293] B. C. Grau, I. Horrocks, M. Krötzsch, C. Kupke, D. Magka, B. Motik and Z. Wang. 'Acyclicity Notions for Existential Rules and Their Application to Query Answering in Ontologies'. In: *JAIR* 47 (2013), pp. 741–808. DOI: 10.1613/jair.3949 (cit. on p. 17).

[294] T. Grote. 'A reactive System for Declarative Programming of Dynamic Applications'. Diploma Thesis. Knowledge Processing and Information Systems, Institute for Computer Science, University of Potsdam, 8th Sept. 2010 (cit. on p. 82).

[295] N. Gupta and D. S. Nau. 'On the Complexity of Blocks-World Planning'. In: *Artif. Intell.* 56.2-3 (1992), pp. 223–254. DOI: 10.1016/0004-3702(92)90028-V (cit. on p. 43).

[296] Y. Gurevich, D. Leinders and J. V. den Bussche. 'A Theory of Stream Queries'. In: *Proceedings of DBPL 2007*, pp. 153–168. DOI: 10.1007/978-3-540-75987-4_11 (cit. on p. 56).

[297] C. Gutierrez, C. A. Hurtado and A. A. Vaisman. 'Introducing Time into RDF'. In: *TKDE* 19.2 (2007), pp. 207–218. DOI: 10.1109/TKDE.2007.34 (cit. on p. 71).

[298] A. Y. Halevy, Z. G. Ives and A. Doan, eds. *Proceedings of ACM SIGMOD 2003*. (San Diego, California, USA). ACM, 9th–12th June 2003. ISBN: 1-58113-634-X (cit. on pp. 265, 274).

[299] A. Harrison. 'Formal Methods for Answer Set Programming'. In: *Proceedings of ICLP 2015*. URL: http://ceur-ws.org/Vol-1433/dc_2.pdf (cit. on p. 23).

[300] M. Helmert, G. Röger and E. Karpas. 'Fast downward stone soup: A baseline for building planner portfolios'. In: *Workshop on Planning and Learning at ICAPS 2011*, pp. 28–35 (cit. on p. 146).

[301] M. Hendrikx, S. Meijer, J. V. D. Velden and A. Iosup. 'Procedural content generation for games: A survey'. In: *TOMCCAP* 9.1 (2013), 1:1–1:22. DOI: 10.1145/2422956.2422957 (cit. on p. 169).

[302] P. Hitzler and T. Lukasiewicz, eds. *Proceedings of RR 2010*. (Bressanone/Brixen, Italy). Vol. 6333. LNCS. Springer, 22nd–24th Sept. 2010. ISBN: 978-3-642-15917-6. DOI: 10.1007/978-3-642-15918-3 (cit. on pp. 266, 272).

[303] H. Hoos, M. T. Lindauer and T. Schaub. 'claspfolio 2: Advances in Algorithm Selection for Answer Set Programming'. In: *TPLP* 14.4-5 (2014), pp. 569–585. DOI: 10.1017/S1471068414000210 (cit. on pp. 141, 146).

[304] G. Ianni, D. Lembo, L. E. Bertossi, W. Faber, B. Glimm, G. Gottlob and S. Staab, eds. *Tutorial Lectures of RW 2017*. (London, UK). Vol. 10370. LNCS. Springer, 7th–11th July 2017. ISBN: 978-3-319-61032-0. DOI: 10.1007/978-3-319-61033-7 (cit. on pp. 277, 284).

[305] Y. E. Ioannidis. 'The History of Histograms (abridged)'. In: *Proceedings of VLDB 2003*, pp. 19–30. URL: http://www.vldb.org/conf/2003/papers/S02P01.pdf (cit. on pp. 148, 149).

[306] Y. E. Ioannidis and V. Poosala. 'Balancing Histogram Optimality and Practicality for Query Result Size Estimation'. In: *Proceedings of ACM SIGMOD 1995*, pp. 233–244. DOI: 10.1145/223784.223841 (cit. on p. 149).

[307] Y. E. Ioannidis and V. Poosala. 'Histogram-Based Approximation of Set-Valued Query-Answers'. In: *Proceedings of VLDB 1999*, pp. 174–185. URL: http://www.vldb.org/conf/1999/P15.pdf (cit. on p. 149).

[308] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs and R. Braynard. 'Networking named content'. In: *Proceedings of CoNEXT 2009*, pp. 1–12. DOI: 10.1145/1658939.1658941 (cit. on p. 96).

[309] H. V. Jagadish and I. S. Mumick, eds. *Proceedings of ACM SIGMOD 1996*. (Montreal, Quebec, Canada). ACM Press, 4th–6th June 1996 (cit. on pp. 279, 293).

[310] T. Janhunen and I. Niemelä. 'The Answer Set Programming Paradigm'. In: *AI Magazine* 37.3 (2016), pp. 13–24. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2671 (cit. on p. 39).

[311] T. Janhunen, I. Niemelä, D. Seipel, P. Simons and J. You. 'Unfolding partiality and disjunctions in stable model semantics'. In: *TOCL* 7.1 (2006), pp. 1–37. DOI: 10.1145/1119439.1119440 (cit. on p. 30).

[312] G. Jaskiewicz. 'Prolog-Scripted Tactics Negotiation and Coordinated Team Actions for Counter-Strike Game Bots'. In: *TCIAIG* 8.1 (2016), pp. 82–88. DOI: 10.1109/TCIAIG.2014.2331972 (cit. on p. 169).

[313] Y. Jiang, T. Harada and R. Thawonmas. 'Procedural generation of angry birds fun levels using pattern-struct and preset-model'. In: *[477]*. 2017, pp. 154–161. DOI: 10.1109/CIG.2017.8080429 (cit. on p. 182).

[314] D. S. Johnson and A. C. Klug. 'Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies'. In: *JCSS* 28.1 (1984), pp. 167–189. DOI: 10.1016/0022-0000(84)90081-3 (cit. on p. 18).

[315] M. Johnson, K. Hofmann, T. Hutton and D. Bignell. 'The Malmo Platform for Artificial Intelligence Experimentation'. In: *[322]*. 2016, pp. 4246–4247. URL: http://www.ijcai.org/Abstract/16/643 (cit. on p. 258).

[316] T. Johnson, S. Muthukrishnan, V. Shkapenyuk and O. Spatscheck. 'A Heartbeat Mechanism and Its Application in Gigascope'. In: *Proceedings of VLDB 2005*, pp. 1079–1088. URL: http://www.vldb.org/archives/website/2005/program/paper/tue/p1079-johnson.pdf (cit. on p. 59).

[317] H. Jordan, B. Scholz and P. Subotic. 'Soufflé: On Synthesis of Program Analyzers'. In: *Proceedings of CAV 2016, Part II*, pp. 422–430. DOI: 10.1007/978-3-319-41540-6_23 (cit. on p. 19).

[318] H. S. W. Jr. 'A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations'. In: *Commun. ACM* 18.4 (1975), pp. 218–220. DOI: 10.1145/360715.360746 (cit. on p. 15).

[319] A. Jutzeler, M. Katanic and J. J. Li. 'Managing Luck: A Multi-Armed Bandits Meta-Agent for the Angry Birds Competition'. 2013. URL: http://aibirds.org/2013-Papers/Team-Descriptions/beaurivage.pdf (cit. on pp. 180, 181).

[320] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz and M. Sellmann. 'Algorithm Selection and Scheduling'. In: *Proceedings of CP 2011*, pp. 454–469. DOI: 10.1007/978-3-642-23786-7_35 (cit. on p. 146).

[321] S. Kadioglu, Y. Malitsky, M. Sellmann and K. Tierney. 'ISAC - Instance-Specific Algorithm Configuration'. In: *Proceedings of ECAI 2010*, pp. 751–756. DOI: 10.3233/978-1-60750-606-5-751 (cit. on p. 146).

[322] S. Kambhampati, ed. *Proceedings of IJCAI 2016*. (New York, NY, USA). IJCAI/AAAI Press, 9th–15th July 2016. ISBN: 978-1-57735-770-4. URL: http://www.ijcai.org/Proceedings/2016 (cit. on pp. 270, 280, 284).

[323] R. Kaminski, T. Schaub and P. Wanko. 'A Tutorial on Hybrid Answer Set Solving with clingo'. In: *[304]*. 2017, pp. 167–203. DOI: 10.1007/978-3-319-61033-7_6 (cit. on p. 35).

[324] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck and P. Buono. 'Research directions in data wrangling: Visualizations and transformations for usable and credible data'. In: *Information Visualization* 10.4 (2011), pp. 271–288. DOI: 10.1177/1473871611415994 (cit. on p. 134).

[325] B. Kaufmann, N. Leone, S. Perri and T. Schaub. 'Grounding and Solving in Answer Set Programming'. In: *AI Magazine* 37.3 (2016), pp. 25–32. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2672 (cit. on p. 22).

[326] G. Kendall. 'Editorial: IEEE Transactions on Computational Intelligence and AI in Games'. In: *TCIAIG* 7.1 (2015), pp. 1–2. DOI: 10.1109/TCIAIG.2015.2409514 (cit. on p. 164).

[327] M. A. u. d. Khan, M. F. Uddin and N. Gupta. 'Seven V's of Big Data understanding Big Data to extract value'. In: *Proceedings of the 2014 ASEE Zone 1 Conference*. IEEE, pp. 1–5. DOI: 10.1109/ASEEZone1.2014.6820689 (cit. on p. 125).

[328] T. Kindberg, M. Chalmers and E. Paulos. 'Guest Editors' Introduction: Urban Computing'. In: *IEEE Pervasive Comput.* 6.3 (2007), pp. 18–20. DOI: 10.1109/MPRV.2007.57 (cit. on p. 51).

[329] P. Kissmann and S. Edelkamp. 'Instantiating General Games Using Prolog or Dependency Graphs'. In: *Proceedings of KI 2010*, pp. 255–262. DOI: 10.1007/978-3-642-16111-7_29 (cit. on p. 169).

[330] C. Kloimüllner, J. Oetsch, J. Pührer and H. Tompits. 'Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs'. In: *Proceedings of INAP and WLP 2011*, pp. 325–344. DOI: 10.1007/978-3-642-41524-1_20 (cit. on p. 251).

[331] P. G. Kolaitis. *On the expressive power of stratified datalog programs*. Preprint. 1987 (cit. on p. 14).

[332] P. G. Kolaitis. 'The Expressive Power of Stratified Programs'. In: *Inf. Comput.* 90.1 (1991), pp. 50–66. DOI: 10.1016/0890-5401(91)90059-B (cit. on p. 14).

[333] P. G. Kolaitis and C. H. Papadimitriou. 'Why not Negation by Fixpoint?' In: *JCSS* 43.1 (1991), pp. 125–144. DOI: 10.1016/0022-0000(91)90033-2 (cit. on p. 14).

[334] M. Kolchin and P. Wetz. *Demo: YABench - Yet Another RDF Stream Processing Benchmark*. 2015 (cit. on p. 75).

[335] M. Kolchin, P. Wetz, E. Kiesling and A. M. Tjoa. 'YABench: A Comprehensive Framework for RDF Stream Processor Correctness and Performance Assessment'. In: *Proceedings of ICWE 2016*, pp. 280–298. DOI: 10.1007/978-3-319-38791-8_16 (cit. on pp. 75, 80).

[336] S. Komazec, D. Cerri and D. Fensel. 'Sparkwave: continuous schema-enhanced pattern matching over RDF data streams'. In: *Proceedings of DEBS 2012*, pp. 58–68. DOI: 10.1145/2335484.2335491 (cit. on p. 74).

[337] M. König, M. Leclère, M. Mugnier and M. Thomazo. 'A Sound and Complete Backward Chaining Algorithm for Existential Rules'. In: *[349]*. 2012, pp. 122–138. DOI: 10.1007/978-3-642-33203-6_10 (cit. on p. 20).

[338] N. Konstantinou, M. Koehler, E. Abel, C. Civili, B. Neumayr, E. Sallinger, A. A. A. Fernandes, G. Gottlob, J. A. Keane, L. Libkin and N. W. Paton. 'The VADA Architecture for Cost-Effective Data Wrangling'. In: *Proceedings of ACM SIGMOD 2017*, pp. 1599–1602. DOI: 10.1145/3035918.3058730 (cit. on p. 130).

[339] D. L. Kovacs. 'BNF definition of PDDL 3.1'. From the IPC-2011 website. 2011 (cit. on p. 40).

[340] R. A. Kowalski. 'Predicate Logic as Programming Language'. In: *IFIP Congress*. 1974, pp. 569–574 (cit. on p. 6).

[341] R. A. Kowalski. 'Algorithm = Logic + Control'. In: *Commun. ACM* 22.7 (1979), pp. 424–436. DOI: 10.1145/359131.359136 (cit. on p. 6).

[342] R. A. Kowalski. *Logic for problem solving*. Vol. 7. The computer science library : Artificial intelligence series. North-Holland, 1979. ISBN: 0444003681. URL: http://www.worldcat.org/oclc/05564433 (cit. on p. 8).

[343] R. A. Kowalski. 'The Early Years of Logic Programming'. In: *Commun. ACM* 31.1 (1988), pp. 38–43. DOI: 10.1145/35043.35046 (cit. on pp. 5, 7, 213).

[344] R. A. Kowalski and F. Sadri. 'From Logic Programming Towards Multi-Agent Systems'. In: *AMAI* 25.3-4 (1999), pp. 391–419. DOI: 10.1023/A:1018934223383 (cit. on p. 185).

[345] M. Koziarkiewicz. *iGROM - an Integrated Development Environment for Answer Set Programs*. 2007–2010. URL: http://igrom.sourceforge.net (visited on 25th Sept. 2017) (cit. on p. 250).

[346] J. Krämer and B. Seeger. 'PIPES - A Public Infrastructure for Processing and Exploring Streams'. In: *Proceedings of ACM SIGMOD 2004*, pp. 925–926. DOI: 10.1145/1007568.1007699 (cit. on p. 62).

[347] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss and M. A. Shah. 'TelegraphCQ: An Architectural Status Report'. In: *IEEE Data Eng. Bull.* 26.1 (2003), pp. 11–18. URL: http://sites.computer.org/debull/A03mar/tcqde.ps (cit. on p. 62).

[348] M. Krötzsch and S. Rudolph. 'Extending Decidable Existential Rules by Joining Acyclicity and Guardedness'. In: *[576]*. 2011, pp. 963–968. DOI: 10.5591/978-1-57735-516-8/IJCAI11-166 (cit. on p. 17).

[349] M. Krötzsch and U. Straccia, eds. *RR 2012*. Vol. 7497: *Proceedings of RR 2012*. (Vienna, Austria). LNCS. Springer, 10th–12th Sept. 2012. ISBN: 978-3-642-33202-9. DOI: 10.1007/978-3-642-33203-6 (cit. on pp. 285, 287).

[350] G. M. Kuper. 'Logic Programming With Sets'. In: *Proceedings of PODS 1987*, pp. 11–20. DOI: 10.1145/28659.28661 (cit. on p. 15).

[351] A. L'Heureux, K. Grolinger, H. F. ElYamany and M. A. M. Capretz. 'Machine Learning With Big Data: Challenges and Approaches'. In: *IEEE Access* 5 (2017), pp. 7776–7797. DOI: 10.1109/ACCESS.2017.2696365 (cit. on pp. 124, 126).

[352] A. Labrinidis and H. V. Jagadish. 'Challenges and Opportunities with Big Data'. In: *PVLDB* 5.12 (2012), pp. 2032–2033. URL: http://vldb.org/pvldb/vol5/p2032_alexandroslabrinidis_vldb2012.pdf (cit. on pp. 126, 127).

[353] J. E. Laird. 'Using a Computer Game to Develop Advanced AI'. In: *IEEE Computer* 34.7 (2001), pp. 70–75. DOI: 10.1109/2.933506 (cit. on p. 164).

[354] R. Lapauw, I. Dasseville and M. Denecker. 'Visualising interactive inferences with IDPD3'. In: *CoRR* abs/1511.00928 (2015) (cit. on p. 251).

[355] R. Lara-Cabrera, M. N. Collazo, C. Cotta and A. J. F. Leiva. 'Game Artificial Intelligence: Challenges for the Scientific Community'. In: *Proceedings of CoSECivi 2015*, pp. 1–12. URL: http://ceur-ws.org/Vol-1394/paper_1.pdf (cit. on p. 167).

[356] Y. Law, H. Wang and C. Zaniolo. 'Relational languages and data models for continuous queries on sequences and data streams'. In: *TODS* 36.2 (2011), 8:1–8:32. DOI: 10.1145/1966385.1966386 (cit. on p. 56).

[357] F. Lécué, S. Tallevi-Diotallevi, J. Hayes, R. Tucker, V. Bicer, M. L. Sbodio and P. Tommasi. 'Smart traffic analytics in the semantic web with STAR-CITY: Scenarios, system and lessons learned in Dublin City'. In: *J. Web Sem.* 27 (2014), pp. 26–33. DOI: 10.1016/j.websem.2014.07.002 (cit. on p. 109).

[358] J.-S. Lee, H.-S. Seon, J.-H. Kim, S.-Y. Joo and K.-J. Kim. 'Angry Plan A+ Team Team Description Paper'. 2014. URL: https://aibirds.org/2014-papers/PlanA+.pdf (cit. on pp. 180, 182).

[359] M. Lenzerini. 'Data Integration: A Theoretical Perspective'. In: *[475]*. 2002, pp. 233–246. DOI: 10.1145/543613.543644 (cit. on p. 138).

[360] N. Leone, M. Manna, G. Terracina and P. Veltri. 'Efficient Query Answering over Datalog with Existential Quantifiers'. In: *Proceedings of SEBD 2012*, pp. 155–162. URL: http://sebd2012.dei.unipd.it/documents/188475/cce9a9ef-dac8-4d0b-b05c-a15806050538 (cit. on p. 17).

[361] N. Leone, M. Manna, G. Terracina and P. Veltri. 'Efficiently Computable Datalog∃ Programs'. In: *[101]*. 2012. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4521 (cit. on pp. 15, 17, 20).

[362] N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri and A. Polleres. 'The DLV System'. In: *Proceedings of JELIA 2002*, pp. 537–540. DOI: 10.1007/3-540-45757-7_50 (cit. on p. 31).

[363] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri and F. Scarcello. 'The DLV system for knowledge representation and reasoning'. In: *TOCL* 7.3 (2006), pp. 499–562. DOI: 10.1145/1149114.1149117 (cit. on pp. 21, 27, 30, 31).

[364] N. Leone and F. Ricca. 'Answer Set Programming: A Tour from the Basics to Advanced Development Tools and Industrial Applications'. In: *Tutorial Lectures of RW 2015*, pp. 308–326. DOI: 10.1007/978-3-319-21768-0_10 (cit. on pp. 27, 204).

[365] J. Leskovec, A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014. ISBN: 978-1107077232 (cit. on p. 129).

[366] B. L. Lewis. 'In the game: The interface between Watson and Jeopardy!' In: *IBM Journal of Research and Development* 56.3 (2012), p. 17. DOI: 10.1147/JRD.2012.2188932 (cit. on p. 165).

[367] J. J. Li. 'Qualitative Spatial and Temporal Reasoning with Answer Set Programming'. In: *Proceedings of ICTAI 2012*, pp. 603–609. DOI: 10.1109/ICTAI.2012.87 (cit. on p. 183).

[368] Q. Li, X. Zhang and Z. Feng. 'PRSP: A Plugin-based Framework for RDF Stream Processing'. In: *Proceedings of WWW 2017*, pp. 815–816. DOI: 10.1145/3041021.3054243 (cit. on p. 74).

[369] S. Liang. 'Non-termination Analysis and Cost-Based Query Optimization of Logic Programs'. In: *[349]*. 2012, pp. 284–290. DOI: 10.1007/978-3-642-33203-6_33 (cit. on p. 155).

[370] S. Liang and M. Kifer. 'Deriving predicate statistics in datalog'. In: *Proceedings of PPDP 2010*, pp. 45–56. DOI: 10.1145/1836089.1836095 (cit. on pp. 148, 153, 155).

[371] S. Liang and M. Kifer. 'Deriving Predicate Statistics for Logic Rules'. In: *[349]*. 2012, pp. 139–155. DOI: 10.1007/978-3-642-33203-6_11 (cit. on pp. 148, 153–156).

[372] Y. Lierler, M. Maratea and F. Ricca. 'Systems, Engineering Environments, and Competitions'. In: *AI Magazine* 37.3 (2016), pp. 45–52. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2675 (cit. on p. 214).

[373] V. Lifschitz. 'Answer Set Planning'. In: *Proceedings of ICLP 1999*, pp. 23–37 (cit. on pp. 5, 39).

[374] V. Lifschitz. 'What Is Answer Set Programming?' In: *Proceedings of AAAI 2008*, pp. 1594–1597. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-270.php (cit. on p. 21).

[375] V. Lifschitz. 'Answer set programming and plan generation'. In: *Artif. Intell.* 138.1-2 (2002), pp. 39–54. DOI: 10.1016/S0004-3702(02)00186-8 (cit. on pp. 21, 183).

[376] V. Lifschitz. 'Datalog Programs and Their Stable Models'. In: *[428]*. 2010, pp. 78–87. DOI: 10.1007/978-3-642-24206-9_5 (cit. on p. 39).

[377] V. Lifschitz. 'Answer Sets and the Language of Answer Set Programming'. In: *AI Magazine* 37.3 (2016), pp. 7–12. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2670 (cit. on p. 39).

[378] V. Lifschitz. 'Achievements in answer set programming'. In: *TPLP* 17.5-6 (2017), pp. 961–973. DOI: 10.1017/S1471068417000345 (cit. on p. 35).

[379] F. Lin and Y. Zhao. 'ASSAT: computing answer sets of a logic program by SAT solvers'. In: *Artif. Intell.* 157.1-2 (2004), pp. 115–137. DOI: 10.1016/j.artint.2004.04.004 (cit. on p. 30).

[380] M. Lindauer, H. H. Hoos, F. Hutter and T. Schaub. 'AutoFolio: Algorithm Configuration for Algorithm Selection'. In: *Proceedings of AlgoConf 2015*. URL: http://aaai.org/ocs/index.php/WS/AAAIW15/paper/view/10106 (cit. on p. 146).

[381] R. J. Lipton and J. F. Naughton. 'Estimating the Size of Generalized Transitive Closures'. In: *Proceedings of VLDB 1989*, pp. 165–171. URL: http://www.vldb.org/conf/1989/P165.PDF (cit. on p. 149).

[382] R. J. Lipton and J. F. Naughton. 'Query Size Estimation by Adaptive Sampling'. In: *JCSS* 51.1 (1995), pp. 18–25. DOI: 10.1006/jcss.1995.1050 (cit. on p. 149).

[383] G. Liu, T. Janhunen and I. Niemelä. 'Answer Set Programming via Mixed Integer Programming'. In: *[101]*. 2012. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4516 (cit. on p. 30).

[384] L. LIU and M. T. ÖZSU, eds. *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009. ISBN: 978-0-387-39940-9 (cit. on pp. 267, 297).

[385] J. W. Lloyd. 'Practical Advtanages of Declarative Programming'. In: *Proceedings of GULP-PRODE 1994*, pp. 18–30 (cit. on p. 213).

[386] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987. ISBN: 3-540-18199-7 (cit. on pp. 5, 6, 213).

[387] J. Lobo, J. Minker and A. Rajasekar. *Foundations of disjunctive logic programming*. Logic Programming. MIT Press, 1992. ISBN: 978-0-262-12165-1 (cit. on p. 14).

[388] D. Long and M. Fox. 'The 3rd International Planning Competition: Results and Analysis'. In: *JAIR* 20 (2003), pp. 1–59. DOI: 10.1613/jair.1240 (cit. on p. 40).

[389] R. E. Ltd. *Angry Birds Chrome*. Discontinued. 2011–2015. URL: http://chrome.angrybirds.com (cit. on p. 177).

[390] S. M. Lucas. 'Computational intelligence and games: Challenges and opportunities'. In: *IJAC* 5.1 (1st Jan. 2008), pp. 45–57. ISSN: 1751-8520. DOI: 10.1007/s11633-008-0045-8 (cit. on p. 167).

[391] S. M. Lucas. 'Computational Intelligence and AI in Games: A New IEEE Transactions'. In: *TCIAIG* 1.1 (2009), pp. 1–3. DOI: 10.1109/TCIAIG.2009.2021433 (cit. on p. 164).

[392] D. Luckham. 'The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems'. In: *Proceedings of RuleML 2008*, p. 3. DOI: 10.1007/978-3-540-88808-6_2 (cit. on p. 66).

[393] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201727897 (cit. on p. 65).

[394] C. Macdonald, I. Ounis and I. Ruthven, eds. *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*. ACM. ISBN: 978-1-4503-0717-8 (cit. on p. 294).

[395] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko and C. Yu. 'Web-Scale Data Integration: You can afford to Pay as You Go'. In: *Proceedings of CIDR 2007*, pp. 342–350. URL: http://cidrdb.org/cidr2007/papers/cidr07p40.pdf (cit. on p. 138).

[396] P. Maes. 'Agents that Reduce Work and Information Overload'. In: *Commun. ACM* 37.7 (1994), pp. 30–40. DOI: 10.1145/176789.176792 (cit. on p. 166).

[397] R. Magoulas and B. Lorica. 'Big data: Technologies and techniques for large scale data'. In: *Jimmy Guterman, Release* 2 (2009) (cit. on p. 125).

[398] D. Maier, A. O. Mendelzon and Y. Sagiv. 'Testing Implications of Data Dependencies'. In: *TODS* 4.4 (1979), pp. 455–469. DOI: 10.1145/320107.320115 (cit. on p. 18).

[399] Y. Malitsky, A. Sabharwal, H. Samulowitz and M. Sellmann. 'Boosting Sequential Solver Portfolios: Knowledge Sharing and Accuracy Prediction'. In: *Proceedings of LION 2013*, pp. 153–167. DOI: 10.1007/978-3-642-44973-4_17 (cit. on p. 146).

[400] M. Maratea, L. Pulina and F. Ricca. 'Automated Selection of Grounding Algorithm in Answer Set Programming'. In: *Proceedings of AI*IA 2013*, pp. 73–84. DOI: 10.1007/978-3-319-03524-6_7 (cit. on pp. 141, 146).

[401] M. Maratea, L. Pulina and F. Ricca. 'On the Automated Selection of ASP Instantiators'. In: *Proceedings of GTTV 2013*, p. 39 (cit. on p. 141).

[402] M. Maratea, L. Pulina and F. Ricca. 'The Multi-Engine ASP Solver me-asp'. In: *Proceedings of JELIA 2012*, pp. 484–487. DOI: 10.1007/978-3-642-33353-8_39 (cit. on p. 141).

[403] M. Maratea, L. Pulina and F. Ricca. *ME-ASP: A Multi-Engine Solver for Answer Set Programming*. 2012 (cit. on p. 141).

[404] M. Maratea, L. Pulina and F. Ricca. 'A multi-engine approach to answer-set programming'. In: *TPLP* 14.6 (2014), pp. 841–868. DOI: 10.1017/S1471068413000094 (cit. on pp. 141, 146).

[405] M. Maratea, L. Pulina and F. Ricca. 'The Multi-engine ASP Solver ME-ASP: Progress Report'. In: *CoRR* abs/1405.0876 (2014). arXiv: 1405.0876 (cit. on p. 141).

[406] M. Maratea, L. Pulina and F. Ricca. 'Multi-engine ASP solving with policy adaptation'. In: *J. Log. Comput.* 25.6 (2015), pp. 1285–1306. DOI: 10.1093/logcom/ext068 (cit. on pp. 141, 146).

[407] M. Maratea, L. Pulina and F. Ricca. 'Multi-level Algorithm Selection for ASP'. In: *[136]*. 2015, pp. 439–445. DOI: 10.1007/978-3-319-23264-5_36 (cit. on p. 146).

[408] E. Marcopoulos, C. Reotutar and Y. Zhang. 'An Online Development Environment for Answer Set Programming'. In: *CoRR* abs/1707.01865 (2017) (cit. on p. 250).

[409] V. W. Marek and M. Truszczyski. 'Stable Models and an Alternative Logic Programming Paradigm'. In: *The Logic Programming Paradigm: A 25-Year Perspective*, pp. 375–398. DOI: 10.1007/978-3-642-60085-2_17 (cit. on p. 39).

[410] A. Margara, J. Urbani, F. van Harmelen and H. E. Bal. 'Streaming the Web: Reasoning over dynamic data'. In: *J. Web Sem.* 25 (2014), pp. 24–44. DOI: 10.1016/j.websem.2014.02.001 (cit. on pp. 51–53, 66, 74).

[411] M. Mariën, J. Wittocx, M. Denecker and M. Bruynooghe. 'SAT(ID): Satisfiability of Propositional Logic Extended with Inductive Definitions'. In: *Proceedings of SAT 2008*, pp. 211–224. DOI: 10.1007/978-3-540-79719-7_20 (cit. on p. 30).

[412] B. Marnette. 'Generalized schema-mappings: from termination to tractability'. In: *Proceedings of PODS 2009*, pp. 13–22. DOI: 10.1145/1559795.1559799 (cit. on p. 17).

[413] J. McCarthy. *Programs with common sense*. RLE and MIT Computation Center, 1960 (cit. on pp. 8, 213).

[414] J. McCarthy and P. J. Hayes. 'Some philosophical problems from the standpoint of artificial intelligence'. In: *Readings in artificial intelligence* (1969), pp. 431–450 (cit. on p. 8).

[415] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld and D. Wilkins. *PDDL-the planning domain definition language*. 1998 (cit. on pp. 21, 40).

[416] E. Mera, P. López-Garca, G. Puebla, M. Carro and M. V. Hermenegildo. 'Combining Static Analysis and Profiling for Estimating Execution Times'. In: *Proceedings of PADL 2007*, pp. 140–154. DOI: 10.1007/978-3-540-69611-7_9 (cit. on p. 156).

[417] E. Mera, P. López-Garca, G. Puebla, M. Carro and M. V. Hermenegildo. 'Using Combined Static Analysis and Profiling for Logic Program Execution Time Estimation'. In: *Proceedings of ICLP 2006*, pp. 431–432. DOI: 10.1007/11799573_36 (cit. on p. 156).

[418] E. Mera, P. López-Garca, G. Puebla, M. Carro and M. V. Hermenegildo. 'Towards Execution Time Estimation for Logic Programs via Static Analysis and Profiling'. In: *CoRR* abs/cs/0701108 (2007). arXiv: cs/0701108 (cit. on p. 156).

[419] A. Mileo, A. Abdelrahman, S. Policarpio and M. Hauswirth. 'StreamRule: A Nonmonotonic Stream Reasoning System for the Semantic Web'. In: *Proceedings of RR 2013*, pp. 247–252. DOI: 10.1007/978-3-642-39666-3_23 (cit. on pp. 96, 102, 103, 105).

[420] A. Mileo, M. Dao-Tran, T. Eiter and M. Fink. 'Stream Reasoning'. In: *Encyclopedia of Database Systems*. From http://doras.dcu.ie/21771. 2017 (cit. on pp. 48, 51, 81).

[421] A. Mileo, S. Germano, T.-L. Pham, D. Puiu, D. Kuemper and M. I. Ali. *User-Centric Decision Support in Dynamic Environments. CityPulse - Real-Time IoT Stream Processing and Large-scale Data Analytics for Smart City Applications*. Report - Project Delivery. Version V1.0-Final. NUIG, SIE, UASO, 31st Aug. 2015. URL: http://cordis.europa.eu/docs/projects/cnect/5/609035/080/deliverables/001-609035CITYPULSED52renditionDownload.pdf (visited on 25th Sept. 2017) (cit. on pp. xii, 109).

[422] A. Mileo, D. Merico, S. Pinardi and R. Bisiani. 'A Logical Approach to Home Healthcare with Intelligent Sensor-Network Support'. In: *Comput. J.* 53.8 (2010), pp. 1257–1276. DOI: 10.1093/comjnl/bxn074 (cit. on p. 27).

[423] A. Mileo, T. Schaub, D. Merico and R. Bisiani. 'Knowledge-based multi-criteria optimization to support indoor positioning'. In: *AMAI* 62.3-4 (2011), pp. 345–370. DOI: 10.1007/s10472-011-9241-2 (cit. on p. 96).

[424] I. Millington and J. Funge. *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann, 2009. ISBN: 978-0-12-374731-0 (cit. on p. 164).

[425] J. Minker. 'On Indefinite Databases and the Closed World Assumption'. In: *Proceedings of CADE 1982*, pp. 292–308. DOI: 10.1007/BFb0000066 (cit. on pp. 13, 21).

[426] J. Minker. 'Overview of Disjunctive Logic Programming'. In: *AMAI* 12.1-2 (1994), pp. 1–24. DOI: 10.1007/BF01530759 (cit. on p. 14).

[427] M. Minsky. *A Framework for Representing Knowledge*. Tech. rep. Cambridge, MA, USA, 1974 (cit. on p. 8).

[428] O. de Moor, G. Gottlob, T. Furche and A. J. Sellers, eds. *Proceedings of Datalog Reloaded 2010*. (Oxford, UK). Vol. 6702. LNCS. Springer, 16th–19th Mar. 2010. ISBN: 978-3-642-24205-2. DOI: 10.1007/978-3-642-24206-9 (cit. on pp. 266, 287).

[429] R. C. Moore. 'Semantical Considerations on Nonmonotonic Logic'. In: *Proceedings of IJCAI 1983*, pp. 272–279. URL: http://ijcai.org/Proceedings/83-1/Papers/063.pdf (cit. on p. 21).

[430] M. Morak, R. Pichler, S. Rümmele and S. Woltran. 'A Dynamic-Programming Based ASP-Solver'. In: *Proceedings of JELIA 2010*, pp. 369–372. DOI: 10.1007/978-3-642-15675-5_34 (cit. on p. 39).

[431] A. MoSSburger, H. Beck, M. Dao-Tran and T. Eiter. 'A Benchmarking Framework for Stream Processors'. In: *Proceedings of EKAW 2016 Satellite Events, EKM and Drift-an-LOD*, pp. 153–157. DOI: 10.1007/978-3-319-58694-6_21 (cit. on p. 96).

[432] L. M. de Moura and N. Bjørner. 'Satisfiability modulo theories: introduction and applications'. In: *Commun. ACM* 54.9 (2011), pp. 69–77. DOI: 10.1145/1995376.1995394 (cit. on p. 21).

[433] F. Mourato, M. P. dos Santos and F. P. Birra. 'Automatic level generation for platform videogames using genetic algorithms'. In: *Proceedings of ACE 2011*, p. 8. DOI: 10.1145/2071423.2071433 (cit. on p. 169).

[434] M. Mugnier. 'Ontological Query Answering with Existential Rules'. In: *Proceedings of RR 2011*, pp. 2–23. DOI: 10.1007/978-3-642-23580-1_2 (cit. on p. 17).

[435] C. Muise. 'Planning.Domains'. In: *ICAPS system demonstration* (2016) (cit. on pp. 242, 250).

[436] C. Muise, S. Vernhes and F. Pommerening. *Solver.Planning.Domains*. 2015–2017. URL: http://solver.planning.domains (visited on 25th Sept. 2017) (cit. on p. 216).

[437] M. R. Naphade, G. Banavar, C. Harrison, J. Paraszczak and R. Morris. 'Smarter Cities and Their Innovation Challenges'. In: *IEEE Computer* 44.6 (2011), pp. 32–39. DOI: 10.1109/MC.2011.187 (cit. on p. 109).

[438] A. Narayan-Chen, L. Xu and J. Shavlik. 'An Empirical Evaluation of Machine Learning Approaches for Angry Birds'. In: *IJCAI 2013 Symposium on AI in Angry Birds*. URL: http://aibirds.org/2013-Papers/Symposium/teamwisc.pdf (cit. on pp. 182, 183).

[439] A. Nareyek. 'AI in Computer Games'. In: *ACM Queue* 1.10 (2004), pp. 58–65. DOI: 10.1145/971564.971593 (cit. on p. 164).

[440] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu and J. Banerjee. 'RDFox: A Highly-Scalable RDF Store'. In: *[37]*. 2015, pp. 3–20. DOI: 10.1007/978-3-319-25010-6_1 (cit. on p. 20).

[441] T. N. Nguyen and W. Siberski. 'SLUBM: An Extended LUBM Benchmark for Stream Reasoning'. In: *Proceedings of OrdRing 2013*, pp. 43–54. URL: http://ceur-ws.org/Vol-1059/ordring2013-paper6.pdf (cit. on pp. 75, 79).

[442] D. Nicklas and Ö. L. Özçep, eds. *Proceedings of Workshop on High-Level Declarative Stream Processing co-located with KI 2015*. (Dresden, Germany). Vol. 1447. CEUR-WS. CEUR-WS.org, 22nd Sept. 2015. URL: http://ceur-ws.org/Vol-1447 (cit. on p. 275).

[443] I. Niemelä. 'Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm'. In: *AMAI* 25.3-4 (1999), pp. 241–273. DOI: 10.1023/A:1018930122475 (cit. on p. 39).

[444] R. Nieuwenhuis, A. Oliveras and C. Tinelli. 'Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)'. In: *J. ACM* 53.6 (2006), pp. 937–977. DOI: 10.1145/1217856.1217859 (cit. on p. 21).

[445] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson and M. Barry. 'An A Prolog decision support system for the Space Shuttle'. In: *Proceedings of 2001 AAAI Spring Symposium on ASP*. URL: http://www.cs.nmsu.edu/~tson/ASP2001/10.ps (cit. on p. 27).

[446] P. Novák. 'Behavioural State Machines: Programming Modular Agents'. In: *Proceedings of AITA 2008*, pp. 49–54. URL: http://www.aaai.org/Library/Symposia/Spring/2008/ss08-02-009.php (cit. on p. 171).

[447] P. Novák. 'Cognitive agents with non-monotonic reasoning'. In: *Proceedings of AAMAS 2008, Doctoral Mentoring Program*, pp. 1746–1747. DOI: 10.1145/1402782.1402794 (cit. on p. 170).

[448] P. Novák. 'Jazzyk: A Programming Language for Hybrid Agents with Heterogeneous Knowledge Representations'. In: *Proceedings of ProMAS 2008*, pp. 72–87. DOI: 10.1007/978-3-642-03278-3_5 (cit. on pp. 170, 171).

[449] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar and Y. Shoham. 'Understanding Random SAT: Beyond the Clauses-to-Variables Ratio'. In: *Proceedings of CP 2004*, pp. 438–452. DOI: 10.1007/978-3-540-30201-8_33 (cit. on p. 141).

[450] A. Nuradiansyah, E. Ziberi, S. Tirtarasa and L. Schweizer. 'SEABirds: An AHP Approach to Solve the Angry Birds AI Challenge'. 2016. URL: https://aibirds.org/2016-papers/SEABirdsReport.pdf (cit. on p. 182).

[451] R. A. O'Keefe. *The Craft of Prolog*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-15039-5 (cit. on p. 7).

[452] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent and B. O'Sullivan. 'Using case-based reasoning in an algorithm portfolio for constraint solving'. In: *Proceedings of AICS 2008*. Ed. by D. Bridge, K. Brown, B. O'Sullivan and H. Sorensen, pp. 210–216 (cit. on p. 146).

[453] E. Oikarinen and T. Janhunen. 'Modular Equivalence for Normal Logic Programs'. In: *Proceedings of ECAI 2006 and PAIS 2006*, pp. 412–416 (cit. on pp. 83, 88, 90, 161).

[454] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill and M. Preuss. 'A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft'. In: *TCIAIG* 5.4 (2013), pp. 293–311. DOI: 10.1109/TCIAIG.2013.2286295 (cit. on p. 209).

[455] J. Orkin. 'Agent Architecture Considerations for Real-Time Planning in Games'. In: *Proceedings of AIIDE 2005*, pp. 105–110 (cit. on p. 169).

[456] J. Orkin. 'Symbolic representation of game world state: Toward real-time planning in games'. In: *Proceedings of CGAI 2004*. Vol. 5, pp. 26–30 (cit. on p. 169).

[457] J. Orkin. 'Three states and a plan: the AI of FEAR'. In: *Proceedings of GDC 2006*. Vol. 2006, p. 4 (cit. on p. 169).

[458] J. Orkin. 'Applying goal-oriented action planning to games'. In: *AI Game Programming Wisdom* 2 (2003), pp. 217–228 (cit. on pp. 169, 211).

[459] G. Orsi and L. Tanca. 'Introduction to the TPLP special issue, logic programming in databases: From Datalog to semantic-web rules'. In: *TPLP* 10.3 (2010), pp. 243–250. DOI: 10.1017/S1471068410000086 (cit. on p. 15).

[460] L. Padovani and A. Provetti. 'Qsmodels: ASP Planning in Interactive Gaming Environment'. In: *[8]*. 2004, pp. 689–692. DOI: 10.1007/978-3-540-30227-8_58 (cit. on pp. 170, 203).

[461] A. D. Palù, A. Dovier, E. Pontelli and G. Rossi. 'GASP: Answer Set Programming with Lazy Grounding'. In: *Fundam. Inform.* 96.3 (2009), pp. 297–322. DOI: 10.3233/FI-2009-180 (cit. on p. 30).

[462] S. Paramonov, C. Bessiere, A. Dries and L. D. Raedt. 'Sketched Answer Set Programming'. In: *CoRR* abs/1705.07429 (2017). URL: http://arxiv.org/abs/1705.07429 (cit. on p. 23).

[463] P. F. Patel-Schneider and I. Horrocks. 'A comparison of two modelling paradigms in the Semantic Web'. In: *J. Web Sem.* 5.4 (2007), pp. 240–250. DOI: 10.1016/j.websem.2007.09.004 (cit. on p. 18).

[464] C. Pavanelli, M. Maresch, T. Calcagniti, F. Martino, D. Meneghetti, L. Ferreira and P. Santos. 'A new combination of Qualitative Spatial Representation and Utility Function for the FEI2 agent in the Angry Birds Domain'. 2013. URL: http://aibirds.org/2013-Papers/Team-Descriptions/fei2.pdf (cit. on p. 182).

[465] P. Peng, Q. Yuan, Y. Wen, Y. Yang, Z. Tang, H. Long and J. Wang. 'Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games'. In: *CoRR* abs/1703.10069 (2017). URL: http://arxiv.org/abs/1703.10069 (cit. on p. 165).

[466] S. Perri, F. Ricca and M. Sirianni. 'Parallel instantiation of ASP programs: techniques and experiments'. In: *TPLP* 13.2 (2013), pp. 253–278. DOI: 10.1017/S1471068411000652 (cit. on p. 148).

[467] T. Pham, S. Germano, A. Mileo, D. Kümper and M. I. Ali. 'Automatic configuration of smart city applications for user-centric decision support'. In: *Proceedings of ICIN 2017*, pp. 360–365. DOI: 10.1109/ICIN.2017.7899441 (cit. on pp. xii, 109).

[468] D. L. Phuoc, M. Dao-Tran, J. X. Parreira and M. Hauswirth. 'A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data'. In: *Proceedings of ISWC 2011*, pp. 370–388. DOI: 10.1007/978-3-642-25073-6_24 (cit. on pp. 72, 102, 104).

[469] D. L. Phuoc, M. Dao-Tran, M. Pham, P. A. Boncz, T. Eiter and M. Fink. 'Linked Stream Data Processing Engines: Facts and Figures'. In: *Proceedings of ISWC 2012, Part II*, pp. 300–312. DOI: 10.1007/978-3-642-35173-0_20 (cit. on pp. 75, 77).

[470] M. Polceanu. 'ORPHEUS: Reasoning and Prediction with Heterogeneous rEpresentations Using Simulation. (ORPHEUS: raisonnement et prédiction avec des représentations hétérogènes en utilisant la simulation)'. PhD thesis. University of Western Brittany, Brest, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01203388 (cit. on p. 181).

[471] M. Polceanu and C. Buche. 'Towards A Theory-Of-Mind-Inspired Generic Decision-Making Framework'. In: *CoRR* abs/1405.5048 (2014). URL: http://arxiv.org/abs/1405.5048 (cit. on pp. 180, 181).

[472] H. J. S. Pollman. 'Probabilistic cost analysis of logic programs'. In: *INGENIARE* 17.2 (2009) (cit. on p. 156).

[473] V. Poosala and Y. E. Ioannidis. 'Selectivity Estimation Without the Attribute Value Independence Assumption'. In: *Proceedings of VLDB 1997*, pp. 486–495. URL: http://www.vldb.org/conf/1997/P486.PDF (cit. on pp. 149, 151).

[474] V. Poosala, Y. E. Ioannidis, P. J. Haas and E. J. Shekita. 'Improved Histograms for Selectivity Estimation of Range Predicates'. In: *[309]*. 1996, pp. 294–305. DOI: 10.1145/233269.233342 (cit. on pp. 148, 149, 151, 152).

[475] L. Popa, S. Abiteboul and P. G. Kolaitis, eds. *Proceedings of PODS 2002*. (Madison, Wisconsin, USA). ACM, 3rd–5th June 2002. ISBN: 1-58113-507-6. URL: http://dl.acm.org/citation.cfm?id=543613 (cit. on pp. 267, 286).

[476] U. of Potsdam. *Potassco, the Potsdam Answer Set Solving Collection*. 2017. URL: https://potassco.org (visited on 25th Sept. 2017) (cit. on p. 220).

[477] *Proceedings of CIG 2017*. (New York, NY, USA). IEEE, 22nd–25th Aug. 2017. ISBN: 978-1-5386-3233-8. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8067294 (cit. on pp. 283, 297).

[478] T. C. Przymusinski. 'On the Declarative and Procedural Semantics of Logic Programs'. In: *J. Autom. Reasoning* 5.2 (1989), pp. 167–205. DOI: 10.1007/BF00243002 (cit. on p. 14).

[479] T. C. Przymusinski. 'Stable Semantics for Disjunctive Programs'. In: *New Generation Comput.* 9.3/4 (1991), pp. 401–424. DOI: 10.1007/BF03037171 (cit. on p. 14).

[480] D. Puiu, P. M. Barnaghi, R. Toenjes, D. Kuemper, M. I. Ali, A. Mileo, J. X. Parreira, M. Fischer, S. Kolozali, N. FarajiDavar, F. Gao, T. Iggena, T. Pham, C. Nechifor, D. Puschmann and J. Fernandes. 'CityPulse: Large Scale Data Analytics Framework for Smart Cities'. In: *IEEE Access* 4 (2016), pp. 1086–1108. DOI: 10.1109/ACCESS.2016.2541999 (cit. on pp. 97, 101, 109, 110, 118).

[481] L. Pulina and A. Tacchella. 'A Multi-engine Solver for Quantified Boolean Formulas'. In: *Proceedings of CP 2007*, pp. 574–589. DOI: 10.1007/978-3-540-74970-7_41 (cit. on p. 146).

[482] S. Rabin. *AI Game Programming Wisdom*. Rockland, MA, USA: Charles River Media, Inc., 2002. ISBN: 1584500778 (cit. on p. 164).

[483] S. Rabin, ed. *AI Game Programming Wisdom, Vol. 2*. Rockland, MA, USA: Charles River Media, Inc., 2003. ISBN: 1584502894 (cit. on p. 164).

[484] S. Rabin. *AI Game Programming Wisdom 3 (Game Development Series)*. Rockland, MA, USA: Charles River Media, Inc., 2006. ISBN: 1584504579 (cit. on p. 164).

[485] S. P. Radziszowski. 'Small ramsey numbers'. In: *Electron. J. Combin, Dynamic Surveys* (1994) (cit. on p. 29).

[486] R. Ramakrishnan, W. G. Roth, P. Seshadri, D. Srivastava and S. Sudarshan. 'The CORAL Deductive Database System'. In: *Proceedings of ACM SIGMOD 1993*, pp. 544–545. DOI: 10.1145/170035.171550 (cit. on p. 19).

[487] J. Rath and C. Redl. 'Integrating Answer Set Programming with Object-Oriented Languages'. In: *Proceedings of PADL 2017*, pp. 50–67. DOI: 10.1007/978-3-319-51676-9_4 (cit. on p. 236).

[488] J. Reades, F. Calabrese, A. Sevtsuk and C. Ratti. 'Cellular Census: Explorations in Urban Data Collection'. In: *IEEE Pervasive Comput.* 6.3 (2007), pp. 30–38. DOI: 10.1109/MPRV.2007.53 (cit. on p. 51).

[489] C. Redl. 'The dlvhex system for knowledge representation: recent advances (system description)'. In: *TPLP* 16.5-6 (2016), pp. 866–883. DOI: 10.1017/S1471068416000211 (cit. on p. 37).

[490] R. Reiter. 'A Logic for Default Reasoning'. In: *[282]*. 1987, pp. 68–93. URL: http://dl.acm.org/citation.cfm?id=42641.42646 (cit. on p. 21).

[491] R. Reiter. 'On Closed World Data Bases'. In: *[282]*. 1987, pp. 300–310. URL: http://dl.acm.org/citation.cfm?id=42641.42663 (cit. on pp. 13, 27, 56).

[492] X. Ren, O. Curé, L. Ke, J. Lhez, B. Belabbess, T. Randriamalala, Y. Zheng and G. Képéklian. 'Strider: An Adaptive, Inference-enabled Distributed RDF Stream Processing Engine'. In: *PVLDB* 10.12 (2017), pp. 1905–1908. URL: http://www.vldb.org/pvldb/vol10/p1905-ren.pdf (cit. on p. 74).

[493] Y. Ren and J. Z. Pan. 'Optimising ontology stream reasoning with truth maintenance system'. In: *[394]*. 2011, pp. 831–836. DOI: 10.1145/2063576.2063696 (cit. on p. 73).

[494] J. Renz. 'AIBIRDS: The Angry Birds Artificial Intelligence Competition'. In: *[93]*. 2015, pp. 4326–4327. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/10050 (cit. on p. 173).

[495] J. Renz and X. Ge. 'Physics Simulation Games'. In: *Handbook of Digital Games and Entertainment Technologies*, pp. 1–19. DOI: 10.1007/978-981-4560-52-8_29-1 (cit. on p. 173).

[496] J. Renz, X. Ge, S. Gould and P. Zhang. 'The Angry Birds AI Competition'. In: *AI Magazine* 36.2 (2015), pp. 85–87. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2588 (cit. on p. 173).

[497] J. Renz, X. Ge, R. Verma and P. Zhang. 'Angry Birds as a Challenge for Artificial Intelligence'. In: *[518]*. 2016, pp. 4338–4339. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12527 (cit. on p. 173).

[498] J. Renz, R. Miikkulainen, N. R. Sturtevant and M. H. M. Winands. 'Guest Editorial: Physics-Based Simulation Games'. In: *TCIAIG* 8.2 (2016), pp. 101–103. DOI: 10.1109/TCIAIG.2016.2571560 (cit. on p. 173).

[499] A. E. Rhalibi, K. W. Wong and M. Price. 'Artificial Intelligence for Computer Games'. In: *Int. J. Computer Games Technology* 2009 (2009), 251652:1–251652:3. DOI: 10.1155/2009/251652 (cit. on p. 164).

[500] F. Ricca. 'The DLV Java Wrapper'. In: *Proceedings of AGP 2003*, pp. 263–274 (cit. on p. 235).

[501] F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano and N. Leone. 'Team-building with answer set programming in the Gioia-Tauro seaport'. In: *TPLP* 12.3 (2012), pp. 361–381. DOI: 10.1017/S147106841100007X (cit. on p. 214).

[502] J. R. Rice. 'The Algorithm Selection Problem'. In: *Advances in Computers* 15 (1976), pp. 65–118. DOI: 10.1016/S0065-2458(08)60520-3 (cit. on p. 146).

[503] E. Rich and K. Knight. *Artificial intelligence (2. ed.)* McGraw-Hill, 1991. ISBN: 978-0-07-052263-3 (cit. on p. 95).

[504] M. Rinne, H. Abdullah, S. Törmä and E. Nuutila. 'Processing Heterogeneous RDF Events with Standing SPARQL Update Rules'. In: *Proceedings of OTM 2012 Workshops*, pp. 797–806. DOI: 10.1007/978-3-642-33615-7_24 (cit. on p. 74).

[505] M. Rinne, E. Nuutila and S. Törmä. 'INSTANS: High-Performance Event Processing with Standard RDF and SPARQL'. In: *Proceedings of ISWC 2012 Posters & Demonstrations Track*. URL: http://ceur-ws.org/Vol-914/paper_22.pdf (cit. on p. 74).

[506] M. Rinne, S. Törmä and E. Nuutila. 'SPARQL-Based Applications for RDF-Encoded Sensor Data'. In: *Proceedings of SSN 2012*, pp. 81–96. URL: http://ceur-ws.org/Vol-904/paper15.pdf (cit. on p. 74).

[507] J. S. Rosenschein and M. R. Genesereth. 'Deals Among Rational Agents'. In: *Proceedings of IJCAI 1985*, pp. 91–99. URL: http://ijcai.org/Proceedings/85-1/Papers/017.pdf (cit. on p. 166).

[508] K. A. Ross. 'Modular Stratification and Magic Sets for Datalog Programs with Negation'. In: *J. ACM* 41.6 (1994), pp. 1216–1266. DOI: 10.1145/195613.195646 (cit. on p. 14).

[509] F. Rossi, ed. *Proceedings of IJCAI 2013*. (Beijing, China). IJCAI/AAAI, 3rd–9th Aug. 2013. ISBN: 978-1-57735-633-2. URL: http://ijcai.org/proceedings/2013 (cit. on pp. 280, 281).

[510] D. Saccà and C. Zaniolo. 'Stable Models and Non-Determinism in Logic Programs with Negation'. In: *Proceedings of PODS 1990*, pp. 205–217. DOI: 10.1145/298514.298572 (cit. on p. 14).

[511] K. Sagonas, T. Swift and D. S. Warren. 'XSB as an Efficient Deductive Database Engine'. In: *Proceedings of ACM SIGMOD 1994*, pp. 442–453. DOI: 10.1145/191839.191927 (cit. on p. 19).

[512] H. Samulowitz and R. Memisevic. 'Learning to Solve QBF'. In: *Proceedings of AAAI 2007*, pp. 255–260. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-039.php (cit. on p. 21).

[513] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu and S. Sutphen. 'Checkers Is Solved'. In: *Science* 317.5844 (2007), pp. 1518–1522. ISSN: 0036-8075. DOI: 10.1126/science.1144079. eprint: http://science.sciencemag.org/content/317/5844/1518.full.pdf (cit. on p. 208).

[514] T. Schaub, G. Friedrich and B. O'Sullivan, eds. *Proceedings of ECAI 2014 and PAIS 2014*. (Prague, Czech Republic). Vol. 263. FAIA. IOS Press, 18th–22nd Aug. 2014. ISBN: 978-1-61499-418-3 (cit. on pp. 271, 298).

[515] S. Schiffer, M. Jourenko and G. Lakemeyer. 'AKBABA: The KBSG 2013 Team for the Angry Birds AI Competition'. 2013. URL: http://aibirds.org/2013-Papers/Team-Descriptions/Akbaba.pdf (cit. on p. 182).

[516] S. Schiffer, M. Jourenko and G. Lakemeyer. 'Akbaba - An Agent for the Angry Birds AI Challenge Based on Search and Simulation'. In: *TCIAIG* 8.2 (2016), pp. 116–127. DOI: 10.1109/TCIAIG.2015.2478703 (cit. on p. 182).

[517] U. Schoning. *Logic for Computer Scientists*. Ed. by R. Constable, J. C. Cherniavsky, R. Platek, J. Gallier and R. Statman. 1st. Birkhauser Boston, 1989. ISBN: 0817634533 (cit. on p. 8).

[518] D. Schuurmans and M. P. Wellman, eds. *Proceedings of AAAI 2016*. (Phoenix, Arizona, USA). AAAI Press, 12th–17th Feb. 2016. ISBN: 978-1-57735-760-5. URL: http://www.aaai.org/Library/AAAI/aaai16contents.php (cit. on p. 294).

[519] B. Schwab. *Ai Game Engine Programming (Game Development Series)*. Rockland, MA, USA: Charles River Media, Inc., 2004. ISBN: 1584503440 (cit. on p. 164).

[520] J. Seipp, M. Braun, J. Garimort and M. Helmert. 'Learning Portfolios of Automatically Tuned Planners'. In: *Proceedings of ICAPS 2012*. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4729 (cit. on p. 146).

[521] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. 'Access Path Selection in a Relational Database Management System'. In: *Proceedings of ACM SIGMOD 1979*, pp. 23–34. DOI: 10.1145/582095.582099 (cit. on p. 161).

[522] J. F. Sequeda and Ó. Corcho. 'Linked Stream Data: A Position Paper'. In: *Proceedings of ISWC 2010*, pp. 148–157. URL: http://ceur-ws.org/Vol-522/p13.pdf (cit. on p. 72).

[523] Y. Shoham. 'Agent-Oriented Programming'. In: *Artif. Intell.* 60.1 (1993), pp. 51–92. DOI: 10.1016/0004-3702(93)90034-9 (cit. on p. 166).

[524] C. Sierra, ed. *Proceedings of IJCAI 2017*. (Melbourne, Australia). ijcai.org, 19th–25th Aug. 2017. ISBN: 978-0-9992411-0-3. URL: http://www.ijcai.org/Proceedings/2017/ (cit. on pp. 270, 277).

[525] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis. 'Mastering the game of Go with deep neural networks and tree search'. In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961 (cit. on p. 165).

[526] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel and D. Hassabis. 'Mastering the game of Go without human knowledge'. In: *Nature* 550.7676 (19th Oct. 2017), pp. 354–359. ISSN: 0028-0836. URL: http://dx.doi.org/10.1038/nature24270 (cit. on p. 165).

[527] B. Silverthorn, Y. Lierler and M. Schneider. 'Surviving Solver Sensitivity: An ASP Practitioner's Guide'. In: *Proceedings of ICLP 2012*, pp. 164–175. DOI: 10.4230/LIPIcs.ICLP.2012.164 (cit. on p. 146).

[528] P. Simons, I. Niemelä and T. Soininen. 'Extending and implementing the stable model semantics'. In: *Artif. Intell.* 138.1-2 (2002), pp. 181–234. DOI: 10.1016/S0004-3702(02)00187-X (cit. on p. 30).

[529] E. Siow, T. Tiropanis and W. Hall. 'Ewya: An Interoperable Fog Computing Infrastructure with RDF Stream Processing'. In: *Proceedings of INSCI 2017*, pp. 245–265. DOI: 10.1007/978-3-319-70284-1_20 (cit. on p. 74).

[530] A. M. Smith. *Answer Set Programming in Proofdoku*. 2017 (cit. on pp. 170, 171).

[531] A. M. Smith, E. Butler and Z. Popovic. 'Quantifying over play: Constraining undesirable solutions in puzzle design'. In: *Proceedings of FDG 2013*, pp. 221–228. URL: http://www.fdg2013.org/program/papers/paper29_smith_etal.pdf (cit. on p. 169).

[532] A. M. Smith and M. Mateas. 'Answer Set Programming for Procedural Content Generation: A Design Space Approach'. In: *TCIAIG* 3.3 (2011), pp. 187–200. DOI: 10.1109/TCIAIG.2011.2158545 (cit. on p. 169).

[533] A. M. Smith, M. J. Nelson and M. Mateas. 'LUDOCORE: A logical game engine for modeling videogames'. In: *Proceedings of CIG 2010*, pp. 91–98. DOI: 10.1109/ITW.2010.5593368 (cit. on p. 169).

[534] G. Smith, J. Whitehead and M. Mateas. 'Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design'. In: *TCIAIG* 3.3 (2011), pp. 201–215. DOI: 10.1109/TCIAIG.2011.2159716 (cit. on p. 169).

[535] P. Spanily. *The Inter4QL Interpreter*. 2012 (cit. on p. 19).

[536] J. Spiegel and N. Polyzotis. 'Graph-based synopses for relational selectivity estimation'. In: *Proceedings of ACM SIGMOD 2006*, pp. 205–216. DOI: 10.1145/1142473.1142497 (cit. on p. 149).

[537] B. Srivastava, J. P. Bigus and D. A. Schlosnagle. 'Bringing Planning to Autonomic Applications with ABLE'. In: *Proceedings of ICAC 2004*, pp. 154–161. DOI: 10.1109/ICAC.2004.23 (cit. on p. 237).

[538] I. Staff, ed. *Proceedings of CIG 2014*. (Dortmund, Germany). IEEE, 26th–29th Aug. 2014. ISBN: 978-1-4799-3546-8. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6919811 (cit. on pp. 278, 297).

[539] M. Stanescu and M. Certický. 'Predicting Opponent's Production in Real-Time Strategy Games With Answer Set Programming'. In: *TCIAIG* 8.1 (2016), pp. 89–94. DOI: 10.1109/TCIAIG.2014.2365414 (cit. on p. 208).

[540] M. Stephenson and J. Renz. 'Procedural generation of complex stable structures for angry birds levels'. In: *Proceedings of CIG 2016*, pp. 1–8. DOI: 10.1109/CIG.2016.7860410 (cit. on p. 182).

[541] M. Stephenson and J. Renz. 'Procedural Generation of Levels for Angry Birds Style Physics Games'. In: *Proceedings of AIIDE 2016*, pp. 225–231. URL: http://aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/view/13983 (cit. on p. 182).

[542] M. Stephenson and J. Renz. 'Generating varied, stable and solvable levels for angry birds style physics games'. In: *[477]*. 2017, pp. 288–295. DOI: 10.1109/CIG.2017.8080448 (cit. on p. 258).

[543] M. Stephenson, J. Renz and X. Ge. 'The Computational Complexity of Angry Birds and Similar Physics-Simulation Games'. In: *Proceedings of AIIDE 2017*, pp. 241–247. URL: https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15829 (cit. on p. 182).

[544] L. Sterling and E. Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994 (cit. on p. 7).

[545] M. E. Stickel. 'A Unification Algorithm for Associative-Commutative Functions'. In: *J. ACM* 28.3 (1981), pp. 423–434. DOI: 10.1145/322261.322262 (cit. on p. 15).

[546] M. Stillger, G. M. Lohman, V. Markl and M. Kandil. 'LEO - DB2's LEarning Optimizer'. In: *[27]*. 2001, pp. 19–28. URL: http://www.vldb.org/conf/2001/P019.pdf (cit. on p. 149).

[547] M. Stonebraker. 'Stream Processing'. In: *[384]*. 2009, pp. 2837–2838. DOI: 10.1007/978-0-387-39940-9_371 (cit. on p. 81).

[548] M. Stonebraker, U. Çetintemel and S. B. Zdonik. 'The 8 requirements of real-time stream processing'. In: *SIGMOD Record* 34.4 (2005), pp. 42–47. DOI: 10.1145/1107499.1107504 (cit. on pp. 50, 63, 64).

[549] I. StreamBase. *Streambase: Real-time, low latency data processing with a stream processing engine*. 2006 (cit. on p. 62).

[550] V. Strobel and A. Kirsch. 'Planning in the Wild: Modeling Tools for PDDL'. In: *Proceedings of KI 2014*, pp. 273–284 (cit. on p. 250).

[551] H. Stuckenschmidt, S. Ceri, E. D. Valle and F. van Harmelen. 'Towards Expressive Stream Reasoning'. In: *Semantic Challenges in Sensor Networks*. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2555/ (cit. on pp. 48, 66).

[552] R. Stühmer, Y. Verginadis, I. Alshabani, T. Morsellino and A. Aversa. 'PLAY: Semantics-Based Event Marketplace'. In: *Proceedings of IFIP WG 5.5, PRO-VE 2013*. Springer, pp. 699–707. DOI: 10.1007/978-3-642-40543-3_73 (cit. on p. 109).

[553] A. Sureshkumar, M. D. Vos, M. Brain and J. Fitch. 'Ape: An ansprolog* environment'. In: *Proceedings of SEA 2007*, pp. 101–115 (cit. on p. 250).

[554] A. N. Swami and K. B. Schiefer. 'On the Estimation of Join Result Sizes'. In: *Proceedings of EDBT 1994*, pp. 287–300. DOI: 10.1007/3-540-57818-8_58 (cit. on pp. 149, 161).

[555] M. Swiechowski and J. Mandziuk. 'Prolog versus specialized logic inference engine in General Game Playing'. In: *[538]*. 2014, pp. 1–8. DOI: 10.1109/CIG.2014.6932864 (cit. on p. 169).

[556] T. Syrjänen. *Lparse*. 1999–2017. URL: http://www.tcs.hut.fi/Software/smodels (visited on 25th Sept. 2017) (cit. on p. 33).

[557] G. Terracina, N. Leone, V. Lio and C. Panetta. 'Experimenting with recursive queries in database and logic programming systems'. In: *TPLP* 8.2 (2008), pp. 129–165. DOI: 10.1017/S1471068407003158 (cit. on p. 32).

[558] G. Tesauro, D. Gondek, J. Lenchner, J. Fan and J. M. Prager. 'Analysis of Watson's Strategies for Playing Jeopardy!' In: *JAIR* 47 (2013), pp. 205–251. DOI: 10.1613/jair.3834 (cit. on p. 165).

[559] M. Thimm. 'Tweety: A Comprehensive Collection of Java Libraries for Logical Aspects of Artificial Intelligence and Knowledge Representation'. In: *[57]*. 2014. URL: http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7811 (cit. on p. 237).

[560] M. Thimm. 'The Tweety Library Collection for Logical Aspects of Artificial Intelligence and Knowledge Representation'. In: *KI* 31.1 (2017), pp. 93–97. DOI: 10.1007/s13218-016-0458-4 (cit. on p. 237).

[561] J. Tiihonen, T. Soininen, I. Niemelä and R. Sulonen. 'A Practical Tool For Mass-Customising Configurable Products'. In: *Proceedings of ICED 2003*. Stockholm, pp. 1290–1299 (cit. on p. 27).

[562] J. Togelius, G. N. Yannakakis, K. O. Stanley and C. Browne. 'Search-Based Procedural Content Generation: A Taxonomy and Survey'. In: *TCIAIG* 3.3 (2011), pp. 172–186. DOI: 10.1109/TCIAIG.2011.2148116 (cit. on pp. 164, 169).

[563] M. Truszczynski. 'Connecting First-Order ASP and the Logic FO(ID) through Reducts'. In: *[208]*. 2012, pp. 543–559. DOI: 10.1007/978-3-642-30743-0_37 (cit. on p. 23).

[564] S. Tsur and C. Zaniolo. 'LDL: A Logic-Based Data Language'. In: *Proceedings of VLDB 1986*, pp. 33–41. URL: http://www.vldb.org/conf/1986/P033.PDF (cit. on p. 15).

[565] N. Tziortziotis, G. Papagiannis and K. Blekas. 'A Bayesian Ensemble Regression Framework on the Angry Birds Game'. 2014. URL: https://aibirds.org/2014-papers/AngryBER-2014.pdf (cit. on p. 182).

[566] N. Tziortziotis, G. Papagiannis and K. Blekas. 'A Bayesian Ensemble Regression Framework on the Angry Birds Game'. In: *TCIAIG* 8.2 (2016), pp. 104–115. DOI: 10.1109/TCIAIG.2015.2494679 (cit. on pp. 182, 183).

[567] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989. ISBN: 0-7167-8162-X (cit. on pp. 9, 148).

[568] A. N. University. *Angry Birds AI Competition*. 2012–2017. URL: http://aibirds.org (visited on 25th Sept. 2017) (cit. on p. 173).

[569] E. D. Valle, S. Ceri, D. F. Barbieri, D. Braga and A. Campi. 'A First Step Towards Stream Reasoning'. In: *Proceedings of FIS 2008*, pp. 72–81. DOI: 10.1007/978-3-642-00985-3_6 (cit. on pp. 48, 50, 55, 66, 67, 69–71).

[570] E. D. Valle, S. Ceri, F. van Harmelen and D. Fensel. 'It's a Streaming World! Reasoning upon Rapidly Changing Information'. In: *IEEE Intell. Syst.* 24.6 (2009), pp. 83–89. DOI: 10.1109/MIS.2009.125 (cit. on pp. 50, 66–68, 82).

[571] R. D. Virgilio, G. Orsi, L. Tanca and R. Torlone. 'NYAYA: A System Supporting the Uniform Management of Large Sets of Semantic Data'. In: *Proceedings of ICDE 2012*, pp. 1309–1312. DOI: 10.1109/ICDE.2012.133 (cit. on p. 20).

[572] J. Vittaut and J. Méhat. 'Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling'. In: *[514]*. 2014, pp. 1121–1122. DOI: 10.3233/978-1-61499-419-0-1121 (cit. on p. 169).

[573] R. Volz, S. Staab and B. Motik. 'Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases'. In: *JoDS* 2 (2005), pp. 1–34. DOI: 10.1007/978-3-540-30567-5_1 (cit. on p. 71).

[574] O. Walavalkar, A. Joshi, T. Finin and Y. Yesha. 'Streaming knowledge bases'. In: *Proceedings of SSWS 2008* (cit. on p. 73).

[575] P. A. Walega, M. Zawidzki and T. Lechowski. 'Qualitative Physics in Angry Birds'. In: *TCIAIG* 8.2 (2016), pp. 152–165. DOI: 10.1109/TCIAIG.2016.2561080 (cit. on pp. 180–182).

[576] T. Walsh, ed. *Proceedings of IJCAI 2011*. (Barcelona, Catalonia, Spain). IJCAI/AAAI, 16th–22nd July 2011. ISBN: 978-1-57735-516-8. URL: http://ijcai.org/proceedings/2011 (cit. on pp. 267, 285).

[577] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-37940-6 (cit. on p. 220).

[578] B. G. Weber, M. Mateas and A. Jhala. 'Applying Goal-Driven Autonomy to StarCraft'. In: *Proceedings of AIIDE 2010*. URL: http://aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2142 (cit. on p. 169).

[579] J. Whaley, D. Avots, M. Carbin and M. S. Lam. 'Using Datalog with Binary Decision Diagrams for Program Analysis'. In: *Proceedings of APLAS 2005*, pp. 97–118. DOI: 10.1007/11575467_8 (cit. on p. 18).

[580] J. E. White. *Telescript technology: The foundation for the electronic marketplace*. General Magic Inc., 1994 (cit. on p. 166).

[581] J. Wielemaker, T. Lager and F. Riguzzi. 'SWISH: SWI-Prolog for Sharing'. In: *CoRR* abs/1511.00915 (2015) (cit. on pp. 242, 250).

[582] J. Wielemaker, T. Schrijvers, M. Triska and T. Lager. 'SWI-Prolog'. In: *TPLP* 12.1-2 (2012), pp. 67–96. DOI: 10.1017/S1471068411000494 (cit. on p. 250).

[583] K.-B. S. G. .-.-. T. Wien. *dlvhex*. 2010–2017. URL: http://www.kr.tuwien.ac.at/research/systems/dlvhex (visited on 25th Sept. 2017) (cit. on p. 37).

[584] K.-B. S. G. .-.-. T. Wien. *The Action Plugin and Action Addon Framework*. 2010–2017. URL: http://www.kr.tuwien.ac.at/research/systems/dlvhex/actionplugin.html (visited on 25th Sept. 2017) (cit. on p. 39).

[585] K.-B. S. G. .-.-. T. Wien. *dlvhex - Software for HEX-Programs on GitHub*. 2012–2017. URL: https://github.com/hexhex (visited on 25th Sept. 2017) (cit. on p. 37).

[586] M. Wooldridge and N. R. Jennings. 'Intelligent agents: theory and practice'. In: *KER* 10.2 (1995), pp. 115–152. DOI: 10.1017/S0269888900008122 (cit. on pp. 164–166).

[587] L. Xu, H. Hoos and K. Leyton-Brown. 'Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection'. In: *Proceedings of AAAI 2010*. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1929 (cit. on p. 146).

[588] L. Xu, F. Hutter, H. H. Hoos and K. Leyton-Brown. 'SATzilla: Portfolio-based Algorithm Selection for SAT'. In: *JAIR* 32 (2008), pp. 565–606. DOI: 10.1613/jair.2490 (cit. on p. 146).

[589] L. Xu, F. Hutter, H. Hoos and K. Leyton-Brown. 'Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors'. In: *Proceedings of SAT 2012*, pp. 228–241. DOI: 10.1007/978-3-642-31612-8_18 (cit. on p. 146).

[590] A. H. Yahya and L. J. Henschen. 'Deduction in Non-Horn Databases'. In: *J. Autom. Reasoning* 1.2 (1985), pp. 141–160. DOI: 10.1007/BF00244994 (cit. on p. 14).

[591] G. N. Yannakakis and J. Togelius. 'A Panorama of Artificial and Computational Intelligence in Games'. In: *TCIAIG* 7.4 (2015), pp. 317–335. DOI: 10.1109/TCIAIG.2014.2339221 (cit. on p. 164).

[592] G. N. Yannakakis and J. Togelius. *Artificial Intelligence and Games*. http://gameaibook.org. Springer, 2017 (cit. on pp. 164, 167, 169).

[593] H. L. Younes and M. L. Littman. *PPDDL 1.0: An extension to PDDL for expressing planning domains with probabilistic effects*. Techn. Rep. CMU-CS-04-162, 2004 (cit. on p. 40).

[594] Y. Yuan, Z. Chen, P. Wu and L. Chang. 'Enhancing Deep Reinforcement Learning Agent for Angry Birds'. 2017. URL: https://aibirds.org/2017/aibirds_BNU.pdf (cit. on p. 182).

[595] C. Zaniolo. ‘Logical Foundations of Continuous Query Languages for Data Streams’. In: *[66]*. 2012, pp. 177–189. DOI: 10.1007/978-3-642-32925-8_18 (cit. on pp. 55–57, 59).

[596] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian and R. Zicari. *Advanced Database Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-443-X (cit. on p. 58).

[597] P. Zhang and J. Renz. ‘Qualitative spatial representation and reasoning in Angry Birds: First results’. In: *Proceedings of QR 2013*, p. 123 (cit. on p. 182).

[598] P. Zhang and J. Renz. ‘Qualitative Spatial Representation and Reasoning in Angry Birds: The Extended Rectangle Algebra’. In: *[57]*. 2014. URL: http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/8021 (cit. on pp. 182, 183).

[599] Y. Zhang, M. Pham, Ó. Corcho and J. Calbimonte. ‘SRBench: A Streaming RDF/SPARQL Benchmark’. In: *Proceedings of ISWC 2012, Part I*, pp. 641–657. DOI: 10.1007/978-3-642-35176-1_40 (cit. on pp. 75, 77).

[600] S. Ziller, M. Gebser, B. Kaufmann and T. Schaub. *An Introduction to claspfolio*. Institute of Computer Science, University of Potsdam, Germany, 2010. URL: http://www.cs.uni-potsdam.de/claspfolio/manual.pdf (cit. on p. 146).

[601] D. Zimmer and R. Unland. ‘On the Semantics of Complex Events in Active Database Management Systems’. In: *Proceedings of ICDE 1999*, pp. 392–399. DOI: 10.1109/ICDE.1999.754955 (cit. on p. 55).

Bibliography Abbreviations

AAAI AAAI Conference on Artificial Intelligence.

AAMAS International Joint Conference on Autonomous Agents and Multiagent Systems.

ACE International Conference on Advances in Computer Entertainment Technology.

ACM Comput. Surv. ACM Computing Surveys.

ACM SIGMOD International Conference on Management of Data.

ACM-ICPS ACM International Conference Proceeding Series.

AGP Joint Conference on Declarative Programming.

AI Commun. AI Communications.

*AI*IA* International Conference of the Italian Association for Artificial Intelligence.

AICS Irish Conference on Artificial Intelligence and Cognitive Science.

AIG Workshop on AI in Games.

AIIDE AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.

AITA AAAI Spring Symposium on Architectures for Intelligent Theory-Based Agents.

ALP Association of Logic Programming.

AMAI Annals of Mathematics and Artificial Intelligence.

AMW Alberto Mendelzon International Workshop on Foundations of Data Management.

APLAS Asian Symposium on Programming Languages and Systems.

ARea International Workshop on Advancing Reasoning on the Web.

ASEE American Society for Engineering Education.

ASPOCP Workshop on Answer Set Programming and Other Computing Paradigms.

ATAL Workshop on Agent Theories, Architectures, and Languages.

AlgoConf AAAI Workshop on Algorithm Configuration.

Artif. Intell. Artificial Intelligence.

BeRSys International Workshop On Benchmarking RDF Systems.

CADE Conference on Automated Deduction.

CAV International Conference on Computer Aided Verification.

CEUR-WS CEUR Workshop Proceedings.

CGAI AAAI Workshop on Challenges in Game Artificial Intelligence.

CIDR Conference on Innovative Data Systems Research.

CIG IEEE Conference on Computational Intelligence and Games.

CP International Conference on Principles and Practice of Constraint Programming.

CSUR ACM Computing Surveys.

CoNEXT ACM Conference on emerging Networking EXperiments and Technologies.

CoRR *Computing Research Repository.*

CoSECivi *Congreso de la Sociedad Española para las Ciencias del Videojuego.*

Commun. ACM *Communications of the ACM.*

DBPL *International Workshop on Database Programming Languages.*

DCSA *Data-Centric Systems and Applications.*

DEBS *ACM International Conference on Distributed Event-Based Systems.*

DSP *Dagstuhl Seminar Proceedings.*

Data Sci. *Data Science.*

ECAI *European Conference on Artificial Intelligence.*

ECML PKDD *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases.*

EDBT *International Conference on Extending Database Technology.*

EKAW *International Conference on Knowledge Engineering and Knowledge Management.*

ESWC *European Semantic Web Conference.*

FAIA *Frontiers in Artificial Intelligence and Applications.*

FDG *International Conference on the Foundations of Digital Games.*

FIA *Future Internet Assembly.*

FIS *Future Internet Symposium.*

GDC *Game Developers Conference.*

GTTV *Workshop on Grounding and Transformations for Theories With Variables.*

GULP-PRODE *Joint Conference on Declarative Programming.*

HOPL *History of Programming Languages Conference.*

ICAC *International Conference on Autonomic Computing.*

ICAPS *International Conference on Automated Planning and Scheduling.*

ICC *IEEE International Conference on Communications.*

*ICCI*CC* *IEEE International Conference on Cognitive Informatics and Cognitive Computing.*

ICDE *IEEE International Conference on Data Engineering.*

ICDT *International Conference on Database Theory.*

ICED *International Conference on Engineering Design.*

ICIN *Conference on Innovations in Clouds, Internet and Networks.*

ICLP *International Conference on Logic Programming.*

ICSC *IEEE International Conference on Semantic Computing.*

ICSOC *International Conference on Service-Oriented Computing.*

ICTAI *IEEE International Conference on Tools with Artificial Intelligence.*

ICWE *International Conference on Web Engineering.*

IEEE Intell. Syst. *IEEE Intelligent Systems.*

IEEE Internet Comput. *IEEE Internet Computing.*

IEEE Pervasive Comput. *IEEE Pervasive Computing.*

IFIP *International Federation for Information Processing.*

IFIP AICT *IFIP Advances in Information and Communication Technology.*

IJAC *International Journal of Automation and Computing.*

IJCAI *International Joint Conference on Artificial Intelligence.*

IJSWIS *International Journal on Semantic Web and Information Systems.*

IMMM	*International Conference on Advances in Information Mining and Management.*
INAP	*International Conference on Applications of Declarative Programming and Knowledge Management.*
INGENIARE	*Revista Chilena de Ingeniería.*
INSCI	*International Conference on Internet Science.*
ISWC	*International Semantic Web Conference.*
Inf. Comput.	*Information and Computation.*
J. ACM	*Journal of the ACM.*
J. Log. Comput.	*Journal of Logic and Computation.*
J. Log. Program.	*Journal of Logic Programming.*
J. Web Sem.	*Journal of Web Semantics.*
JAIR	*Journal of Artificial Intelligence Research.*
JCSS	*Journal of Computer and System Sciences.*
JELIA	*European Conference on Logics in Artificial Intelligence.*
JIST	*Joint International Semantic Technology Conference.*
JoDS	*Journal on Data Semantics.*
KER	*Knowledge Engineering Review.*
KI	*German Conference on AI.*
KI	*Künstliche Intelligenz.*
LICS	*IEEE Symposium on Logic in Computer Science.*
LION	*Learning and Intelligent OptimizatioN Conference.*
LNCS	*Lecture Notes in Computer Science.*
LPNMR	*Logic Programming and Nonmonotonic Reasoning.*
MAAMAW	*European Workshop on Modelling Autonomous Agents in a Multi-Agent World.*
MobiQuitous	*International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services.*
Nature	*Nature.*
NeFoRS	*International Workshop on New Forms of Reasoning for the Semantic Web.*
New Generation Comput.	*New Generation Computing.*
OTM	*On the Move to Meaningful Internet Systems.*
OrdRing	*International Workshop on Ordering and Reasoning.*
PACT	*International Conference on Parallel Architecture and Compilation Techniques.*
PADL	*International Symposium on Practical Aspects of Declarative Languages.*
PAI	*Workshop Popularize Artificial Intelligence.*
PAIS	*Prestigious Applications of Intelligent Systems.*
PLDI	*ACM SIGPLAN Conference on Programming Language Design and Implementation.*
PODS	*ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.*
POPL	*ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*
PPDP	*International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming.*
PRICAI	*Pacific Rim International Conference on Artificial Intelligence.*
PVLDB	*Proceedings of the VLDB Endowment.*
ProMAS	*International Workshop on PROgramming Multi-Agent Systems.*

Procedia Comput. Sci. *Procedia Computer Science.*

QR *International Workshop on Qualitative Reasoning.*

RCRA *Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion.*

RR *International Conference on Web Reasoning and Rule Systems.*

RW *Reasoning on the Semantic Web. International Summer School.*

RuleML *International Web Rule Symposium.*

SAC *ACM Symposium on Applied Computing.*

SAT *International Conference on Theory and Applications of Satisfiability Testing.*

SEA *Software Engineering for Answer Set Programming.*

SEBD *Italian Symposium on Advanced Database Systems.*

SIGIR *Special Interest Group on Information Retrieval.*

SIGIR *International ACM SIGIR Conference on Research and Development in Information Retrieval.*

SIGMOD Record *Special Interest Group on Management of Data Record.*

SR *International Workshop on Stream Reasoning.*

SSN *International Workshop on Semantic Sensor Networks.*

SSWS *International Workshop on Scalable Semantic Web Knowledge Base Systems.*

SWDB *Workshop on Semantic Web and Databases.*

SWJ *Semantic Web.*

TCIAIG *IEEE Transactions on Computational Intelligence and AI in Games.*

TKDE *IEEE Transactions on Knowledge and Data Engineering.*

TOCL *ACM Transactions on Computational Logic.*

TODS *ACM Transactions on Database Systems.*

TOPLAS *ACM Transactions on Programming Languages and Systems.*

TPLP *Theory and Practice of Logic Programming.*

Theor. Comput. Sci. *Theoretical Computer Science.*

VLDB *International Conference on Very Large Data Bases.*

VLDB J. *International Journal on Very Large Data Bases.*

WI *IEEE / WIC / ACM International Conference on Web Intelligence.*

WIMS *International Conference on Web Intelligence, Mining and Semantics.*

WLP *Workshop on Logic Programming.*

WWW *International Conference on World Wide Web.*

“It is the time you have wasted for your rose that makes your rose so important.”
– *Antoine de Saint-Exupéry, The Little Prince*