DIPARTIMENTO DI MATEMATICA E INFORMATICA

DOTTORATO DI RICERCA
XXXIII CICLO

SETTORE SCIENTIFICO DISCIPLINARE INF/01 - INFORMATICA

TESI DI DOTTORATO

REASONING IN HIGHLY DYNAMIC ENVIRONMENTS

FRANCESCO PACENZA

SUPERVISORE
Ch.mo Prof. Giovambattista Ianni

COORDINATORE
Ch.mo Prof. Gianluigi Greco

CO-SUPERVISORE
Dott.ssa Jessica Zangari

Novembre 2017 - Gennaio 2021

CONTENTS

1 INTRODUCTION 1

I CONTEXT AND FOUNDATIONS

2 ANSWER SET PROGRAMMING 8

2.1 Syntax . . . 9

2.1.1 Terms . . . 9

2.1.2 Atoms and Literals . . . 10

2.1.3 Rules, Constraints, Queries and Programs . . . 11

2.2 Semantics . . . 16

2.2.1 Theoretical Instantiation . . . 16

2.2.2 Interpretations . . . 18

2.2.3 Answer Sets . . . 20

2.3 Advanced Constructs . . . 23

2.4 Safety Restrictions . . . 24

2.5 Modelling with Answer Set Programming . . . 27

2.6 Knowledge Representation and Reasoning via ASP . . . 28

2.6.1 Guess, Check and Optimize Paradigm . . . 28

2.6.2 A Real World Domain: Hydraulic Leaking . . . 33

3 THE STREAM REASONING WORLD: AN OVERVIEW 38

3.1 The importance of Stream Reasoning . . . 38

3.2 ASP and SR: Current State of the Art and its Limits . . . 39

3.2.1 Incremental ASP . . . 40

3.2.2 Reactive ASP . . . 42

3.3 Incremental and Online Features of Clingo 5 . . . 47

3.4 The LARS framework . . . 48

3.4.1 Preliminary Definition . . . 49

3.5 Ticker . . . 50

3.6 Laser . . . 51

4 GAME PROGRAMMING AND ARTIFICIAL INTELLIGENCE IN GAMES 53

4.1 History of AI and Games . . . 54

4.2 Artificial Intelligence in Videogames . . . 55

4.3 AI in Games: Frameworks and Competitions . . . 56

4.3.1 General Video Game AI Competition . . . 56

4.3.2 Angry Birds AI Competition . . . 58

II INCREMENTAL INSTANTIATION VIA OVERGROUNDING

5 OVERGROUNDED PROGRAMS AND EMBEDDINGS 61

5.1 Overgrounding: an overview . . . 62

5.2 Towards Overgrounded Programs: roadmap and contributions . . . 63

5.3 Preliminaries 64
5.4 Embedding Programs 65
5.5 The Overgrounding Technique 69
6 INCREMENTAL MAINTENANCE OF OVERGROUNDED PROGRAMS WITH TAILORED SIMPLIFICATIONS 71
6.1 Overgrounding with tailored simplifications: an overview 72
6.2 Tailored embeddings 74
6.3 Overgrounding with Tailoring 80
6.4 Tailored Embeddings and Related Work 85
7 INCREMENTAL GROUNDING: DATA STRUCTURES AND ALGORITHMS 87
7.1 Data Structures 87
7.2 Incremental Semi-Naïve Algorithm 89
7.2.1 Preliminaries 89
7.2.2 Algorithm 89

III IMPLEMENTATION AND APPLICATIONS
8 $\mathscr{I}^2$-*DLV*: A DECLARATIVE OVERGROUNDING-BASED SYSTEM 96
8.1 System Architecture 96
8.2 Experimental Evaluation 97
8.2.1 Multi-shot Sudoku 98
8.2.2 Multi-shot Pac-Man 100
8.2.3 Content Caching 105
9 A REAL APPLICATION SCENARIO: INTEGRATING DECLARATIVE TOOLS IN THE VIDEOGAME DEVELOPMENT LIFECYCLE 109
9.1 Unity Game Engine 111
9.2 Integrating Reasoning Modules in Game Engines 114
9.3 The ThinkEngine Framework Architecture 115
9.4 The Information Passing Layer of the ThinkEngine Framework 117
9.4.1 Sensors: Definitions and Example Configurations 118
9.4.2 Actuators: Definitions and Example Configurations 121
9.5 Declarative Side Semantic 122
9.6 The ThinkEngine Implementation: Unity and ASP 123
9.7 Rule-based reasoning modules into Unity at work 125
9.7.1 The Frogger showcase 125
9.7.2 The Tetris showcase 131
9.8 Benchmark 136
10 DECLARATIVE CONTENT SPECIFICATION VIA ANSWER SET PROGRAMMING 140
10.1 Procedural content generation: an overview 141
10.2 Declarative content generation with space partitioning in ASP 143
10.3 Implementation: an overview 146
10.3.1 Unity Asset 146
10.3.2 GVGAI plug-in 147

10.4 Performance considerations and conclusions 148
11 CONCLUSIONS 150
BIBLIOGRAPHY 153

LIST OF FIGURES

Figure 2.1 Dependency and Component Graphs. 16
Figure 2.2 ASP simple cycle 27
Figure 2.3 ASP development loop 28
Figure 2.4 A *Hydraulic Leaking* example showed step by step. 37
Figure 3.1 Incremental ASP loop 42
Figure 3.2 A blocksworld configuration example showed step by step using *oClingo*. 46
Figure 3.3 *clingo* program encoding and Python script for successive n-Queens solving 48
Figure 4.1 A screenshot of some of the test games of the GVGAI competition 57
Figure 4.2 A screenshot of the output of the computer vision module of the Angry Birds AI Competition Framework. Screenshot taken from http://aibirds.org/ 58
Figure 7.1 Predicate extension map overview - First iteration. 88
Figure 7.2 Predicate extension map - Second iteration. 88
Figure 7.3 Incremental Semi-Naïve Algorithm 92
Figure 7.4 93
Figure 8.1 A general infrastructure of an ASP overgrounding-based reasoner relying on $\mathscr{I}^2$-*DLV*. 97
Figure 8.2 Experiments on Sudoku benchmarks 98
Figure 8.3 Grounding times for all iterations of a 81×81 Sudoku instance. 99
Figure 8.4 Results of Pac-Man benchmark. Comparison of instantiation sizes for both grounders. 102
Figure 8.5 Results of Pac-Man benchmark. Comparison of grounding time for both grounders. 103
Figure 8.6 Results of Pac-Man benchmark. Comparison of solving time.103
Figure 8.7 Results of Pac-Man benchmark. Reports about cumulative execution time for four possible combinations of grounders and solvers. 104
Figure 8.8 Results of Content Caching benchmark. Comparison of instantiation sizes for both grounders. 106
Figure 8.9 Results of Content Caching benchmark. Comparison of grounding time for both grounders. 106
Figure 8.10 Results of Content Caching benchmark. Comparison of solving time. 107

Figure 8.11 Results of Content Caching benchmark. Reports about cumulative execution time for four possible combinations of grounders and solvers and the *Ticker* system in its two implementation. 107
Figure 9.1 Unity run-time execution workflow 113
Figure 9.2 Ideal architecture for threads interaction. 115
Figure 9.3 General run-time architecture of the *ThinkEngine* framework 116
Figure 9.4 The *ThinkEngine* asset class diagram 124
Figure 9.5 Sensors Configurator Component 125
Figure 9.6 Configuring a new sensor 126
Figure 9.7 Configuration of the brain. 127
Figure 9.8 Editor window for the sensor configuration 132
Figure 9.9 Configuration of the brain. 133
Figure 9.10 Framerate evaluation on Frogger game. 137
Figure 9.11 Frame rate evaluation on Tetris game. 137
Figure 10.1 Steps of Space Partitioning. (a) First step: the level is divided by a vertical line and two sub zones are defined, A and B. (b) Further steps divide each zone into smaller areas. Here A has been divided by a horizontal line, while B has been split by a vertical line. (c) After n steps, a given criterion is met, and the level will be subdivided into sectors. 142
Figure 10.2 Each obtained sector is filled with a room, which will be connected afterwards using the BSP tree. 143
Figure 10.3 Declarative Content Generation: work-flow of the proposed approach. 144
Figure 10.4 A semi-finished map obtained after the **Partition Creation** step, shown in the Unity plugin. 145
Figure 10.5 The full Unity interface. A complete map where areas are transformed in rooms and corridors is shown. On the right, it is highlighted our Unity asset. 146
Figure 10.6 GVGAI integration. 147
Figure 10.7 Benchmark test on scalability over Setting 1 (left side) and Setting 2 (right side). Times are reported in milliseconds. 148
Figure 10.8 Two sample maps obtained with Setting 1 (left-hand side) and with Setting 2 (right-hand side). 149

LIST OF TABLES

Table 5.1 Comparison of the classic model-theoretic semantics for logic programs and the embedding program semantics. . . 66
Table 6.1 Comparison between systems 86
Table 7.1 Instantiation step for a single rule with our Semi-Naïve algorithm 94
Table 8.1 Sudoku times 100
Table 9.1 Coupling goal for the ThinkEngine framework. 115
Table 9.2 Generation time 139

ACRONYMS

AI Artificial Intelligence

AIBIRDS Angry Birds AI competition

ASP Answer Set Programming

DLP Disjunctive Logic Programming

GCO Guess/Check/Optimize

GO Game Object

GVG General Video Game

GVGAI General Video Game AI

GVGP General Video Game Player

IoT Internet of Things

KB Knowledge Base

JTMS Justification-based Truth Maintenance System

KRR Knowledge Representation and Reasoning

LARS **L**ogic-based framework for **A**nalyzing **R**easoning over **S**treams

LP Logic Programming

ML Machine Learning

OPT Overgrounded Program with Tailoring

PET Predicate Extension Table

SCC Strongly Connencted Component

SP Stream Processing

SR Stream Reasoning

TMS Truth-Maintenance System

VGDL Video Game Description Language

SOMMARIO

Negli ultimi anni, la necessità sempre più emergente di eseguire ragionamenti continui sui flussi di dati ha dato origine all'area di ricerca chiamata *Stream Reasoning* (SR). Questo campo di ricerca si è rivelato di particolare importanza grazie alle sue numerose applicazioni pratiche, che includono processi decisionali per gli agenti intelligenti, la robotica, il settore automobilistico, le città intelligenti, l'automazione domestica e applicazioni su dispositivi detti *Internet of Things* (IoT). Tutte le applicazioni menzionate condividono la necessità di elaborare in modo efficiente flussi di eventi provenienti da ambienti altamente dinamici.

I metodi basati sulla rappresentazione della conoscenza, e in particolare l'Answer Set Programming (ASP), possono svolgere un ruolo significativo in questi contesti e, in effetti, alcuni di essi sono stati già applicati con successo nel contesto SR. Molte soluzioni basate su ASP sono state riadattate o sono state progettate esplicitamente per lo SR. Queste soluzioni, tuttavia, non sono in grado di risolvere definitivamente alcuni dei problemi noti in questo ambito, quali, ad esempio, la perdita di semplicità e dichiaratività, e la mancanza di prestazioni adeguate per applicazioni molto esigenti.

In questo lavoro di tesi, proponiamo una serie di nuovi metodi per soluzioni basate su ASP in ambienti altamente dinamici, che mirano ad affrontare le problematiche menzionate precedentemente. In particolare, proponiamo un approccio per il grounding incrementale in ASP e mostriamo come affrontare alcuni problemi di integrazione che sorgono quando queste soluzioni devono essere distribuite in contesti reali. Ci siamo concentrati, particolarmente, sulla possibilità di generare programmi ground incrementalmente sempre più grandi *(programmi overgrounded)* equivalenti a un dato programma logico non-ground, in modo che possano essere riutilizzati deliberatamente, con set di input diversi tra loro. L'approccio proposto è progettato per funzionare in modo "trasparente", sollevando, quindi, ingegneri e progettisti di programmi logici dall'utilizzo di aspetti tecnici specifici intrinsechi dei sistemi ASP. La natura incrementale di questa strategia consente di ridurre considerevolmente il tempo necessario per eseguire l'istanziazione di programmi logici quando la fase di grounding viene ripetuta su serie di basi di conoscenza (*Knowledge Representation* (KR)) simili tra loro. Inoltre, presentiamo una versione ottimizzata di questa strategia, e cioè l'*overgrounding con tailoring*. Più in dettaglio, i nostri Overgrounded Program with Tailoring (OPT) introducono tecniche di semplificazione per i programmi ground che consentono di (i) limitare il numero e (ii) ridurre la dimensione delle regole generate, al fine di ottimizzare le performace complessive dei programmi *overgrounded* standard.

In questa tesi, introdurremo, inizialmente ed in maniera formale, le basi teoriche dei programmi *overground* classici e la loro versione migliorata, ossia i *programmi overground con tailoring*. Successivamente, ne descriveremo le loro proprietà e illustreremo un algoritmo di aggiornamento OPT. In seguito, mostreremo la nostra implementazione e le sue prestazioni. Per mostrare l'efficacia di queste tecniche, abbiamo eseguito diversi esperimenti concentrandoci su alcuni domini applicativi classici e al contesto dei videogames. Riguardo questi ultimi, abbiamo anche indagato sulle problematiche relative all'integrazione di soluzioni basate sull'ASP nel vero *lifecycle* di un videogame.

ABSTRACT

In the last years, the emergent need to perform continuous reasoning over streams has given rise to the *Stream Reasoning* (SR) research area. This research field turned out to be of particular importance thanks to its several practical applications, which include decision making for agents, robotics, automotive, smart cities, home automation and general *Internet of Things* (IoT) applications. All the above applications share the necessity of efficiently processing fast-paced event flows coming from highly dynamic environments.

Methods based on *Knowledge Representation* (KR), and especially Answer Set Programming (ASP), can play a significant role in the above contexts, and indeed have been already applied to the SR field with some degree of success. Many ASP-based solutions can be adapted, or were explicitly designed having SR in mind. Such solutions, however, leave open some issues that need to be tackled, such as some loss of simplicity and declarativity, and a performance not yet sufficient for highly-demanding applications.

In this thesis work, we propose a number of new methods for ASP-based solutions in highly dynamic environments, which aim to face the above issues. In particular, we propose an incremental grounding approach for the answer set semantics and we study how to cope with some nonobvious integration problems arising when reasoning-based solutions must be deployed in real contexts. We focused particularly on the possibility of generating incrementally larger ground programs (*overgrounded programs*) equivalent to a given non-ground logic program, so that they can be reused in combination with deliberately many different sets of inputs. The proposed approach is designed to work "transparently", thus relieving knowledge engineers and designers of logic programs from using specific technical aspects related to ASP solvers. The incremental nature of this strategy permits to considerably reduce time performance when grounding is repeated on a series of similar knowledge bases. Furthermore, we also present an optimized version of this strategy, namely *overgrounding with tailoring*. More in detail, our Overgrounded Program with Tailoring (OPT) introduce simplification techniques for ground programs allowing to (i) limit the number and (ii) reduce the size of the generated rules, in order to optimize the overall performance of the basic *overgrounded* programs.

In this thesis, we first formally introduce the theoretical basis of classical *overgrounded programs* and their enhanced version, *overgrounded programs with tailoring*. Then, we describe their properties and we illustrate an OPT update algorithm. We then report about our implementation and its performance. To show the effectiveness of these techniques, we performed several experiments focusing in some applicative domains and particularly in the videogames context. Concerning the latter, we also investigate on the challenging issues of integrating ASP-based solution in the real videogame development lifecycle.

1 INTRODUCTION

Motivation and Objectives

Recently, the availability of a number amount of *data streams* greatly stimulated the research of suitable processing paradigms and tools. Consequently, the research community looked with interest in advancing fast and "non-materializing" stream processing techniques; such techniques are based on the idea of storing as little data as possible (in contrast with static processing on stored databases), and on pushing outputs to consumers on-the-fly as soon as they become available.

This growing need to perform continuous reasoning over streams has given rise to *Stream Reasoning* (SR), a high impact research area, also plenty of unexplored paths. A *stream* can be formally seen as an unbounded sequence of time-varying data elements. This notion is general enough to encompass a large panorama of practical applications [41], among which one can mention, e.g., decision making for agents, robotics, automotive, *Internet of Things* (IoT), and real-time videogames [115, 29].

Among exemplary SR's application fields, it is worth mentioning the IoT area. Enabling advanced and complex reasoning capabilities in IoT applications, could be largely beneficial, especially when massive groups of IoT devices need to be leveraged. In the last ten years, this area has had a very fast growth and it is predicted to expand even more in the future, thanks to the new technologies coming from domotics, home automation, health-care and transportation fields. The IoT concept represents the new frontier of the use of the Internet, through which objects (also called "things"), become recognizable and share data with other objects. Objects acquire intelligence thanks to this continuous interaction and communication: e.g., alarms may sound earlier in case of traffic; smart-watches could keep track of usual fitness activity and remind when it is required to perform some activity like walking, running or cycling; smart TVs could automatically understand which are your favourite TV programs and automatically show appropriate reminders. In general, in the IoT context, devices first acquire information about the surrounding environment, then some decision-making logic takes actions based on the input knowledge, which appears in the form of an event stream. Clearly, since most of such information *flows* come at a relatively high pace and high volume, the ability to reason on such data in real-time constitutes a challenge that, if solved, could benefit many advanced applications.

Another interesting field in which event streams play a crucial role, is *Artificial Intelligence* (AI) applied to videogames. Decision-making based on some AI technique is a central component encoded in almost all videogames and it is often implemented

with ad-hoc solutions for the specific game. Historically, very basic techniques have been used to design and implement game AI. As in the videogame industry there is growing demand for both high runtime performance and short development times, more general and advanced approaches, possibly based on SR techniques, would be welcome.

Requirements for stream reasoners are usually quite strict: for instance, an artificial player, deployed in a real-time videogame, is subject to a very fast flow of input events, yet it is allowed a very limited time for each decision, like in the *General Video Game AI* (GVGAI) competition [104] where this limit is just 40 milliseconds. At the same time the IoT devices collect huge amounts of "complex" data, i.e., data organized in a hierarchical structure, and these need to be extracted, filtered and analysed in a very restricted time limit. The current explored approaches still do not close some strategic gaps, among which one can mention:

a) generalization gaps: although very powerful, *Machine Learning* (ML) techniques require to build specific ad-hoc solutions which work just on a per application basis. One might want to look at solutions in which general middleware tools are available and allow to easy tune, refine and prototype "intelligent" processing capabilities;

b) explainability and "tunability" gaps: a stream processing solution should be based on intelligible, modifiable and explainable knowledge; this would let human actors to better convey their desiderata to implemented solutions;

c) gaps due to limited and opaque efficiency: in many industrial applications, developers expect to deal with optimized stream processing techniques in which fine-tuning and human intervention is reduced to the bare minimum or it is not necessary at all.

Answer Set Programming (ASP) appears to be a good candidate in order to deal with the above gaps. ASP is a declarative programming paradigm which has its roots in *Knowledge Representation and Reasoning* (KRR) and Logic Programming (LP). As a flexible language for declarative problem solving, ASP allows the user to refrain from providing an algorithm for solving the problem at hand: it is sufficient to specify the properties a desired solution must (or must preferably) have in the form of an executable logic specification and run the resulting program to obtain the so called *answer sets*, i.e., a set of possible solutions for the current instance of the problem.

Moreover, ASP is a very expressive language enable to model a large variety of problems. ASP attributes a non-monotonic semantics to disjunctions in the head of the rules, and to unstratified negation. The availability of non-monotonic reasoning features facilitates modelling of common-sense knowledge, and real world problems in general. In the latest years, ASP has been widely used in the field of AI as well as in the industries in order to deploy many applications, also in scenarios where a

high response reactivity was required [55, 28, 81, 99, 113, 127]. In the SR context, it is implied that reasoning tasks could be triggered by events which come at a fast pace, thus requiring to continuously repeat evaluations on very similar inputs. In this respect, having fast, possibly incremental, processing techniques can make a clear difference with regard to performance.

The typical workflow of ASP systems consists in an *instantiation* (or *grounding*) phase and a subsequent *solving* (or *answer sets search*) phase. In the first step, a *grounder* module produces an equivalent propositional program $gr(P \cup F)$ from an input non-ground logic program P and a set of facts F; in the latter step, a *solver* module applies dedicated search techniques on $gr(P \cup F)$ for computing the actual semantics of $P \cup F$ in the form of *answer sets* [75]. Repeated executions, called *shots* or *iterations*, can be conceptually abstracted to the task of finding the set of answer sets $AS(P \cup F_i)$ for a sequence of input fact sets $F_1, \ldots, F_n$.

The research effort towards the development of incremental reasoning techniques in the ASP community has focused on different aspects. In the clingo system and its earlier prototypes [63] a knowledge designer is allowed to procedurally control how and which parts of the logic program at hand must be incremented, updated and evaluated among consecutive shots. This enables a possibility of manually model which and how ground subprograms and partial answer sets must be maintained. Also, it is possible to control which parts of a program are subject to the incremental evaluation with respect to an iteratively increasing integer parameter. This approach introduces ample flexibility but requires a non-negligible knowledge of solver-specific internal algorithms. Nevertheless, declarativity and fast-prototyping capabilities are a priority in many development scenarios, such as the previously mentioned videogame industry where designers, who do not have knowledge of backstage technical aspects of declarative logic programming at all, look for easy and off-the-shelf scripting solutions.

Another approach to incremental reasoning is the *Ticker* evaluation system [16]. *Ticker* implements LARS, a **L**ogic-based framework for **A**nalyzing **R**easoning over **S**treams using with ASP-like semantics, and makes use of incremental techniques for a fragment of the LARS input language. LARS allows omni comprehensive, yet demanding, temporal data management features which make implementation difficult and complex.

Thesis Overview and Contributions

The main goal of this thesis is to propose and use new advanced incremental techniques in order to promote efficient, easy to use declarative reasoning under the answer set semantics in highly-dynamic applicative environments. The contributions

of this thesis can be categorized in two groups: (a) incremental grounding techniques for ASP solvers, and (b) applications of ASP and incremental techniques in real domains, with particular focus on videogames.

Concerning group (a), we concentrated our research on the ASP grounding step, particularly looking at the possibility of caching and re-using ground programs. We first formalized two notions of ground programs respectively called *embeddings* and *tailored embeddings*, which capture two different notions of optimized ground program. The first notion is clean and simpler, whereas the second, more refined one, captures smaller and better optimized ground programs at the price of simplicity.

By using the formal notions above, we implemented two incremental strategies based respectively on maintaining an *overgrounded program* G_P, or a *Overgrounded Program with Tailoring (*OPT*)* TG_P. Both types of programs are made "compatible" with new input facts by monotonically and dynamically enlarging them from one shot to another.

When updating overgrounded programs (possibly, with tailoring), the computational effort is generally small as overgrounded programs are updated only with the addition of new logical assertions, while the likelihood of generating new insertions fades away with further iterations. Overgrounding is attractive, works transparently and does not require programming burden, since no operational statements are required to incrementally drive the computation. Moreover, an overgrounded program, after some update iterations, converges to a propositional theory general enough to be reused together with possible future inputs, with no further update required and virtually eliminating grounding activities in later iterations, making possible a consistent enhancement of the time performance[1].

Additionally, OPTs overcome the limitations of overgrounded programs by introducing already known simplification techniques for ground programs which allow to (i) limit the number and (ii) reduce the size of the generated rules [65, 51].

As a second category of contributions, we explored the possibilities of ASP and our incremental techniques in some appropriate scenarios of choice, and particularly in the context of videogames. More in detail,

1. we implemented a new grounder based on our proposed technique and we conducted proper performance evaluation which we report about; the new grounder, called $\mathscr{I}^2$*-DLV*, implements both the overgrounding and/or tailored

1 Note that in some applications, it is possible to obtain such a propositional theory before the very first shot, as in [48]. This pre-grounding approach can be used on the assumption that all the input constants that can appear in a logic program are known apriori. Pre-grounding is not flexible enough in case new invented constant values appear dynamically in later shots. Also, pre-grounding often requires to resort to a large overestimate of the possible input constant symbols, thus negatively impacting on the size of the "pre-grounded" program.

overgrounding policy; in our experiments we show the pros and cons of both along with a comparison with non-incremental grounders;

2. we investigated how reasoning modules behave when integrated in a real videogame execution loop; in particular we report about our experiments on some videogame domains;

3. eventually, we explored also the possibility of using declarative techniques in other demanding areas of industrial videogame development, such as generation of game contents based on declarative content specifications.

Organization of the Thesis

This thesis is organized into three parts:

- In the first part we present an overall introduction of the main concepts that will be discussed in this work. More in detail, in Chapter 2 we will briefly introduce declarative languages focusing on Answer Set Programming. We will formalize its syntax and semantics and we will provide some example of known *Knowledge Representation and Reasoning* (KRR) problems solved using ASP; Chapter 3 will provide some preliminaries about the *Stream Reasoning* research area, motivate its importance in the current research and report about the state of art of current systems; finally, Chapter 4 will provide a brief introduction to the vast research area of *Artificial Intelligence* (AI) in Games and show its relationship with the *Stream Reasoning* (SR) world.

- The second part of this work focuses on the design techniques for an efficient incremental instantiation via overgrounding. In Chapter 5, *embeddings* and the basic *overgrounding* technique will be formalized and described; whereas, in Chapter 6, *tailored embeddings* and *overgrounded programs with tailoring* will be described. A detailed description of the algorithm able to manage overgrounded programs with tailoring is given in Chapter 7.

- The third part of this thesis is devoted to the description of the implemented system and to its application in real world contexts. More in detail, Chapter 8 will focus on the architecture of $\mathscr{I}^2$-*DLV*, i.e., the system that implements the *overgrounding* techniques previously described, and on the experimental evaluation we conducted to test its performance; Chapters 9 and 10 expands on some of the experimental domains coming from the videogame world, and on other possible application of declarative *Artificial Intelligence* (AI) in the videogame context, showing how repeated evaluation techniques could be beneficial in such a category of real world domain.

The thesis is closed with a conclusion chapter, where future and on-going work is outlined.

Publications

It is worth to point out that the national and international scientific community already acknowledged part of the contribution given within this thesis on the following papers:

[IPZ20] Giovambattista Ianni, Francesco Pacenza, and Jessica Zangari. Incremental maintenance of overgrounded logic programs with tailored simplifications. *Theory and Practice of Logic Programming*, 20(5):719-734 (2020).

[CIP+19] Francesco Calimeri, Giovambattista Ianni, Francesco Pacenza, Simona Perri, and Jessica Zangari. Incremental answer set programming with overgrounding. *Theory and Practice of Logic Programming*, 19(5-6):957–973, 2019.

[Pac19] Francesco Pacenza. Reasoning in highly reactive environments. In Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang, editors, *Proceedings 35th International Conference on Logic Programming (Technical Communications), International Conference on Logic Programming 2019 Technical Communications, Las Cruces, NM, USA, September 20-25*, 2019.

[IPZ19] Giovambattista Ianni, Francesco Pacenza, and Jessica Zangari. An infrastructure for multi-shot reasoning with incremental grounding. In Alberto Casagrande and Eugenio G. Omodeo, editors, *Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, June 19-21, 2019*, volume 2396 of *CEUR Workshop Proceedings*, pages 269–278. CEUR-WS.org, 2019.

[AIP19] Denise Angilica, Giovambattista Ianni, and Francesco Pacenza. Tight integration of rule-based tools in game development. In Mario Alviano, Gianluigi Greco, and Francesco Scarcello, editors, *Italian Association for Artificial Intelligence 2019 - Advances in Artificial Intelligence - XVIIIth International Conference of the Italian Association for Artificial Intelligence, Rende, Italy, November 19-22, 2019, Proceedings*, volume 11946 of *LNCS*, pages 3–17. Springer, 2019.

[CGI+18a] Francesco Calimeri, Stefano Germano, Giovambattista Ianni, Francesco Pacenza, Armando Pezzimenti, and Andrea Tucci. Answer set programming for declarative content specification: A scalable partitioning-based approach. In Chiara Ghidini, Bernardo Magnini, Andrea Passerini, and Paolo Traverso, editors, *Italian Association for Artificial Intelligence 2018 - Advances in Artificial Intelligence - XVIIth International Conference of the Italian Association for Artificial Intelligence, Trento, Italy, November 20-23, 2018, Proceedings*, volume 11298 of *LNCS*, pages 225–237. Springer, 2018.

[CGI+18b] Francesco Calimeri, Stefano Germano, Giovambattista Ianni, Francesco Pacenza, Simona Perri, and Jessica Zangari. Integrating rule-based AI tools into mainstream game development. In *International Joint Conference on Rules and Reasoning 2018*, volume 11092 of *LNCS*, pages 310–317, 2018.

Part I

CONTEXT AND FOUNDATIONS

2 ANSWER SET PROGRAMMING

The primary intent of Computer Science is problem solving by means of machines. In this context, programming languages permit to obtain the solution(s) of a given problem, enabling the "communication" between humans and machines. Such communication might follow two radically different approaches: imperative or declarative. Imperative languages are *machine-oriented*: they require to model in a formal, machine-oriented wording how a problem should be solved. Determining efficient algorithms to solve complex (yet, tractable) problems often, requires advanced knowledge and quite good programming skills. In addition, since the conceptualization of a problem and its solution(s) are implicitly wired in the code, the imperative approach demonstrates, in general, a low *elaboration tolerance*, that is slight updates to the problem specifications, often, require a significant effort to modify the code accordingly.

An opposite approach is provided by declarative languages, which, instead, are more human-oriented, as they permit programmers to concentrate on problem definitions. Consequently, variations in specifications tend to have a much smaller impact on the code, since it explicitly reflects problem specifications.

Around the 1950s, John McCarthy [77] discussed how logic is particularly suited to be a full-fledged declarative programming paradigm, allowing to model problems in a natural and human-oriented fashion, and effectively represent knowledge representation and rational human reasoning. In the same years, a new computer science field was born: AI, and logic-based languages gained more and more importance and popularity. A breakthrough happens when Alain Colmerauer and its research group introduced Prolog [38] (from the French, PROgramming en LOGic), the first logic programming language. However, it emerged that the first-order logic on which Prolog is based is not capable of modelling the commonsense human reasoning, which is non-monotonic: we as humans, starting from some premises, may rationally regret them whenever new information become available, while in first-order logic, logical consequences cannot be invalidated since the underlying reasoning is monotonic. Subsequently, new logic formalisms devoted to represent non-monotonic reasoning were introduced, such as Default Logic [109], Autoepistemic Logic [94] and Circumscription [90]. In the late '80s and early '90s, Michael Gelfond and Vladimir Lifschitz presented the logic formalism Answer Set Programming (ASP) [70, 71] allowing to express non-monotonic reasoning in purely declarative fashion [21, 44, 45, 71, 89, 98]. ASP became widely used in AI and recognized as a powerful tool for KRR.

In this chapter we introduce Answer Set Programming (ASP). The core of the language consists in Disjunctive Datalog with nonmonotonic negation under the stable model semantics. Nevertheless, over the years a significant amount of work has been carried out by the scientific community in order to enrich the basic language, and several extensions have been studied and proposed. Recently, the community agreed on a standard input language for ASP systems: ASP-Core-2 [26], the official language of the ASP Competition series [28, 66, 67].

The chapter is structured as follows. Sections 2.1 and 2.2 present a formal definition of the syntax and the semantics of ASP. Section 2.3 describes some syntactic shortcuts. Section 2.4 introduces the concept of *safety*. Eventually, Sections 2.5 and 2.6 describe the capability of ASP as a tool for KRR in real world domains.

The herein reported definitions are compliant with ASP-Core-2 v.2.03c (the latest version at the time of writing).

2.1 Syntax

Let $\mathscr{I}$ be a set of *identifiers*. An identifier is a not empty string starting with some lowercase letter and containing only alphanumeric symbols and the symbol "_" (underscore).

Example 2.1.1. Examples of identifiers are: a, $a1_B$, a_ID, $vertex$

2.1.1 Terms

A term is either a *constant*, a *variable*, an *arithmetic term* or a *functional term*. In particular, constants and variables can be considered as "basic terms", while arithmetic and functional terms are defined inductively as combinations of terms.

Definition 2.1.1 (Constant Term)**.** A *constant* is either a *symbolic constant*, if it is an identifier, a *string constant*, if it is a quoted string, or an *integer*.

Definition 2.1.2 (Variable Term)**.** A *variable* is a not empty string starting with some uppercase letter and containing only alphanumeric symbols and the symbol "_" (underscore).

Furthermore a special variable, namely *anonymous variable*, is represented by the symbol "_" (underscore). This syntactic shortcut is intended to indicate a *fresh* variable, that does not appear elsewhere in the context in which it is located.

Definition 2.1.3 (Arithmetic Term)**.** An *arithmetic term* has form -(t) or $(t_1 \diamond t_2)$ for terms t_1 and t_2 with $\diamond \in \{+, -, *, /\}$. Parentheses can optionally be omitted, and standard operator precedences apply.

Definition 2.1.4 (Functional Term)**.** A *functional term* has form $f(t_1, \ldots, t_n)$, where f is an identifier, known as *functor*, $t_1, \ldots, t_n$ are terms and $n > 0$.

Example 2.1.2. Examples of terms are:

– Constants: a, x, "*http://google.com*", 0, 123

– Variables: X, X_134, $X2$, $Color$

– Arithmetic terms: -X, $X+Y$, $2*(\text{-}5)$, $X+ab$, $X/3$

– Functional terms: $f(X)$, $father(aristotle)$, $g(2*5,\text{"}abc\text{"})$

A term is *ground* (i.e., variable-free) if it does not contain any variable. For, instance in Example 2.1.2 all the constants, the arithmetic term $2*(\text{-}5)$ and the functional terms $father(aristotle)$ and $g(2*5,\text{"}abc\text{"})$ are ground.

2.1.2 Atoms and Literals

Definition 2.1.5 (Predicate)**.** Given an identifier p and an integer n with $n \geq 0$, the expression p/n represents a *predicate*. p is said *predicate symbol* and n represents the associated arity.

Example 2.1.3. Examples of predicates are: $a/2$, $p/3$, $predicate_3/1$, $true/0$.

In the following, when no ambiguities arise we denote simply as p a predicate p/n.

Definition 2.1.6 (Predicate Atom)**.** A *predicate atom* has form $p(t_1,\ldots,t_n)$, where $n \geq 0$, p/n is a predicate with predicate name p and arity n and $t_1,\ldots,t_n$ are terms; if $n=0$, parenthesis are omitted and the notation p is used.

Definition 2.1.7 (Classical Atom)**.** A *classical atom* is either -a or a where a is a predicate atom and $-$ denotes the *strong negation* symbol.

Definition 2.1.8 (Built-in Atom)**.** A *built-in atom* has form $t_1 \rhd t_2$ where t_1,t_2 are terms and $\rhd \in \{<,\leq,=,<>,\neq,>,\geq\}$.

Definition 2.1.9 (Naf-Literal)**.** A *naf-literal* can either be a built-in atom or have form a or *not* a where a is a classical atom, and *not* is the *negation as failure* symbol.

Example 2.1.4. Some examples are shown below.

– Predicate Atoms: $edge(X,Y)$, $atom(f(a,b),c)$, $true$

– Classical Atoms: $edge(X,Y)$, $atom(f(a,b),c)$, $true$, -$true$

– Built-in Atoms: $father(aristotle) = nicomachus$, $X \neq Y$, $X*2=Y$

– Naf-Literals: $father(aristotle) = nicomachus$, $X \neq Y$, $X*2=Y$, $edge(X,Y)$, -$atom(f(a,b),c)$, $true$, -$true$, *not* -$true$, *not* $true$

In addition to the type of atoms above illustrated, *aggregate atoms* have been introduced to permit aggregation operations on multi-sets of terms by means of concise expressions.

Definition 2.1.10 (Aggregate Element)**.** An *aggregate element* is composed as: $t_1,\ldots,t_m : l_1,\ldots,l_n$, where $t_1,\ldots,t_m$ are terms $l_1,\ldots,l_n$ are naf-literals for $n \geq 0$, $m \geq 0$.

Definition 2.1.11 (Aggregate Atom)**.** An *aggregate atom* has form:

$$af\{e_1,\ldots,e_n\} \rhd t$$

where:

- $af \in \{\#count, \#sum, \#max, \#min\}$
- $e_1,\ldots,e_n$ are aggregate elements for $n \geq 0$
- $\rhd \in \{<, \leq, =, <>, \neq, >, \geq\}$
- t is a term

Definition 2.1.12 (Aggregate Literal)**.** An *aggregate literal* is either a or *not* a where a is an aggregate atom.

Example 2.1.5. For instance, the following are aggregate literals: *not* $\#max\{X,Y : age(X,Y)\} < 20$, $\#sum\{X,Y : age(X,Y)\} = s(S)$, $\#count\{1 : a(1)\} > 3$. Moreover, the latter two literals are also aggregate atoms.

An atom is *ground* if it does not contain any variable. A literal is ground if its atom is ground. In Examples 2.1.4 and 2.1.5 $father(aristotle) = nicomachus$, -$atom(f(a,b),c)$, *true*, *-true*, *not -true*, *not true*, $\#count\{1 : a(1)\} > 3$ are ground.

In the following we will refer to classical, built-in and aggregate atoms as *atoms*. Similarly, we will indicate naf and aggregate literals as *literals*. A literal is *negative* if the *not* symbol is present, otherwise it is *positive*.

2.1.3 Rules, Constraints, Queries and Programs

After defining the basic constructs, we now describe the main components of an ASP logic program.

Definition 2.1.13 (Rule)**.** A *rule* r has the following form:

$$a_1 \mid \ldots \mid a_n \text{ :- } b_1,\ldots,b_m.$$

where:

- $a_1,\ldots,a_n$ are classical atoms
- $b_1,\ldots,b_m$ are literals
- $n \geq 0, m \geq 0$

The *disjunction* $a_1 \mid \ldots \mid a_n$ is the *head* of r, while the *conjunction* $b_1, \ldots, b_m$ is the *body* of r. We denote by $H(r)$ the set $\{a_1, \ldots, a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, \ldots, b_m\}$ of the body literals. $B^+(r)$ denotes the set of literals occurring positively in $B(r)$; while $B^-(r)$ is the set of negative literals in $B(r)$. A rule having precisely one head literal (i.e., $n = 1$) is said to be a *normal rule*; if $n > 1$ the rule is *disjunctive*.

Example 2.1.6. Examples of rules are:

$$hasUmbrella(X) \mid doesNotHaveUmbrella(X) \text{ :- } person(X).$$

$$isRaining \mid \text{-}isRaining \text{ :- } cloudyWeather.$$

Definition 2.1.14 (Fact)**.** A rule r is a *fact* with $B(r) = \emptyset$, $|H(r)| = 1$ and $H(r) = \{a\}$ where a is a classical ground atom.

Example 2.1.7. Examples of facts are:

$$cloudyWeather.\ \text{-}isRaining.\ person(alice).\ person(bob).$$

the :- sign is usually omitted.

In the following, as it is common, we will adopt the notation reported next to represent in a compact way a set of facts: $p(m_{1_1}..m_{1_2}, \ldots, m_{n_1}..m_{n_2})$. where p/n is a predicate of arity n, and m_{i_j} with $i \in \{1, \ldots n\}$ and $j \in \{1,2\}$ are terms. For instance, $a(1..2, f(3..4))$. defines the facts: $a(1, f(3))$. $a(2, f(3))$. $a(1, f(4))$. $a(2, f(4))$.

A predicate p/n is referred to as an *EDB* predicate if, for each rule r in which p/n appears in $H(r)$, r is a fact; all others predicates are referred to as *IDB* predicates. The set of facts in which *EDB* predicates occur, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

Definition 2.1.15 (Strong (or Integrity) Constraint)**.** A *strong constraint s* is a rule with $|H(s)| = \emptyset$.

Definition 2.1.16 (Weak Constraint)**.** A *weak constraint c* is a special type of rule, having form:

$$:\sim\ b_1, \ldots, b_m.\ [w@l, t_1, \ldots, t_n]$$

where:

- $n \geq 0, m \geq 0$
- $b_1, \ldots, b_m$ are literals
- $w, l, t_1, \ldots, t_n$ are terms; w and l are referred to, respectively, as *weight* and *level* for c; if $l = 0$, the expression @0 can be omitted.

Basically, a weak constraint is like a strong one, where the implication symbol :- is replaced by :∼. The informal meaning of a weak constraint :∼ B. is "try to falsify B," or "B should preferably be false".

For a weak constraint c we will indicate as *weak specification*, denoted $W(c)$, the part within the square brackets.

Example 2.1.8. Examples of constraints are:

$$:\!-\ isRaining,\ not\ isWetStreet.$$
$$:\sim\ isRaining, person(X),\ not\ hasUmbrella(X).\ [1]$$

A rule r is *ground* if all the atoms in $H(r)$ are ground and all the literals in $B(r)$ are ground. A strong constraint s is ground if all the literals in $B(s)$ are ground. A weak constraint c is ground if all the literals in $B(c)$ are ground, and all the terms in its weak specification $W(c)$ are ground.

In Example 2.1.6 the rule *isRaining* | *-isRaining*:- *cloudyWeather*. is ground, as well as the two constraints in Example 2.1.8.

For a literal l, let $var(l)$ be the set of variables appearing in l; if l is ground $var(l) = \emptyset$. For a conjunction of literals C, $var(C)$ denotes the set of variables occurring in the literals in C; similarly, for a disjunction of atoms D, $var(D)$ denotes the set of variables in the atoms in D. Inductively, for a rule r, $var(r) = var(H(r)) \cup var(B(r))$; for a strong constraint s, $var(r) = var(B(s))$; for a weak constraint c, $var(c) = var(B(c)) \cup var(W(c))$.

Given a rule or weak constraint r, a variable X is *global* if it appears outside of an aggregate element in r; we denote as $var_g(r)$ the set of global variables in r. Given an aggregate element e in a rule or weak constraint r, $var_l(e) = var(e) \setminus var(r)$ denotes the set of *local* variables of e, i.e., the set of variables appearing only in e, while the set of *global* variables of e contains variables appearing in both r and e, i.e., $var_g(e) = var(r) \cap var(e)$. Suppose that r contains the aggregate elements $E = \{e_1, \ldots, e_n\}$, then $var(r)$ can be also defined as $var(r) = var_g(r) \cup \bigcup_{i=1}^{n} \{var_l(e_i) | e_i \in E\}$; if $n = 0$ and thus $E = \emptyset$, i.e., r does not contain any aggregate element, then $var(r) = var_g(r)$.

Example 2.1.9. As an example, given the following rule r:

$$a(X) :\!-\ b(X),\ \ not\ c(X), \#sum\{Y : d(X,Y); Z : f(Z)\}.$$

we can observe that $var(r) = \{X,Y,Z\}$, $var_g(r) = \{X\}$, $var_l(Y : d(X,Y)) = \{Y\}$, $var_l(Z : f(Z)) = \{Z\}$.

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

Definition 2.1.17 (Query)**.** A *query* has form: a? where a is a classical atom.

Example 2.1.10. Examples of queries are: -*isRaining*?, $hasUmbrella(X)$?.

A query is *ground* if its atom is ground. In Example 2.1.10 -*isRaining*? is ground.

Definition 2.1.18 (Program)**.** A *program* is a finite set of rules, possibly accompanied by a single query.

A program is *ground* if all its rule, constraints, and the possible query are ground. A program containing disjunctive rules is *disjunctive*, otherwise it is *non disjunctive*.

Example 2.1.11. The following constitutes a *disjunctive* program:

$hasUmbrella(X) \mid doesNotHaveUmbrella(X)$:- $person(X)$.
isRaining | -*isRaining* :- $cloudyWeather$.
:- *isRaining*, *not* $isWetStreet$.
:∼ *isRaining*, $person(X)$, *not* $hasUmbrella(X)$. [1]
$cloudyWeather$. -*isRaining*.
$person(alice)$. $person(bob)$.
$hasUmbrella(bob)$?

Programs are also classified according to their structural properties, such as dependencies among predicates [25].

Definition 2.1.19. (Dependency Graph) The *Dependency Graph* of P is a directed graph $G_P = \langle N, E\rangle$, where N is the set of IDB predicates of P, and E contains an edge $(p/n, q/m)$ if there is a rule r in P such that q/m occurs in the head of r and p/n occurs in a classical atom of $B(r)$ or in a classical atom within an aggregate literal of $B(r)$.

The graph G_P induces a partition of P into subprograms (also called *modules*). For each strongly connected component (SCC)[1] C of G_P (a set of predicates), the set of rules defining the predicates in C is called *module* of C and is denoted by M_C. A rule r occurring in a module M_C (i.e., containing in its head some predicate $q/m \in C$) is said to be *recursive* if there is a predicate $p/n \in C$ in the positive body of r; otherwise, r is said to be an *exit rule*. Moreover, we say that p/n and q/m are *recursive* predicates. A program containing at least a recursive rule is said to be *recursive*.

Definition 2.1.20. (Component Graph) The *Component Graph* of a program P is a directed labelled graph $G_P^c = \langle N, E, lab\rangle$, where N is the set of strongly connected components of G_P, and E contains:

- an edge (B,A) with $lab((B,A)) =$"+", if there is a rule r in P such that $a \in A$ occurs in the head of r and $b \in B$ occurs in a classical atom of $B(r)$ or in a classical atom within an aggregate literal of $B(r)$;

1 We briefly recall that a strongly connected component of a directed graph is a maximal subset of the vertices, such that every vertex is reachable from every other vertex.

– an edge (B,A), with $lab((B,A)) =$"-", if there is a rule r in P such that $a \in A$ occurs in the head of r and $b \in B$ occurs in a negative naf-literal of $B(r)$ or in a negative naf-literal within an aggregate literal of $B(r)$, and there is no edge e' in E, with $lab(e') =$"+".

A predicate p/n is *stratified* [8] with respect to negation if it does not occur in cycles in G^c_P involving negative dependencies (i.e., edges labelled with "-"), otherwise p/n is said *unstratified*. Consequently, a program P is *stratified* with respect to negation if every predicate appearing in it is stratified, or equivalently, if no cycles in G^c_P involve negative dependencies, otherwise P is said *unstratified*. A predicate is *solved* if: (i) p/n is defined solely by non-disjunctive rules (i.e., all rules with p/n in the head are non-disjunctive), and (ii) q does not depend (even transitively) on any unstratified predicate or disjunctive predicate (i.e., a predicate defined by a disjunctive rule).

The *Component Graph* induces a partial ordering among the SCCs of the *Dependency Graph* as follows. For any pair of nodes A,B of G^c_P, *A positively precedes B* in G^c_P (denoted $A \prec_+ B$) if there is a path in G^c_P from A to B in which all arcs are labeled with "$+$"; *A negatively precedes B* (denoted $A \prec_- B$), if there is a path in G^c_P from A to B in which at least one arc is labeled with "$-$". This ordering induces admissible component sequences $C_1,\dots,C_n$ of SCCs of G_P such that for each $i < j$

– $C_j \not\prec_+ C_i$;

– if $C_j \leftarrow C_i$ then there is a cycle in G^c_P from C_i to C_j (i.e., either $C_i \prec_+ C_j$ or $C_i \leftarrow C_j$).

Several sequences exist in general.

Example 2.1.12. As an example let us consider the following program P_1:

$$r_1 : \ a(X) \text{ :- } b(X),\ not\ c(X).$$
$$r_2 : \ b(Y) \text{ :- } a(Y), Y = X+1, f(X).$$
$$r_3 : \ c(X) \text{ :- } d(X),\ not\ a(X).$$
$$r_4 : \ d(X) \text{ :- } f(X),\ not\ g(X).$$

The dependency and component graphs are illustrated in Figure 2.1. In the dependency graph G_{P_1}, there are three components: (1) a first component C_1 is formed by predicate $a/1$ and $b/1$, (2) the predicate $c/1$ forms another component C_2, and (3) a third component C_3 is composed by the predicate $d/1$. Hence, $M_{C_1} = \{r_1, r_2\}$, $M_{C_2} = \{r_3\}$, $M_{C_3} = \{r_4\}$. Moreover, the rules r_1 and r_2 are recursive, thus P_1 is recursive. Finally, in the component graph $G^c_{P_1}$ there is a cycle involving components $\{a/1, b/1\}$ and $\{c/1\}$, and so P_1 is *unstratified* under negation.

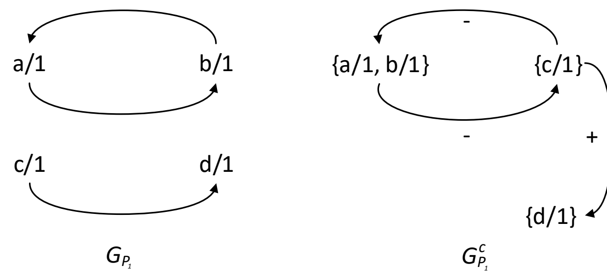


Figure 2.1: Dependency and Component Graphs.

2.2 SEMANTICS

The semantics of an ASP program is given by the set of its *answer sets*. Each *answer set* corresponds to a solution for the encoded problem. Notably, ASP is a fully declarative paradigm: the order in which the program is composed by rules, constraints and query, as well as the order of literals and atoms in the rules bodies and heads, have no effect on the semantics.

Furthermore, answer sets are defined for ground programs only. However, for every non-ground program, a semantically equivalent ground program can be defined. The process of producing such a ground program is referred to as *instantiation* or *grounding*. Essentially, for each rule of a non–ground program its variables are considered universally quantified and ranging over the set of ground terms defined by the program Herbrand Universe. Intuitively, variables are just an abstraction to represent ground terms.

In the following, we formalize the semantics of ASP-Core-2, obtained by inheriting the semantics proposed in [71] as a generalization of *stable models* semantics [70], extended to aggregates according to [52, 53].

2.2.1 THEORETICAL INSTANTIATION

Let P be an ASP program.

Definition 2.2.1 (Herbrand Universe)**.** The *Universe of Herbrand* of P, U_P, is the set of all integers and ground terms constructible from constants and functors appearing in P. In case no constant appears in P an arbitrary constant c is added to U_P.

Definition 2.2.2 (Herbrand Base)**.** The *Base of Herbrand* of P, B_P, is the set of all ground classical atoms obtainable by combining predicate names appearing in P with terms from U_P as arguments.

Example 2.2.1. As running example, let us consider the program P_1:

$$b(1).\ b(2).\ c(1).$$
$$a(X) \text{ :- } b(X),\ \ not\ c(X*1).$$
$$d(Y) \text{ :- } \#count\{X:\ a(X)\} \ =\ Y.$$

then $U_{P_1} = \{1,2\}$ and $B_{P_1} = \{a(1),a(2),b(1),b(2),c(1),c(2),d(1),d(2)\}$.

Definition 2.2.3 (Substitution)**.** Given a Herbrand Universe U_P of a program P and a set of variables V, a *substitution* is total function $\sigma : V \mapsto U_P$ that maps each variable in V to an element in U_P. For some object O occurring in P (term, atom, aggregate atom, literal, rule, weak constraint, query, etc.), we denote by $O\sigma$ the object obtained by replacing each occurrence of a variable $v \in var(O)$ by $\sigma(v)$ in O. σ is well-formed if the arithmetic evaluation, performed in the standard way, of each arithmetic sub-term t in $O\sigma$ is well-defined.

In the following, we will denote a substitution σ also as the set $\{X = c \mid \sigma(X) = c\}$.

Definition 2.2.4 (Global and Local Substitutions)**.** Given a rule or weak constraint r in P a substitution is *global* if it involves variables in $var_g(r)$; for an aggregate element e in r, a substitution is *local* if it involves variables in $var_l(e)$.

We remark that for terms, classical atoms, naf-literals and queries a substitution is implicitly global, due to the absence of aggregate elements. In the following for the above mentioned constructs we will indicate substitutions for them as *global* substitutions.

Example 2.2.2. Consider the rule r_1 from P_1 and the (global) substitution $\sigma_1 = \{X = 1\}$, then $r_1\sigma_1 = a(1) \text{ :- } b(1),\ \ not\ c(1*1)$. Note that σ_1 is well-formed, while for instance, supposing that U_P contained also the symbolic constant *abc*, then a substitution $\sigma_2 = \{X = abc\}$ would not be well-formed.

Now, consider the rule r_2 from P_1 and the global substitution $\sigma_3 = \{Y = 1\}$, then $r_2\sigma_3 = d(1) \text{ :- } \#count\{X:\ a(X)\} \ =\ 1$. If instead, we consider the local substitution $\sigma_4 = \{X = 1\}$, then $r_2\sigma_4 = d(Y) \text{ :- } \#count\{1:\ a(1)\} \ =\ Y$.

The instantiation of an aggregate element e is obtained by considering well-formed local substitutions for e; formally, the instantiation of e consists in the following set of ground aggregate elements:

$$inst(e) = \{e\sigma | \sigma \text{ is a well-formed local substitution for } e\}$$

Inductively, the instantiation of a series of aggregate elements $\{e_1,\ldots,e_n\}$ is provided by the set of aggregate elements reported below:

$$inst(\{e_1,\ldots,e_n\}) = \bigcup_{i=1}^{n}\{e_i\sigma | \sigma \text{ is a well-formed local substitution for } e_i\}$$

A *ground instance* of a term, classical atom, naf-literal, a rule, weak constraint, or query o is obtained in two steps: (i), a well-formed global substitution σ for o is applied to o; (ii), for every aggregate atom $af\{e_1,\ldots,e_n\} \triangleright t$ in $r\sigma$ its aggregate elements $\{e_1,\ldots,e_n\}$ are replaced by $inst(\{e_1,\ldots,e_n\})$.

Example 2.2.3. Consider the aggregate element $e = \{X :\ a(X)\}$ of rule r_2 from P_1, then the instantiation $inst(e)$ of e consists in $inst(e) = \{1 : a(1); 2 : a(2)\}$.

At this point, a ground instance of r_2 is obtained by applying the substitution $\sigma_3 = \{Y = 1\}$, and replacing e with $inst(e)$: $d(1) \text{ :- } \#count\{1 : a(1); 2 : a(2)\} \ = \ 1.$

The arithmetic evaluation of a ground instance g of some term, classical atom, naf-literal, rule, weak constraint or query is obtained by replacing any maximal arithmetic subterm appearing in g 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, strong and weak constraints in P.

Example 2.2.4. Eventually, let us consider P_1, $grnd(P_1)$ consists in:

$$b(1).\ b(2).\ c(1).$$
$$a(1) \text{ :- } b(1),\ \ not\ c(1).$$
$$a(2) \text{ :- } b(2),\ \ not\ c(2).$$
$$d(1) \text{ :- } \#count\{1 : a(1); 2 : a(2)\} \ = \ 1.$$
$$d(2) \text{ :- } \#count\{1 : a(1); 2 : a(2)\} \ = \ 2.$$

Note that the substitution $\{X = 1, X = 2\}$ has been applied to r_1, and the arithmetic terms $(1*1)$ and $(2*1)$ have been evaluated respectively to 1 and 2.

Remark 2.2.1. The instantiation of a program is idempotent: for each program P, $ground(P) = ground(ground(P))$.

2.2.2 INTERPRETATIONS

Once that a ground program is obtained, the truth values of atoms, literals, rules, constraints etc. are properly defined according to interpretations.

Definition 2.2.5 (Herbrand Interpretation)**.** A *(Herbrand) interpretation I* for P is a *consistent* subset of B_P; to this end, for each predicate atom $a \in B_P$, $\{a, -a\} \not\subseteq I$ must hold.

Literals can be either true or false w.r.t. an interpretation. To illustrate how their truth values are determined, as a preliminary step, we need to define a proper total order $\preceq$ on terms in U_P. Several orderings may be defined, in ASP-Core-2 has been adopted the one reported next.

Let t and u be two arithmetically evaluated ground terms, then:

- $t \preceq u$ for integers t and u if $t \leq u$,
- $t \preceq u$ if t is an integer and u is a symbolic constant,
- $t \preceq u$ for symbolic constants t and u with t lexicographically smaller or equal to u,
- $t \preceq u$ if t is a symbolic constant and u is a string constant,
- $t \preceq u$ for string constants t and u with t lexicographically smaller or equal to u,
- $t \preceq u$ if t is a string constant and u is a functional term,
- $t \preceq u$ for functional terms $t = f(t_1, \ldots, t_n)$ and $u = g(u_1, \ldots, u_n)$ if either:
  - $m < n$ or,
  - $m = n$ and $g \not\preceq f$ (f is lexicographically smaller than g) or,
  - $m = n$, $f \preceq g$ and, for any $1 \leq j \leq m$ such that $t_j \not\preceq u_j$, there is some $1 \leq i < j$ such that $t_i \not\preceq u_i$ (i.e., the tuple of terms of t is smaller than or equal to the arguments of u).

At this point, we are ready to properly define literals satisfaction. Let I $I \subseteq B_P$ be a consistent interpretation for P.

The satisfaction of built-in atoms can be easily defined according to the total order $\preceq$, in the intuitive way, as they represent comparisons among terms. A ground classical atom $a \in B_P$ is *true* w.r.t. I if $a \in I$. A positive ground naf-literal a is *true* w.r.t. I if a is a classical or built-in atom that is true w.r.t. I; otherwise, a is false w.r.t. I. A negative ground naf-literal *not* a is true (or false) w.r.t. I if a is false (or true) w.r.t. I.

Given a ground aggregate atom $af\{e_1, \ldots, e_n\} \rhd t$, in order to correctly evaluate its semantics according to its aggregate function, the expression $af\{e_1, \ldots, e_n\}$ has to be mapped to a term, say u. Indeed, aggregate functions can be seen as mappings from set of terms to a term. Let T be the set of terms in $\{e_1, \ldots, e_n\}$, then:

- if $ag = \#count$, then $u = |T|$;
- if $ag = \#sum$, then $u = \sum_{t_i \in T} t_i$ is an integer;
- if $ag = \#max$, then $u = max\{t_i | t_i \in T\}$
- if $ag = \#min$, then $u = min\{t_i | t_i \in T\}$

Essentially, *#count* depends on the cardinality of the set of terms T, *#sum* is evaluated as the sum of the integers in T, while *#max* and *#min* functions strictly rely on the total order $\preceq$ on terms in U_P. In case $T = \emptyset$, the following convention is adopted: $\#max\{\emptyset\} \preceq u$ and $u \preceq \#min\{\emptyset\}$ for every term $u \in U_P$.

Fixed an interpretation some aggregate elements may not contribute to the semantics of an aggregate atom. Intuitively, an interpretation can filter out some aggregate

elements according to their truth values w.r.t. the interpretation itself. More formally, the interpretation I maps a collection E of aggregate elements to the following set of tuples of ground terms:

$$eval(E,I) = \{(t_1,\ldots,t_n) | \{t_1,\ldots,t_n : l_1,\ldots,l_m\} \text{ occurs in } E \text{ and } \{l_1,\ldots,l_m\} \text{ are true w.r.t. } I\}$$

Let $a = af\{e_1,\ldots,e_n\} \rhd t$ be an aggregate atom, a is true (or false) w.r.t. I if $af\{eval(e_1,\ldots,e_n,I)\} \rhd t$. A positive aggregate literal a is true (or false) w.r.t. I if a is an aggregate atom that is true (or false) w.r.t. I. A negative aggregate literal *not* a is true (or false) w.r.t. I if a is false (or true) w.r.t. I.

Let r be a ground rule in $grnd(P)$. The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

Example 2.2.5. Let

$$I_1 = \{b(1), b(2), c(1), a(1), d(1), d(2)\}$$

be an interpretation for $grnd(P_1)$, then:

$b(1).\ b(2).\ c(1).$	are satisfied w.r.t. I_1.
$a(1)$:- $b(1),\ not\ c(1).$	is not satisfied because *not* $c(1)$ is false w.r.t. I_1.
$a(2)$:- $b(2),\ not\ c(2).$	is not satisfied because $a(2)$ is false w.r.t. I_1.
$d(1)$:- $\#count\{1 : a(1); 2 : a(2)\} = 1.$	is satisfied w.r.t. I_1 since $eval(\{1 : a(1); 2 : a(2)\}, I_1) = \{1 : a(1)\}$, and $\#count\{1 : a(1)\} = 1$.
$d(2)$:- $\#count\{1 : a(1); 2 : a(2)\} = 2$	is not satisfied w.r.t. I_1 because of the evaluation reported above.

2.2.3 Answer Sets

Definition 2.2.6 (Model)**.** A *model* for P is an interpretation M for P such that every rule $r \in grnd(P)$ is true w.r.t. M.

Definition 2.2.7 (Minimal Model)**.** A model M for P is *minimal* if no model N for P exists such that N is a proper subset of M. The set of all minimal models for P is denoted by $\mathrm{MM}(P)$.

Example 2.2.6. The interpretation I_1 is not a model for P_1, while the interpretation

$$I_2 = \{b(1), b(2), c(1), a(2), d(1)\}$$

is a model for P_1:

$b(1).\ b(2).\ c(1).$	are satisfied w.r.t. I_2.
$a(1) \text{ :- } b(1),\ \textit{not } c(1).$	is satisfied w.r.t. I_2 because both the body and the head are false.
$a(2) \text{ :- } b(2),\ \textit{not } c(2).$	is satisfied w.r.t. I_2.
$d(1) \text{ :- } \#count\{1 : a(1); 2 : a(2)\} = 1.$	is satisfied w.r.t. I_2 since $eval(\{1 : a(1); 2 : a(2)\}, I_2) = \{2 : a(2)\}$, and $\#count\{2 : a(2)\} = 1$.
$d(2) \text{ :- } \#count\{1 : a(1); 2 : a(2)\} = 2$	is satisfied w.r.t. I_2 because of the evaluation reported above, thus both the body and the head are false.

Moreover, I_2 is also a minimal model for P_1.

Definition 2.2.8 (Reduct)**.** Given a ground program P and an interpretation I, the *reduct* of P w.r.t. I is the subset P^I of P, which is obtained from P by deleting rules in which a body literal is false w.r.t. I.

It is worthwhile noting that the above definition of reduct, proposed in [52], simplifies the original definition of Gelfond-Lifschitz (GL) transform [71], but is fully equivalent to the GL transform for the definition of answer sets [52].

Definition 2.2.9 (Answer Set)**.** [108, 71] Let I be an interpretation for a program P. I is an *answer set* for P if $I \in MM(P^I)$ (i.e., I is a minimal model for the program P^I). The set of all answer sets for P is denoted by $AS(P)$.

Example 2.2.7. Let us consider $grnd(P_1)$ and I_2, then the *reduct* $grnd(P)^{I_2}$ is:

$b(1).\ b(2).\ c(1).$
$a(2) \text{ :- } b(2),\ \textit{not } c(2).$
$d(1) \text{ :- } \#count\{2 : a(2)\} = 1.$

I_2 is a minimal model for $grnd(P_1)^{I_2}$ since no proper subset of I_2 exists such that is a model for it, and thus I_2 an *answer set* for P_1.

In particular, since P_1 is a *not disjunctive* and *stratified* program, I_2 is the unique answer set for P_1, i.e., $AS(P_1) = \{I_2\}$. This type of program admits a unique answer set, which corresponds to its *perfect model* [46].

Furthermore, we distinguish *coherent* and *incoherent* programs: *coherent* programs admit at least one answer set, while *incoherent* programs have no answer sets.

Equivalence of logic programs is a fundamental property also from a practical perspective. For instance, it is common that one looks for an equivalent version Π' of a logic program Π which can be possibly evaluated more efficiently.

Definition 2.2.10 (Equivalent Logic Programs)**.** [70] A logic program Π_1 is said to be *equivalent* to a logic program Π_2 in the sense of the answer set semantics if Π_1 and Π_2 have the same answer sets.

Definition 2.2.11 (Strongly Equivalent Logic Programs)**.** [87] A program Π_1 is said to be *strongly equivalent* to a program Π_2 if for every logic program Π, $\Pi_1 \cup \Pi$ has the same answer sets as $\Pi_2 \cup \Pi$.

In case of weak constraints, answer sets need to be further examined, and classified as *optimal* or not. Intuitively, strong constraints represent conditions that *must* be satisfied, while *weak constraints*, introduced originally in [80, 23], indicate conditions that *should* be satisfied; their semantics involves minimizing the number of violations, thus allowing to easily encode optimization problems.

Optimal answer sets of P are selected among $AS(P)$, according to the following schema. Let I be an interpretation, then:

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

For any integer l, let

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

denotes the sum of integers w over tuples with $w@l$ in $weak(P,I)$.

In other words, for each weak constraints satisfied by I in $grnd(P)$ we sum the weights per level: these numbers represent a sort of penalty paid by I: the lower they are, the higher is the possibility for I, if it represents an answer set, to be optimal.

More formally, we define the notion of *domination* among answer sets as follows. Given an answer set $A \in AS(P)$, it is said *dominated* by another answer set A' if there is some integer l such that $P_l^{A'} < P_l^A$ and $P_{l'}^{A'} < P_{l'}^A$ for all integers $l' > l$. An answer set $A \in AS(P)$ is optimal if there is no $A' \in AS(P)$ such that A is dominated by A'. In general, a coherent program may have one or more optimal answer sets.

Example 2.2.8. Let us consider the following ground program P_2:

$c(1).\ c(2).$
$a(1) \mid b(1)$:- $c(1).$
$a(2) \mid b(2)$:- $c(2).$
:$\sim a(1).\ [1@1]$
:$\sim b(1).\ [1@2]$
:$\sim a(2).\ [2@1]$
:$\sim b(2).\ [2@2]$

The set $AS(P_2)$ consists in:

$as_1:\ c(1).\ c(2).\ a(1).\ b(2)$
$as_2:\ c(1).\ c(2).\ a(1).\ a(2)$
$as_3:\ c(1).\ c(2).\ b(1).\ b(2)$
$as_4:\ c(1).\ c(2).\ b(1).\ a(2)$

For an answer set a, we will represent as $< \{w_1, l_1\}, \dots, \{w_n, l_n\} >$ the sum of weights $w_1, \dots, w_n$ for $l_1, \dots, l_n$, $n \geq 0$. Now, for as_1 we obtain $< \{1,1\}, \{2,2\} >$, for as_2 $< \{3,1\}, \{0,2\} >$, for as_3 $< \{0,1\}, \{3,2\} >$ and finally for as_4 $< \{2,1\}, \{1,2\} >$. Hence, P_2 admit a unique optimal answer set, namely as_2, since it is not dominated by any other answer sets.

2.3 Advanced Constructs

Choice Rules

ASP-Core-2 introduces another type of rule, namely *choice rules*; they represent a syntactic shortcut that can be simulated by the rule types previously introduced. However, choice rules, originally proposed in the system *lparse* [122], can greatly ease the task of encoding a computational problem into ASP.

Definition 2.3.1 (Choice Element)**.** A *choice element* has form: $a : l_1, \dots, l_m$, where a is a *classical atom* and l_i for $i \in \{1, \dots, m\}$ is a naf-literal and $m \geq 0$.

Definition 2.3.2 (Choice Atom)**.** A *choice atom* has form:

$$\{e_1; \dots; e_n\} \rhd u$$

where,

- $n \geq 0$,
- e_i for $i \in \{1, \dots, n\}$ is a choice element,

– $\triangleright \in = \{<, \leq, =, <>, \neq, >, \geq\}$,

– u is a term.

The operator $\triangleright$ and the term u can be omitted whenever $u = 0$ and $\triangleright$ corresponds to $\geq$.

Definition 2.3.3 (Choice Rule)**.** A *choice rule* consists in a rule with a single choice atom a in its head, and literals $b_1, \ldots, b_n$ for $n \geq 0$ in its body:

$$a \text{ :- } b_1, \ldots, b_n.$$

Example 2.3.1. An example of choice rule is $\{a; b; c\} \leq 3$. Intuitively, fixed an interpretation I if the body is satisfied w.r.t. I, as in this case since it is empty, it is sufficient that a possibly empty subset of the choice elements is true w.r.t. I to satisfy the choice atom. Hence, by selecting arbitrarily a, b, c as true or false to satisfy the rule. Thus, the rule can be rewritten in the following rules:

$a \mid$ -a.
$b \mid$ -b.
$c \mid$ -c.

Formally, a choice rule corresponds to the rules, for $i \in \{1, \ldots, n\}$:

$$a_i | \overline{a}_i \text{ :- } l_1, \ldots, l_m, b_1, \ldots, b_k$$

and to the constraints:

$$\text{:- } b_1, \ldots, b_k, \text{not } \#\text{count}\{\overline{a}_1 : a_1, l_{11}, \ldots, l_{m1}; \ldots; \alpha_n : a_n, l_{11}, \ldots, l_{mn}\} \triangleright u$$

where, for each classical atom $s = p(t_1, \ldots, t_n)$, $\overline{s} = p'(1, t_1, \ldots, t_n)$ and $\overline{\text{-}s} = p'(0, t_1, \ldots, t_n)$, with p' an arbitrary predicate associated to p.

2.4 Safety Restrictions

In order to instantiate a (non-ground) rule r, all its variables are considered universally quantified and ranging over the set of all ground terms of the program of which r is part. However, to ensure the semantics not all ground terms need to be considered, but we can restrict the actual domain for variable substitutions, and in turn to limit the size of the produced instantiation. To this end, typically, ASP grounders imposed some conditions on the accepted input, such as *lparse* ω-restrictedness [108, 122], the λ-restrictedness [69] of the first *gringo* releases (up to version 3.0), and *DLV* safety restriction. In the latest years, ASP-Core-2 established safety as the standard restriction for modern ASP systems. Essentially, this is a restriction on variables that guarantees that a rule is logically equivalent to the set of its Herbrand instances. Originally, safety has been introduced in the field of databases, in order to ensure that

queries over databases are independent from the set of constants considered; similarly, in ASP, safety ensures that programs do not depend on the Universe considered.

Let $L = \{l_1, \dots, l_n\}$ for $n \geq 0$ be a set of literals. For a term, literal, rule e we denote as $var'(e) \subseteq var(e)$ the set of variables in e occurring outside of arithmetic terms. The set of safe variables, denoted $Safe(L) \subseteq var(L)$, initially corresponding to $\emptyset$, is computed inductively according to the following schema:

1. for each classical atom $a \in L$, $Safe(L) = Safe(L) \cup var'(l)$;
2. for each built-in atom $a \in L$ of form $v = t$ or $t = v$, where t is a term with $var(t) \subseteq Safe(L)$ and v is a variable, $Safe(L) = Safe(L) \cup v$;
3. for each aggregate atom $a \in L$ of form $af\{e_1, \dots, e_k\} = v$, where v is a variable, $Safe(L) = Safe(L) \cup v$.

We will denote aggregate and built-in atoms of types 2 and 3 as *assignment* atoms. Moreover, let V be the set of variables that an atom a adds to $Safe(L)$ we say that a *binds* V in L, or equivalently that a is a *binder* for V in L.

A naf-literal $l \in L$ is safe if $var(l) \subseteq Safe(L)$. An aggregate element $t_1, \dots, t_p : b_1, \dots, b_q$ appearing in the set E of aggregate elements of an aggregate literal $l \in L$ of form $af\{E\} \rhd t$ is safe if $var_g(b_1, \dots, b_q) \subseteq Safe(L)$ and $var_l(b_1, \dots, b_q) \subseteq Safe(b_1, \dots, b_q)$; consequently, $af\{E\} \rhd t$ is safe if all its aggregate elements are safe, and $var(t) \subseteq Safe(L)$. For a literal $l \in L$, let $VarToSafe(l) = var_g(l) \setminus Safe(l)$. If $VarToSafe \neq \emptyset$ we refers to a set of literals $\{l'_1, \dots, l'_m\} \subseteq L$ $(m \leq n)$ binding the set of variables $VarToSafe(l, l'_1, \dots, l'_m)$ as *saviours* for l. In general, for the same literal several set of saviours might exist.

A non-choice rule r is *safe* if every literal in its body is safe and $var(H(r)) \subseteq Safe(B(r))$. A choice rule r is *safe* if every literal in its body is safe and for every choice element of form $a : N$, where a is a classical atom and N is a set of naf-literals, $var(a) \subseteq Safe(N) \cup Safe(B(r))$. A weak constraint w is *safe* if every literal in its body is safe and $var(W(w)) \subseteq Safe(B(w))$. A query $a?$ is safe if $var(a) \subseteq Safe(a)$. A program P is *safe* if every rule and query composing P is safe.

Example 2.4.1. For instance, the following rules are safe:

$$a(X) \text{ :- } b(X), c(X+1).$$
$$a(Y) \text{ :- } b(X), Y = X+1.$$
$$a(X,Y) \text{ :- } b(X,Y),\ not\ c(X,Y).$$
$$a(Z) \text{ :- } h(Y), Z = \#count\{X : g(X),\ not\ f(X,Y)\}.$$
$$a(X,Y) \text{ :- } Y = \#sum\{W : g(W)\}, X = \#count\{Z : g(Z),$$
$$not\ f(Z,Y)\},\ not\ b(X,Y).$$
$$\text{:- } \#min\{X,S : b(T,X), S = (2*T)-X\} = Y.$$
$$\text{:}\sim \#max\{X : b(X,X+1)\} = Y.\ [1@1, f(Y*Y)]$$
$$\{a(X,Y) : b(Y); b(X) : c(X*3)\} \text{ :- } d(X).$$

These other rules are not safe:

$$a(X) \text{ :- } c(X+1).$$
$$a(Y) \text{ :- } Y = X+1.$$
$$a(X,Y) \text{ :- } \; not \; c(X,Y).$$
$$a(Z) \text{ :- } Z = \#count\{X : g(X), \; not \; f(X,Y)\}.$$
$$a(X,Y) \text{ :- } X = \#count\{Z : g(Z), \; not \; f(Z,Y)\}, \; not \; b(X,Y).$$
$$\text{:- } \#min\{X,S : b(T,X), S+X = (2*T)\} = Y.$$
$$:\sim \#max\{X : b(X,X+1)\} = Y. \; [1@1, f(Y*Z)]$$
$$\{a(X,Y) : b(Y); b(X) : c(X*3)\} \text{ :- } \; not \; d(X).$$

Now, let us consider the following rule r_1:

$$a(X,Y,Z) \text{ :- } b(X,Y), c(Y,Z), Z = Y+1, d(Z+1), \#count\{T : e(T,X)\} < Y.$$

It is easy to see that r_1 is safe. In addition:

- $b(X,Y)$ binds the variables X and Y;
- $c(Y,Z)$ binds the variables Y and Z;
- $Z = Y+1$ binds the variable Z;
- $d(Z+1)$ and $\#count\{T : e(T,X)\} < Y$ do not bind any variable.

For the literals $b(X,Y)$ and $c(Y,Z)$ $VarToSafe = \emptyset$, while for $Z = Y+1$ $VarToSafe = Y$, hence $b(X,Y)$ is a saviour for it, as well as $c(Y,Z)$, because both atoms bind Y. For $d(Z+1)$, $VarToSafe = Z$, and so there are three possible sets of saviours: $\{Z = Y+1, b(X,Y)\}$, $\{Z = Y+1, c(Y,Z)\}$, or simply $\{c(Y,Z)\}$. Note that even if $Z = Y+1$ binds Z the built-in alone is not a saviour, because it needs a saviour for Y. Finally, for $\#count\{T : e(T,X)\} < Y$ possible saviours are: $\{b(X,Y)\}$, $\{b(X,Y), c(Y,Z)\}$.

2.5 Modelling with Answer Set Programming

ASP represents a given computational problem by a logic program whose answer sets correspond to solutions. In other words, the basic idea 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*) of KB correspond to solutions of an input scenario. It is worth noting that, an ASP *Knowledge Base* might have none, one, or many Answer Sets, depending on the problem and the instance at hand.

ASP rules are semantically interpreted according to common sense principles and to the classical *closed-world* assumption/semantics (CWA) of deductive databases; the field of ASP is growing, and several extensions of the basic language have been proposed and used in applications such as *ontology-based query answering* [24, 101] according to the classical *open-world* semantics/assumption (OWA) of first-order logic [5].

The typical computation combines two modules: grounder and solver [75]. As shown in Figure 2.2, the first module takes a program Π and instantiates it by producing a propositional program Π' semantically equivalent to Π but containing no variables; the second module computes answer sets of Π' by adapting and significantly extending SAT solving techniques [84].

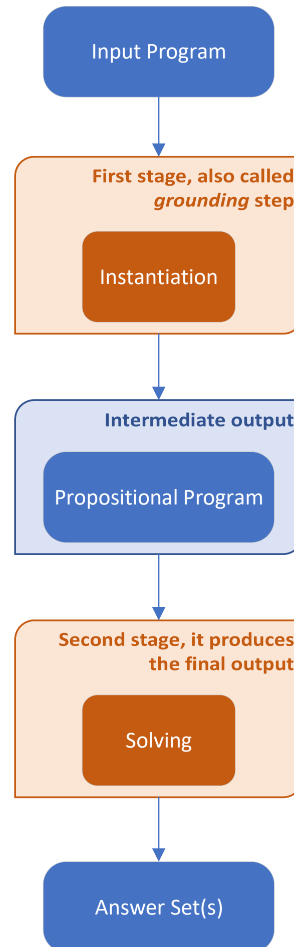


Figure 2.2: ASP simple cycle

As with traditional computer programming, the software engineering by means of ASP technology amounts to a closed loop whose steps, as shown in Figure 2.3, can be roughly classified into: 1) Modelling; 2) Grounding and Solving; 3) Visualizing and 4) Software Engineering.

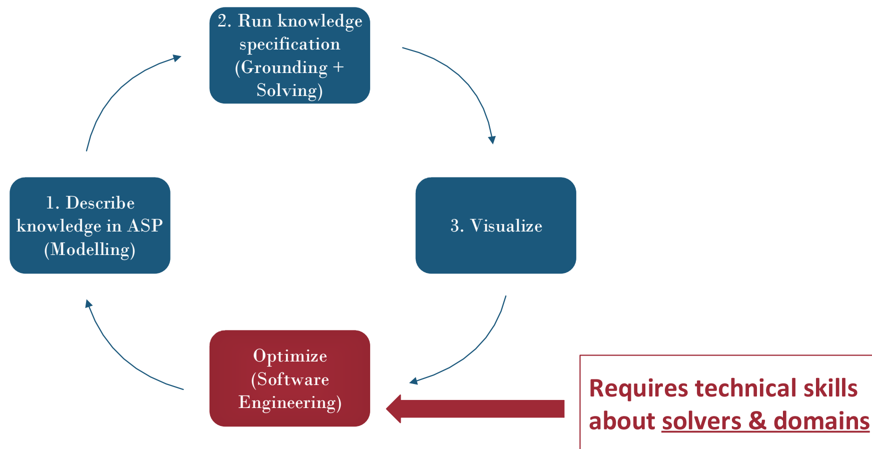


Figure 2.3: ASP development loop

2.6 Knowledge Representation and Reasoning via ASP

Thanks to its high knowledge-modeling power and the availability of reliable, high-performance implementations [28, 66], over time ASP has been increasingly used for solving a wide range of complex problems in several scientific areas [32] such as Artificial Intelligence [10, 112], Knowledge Management [11] and Databases [88, 20, 78, 17].

ASP is a fully declarative language. Moreover, the declarative semantics given to unstratified negation and the closed world assumption make it particularly suitable for representing incomplete knowledge and non-monotonic reasoning. The expressive power is a further key property: theoretically, the introduction of disjunction in rule heads yields to a more expressive language allowing to capture the complexity class Σ_2^P (NP^{NP}). More precisely, ASP programs may express, in a precise mathematical sense, every property of finite structures over a function-free first-order structure that is decidable in nondeterministic polynomial time with an oracle in NP [40].

In this section we will, first, introduce the GCO programming methodology by means of proper examples, then, we will show the capabilities of ASP as a tool for KRR in a real world domain.

2.6.1 Guess, Check and Optimize Paradigm

One of the most important element that distinguishes ASP from other Logic Programming languages is the implementation of the so called "*Guess & Check*" paradigm

which is, sometimes, also called "*Generate & Test*" methodology [85, 47]. The idea behind the *Guess & Check* paradigm is to proceed as follows:

- the *guess* part uses nondeterminism that comes with unstratified negation, or equally well with disjunction in rule heads, to create candidate solutions to a problem (program part $G \subseteq P$), whereas
- the *check* part, with further rules and/or constraints, filters the solution candidate in such a way that they are admissible for the current program instance (program part $C \subseteq P$). This part may also involve auxiliary predicates, if needed.

More in details, the part G defines the search space, and the part C prunes illegal branches.

$$inS(X) \mid outS(X) \text{ :- } node(X). \Big\} \; Guess$$

$$\text{:- } edge(X,Y),\ not\ inS(X),\ not\ inS(Y). \Big\} \; Check$$

An extension and refinement of the *Guess & Check* paradigm is the *Guess/Check/Optimize* (GCO) methodology, which adds one more step to the traditional ones (the optimization part) which can be described as follows:

- the *optimization* part $O \subseteq P$ of the program allows to express a quantitative cost evaluation of solutions by using weak constraints.

It implicitly defines an objective function $f : AS(G \cup C \cup F_I) \rightarrow N$ mapping the answer sets of $G \cup C \cup F_I$ to natural numbers, where F_I represents a set of facts that specify an instance I of some problem $\mathscr{P}$. The semantics of $G \cup C \cup F_I \cup O$ optimizes f by filtering those answer sets having the minimum value; this way, the optimal (least cost) solutions are computed [79].

$$:\sim\ outS(X).\ [1@1,X] \Big\} \; Optimize$$

Sometimes, an additional *define* part is used to model auxiliary predicates and their relationships with other parts of the logic programs. This part usually consists of rules [86].

For further and deeper details about GCO methodology, the reader can refer to [85, 79, 47]

3-COLORABILITY The first ASP use case example showed is the well known *3-colorabiliy* problem.

The graph 3-colorability problem is a decision problem in graph theory which asks if it is possible to assign a color to each vertex of a given graph using at most three

colors, satisfying the condition that every two adjacent vertices have different colors. It has been proved that the graph 3-colorability problem belongs to *NP-complete* class of problems which no polynomial resources solution is found for them yet.

Firstly, we can start defining a problem instance via facts. In this case, we need to model an undirected graph $G = (V,E)$:

- vertices can be encoded as facts of the form $vertex(x)$;
- for edges, facts of type $edge(x,y)$ can be used to express that there is an edge between the vertices x and y.

Then, the following ASP encoding can be used to determine all the admissible sets of solutions, i.e., all the possible ways to color the given graph. For this problem, we do not need an optimize part, indeed, there are not preferences among solutions to be expressed.

% *Guessing Part* (1 *rule*)

$r_1: \quad color(X,red) \mid color(X,green) \mid color(X,blue) \text{ :- } vertex(X).$

% *Checking Part* (1 *rule*)

$r_2: \quad \text{:- } edge(X,Y),\ color(X,C),\ color(Y,C).$

Rule r_1 represents the **guess** stage and states that every node of the graph must be colored of red, green or blue whereas rule r_2 is the **check** stage that prunes the search space forbidding the assignment of the same color to any couple of adjacent nodes.

Notably, thanks to the declarative capability of ASP, when designing the encoding the focus is on how to *model* the problem at hand, rather than on how to actually *solve* it.

By coupling $P_{3_{col}}$ with a set of facts F for *vertex* and *edge*, if the program $P_{3_{col}} \cup F$ is coherent, than each answer set represents an admissible solution. For instance, suppose that:

$$F = \{vertex(1), vertex(2), vertex(3), edge(1,2), edge(1,3), edge(2,3)\}$$

Then the input graph is complete (i.e., every pair of distinct vertices is connected by an edge) and the resulting ground program will be:

$color(1,green) \mid color(1,blue) \mid color(1,red).$
$color(2,green) \mid color(2,blue) \mid color(2,red).$
$color(3,green) \mid color(3,blue) \mid color(3,red).$
$:\!-\ color(2,green), color(1,green).$
$:\!-\ color(2,blue), color(1,blue).$
$:\!-\ color(2,red), color(1,red).$
$:\!-\ color(3,green), color(1,green).$
$:\!-\ color(3,blue), color(1,blue).$
$:\!-\ color(3,red), color(1,red).$
$:\!-\ color(3,green), color(2,green).$
$:\!-\ color(3,blue), color(2,blue).$
$:\!-\ color(3,red), color(2,red).$
$edge(1,2).\ edge(1,3).\ edge(2,3).$
$vertex(1).\ vertex(2).\ vertex(3).$

Finally, the answer sets of $P_{3_{col}} \cup F$ are evaluated during the solving phase and they are:

$as_1 : \{color(1,blue).\ color(2,red).\ color(3,green).\}$
$as_2 : \{color(1,blue).\ color(2,green).\ color(3,red).\}$
$as_3 : \{color(1,red).\ color(2,blue).\ color(3,green).\}$
$as_4 : \{color(1,red).\ color(2,green).\ color(3,blue).\}$
$as_5 : \{color(1,green).\ color(2,red).\ color(3,blue).\}$
$as_6 : \{color(1,green).\ color(2,blue).\ color(3,red).\}$

SUDOKU The classic *Sudoku puzzle*, or simply *Sudoku*, consists of a tableau featuring 81 cells, or positions, arranged in a 9 by 9 grid. The grid is subdivided into 9 “mini-grids” of size 3×3 each containing the numbers 1,...,9 so that no number is repeated in any row, column or mini-grid. When solving a Sudoku, players typically adopt deterministic inference strategies allowing, possibly, to obtain a solution. Several deterministic and nondeterministic strategies are known [31] and can be encoded in ASP.

$\%\ Guessing\ Part\ (1\ rule)$
$r_1:\quad cell(X,Y,N)\ |\ nocell(X,Y,N) \text{ :- } pos(X),\ pos(Y),\ symbol(N).$

$\%\ Checking\ Part\ (6\ rules)$
$r_2:\quad \text{:- } cell(X,Y,N),\ cell(X,Y,N1),\ N1 \neq N.$
$r_3:\quad \text{:- } pos(X), pos(Y),\ not\ assigned(X,Y).$
$r_4:\quad \text{:- } cell(X,Y1,Z),\ cell(X,Y2,Z),\ Y1 \neq Y2.$
$r_5:\quad \text{:- } cell(X1,Y,Z),\ cell(X2,Y,Z),\ X1 \neq X2.$
$r_6:\quad \text{:- } cell(X1,Y1,Z),\ cell(X2,Y2,Z),\ Y1 \neq Y2,\ samesquare(X1,Y1,X2,Y2).$
$r_7:\quad \text{:- } cell(X1,Y1,Z),\ cell(X2,Y2,Z),\ X1 \neq X2,\ samesquare(X1,Y1,X2,Y2).$

$\%\ Auxiliary\ Rules\ (4\ rules)$
$r_8:\quad assigned(X,Y) \text{ :- } cell(X,Y,N).$
$r_9:\quad insquare(Z,X,Y) \text{ :- } div(X,N,R),\ div(Y,N,S),\ Y1 = S*N,\ Z = R+Y1,$
$\qquad pos(Z),\ pos(X),\ pos(Y),\ sizeBlock(N).$
$r_{10}:\quad samesquare(X1,Y1,X2,Y2) \text{ :- } insquare(Z,X1,Y1),\ insquare(Z,X2,Y2).$
$r_{11}:\quad div(X,Y,Z) \text{ :- } XminusDelta = Y*Z,\ X = XminusDelta + Delta,\ Delta < Y,$
$\qquad pos(X),\ pos(Y),\ pos(Z),\ pos(XminusDelta),\ pos(Delta).$

Rule r_1 guesses a number (*symbol*) to assign to each *cell* of the matrix without taking care of any of the game rules, e.g., r_1 could assign to the same cell more than one number. Rules r_2 and r_3 ensure that to each *cell* of the grid is filled with exactly one number. Finally, rules $r_4 - r_7$ check that a number does not occur more than once in the same row, column and mini-grid (also defined as *block*). Note that rule $r_8 - r_{11}$ are just auxiliary rules useful to better encode the program.

N-QUEENS The *N-Queens* is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other, i.e., they are not on the same row, column or diagonal. The N-Queens problem has been shown to be both NP-Complete and #P-Complete [73].

$\%\ Guessing\ Part\ (1\ rule)$
$r_1:\quad \{queen(I,J)\ :\ row(I),\ col(J)\} = n.$

$\%\ Checking\ Part\ (4\ rules)$
$r_2:\quad \text{:- } queen(I,J1),\ queen(I,J2),\ J1 \neq J2.$
$r_3:\quad \text{:- } queen(I1,J),\ queen(I2,J),\ I1 \neq I2.$
$r_4:\quad \text{:- } queen(I,J),\ queen(II,JJ),\ (I,J) \neq (II,JJ),\ I+J = II+JJ.$
$r_5:\quad \text{:- } queen(I,J),\ queen(II,JJ),\ (I,J) \neq (II,JJ),\ I-J = II-JJ.$

Rule r_1 is a *choice rule* and guesses, for each of the n queens, a possible position on the board. Rules $r_2 - r_5$ prevent queens' attacks by adding the necessary integrity constraint. More in details, rule r_2 forbids horizontal attacks (two queens in the same

row); rule r_3 forbids vertical attacks (two queens in the same column); rules r_4 and r_5 forbid diagonal attacks.

MAXIMAL CLIQUE So far, we considered problems without an optimizing part. As example of complete *GCO* program let us consider a NP-hard problem, namely *Maximal Clique*.

> Given an undirected graph $G = (V, E)$, determine a clique C of maximal size in G, i.e., for each other clique C' in G, the number of vertices in C must be larger than or equal to the number of nodes in C'.

We obtain the following encoding $P_{max-clique}$:

$$
\begin{array}{ll}
\% \ Guessing\ Part\ (2\ rule) & \\
r_1: & clique(X) \text{ :- } vertex(X),\ not\ nonClique(X). \\
r_2: & nonClique(X) \text{ :- } vertex(X),\ not\ clique(X). \\
 & \\
\% \ Checking\ Part\ (1\ rules) & \\
r_3: & \text{:- } clique(X), clique(Y), X \neq Y,\ not\ edge(X,Y),\ not\ edge(Y,X). \\
 & \\
\% \ Optimizing\ Part\ (1\ weak\ constraint) & \\
r_4: & \text{:}\sim nonClique(X).\ [1@X]
\end{array}
$$

2.6.2 A REAL WORLD DOMAIN: HYDRAULIC LEAKING

The *Hydraulic Leaking* [33] is a simplified version of the hydraulic system on a space shuttle which consists of a directed graph, G, such that: (i) nodes of this graph are labeled as tanks, jets, or junctions; (ii) every link between two nodes is labeled by a valve; (iii) there are no paths in G between any two tanks; (iv) for every jet there is always a path in G from a tank to this jet.

When modelling this problems, one has to take into account that *Tanks* can be full or empty; *Valves* can be open or closed; some of the valves may be leaking. A state of G is specified by the set of full tanks, the set of open valves, and the set of leaking valves. A node of G is called pressurized in state S if it is a full tank or if there exists a path from some full tank of G to this node such that all the valves on the edges of this path are open. We assume that in a state S a shuttle controller can open a valve $V2$ corresponding to a directed link $\langle N1, N2\rangle$ only if $N1$ is pressurized. (Note, a leaking valve can be opened, and all valves are assumed to be all closed in their initial state.)

A possible encoding P_{hyd} for this problem follows.

$\%$ *Graph nodes*

$r_1:$ $node(N)$:- $jet(N)$.

$r_2:$ $node(N)$:- $junction(N)$.

$r_3:$ $node(N)$:- $tank(N)$.

$\%$ *Lengths of paths to goal bounded by the number of valves, recording also the number*
$\%$ *of leaking valves on the path*

$r_4:$ $dist(J,0,0)$:- $goal(J)$.

$r_5:$ $dist(N,DN,LN)$:- $dist(K,DK,LK)$, $link(N,K,V)$, $DN = DK+1$, $LN = LK+1$, $numValves(NV)$, $DN \leq NV$, $leaking(V)$.

$r_6:$ $dist(N,DN,LK)$:- $dist(K,DK,LK)$, $link(N,K,V)$, $DN = DK+1$, $numValves(NV)$, $DN \leq NV$, $not\ leaking(V)$.

$\%$ *Minimum leaking distance of a node to the goal node*

$r_7:$ $nodemindist(N,DN,LDN)$:- $node(N)$, $minLkDist(N,LDN)$, $minDist(N,DN,LDN)$.

$r_8:$ $minLkDist(N,LDN)$:- $dist(N,Fv1,LDN)$, $not\ existLdnLessThan(N,LDN)$.

$r_9:$ $existLdnLessThan(N,LDN)$:- $dist(N,Fv1,LDN)$, $dist(N,Fv2,LDN1)$, $LDN1 < LDN$.

$r_{10}:$ $minDist(N,DN,LDN)$:- $dist(N,DN,LDN)$, $not\ existDnLessThan(N,DN,LDN)$.

$r_{11}:$ $existDnLessThan(N,DN,LDN)$:- $dist(N,DN,LDN)$, $dist(N,DN1,LDN)$, $DN1 < DN$.

$\%$ *Minimum leaking distance of a full tank to the goal node*

$r_{12}:$ $fulltankmindist(T,DT,LDT)$:- $tank(T)$, $full(T)$, $nodemindist(T,DT,LDT)$.

$\%$ *Fail if no full tank can be reached from the goal*

$r_{13}:$ $reachablefulltankexists$:- $fulltankmindist(T,DT,LDT)$.

$r_{14}:$:- $not\ reachablefulltankexists$.

$\%$ *The full tanks and their minimum leaking distances to the goal node, which have*
$\%$ *the minimum leaking distance over all full tanks*

$r_{15}:$ $bestfulltankldist(T,SD,SDL)$:- $fulltankmindist(T,SD,SDL)$, $not\ existFTLLessThan(SDL)$.

$r_{16}:$ $existFTLLessThan(SDL)$:- $fulltankmindist(Fv1,Fv2,SDL)$, $fulltankmindist(Fv3,Fv4,SDL1)$, $SDL1 < SDL$.

$\%$ $Among\ the\ full\ tanks\ with\ minimum\ leaking\ distance\ choose\ those\ with\ minimum$
$\%$ $distance\ from\ the\ goal$

$r_{17}:$ $bestfulltankdist(T,SD,SDL)$:- $bestfulltankldist(T,SD,SDL),$
$not\ existFTLessThan(SD,\ SDL).$

$r_{18}:$ $existFTLessThan(SD,SDL)$:- $fulltankmindist(Fv1,SD,SDL),$
$fulltankmindist(Fv2,SD1,SDL),\ SD1 < SD.$

$r_{19}:$ $goodtank(T)$:- $bestfulltankdist(T,Fv1,Fv2).$

$\%$ $"Choose"\ the\ lexicographically\ smallest\ good\ tank,\ as\ any\ will\ do$

$r_{20}:$ $nbesttank(T)$:- $goodtank(T),\ goodtank(T1),\ T1 < T.$

$r_{21}:$ $besttank(T)$:- $goodtank(T),\ not\ nbesttank(T).$

$\%$ $Now\ go\ back\ to\ the\ goal,\ starting\ from\ the\ chosen\ tank$

$r_{22}:$ $reached(T,SD,SDL)$:- $bestfulltankdist(T,SD,SDL).$

$\%$ $In\ any\ step,\ choose\ the\ smallest\ valve\ among\ those\ linking\ the\ reached\ node\ of\ distance$
$\%$ $D\ to\ other\ nodes\ of\ distance\ D-1,\ tracking\ also\ the\ leaking\ distance\ (stays\ equal\ for$
$\%$ $non-leaking\ valves,\ decreases\ by\ one\ for\ leaking\ valves)$

$r_{23}:$ $cand(V1,D1,LD1)$:- $reached(N,D,LD),\ LD1 = LD-1,\ D1 = D-1,$
$link(N,N1,V1),\ nodemindist(N1,D1,LD1),\ leaking(V1).$

$r_{24}:$ $cand(V1,D1,LD)$:- $reached(N,D,LD),\ D1 = D-1,\ link(N,N1,V1),$
$nodemindist(N1,D1,LD),\ not\ leaking(V1).$

$r_{25}:$ $nsmallestvalve(V2,D1,LD2)$:- $cand(V1,D1,LD1),\ cand(V2,D1,LD2),\ V1 < V2.$

$r_{26}:$ $smallestvalve(V1,D1,LD1)$:- $cand(V1,D1,LD1),$
$not\ nsmallestvalve(V1,D1,LD1).$

$\%$ $Reverse\ order\ for\ switching\ on\ (moving\ from\ tank\ to\ jet,\ starting\ from\ 0)$

$r_{27}:$ $switchon(V,DN)$:- $smallestvalve(V,D,Fv1),\ bestfulltankdist(Fv2,BD,Fv3),$
$DX = BD-D,\ DN = DX-1.$

$\%$ $Now\ choose\ the\ smallest\ node\ linked\ by\ the\ chosen\ valve\ as\ being\ reached$

$r_{28}:$ $nsmallestnodeforvalve(N1,D1,LD1)$:- $reached(N,D,LD),$
$smallestvalve(V,D1,LD1),\ nodemindist(N1,D1,LD1),\ N1 > N2,\ link(N,N1,V),$
$link(N,N2,V),\ nodemindist(N2,D1,LD1).$

$r_{29}:$ $smallestnodeforvalve(N1,D1,LD1)$:- $smallestvalve(V,D1,LD1),$
$reached(N,D,LD),\ D1 = D-1,\ link(N,N1,V),$
$not\ nsmallestnodeforvalve(N1,D1,LD1).$

$r_{30}:$ $reached(N,D,LD)$:- $smallestnodeforvalve(N,D,LD).$

Consider the following set of facts F_{hyd} for the problem described above:

$tank(t111).\ tank(t112).\ tank(t113).\ jet(j1).\ jet(j2).\ jet(j3).$
$junction(p1).\ junction(p2).\ junction(p3).\ junction(p4).\ junction(p5).\ junction(p6).$
$valve(v1).\ valve(v2).\ valve(v3).\ valve(v4).\ valve(v5).\ valve(v6).\ valve(v7).\ valve(v8).$
$valve(v9).\ valve(v10).\ valve(v11).\ valve(v12).\ valve(v13).$
$numValves(13).$
$link(t111,p1,v1).\ link(p1,p2,v2).\ link(p2,j1,v3).\ link(t112,p3,v4).\ link(p4,p3,v5).$
$link(p4,j2,v6).\ link(t113,p5,v7).\ link(p5,p6,v8).\ link(p6,j3,v9).\ link(p1,p3,v10).$
$link(p3,p5,v11).\ link(p4,p2,v12).\ link(p6,p4,v13).$
$full(t111).\ leaking(v2).\ goal(j1).$

Figure 2.4 shows the resulting graph generated by $P_{hyd} \cup F_{hyd}$. Each *tank, jet* and *junction* represent a node whereas each *valve* represents a conjunction between two nodes. The initial configuration represents the state in which:

- tank $t111$ is the only *full tank*, i.e., it represents the starting point of the graph;
- jet $j1$ is the goal that must be reached;
- valve $v2$ is a leaking valve whereas all the others are closed (and non leaking) valves.

$P_{hyd} \cup F_{hyd}$ produces in output a sequence of atoms of the form $switchon(v,t)$. meaning that at time step t, valve v must be open. The switches should occur at consecutive time steps beginning from 0. A possible solution to this problem when combined with F_{hyd} is composed by the following ordered sequence of valves: $Sol = \{v1,\ v10,\ v11,\ v8,\ v13,\ v12,\ v3\}$ as showed in Figure 2.4.

Note that the *Hydraulic Leaking* problem does not strictly follows the GCO paradigm, in that a solution can be constructed using rules instead of being "guessed". In this respect almost all the above program can be seen as a constructive *define* module [86]; only the constraint r_{14} is used to discard the solution constructed by definition in case a full tank cannot be reached at all from the goal node.

OTHER POSSIBLE ASP APPLICATION CONTEXT Together with the evolution of the new technologies, new issues that require ever faster and more efficient evaluations have also raised as a "side effect". Consequently, a further effort was required from the logic programming community, which has made huge steps forward in adapting its systems and protocols in order to make them compatible with these new needs. In such context, a fundamental role has been assumed by the ASP community.

Thanks to its fully declarative nature, ASP allows to encode a large variety of problems by means of simple and elegant logic programs. Namely, its appealing combination of: (i) a rich yet simple modelling language, (ii) a high-performance solving capacities; (iii) efficient and solid ASP systems for computing answer sets [28, 68]; (iv) the availability of a number of tools allowing to "embed" ASP into imperative code, made this paradigm a popular approach to declarative problem solving in the field

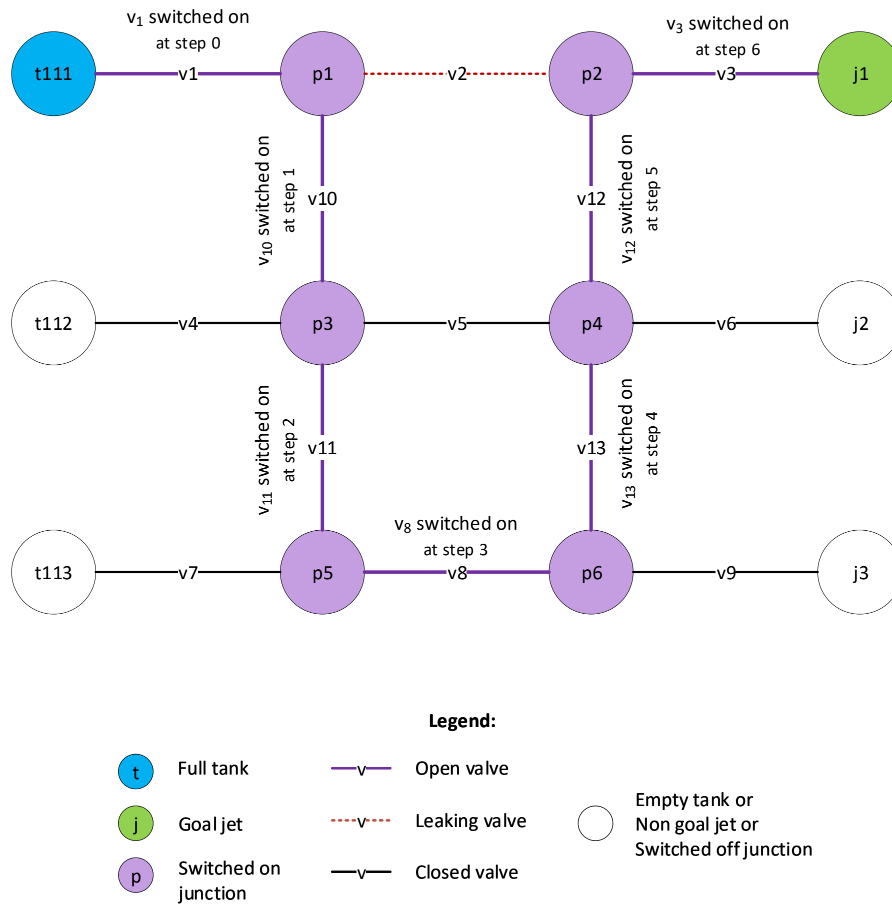


Figure 2.4: A *Hydraulic Leaking* example showed step by step.

of KRR and stimulated the development of a wide range of practical applications relying on ASP both in academia and business. Such applications include, and are not limited to, product configuration, decision support systems for space shuttle flight controllers, large-scale biological network repairs, data-integration and scheduling systems (cfr. [49, 81]).

The flexibility offered by ASP, which has been proven to be a promising programming paradigm thanks to its robust and optimized modeling, reasoning capabilities and inference systems, has led more and more researchers to use it to solve various problems. In particular, given the importance that the so-called "data streams" are acquiring in recent years, most of the efforts of the ASP community have been directed towards the improvement of the already existing ASP techniques, in order to make them able to solve stream reasoning issues.

3 THE STREAM REASONING WORLD: AN OVERVIEW

A *Data streams* is an unbounded sequence of time-varying data elements, i.e., a "continuous" flow of information. In this context, usually most recent values coming from a stream are considered more relevant as they better describe the current state of a dynamic environment.

Stream Reasoning (SR) is the task of continuously deriving conclusions over a *data stream* coming in a real-time, highly dynamic environments in order to support a specific decision process [15, 129]. The ultimate goal of SR is to design a new generation of systems capable of addressing data variety and velocity simultaneously.

3.1 THE IMPORTANCE OF STREAM REASONING

Nowadays, we live in a streaming world in which data show a high variety and are produced faster and faster every day. *Internet of Things* (IoT) devices, sensors for smart cities, smart grid, autonomous automobile systems, medical monitoring, industrial control systems, as well as robotics systems provide data continuously in the form of data streams.

Data streams are very useful to infer new knowledge about the surrounding environment allowing us to answer to an incredible amount of questions, e.g., *could we know in advance if there will be a traffic jam on the highway? Can we reroute travelers on the basis of the forecast? Can we understand shifts of interests of the person behind the computer? Can we discover positive COVID-19 cases analysing data provided by heart rate and* spO_2 *sensors? Or also, can we detect and prevent potential terroristic attacks analysing social media public messages?* Although the information needed to answer these questions is increasingly available on the Semantic Web, there is currently no software system able to (i) compute the answers in a reasonable amount of time; (ii) navigate in a constantly evolving environments at a semantic level and (iii) address multiple new data problems at the same time, such as data volume and data velocity.

Consequently, it becomes necessary to adapt the existing protocols in order to correctly analyse all the information provided by such system. To this end, different real-time approaches have been studied to ensure that a query answer is pushed as soon as it becomes available to the consumers, in order to improve the crucial time requirements of many applications despite the increasing volume of data streams.

Moreover, the increasing availability of mobile or light terminals, characterized by good computational power and by the accessibility to sensors of various kind, paves the way to distributed and widespread artificial intelligence applications. In this respect, it is necessary to have special tools ensuring that these terminals can autonomously perform intelligent tasks, and not only communicate with a centralized "remote intelligence" [93, 92].

In this setting doing research in the field of SR looks extremely promising. Indeed, stream reasoning aims not only at providing languages and tools for data that changes at high rate, but also at introducing advanced reasoning capabilities under a well understood and formalized framework [14, 16].

Moreover, since stream reasoning is a very recent and evolving research field, many different issues still need to be solved. Indeed, currently there are no solutions that allow complex decision-making at the top of data flows and, although existing data stream management systems allow for high-performance *Stream Processor* (SP), they lack dynamic capabilities.

One can argue that LP, with its unique reasoning peculiarities, can help not only to provide a formal representation of the fundamental principles of this field, but can also, in conjunction with other techniques, become a powerful reasoning tool for the problems of this domain.

Several LP formalisms could be theoretically used in stream reasoning fields. However, in the following section, we will principally focus on ASP as the formalism that we chose for our research work in this area thanks of its robust and optimized modeling, reasoning capabilities and inference systems.

3.2 ASP and SR: Current State of the Art and its Limits

A recent survey [132], reveals that more sophisticated and expressive formalisms are taken as one of the key recommendations for further studies in the field of SR. In this respect, ASP has already proven to be a promising paradigm, with its rich modelling and reasoning capabilities, as well as robust and optimized inference systems. Over the last few years, several research lines, that targeted either extending the ASP language for streams [58, 14], or combined ASP systems with stream processing solutions [42, 91] have been pursued.

A move towards reasoning over complex data has been taken by time-decaying logic programs [58], reactive ASP [59], and incremental logic programs [60] supported by the reasoner *clingo*. Describing stream reasoning through logic languages as ASP is an endeavour conducted within the *LARS* system [14]. From the practical point of view, current reasoners have been coupled with SP solutions [42, 91] or dedicated

SR engines have been suggested [16, 12], resulting in an impressive improvement over the recent years, both on the theoretical and practical level.

Existing SR applications pose several challenges of scalability, including heterogeneity of data and intrinsic noise. A general architecture for an ASP-based SR engine has been proposed in [42]. In this work, many functional parameters that affect the execution of SR have been considered (such as frequency of streaming input data or time taken by a reasoner to complete the inference task), but no ways to mitigate the arising scalability problems have been suggested. The topic has been later discussed in [74], where the issue of automatic window size adaptation was singled out. In [106], the scalability challenge has been faced by a so-called Input Dependence Analysis that consists in analysing the connections between the incoming stream data so that it can be separated and interpreted in parallel by a group of reasoners. Additionally, further algorithmic optimisations of current stream reasoning engines are required in order to allow the development of efficient real world applications.

3.2.1 Incremental ASP

A remarkable contribution with respect incremental Answer Set Programming has been introduced with the *iclingo* system [64, 63]. In *iclingo*, a designer can procedurally control how and which parts of a logic program must be incremented, maintained and evaluated among consecutive shots with respect to an iteratively increasing integer parameter t, thus allowing the caching of ground subprograms and of partial answer sets. More in details, the *iclingo* framework is an incremental computing system that extends the functionalities of *clingo* and incorporates both grounding and solving steps in a stateful way, i.e., it has to maintain its previous state for processing the current program slices. Thanks to this new approach, the system is able to (i) reduce efforts by avoiding reproducing previous ground rules during the instantiation step and (ii) reduce redundancy by avoiding the re-evaluation of ground rules during the solving step. In addition to the classical ASP syntax, *iclingo* is able to handle also statements of the following types:

- `#base.` meaning that the subsequent part of a logic program is declared as static, i.e., it is processed only once at the beginning of an incremental computation;
- `#cumulative t_constant.` meaning that in the subsequent part of a logic program, the `t_constant` is replaced in each step with the current step number, and the resulting rules, facts, and integrity constraints are accumulated over a whole incremental computation;
- `#volatile t_constant.` meaning that the subsequent logic program part is local to steps, i.e., all rules, facts, and integrity constraints computed in one step are dismissed before the next incremental step.

The *iclingo* system has been now generalized and integrated in the latest version of the monolithic *clingo* system.

BLOCKSWORLD PROBLEM WITH *iclingo* Blocksworld is a very simple problem where the incremental technique is easily exemplified. There are n blocks stacked on top of each other and it is possible to move only one block at the same time. The goal is to stack the blocks in the inverse order.

The problem needs to be subdivided into three different part, *static*, *volatile* and *cumulative*, in order to be encoded using the *iclingo* statements. Under the `#base` statement all the n blocks need to be declared. Also, the static rules of the problem need to be specified, e.g., what is an admissible location, where each block can be put on and which is the starting position of each block.

$\#base.$

$r_1: \quad block(1..n).$

$r_2: \quad location(X) \text{ :- } block(X).$

$r_3: \quad location(table).$

$r_4: \quad on(X, X+1, 0) \text{ :- } block(X),\ X < n.$

$r_5: \quad on(n, table, 0).$

After the declaration of the static part of the program, it is needed to describe which are the valid moves for the blocks. This is done in the cumulative part, since moves can be done in each step.

$\#cumulative\ k.$

$r_6: \quad 1\{move(X,Y,k) : block(X) : location(Y) : X \neq Y\}1.$

$r_7: \quad on(X,Y,k) \text{ :- } move(X,Y,k).$

$r_8: \quad moved(X,k) \text{ :- } move(X,Y,k).$

$r_9: \quad on(X,Y,k) \text{ :- } on(X,Y,k-1),\ block(X),\ location(Y),$
$\quad\quad not\ moved(X,k).$

$r_{10}: \quad blocked(Y,k) \text{ :- } block(Y),\ on(X,Y,k-1).$

$r_{11}: \quad \text{:- } move(X,Y,k),\ blocked(X,k).$

$r_{12}: \quad \text{:- } move(X,Y,k),\ blocked(Y,k).$

More in detail, a move is of the form *move(X,Y,k)* where X is the block that is moved, Y is the new position of the block and k is the step when the moved is executed. Rule r_6 specify that exactly one move can be executed at each step whereas rules from r_7 to r_{12} are used to update or to confirm the position of the blocks.

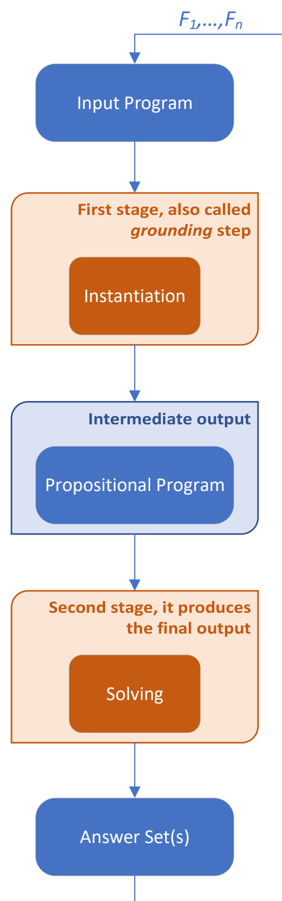


Figure 3.1: Incremental ASP loop

After the definition of the initial setting and of the program's constraints, a query must be formulated.

$$\#volatile\ k.$$
$$r_{13}: \quad :\!-\ not\ on(X+1, X, k),\ block(X), X < n.$$

Rule r_{13} expresses the condition for which the computation has to stop, i.e., all blocks are on top of each other in reversed order. For this reason, it must be specified in the volatile part.

Nonetheless, declarativity and fast-prototyping capabilities are a priority in many development scenarios, such as IoT or videogame industry where designers, who do not have any knowledge of declarative logic programming, just look for easy and off-the-shelf scripting solutions.

Consequently, it is necessary to develop a novel generation of stream reasoning systems in order to overcome the limitations that users are currently experiencing with state-of-the-art reasoners. Such systems should be made capable of supporting advanced query technologies which involve, for instance, types of temporal reasoning at different time granularity, as well as methods for classifying the events that are continually generated at the operational level.

One of the aspects that will be addressed in Chapters 5 and 6 will concern the incremental evaluation and maintaining of ASP logic programs. Our attention will be particularly focused on the usage of the pure ASP semantics in a repeated evaluation setting, since, one of the contributes of this work is precisely to provide a new and original technique for the "incremental" instantiation of ASP logic programs known as "overgrounding". This technique has been implemented in the $\mathcal{I}^2$-*DLV* system, which extends the functionality of $\mathcal{I}$-*DLV*, the *DLV* grounder for ASP and is focused on saving time and memory from one computation to the next one and makes use of a new concept of "ASP loop" as shown in Figure 3.1.

3.2.2 Reactive ASP

Most of the ASP systems are designed for offline usage only, lacking online capacities. However, in the last years, researchers focused their attention on this open problem and, nowadays, there are some new systems able to perform computation for real-time complex applications. These systems are able to operate in online and changing

environments adopting some *reactive solutions*. In this way ASP can be now applied in many new challenging areas, dealing with sensors, agents, (ro)bots, etc.

In this new approach, reasoning is driven by series of data coming at different time steps: the major technical challenge consists in the grounding and solving in view of possible and unknown future events. In fact, in order to avoid redundancy, it is necessary to continuously integrate the new program parts without re-evaluating more than one time the previously treated programs. Moreover, simplifications related to events must be postponed until they become decided. Once a general method is defined, there is room for different application scenarios.

To this end, [36] proposed a new approach where incrementality is meant in a slightly different way: one has a base program P, which is coupled with additional module layers $M_1, \ldots, M_n$. An answer set A of $P \cup M_1$ can be "incremented" with new atoms so to build an answer set A' of $P \cup M_1 \cup M_2$. This approach has been implemented firstly in the online answer set solver *oClingo* and then integrated in the version 5 of *clingo* answer set solver [63]. According to the *oClingo* philosophy, one has to model her/his problem by thinking in terms of "layers" of modules. Therefore, the incremental program constitutes the offline counterpart of an online progression and it is meant to provide a general description of an underlying dynamic system whereas its online counterpart deals with external knowledge acquired asynchronously.

Similarly to what happens with the *iclingo* programs, it is possible to split the input program into three parts via the declarations of the `#base.`, `#cumulative t.` and `#volatile t.` statements, where `t` serves as the parameter. In addition on can use the `#external` directive, in order to state which is the input to the cumulative part provided by future online progressions.

The application-oriented features of *oclingo* also include declarations in the form `#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 with the new external knowledge.

BLOCKSWORLD PROBLEM WITH *oclingo* Just as the *iclingo* solution for the *Blocksworld* puzzle presented in Section 3.2.1, this game can be easily reproduced in its online version and solved making use of *oclingo* system[1]. There are three different input programs file: (*i*) `blocksworld.lp` in which the main rules and constraints of the problem are described, (*ii*) `instance.lp` which contains the initial input

1 The full working example has been taken from https://github.com/grote/oclingo/tree/master/examples/oclingo/blocksworld

and (*iii*) `online.lp` in which the steps, together with the new input data, are specified.

$\%\%\%\%\%\%\%\%$ *blocksworld.lp* $\%\%\%\%\%\%\%\%$

$\#base.$

$r_1: \quad block(B) \text{ :- } nblock(B).$

$r_2: \quad block(B) \text{ :- } rblock(B).$

$\#hide\ block/1.$

$r_3: \quad loc(L) \text{ :- } nblock(L).$

$r_4: \quad loc(table).$

$\#hide\ loc/1.$

$\#cumulative\ t.$

$\#external\ rain/3.$

$r_5: \quad 1\{move(B,L,t) : block(B) : loc(L) : B \neq L\}1 \text{:- } not\ goal(t).$

$r_5: \quad on(B,L,t) \text{ :- } move(B,L,t-1), block(B), loc(L).$

$r_6: \quad on(B,L,t) \text{ :- } on(B,L,t-1), block(B), loc(L),$
$\qquad\qquad not\ move(B,L2,t-1) : loc(L2).$

$r_7: \quad \text{:- } on(B,C,t), move(C,L,t), block(B;C), loc(L), B \neq C.$

$r_8: \quad \text{:- } on(B,C,t), move(D,C,t), block(B;C;D), B \neq C, B \neq D.$

$r_9: \quad \text{:- } on(B,L,t), move(B,L,t), block(B), loc(L).$

$r_{10}: \quad \text{:- } move(B,L,t), not\ rained(B,t), rblock(B), loc(L).$

$r_{11}: \quad on(B,L,t) \text{ :- } rain(B,L,t-1), rblock(B), loc(L).$

$r_{12}: \quad rained(B,t) \text{ :- } rain(B,L,t), rblock(B), loc(L).$

$r_{13}: \quad rained(B,t) \text{ :- } rained(B,t-1), rblock(B).$

$\#hide\ rained/2.$

$r_{14}: \quad goal(t) \text{ :- } on(B,L,t) : goal_on(B,L).$

$\#volatile\ t.$

$r_{15}: \quad \text{:- } not\ goal(t).$

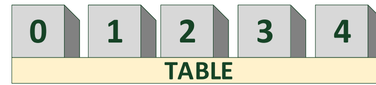
```
%%%%%%%% instance.lp %%%%%%%%
#base.
r1 :    nblock(0..4).

#hide nblock/1.
r2 :    rblock(50..55).

#hide rblock/1.
r3 :    on(B,table,0) :- nblock(B).
r4 :    goal_on(0,table).
r5 :    goal_on(1,0).
r6 :    goal_on(2,1).
r7 :    goal_on(3,2).
r8 :    goal_on(4,3).
#hide goal_on/2.

%%%%%%%% online.lp %%%%%%%%
#step 2.
r1 :    rain(50,3,2).
#endstep.

#stop.
```

With respect to the corresponding *iclingo* model, one can observe the possibility of keeping the number of steps open (so to obtain a continuous computing regime), and the possibility of declaring `#external` predicates. These allow to declare which parts of cumulative sections are fed in input to other parts of the program.

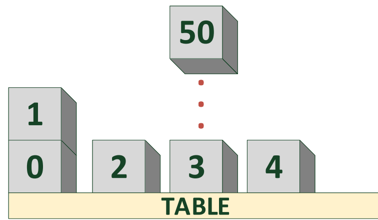
The output of the program after receiving the input from step 2 is reported below. As shown in Figure 3.2a, the initial configuration of the game is the following: all the blocks are located on the *table*. The first *move* to perform (chosen at step 1 in Figure 3.2b) is to move block 1 from *table* to block 0. Once the move has been completed, the configuration of the current state of the game is the same as shown in Figure 3.2c. At this timepoint, the next move chosen by the *oClingo* solver is to move block number 2 on block number 1; moreover, at step 2, a block labeled with number 50 is added on the top of the block 3. Figure 3.2d shows the configuration of the game blocks after the addition of the new block and after performing the last move. In the next steps the algorithm decides to move (i) block 50 on the table; (ii) block 3 on block 2; and, eventually (iii) block 4 on block 3 as shown in Figures 3.2d, 3.2e, 3.2f. Figure 3.2g shows the final configuration when the goal is reached. Indeed, all the

blocks from 0 to 4 are organized one upon the other in the right order (as specified in the `instance.lp` file).

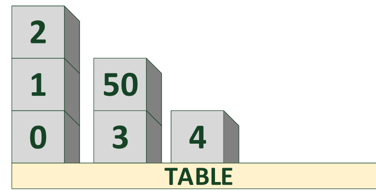
(a) Step 0: initial configuration.



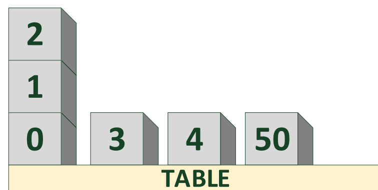
(b) Step 1: move block 1 on 0.



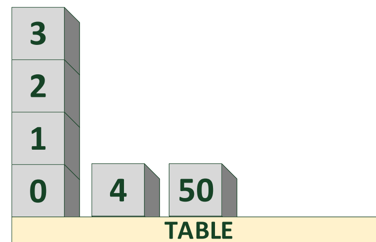
(c) Step 2: move block 2 on 1; a raining block 50 is falling on block 3.



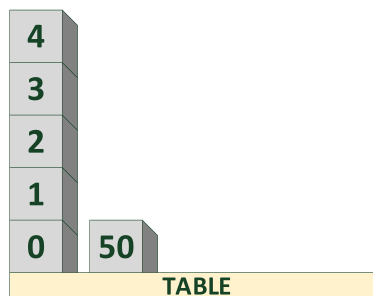
(d) Step 3: move block 50 on table.



(e) Step 4: move block 3 on 2.



(f) Step 5: move block 4 on 3.



(g) Step 6: goal reached.

Figure 3.2: A blocksworld configuration example showed step by step using *oClingo*.

$Got\ input:$
$\#step2.$
$rain(50,3,2).$
$\#endstep.$

$Answer:\ 1$
0. $on(0,table,0)\ on(1,table,0)\ on(2,table,0)\ on(3,table,0)\ on(4,table,0)$
1. $move(1,0,1)\ on(0,table,1)\ on(1,table,1)\ on(2,table,1)\ on(3,table,1)\ on(4,table,1)$
2. $move(2,1,2)\ on(0,table,2)\ on(1,0,2)\ on(2,table,2)\ on(3,table,2)\ on(4,table,2)\ rain(50,3,2)$
3. $move(50,table,3)\ on(0,table,3)\ on(1,0,3)\ on(2,1,3)\ on(3,table,3)\ on(4,table,3)\ on(50,3,3)$
4. $move(3,2,4)\ on(0,table,4)\ on(1,0,4)\ on(2,1,4)\ on(3,table,4)\ on(4,table,4)\ on(50,table,4)$
5. $move(4,3,5)\ on(0,table,5)\ on(1,0,5)\ on(2,1,5)\ on(3,2,5)\ on(4,table,5)\ on(50,table,5)$
6. $goal(6)\ on(0,table,6)\ on(1,0,6)\ on(2,1,6)\ on(3,2,6)\ on(4,3,6)\ on(50,table,6)$

3.3 Incremental and Online Features of Clingo 5

Both the ideas of *iclingo* and *oclingo* are nowadays included in the *clingo* solver version 5, in which volatile, cumulative and base directives have been generalized and can be programmed using *Python* or *Lua* scripts [63]. The #*program* keyword defines now modules in the generic sense (no matter whether they will play the role of a cumulative, base, or volatile section), while the #*external* keyword can be used to define rules that need special treatment with respect to incremental grounding. Using the new directives, the definition of a n-Queens incremental encoding is as shown in Figure 3.3 (from [63]):

$\#show\ queen/2.$

$\#program\ board(n).$
$\#external\ attack(n,1..n,h).$
$\#external\ attack(1..n,n,v).$

$target(n,X,X,n,b,n)$:- $X = 1..n-1.$	$\%\ diagonal\ b$
$target(Y,n-1,n,Y-1,b,n)$:- $Y = 2..n-1.$	$\%\ diagonal\ b$
$target(X,n-1,X+1,n,f,n)$:- $X = 1..n-1.$	$\%\ diagonal\ f$
$target(n-1,Y,n,Y+1,f,n)$:- $Y = 1..n-2.$	$\%\ diagonal\ f$
$target(X,n,X-1,n,h,n)$:- $X = 2..n.$	$\%\ horizontal$
$target(n,Y,n-1,Y,h,n)$:- $Y = 1..n-1.$	$\%\ horizontal$
$target(Y,X,Y,X-1,v,n)$:- $target(X,Y,X-1,Y,h,n).$	$\%\ vertical$

$\{queen(1..n,n); queen(n,1..n-1)\}.$

$attack(X',Y',D) \text{ :- } target(X,Y,X',Y',D,n), queen(X,Y).$
$attack(X',Y',D) \text{ :- } target(X,Y,X',Y',D,n), attack(X,Y,D).$
$\text{:- } target(X,Y,X',Y',D,n), attack(X',Y',D), queen(X',Y').$

$\text{:- } not\ queen(1,n),\ not\ attack(1,n,h).$
$\text{:- } not\ queen(n,1),\ not\ attack(n,1,v).$

```
#script(python)
from clingo import Number

def main(prg):
    n = 0
    parts = []
    for arg in prg.get_const("calls").arguments:
        lower = arg.arguments[0].number
        upper = arg.arguments[1].number
        while n < upper:
            n += 1
            parts.append(("board", [Number(n)]))
            if n >= lower:
                prg.ground(parts)
                parts = []
                print('SIZE {0}'.format(n))
                prg.solve()
#end.
```

Figure 3.3: *clingo* program encoding and Python script for successive n-Queens solving

It must be noted that this approach introduces ample flexibility, in that the processing of *#program* sections and *#external* directives can be customized at will: it however requires a non-negligible knowledge of solver-specific internal algorithms and a radically different modelling approach if compared with the standard one.

3.4 The LARS framework

A conceptual structure for reasoning over streams has been formalized within a **L**ogic-based framework for **A**nalyzing **R**easoning over **S**treams (LARS) [14] which provides a very expressive language based on ASP. Hence, it is an extension of ASP for stream reasoning. Therefore, LARS' importance lies in allowing the formal characterization of various semantic concepts of different Stream Processing/Reasoning formalisms and engines in a common language, making them, for the first time, analytically

comparable. Furthermore, it provides a rule-based formalism with different means to refer to or abstract from time. Between these, it is worth mentioning the introduction of a window operator under the ASP semantics, representing a flexible method to retrieve (finite) parts of a stream (substreams).

More in detail, the extensions provide controls for treating temporal modalities of a formula (the traditional "sometimes", "at", and "always" modal operators) that facilitate asking about the validity of the formula. Modal operators can be used in combination with window operators, thus enabling the possibility to quantify over a desired substream specified by a time interval, or by a given amount of data.

The proposed extensions do not yet include some typical query designer needs [132]. Currently, for example, some of the suggested challenges are usage of aggregation and arithmetic operations.

On the other hand, in order to achieve these objectives it is imperative, first of all, to fill the gaps between theoretical models and practice [132]. The expressivity and complexity of LARS turned out to be intractable in general, thus focus has been given on tractable fragments thereof, or in general, on linguistic fragments for which known incremental techniques can be directly exploited or can be feasibly adapted.

Ticker [16] and *Laser* [12], implement respectively a fragment of the full LARS language. In particular, *Ticker* represents a significant step towards implementation, thanks to its incremental version that uses a truth-maintenance method. The implemented fragment, called *Plain* LARS, allows to express knowledge bases in terms of set of rules, allowing *extended* atoms in their bodies. On the other hand, *Laser* restricts *Plain* LARS to positive and stratified programs. Thanks to this restriction higher performance can be achieved.

3.4.1 Preliminary Definition

Throughout, we distinguish extensional atoms A^E for input data and intensional atoms A^I for derived information. By $A = A^E \cup A^I$, we denote the set of atoms.

Definition 3.4.1 (Stream)**.** A stream $S = (T, \upsilon)$ consists of a *timeline* T, which is a closed non-empty interval in $\mathbb{N}$, and an evaluation function $\upsilon : \mathbb{N} \longmapsto 2^A$. The elements $t \in T$ are called time points.

Definition 3.4.2 (Window Function)**.** Any (computable) function w that returns, given a stream $S = (T, \upsilon)$ and a time point $t \in \mathbb{N}$, a *window* S' of S is called a *window function*.

Time-based window functions, which select all atoms appearing in last n time points, and tuple-based window functions, which select a fixed number of latest tuples are widely used. To this end, we define the tuple size $|S|$ of a stream $S = (T, \upsilon)$ as $|\{(a,t) | t \in T, a \in \upsilon(t)\}|$.

Definition 3.4.3 (Window Operators $\boxplus^w$)**.** A window function w can be accessed in rules by window operators. That is to say, an expression $\boxplus^w \alpha$ has the effect that α is evaluated on the "snapshot" of the data stream delivered by its associated window function w. Within the selected snapshot, LARS allows for controlling the temporal semantics with further modalities.

Definition 3.4.4 (Temporal Modalities)**.** Let $S = (T, \upsilon)$ be a stream, $a \in A$ and $B \subseteq A$ static *background data*. Then, at time point $t \in T$,

- a holds if $a \in \upsilon t$ or $a \in B$
- $\Diamond a$ holds, if a holds at some time point $t' \in T$;
- $\Box a$ holds, if a holds at all time points $t' \in T$; and
- $@_{t'} a$ holds, where $t' \in \mathbb{N}$, if $t' \in T$ and a holds at t'.

Definition 3.4.5 (Extended Atoms)**.** The set A^+ of *extended atoms* e is given by the grammar $e ::= a | @_t a | \boxplus^w @_t a | \boxplus^w \Diamond a | \boxplus^w \Box a$, where $a \in A$ and t is any time point. The expressions $@_t a$ are called $@ - atoms$; $\boxplus^w \star a$, where $\star \in \{@_t, \Diamond, \Box\}$ are *window atoms*.

3.5 TICKER

In this section we briefly report about *Ticker* [16], a stream reasoning engine written in Scala which implements LARS. *Ticker* has two high-level processing methods for a given time point: *append* is adding input signals, and *evaluate* returns the model. *Ticker* is an approach for practical, fully incremental reasoning, i.e., for sliding time- and tuple-based windows. Two implementations of this interface are provided: the first one is defined as the "one-shot solving" and relies on the *clingo* solver; the second one is, instead, based on an ad-hoc extension of the seminal *Justification-based Truth Maintenance System* (JTMS) [43] for a LARS fragment later called *plain LARS*. Plain LARS extends normal logic programs essentially by so-called *extended atoms* for controlling the streaming aspects.

ONE-SHOT SOLVING USING CLINGO. The ASP solver *clingo* is a practical choice for stratified programs, where no ambiguity arises which model to compute. At every time point, resp., at the arrival of a new atom, the static LARS encoding is streamed to the solver and results are parsed as soon as *clingo* reports a model. In case of multiple models, the first one is taken under consideration. Apart from this so-called push-based mode, where a model is prepared after every *append* call, *Ticker* also provide a pull-based mode, where only *evaluate* triggers model computation.

INCREMENTAL EVALUATION BY JTMS. In this strategy, the model is maintained continuously using an ad-hoc implementation of the TMS [43]. A TMS *network* can be seen as logic program P and data structures that reflect a so-called *admissible*

model M for P. Given a rule r, the network is updated such that it represents an admissible model M' for $P \cup \{r\}$, thereby reconsidering the truth value of atoms in M only if they may change due to the network. *Ticker* analogously allows for rule removals, i.e., obtaining an admissible model M' for $P \setminus \{r\}$. More in detail, when new data is streaming in, the system computes the incremental rules, adds them to the TMS network and removes the expired ones, which results in an immediate model update.

3.6 Laser

In this section we report about *Laser* [12], a novel stream reasoning system based on LARS, which extends ASP for stream reasoning. *Laser* focuses on positive and stratified programs, i.e., both stream-stratification and stratified negation. Existing semi-naive evaluation techniques as in Datalog [4] have been extended to deal with the temporal dimension of LARS. In particular, for time-based windows, substitutions for non-ground formulas are annotated with two time markers that express the interval during which the according ground formula is guaranteed to hold. As long as the evaluation time is included in such an interval, the substitution can be retained. Thus, when time progresses, parts of the model may be carried over instead of being recomputed. Moreover, further annotations (i.e., guarantees) might be added incrementally for existing substitutions. In this way, the update process may partially reduce to updating annotations of existing derivations. On the other hand, substitutions may expire and are then removed efficiently due to the lookup of the respective time marker. The approach works similarly for tuple-based windows, under analogous annotations that refer to the global tuple count.

In *Laser*, programs are sets of rules which are constructed on formulae that contain window operators and temporal operators. Thereby, *Laser* has a fully declarative semantics amenable for formal comparison. To address the trade-off between expressiveness and data throughput, *Laser* employs a tractable fragment of LARS that ensures uniqueness of models. Thus, in addition to typical operators and window functions, *Laser* also supports operators such as $\Box$, which enforces the validity over intervals of time points, and @, which is useful to state or retrieve specific time points at which atoms hold. Moreover, *Laser* provide a novel evaluation technique which annotates formulae with two time markers. When a grounding of a formula φ is derived, it is annotated with an interval $[c,h]$ from a *consideration time* c to a *horizon time* h, during which φ is guaranteed to hold. By efficiently propagating and removing these annotations, *Laser* obtain an incremental model update that may avoid many unnecessary re-computations.

The *Laser* incremental procedure consists in continuously grounding and then annotating formulae with two time points that indicate when and for how long formulae hold. *Laser* address two important sources of inefficiency: grounding (including time variables) and model computation. *Laser* deliberately focuses on exploiting purely

sliding windows. The longer a (potential) step size, the less incremental reasoning can be applied. In the extreme case of a tumbling window (i.e., where the window size equals the step size) there is nothing that can be evaluated incrementally. However, as long as the two subsequent windows share some data, the incremental algorithm can be beneficial. For further and deeper details about the system implementation and algorithm, the reader can refer to [12].

4 GAME PROGRAMMING AND ARTIFICIAL INTELLIGENCE IN GAMES

Videogames development is becoming a more and more interesting field for researchers. The relation between videogames and artificial intelligence techniques is a longstanding and still exciting story of reciprocal knowledge exchange. Although it must be observed that the notion of "AI" is understood differently in the academic world with respect to the videogame development industry, this relationship is mutually beneficial. On the one hand, videogames offer unsolved challenges which are as hard as challenges offered by what we can call "serious" applications, yet in a controlled and reproducible setting. This characteristics makes videogames the ideal playground in which to invent, experiment, and test new AI paradigms, methodologies, and techniques. The AI research community frequently uses videogames as a research ground, by organizing ad-hoc competitions [37, 110] and proposing general videogame testbeds for AI [107, 105]. This is not surprising since the implementation of videogames AI requires to include techniques of autonomous decision making, mapping, path-finding, planning, fine-grained motion planning, spatial and temporal reasoning, all of which under strict time constraints. Many disciplines share these similar goals, among which, e.g., robotics [6] and, if one especially looks at requirements on processing large and fast-paced data flows, stream reasoning [41].

On the other hand, the videogame industry, whose world market value is estimated at 200 billion USD as of 2020 [3], customarily looks very greedily at progresses in the field of artificial intelligence, in order to gain practical advancements. One can cite the historical example of the F.E.A.R. game [100], using STRIPS-based planning [133], and many other games using some form of learning and/or hierarchical planning, such as Halo [2] and Black & White [1].

In real time videogames, the artificial players (intelligent agents) and their opponents (*Non-Player Characters* (NPCs)) need AI that is as credible and "real" as possible. To do this, artificially intelligent agents must be able, first of all, to react quickly to changes in the surrounding environment (reactive environment). To give an idea, let's consider that in many videogames the reaction time of NPCs should not exceed the threshold of 40ms, in order to ensure fluidity and responsiveness of the game. It is therefore clear that an intelligent agent is constantly overwhelmed with an immense amount of data, which describe the current state of the surrounding environment and which need to be continuously analysed in order to "act" and "react" intelligently based on the context in which the agent is located.

Therefore, the application of stream reasoning techniques in videogames could be seen as a sort of "gym" where researchers can "train" techniques that could be subsequently used for real-world data analysis.

In this chapter we will present an overview of the world of artificial intelligence employed in the videogames development context with its current applications.

4.1 History of AI and Games

Games and *Artificial Intelligence* have a long history together. Much research on AI for games is concerned with constructing agents for playing games, with or without a learning component.

With the term game AI, we refer to a broad set of algorithms that also includes techniques from control theory, robotics, computer graphics and computer science in general. Most early research on game-playing AI was focused on classic board games, such as *Checkers* and *Chess*. A milestone in AI research in games is the backgammon software named *TD-Gammon* which was developed in 1992 [124, 125].

Later on, during the golden age of arcade videogames, the idea of AI opponents was largely popularized in the form of graduated difficulty levels, distinct movement patterns, and in-game events dependent on the player's input. Modern games often implement existing techniques such as pathfinding and decision trees to generate responsive, adaptive or intelligent behaviors primarily in NPCs similar to human-like intelligence and guide their actions depending of the context. Namely, game AI for NPCs is centered on appearance of intelligence and good gameplay within environment restrictions. Finally, AI is also often used in mechanisms which are not immediately visible to the user, such as data mining and *Procedural Content Generation* (PCG).

It must be noted, that beside a longstanding mutual knowledge exchange process, the videogame industry is an emblematic representative of a real-world application field where practical barriers prevent the wide adoption of AI methodologies, no matter whether these are symbolic/deductive or non-symbolic/inductive. These barriers are, first, of cultural kind: the separation between the videogame industry and academy starts from the different conception of what AI is and means. Also, elevated standards of effectiveness, efficiency and ease of use set a hard to reach cutoff threshold for the introduction of promising paradigms such as ASP in the above contexts; game developers indeed, often prefer mechanical solutions to sophisticate "academic" AI solutions.

Nonetheless, *Artificial Intelligence* has been an integral part of videogames since their inception in the 1950s and, as many industries and corporate voices claim. From then on, the so called videogame AI has come a long way, revolutionizing the way

humans interact with all forms of technology. Indeed, AI in videogames is a distinct subfield and its purpose differs from academic AI. As a matter of facts, it serves to improve the game-player experience rather than machine learning or decision making.

4.2 Artificial Intelligence in Videogames

Whether we know it or not, AI is already part of our everyday life. When we use a smart speaker, when we see a self-driving car and also when we use our Google Photos application to search for images of our cat, we are actually using AI. Also, there is a good chance we have unknowingly used AI while playing a videogame, like for example God of War or Red Dead Redemption 2, but also the simple Pac-Man game.

In reality, over the years AI is getting better and better at playing certain games. For example, beating your computer at chess, when the AI is set at the hardest difficulty level, is pretty much impossible, and the *DeepMind* software can now defeat you even if you are a pro *StarCraft* player. Indeed, DeepMind-based AI use neural network techniques that learn how to play videogames in a fashion similar to that of humans. Some examples are the *AlphaStar* [114, 9] AI and *AlphaGo* [120], two DeepMind-based systems able to efficiently play *StarCraft* and the *Go* game, respectively. Both of them were initially trained using imitation learning to mimic human play, and then improved through a combination of reinforcement learning and self-play.

Through the deep learning and DeepMind revolution, researchers at universities and tech companies have made astounding progresses at giving a machine the means to improve themselves over time. However, this kind of AI is not desirable in commercial games. Indeed, the “canonical” AI employed in digital systems and autonomous vehicles is self-learning and really fast, but it is also really unpredictable. In contrast, when a game programmer designs AI-based NPCs, typically he wants to know what the player will experience and, of course, he wants to ensure a configurable good game experience to the player. For this reason, this kind of AI has to be predictable. By the way, this is not possible, since there is always a very good chance that something unexpected might happen, and it could break the game. Of course, this is a problem for a designer that should create a world able to be completely predictable (no randomicity at all). Also one has to consider that neural-network based AI cannot be easily modified without a re-training step.

What is remarkable is that the AI inside of a videogame has been basically built with the same core set of principles for decades. Namely, the AI you encounter in games today has not really changed that much over the years. Take, for instance, a classic game like Pac-Man. At each time step, the ghosts evaluate where the Pac-Man player is in the map and where he might be going, and then they either chase the player, or they run away from him. Even though it is not exactly ground-breaking

AI, this is still considered a videogame AI, if seen from the videogame community perspective. Indeed, two of the main core components of commercial game AI are usually (i) pathfinding and (ii) finite state machines. The first one is how to get from point A to point B in a simple way and respecting some constraint (i.e. avoiding obstacle, collecting useful objects, etc.), and it is used in all games all the time. Whereas, the second one is a construct where NPCs can be in different states depending on the game scene, and can transition between them. Of course, real AI in commercial games is more complex than that, but those are some of the founding principles. Using these basics, developers have created ever-more realistic game worlds and characters, but such software are not exactly "intelligent" if seen from a philosophical perspective.

Another advancement in the field of game AI has been reached through the use of a new technique known as *Procedural Content Generation* (PCG) allowing to generate game worlds entirely from scratch.

4.3 AI in Games: Frameworks and Competitions

In this section, we present some frameworks and competitions related to videogames and AI conducted in order to better define the current status of the works in this field.

4.3.1 General Video Game AI Competition

One of the most famous frameworks in this field is the well known *General Video Game AI* (GVGAI) [102, 103] framework. GVGAI is a Java framework used for different purposes, such as an AI benchmark to test intelligent agents and for general level generation for any game. This framework, currently used for hosting the *General Video Game Player* (GVGP) competition. can be easily employed to play any game described using the standard *Video Game Description Language* (VGDL) [97]. In fact, all games in GVGAI are expressed via VGDL, a text based description language initially developed by Tom Schaul [116]. In particular, VGDL was built to design 2-dimensional arcade-type games around objects *(sprites)*, one of which represents the player *(avatar)*. The strength point of this language is that it separates the logic into two different components: the *game* and the *level* description, both defined in plain text files, allowing the engine to be independent from them. Thanks to this, it is possible to design a language fit for purpose, which can be concise and expressive enough to be employed for the development of many different games. Moreover, the versatility of VGDL eases the organisation of competitions, especially in the field of game AI community, where it is enough to implement only the reasoning module without the need to focus on the game graphics, which is already provided by the framework itself.

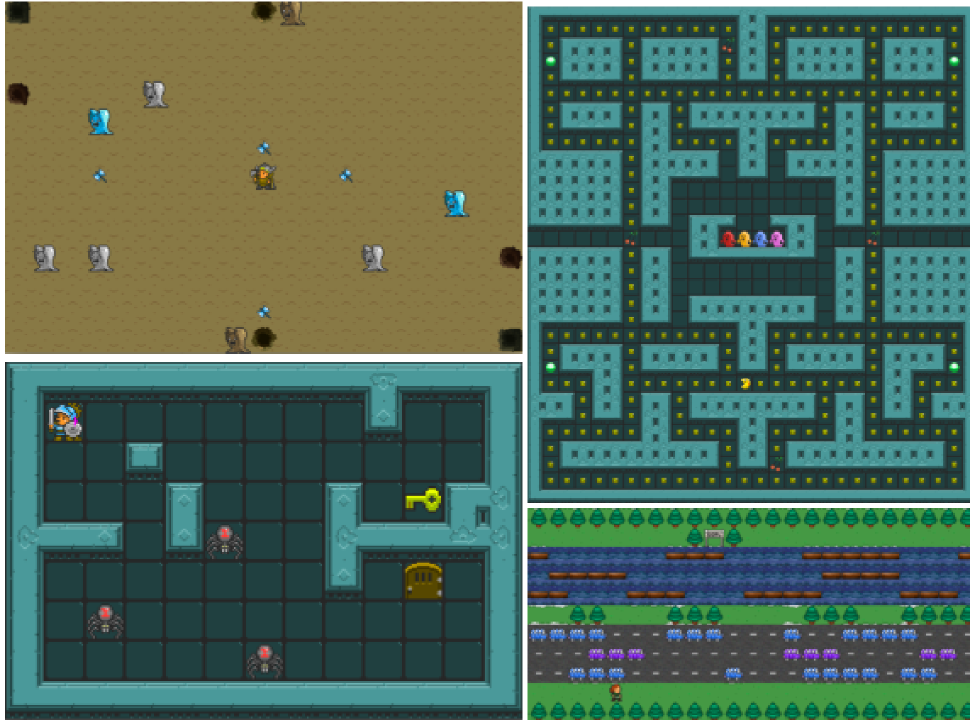


Figure 4.1: A screenshot of some of the test games of the GVGAI competition

One of the most famous AI competitions is the GVGAI competition. It is associated to the GVGAI framework and its purpose is to propose a challenge in which researchers can test their favourite AI methods, with a potentially infinite number of games created using the *Video Game Description Language*. In the GVGAI Competition[1] the participants must submit a Java controller that plays in whatever game that can be supported by the VGDL framework. Each participant must download and use the starter kit provided by the organizers, which contains the VGDL Framework code, some sample controllers that will help the participants to get used to the GVGAI Framework and 30 games, each one of them with 5 levels, to train the controller. The code has to be submitted before the deadline by the participants on an online server, which will be able to compile and execute the submitted controller in any of the three Training sets or in the Validation game set. The results of this submission phase evaluation will not influence the final results of the competition and will be shown on the competition webpage. Finally, all controllers will be executed in the final Test game set, and the results of these evaluations will produce the final competition results.

1 All the information about the competition steps has been taken from the official competition webpage at the following link http://www.gvgai.net/bot_competition.php

4.3.2 Angry Birds AI Competition

Angry Birds is one of the most famous multi-platform games. It has a very simple graphics, gameplay and tasks. The series focuses on multi-colored birds that try to save their eggs from their enemies, which are green-colored pigs. So, the aim of the game is to kill all the green pigs in each level by throwing some different birds at them. By the way, pigs can be protected by various structures. Such structures are usually rectangular, but, in general, they could have several shapes and being of different color, material (wood, ice, stone, etc.), and physical properties like mass, density and friction. Consequently, the player needs to find a good strategy in order to destroy all the structures that protects the pigs and killing, at the same time, all of them. Note that, depending on the mass and density of a structure, pigs could die when it fall down on them.

A player can perform two different actions: (a) selecting the trajectory of the birds which will be threw and (b) tapping on the screen at a specific time t after release in order to activate the birds' optional special power, indeed, different birds have different behaviours and special powers. A game level is solved if, after executing a selected sequence a and b of actions, it leads to a game state that satisfies certain victory conditions (no alive pigs remaining in the game level and minimum point score reached).

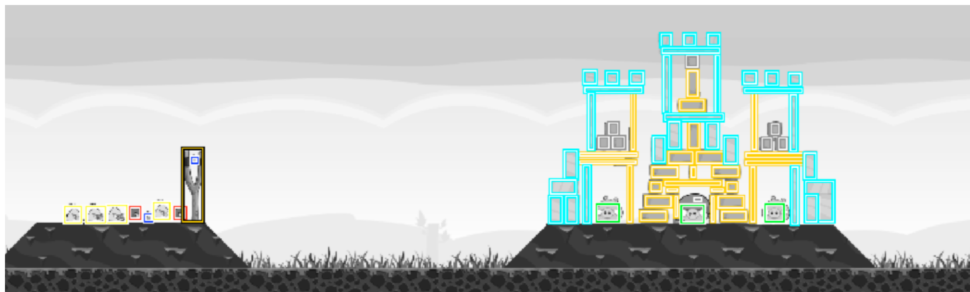


Figure 4.2: A screenshot of the output of the computer vision module of the Angry Birds AI Competition Framework. Screenshot taken from http://aibirds.org/
.

Namely, the score of the player increases every time an object or a pig is destroyed, and spared birds let the player to obtain additional bonuses. Interestingly, after each player's shot, the scenario evolves complying with laws of physics, like for example the crash of object structures and a generally complex interaction of subsequent falls.

In this competition, the main goal is to develop an agent that can successfully play *Angry Birds* even better than the best human players. To do so, the agent has to successfully solve several problems, such as: (i) predict the outcome of all the possible future physical actions without having complete knowledge of the world; (ii) detect and classify known and unknown objects of the scene; (iii) learn properties of the new discovered objects; (iv) select a good action out of infinitely many possible

ones; (v) plan a successful action sequence. while, at the same time, keeping in mind that the game environment behaves according to the laws of physics. This is an essential capability of future AI systems that interact with the physical world.

The *Angry Birds AI competition* (AIBIRDS) provides a simplified and controlled environment for developing and testing these capabilities. The framework is composed of a basic game playing software that includes a computer vision module, a trajectory planning module, and the game interface that works with the *Chrome* version of *Angry Birds*. Figure 4.2 shows a typical output of the computer vision module able to detect and categorise the relevant objects of the game scene. Furthermore, it places a bounding box around them in order to simplify the step of the object detection. It is also possible to create and substitute the current computer vision module with a new one implemented by the participant itself or download a freely available module coming from the participants of the previous competition. Indeed, the organizers highly suggest the developers to make their source code freely available online.

Part II

INCREMENTAL INSTANTIATION VIA OVERGROUNDING

5 OVERGROUNDED PROGRAMS AND EMBEDDINGS

As mentioned in Chapter 3, a wide range of applications require to perform continuous reasoning over event streams. In turn, this requires the repeated execution of a reasoning task over the same fixed logic program, but with changing inputs via multiple "shots" [16, 63]. Recall that the typical workflow of Answer Set Programming (ASP) systems consists in an *instantiation* (or *grounding*) phase and a subsequent *solving* (or *answer sets search*) phase. In the first step, a *grounder* module produces an equivalent propositional program $gr(P \cup F)$ from an input non-ground logic program P and a set of facts F; in the latter step, a *solver* module applies dedicated search techniques on $gr(P \cup F)$ for computing the actual semantics of $P \cup F$ in the form of *answer sets* [75]. We will conceptually model the notion of repeated execution as the task of finding the set of answer sets $AS(P \cup F_i)$ for a sequence of input fact sets $F_1, \dots, F_n$. The computation of a particular value $AS(P \cup F_i)$ for a given F_i, will be called *shot* or *iteration*.

Both the grounding and solving performance is critical when highly paced repeated executions are required. Indeed, only short time windows are allowed between subsequent shots. Therefore, instantiation of such applications could be a significant time bottleneck. We focused on the usage of the pure ASP semantics in a repeated evaluation setting, by proposing methods for maintaining two types of ground programs: (i) "overgrounded programs" and (ii) "overgrounded programs with tailoring". As we will report next, both families of ground programs share useful properties: they can be incrementally updated, and can be reused in combination with deliberately many different sets of inputs. Also the knowledge designer is relieved from the burden of manually controlling the computational procedure.

In this chapter we will focus on the description of the *overgrounded programs*, whereas *overgrounded programs with tailoring* will be discussed in detail in Chapter 6. In Section 5.1 we will briefly introduce an overview of the idea behind overgrounding by means, also, of a simple example; then, a detailed roadmap illustrating how we addressed formal and practical issues concerning overgrounding is reported in 5.2. In Section 5.3 we will introduce some notions and theorems used throughout the rest of the chapter. Finally, in Sections 5.4 and 5.5 we will introduce the notion of *embedding* and we will devise the an incremental grounding strategy relying on the theoretical results illustrated before.

5.1 OVERGROUNDING: AN OVERVIEW

A canonical ASP system works by first *instantiating* a non-ground logic program P over input facts F, obtaining a propositional logic program which we will call $gr(P,F)$ and then computing the corresponding intended models, i.e., the answer sets $AS(gr(P,F))$. Importantly, systems build $gr(P,F)$ as a significantly smaller and refined version of the theoretical instantiation $grnd(P \cup F)$, defined via the Herbrand base (see,e.g., [25]), but preserve semantics, i.e., $AS(gr(P,F)) = AS(P \cup F)$. The choice of the instantiation function gr impacts both computing time and the size of the obtained instantiated program. gr usually maintains a set PT of "possibly true" atoms, initialized as $PT = F$; then, PT is iteratively incremented and used for instantiating only "potentially useful" rules, up to a fixpoint. Strategies for decomposing programs and for rewriting, simplifying and eliminating redundant rules can be of great help in controlling the size of the final instantiation. For an overview of grounding optimization techniques the reader can refer to [64, 27, 35].

Example 5.1.1. We show the overgrounding approach with a simple example. Let us consider the program P_0:

$$r(X,Y) \coloneq e(X,Y),\ not\ ab(X). \quad r(X,Z) \mid s(X,Z) \coloneq e(X,Y),\ r(Y,Z).$$

When taking the set of facts $F_1 = \{e(c,a),\ e(a,b)\}$ into account, there are several ways for building a tailored instantiation of $P_0 \cup F_1$. For instance, one can simply assume F_1 as the initial set of "possibly true" facts, then generate new rules and new possibly true facts by iterating through positive head-body dependencies, obtaining the ground program G_1:

$$r_1 : r(a,b) \coloneq e(a,b),\ not\ ab(a). \quad r_2 : r(c,b) \mid s(c,b) \coloneq e(c,a),\ r(a,b).$$
$$r_3 : r(c,a) \coloneq e(c,a),\ not\ ab(c).$$

Let us assume that, at some point, a subsequent run requires P_0 to be evaluated over a different set of input facts $F_2 = \{e(c,a),\ e(a,d),\ ab(c)\}$. Note that, with respect to F_1, F_2 features the additions $F^+ = \{e(a,d), ab(c)\}$ and the deletions $F^- = \{e(a,b)\}$. Nonetheless, differently from G_1', G_1 can be easily incrementally updated by adding F^+ to the set of possibly true facts, yet preserving semantics. More precisely, one can ground P_0 with respect to the new set of possibly true facts $F_1 \cup F_2$; then, this new information can be propagated and only some additional rules $\Delta G_1 = \{r_4, r_5\}$, must be added, thus obtaining G_2:

$$r_1 : r(a,b) \coloneq e(a,b),\ not\ ab(a). \quad r_2 : r(c,b) \mid s(c,b) \coloneq e(c,a),\ r(a,b).$$
$$r_3 : r(c,a) \coloneq e(c,a),\ not\ ab(c). \quad \mathbf{r_4 : r(c,d) \mid s(c,d) \coloneq e(c,a),\ r(a,d).}$$
$$\mathbf{r_5 : r(a,d) \coloneq e(a,d),\ not\ ab(a).}$$

We now have that G_2 is equivalent to P_0, both when evaluated over F_1 and when evaluated over F_2 as input facts, although only different portions of the whole set

of rules in G_2 can be considered as relevant when considering F_1 or F_2 as inputs, respectively. On the other hand, G'_1 cannot be easily incremented with new rules so to be “compatible” with F_2, because the rule $r(c,a)$:- $e(c,a)$ would cause wrong inferences. Even more interestingly, if a third *shot* of reasoning is requested over input facts $F_3 = \{e(a,d), e(c,a), e(a,b)\}$, we notice that G_2 does not require any further incremental update, as possibly true facts remain unchanged (i.e., $F_1 \cup F_2 = F_1 \cup F_2 \cup F_3$), i.e. the set ΔG_2 containing new incremental additions to G_2 would be empty.

5.2 Towards Overgrounded Programs: roadmap and contributions

It turns out that an instantiation strategy like the one producing G_1 and then G_2 should comply with specific properties that should allow to safely define a sound incremental grounding strategy.

More in detail, we aimed to characterizing a class of ground logic programs equivalent to the theoretical instantiation, called *embedding programs* or, simply, *embeddings* (Section 5.4); they allowed us to introduce a model-theoretic-like notion of ground programs with some desirable properties, and make dealing with formal properties of ground programs cleaner.

Overgrounded programs are built on the notion of embedding. Overgrounded programs (Section 5.5) are treated in terms of series of embedding programs growing monotonically. These have both theoretical and practical impact: for instance, overgrounded programs can be easily generalized to other semantics for logic programming, such as the well-founded semantics; moreover, their incremental growth allows for easily implementing caching policies in many practical scenarios.

We, also, proposed an incremental grounding strategy, allowing to reuse previously grounded programs in consecutive evaluation shots. In particular, while dealing with a specific reasoning task, we maintain a stored ground logic program which grows monotonically from one shot to another; such an *overgrounded program* becomes more and more general while moving from a shot to the next, increasingly adding potentially useful rules. Importantly, intermediate subsequent updates to the ground program are considerably less time-consuming: our technique allows, in a sense, to trade memory for time.

The above framework allows to relieve designers of logic programs from two specific burdens. First, there is no need for procedural control over the incremental evaluation process, as features of the herein proposed incremental framework are almost transparent to designers. Second, there is no need to worry about which parts of logic programs might be more grounding-intensive; also highly general code, even

non-optimized, can benefit from overgrounding. This allows to focus on representing knowledge in a single-encoding/multiple-inputs setting, thus preserving some desirable properties of knowledge representation and reasoning via ASP, namely declarativity and ease of modelling.

5.3 PRELIMINARIES

It is possible to compute a ground program equivalent to $grnd(P) \cup F$ by means of the operator defined next. In this section we will follow [25]: note, however, that the instantiation operator defined here is slightly differently formalized, in that we purposely keep facts explicitly separated from rules.

Definition 5.3.1. *[cf. [25]].* Given a program P and a set of ground atoms S, we define the operator $Inst(P,S)$ as $Inst(P,S) = \{r \in grnd(P)\ s.t.\ B^+(r) \subseteq S\}$. With a slight abuse of notation, for a set R of ground rules we define $Inst(P,R)$ as $Inst(P,Heads(R))$. $Inst^k(P,F)$ is defined as the k-th element of the sequence $Inst^0(P,F) = Inst(P,\emptyset \cup F), \ldots, Inst^k(P,F) = Inst(P,Inst^{k-1}(P,F) \cup F)$.

Intuitively, the notion above formalizes the idea of selecting, among all ground instances of rules in P, those *supported* by a given set S or by the heads of a given set of ground rules R. Given that the sequence above is defined by a monotonic operator it converges to its least fixpoint.

Proposition 5.3.1. [cf. [25]]. *Given a program P and a set of facts F, the sequence $Inst^k(P,F)$, $k \geq 0$ converges to its least fixpoint $Inst^\infty(P,F)$.*

Interestingly, given a program P and a set of facts F, the fixpoint above can be useful for computing a ground program that is equivalent to $grnd(P) \cup F$.

Theorem 5.3.1. [cf. [25]] *For a program P and a set of facts F, $AS(P \cup F) = AS(Inst^\infty(P,F) \cup F)$.*

Definition 5.3.2. (Adapted from [54]) For a non-disjunctive program P and set of facts F, an Herbrand interpretation I for $P \cup F$ is said to be *well-supported* iff there exists a strict well-founded partial order $\prec$ on I such that for any atom $a \in I$ there exists a rule $R \in grnd(P \cup F)$ for which $H(r) = a$, $I \models B(R)$ and for any $b \in B^+(r)$, $b \prec a$.

Theorem 5.3.2. [cf. Th.3.1 of [54]] *For a non-disjunctive logic program P and set of facts F, the well-supported models of $P \cup F$ are exactly the answer sets of $P \cup F$.*

Definition 5.3.3. (Adapted from [82]) It is given a program P and set of facts F. For a given k, a *computation* is a sequence of ground atoms $\{V_n\}_{n \leq k}$ for which it holds:

- $V_0 = F$, and for $i > 0$,
- $V_i = V_{i-1} \cup X$, where $X \subseteq \{a \in H(r) | r \in grnd(P \cup F)$ and $B^+(r) \subseteq V_{i-1}\}$.

Theorem 5.3.3 (Adapted from Theorem 6.23 of [82])**.** *For a program P, set of facts $P \cup F$, let $A \in AS(P \cup F)$. Then $A = V_k$, for some finite k and some computation $\{V_n\}_{n \leq k}$*

The following theorem extends Theorem 3.1 of [54] to the case of logic programs allowing disjunction in the heads; the proof descends as a corollary from the notion of computation and Theorem 6.23 as given in [82].

Proposition 5.3.2. *For a given program P, a set of facts F and an answer set A for P, we can assign to each atom $a \in A$ an integer value $stage(a) = i$ so that stage encodes a strict well-founded partial order over all atoms in A, in such a way that there exists a rule $r \in grnd(P) \cup F$ with (i): $a \in H(r)$, (ii): $A \models B(r)$ and (iii): for any atom $b \in B^+(r)$, $stage(b) < stage(a)$.*

Proof. The *stage* assignment can be built from a computation $\{V_n\} = A$, whose existence is guaranteed by Theorem 5.3.3.

□

5.4 Embedding Programs

We introduce a declarative characterization of a class of equivalent ground programs, called *embeddings*. This notion is useful for proving, given a program P and a set of facts F, whether a ground program G having certain features is equivalent to $P \cup F$, i.e., G is a “correct grounding” that “embeds” $P \cup F$. In a sense, embeddings relate to partial instantiations generated by the *Inst* operator, like Herbrand models relate to supported interpretations generated by the immediate consequence operator. An illustrative comparison between models and embedding programs is reported in Table 5.1.

Definition 5.4.1. [*Embedding.*] For a program P, a set of facts F, a set of ground rules $R \subseteq (grnd(P) \cup F)$, and a rule $r \in (grnd(P) \cup F)$, we say that:

- R embeds the body of r, denoted $R \vdash_b r$, if $\forall a \in B^+(r)\ \exists r' \in R$ s.t. $a \in H(r')$;
- R embeds the head of r, denoted $R \vdash_h r$, if $r \in R$.
- R embeds r, denoted $R \vdash r$, if either: (i) $R \nvdash_b r$, or (ii) $R \vdash_h r$.

Given a logic program P and a set of input facts F, a set of ground rules $R \subseteq grnd(P) \cup F$ is an *embedding program* for $P \cup F$, if $\forall r \in grnd(P) \cup F$, $R \vdash r$.

Let P be a program, F a set of facts, and $r \in grnd(P) \cup F$	
Model-Theoretic Semantics	***Embedding Programs Semantics***
I : Set of *atoms*	S : Set of *rules*
$I \models B(r)$, if $B^+(r) \subseteq I$ and $B^-(r) \cap \{\text{not } a \mid a \in I\} = \emptyset$	$S \vdash_b r$, if $\forall a \in B^+(r)\ \exists r' \in S$ s.t. $a \in H(r')$
$I \models H(r)$, if $H(r) \in I$	$S \vdash_h r$, if $r \in S$
$I \models r$, if either: (i) $I \not\models B(r)$, or (ii) $I \models H(r)$	$S \vdash r$, if either: (i) $S \nvdash_b r$, or (ii) $S \vdash_h r$

Table 5.1: Comparison of the classic model-theoretic semantics for logic programs and the embedding program semantics.

Example 5.4.1. Let us consider the program P_0 of Example 5.1.1 along with the set of facts $F = \{e(a,b), e(b,b)\}$. The ground program below represents $F \cup grnd(P_0)$.

$$
\begin{aligned}
&r_1 : r(a,a) \text{ :- } e(a,a),\ not\ ab(a).\\
&r_2 : r(a,b) \text{ :- } e(a,b),\ not\ ab(a).\\
&r_3 : r(b,a) \text{ :- } e(b,a),\ not\ ab(b).\\
&r_4 : r(b,b) \text{ :- } e(b,b),\ not\ ab(b).\\
&r_5 : r(a,a) \mid s(a,a) \text{ :- } e(a,a),\ r(a,a).\\
&r_6 : r(a,a) \mid s(a,a) \text{ :- } e(a,b),\ r(b,a).\\
&r_7 : r(a,b) \mid s(a,b) \text{ :- } e(a,a),\ r(a,b).\\
&r_8 : r(a,b) \mid s(a,b) \text{ :- } e(a,b),\ r(b,b).\\
&r_9 : r(b,a) \mid s(b,a) \text{ :- } e(b,a),\ r(a,a).\\
&r_{10} : r(b,a) \mid s(b,a) \text{ :- } e(b,b),\ r(b,a).\\
&r_{11} : r(b,b) \mid s(b,b) \text{ :- } e(b,a),\ r(a,b).\\
&r_{12} : r(b,b) \mid s(b,b) \text{ :- } e(b,b),\ r(b,b).\\
&r_{13} : e(a,b).\\
&r_{14} : e(b,b).
\end{aligned}
$$

By definition, $grnd(P_0) \cup F$ is clearly an embedding of $P_0 \cup F$. The set $E_1 = \{r_2, r_4, r_8, r_{11}, r_{12}, r_{13}, r_{14}\}$ is also an embedding of $grnd(P_0) \cup F$. Indeed, for every rule $r \in E_1$, $E_1 \vdash_h r$, and for every rule $r \in (grnd(P_0) \cup F) \setminus E_1$ it holds that $E_1 \nvdash_b r$. The set $E_2 = \{r_4, r_8, r_{12}, r_{13}, r_{14}\}$ is not an embedding of $grnd(P_0) \cup F$; indeed, $E_2 \nvdash r_2$ since $E_2 \vdash_b r_2$ and $E_2 \nvdash_h r_2$.

Embedding programs enjoy a number of interesting properties, some of which are reported next. First, an embedding program is equivalent to $P \cup F$ (Theorem 5.4.1); also, an intersection of embedding programs is an embedding program, similarly to the intersection of models (Proposition 5.4.1).

Definition 5.4.2 (Reduction)**.** [71] For any set I of ground atoms, the *reduct* of P relative to I is the set of rules without negation obtained from P by first dropping every rule such that at least one of the atoms C_i in its body belongs to I, and then dropping the parts *not* $C_1, \dots$,*not* C_n from the bodies of all remaining rules.

Definition 5.4.3 (Reduction)**.** [52] Given a ground DLP^A program P and an interpretation I, let P^I denote the transformed program obtained from P by deleting rules in which a body literal is false w.r.t. I:

$$P^I = \{r | r \in P, \forall b \in B(r) : I \models b\}.$$

Lemma 5.4.1. *Let P be a program, F a set of facts, $\mathscr{E}$ of $P \cup F$ an embedding program, and $A \in AS(P \cup F)$ an answer set. Then, for each $a \in A$ there exists a rule $r_a \in (grnd(P) \cup F)$ s.t. $a \in H(r_a)$ and $r_a \in \mathscr{E}$; thus, $A \subseteq Heads(\mathscr{E})$.*

Proof. By Proposition 5.3.2, each $a \in A$ is associated to an integer value $stage(a)$ and there exists a rule $r_a \in grnd(P) \cup F$, with $a \in H(r_a)$. Note that $r_a \in (grnd(P) \cup F)^A$ since $A \models B(r)$. We now show that $r_a \in \mathscr{E}$ by induction on the *stage* associated to $a \in A$. If $stage(a) = 1$, r_a is such that $B(r_a) = \emptyset$. Hence, since $\mathscr{E}$ is an embedding program for $P \cup F$, and $\mathscr{E} \vdash_b r_a$, it must hold that $r_a \in \mathscr{E}$. Now, (inductive hypothesis) assume that for $stage(a) < j$, $r_a \in \mathscr{E}$. We show that for $stage(a) = j$, $r_a \in \mathscr{E}$. Indeed r_a is such that for each $b \in B^+(r_a)$, $stage(b) < j$, and hence there exists a rule $r_b \in \mathscr{E}$ with $b \in H(r_b)$. Hence $\mathscr{E} \vdash_b r_a$ and thus, since $\mathscr{E}$ is an embedding program for $P \cup F$, $\mathscr{E} \vdash_h r_a$, and $r_a \in \mathscr{E}$. □

Theorem 5.4.1. *(Embedding equivalence). For a program P, a set of facts F and an embedding program $\mathscr{E}$ for $P \cup F$, $AS(grnd(P) \cup F) = AS(\mathscr{E})$.*

Proof. $[AS(grnd(P) \cup F) \subseteq AS(\mathscr{E})]$. Let $A \in AS(grnd(P) \cup F)$. We will show that $(grnd(P) \cup F)^A = \mathscr{E}^A$, thus the statement trivially follows. Indeed, since $\mathscr{E} \subseteq grnd(P) \cup F$, it holds that $\mathscr{E}^A \subseteq (grnd(P) \cup F)^A$. So, if $(grnd(P) \cup F)^A$ and $\mathscr{E}^A$ differ, there must exists a rule $r \in grnd(P) \setminus \mathscr{E}$, and, obviously, such that $r \in (grnd(P) \cup F)^A$. But since $\mathscr{E}$ is an embedding program for $P \cup F$, this means that $\mathscr{E} \nvdash B(r)$. However, $A \models B(r)$, and hence $\forall b \in B^+(r)$ we know that $b \in A$. By Lemma 6.2.2, $A \subseteq Heads(\mathscr{E})$, and then $\forall b \in B^+(r)$ there exists a rule $r' \in \mathscr{E}$ such that $b \in H(r')$, thus leading to a contradiction with $\mathscr{E} \nvdash B(r)$.
$[AS(\mathscr{E}) \subseteq AS(grnd(P) \cup F)]$. Let $A \in AS(\mathscr{E})$. We again show that $\mathscr{E}^A = (grnd(P) \cup F)^A$. Similarly to the case above, since $\mathscr{E}^A \subseteq (grnd(P) \cup F)^A$, there must exists a rule $r \in (grnd(P) \cup F) \setminus \mathscr{E}$, s.t. $\mathscr{E} \nvdash B(r)$. Moreover, $A \models B(r)$, and thus $\forall b \in B^+(r)$ we know that $b \in A$. However, A is an answer set for $\mathscr{E}$; this clearly means that $\forall b \in B^+(r)$ there exists $r' \in \mathscr{E}$ such that $b \in H(r')$, which in turn means that $\mathscr{E} \vdash B(r)$, thus leading to a contradiction. □

Proposition 5.4.1. *(Intersection of embedding programs). Given a logic program P, a set of facts F, $\mathscr{E}_1$ and $\mathscr{E}_2$ embedding programs for $P \cup F$, $\mathscr{E} = \mathscr{E}_1 \cap \mathscr{E}_2$ is an embedding program for $P \cup F$.*

Proof. By contradiction, assume that $\mathscr{E}$ is not an embedding program for $P \cup F$. Then, $\exists r \in (grnd(P) \cup F)$ such that $\mathscr{E} \nvdash r$, that is, $\mathscr{E} \nvdash_h r$ and $\mathscr{E} \vdash_b r$. Since $\mathscr{E} \nvdash_h r$, we have that $r \notin \mathscr{E}$, and hence at least one of the following statements hold: (i) $r \notin \mathscr{E}_1$, (ii) $r \notin \mathscr{E}_2$. Without loss of generality, assume $r \notin \mathscr{E}_1$. By hypothesis, $\mathscr{E}_1$ is an embedding program for $P \cup F$, thus it must hold that $\mathscr{E}_1 \nvdash_b r$. Then, by definition, there exists $b \in B^+(r)$ s.t. $\nexists r' \in \mathscr{E}_1$ with $b \in H(r')$; this implies that such r' cannot exist in $\mathscr{E}$, thus contradicting the fact that $\mathscr{E} \vdash_b r$. □

Example 5.4.2. Consider again the program P_0, the embedding E_1 of the example above and the set of facts $F = \{e(a,b),\ e(b,b)\}$. It is easy to see that the set of rules $E_3 = \{r_2, r_4, r_7, r_8, r_{12}, r_{13}, r_{14}\}$ is also an embedding of $F \cup grnd(P_0)$, and that $E_4 = E_1 \cap E_3 = \{r_2, r_4, r_8, r_{12}, r_{13}, r_{14}\}$ is an embedding for $F \cup grnd(P_0)$ as well.

Finally, the next theoretical results show that $P \cup F$ has a minimal embedding program, corresponding to the intersection of all embedding programs. Also, the minimal embedding program can be computed as the fixpoint of *Inst*, thus establishing a correspondence between embedding programs and the operational semantics of grounders.

Theorem 5.4.2. *Given a program P and a set of facts F, $\mathscr{E} \subseteq grnd(P) \cup F$ is an embedding program for $P \cup F$ iff $\mathscr{E} \supseteq Inst(P, \mathscr{E}) \cup F$.*

Proof. ($\Rightarrow$) Assume that $\mathscr{E}$ is an embedding program for $P \cup F$. By contradiction, assume there is a rule $r \in Inst(P, \mathscr{E}) \cup F$ such that $r \notin \mathscr{E}$. Clearly, $B^+(r) \subseteq Heads(\mathscr{E})$ (by definition of *Inst*). This means that $\mathscr{E} \vdash_b r$, and, since $\mathscr{E} \vdash r$, this implies $\mathscr{E} \vdash_h r$, i.e., $r \in \mathscr{E}$, thus contradicting our assumption.
($\Leftarrow$) Assume that for a set of rules $\mathscr{E}$, $\mathscr{E} \supseteq Inst(P, \mathscr{E}) \cup F$ and, by contradiction, that $\mathscr{E}$ is not an embedding program for $P \cup F$. Then, there must be a rule $r \in grnd(P) \cup F$ such that $\mathscr{E} \nvdash r$. Clearly, $r \notin \mathscr{E}$ (otherwise, we would have $\mathscr{E} \vdash_h r$), and $\mathscr{E} \vdash_b r$. This means that $B^+(r) \subseteq Heads(\mathscr{E})$ and thus $Inst(P, \mathscr{E}) \subseteq \mathscr{E}$ must contain r, contradicting our assumption. □

Theorem 5.4.3 (Lattice-Theoretical Fixpoint Theorem). *[123]Let (i) $\upsilon = \langle A, \leq \rangle$ be a complete lattice, (ii) f be an increasing function on A to A (iii) P be the set of all fixpoints of f. Then the set P is not empty and the system $\langle , \leq \rangle$ is a complete lattice; in particular we have:*

$$\bigcup P = \bigcup E_x[f(x) \geq x] \in P$$

and

$$\bigcap P = \bigcap E_x[f(x) \leq x] \in P.$$

Theorem 5.4.4. *Let* **ES** *be the set of embeddings of $P \cup F$; then,*

$$Inst^{\infty}(P, F) \cup F = \bigcap_{\mathscr{E} \in \mathbf{ES}} \mathscr{E}.$$

Proof. Let define the monotone operator $\overline{Inst}(P,F) = Inst(P,F) \cup F$, and consider the complete lattice of subsets of $grnd(P) \cup F$ under set containment.

The proof then follows by Theorem 5.4.2 and by Knaster-Tarski theorem [123] by observing that

$$Inst^{\infty}(P,F) \cup F = lfp(\overline{Inst}) = inf\{\mathscr{E} \subseteq grnd(P) \cup F \mid \overline{Inst}(P,F) \subseteq \mathscr{E}\} = \bigcap_{\mathscr{E} \in \mathbf{ES}} \mathscr{E}$$

and that $Inst^{\infty}(P,F) \cup F$ is the least fixpoint for $\overline{Inst}(P,F)$. □

Example 5.4.3. Note that $Inst^{\infty}(P_0,F) \cup F$ for the program P_0 of example 5.4.2 coincides with E_4. It can be verified that E_4 corresponds to the intersection of all embedding programs for $F \cup grnd(P_0)$. In order to grasp the intuition, one could note that rules r_{13} and r_{14} belong to all embeddings as they are facts; moreover, rule r_2, r_4, r_8 and r_{12} must belong to all embeddings as well, as their bodies are embedded by any set of rules containing F.

Theorem 5.4.5. *Given a logic program P and a set of facts F, let* **ES** *be the set of embeddings of $P \cup F$. Then*

$$AS(P \cup F) = AS\left(\bigcap\nolimits_{\mathscr{E} \in \mathbf{ES}} \mathscr{E}\right).$$

Proof. The proof follows from Theorem 5.4.1 and Proposition 5.4.1. □

Note that Theorem 5.4.4 combined with Theorem 5.4.1 constitutes an alternative, cleaner proof of Theorem 5.3.1.

5.5 The Overgrounding Technique

In the following we assume we are given a program P and a sequence of sets of facts F_1, ..., F_n; then, let us assume that we need to perform a series of distinct evaluations over a different F_i at each shot. In other words we aim at computing all the sets $AS(P \cup F_1)$, ..., $AS(P \cup F_n)$.

Definition 5.5.1. For an integer k, s.t. $1 \le k \le n$, we define $AF_k = \bigcup_{1 \le i \le k} F_i$ as the sets *accumulated facts* at shot k. Moreover, we define $G_k = Inst^{\infty}(P, AF_k)$ as the *overgrounded program* at shot k.

Each overgrounded program G_k is equivalent to $P \cup F_i$ for $1 \le i \le k$.

Theorem 5.5.1. *The following two statements hold:*
(1): $Inst^{\infty}(P,AF_{k-1}) \subseteq Inst^{\infty}(P,AF_k)$; (2): $AS(Inst^{\infty}(P,AF_k) \cup F_i) = AS(P \cup F_i)$.

Proof. Let $IU = Inst^{\infty}(P,AF_k) \cup F_i$, and for each i, $i \le k$, $IF_i = Inst^{\infty}(P,F_i) \cup F_i$. Recall that each IF_i is an embedding for $P \cup F_i$.

By monotonicity of $Inst$, we have point (1) above and that $IU \supseteq IF_i$ for each $i \leq k$. Each IF_i is clearly an embedding program for $P \cup F_i$ by Theorem 5.4.4. We show that point 2 follows by showing that IU is an embedding program for $P \cup F_i$ and from Theorem 5.4.1.
For a given $i \leq k$, consider a rule $r \in (grnd(P) \cup F_i)$; If $r \in IU$ then $IU \vdash r$. Let us consider the case in which $r \in (grnd(P) \cup F_i) \setminus IU$. Note that $IF_i \vdash r$ and thus $IF_i \nvdash_b r$. Now, either $IU \nvdash_b r$ or $IU \vdash_b r$. In the former case, clearly $IU \vdash r$. In the latter case, we have that $\forall a \in B^+(r)\ a \in Heads(IU)$, and this means that $r \in IU$ by definition of IU, thus contradicting the assumption that $r \notin IU$. Thus IU is an embedding for $P \cup F_i$ for all $i \leq k$. $\square$

We can now devise an incremental grounding strategy relying on the theoretical results illustrated so far. For the sake of simplicity, we illustrate the core of the idea and omit all technical implementation details.

At each shot k, we keep the set of accumulated facts AF_k and the overgrounded program G_k by incrementally updating AF_{k-1} and G_{k-1}. In this setting, $AS(P \cup F_k)$ can be obtained by computing $AS(G_k \cup F_k)$. More in detail, overgrounded programs are managed as follows:

- At shot 1, we set $AF_1 = F_1$, and $G_1 = Inst^{\infty}(P, AF_1)$
- At generic shot k:
  1. we set $AF_k = AF_{k-1} \cup F_k$,
  2. we compute a set of additional ground rules ΔG_k, and
  3. we set $G_k = G_{k-i} \cup \Delta G_k$.

The computation of ΔG_k can be efficiently performed by using an optimized incremental algorithm that takes in input the newly added facts $F_k \setminus AF_{k-1}$ and produces the new rules appearing in $G_k \setminus G_{k-1}$. In our case, we developed a variant of the typical incremental iteration of the semi-naive algorithm [131]. More details on our implementation are given in Chapter 8.

6 INCREMENTAL MAINTENANCE OF OVERGROUNDED PROGRAMS WITH TAILORED SIMPLIFICATIONS

In Chapter 5 we presented an approach to incremental reasoning under the answer set semantics consisting in the use of overgrounding techniques where the grounding step is incrementally performed by maintaining an *overgrounded program* G_P, which is made "compatible" with new input facts by monotonically enlarging it from one shot to another. This approach is attractive since no operational statements are required to incrementally drive the computation and, moreover, the time performance of this technique is promising. Indeed, an overgrounded program, after some update iterations, *converges* to a propositional theory general enough to be reused together with possible future inputs, with no further update required. This virtually eliminates grounding activities in later iterations. However, despite the advantages described so far, the performance of solvers could decrease because of larger input programs.

For this reason, one can think at overcoming the limitations of overgrounding approaches by introducing techniques limiting the number of generated rules and reducing their size by applying known simplification methods for ground logic programs [65, 51]. However, nonobvious technical obstacles prevent a straightforward extension of overgrounding techniques in the above direction: in general, indeed, simplification criteria are applied based on specific inputs. Consider, e.g., if one simplifies a ground program by properly removing atoms which are known to be true in all answer sets at a fixed shot. This, and more sophisticated simplification techniques can however be invalidated in later shots, as, for instance, if a logical assertion is no longer supported by the current input. Thus, diverse general questions arise. One could wonder which properties a ground program should have in order to be "reusable" with a family $\mathscr{F}$ of different inputs; also, it remains open whether a ground program can be modified in a way such that $\mathscr{F}$ can be enlarged with small computational cost, and how.

To this end, we proposed an optimized version of the incremental grounding strategy described in Chapter 5 able to overcome the limitations of the overgrounding approaches previously mentioned.

More in detail, we characterized a class of ground programs equivalent to the theoretical instantiation, called *tailored embeddings*. Tailored embeddings make it simpler to deal with equivalence properties of simplified programs. *Overgrounded programs with tailoring* (OPTs in the following) are series of tailored embeddings that keep a monotonic growth approach, yet permitting simplification techniques.

We proposed a new incremental grounding strategy, allowing to seamlessly adapt and reuse a ground program in consecutive evaluation shots. In particular, OPTs are generated by alternating *desimplification* steps, taking care of restoring previously deleted and reduced rules and *incremental* grounding steps, which add and simplify new rules. The maintained program becomes more and more general (i.e., the family of "compatible" input facts becomes increasingly larger) while moving from a shot to the next, and the update activity becomes progressively lighter. Indeed, both desimplification and incremental operations are monotonic, in that, respectively, one restores formerly deleted rules and formerly deleted literal, whereas the other produces new rules to be added to existing ones.

We implemented the above strategy in the $\mathscr{I}$-*DLV* grounder. We report about the experimental activities we conducted, comparing with other state-of-the-art systems. Results confirm that grounding times blend over iterations in the incremental setting, and that the performance of solvers takes advantage from the reduced size of OPTs with respect to plain overgrounded programs. This different approach is especially beneficial for grounding-intensive logic programs.

The tailored overgrounding approach has a very important theorical advantage: tailored embeddings overcome many limits of the previous notion of embedding. Furthermore, it can be easily generalized to other semantics for logic programming, such as the well-founded semantics.

In this chapter, after overviewing our approach and briefly presenting preliminary notions, we will introduce the notion of tailored embedding and its properties. More in detail, in Section 6.3, we will present OPTs and a maintenance algorithm thereof; then, in Section 8.1 we will show the architecture of the system and, finally, in Section 8.2, we will report about our system and its experimental evaluation.

6.1 Overgrounding with tailored simplifications: an overview

As mentioned, canonical ASP systems work by first *instantiating* a non-ground logic program P over input facts F, obtaining a propositional logic program $gr(P \cup F)$, and then computing the corresponding models, i.e., the set of answer sets $AS(gr(P \cup F))$. Notably, systems build $gr(P \cup F)$ as a significantly smaller and refined version of the theoretical instantiation but preserve semantics, i.e., $AS(gr(P \cup F)) = AS(P \cup F)$.

In this section, we will demonstrate how our approach can be applied to the Example 5.1.1 showed in Chapter 5, allowing to reduce the size and the number of the generated ground rules. Let us consider, again, the program P_0 consisting of rules:

$$r(X,Y) \text{ :- } e(X,Y),\ not\ ab(X). \qquad r(X,Z) \mid s(X,Z) \text{ :- } e(X,Y),\ r(Y,Z).$$

and the set of input facts $F_1 = \{e(c,a),\ e(a,b),\ ab(c)\}$.

Our more "aggressive" grounding strategy could also cut or simplify rules: literals identified as definitely true can be eliminated and rules that cannot fire can be deleted. In the case above, it is easy to see that $ab(a)$ and $ab(c)$ have no chance of being true; hence, removing *not* $ab(a)$ and *not* $ab(c)$ from the rule bodies leads to the generation of G'_1: We can remove facts $e(a,b)$, and $e(c,a)$ from bodies of r_1 and r_2, respectively, and rule r_3 entirely, obtaining TG_1, composed of rules r'_1 and r'_2:

$r'_1 : r(a,b)$:- ~~$e(a,b)$~~, *not* $ab(a)$. $r'_2 : r(c,b) \mid s(c,b)$:- ~~$e(c,a)$~~, $r(a,b)$.
r_3 : ~~$r(c,a)$:- $e(c,a)$, *not* $ab(c)$.~~

Nevertheless, TG_1 can be seen as less re-usable than G_1, as it cannot be easily extended to a program which is equivalent to P_0 with respect to different input facts.

To this end, our proposed technique allows to adapt a simplified ground program TG_x to a new input F_{x+1} by iterating a *desimplification* step and an *incremental* step on TG_x. The desimplified version of TG_x is enriched with new simplified rules added in the incremental step. When $F_2 = \{e(c,a),\ e(a,d)\}$ is provided as input, the desimplification step restores r_3 and reverts r'_1 to r_1:

$r_1 : r(a,b)$:- $\mathbf{e(a,b)}$, *not* $ab(a)$. $r'_2 : r(c,b) \mid s(c,b)$:- ~~$e(c,a)$~~, $r(a,b)$.
$r_3 : \mathbf{r(c,a)}$:- $\mathbf{e(c,a)}$, **not** $\mathbf{ab(c)}$.

Moreover, in the incremental step two new rules r_4 and r_5 are added depending on the new fact $e(a,d)$:

$\mathbf{r_4 : r(c,d) \mid s(c,d)}$:- $\mathbf{e(c,a)}$, $\mathbf{r(a,d)}$. $\mathbf{r_5 : r(a,d)}$:- $\mathbf{e(a,d)}$, **not** $\mathbf{ab(a)}$.

Then, r_4 can be simplified by removing $e(c,a)$, whereas $e(a,d)$ can be deleted from r_5, obtaining:

$\mathbf{r'_4 : r(c,d) \mid s(c,d)}$:- ~~$\mathbf{e(c,a)}$~~, $\mathbf{r(a,d)}$. $\mathbf{r'_5 : r(a,d)}$:- ~~$\mathbf{e(a,d)}$~~, **not** $\mathbf{ab(a)}$.

Thus, $TG_2 = \{r_1, r'_2, r_3, r'_4, r'_5\}$, whereas r'_4 and r'_5 were not formerly present in TG_1. We have now that TG_2 is equivalent to P_0 when evaluated over F_2 as input facts, whereas TG_2 with input facts F_1 would cause wrong inferences. Indeed, r'_5 is simplified according to new facts belonging to F_2 but not to F_1. Nevertheless, if $F_3 = F_1$ is submitted as input, the desimplification step would generate TG_3 from TG_2 by reverting the rule r'_5 to r_5.

One might notice that TG_3 is built assuming $F_1 \cup F_2 \cup F_3$ as possibly true facts, and assuming $F_1 \cap F_2 \cap F_3$ as certainly true facts. Intuitively, the last element of the series $\{TG_i\}$ is the one embracing the larger family of inputs for which "compatibility" is guaranteed, and requiring lesser update work in later iterations: in case a fourth shot is requested over input facts $F_4 = \{e(a,d),\ e(c,a),\ e(a,b)\}$, the desimplification step will leave TG_3 unaltered and the incremental step will not generate new rules; this happens since possibly true facts and persistent facts are left unchanged. Thus, $TG_4 = TG_3$. We illustrate next how programs like TG_1, TG_2 and TG_3 are related to each other, and which formal requirements are necessary to develop a correct incremental grounding strategy.

6.2 TAILORED EMBEDDINGS

We are given a logic program P and a set of input facts F. We will now work with possibly simplified versions of rules of $grnd(P)$. For a rule $r \in grnd(P) \cup F$, a *simplified rule* (or *simplified version*) s of r is a rule annotated with the set $B^*(s)$, where $B^*(s) = B(r) \setminus B(s)$. The rule r is denoted as $hom(s)$, i.e., r is the *homologous rule* of s belonging to the theoretical grounding whose body is obtainable as $B(hom(s)) = B(s) \cup B^*(s)$. For a set of simplified rules S we define $hom(S) = \{r \in grnd(P) \cup F \mid \exists s \in S$ and $hom(s) = r\}$. A rule $q \in grnd(P) \cup F$ is regarded as a simplified rule with $B^*(q) = \emptyset$ and $hom(q) = q$. Similarly, a set $Q \subseteq grnd(P) \cup F$ is regarded as a set of simplified rules with $hom(Q) = Q$.

Note that sets of simplified rules are not comparable under set inclusion, although one can consider, e.g., the set of rules $a{:-}b$ and $c{:-}d$ as a "somewhat smaller" subset of the set composed by rules $a{:-}b,c$ and $c{:-}d$. We thus appropriately generalize set inclusion and set intersection to sets of simplified rules. Given two sets of simplified rules S and R, we say that S is a *simplified subset* of R ($S \sqsubseteq R$) if for each $s \in S$ there is a rule $r \in R$ s.t. $B(s) \subseteq B(r)$ and $hom(s) = hom(r)$. The *simplified intersection* $R \sqcap Q$ of two set of simplified rules R and Q is:

$$\begin{aligned} R \sqcap Q = \{t \mid r \in R, q \in Q, B(t) = B(r) \cap B(q), \\ B^*(t) = B^*(r) \cup B^*(q), \\ hom(t) = hom(r) = hom(q)\}. \end{aligned}$$

Example 6.2.1. Let us consider rules $r_1, \ldots, r_5$ and their primed versions as mentioned in Section 6.1. We assume that for each $i = 1 \ldots 5$, $hom(r'_i) = r_i$. Given $TG_3 = \{r_1, r'_2, r_3, r'_4, r_5\}$ and $TG_1 = \{r'_1, r'_2\}$, we have that $TG_1 \sqsubseteq TG_3$. For $T_0 = \{r_1, r'_2\}$, the intersection $T_0 \sqcap TG_1$ is instead the set $\{r'_1, r'_2\}$.

Definition 6.2.1. Given two sets of simplified rules R and Q, we define $Simpl(R, Q)$ as an operator working on each simplified rule $r \in R$ according to the following simplification types.

1. r is removed from R, if there is a literal $not\ a \in B^-(hom(r))$ s.t. $a \in Facts(Q)$;
2. r is removed from R if there is a atom $a \in B^+(hom(r))$ and $a \notin Heads(Q)$;
3. we move from $B(r)$ to $B^*(r)$ each atom $a \in B^+(hom(r))$ s.t. $a \in Facts(Q)$.

Intuitively, the types 1 and 3 depend on atoms which are assumed to be certainly true in any answer set; and the type 2 depends on atoms that are assumed to be certainly false in any answer set. With slight abuse of notation, we define $Simpl(R)$ as $Simpl(R, R)$. A number k of repeated applications of $Simpl$ to the same set R is denoted as $Simpl^k(R)$. Note that, for $k \geq 1$, $Simpl^{k+1}(R) \sqsubseteq Simpl^k(R)$: we denote the fixed point reached in finitely many steps by the sequence of values $Simpl^k(R)$

as $Simpl^{\infty}(R)$. We trivially extend the operators $\vdash$, $\vdash_h$ and $\vdash_b$ (as given in Definition 5.4.1) for a set of simplified rules on the left-hand side and for simplified rules on the right-hand side. As given next, a *tailored embedding* is a set of simplified rules which extends the notion of embedding by including the possibility of using simplification operations in order to obtain smaller, yet correct, ground programs.

Definition 6.2.2. [*Tailored embedding*] Given a set of simplified rules R and a rule $r \in grnd(P) \cup F$, we say that R *tailors* r ($R \Vdash r$) if at least one of the following holds:

1. $R \vdash r$;
2. there exists a simplified rule $s \in R$ such that $hom(s) = r$, $R \vdash_h s$ and $R \vdash_h a$ for each atom $a \in (B^+(r) \setminus B^+(s))$;
3. there is a literal *not* $a \in B^-(r)$ and $R \vdash a$.

A set of simplified rules E is a *tailored embedding* for $P \cup F$, if $\forall r \in grnd(P) \cup F$, $E \Vdash r$.

Intuitively, a ground rule r is tailored according to the new operator "$\Vdash$" either if it is embedded by R or, otherwise, there are in R the conditions for applying one of the possible simplification types to r. Note that $R \vdash b$ is meant as a shortcut for $R \vdash \{b\text{:-}\emptyset\}$.

Informally speaking, the notion of tailored embedding overcomes the one of embedding: although remarkably simple and useful, the latter notion lacks the fact that there are many other classes of optimized ground programs which are of interest, both theoretically and practically. In other words, embeddings do not properly formalize smaller, yet equivalence-preserving, ground programs produced by actual grounders. The new conditions describe equivalence-preserving ground programs in which a ground rule can be shortened or deleted at all, provided it is "tailored". This narrows the gap between the formalization [30] and real applications.

Example 6.2.2. Let us consider the ground program $TG_3 = \{r_1, r'_2, r_3, r'_4, r_5\}$ as shown in Section 6.1, and the set of facts F_1. $T = TG_3 \cup F_1$ is a tailored embedding since: *(a)* T tailors r_1, r_3, r_5 and all the facts in F_1, since T embeds all such rules; *(b)* $T \Vdash r_2$ since r'_2 is a simplified version of r_2 for which $T \vdash_h r'_2$ and $T \vdash e(c,a)$; *(c)* similarly, $T \Vdash r_4$ since $T \vdash_h r'_4$ and $T \vdash e(c,a)$. Any other rule $r \in grnd(P_0) \cup F_1$ is trivially tailored since it holds that $T \nvdash_b r$ thus implying $T \vdash r$.

Tailored embeddings enjoy a number of interesting properties: an embedding is a tailored embedding (Proposition 6.2.4); a tailored embedding is equivalent to $P \cup F$ (Theorem 6.2.1); also, a simplified intersection of tailored embeddings is a tailored embedding (Proposition 6.2.5); and, importantly, the intersection of all tailored embeddings represents the least tailored embedding under simplified set inclusion and corresponds to an iterative, operational construction made using the *Simpl* and *Inst* operators (Theorem 6.2.2 and Corollary 1).

Proposition 6.2.1. *For a ground logic program* G *and* $A \in AS(G)$*,* $A \subseteq Heads(G)$*.*

Proposition 6.2.2. *For a ground logic program* G *and* $A \in AS(G)$*,* $Facts(G) \subseteq A$*.*

The following Proposition re-adapts Theorem 6.22, as given in [82].

Proposition 6.2.3. For a given tailored embedding E for $P \cup F$, let us consider the superset $Facts(E)$ of F. We can assign to each atom $a \in Facts(E)$ an integer value $stage'(a) = i$ so that $stage'$ represents a strict well-founded partial order over all atoms in $Facts(E)$, in such a way that $hom(a)$ is structured as follows: $\{a\} = H(hom(a))$, $\forall b \in B^+(hom(a)), b \in Facts(E)$ and $stage'(a) < stage'(b)$.

Lemma 6.2.1. For a tailored embedding E of $P \cup F$ and an answer set $A \in AS(P \cup F)$, $Facts(E) \subseteq A$.

Proof. The proof is given by induction on the function $stage'$ applied to $Facts(E)$ as given by Proposition 6.2.3. W.l.o.g. we assign $stage'(a) = 1$ to each atom $a \in Facts(E) \cap Facts(P \cup F)$. These atoms clearly belong to A. We assume then that for each $a \in Facts(E)$ with $stage'(a) < j$ we know that $a \in A$, and show that this implies that for all $a \in Facts(E)$ for which $stage'(a) = j$, $a \in A$ as well. By Proposition 6.2.3 and the inductive hypothesis, we have that $hom(a)$ is such that each $b \in B^+(hom(a))$ belongs to A, and thus $A \models B^+(hom(a))$. Finally, the Lemma is proven by observing that $B^-(hom(a)) = \emptyset$. □

Lemma 6.2.2. Given a tailored embedding E of $P \cup F$ and an answer set $A \in AS(P \cup F)$. Then, for each $a \in A$ there exists a rule $r_a \in E$ s.t. $hom(r_a) \in (grnd(P) \cup F)$; thus, $A \subseteq Heads(E)$.

Proof. By Proposition 5.3.2, each $a \in A$ is associated to an integer value $stage(a)$ and there exists a rule $r_a \in grnd(P) \cup F$, with $a \in H(r_a)$. Note that $r_a \in (grnd(P) \cup F)^A$ since $A \models B(r)$. We now show that $r_a \in hom(E)$ by induction on the *stage* associated to $a \in A$. W.l.o.g. we can assign $stage(a) = 1$, whenever r_a is such that $H(r_a) = \{a\}$, $B^+(r) = \emptyset$ and for all b s.t. *not* $b \in B^-(r)$ we have that $b \notin A$. When $stage(a) = 1$, since E is a tailored embedding for $P \cup F$, it is easy to check that $E \vdash_b r_a$, and thus $r_a \in E$.

Now, (inductive hypothesis) assume that for $stage(a) < j$, $r_a \in hom(E)$. We show that for $stage(a) = j$, $r_a \in hom(E)$. Given the above, r_a is such that for each $b \in B^+(r_a)$, $stage(b) < j$, and hence there exists a rule $r_b \in E$ with $b \in H(r_b)$. Hence $E \vdash_b r_a$. Since E is a tailored embedding for $P \cup F$, and thus $E \Vdash r_a$, we have that at least one of cases in Definition 6.2.2 apply. In particular:

- If the case 1 applies, $E \vdash_b r_a$ implies $r_a \in E$;
- If the case 2 applies, there is clearly a rule $r'_a \in E$ for which $hom(r'_a) = hom(r_a)$;
- If the case 3 applies, it must be that for some *not* $b \in B^-(r_a)$, $b \in Facts(E)$. But on the other hand $A \models B(r)$ and thus $b \notin A$. However, by Lemma 6.2.1, $b \in A$, which leads to a contradiction.

We conclude that either the case 1 or the case 2, i.e., $a \in Heads(E)$. □

Proposition 6.2.4. *An embedding E for $P \cup F$ is a tailored embedding for $P \cup F$.*

Proof. We observe that given an embedding E for $P \cup F$, for each $r \in grnd(P) \cup F$, we have that $E \vdash r$. Then $E \Vdash r$ by the case 1 of Definition 6.2.2. □

Example 6.2.3. Let us consider again the example of Section 6.1 and in particular, the ground program $G_1 = \{r_1, r_2, r_3\} \cup F_1$. G_1 is an embedding for $P_0 \cup F_1$ as each rule $r \in grnd(P_0) \cup F_1$ is embedded by G_1: if $r \in G_1$, $G_1 \vdash_h r$, whereas if $r \in \{grnd(P_0) \cup F_1 \setminus G_1\}$, $G_1 \vdash_b r$. Also, G_1 is a tailored embedding for $P_0 \cup F_1$ because for each rule $r \in grnd(P_0) \cup F_1$ we have that $G_1 \vdash r$, since we can apply the case 1 of Definition 6.2.2. Note that for $P_0 \cup F_1$, the ground program $TG_1 = \{r'_1, r'_2\} \cup F_1$ is a tailored embedding but it cannot be an embedding. Indeed, $TG_3 \nvdash r_3$ since $TG_3 \vdash_b r_3$ and $TG_3 \nvdash_h r_3$ and according to Definition 5.4.1, TG_3 had to embed all rules in $grnd(P_0) \cup F_1$ to be an embedding.

Theorem 6.2.1. *[Equivalence]. Given a tailored embedding E for $P \cup F$, then $AS(grnd(P) \cup F) = AS(E)$.*

Proof. We show that a given set of atoms A is in $AS(grnd(P) \cup F)$ iff A is in $AS(E)$. We split the proof in two parts.

$[AS(grnd(P) \cup F) \subseteq AS(E)]$. Let $A \in AS(grnd(P) \cup F)$. We show that A is a minimal model of E^A. First we show that A is model for E^A. Indeed, let us assume that there is a simplified rule $r \in E^A$ such that $A \not\models r$. This can happen only if $A \models B(r)$ but $A \not\models H(r)$. However, $A \models hom(r)$, which implies that either:

- $A \not\models B(hom(r))$. This implies that $\exists l \in B(hom(r))$ such that $A \not\models l$. We have an immediate contradiction if $l \in B(r)$. Contradiction arises also if $l \notin B(r)$: indeed, since E is a tailored embedding, l does not appear in $B(r)$ only if the case 2 of Definition 6.2.2 has been applied, which means that a simplification of type 3 has been applied. By Lemma 6.2.1, we have a contradiction, since $Facts(E) \subseteq A$ implies that l must appear in A.
- $A \models B(hom(r))$ and thus $A \models H(hom(r))$. Note that $A \models H(hom(r))$ implies that $A \models H(r)$ since $H(r) = H(hom(r))$.

We then show that there is no smaller model for E^A. Let us assume that there exist a set A', $A' \subset A$, which is a model for E^A and thus A is not a minimal model of E^A. Note that A is a minimal model of $(grnd(P) \cup F)^A$ and thus there must exist $r \in (grnd(P) \cup F)^A$ for which $A' \not\models r$. Such a rule can be either such that:

(a) There is no $s \in E$ s.t. $r = hom(s)$;

(b) There is $s \in E$ s.t. $r = hom(s)$ and $s \notin E^A$;

(c) There exists $s \in E^A$ s.t. $r = hom(s)$.

We show that r cannot fall in the cases *(a)* and *(b)*, while the case *(c)* implies that A' cannot be a model for E^A.

Case (a). Since $r \in (grnd(P) \cup F)^A$ it is the case that $A \models H(r)$ and $A \models B(r)$. However, by Lemma 6.2.2, we know that $A \subseteq Heads(E)$. Also, we know that $E \Vdash r$, but there is no $s \in E$ for which $r = hom(s)$. This means that r should be tailored either by the case 1 or 3 of Definition 6.2.2.

If the case 1 applies, then it must be that $E \nvdash_b r$ or $E \vdash_h r$. On the one hand, Lemma 6.2.2 forces us to conclude that $E \vdash_b r$; thus it should be the case that $E \vdash_h r$, which contradicts the assumption that r has no $s \in E$ for which $r = hom(s)$. If the case 3 applies, there exists *not* $a \in B^-(r)$ s.t. $a \in Facts(E)$. But by Lemma 6.2.1, $Facts(E) \subseteq A$, which contradicts $A \models B(r)$.

Case (b). In this case, there is $s \in E$ s.t. $r = hom(s)$ and $s \notin E^A$; Again, note that $A \models H(r)$ and $A \models B(r)$, which in turn implies that $A \models B(s)$ and $A \models H(s)$. Thus this case cannot apply, since it turns out that $s \in E^A$.

Case (c). Since the two cases above cannot apply, r must fall in this latter case. Since $A' \not\models r$, it must be the case that $A' \not\models H(r)$ and $A' \models B(r)$. Note that $B(s) \subseteq B(r)$ and $H(s) = H(r)$. Thus, $A' \not\models H(s)$ and $A' \models B(s)$, which implies $A' \not\models s$. We conclude that A' cannot be a model for E^A.

[$\mathbf{AS(E) \subseteq AS(grnd(P) \cup F)}$]. Let $A \in AS(E)$. We first show that $A \models (grnd(P) \cup F)$. We split all the rules of $(grnd(P) \cup F)^A$ in two disjoint sets: $hom(E^A)$ and $(grnd(P) \cup F) \setminus hom(E^A)$.

For a rule $r \in hom(E^A)$, let s be such that $r = hom(s)$. We have that $A \models B(s)$ and $A \models H(s)$. Since $H(r) = H(s)$, this latter implies that $A \models H(r)$. Let us examine each literal $l \in B^*(s)$, which has been eliminated by the case 2 of Definition 6.2.2. We have that $l \in Facts(E)$, and thus $A \models l$ by Proposition 6.2.2. We can thus conclude that $A \models B(r)$ and, consequently $A \models r$.

Let us now consider a rule $r \in (grnd(P) \cup F) \setminus hom(E^A)$. We show that $A \models r$. Let us assume, by contradiction that $A \not\models r$, i.e., $A \models B(r)$ but $A \not\models H(r)$. We distinguish two subcases: either $r \in hom(E)$, or $r \notin hom(E)$.

If $r \in hom(E)$, we let s be such that $r = hom(s)$. Since $r \notin hom(E^A)$, we have that $s \notin E^A$, i.e., $A \not\models B(s)$ which implies $A \not\models B(r)$, which contradicts the assumption that $A \not\models r$. If $r \notin hom(E)$, we however know that $E \Vdash r$. This can be either because of the case 1 or the case 3 of Definition 6.2.2.

If r falls in the case 1, we have that $hom(r) = r$ and either $E \nvdash_b r$ or $E \vdash_h r$. Since $r \notin hom(E)$, it must then be that $E \nvdash_b r$, i.e., there exists at least one $a \in B^+(r)$ s.t. it does not exist a rule $r' \in E$ for which $E \vdash_h r'$. Then, $a \notin A$ by proposition 6.2.1 and thus $A \not\models B(r)$.

If r falls in the case 3, we have that there exist a literal $not\ a \in B^-(r)$ for which $a \in Facts(E)$. Clearly, by proposition 6.2.2, $a \in A$, and thus $A \not\models B(r)$.

Thus $A \models (grnd(P) \cup F)^A$. We know show that A is a minimal model for $(grnd(P) \cup F)^A$. Let us consider a set $A' \subset A$ and assume that $A' \models (grnd(P) \cup F)^A$. However, we know that A is a minimal model of E^A and thus $A' \not\models E^A$. We can show that this implies that $A' \not\models (grnd(P) \cup F)^A$. Indeed if $A' \not\models E^A$, then there exists a rule $r \in E^A$ for which $A' \not\models r$. This, as we will show implies that $A' \not\models hom(r)$ (note that it can be easily shown that $hom(r)$ belongs to $(grnd(P) \cup F)^A$).

Indeed, we know that $A' \models B(r)$ and $A' \not\models H(r)$. Also it is the case that $A' \models B(r), B^*(r)$. In fact if we assume, by contradiction, that $A' \not\models B(r), B^*(r)$ there should exist a literal $l \in B^*(r)$ for which $A' \not\models l$. l cannot be negative since $A \models l$ and $A' \subset A$. If l is positive, the case 2 of Definition 6.2.2 tells us that $l \in Facts(E)$, i.e., $Facts(E) \not\subset A'$, which in turn implies that A' cannot be a model for $(grnd(P) \cup F)^A$. This concludes the proof. □

Proposition 6.2.5. *[Intersection]. Given two tailored embeddings $\mathscr{E}_1$ and $\mathscr{E}_2$ for $P \cup F$, $\mathscr{E}_1 \sqcap \mathscr{E}_2$ is a tailored embedding for $P \cup F$.*

Proof. Let $E = \mathscr{E}_1 \sqcap \mathscr{E}_2$, and let us consider a rule $r \in (grnd(P) \cup F)$. We show that $E \Vdash r$. Preliminarily, we observe two facts which hold by definition of simplified intersection and by the fact that both E_1 and E_2 are tailored embeddings. We are given a literal a and one of E_1 or E_2 (w.l.o.g., we choose E_1):

(a) $a \in Facts(\mathscr{E}_1)$ implies that $a \in Facts(E)$.

(b) $a \notin Heads(\mathscr{E}_1)$ implies that $a \notin Heads(E)$;

By contradiction, let us assume that $E \nVdash r$, and we split the proof in two parts, depending on whether $r \in hom(E)$ or whether $r \notin hom(E)$.

($r \in hom(E)$). This implies that there are rules $s \in E_1$, $q \in E_2$ and $t \in E$ such that $r = hom(s) = hom(q) = hom(t)$. Note that, for each (positive) literal $l \in B^*(t)$, the case 2 of Definition 6.2.2 can be applied i.e., $l \in Facts(E_1)$ or $l \in Facts(E_2)$ which implies $l \in Facts(E)$ (Fact (a) above);

($r \notin hom(E)$). In this case we have that either $r \notin hom(\mathscr{E}_1)$ or $r \notin hom(\mathscr{E}_2)$. W.l.o.g. we assume $r \notin hom(\mathscr{E}_1)$. By Definition 6.2.2, this can be the case if either

1. $\mathscr{E}_1 \nvdash_h r$ because there exists $a \in B^+(r)$ and $a \notin Heads(\mathscr{E}_1)$. Note that Fact (b) implies that $a \notin Heads(E)$, hence $E \Vdash r$.

2. $\mathscr{E}_1 \nvdash_b r$; this implies that $E \nvdash_b r$ hence $E \Vdash r$;

3. $\mathscr{E}_1 \nvdash_h r$ because there exists $not\ a \in B^-(r)$, and $a \in Facts(\mathscr{E}_1)$. Note that Fact (a) implies that $a \in Facts(E)$, hence $E \Vdash r$.

□

Theorem 6.2.2. *Let* $\mathbf{TE}$ *be the set of tailored embeddings of* $P \cup F$ *and* $\mathscr{E} = Inst^{\infty}(P,F) \cup F$*. Then,*

$$Simpl^{\infty}(\mathscr{E}) = \bigsqcap_{T \in \mathbf{TE}} T.$$

Proof. Let $\mathscr{T} = \bigsqcap_{T \in \mathbf{TE}} T$. By Proposition 2 we notice that $\mathscr{E} = \bigsqcap_{E \in \mathbf{ES}} E$. The single argument operator *Simpl* is both deflationary and monotone when restricted over the complete lattice $(L, \sqsubseteq)$, where $L = \{T \in \mathbf{TE} | T \sqsubseteq \mathscr{E}\}$: thus, the iterative sequence $E^0 = sup_{\sqsubseteq}(L) = \mathscr{E}$, $E^{i+1} = Simpl(E^i)$ converges to the least fixpoint $inf_{\sqsubseteq}(\{T \in L | Simpl(T) \sqsubseteq T\}) = \mathscr{T} = Simpl(\mathscr{T})$. □

Corollary 1. *By combining Th. 6.2.1, Pr. 6.2.5 and Th. 6.2.2, we have that:*

$$AS(P \cup F) = AS(\bigsqcap_{T \in \mathbf{TE}} T) = AS(Simpl^{\infty}(Inst^{\infty}(P,F) \cup F)).$$

Example 6.2.4. For program P_0 and facts F_1 of Section 6.1, the least tailored embedding of $P_0 \cup F_1$ under simplified set inclusion is the set $\{r_1', r_2'\} \cup F_1$.

6.3 Overgrounding with Tailoring

We illustrate in this section our technique for maintaining appropriate series of tailored embeddings which we call *overgrounded programs with tailoring* (OPTs).

In the following, the logic program P will be coupled with a sequence of sets of input facts $F_1, \ldots, F_n$. We aim to incrementally compute the sets $AS(P \cup F_1), \ldots, AS(P \cup F_n)$ by reducing the burden of the grounding step at the bare minimum, especially in later iterations. We update and maintain one element of the series of OPTs $G_1, \ldots, G_n$ via the repeated execution of an incremental instantiation function called IncrInst, and taking as arguments the program P, a ground program G and a set of input facts F. At iteration 1, we initialize the global sets of ground atoms $D = AF = PF = \emptyset$, and we let $G_1 = \text{IncrInst}(P, \emptyset, F_1)$. For an iteration $i > 1$, we will set $G_i = \text{IncrInst}(P, G_{i-1}, F_i)$.

The series $G_1, \ldots, G_n$ has three useful properties: *(i)* for each i, $G_i \cup F_i$ is a tailored embedding for $P \cup F_i$ and thus $AS(P \cup F_i) = AS(G_i \cup F_i)$; *(ii)* for the shot $i+1$, the IncrInst function obtains G_{i+1} from G_i by means of an iterative process, which repeatedly undoes now invalid simplifications in G_i (the *desimplification* step) and then computes additional new rules ΔG_{i+1} (the *incremental grounding* step); *(iii)* G_{i+1} extends G_i, as all the rules of G_i appear in G_{i+1} possibly in their desimplified version, i.e., $G_i \sqsubseteq G_{i+1}$. The global set D collects the rules that were deleted at some iteration and could be restored later on, whereas AF and PF keep record of so called *accumulated facts* and *persistently true facts*, respectively. After computing G_i for a shot i, we will have that $AF = \bigcup_{1 \leq k \leq i} F_k$ and $PF = \bigcap_{1 \leq k \leq i} F_k$. Intuitively, $Heads(G_i)$ will represent *possibly true atoms* built by applying the *Inst* operator starting from

AF as initial set of possible atoms. An atom outside $Heads(G_i) \cup AF$ is assumed to be *certainly false* and might trigger simplifications of type 2. Similarly, $Facts(G_i)$ will be taken as the set of *certainly true* atoms which allow to apply simplifications of types 1 and 3.

OUTLINE OF THE INCRINST FUNCTION. An abstract version of the INCRINST function is given in the next page. Let us assume to be at iteration $i+1$ for $i > 1$. The INCRINST function is composed of a DESIMPL step and a ΔINST step. On the one hand, in the DESIMPL step the rules in G_i are possibly *desimplified* whereas previously deleted rules are possibly restored. On the other hand, undeleted rules and new facts $NF = F_{i+1} \setminus AF$ can trigger the generation of new rules, which are incrementally processed in the ΔINST step. These new rules are simplified and added to G_{i+1}.

The set NR keeps track of rules restored from D and of new rules added in the ΔINST step. Atoms in $Heads(NR) \cup NF$ can invalidate simplifications of type 2 as they represent no longer certainly false atoms. The set OF is instead used to keep track of atoms that are no longer assumed to be certainly true at the current shot; the atoms in OF can invalidate former simplifications of types 1 and 3. The iterative process internal to INCRINST continues until no new rules are added and no new derived facts need to be retracted, i.e., when both NR and OF do not change anymore.

DESIMPLIFICATION STEP. The DESIMPL step makes an update the current ground program G (for the sake of readability, modifications on G are done on a copy DG initially set to G) in which simplifications of types 1 through type 3 are undone. Note that the desimplification might trigger new additions to OF and NR, which in turn can cascade new desimplifications and/or new incremental additions. We purposely allow redundant desimplifications: an atom $f \in Facts(G)$ might be added to OF as soon as a rule r where f is the only atom in $H(r)$ (i.e., $H(r) = \{f\}$) is desimplified (line 22). However, although there can be some other rule r_f in G such that $H(r_f) = \{f\}$ and $B(r_f) = \emptyset$, the restore operation on r does not affect the correctness of DG.

INCREMENTAL GROUNDING STEP. In this step we instantiate and simplify each rule $r \in P$ that can be constructed using the new ground atoms available in $Heads(NR) \cup NF$ up to a fixpoint. The *getInstances* function processes a non-ground rule r, the ground program DG and the set $NR \cup NF$. All possible new matches for the input rule r are differentially obtained and simplified. The *getInstances* function can be implemented by carefully adapting semi-naive evaluation techniques. This can avoid the generation of duplicated rules, thus saving computation time and memory consumption.

SIMPLIFICATIONS. Our algorithm applies simplifications over new rules NR only and in two separate moments: *(i)* as soon as a new rule is generated (line 34) and *(ii)* at the end of the main cycle (line 40). In the latter case, we apply all simplification types. In the former case, the $Simpl[1,3]$ operator is meant to apply only simplifications

```
Input: Non-ground program P, ground program G, input facts F
Output: A desimplified and enlarged ground program
Updates: the set of deleted rules D, collection of sets AF and PF
 1: function INCRINST(P,G,F)
 2:     DG = G,                                       /* D = Deleted Rules */
 3:     NR = ∅,               /* NR = New Rules and rules restored from D */
 4:     NF = F \ AF,                                     /* NF = New Facts */
 5:     OF = PF \ F,                      /* OF = Old Facts (no longer true) */
 6:     AF = AF ∪ F,                              /* AF = Accumulated Facts */
 7:     PF = PF ∩ F,                          /* PF = Persistently True Facts */
 8:     while NR or NF or OF have new additions do
 9:         // DESIMPL step
10:         for all r ∈ D do                       /* undo simpl. types 1 and 2 */
11:             L1 = {not a ∈ B⁻(r) s.t. a ∈ OF}
12:             L2 = {a ∈ B⁺(r) s.t. a ∈ Heads(NR) ∪ NF}
13:             if L1 ∪ L2 ≠ ∅ then
14:                 D = D \ {r}
15:                 NR = NR ∪ {r}
16:             end if
17:         end for
18:         for all r ∈ DG do
19:             L3 = {a ∈ B⁺(r) s.t. a ∈ OF}
20:             for all l ∈ L3 do                          /* undo simpl. type 3 */
21:                 if B(r) = ∅ ∧ ‖H(r)‖ = 1 then
22:                     OF = OF ∪ H(r)
23:                 end if
24:                 B(r) = B(r) ∪ {l}
25:             end for
26:         end for
27:         // ΔINST step
28:         do
29:             for all r ∈ P do
30:                 for all g ∈ getInstances(r,DG,NR ∪ NF) do
31:                     if Simpl[1,3]({g},NR ∪ F) = ∅ then          /* g is deleted */
32:                         D = D ∪ {g}
33:                     else
34:                         NR = NR ∪ Simpl[1,3]({g},NR ∪ F)
35:                     end if
36:                 end for
37:             end for
38:         while there are additions to NR
39:     end while
40:     S = Simpl^∞(NR,DG ∪ NR ∪ F), D = D ∪ hom(NR) \ hom(S)
41: return DG ∪ S
42: end function
```

of types 1 and 3. These two simplification types can be applied earlier and can prevent the generation of rules that will be nonetheless deleted later. We observe that we simplify only newly added rules appearing in *NR*, but with respect to the

current value of F. This will make G_{i+1} not "compatible" with inputs F_k, $1 \leq k \leq i$. Nevertheless, if some F_k appears again as input in a later iteration, the correctness of G_{i+1} can be achieved with a further desimplification step. It is worth noting that a more conservative strategy could consider only simplifications depending on PF.

Example 6.3.1. Let us recall again the example given in Section 6.1 and consider program P_0, the intermediate program $TG_1 = \{r'_1, r'_2\}$ and the set of input facts $F_1 = \{e(c,a),\ e(a,b),\ ab(c)\}$. $TG_1 \cup F_1$ is a tailored embedding for $P_0 \cup F_1$. Assume also that at this stage $D = \{r_3\}$. Given a new set of input facts $F_2 = \{e(c,a),\ e(a,d)\}$, we have that $\text{IncrInst}(P, TG_1, F_2)$ works as follows: NF is initially set to $\{e(a,d)\}$, $OF = \{e(a,b),\ ab(c)\}$, and DG is initially set to TG_1. The DESIMPL step will produce the updated set $DG = \{r_1, r'_2, r_3\}$ by modifying r'_1 in r_1, while the rule r_3 is undeleted and moved from D to NR. The ΔINST step generates the new rules r'_5 and r'_4 and adds them to NR. r'_5 is a simplified version of r_5 constructed using the new atom $e(a,d)$, while r'_4 is a reduced version of r_4 built using the new atom $r(a,d)$. No further desimplifications and changes to DG, NR and OF happen in the next DESIMPL and ΔINST steps nor in the final simplification. The set $\{r_1, r'_2, r_3, r'_4, r'_5\}$ is eventually returned.

Example 6.3.2. Suppose to enrich P'_0 with the following rule:

$$ab(X) \text{ :- } e(X,X).$$

The resulting program P_0 is as follows

$r(X,Y) \text{ :- } e(X,Y),\ not\ ab(X).$ $\quad$ $r(X,Z) \mid s(X,Z) \text{ :- } e(X,Y),\ r(Y,Z).$

$ab(X) \text{ :- } e(X,X).$

Let us consider the following set of input facts $F'_1 = \{e(c,a), e(a,b), e(c,c).\}$. Our algorithm will produce in output a simplified program TG'_1 as follows:

$r_1 : r(c,a) \mid s(c,a) \text{ :- }$ ~~$e(c,a)$~~$, r(c,a).$ $\quad$ $r_2 : r(c,c) \mid s(c,c) \text{ :- }$ ~~$e(c,a)$~~$, r(c,c).$

$r_3 : r(a,b) \mid s(a,b) \text{ :- }$ ~~$e(a,b)$~~$, r(a,b).$ $\quad$ $r_4 : r(c,a) \mid s(c,a) \text{ :- }$ ~~$e(c,c)$~~$, r(c,a).$

$r_5 : r(c,c) \mid s(c,c) \text{ :- }$ ~~$e(c,c)$~~$, r(c,c).$ $\quad$ $r_6 : ab(c) \text{ :- }$ ~~$e(c,c)$~~.

$r_7 : r(a,b) \text{ :- } e(a,b),\ not\ ab(a).$ $\quad$ $r_8 :$ ~~$r(c,c) \text{ :- } e(c,c),\ not\ ab(c).$~~

$r_9 :$ ~~$r(c,a) \text{ :- } e(c,a),\ not\ ab(c).$~~

Thus, $TG'_1 = \{r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7\}$ where r'_x is a simplified version of the corresponding rule r_x, whereas the set of deleted rules D is composed by rules r_8 and r_9. Given a new set of input facts $F'_2 = \{e(c,a), e(a,d)\}$ our grounding strategy will work as follows: NF is initially set to $\{e(a,d)\}$, $OF = \{e(a,b), e(c,c)\}$ and DG is initially set to TG'_1. The DESIMPL step will restore rules r_8 and r_9 from the set of deleted rules D adding them in NR and will apply desimplifications of type 3 on rules r'_3, r'_4, r'_5, r'_6 restoring them to their original form; furthermore, the ΔINST step will instantiate some new rules (r_{10}, r_{11} and r_{12}) which will be added to the NR.

$r_1 : r(c,a) \mid s(c,a) \text{ :- } \cancel{e(c,a)}, r(c,a).$

$r_2 : r(c,c) \mid s(c,c) \text{ :- } \cancel{e(c,a)}, r(c,c).$

$r_3 : r(a,b) \mid s(a,b) \text{ :- } e(a,b), r(a,b).$

$r_4 : r(c,a) \mid s(c,a) \text{ :- } e(c,c), r(c,a).$

$r_5 : r(c,c) \mid s(c,c) \text{ :- } e(c,c), r(c,c).$

$r_6 : ab(c) \text{ :- } e(c,c).$

$r_7 : r(a,b) \text{ :- } e(a,b),\ not\ ab(a).$

$r_8 : r(c,c) \text{ :- } e(c,c),\ not\ ab(c).$

$r_9 : r(c,a) \text{ :- } e(c,a),\ not\ ab(c).$

$r_{10} : \mathbf{r(a,d)} \text{ :- } \cancel{\mathbf{e(a,d)}},\ \mathbf{not\ ab(a)}.$

$r_{11} : \mathbf{r(a,d)} \mid \mathbf{s(a,d)} \text{ :- } \cancel{\mathbf{e(a,d)}}, \mathbf{r(a,d)}.$

$r_{12} : \mathbf{r(a,d)} \mid \mathbf{s(a,d)} \text{ :- } \mathbf{e(a,b)}, \mathbf{r(a,d)}.$

Since no further desimplifications and changes to DG, NR and OF happen in the next DESIMPL and ΔINST steps nor in the final simplification, the set $\{r'_1,\ r'_2,\ r_3,\ r_4,\ r_5,\ r_6,\ r_7,\ r_8,\ r_9,\ r'_{10},\ r'_{11},\ r_{12}\}$ is returned.

Proposition 6.3.1. *Let P be a non-ground program. Let C_P be the trivial component including all the predicate names in P. Then $G = \text{INCRINST}(P,\emptyset,F_1)$ coincides with Π as modified by the invocation of the instantiation procedure* Instantiate$(P \cup F_1, C_P, \Pi)$ *described in [51].*

Theorem 6.3.1. *Let $G_1 = \text{INCRINST}(P,\emptyset,F_1)$. For each i s.t. $1 < i \leq n$, let $G_i = \text{INCRINST}(P,G_{i-1},F_i)$. Then for each i s.t. $1 \leq i \leq n$, $AS(G_i \cup F_i) = AS(P \cup F_i)$.*

Proof. The proof goes along the lines of showing how to enlarge, under simplified set inclusion, a tailored embedding $G_{i-1} \cup F_{i-1}$ for $P \cup F_{i-1}$ to a tailored embedding $G_i \cup F_i$ for $P \cup F_i$. It is given by induction on the shot indices. Let $AS_i = AS(P \cup F_i)$. In the base case $(i = 1)$, $AS(G_1 \cup F_1) = AS_1$ since the DESIMPL step has no effect and the first iteration of the ΔINST step coincides with the typical grounding procedure of [51] (Proposition 6.3.1). In the inductive case ($i > 1$), we assume that $G_i \cup F_i$ is a tailored embedding for $P \cup F_i$, and we show that $G_{i+1} \cup F_{i+1}$ is a tailored embedding for $P \cup F_{i+1}$. Let $G_{i+1} = \text{INCRINST}(P, G_i, F_{i+1}, i)$. At the final iteration of the INCRINST algorithm, we have that $G_{i+1} = DG \cup Simpl^{\infty}(NR, DG \cup NR \cup F_{i+1})$, where DG is a desimplified version of G_i and NR is an additional set of rules both obtained by repeated application of DESIMPL and INCRINST steps.

Observe that $DG \cup F_i$ is such that $G_i \cup F_i \sqsubseteq DG_i \cup F_i$ and is a tailored embedding for $P \cup F_i$; then, let $AG_{i+1} = Inst^{\infty}(P, DG \cup F_{i+1})$. $DG \cup AG_{i+1} \cup F_{i+1}$ is a tailored embedding for $P \cup F_{i+1}$; it then follows that $DG \cup Simpl^{\infty}(AG_{i+1}) \cup F_{i+1}$ is a tailored embedding for $P \cup F_{i+1}$. Let $CG_{i+1} = \{s \in AG_{i+1} \mid \nexists r \in DG \text{ s.t. } hom(r) = hom(s)\}$. $DG_i \cup CG_{i+1} \cup F_{i+1}$ is a tailored embedding for $P \cup F_{i+1}$. Then we show that $CG_{i+1} \sqsubseteq NR$. It follows that $DG \cup NR \cup F_{i+1} = G_{i+1} \cup F_{i+1}$ is a tailored embedding for $P \cup F_{i+1}$, and that thus $DG \cup Simpl^{\infty}(NR, DG \cup NR \cup F_{i+1})$ is a tailored embedding for $P \cup F_{i+1}$, which concludes the proof. $\square$

6.4 Tailored Embeddings and Related Work

One of the differentiating ideas of our approach is that OPTs can be "patched" and adapted to new inputs only by adding new information. In other words, OPTs grow monotonically, although an effort has been done to slow down their growth. A second characteristic of our approach is that we do not include modelling directives for controlling incrementality and multi-shot programs. This choice comes with many advantages (like easier usage and modelling) at the price of the loss of control. In the above respects, our proposal has connections with several lines of research.

It is worth mentioning the early and recent work surveying and proposing incremental update of pool of views [95]. The aforesaid approaches focus on query answering over stratified Datalog programs and aim to just materialize query answers; our focus is on a generalized setting, where disjunction and unstratified negation are allowed and propositional logic programs are materialized and maintained, for the purpose of computing answer sets. In contrast with typical delete/rederive techniques, which require an additional effort to avoid overdeletions and ensure correctness, we purposely allow to perform more desimplifications than necessary. Also, the absence of rederivation activities allows us to keep incremental grounding times low.

In the *clingo* approach [63] the notion of incrementality is conceived in a different way: problems are modelled by thinking in terms of "layers" of modules, for which users specify the grounding and solving sequence thereof. In the context of overgrounded programs, modelling can be focused on a single declarative program. The incremental evaluation over the sequence of input facts is implicit and does not require the user's attention. Moreover, the *clingo* system is able to incrementally generate/update an answer set shot by shot, but the truth value of atoms associated to a given layer is decided at each shot and cannot be changed in subsequent iterations, whereas the overgrounding works on a fixed logic program P which can potentially generate completely different answer sets when moving from one shot to the next one.

Full incremental reasoning, the widest general setting in which a logic program is subject to arbitrary changes and one aims to implicitly maintain answer sets, is to date an almost unexplored topic. The Ticker system [16] can be seen as a significant effort in this direction, as it implements the LARS stream reasoning formal framework by using back-end incremental truth maintenance techniques under ASP semantics, but is limited to a language fragment with no recursion. We believe a tighter level of integration between grounders and solvers is necessary in order to achieve full incremental reasoning. Among tightly integrated approaches, it is worth mentioning lazy grounding [39, 76, 18]. Note that overgrounding is essentially orthogonal to lazy grounding techniques, since these latter essentially aim at blending grounding tasks within the solving step for reducing memory consumption; rather, our focus is on making grounding times negligible on repeated evaluations by explicitly allowing the usage of more memory, while still keeping the two evaluation steps separated.

Finally, it is worth highlighting that overgrounded programs with tailoring can be seen as an application of relativized hyperequivalent logic programs [130]. A member G of a sequence of OPTs is a logic program which is equivalent to P relative to (part of) a finite set of inputs $F_1, \ldots, F_n$. Investigation on the hyperequivalence properties of OPTs, possibly under semantics other than the answer set one, deserves further research.

	i^{2}dlv-noisd/ i^{2}dlv-isd	**iClingo/oClingo**	**Clingo5**	**Lars + Ticker**	**Alpha**
Syntax	Plain ASP	ASP + control directives	ASP + control directives + external scripts	Plain ASP + Window operators	Plain ASP
Modelling paradigm	Fixed ASP program P, changing input facts	Based on identifying and combining cumulative, volatile and base submodules	Customizable	Fixed Lars program, changing input stream	Single shot
Control of evaluation sequence	In charge of system designer	In charge of knowledge designer	In charge of knowledge designer	In charge of system designer	In charge of system designer

Table 6.1: Comparison between systems

A comparison of the main features of the systems discussed above can be found in Table 6.1; all systems make use the ASP semantic, however, some of them extend the language using special directives or new operators. Also the modelling paradigm can be different, and whether the burden to control the shot sequence is left on knowledge designers or it is builtin in the system. Knowledge designers, when using *iClingo/Clingo* systems, have to identify which part of the program need to be marked as base, cumulative and volatile; *Clingo5* users can customize their program using external directives; when using *Ticker*, the designer has to use a fixed LARS program with a changing input stream (similar to the $\mathscr{I}^2$-*DLV* system); finally, *Alpha* is a solver based on lazy grounding, working in the traditional “single shot” setting.

7 INCREMENTAL GROUNDING: DATA STRUCTURES AND ALGORITHMS

In this chapter we provide a more detailed description of the strategy we used in our incremental system, in order to make it available to those who want to integrate it within their grounder.

An existing, canonical grounder can be extended with incremental techniques by introducing some specific changes. In particular, these are: (i) disabling some of the known techniques currently available for simplifying ground logic programs; (ii) adding a desimplification method in order to make the program compatible with a series of subsequent different input facts; (iii) modifying the structure of the Semi-Naïve algorithm, which is at the core of a typical grounder, in order to eliminate the generation of duplicates during the incremental instantiation.

In this chapter, in Section 7.1, we will illustrate a way of adapting data structures in order to make them compatible with our strategy; then, an outline of our modified incremental Semi-Naïve algorithm will be shown in Section 7.2.2.

7.1 Data Structures

In the data structures shown in this section, for each predicate p, its current extension will be categorized in the three sets p_{OLD}, p_{NEW} and p_{ALL}. Namely, let us assume to be at iteration i for $i > 1$. p_{OLD} represents all the input and derived atoms for p from iteration 1 to iteration $i-1$, i.e., all the atoms that are certainly or possibly true from the first to the previous iteration; p_{NEW} contains all the input and derived atoms for p in the current iteration. The union between the set of atoms p_{OLD} and p_{NEW} will be called p_{ALL}. Predicates are stored in a table called *Predicate Extension Table* (PET), where each predicate points to a separate table storing its respective extension.

Example 7.1.1. In the following we show how the PET is initialized and then updated after each iteration with a simple example. Let us consider the program P_0 consisting of the following rule r_1:

$$a(X,Y,Z) \text{ :- } b(X),\ c(Y),\ d(Z).$$

Suppose to receive the set $F_1 = \{b(1..2),\ c(1..2),\ d(1..2)\}$ as input fact at the first iteration. Rule r_1 represents a cartesian product, consequently it will derive all the instances $a(1..2, 1..2, 1..2)$. The data structure appears as shown in Figure 7.1.

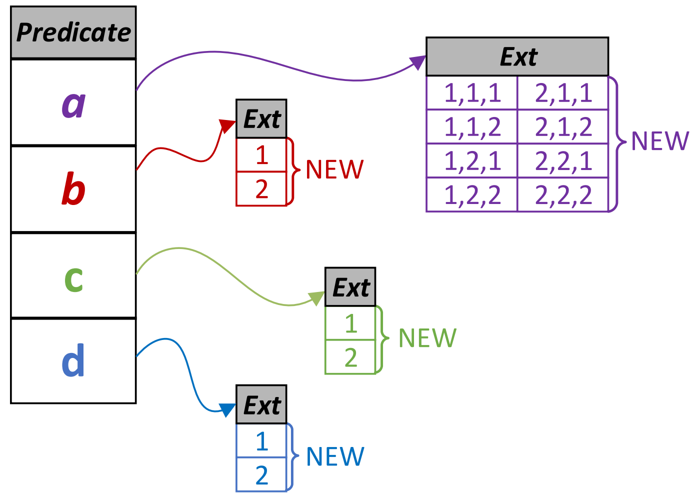


Figure 7.1: Predicate extension map overview - First iteration.

Let us assume that a subsequent run requires P_0 to be evaluated over a different set of input facts $F_2 = F_1 \cup \{b(3),\ c(3)\}$. At this point, the PET needs to be updated. All the atoms that will be added as input or derived in the current iteration will be marked as *NEW*, whereas all the others will be tagged as *OLD* as shown in Figure 7.2.

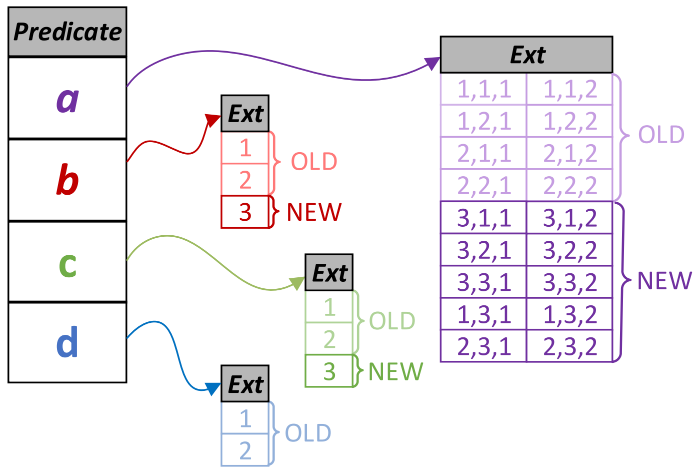


Figure 7.2: Predicate extension map - Second iteration.

7.2 Incremental Semi-Naïve Algorithm

In this section, after introducing some preliminaries about the foundation of the basic grounding methods, we will show our modified version of the Semi-Naïve algorithm, which is now able to perform an incremental evaluation without generating duplicate instances during the grounding step.

7.2.1 Preliminaries

The instantiation task could be computationally expensive and its efficiency is important for the performance of the entire system. Indeed, the grounding frequently forms a bottleneck and is crucial in real-world applications involving large input data. For such reasons, different strategies have been studied in order to reduce the time required during the grounding step thus reducing its computational effort.

A possible strategy, which has proved to be one of the most efficient, consists in the subdivision of the program into modules. Subsequently, these modules will be ordered to induce an admissible component sequence which will be followed during the instantiation phase.

These steps can be performed through the generation of *dependency* and *component graphs* which have been defined in Definition 2.1.19 and 2.1.20 of Section 2.1.3.

7.2.2 Algorithm

The presented algorithm optimizes and better details the operations performed from line 29 to 38 of our IncrInst algorithm. In particular, we optimized the selection order of rules to be grounded (line 29 of IncrInst) and the general management of data structures. More in detail:

1. in line 29 of our IncrInst algorithm we pick an arbitrary $r \in P$. Our refined algorithm follows a precise rule selection order strategy, which improves its efficiency (see line 3 of SemiNaiveInstantiation). This selection is implemented with the function OrderedNodes which takes into account component ordering and the difference between exit and recursive rules (see lines 5 and 12 of SemiNaiveInstantiation);

2. it is better detailed how a single non-ground rule r is incrementally grounded (see Figure 7.4).

In the following, we will firstly describe, more in detail, all the utility functions which will be used in the main algorithm to perform all its steps; then, a description of the main SemiNaiveInstantiation procedure will be reported.

ORDEREDNODES The ORDEREDNODES function takes as input the Component Graph G_P^c and returns in output an admissible component sequence $(C_1, \ldots, C_n)$.

EXITRULES AND RECURSIVERULES The EXITRULES procedure takes as input a component c and the program P to be instantiated. It outputs the set of all the *exit rules* of the program P for the current component c. Similarly, the RECURSIVERULES method outputs the set of all the *recursive rules* for c.

UPDATEDPREDICATES AND RECURSIVEPREDICATES The UPDATEDPREDICATES function takes as input a rule r and returns as output the set of updated predicates UP contained in $B(r)$. More in detail, the procedure iterates over the body of r and if a predicate is marked as updated, it will be added to the set UP. Similarly, the RECURSIVEPREDICATES method, returns the set of recursive predicate RP contained in $B(r)$.

INSTANTIATERULE The INSTANTIATERULE procedure is in charge of generating a set of ground instances for the input rule r on the basis of the set of atoms $match$. In our implementation this procedure is based on a *backjumping strategy* [51].

GETPREDICATESINBODY The GETPREDICATESINBODY function receives as input a rule r and returns a set of tuples ST. Each of these tuple contains a predicate (p) and its current update state ($isUpdated$). The predicate and its current update state are used to determine which extension of the predicate must be taken in consideration for the current instantiation.

GETPREDICATEEXTENSION The GETPREDICATEEXTENSION function takes as input a predicate p and the type of predicate extension required (OLD,NEW or ALL) and it returns as output a set S containing the requested part of the predicate extension of p.

SEMINAIVEINSTANTIATION AND NEXTINCREMENTALMATCH Our procedure shown in Figure 7.3 represents the core of the incremental instantiation. It takes as input both a program P to be instantiated and the Component Graph G_P^c, and outputs a set Π of ground rules containing only atoms which can possibly be derived from P. The input program P is divided into modules which are evaluated one at a time following an admissible component sequence as described in Subsection 7.2.1. This step is performed at line 3 of the SEMINAIVEINSTANTIATION procedure through a call to the function ORDEREDNODES. The algorithm starts the grounding process from the evaluation of the *exit rules* (line 5). For each of these rules, the algorithm finds all the predicates which result to be updated in the rule body, i.e., atoms which are given in input by the user or that have been derived at the current iteration. This step is performed by making use of the utility function UPDATEDPREDICATES (line 6). Then, the procedure iterates over all the updated predicates (*token*) and, at each iteration step, it firstly invokes the method NEXTINCREMENTALMATCH in order to

find a possible set of atoms which can be used to derive new instances (line 7) and, then, it execute the INSTANTIATERULE function to produces the ground instances for the rule r with the current set of predicate extensions (line 8).

The most important step of the algorithm is performed by the function showed in Figure 7.4, i.e., NEXTINCREMENTALMATCH. The procedure takes as input a rule r and the current updated predicate (*token*) and returns a set of predicate extensions (*OLD*, *NEW* or *ALL*) that can be used to instantiate the rule r avoiding the generation of duplicates during the incremental instantiation. If no match are available, it returns an empty set. This procedure makes use of a precedence operator ($\prec$) that compares the current *token* with all the predicates that have been updated in the rule body. Depending on the result of this comparison, the procedure determines which extension of the predicate must be taken in consideration during the grounding step, i.e., *OLD*, *NEW* or *ALL* making use of the function GETPREDICATEEXTENSION (lines 6, 8 and 10).

Similarly to the *exit rules*, the same steps apply for the *recursive rules* with the addition of some extra iterations until no further heads have been derived (from line 12 to line 30).

Input: Non-ground program P, component graph G_P^c
Output: Ground program Π

```
 1: function SEMINAIVEINSTANTIATION(P, G_P^c, F)
 2:     Π = ∅
 3:     for all c ∈ ORDEREDNODES(G_P^c) do          /* Admissible component sequence */
 4:                                        /* Incremental instantiation of exit rules */
 5:         for all r ∈ EXITRULES(c, P) do
 6:             for all token ∈ UPDATEDPREDICATES(r) do
 7:                 match = NEXTINCREMENTALMATCH(r, token)
 8:                 Π = Π ∪ INSTANTIATERULE(r, match)
 9:             end for
10:         end for
11:                                              /* First iteration for recursive rules */
12:         Π′ = ∅
13:         for all r ∈ RECURSIVERULES(c, P) do
14:             for all token ∈ UPDATEDPREDICATES(r) do
15:                 match = NEXTINCREMENTALMATCH(r, token)
16:                 Π′ = Π′ ∪ INSTANTIATERULE(r, match)
17:                 Π = Π ∪ Π′
18:             end for
19:         end for
20:                                           /* Further iterations for recursive rules */
21:         while Π′ ≠ ∅ do
22:             Π′ = ∅
23:             for all r ∈ RECURSIVERULES(c, P) do
24:                 for all token ∈ RECURSIVEPREDICATES(r) do
25:                     match = NEXTINCREMENTALMATCH(r, token)
26:                     Π′ = Π′ ∪ INSTANTIATERULE(r, match)
27:                     Π = Π ∪ Π′
28:                 end for
29:             end for
30:         end while
31:     end for
32: return Π
33: end function
```

Figure 7.3: Incremental Semi-Naïve Algorithm

Input: Non-ground rule r, a predicate $token$
Output: Set of atoms S
1: **function** NEXTINCREMENTALMATCH(r,$token$)
2: $S = \emptyset$
3: $(\langle p_1, isUpdated_1\rangle, ..., \langle p_n, isUpdated_n\rangle) = \text{GETPREDICATESINBODY}(r)$
4: **for all** $i \in 1,...,n$ **do**
5: **if** $isUpdated_i \wedge p_i \prec token$ **then**
6: $S = S \cup \text{GETPREDICATEEXTENSION}(p_i,\ OLD)$
7: **else if** $isUpdated_i \wedge p_i = token$ **then**
8: $S = S \cup \text{GETPREDICATEEXTENSION}(p_i,\ NEW)$
9: **else**
10: $S = S \cup \text{GETPREDICATEEXTENSION}(p_i,\ ALL)$
11: **end if**
12: **end for**
13: **return** S
14: **end function**

Figure 7.4

With the following simple example, we will show how the modified version of the Semi-Naïve algorithm works.

Example 7.2.1. Let us consider the program P_0 consisting of the following rule r_1:

$$a(X,Y,Z) \text{ :- } b(X),\ c(Y),\ d(Z).$$

When taking the set of facts $F_1 = \{b(1..2),\ c(1..2),\ d(1..2)\}$ into account, the algorithm, at the first iteration, will combine **all** the predicate with each other and the resulting (non simplified) ground program will be:

$$\begin{array}{ll} a(1,1,1) \text{ :- } b(1),\ c(1),\ d(1). & a(1,1,2) \text{ :- } b(1),\ c(1),\ d(2). \\ a(1,2,1) \text{ :- } b(1),\ c(2),\ d(1). & a(1,2,2) \text{ :- } b(1),\ c(2),\ d(2). \\ a(2,1,1) \text{ :- } b(2),\ c(1),\ d(1). & a(2,1,2) \text{ :- } b(2),\ c(1),\ d(2). \\ a(2,2,1) \text{ :- } b(2),\ c(2),\ d(1). & a(2,2,2) \text{ :- } b(2),\ c(2),\ d(2). \end{array}$$

Let us assume that a subsequent run requires P_0 to be evaluated over a different set of input facts $F_2 = F_1 \cup \{b(3),\ c(3)\}$. At this point, it is not possible to combine again all the instances of the predicates with each other, indeed, this process will cause some duplicates during the instantiation of the rule. In order to avoid this issue, our Semi-Naïve algorithm has been modified in order to automatically determine which predicate extensions need to be taken into account during the incremental instantiation step. More in detail, during the instantiation of the program P_0 over the set of facts F_2, the algorithm will select exactly which parts of predicate extensions must be combined with each other, i.e.,

	b	**c**	**d**
step_1	*b(3)*	*c(1..3)*	*d(1..2)*
step_2	*b(1..2)*	*c(3)*	*d(1..2)*

Thus, the instantiation step will generate the following new rules:

```
% Step 1
```

$a(3,1,1)$:- $b(3)$, $c(1)$, $d(1)$. $a(3,1,2)$:- $b(3)$, $c(1)$, $d(2)$.
$a(3,2,1)$:- $b(3)$, $c(2)$, $d(1)$. $a(3,2,2)$:- $b(3)$, $c(2)$, $d(2)$.
$a(3,3,1)$:- $b(3)$, $c(3)$, $d(1)$. $a(3,3,2)$:- $b(3)$, $c(3)$, $d(2)$.

```
% Step 2
```

$a(1,3,1)$:- $b(1)$, $c(3)$, $d(1)$. $a(1,3,2)$:- $b(1)$, $c(3)$, $d(2)$.
$a(2,3,1)$:- $b(2)$, $c(3)$, $d(1)$. $a(2,3,2)$:- $b(2)$, $c(3)$, $d(2)$.

Therefore, in this specific case, the algorithm will combine the predicates b, c and d as summarized in Table 7.1.

	b	**c**	**d**
step_1	*NEW*	*ALL*	*ALL*
step_2	*OLD*	*NEW*	*ALL*

Table 7.1: Instantiation step for a single rule with our Semi-Naïve algorithm

Part III

IMPLEMENTATION AND APPLICATIONS

8

$\mathscr{I}^2$-*DLV*: A DECLARATIVE OVERGROUNDING-BASED SYSTEM

In this chapter, we will discuss the system implementing our overgrounding maintenance algorithms as described in the previous chapters. Moreover, the results of the benchmark tests performed by using both these strategies will be reported and discussed.

8.1 System Architecture

The overgrounding strategies described in Chapter 5 and 6 have been implemented by extending the $\mathscr{I}$-*DLV* grounder [27, 35] to a version called $\mathscr{I}^2$-*DLV*. The resulting architecture of the system is depicted in Figure 8.1. The new system allows to preload a non-ground logic program P, to iteratively submit input facts F_i, and to obtain $AS(P \cup F_i)$. During the process, a simplified subset TG of $grnd(P)$ is kept in memory. Whenever new input F_{i+1} is submitted, TG is updated according to the tailored overgrounding strategy; a filtering stage then pipes relevant rules to the solver of choice. Homologous, simplified and deleted rules are kept track of by adding mark-up to a single copy of each rule.

More in detail, the system behaves as a process staying alive and providing a service-oriented behaviour, waiting for requests. An external application EA can open a working session and specify tasks to be carried out; working sessions are handled by a Session Manager component. After submitting a load request for a logic program P along with an initial set of facts F_1, EA can ask to compute the answer sets of P over F_1: the Grounder component is in charge of producing and storing the overgrounded program $Inst^\infty(P,F_1)$; an external Solver system is adopted to compute the answer sets of $Inst^\infty(P,F_1) \cup F_1$. This process can be repeated/iterated: EA can provide additional sets of facts F_k for $2 \leq k \leq n$ so that $Inst^\infty(P,AF_k)$ with $AF_k = \bigcup_{1 \leq i \leq k} F_i$ is incrementally produced and stored.

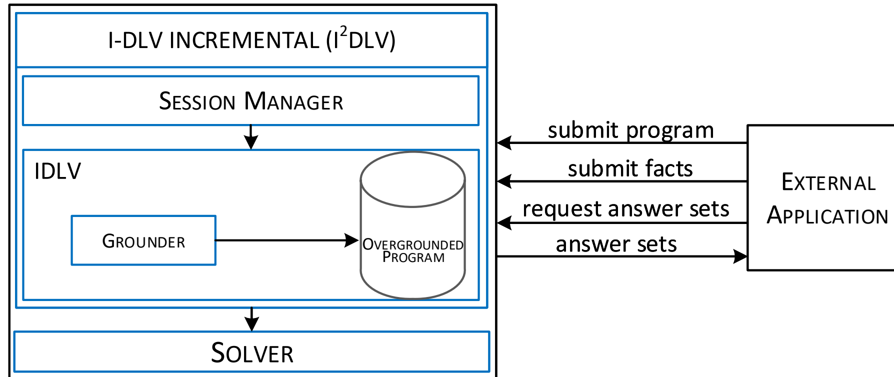


Figure 8.1: A general infrastructure of an ASP overgrounding-based reasoner relying on $\mathscr{I}^2$-*DLV*.

At each step k, the system is in charge of internally managing incremental grounding steps and automatically optimizing the computation by avoiding the re-instantiation of ground rules generated in a step $i < k$. Again, the solver can be invoked to compute the answer sets of $Inst^{\infty}(P, AF_k) \cup F_k$.

On the practical side, the tailored overgrounding approach has several advantages: the monotonic growth of the tailored overgrounded programs allows for easily implementing caching policies; if a grounding task must be interrupted, restarts on a new shot are almost straightforward to be implemented, since almost no rollback is required; the proposed framework is transparent to knowledge designers; highly general, non-optimized code can benefit from tailored overgrounding as there is no need to worry about which parts of logic programs might be more grounding-intensive. In the following, we show the results of an experimental evaluation on the system, which demonstrate how this approach allows to reduce both the time and the memory consumption. We expect this setting to be particularly favourable when non-ground input programs may contain grounding-intensive rules, as for instance, in the case of declaratively programmed robots or videogame agents.

8.2 Experimental Evaluation

In this section, we report about the experimental activities we conducted, aimed at assessing the practical impact of our approach; results show that it pays off in terms of performance, by reducing grounding times and keeping solving times within more than reasonable bounds. In particular, the evaluation was conducted in order to assess *(a)* the size of inputs fed to solvers and *(b)* the evolution of the performance of the combination of grounder and solver, given also the changing instantiation size. Since sources of choice points are left substantially unchanged by simplification

activities, we expected good improvements in performance due to faster solving times for deterministic parts of ground programs. We considered three benchmarks taken from three real world settings with different specific features: *Sudoku* game, Pac-Man [29] and Content Caching [13, 48]. The three benchmarks constitute good and generalizable real cases of incremental scenarios: the Sudoku domain allows to perform a stress-test of the approach on grounding-intensive tasks, the Pac-Man game allows to assess effectiveness of overgrounding for continuous reasoning in the context of videogames without the need for manual and involved customizations, whereas Content Caching is a typical example of decision making over fast-paced event streams. Experiments have been performed on a NUMA machine equipped with two 2.8GHz AMD Opteron 6320 CPUs, with 16 cores and 128GB of RAM. The measurements have been performed using *WASP* version 3.0.0, *clasp* integrated in *clingo* version 5.4.0 and Ticker version 1.0. We used two grounder versions: $\mathscr{I}^2$-*DLV*-isd stands for our new grounder featuring the new incremental simplification and desimplification techniques (*isd* in the following) as presented in Chapter 6, whereas $\mathscr{I}^2$-*DLV*-no-isd is a new improved implementation of plain overgrounded programs [30], in which isd techniques are disabled as reported in Chapter 5.

8.2.1 Multi-shot Sudoku

The classic Sudoku puzzle, or simply "Sudoku", consists of a tableau featuring 81 cells, or positions, arranged in a 9 by 9 grid. When solving a Sudoku, players typically adopt deterministic inference strategies allowing, possibly, to obtain a solution. Several deterministic strategies are known [31] and can be encoded in ASP; herein, we took into account two simple strategies, namely, "naked single" and "hidden single".

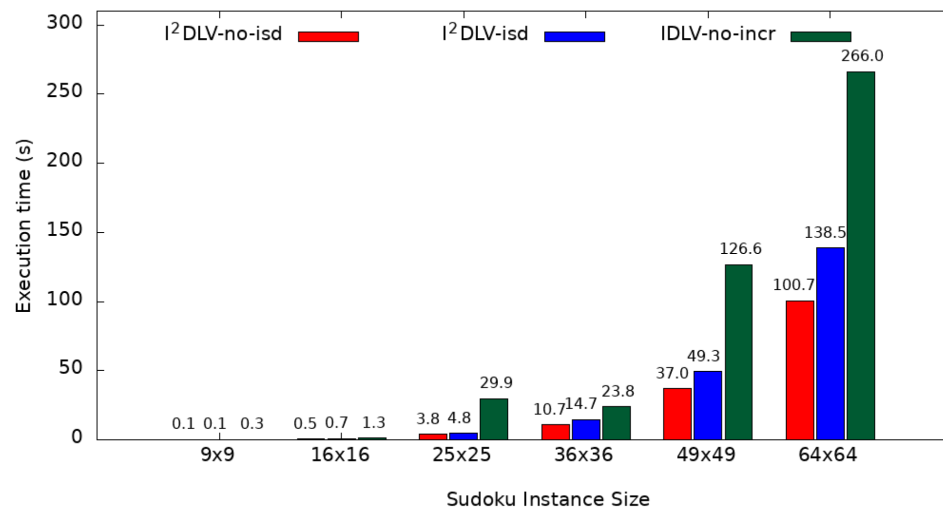


Figure 8.2: Experiments on Sudoku benchmarks

The encoding of Sudoku deterministic inference rules is generally grounding-intensive, like in the following code fragment, encoding the naked single strategy:

$$candidatesAreMoreThan2(X,Y) \text{ :- } candidate(X,Y,N), candidate(X,Y,N1), N \neq N1.$$
$$newValue(X,Y,N) \text{ :- } candidate(X,Y,N),\ not\ candidatesAreMoreThan2(X,Y), nogiven(X,Y).$$

where an atom *candidate(X,Y,N)* specifies that number N can be possibly assigned to the cell (X,Y), and *nogiven(X,Y)* tells that the value of the cell (X,Y) has not been inferred yet.

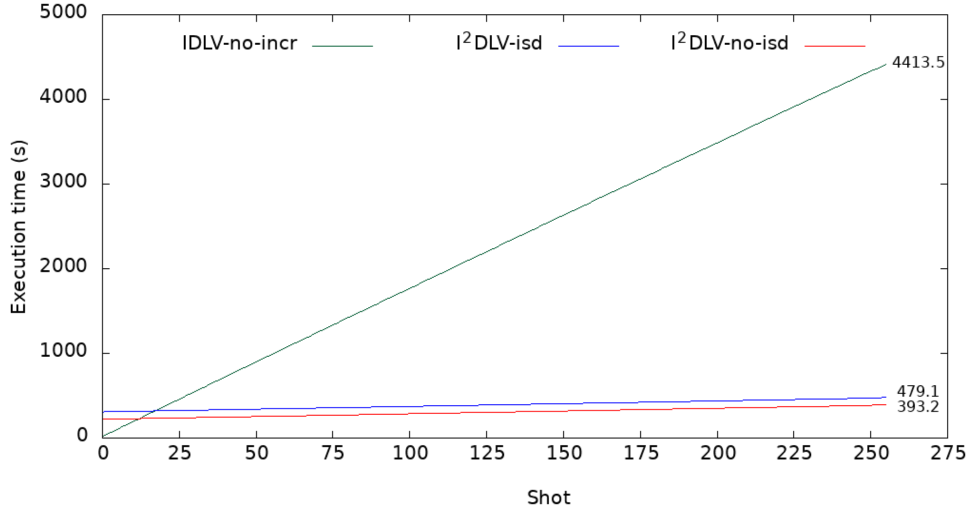


Figure 8.3: Grounding times for all iterations of a 81×81 Sudoku instance.

In the experiments we considered generalized Sudoku tables of size from 9×9 to 81×81, and tested logic programs under answer set semantics encoding deterministic inference rules. We compared three different evaluation strategies: (i) $\mathscr{I}^2$-DLV-isd implementing the incremental approach with simplification and desimplification steps enabled, (ii) $\mathscr{I}^2$-DLV-no-isd implementing the incremental approach without simplification and desimplification steps enabled, and (iii) $\mathscr{I}$-DLV-no-incr in which no incremental evaluation policy is applied. All the strategies have been executed in an online setting, in which consecutive series of input facts are submitted. For a given Sudoku table, the two inference rules above were modelled via ASP logic programs (see [31]). The resulting answer set encodes a new Sudoku grid, possibly deriving new numbers to be associated to initially empty cells, and reflecting the application of inference rules; the new partially completed grid is then given as input to the system and, again, by means of the same inferences, new cell values are possibly entailed. The process is iterated until no further association is found. In general, given a Sudoku, it cannot be assumed that the deterministic approach leads to a complete solution; however, for each considered Sudoku size, we selected only instances which are completely solvable with the two inference rules described above.

Grounding times of $\mathscr{I}^2$-*DLV*-isd, $\mathscr{I}^2$-*DLV*-no-isd and $\mathscr{I}$-*DLV*-no-incr are plotted in Figure 8.2: for each instance, the total grounding time (in seconds) computed over all iterations (or shots) is reported. $\mathscr{I}^2$-*DLV*-isd and $\mathscr{I}^2$-*DLV*-no-isd required similar times to iteratively solve each instance, and performed clearly better than $\mathscr{I}$-*DLV*-no-incr. Figure 8.3 shows a closer look on the performance obtained on size 81×81 which is the one requiring the highest amount of time to be solved and the highest number of iterations: for each shot, the grounding time (in seconds) is reported. At the first shot, both $\mathscr{I}^2$-*DLV*-isd and $\mathscr{I}^2$-*DLV*-no-isd required a huge amount of time if compared with $\mathscr{I}$-*DLV*-no-incr. This is due to the higher number of rules which will be generated, using the deterministic ASP encoding; for each further iteration, however, $\mathscr{I}^2$-*DLV*-isd required an average time of 0.67 seconds: a time reduction of 96% w.r.t. $\mathscr{I}$-*DLV*-no-incr, that showed an average time of about 17.23 seconds. On the overall, this confirms the potential of our incremental grounding approach in scenarios involving updates in the underlying input facts.

	Deterministic strategy			***Non-deterministic strategy***	
	i^{2}dlv-isd	**wasp**	**i^{2}dlv-isd+wasp**	**clingo5**	**dlv2**
Size 9x9	0.10	0.89	0.99	0.03	0.03
Size 16x16	0.71	8.15	8.87	0.27	0.27
Size 25x25	4.75	227.62	232.40	1.72	1.88
Size 36x36	14.70	249.093	263.79	7.29	6.85
Size 49x49	49.31	1781.34	1831.65	27.50	23.08
Size 64x64	138.53	5387.53	5526.06	81.48	66.64
Size 81x81	479.14	112376.64	112855.78	killed (144h)	killed (144h)

Table 8.1: Sudoku times

Furthermore, we also run *clingo5* and *dlv2* over an ASP encoding implementing a non-deterministic strategy for solving the Sudoku Puzzle; then we compared the results. Table 8.1 shows the times obtained for each Sudoku instance. Both *clingo5* and *dlv2* performs much better than the incremental grounder when run on smaller sizes of the puzzle with a non-deterministic encoding, however, they do not scale well on larger sizes as, for example, on size 81×81. In this setting, both *clingo5* and *dlv2* ran more than 144 hours without producing any admissible solution.

8.2.2 Multi-shot Pac-Man

In our work [29] we show how to integrate an ASP-based reasoning module in the Unity game development framework, and showcase an artificial player for the classic real-time game Pac-Man. The Pac-Man moves in a board containing a number of pellets and four ghosts chasing him: the goal is to eat all pellets while avoiding the four ghosts. The Pac-Man must continuously decide the way to take, depending on "dynamic" (i.e., changing at each reasoning shot) facts representing the current status of the board (empty/pellet tiles, positions of the four ghosts). The new direction of the

Pac-Man depends on which logical assertion among *next(up)*, *next(down)*, *next(left)*, *next(right)* is in the "best" guessed answer set. The intelligence of the Pac-Man is encoded via a logic program P_{pac}, that must be repeatedly executed; this requires multiple grounding+solving jobs over slightly different and unforeseen inputs.

Next, we will particularly focus only on parts of P_{pac} requiring a significant effort at the grounding stage. Instances of the *tile* predicate encode the game board divided into tiles; the fact *pacman(x,y)* represents the current position of the Pac-Man, while *ghost(x,y,g)* represents the position of ghost g; atoms of the form *nextTile(X,Y)* encode the possible next positions of the Pac-Man. The strategy adopted by P_{pac} is quite simple: priority is to get away from ghosts. To this end, distances between the next position candidates and all ghosts are computed: the next position chosen is among the ones increasing the distance to the nearest ghost. This behaviour is achieved via an ASP code fragment similar to the following[1]:

$r_1:$ $nextTile(X,Y) \text{ :- } pacman(Px,Y),\ next(right),\ X = Px+1,$
$tile(X,Y).$

$r_2:$ $nextTile(X,Y) \text{ :- } pacman(Px,Y),\ next(left),$
$X = Px-1,\ tile(X,Y).$

$r_3:$ $nextTile(X,Y) \text{ :- } pacman(X,Py),\ next(up),\ Y = Py+1,\ tile(X,Y).$

$r_4:$ $nextTile(X,Y) \text{ :- } pacman(X,Py),\ next(down),\ Y = Py-1,$
$tile(X,Y).$

$r_5:$ $adjacent(X1,Y1,X2,Y2) \text{ :- } tile(X1,Y1),\ tile(X2,Y2),$
$step(DX,DY),\ X2 = X1+DX,\ Y2 = Y1+DY.$

$r_6:$ $adjacent(X1,Y1,X2,Y2) \text{ :- } tile(X1,Y1),\ tile(X2,Y2),$
$step(DX,DY),\ X2 = X1-DX,\ Y2 = Y1-DY.$

$r_7:$ $distance(X1,Y1,X2,Y2,1) \text{ :- } tile(X1,Y1),\ adjacent(X1,Y1,X2,Y2).$

$r_8:$ $distance(X1,Y1,X3,Y3,D) \text{ :- } number(D),$
$distance(X1,Y1,X2,Y2,D-1),\ adjacent(X2,Y2,X3,Y3).$

$r_9:$ $distPacmanGhost(D,G) \text{ :- } nextTile(Xp,Yp),$
$ghost(Xg,Yg,G),\ minDistance(Xp,Yp,Xg,Yg,D).$

$r_{10}:$ $noMinDistPacmanGhost(X) \text{ :- } distance(X),$
$distPacmanGhost(X,_),\ distPacmanGhost(Y,_),\ X > Y.$

$r_{11}:$ $minDistancePacmanNextGhost(MD) \text{ :- } distPacmanGhost(MD,_),$
$not\ noMinDistPacmanGhost(MD).$

1 The full ASP code used to encode the Pac-Man AI makes also use of weak constraints to optimize the choose of the next move. A full encoding can be found at the following link https://github.com/DeMaCS-UNICAL/ThinkEngine-Games-Pacman/tree/main/Pac-Man-Unity-EmbASP-master/encodings

The above code contains parts which will likely have the same instantiation regardless of input facts (e.g., the *adjacent* and *distance* predicates), and parts whose instantiation will slightly differ depending on input facts (i.e., *nextTile* and *distPacmanGhost*, which depend on current positions of Pac-Man and ghosts). Such two categories of code fragments would roughly correspond to parts respectively marked with the former `#base` and `#cumulative` directives of *iclingo* [63, 64], and later generalized with the `#program` keyword of *clingo*; however, it is not necessary to introduce specific grounding directives within ASP code while using $\mathscr{I}^2$-*DLV*-isd. In [29], in the absence of an overgrounding engine, we manually tabled the instantiation of the logic program above, we resorted to procedural code for many aspects of the reasoning process, and we limited the maximum visibility horizon of the Pac-Man to 10 tiles. By adopting the overgrounding approach, we were able to avoid manual optimizations and achieved a fully automatic incremental approach: the grounded program is internally stored right after the first computation, thus bypassing re-computations of heavy grounding tasks.

It is worth noting that the *adjacent* and *distance* predicates are defined in a general but inefficient way; this can likely be the case if such a predicate was taken from a predicate module library. Alternative definitions, improving grounding times, would be in principle possible: for instance, the distance between tiles is encoded with the predicate $distance(X_1,Y_1,X_2,Y_2,D)$, where D is computed for all couple of points $(X_1,Y_1) \times (X_2,Y_2)$; this is what one can expect from a modeller not knowing, and probably not wishing to know how to optimize this code. Pushing $nextTile$ atoms within rule bodies would allow to reduce the grounding size by limiting grounding only to actually necessary distance values. Nonetheless, besides making code less readable and less declarative, this modification would disrupt modularity, and it requires some expertise on operational details of grounders.

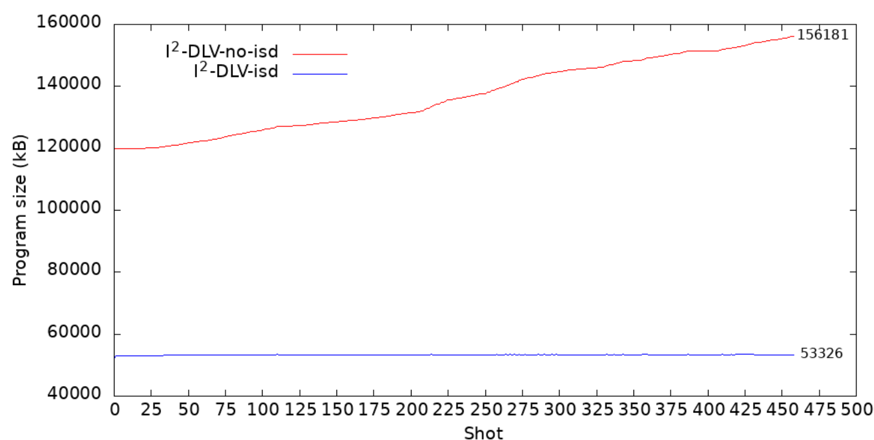


Figure 8.4: Results of Pac-Man benchmark. Comparison of instantiation sizes for both grounders.

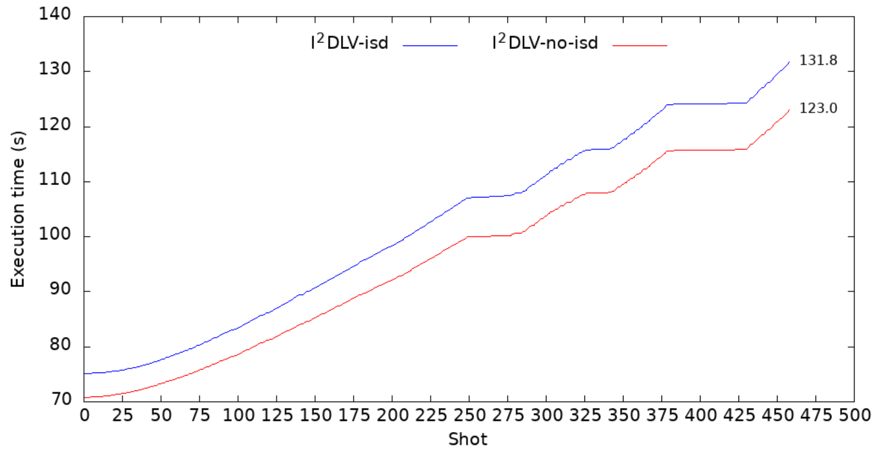


Figure 8.5: Results of Pac-Man benchmark. Comparison of grounding time for both grounders.

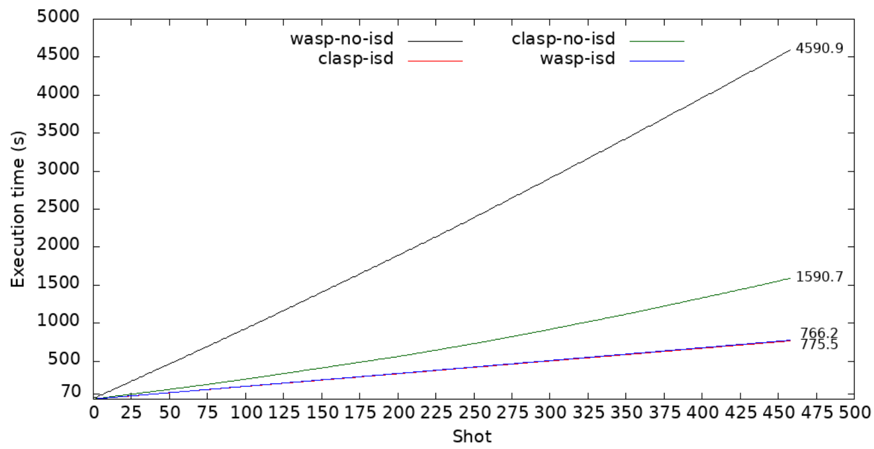


Figure 8.6: Results of Pac-Man benchmark. Comparison of solving time.

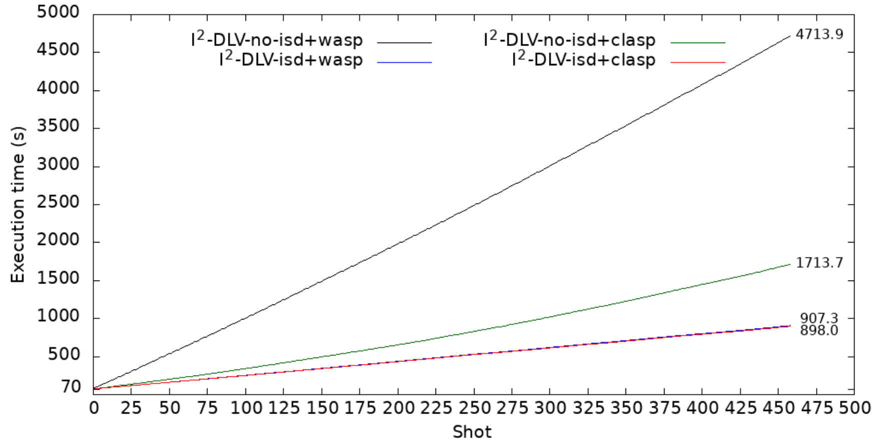


Figure 8.7: Results of Pac-Man benchmark. Reports about cumulative execution time for four possible combinations of grounders and solvers.

Experiments in the Pac-Man domain were conducted by logging a series of 459 consecutive steps taken during an actual game; each step encodes a different status of the game board in terms of logical assertions. Such inputs were, in turn, run along with a program P_{pac} in a controlled environment outside the game engine, averaging times over five separate runs. P_{pac} is repeatedly executed together with different inputs describing the current status of the game map, like e.g., the current position of enemy ghosts, the position of pellets, etc. Several parts of P_{pac} can be considered "grounding-intensive", like the ones describing the predicate $distance(X_1,Y_1,X_2,Y_2,D)$, where D is computed for all pairs of points $(X_1,Y_1) \times (X_2,Y_2)$, taking care of the shortest path between (X_1,Y_1) and (X_2,Y_2), given the shape of the labyrinth in the game map. Grounding was performed by our new engine called $\mathscr{I}^2$-*DLV*-isd and by an improved implementation of plain overgrounded programs called $\mathscr{I}^2$-*DLV*-no-isd. The latter does not feature simplification techniques.

The solving task was instead performed using WASP [7] and clasp [61]. In order to assess performance in the worst case scenario, we allowed the Pac-Man to have a visibility horizon of 30 tiles in each direction.

Figure 8.4 compares instantiation sizes for both grounders; Figure 8.5 reports about cumulative grounding time for both $\mathscr{I}^2$-*DLV*-isd and $\mathscr{I}^2$-*DLV*-no-isd. $\mathscr{I}^2$-*DLV*-isd require a little bit more time if compared to $\mathscr{I}^2$-*DLV*-no-isd due to the time spent during the simplification and desimplification steps. Figure 8.6 shows cumulative solving times for all the possible combination of solver and ground instances. Finally, Figure 8.7 reports about cumulative execution times for the four possible combinations of grounders and solvers. The X axis diagrams data in shot execution order. Results show that both solvers benefit of the smaller inputs produced by tailored overgroundings, with *wasp* showing a remarkable improvement. For all the four combinations, the slope of the cumulative time curve reflects an almost constant

execution time, with the exception of the first shot measuring around 70 seconds. In this shot, grounding times account for almost all the computation time. A slight progressive worsening in the execution time per shot can be seen especially for combinations involving the old grounder $\mathscr{I}^2$-*DLV*-no-isd. This is due to the larger program input fed to solvers. For both grounders, we noticed that instantiation times become immediately negligible in later iterations, with $\mathscr{I}^2$-*DLV*-isd being around 7% less performant than $\mathscr{I}^2$-*DLV*-no-isd because of simplification and desimplification activity.

8.2.3 Content Caching

In this benchmark, the caching policy of a given video content is controlled using a logic program P_{cc}. The caching policy of choice is encoded in the answer sets of $P_{cc} \cup E$ where E encodes a continuous stream of events describing the evolving popularity level of the content at hand. This application has been originally designed in the LARS framework using time window operators in order to quantify over past events [14]. We adapted the conversion method specified in the work presenting Ticker [16] to obtain P_{cc} as a plain logic program under answer set semantics, while events were converted to corresponding sets of input facts. These kinds of stream reasoning applications can be fairly challenging, depending on the pace of events and the size of the time window at hand. Our experiments were run in a worst-case scenario in which the caching policy could be decided based on events happening in the last 100 seconds, were the event pace was assumed to be 0.1 seconds. In this setting, a stream reasoning system must be able to deal with a total of 100×10 different timestamp symbols, and with proportionally large ground programs. Again, we compared the four combinations of grounders and solvers, and the *Ticker* system in its two implementations: the *Ticker* ad-hoc truth maintenance based version (ticker-incr), and the *clingo*-based one (*Ticker-asp*) as described in Subsection 3.5 of Chapter 3.

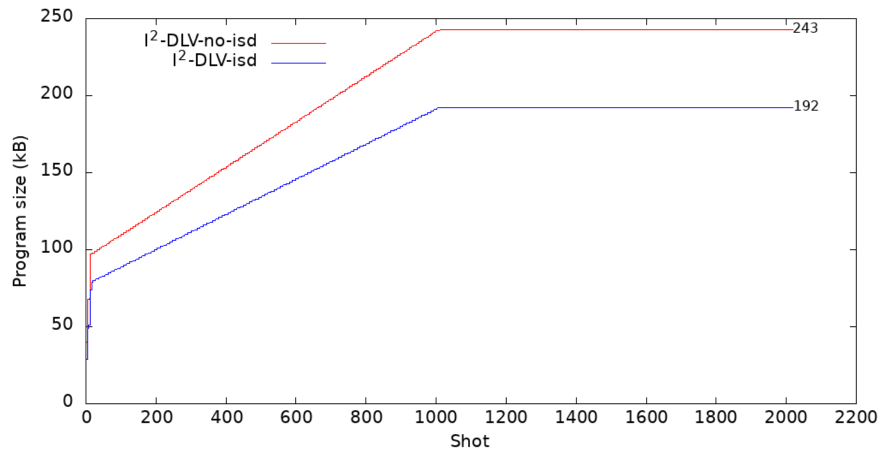


Figure 8.8: Results of Content Caching benchmark. Comparison of instantiation sizes for both grounders.

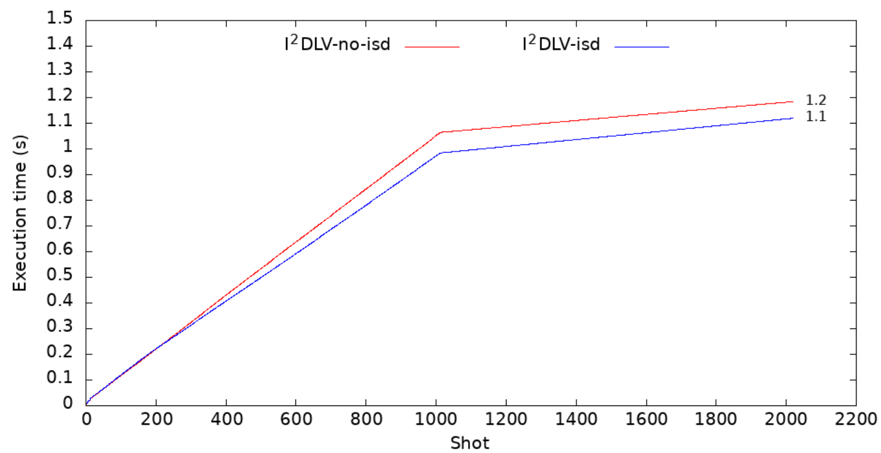


Figure 8.9: Results of Content Caching benchmark. Comparison of grounding time for both grounders.

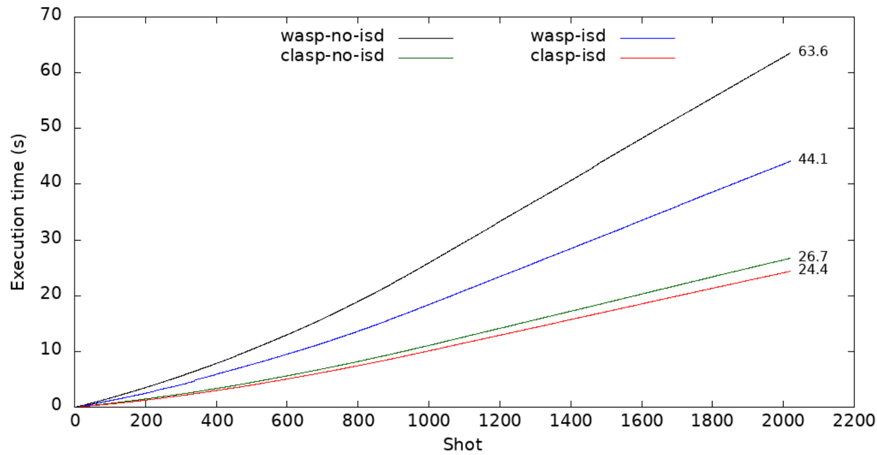


Figure 8.10: Results of Content Caching benchmark. Comparison of solving time.

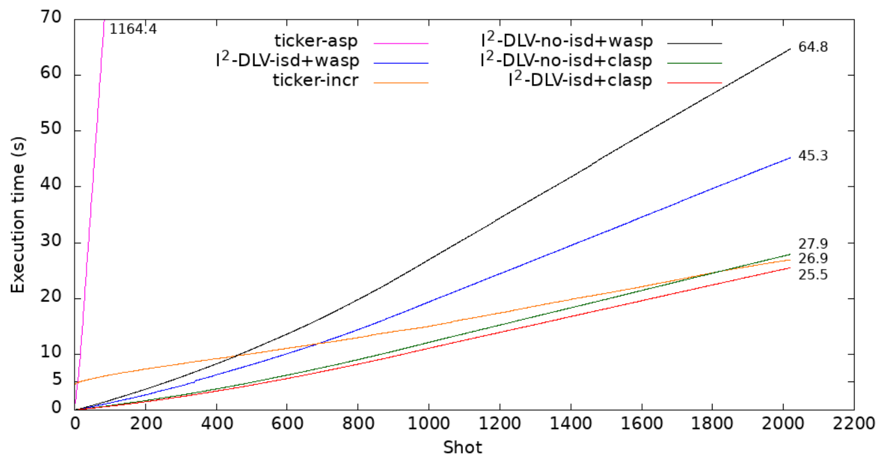


Figure 8.11: Results of Content Caching benchmark. Reports about cumulative execution time for four possible combinations of grounders and solvers and the *Ticker* system in its two implementation.

Figure 8.8 shows that both $\mathscr{I}^2$-*DLV*-isd and $\mathscr{I}^2$-*DLV*-no-isd grounders add new rules to their respective overgrounded program up to around shot 1000, which corresponds to the number of time points allowed in the chosen 100 seconds window. After this threshold, instantiated programs stay constant, with $\mathscr{I}^2$-*DLV*-isd generally producing a smaller input. Figures 8.9 and 8.10 show, the grounding time for both $\mathscr{I}^2$-*DLV*-isd and $\mathscr{I}^2$-*DLV*-no-isd, and the solving time of all the possible combination of solvers and ground instances, respectively. In Figure 8.11, the slope of cumulative times shows that *Ticker-incr* has some initial computational cost due to its pre-grounding phase, then performs better in terms of later per-shot times. The four combinations using our grounders have less initial computational cost, while their per-shot times increase slightly in later iterations, with $\mathscr{I}^2$-*DLV*-no-isd paired with *clasp* having the

best performance, which is quite close to *Ticker-incr*. *Ticker-asp* does not feature incremental optimization strategies, thus it is not comparable with other solutions.

9 A REAL APPLICATION SCENARIO: INTEGRATING DECLARATIVE TOOLS IN THE VIDEOGAME DEVELOPMENT LIFECYCLE

One might wonder whether declarative tools, possibly improved by means of incremental reasoning techniques, fit in a real applicative scenario with strict requirements on repeated evaluation of reasoning tasks. The answer to this question is not straightforward, as several obstacles arise, especially about how to integrate declarative modules in existing software applications. In this chapter we investigate on these obstacles, particularly focusing on the demanding domain of videogame development.

When comparing declarative, rule-based, formalisms with imperative languages, one can notice how it can be much easier to solve a problem using the first approach rather than the second one in a variety of settings. On the one hand, rule-based declarative formalisms can be of great help for the definition of AI. Indeed, the declarative nature of rule-based knowledge bases allow to focus on specifying desiderata, thus getting rid of the burden of defining how to meet them (declarative specifications do not need algorithm design/encoding). Furthermore, knowledge is typically explicitly represented. Specifications written using declarative rules are definitely easier to be modified than implicit imperative solutions and, so, easier to be maintained: one can start by quickly defining a first version and then iteratively rearrange it in increasingly sophisticated ways. On the other hand, when used in these contexts, declarative-based AI modules need to interact with other "non-declarative" components; i.e., one or more logic-based reasoning modules, handled by suitable declarative systems, must be integrated into larger, hybrid, architectures. Moreover, imperative languages enjoy a better efficiency, a much wider user base, easier interoperability and better handling of arbitrary data structures.

Combining these two radically different paradigms, so to achieve the benefits of both worlds, is therefore desirable and several ways for combining declarative paradigms, such as ASP, with traditional ones have been significantly studied in the recent years. However, until now this road has been unexplored in several relevant contexts like, for example, the videogame world, which is one of the applications based on imperative languages with strict real-time requirements.

The integration of declarative paradigms in the videogame development workflow is a challenging topic in the *Knowledge Representation* (KR) area. In general, AI techniques could play a role in numerous task applications in the game industry,

ranging from programming the behavior of non-player characters to automated game content generation. The particular context of real-time videogames is challenging for researchers since it means to work within a highly reactive environment: this environment usually produces a fast-paced stream of input events, thus requiring really fast responses from a reasoning system. For a reasoning module deployed within a videogame logic, it is, also, crucial to have the possibility of stopping and/or quickly restarting reasoning tasks, especially if the KR system at hand is not able to guarantee the required reactivity.

Besides the performance issues, a second technical obstacle concerns the practical integration of an external KR module in an object-oriented environment. The two paradigms offer a world representation on different abstraction levels: the level of abstraction is usually higher for the KR module and lower and more operational for the object oriented one. Indeed, reasoning in terms of a low-level abstraction is not feasible, and an adaptment layer from this to the high-level is needed. For instance, game maps typically require a discretization in grid cells, so to avoid to work at the pixel level; floating point-based physical simulations need simplification and abstraction; and so on.

Multiple strategies can be used to achieve such an integration [50], each one with its advantages and disadvantages depending on the context in which one has to work. Specifically, in a game development engine it is important to clearly distinguish the run-time context from the design-time context. Also it must be noted that, at run-time, a main module that we will call the *procedural side*, takes care of updating the game world: embedding one or more reasoning modules, which will belong to what we will call the *reasoning side*, requires the introduction of multiple concurrent execution flows, in order to prevent lagging in the screen update, and the resolution of a number of integration issues between the two sides.

General ways for easing the embedding of declarative paradigms into imperative, and in particular object-oriented, programming approaches, have been significantly explored in the recent years, especially for Answer Set Programming [21, 72, 28], for which a number of solutions have been proposed [111, 56, 62, 117, 126, 57].

Nonetheless, given the general impedance between the two different paradigm categories, the game development context still lack tools for the integration of rule-based formalisms.

In order to overcome such limits, we developed an integration of AI declarative modules within applications in the known Unity engine: the *ThinkEngine* module.

More in detail, the *ThinkEngine* features a tight data sharing model between the procedural side and the declarative side of a developed game, based on the notion of *sensors* and *actuators*; an appropriate asynchronous execution model and a suitable information passing strategy allows to handle time-consuming reasoning tasks with

no interferences with the game main thread. Reasoning tasks can be attached to trigger conditions or can be executed at scheduled times allowing a good degree of flexibility.

In Section 9.1 we will introduce the Unity Engine and the classical Unity videogame workflow. The, in Section 9.2 we will discuss how automated reasoning modules can be integrated in a real-time execution loop, such as the typical game execution loop used when running actual videogames. This integration is particularly strategic when developing NPCs players acting in the game world. In Section 9.3 we will provide a detailed description of our *ThinkEngine* framework; then in Section 9.4 and 9.5 we will provide some definitions about *Sensors, Actuators* and the *declarative semantic* used to implement the framework. In Section 9.6 we will describe the internal implementation and mappings done by the system. Finally, practical examples of how this approach can be applied to the videogame context will be given in Section 9.7. In particular, we will illustrate two simple showcases: the Frogger game in Subsection 9.7.1 and the Tetris puzzle in Subsection 9.7.2.

9.1 Unity Game Engine

Thanks to the advent of online distribution systems, as well as the mobile market for Android and iOS devices, the videogame industry has grown rapidly since the early 2000s. In fact, these systems are making it easier to publish indie games, i.e., videogames published without the funding of a publisher. Around this gaming market several platforms are springing up, known as *Game Engines*. These are object-oriented software development environments designed for people to build videogames [134]. They offer the possibility to easily integrate in a software different things, for example:

- physics,
- rendering, and
- AI.

There are more than 100 solutions on the market, making it more difficult to choose which one to use when developing a game. Among these solutions, one of the most used is the Unity game development engine.[1]

Unity is a cross-platform game engine first announced and released in 2005. Nowadays, the engine has been extended to support over 25 platforms (Windows, Android, iOS and so on). Unity offers facilities for developing both 2D and 3D videogames. One can use this framework to assemble assets (audio, special effects and so on) and art into environments and scenes, and concomitantly play, test and edit the game if necessary. Developers can use a primary scripting API in C# (for both the Unity editor in the form of plugins, and games themselves) but also visual editing facilities

1 https://unity3d.com/unity

like drag and drop, property editing etc.. Unity makes available an asset store[2] on which developers can put their solutions (editor plugins, models, SDKs, templates...) for selling (some assets are also proposed for free). Assets are divided in macro and micro categories offering a rich collection of solutions for different purposes such as, for instance, characters, textures and AI. Within this engine, *Game Objects* are the fundamental data structures that represent characters, props and scenery. Game objects do not accomplish much in themselves but they act as containers for *Components*, which implement the real functionalities. The *Game world*, or *environment*, is the set of all the objects used in a game project. At run-time stage, the game world is highly dynamically changing, requiring that all the processes executed inside the *game logic* (i.e., all the procedures needed to the game execution) must be fast and reactive.

One of the main disadvantages of this game engine is that the adoption of multi-threading in videogame development is a controversial topic. The usage of threads with this engine, indeed, is not trivial since Unity is not strictly designed to be thread safe [19]. In order to keep thread safety, the Unity APIs are accessible only by the main thread. However, there could be heavy tasks that could slow down the main thread and thus the whole game. These tasks include the execution of AI algorithms, which are of special interest for our thesis. When dealing with the execution of such type of CPU-bound code, Unity offers the possibility of using *Coroutines*[3] or the *Job System*[4]. This can avoid dramatic decreases of performance.

Figure 9.1, describes the main run-time execution workflow for a Unity videogame. This workflow is mostly single-threaded, with the expectation of some parallel code in the physics engine. Game designers can customize the game behavior by implementing specific user callback functions, which are executed within the main thread. For instance, the game designer can provide her/his own code for the *FixedUpdate* block, or provide her/his own coroutine. *Coroutines* constitute a way for implementing asynchronous cooperative multitasking within a single thread.

The collection of *Game Objects* (GOs) constituting the game world, are subject to continuous updates depending on user input; on the physics simulation of the game world, and on the game logic enforced by the game designer. GOs contain a recursive hierarchy of basic properties, such as numeric, string and boolean fields, and complex properties, such as matrices, collections, nested objects, etc.

At design-time, it is possible to work on GOs using the above property-based philosophy, while the game logic can be edited by attaching scripted code to specific game events.

2 https://www.assetstore.unity3d.com/
3 https://docs.unity3d.com/Manual/Coroutines.html
4 https://docs.unity3d.com/Manual//JobSystem.html

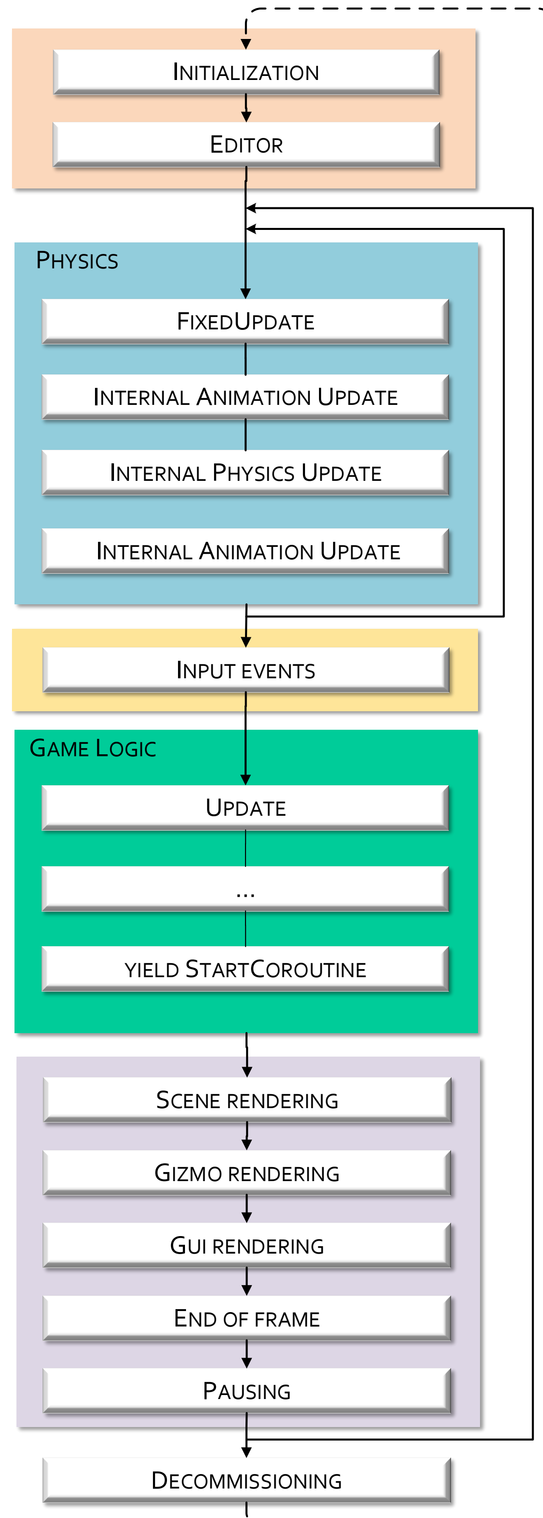


Figure 9.1: Unity run-time execution workflow

9.2 Integrating Reasoning Modules in Game Engines

When proposing a new tool, like a declarative reasoning engine, for a particular game development environment, one has to deal with the game engine of choice and its runtime execution architecture; the new tool must be reactive and easy to use. Making a declarative language module suitable for game development, indeed, is not so trivial. Even if declarative paradigms are expected to be easy to use, game developers usually are specialized only in object-oriented programming. That is why, in addition to the issues exposed in the previous chapter, new obstacles arise in the integration, like

- providing to the declarative side a good representation of the game state, and this means passing from an object-oriented representation to a logical representation;
- returning to the game main program (the procedural side) the results computed on the reasoning side, thus passing from logical assertions to object-oriented data structures;
- keeping high performance of the game while reasoning.

Providing a tool where the above problems have been solved, allows developers to focus on their main goal, i.e., create a high quality AI, instead of dealing with integration aspects. Since reasoning tasks are time-consuming and can easily slow down the game work ow if executed within the main *thread* of the game, we decided to keep a loose CPU coupling, thus letting the reasoning module run in a thread that is not the main one. Unfortunately, a common feature of almost all the game engines is that the main game execution ow is basically *thread unsafe*. A common consequence of this fact is that game engines allow the access to (parts of) the game logic APIs only to the main thread (like in Unreal Engine, Unity, Godot). This feature has an heavy impact on the data coupling level. Indeed, in the ideal configuration for a multi-thread scenario, shown in Figure 9.2, the main thread delegates the execution of a declarative specification to an external thread. Before starting the execution, the external thread gets the current state of the world. Once that the needed information is encoded in logical assertions, the specific solver (i.e., an engine capable of executing declarative specifications) is run. When the solver completes its execution, the main thread is provided with the results found. The main program can now update the objects of the game according with the solution provided by the reasoning side. One can then believe that only a form of loose data coupling is possible.

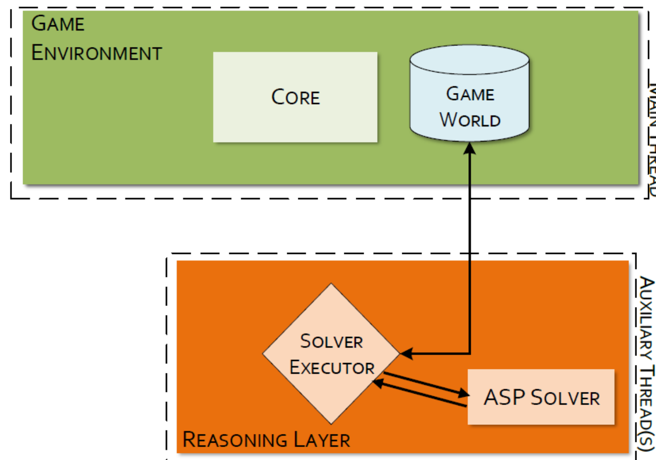


Figure 9.2: Ideal architecture for threads interaction.

However, the game logic data structures can be accessed from the declarative side by introducing a transparent information passing layer that allows to achieve a virtual tight data coupling. To this end, we aim to achieve the coupling configuration shown in Table 9.1, i.e., we aim to develop a tool in which we have loose computational coupling, but tight data coupling is obtained nonetheless.

CPU \ Data	Loose	Tight
Loose		✗
Tight		

Table 9.1: Coupling goal for the ThinkEngine framework.

9.3 The ThinkEngine Framework Architecture

Under the previous assumptions, the architecture in Figure 9.2 should be reviewed such that an external thread can be aware about what is happening in the game logic. We thus delegated reasoning tasks to auxiliary threads and introduced an information passing layer allowing the reasoning side to access and act on a representation of the game world. This representation is independent from the game engine APIs and can be accessed separately. The whole run-time *ThinkEngine* architecture is shown in Figure 9.3.

In particular the *ThinkEngine* consists of:

1. A *reasoning layer*, in which the game world is accessible and encoded in terms of logical assertions. A reasoning engine can elaborate the current state of the game and produces decisions encoded in its own format;

2. An *information passing* layer which allows to mediate between the reasoning layer and the actual game logic. In this layer, *sensors store* data originated from the upper layers. Sensors correspond to parts of the game world data structures which are visible from the reasoning layer. On the other hand, *actuators* collect decisions taken by the reasoning layer and are used to modify the game state.

3. A *reflection layer*, in which a *Sensors Manager* and an *Actuators Manager* keep the mapping between the game world data structures and the lower layers. On the one hand, the sensors manager reads selected game world data which, this way, is made accessible from the reasoning layer. On the other hand, the actuators manager updates selected parts of the game world, based on input coming from the reasoning layer.

4. One or more *brains* that can control the three layers. Each brain can access his own view of the world (i.e., a selected collection of sensors and actuators), and can be used for programming a separate reasoning activity, like a separate artificial player logic, etc.

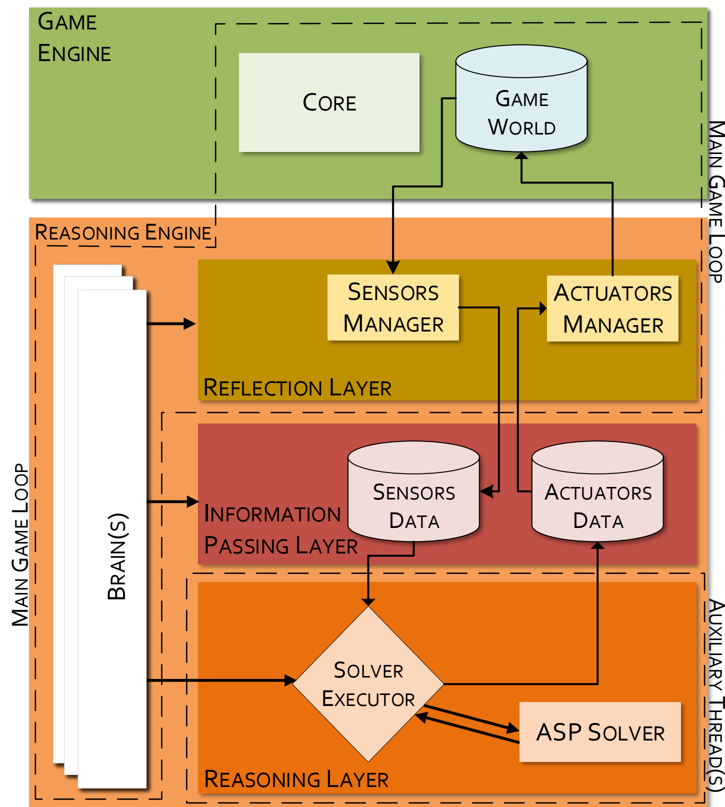


Figure 9.3: General run-time architecture of the *ThinkEngine* framework

A brain interacts with both the sensors manager and the *Solver Executor*. The former is activated periodically or when a trigger condition is met. The sensors manager is responsible of updating all the sensors data mapped to the current brain. Once that a sensors update is completed or a trigger condition is met, the brain starts the solver executor. It will generate a representation W of the world expressed in terms of logical assertions and it will invoke the solver. The solver is fed in input with W and with a logical knowledge base KB encoding the AI of the current brain. As soon as the solver provides decisions, the solver executor populates the actuators associated with the corresponding brain. The actuators manager monitors the actuators values and updates accordingly the properties of the game object associated with each actuator.

In order to better understand how sensors and actuators work, in the following we will introduce some formal definitions for the main elements of the framework. For the procedural side we will use a pseudo-object-oriented language whereas for the declarative one we will stick with the ASP notation which formal definitions and examples are given in Chapter 2.

9.4 The Information Passing Layer of the ThinkEngine Framework

In the following we give some fundamental definitions; then we will formally define both sensors and actuators semantics on the procedural and declarative side.

Definition 9.4.1 (Object Type)**.** An *object type* T is a data structure that can include multiple *direct* sub-properties. Each sub-property has a name P_T , and an associated data type D_{P_T}. Property types can be either: a basic type such as string, integer and boolean; an object type itself; or a homogeneous collection. A property P of a sub-property object type S_T of T is said to be an *indirect* property of T and is also denoted, using a dotted notation, by $T.S.P$.

Definition 9.4.2 (Element Type)**.** A collection property models either an array, a list or a vector. The *element type* $\mathcal{T}(C)$ of a collection of objects C is the type of the objects contained in the collection itself. A sub-property P of the i-th element of C is said to be an *implicit* sub-property of T and denoted as $T.C[i].\mathscr{T}(C).P$.

Definition 9.4.3 (Object Instance)**.** An *object instance* O of an object type T is a value assignment to all the sub-properties of T. For a basic type property P, a value assignment is a value of the given data type; a value assignment for an object property P' of data type T' is an object instance of type T'; a value assignment for a collection property C is a possibly empty sequence $O_1 \dots O_n$ of object instances each of type $\mathscr{T}(C)$.

Definition 9.4.4 (Frame)**.** A *frame* of a videogame is a snapshot of the game state taken at (almost) regular intervals. The *frame rate* represents the frequency at which consecutive frames are taken (for instance 60 frames per second).

9.4.1 Sensors: Definitions and Example Configurations

Procedural Side Configuration and Implementation

Definition 9.4.5 (Sensor Configuration)**.** It is given an object type T and its properties $\mathscr{P} = P_t{}^1, \ldots, P_t{}^i, T.S.P^1, \ldots, T.S.P^j, T.C.\mathscr{T}(C).P^1, \ldots, \mathscr{T}(C).P^k, i, j, k`0$. A *Sensor Configuration* SC^T is a subset of $\mathscr{P}$.

Definition 9.4.6 (Sensor Reading)**.** It is given a sensor configuration SC^T and an object instance O of type T; a *sensor reading* $v(P)$ of a direct or indirect basic property P such that $P \in SC^T$ is the value assignment of P for the object O at a specific frame of the game.

Definition 9.4.7 (Simple Sensor)**.** It is given a sensor configuration SC^T and an object instance O of type T; a *Simple Sensor* $SS(SC^T)$ is a set of sequences of sensor readings $V = \{\langle v(P_1)_1, \ldots, v(P_1)_n \rangle, \ldots, \langle v(P_m)_1, \ldots, v(P_m)_n \rangle\}$ for each direct and indirect basic property P_i such that $P_i \in SC^T$. Each sensor is associated with a unique name.

Definition 9.4.8 (Advanced Sensor)**.** It is given a sensor configuration SC^T and an object instance O of type T; an *Advanced Sensor* $AS(SC^T)$ is a simple sensor enriched with a set of n collections of simple sensors $\{C_1, \ldots, C_n\}$ where each C_i corresponds to either a bidimensional array or a list property P_i such that $P_i \in SC^T$. Each C_i contains a number of simple sensors equal to the number of object instances contained in the value assignment of P_i for the object instance O. We denote by C_j^i each simple sensor storing a single sensor reading for each implicit property $T.P_i[j].\mathscr{T}(P_i).P$.

Example 9.4.1. Suppose it is given a game object player with the following properties:

- *name* whose type is a string,
- *dead* whose type is boolean,
- *position* which is an object with three integer properties *x,y,z*, and
- *neighbors* which is a list of player instances.

The four properties *name, dead, position* and *neighbors* are direct properties of player whereas *x, y, z* are indirect properties of *player* by means of the object referred by the property *position*. The direct properties of the objects of type player contained in *neighbors* are implicit properties of player by means of the property *neighbors*. We can create a sensor configuration as

$$SC^{player} = \{name,\ position.x,\ position.y,\ position.z,\ neighbors.player.dead\}.$$

A sensor $AS(SC^{player})$, besides a name `sensorName`, will include the following data structures

$$Map\langle String,\ List\langle String\rangle\rangle\ stringProperties$$
$$Map\langle String,\ List\langle Integer\rangle\rangle\ integerProperties$$
$$Map\langle String,\ List\langle SimpleSensor\rangle\rangle\ listsProperties$$

The map `stringProperties` will contain only one entry

$$\langle name,\ \{playerName_1, \dots, playerName_n\}\rangle$$

whereas the `integerProperties` map will contain three different entries

$$\langle position.x, \{x_1, \dots, x_n\}\rangle$$
$$\langle position.y, \{y_1, \dots, y_n\}\rangle$$
$$\langle position.z, \{z_1, \dots, z_n\}\rangle$$

where $playerName_i$, x_i, y_i, z_i are the sensor readings, respectively, of the properties *name, x, y* and *z* at the i-th frame. The map `listsProperties` will contain a number of entries equal to the number of objects in the list neighbors[5].

DECLARATIVE SIDE MAPPING The *ThinkEngine* framework offers an ASP representation for each sensor that has been configured. The translations from object data structures to logical assertion is preceded by an aggregation phase in which the collection of sensor readings is replaced by the results of some aggregation function.

Definition 9.4.9 (Window)**.** A *window* is a function $w^{\#n}(l_1) = l_2$ that takes in input a sequence of values $l_1 = \langle e_1, \dots, e_m\rangle$ and returns a new collection l_2 where

$$l_2 = \begin{cases} \langle e_{m-n+1}, \dots, e_m\rangle & \text{if } m < n \\ l_1 & \text{if } m \leq n \end{cases}$$

Definition 9.4.10 (Aggregate Function)**.** An *aggregate function* $f(w^{\#n}(l_1)) = \bar{l}$ takes as argument a collection of at most n values $w^{\#n}(l_1)$ and returns a single summary value $\bar{l}$.

Examples of aggregate functions are *maximum, minimum, average, mode, median, first* and *last*.

Definition 9.4.11. It is given a simple sensor $SS(SC^T)$, an aggregate function f_i for each property P_i such that $P_i \in SC^T$ and a window function $w^{\#n}$; the *value* of P_i with respect to $SS(SC^T)$ is defined as $\bar{V}_i = f_i(w^{\#n}(V_i))$.

With this approach, each property tracked by an advanced sensor $AS(SC^T)$ will correspond to exactly one ASP atom a. The syntax of a can be described in an extended Backus-Naur form as follows:

5 Note that, currently the *ThinkEngine* implementation does not store historical readings for implicit properties: only one sensor reading at the time is materialized. At each update the previous reading will be overwritten.

$$atom = sensorName, (, gameObjectName, (, property,)).$$
$$sensorName = lowCaseString$$
$$gameObjectName = lowCaseString$$
$$property = directBasicProperty|directArrayRank2Property|$$
$$directListProperty|indirectProperty$$
$$lowCaseString = lowLetter, \{letter|digit|punctuation\}$$
$$directBasicProperty = lowCaseString, (, propertyValue,)$$
$$directArrayRank2Property = lowCaseString, (, digit,$$
$$\{digit\}, arrayType, (, directBasicProperty,))$$
$$directListProperty = lowCaseString, (, digit,$$
$$listType, (, directBasicProperty,))$$
$$indirectProperty = lowCaseString, (, property,)$$
$$propertyValue = ", lowCaseString, "|lowCaseString|$$
$$digit, [",", \{digit\}]$$
$$letter = lowLetter|upLetter$$
$$lowLetter = a - z$$
$$upLetter = A - Z$$
$$digit = 0 - 9$$
$$punctuation = ","|.|!|?$$

Definition 9.4.12 (Sensor Mapper)**.** It is given an advanced sensor $AS = AS(SC^T)$, with name μ, and an object instance O of type T, with name γ; a *sensor mapper* $MS(AS, P_i) = s_i$ is a function that takes as arguments AS and a property P_i such that $P_i \in SC^T$ and returns a string s_i representing a logical assertion encoding P_i. MS behaves differently based on the property type

- if P_i is a direct property denoted by PT then $s_i = \mu(\gamma(P_T(v(P_i))))$. where $v(P_i)$ is the value of P_T with respect to AS;
- if P_i is a indirect property denoted by $T.S.P$ then $s_i = \mu(\gamma(T(S(P(v(P_i))))))$. where $v(P_i)$ is the value of $T.S.P$ with respect to AS;
- if P_i is a bidimensional array implicit property denoted by $T.C[j][k].\mathscr{T}(C).P$ then $s_i = \mu(\gamma(T(C(j, k, \mathscr{T}(C)(P(v(P_i)))))))$. where $v(P_i)$ is the value of $T.C[j][k].\mathscr{T}(C).P$ with respect to AS;
- if P_i is a list implicit property denoted by $T.C[j].\mathscr{T}(C).P$ then $s_i = \mu(\gamma(T(C(j, \mathscr{T}(C)(P(v(P_i)))))))$. where $v(P_i)$ is the value of $T.C[j].\mathscr{T}(C).P$ with respect to AS.

Remark. In our current implementation, the size of the window function used by sensors is set to be equal to 200 in order to avoid memory and performance issues. For matrix and list properties it is not needed to apply aggregation functions as a default window of size 1 is taken.

9.4.2 Actuators: Definitions and Example Configurations

Procedural Side Configuration and Implementation

Definition 9.4.13 (Actuators Configuration)**.** It is given an object type T and its properties

$$\mathscr{P} = \{P_T^1, \ldots, P_T^i, T.S.P^1, \ldots, T.S.P^j\},\ i, j \geq 0$$

an *Actuator Configuration* AC^T for T is a subset of $\mathscr{P}$.

Definition 9.4.14 (Actuator)**.** It is given an actuator configuration AC^T and an object instance O of type T; an *Actuator* $A(AC^T)$ consists of a single value for each property P such that $P \in AC^T$. These values are used to update the corresponding property of the object instance O. Each actuator is associated with a unique name.

Example 9.4.2. Recalling the game object player of the Example 9.4.1, we can create an actuator configuration as

$$AC^{player} = \{dead,\ position.x\ position.y,\ position.z\}$$

An actuator $A(AC^{player})$, besides a name `actuatorName`, will have the following data structures

$$Map\langle String,\ Boolean\rangle\ boolProperties$$
$$Map\langle String,\ Integer\rangle\ integerProperties$$

The map `boolProperties` will contain only one entry

$$\langle dead,\ isDead\rangle$$

whereas the `integerProperties` map will contain three different entries

$$\langle position.x,\ nextX\rangle\langle position.y,\ nextY\rangle\langle position.z,\ nextZ\rangle$$

where *isDead, nextX, nextY, nextZ* are the value computed by the reasoning module and that should be set to the properties dead, *x, y* and *z*.

DECLARATIVE SIDE MAPPING Values for the properties of each actuator are retrieved from an answer set resulting from an ASP program execution. The syntax of the atoms mapped to an actuator $A(AC^T)$ can be described in an extended Backus-Naur form as follows:

$$
\begin{aligned}
& atom = \text{"}setOnActuator(\text{"}, actuatorName, (, gameObjectName, \\
& \qquad (, property,))).\\
& actuatorName = lowCaseString\\
& gameObjectName = lowCaseString\\
& property = directBasicProperty|indirectProperty\\
& lowCaseString = lowLetter, \{letter|digit|punctuation\}\\
& directBasicProperty = lowCaseString, (, propertyValue,)\\
& indirectProperty = lowCaseString, (, property,)\\
& propertyValue = lowCaseString|digit, [\text{","}, \{digit\}]\\
& letter = lowLetter|upLetter\\
& lowLetter = a - z\\
& upLetter = A - Z\\
& digit = 0 - 9\\
& punctuation = \text{","}|.|!|?
\end{aligned}
$$

Definition 9.4.15 (Actuator Mapper)**.** It is given an actuator $A = A(AC^T)$, with name μ, and an object instance O of type T, with name γ; an *actuator mapper* $M_A(A, s_i) = P_i$ is a function that takes as arguments A and a string s_i representing a logical assertion encoding a property P_i such that $P_i \in AC^T$ and returns a value assignment for P_i. M_A behaves differently based on the property type:

- if P_i is a direct property denoted by P_T then $s_i = \mu(\gamma(P_T(v(P_i))))$. where $v(Pi)$ is the value to assign to P_i for the property A;
- if P_i is a indirect property denoted by $T.S.P$ then $s_i = \mu(\gamma(T(S(P(v(P_i))))))$. where $v(P_i)$ is the value to assign to Pi for the property A.

9.5 DECLARATIVE SIDE SEMANTIC

Definition 9.5.1 (Brain)**.** We define a *brain* B as a triple $\langle SC, AC, \Pi \rangle$, where:

- $SC = \langle SC^{T_1}, \ldots, SC^{T_i} \rangle = \langle SC_1, \ldots, SC_i \rangle$ is a set of sensor configurations;
- $AC = \langle AC^{T_{i+1}}, \ldots, AC^{T_n} \rangle = \langle AC_{i+1}, \ldots, AC_n \rangle$ is a set of actuator configurations;
- Π is an ASP program.

Definition 9.5.2. It is given a brain B and

- a set of advanced sensors $AS = \langle AS(SC_1), \ldots, AS(SCT_i) \rangle = \langle AS_1, \ldots, AS_i \rangle$;
- a set of actuators $A = \langle A(AC_{i+1}), \ldots, A(AC_n) \rangle = \langle A_{i+1}, \ldots, A_n \rangle$;
- a set of sensor mappings
$\mathscr{S} = \{M_S(AS_1, P_1^{SC_1}), \ldots, M_S(AS_1, P_j^{SC_1}), \ldots, M_S(AS_i, P_1^{SC_i}), \ldots, M_S(AS_i, P_k^{SC_i})\}$;
- a set of actuator mappings
$\mathscr{A} = \{M_A(A_{i+1}, P_1^{AC_{i+1}}), \ldots, M_A(A_{i+1}, P_l^{AC^{i+1}}), \ldots, M_A(A_n, P_1^{AC_n})), \ldots, M_A(A_n, P_m^{AC_n})\}$;

Let

- $\mathscr{F}(\Pi)$ be the set of the input facts of Π;
- $Ans(\Pi \bigcup \mathscr{F}(\Pi))$ be the ordered set of the answer sets of $\Pi \bigcup \mathscr{F}(\Pi)$;
- $Ans(\Pi \bigcup \mathscr{F}(\Pi))[0]$ be the first answer set in $Ans(\Pi \bigcup \mathscr{F}(\Pi))$
- $\overline{Ans} = Ans(\Pi \bigcup \mathscr{F}(\Pi))[0] \setminus \mathscr{S}$

then $\mathscr{A}$ is a valid *decision* for B if $\mathscr{A} \subset \overline{Ans}$

9.6 THE THINKENGINE IMPLEMENTATION: UNITY AND ASP

We provided an actual implementation of the *ThinkEngine* framework deployed in the Unity 3D game engine and featuring an answer set solver at the core of the declarative side. The *ThinkEngine* has been developed as an Unity asset using the C# programming language. Every element discussed in this chapter, except for the ASP solver, has been implemented as a C# class whereas the object instances introduced in Definitions 9.4.1, 9.4.2 and 9.4.3 are Unity's game objects. Figure 9.4 shows the framework asset class diagram.

Game developers interact with the *ThinkEngine* by means of some graphical editor views at design time. Sensor and actuator configurations have been implemented as classes containing the names of the properties (either direct, indirect or implicit) of some `GameObject`. At run-time an instance of the class Brain instantiates an `AdvancedSensor` for each `SensorConfiguration` attached to it. In the same way, it instantiates a `SimpleActuator` for each `ActuatorConfiguration`. The `SensorsManager` keeps a map in which each Brain instance of the game execution is associated with its own sensors. In the same way, the `ActuatorsManager` keeps a map in which each Brain instance is associated with its own actuators. The Brain instance demands a sensors update to the `SensorManager` on a trigger event, or periodically. These trigger events are implemented by polling within a Unity co-routine some trigger boolean function. In the same way, another co-routine checks for others boolean functions in order to demand to a `SolverExecutor` instance an execution of the ASP solver. Each `SolverExecutor` runs in a separated thread that waits to be noti fied by a brain to start its computation. As

soon as the `SolverExecutor` is notified to start the execution, it demands to an `ASPAdvancedSensorMapper` for the logical assertion representation of all the sensors attached to the Brain instance. The result is written in a file and the ASP solver is invoked with both the ASP program and the input facts. Once that the solver terminates its own execution, the `SolverExecutor` sends the provided answer set to each actuator of the `Brain`. In this way, each `SimpleActuator` populates its own data structures. The `ASPAdvancedSensorMapper` translate the data structures of an `AdvancedSensor` by means of other mappers (`ASPBoolMapper`, `ASPIntegerMapper` and etc.). Each of these latter take care of translating properties of a specific type. Finally, the `ActuatorsManager` periodically checks if there are action (`SimpleActuator`) to apply to the game world. These updates are performed only if some precondition boolean function is satisfied.

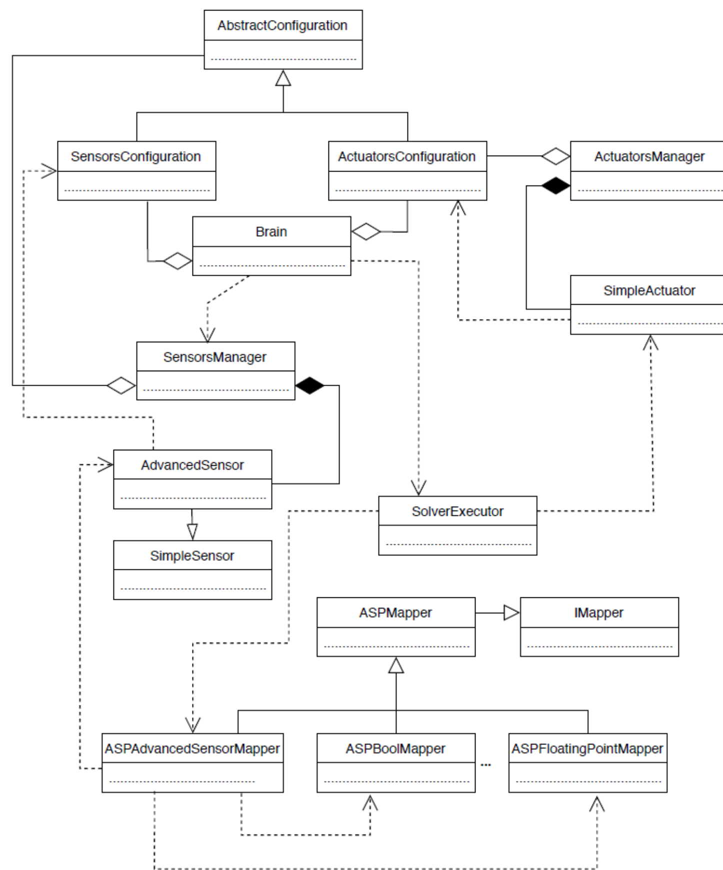


Figure 9.4: The *ThinkEngine* asset class diagram

9.7 RULE-BASED REASONING MODULES INTO UNITY AT WORK

In order to give an idea of how AI declarative modules can be integrated within Unity applications via the *ThinkEngine*, we developed two showcase games: Frogger and Tetris. In the following sections, for each of these classic videogames, we will briefly describe how our framework has been set up and configured in order to cooperate with the Unity game scene, how we configured the *sensors* and *actuators modules* and how the *brain component* were set up. Finally, we will describe our ASP *encoding*.

9.7.1 THE FROGGER SHOWCASE

We herein report about an extended version of the classic game Frogger, in which we added a sample automated player whose artificial intelligence is managed by an ASP program. In the historical Frogger game, a man-controlled frog, starting from the bottom of the screen, must cross a highway plenty of running cars (the street zone, bottom of the screen), and then jump over some logs flowing in a river (the river zone, top of the screen), in order to reach some goal cells put at the top of the screen. A safe zone (middle horizontal line) delimits the two main zones above. The frog character moves from lower coordinates ($Y = 0$) up to goal cells ($Y = 12$). We considered the basic setting of the game with logs and cars as moving characters, and not its full implementation (which included snakes, turtles, and other fancy obstacles). The game includes a time limit of 30 seconds for completing each screen. This has not been considered as a variable since our artificial player was far quicker than the time limit.

SENSORS AND ACTUATORS CONFIGURATION. When the developer wants to attach a sensor (resp. actuator) to a given GO g, it is enough to add a script component, c, to g choosing the *SensorConfigurator* (respectively, the *ActuatorConfigurator*) script. As shown in Figure 9.5, the configurator lists the configuration already available for g. One can choose to configure a new sensor or to modify or delete an existing configuration.

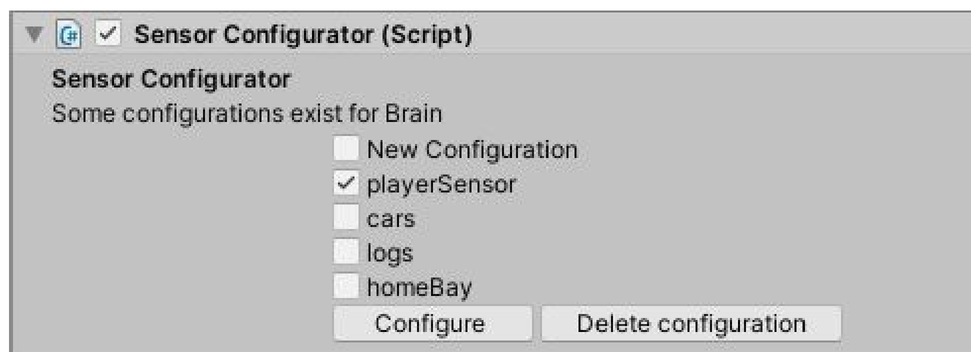


Figure 9.5: Sensors Configurator Component

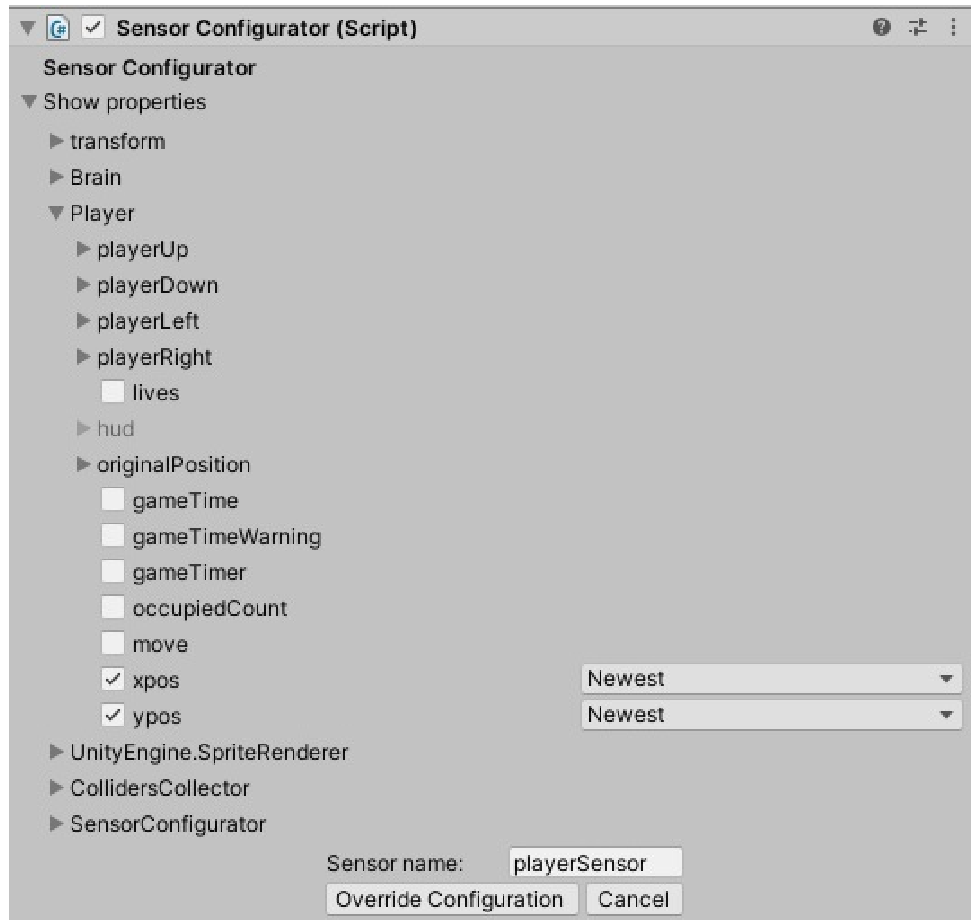


Figure 9.6: Configuring a new sensor

When configuring a new sensor, it is possible to browse g's objects and select which properties are mapped on the reasoning side, as it can be seen in Figure 9.6.

The important entities in the game are:

PLAYER: the player represents an active game character. In this case the current frog is the unique artificial player, whose goal is to reach a safe place in the game's grid;

CARS: it is a list of all the cars that can be found in the street zone of the game's grid;

LOGS: it is a list of all the logs that can be found in the river zone of the game's grid;

HOMEBAYS: it is a list of safe places that a frog has to reach to win a game and that can contain at most one frog; in order to win the match, five distinct frogs have to reach five different home bays.

We bounded to the reasoning side, as sensors, the *player*, the *logs*, the *cars* and the *home bays* and, in a similar way, we configured the only actuator needed. The single selected actuator contains the property *move* whose value can be one among: still; up; left; right; down. An example of what would look like a set of input facts for the Frogger game is given in the following:

$cars(brain(collidersCollector(cars(0,car(moveRight(false)))))).$
$cars(brain(collidersCollector(cars(0,car(xpos(2)))))).$
$cars(brain(collidersCollector(cars(0,car(ypos(1)))))).$
$playerSensor(brain(player(xpos(8)))).$
$playerSensor(brain(player(ypos(0)))).$
$logs(brain(collidersCollector(logs(0,collidableObject(leftMargin(0)))))).$
$logs(brain(collidersCollector(logs(0,collidableObject(rightMargin(2)))))).$
$logs(brain(collidersCollector(logs(0,collidableObject(y(7)))))).$
$logs(brain(collidersCollector(logs(0,collidableObject(right(true)))))).$
$home(brain(collidersCollector(bay(0,collidableObject(isOccupied(false)))))).$
$home(brain(collidersCollector(bay(0,collidableObject(leftMargin(10)))))).$
$home(brain(collidersCollector(bay(0,collidableObject(rightMargin(12)))))).$
$home(brain(collidersCollector(bay(0,collidableObject(y(12)))))).$

BRAIN COMPONENT. In addition to configuring the sensors (*playerSensor*, *cars*, *logs* and *homebays*) and the actuator called *player*, we added to the GO hierarchy a new GO with an attached component of type brain. The brain consists in a standard script belonging to the *ThinkEngine* asset that will coordinate sensors, the actuator and the solver executor.

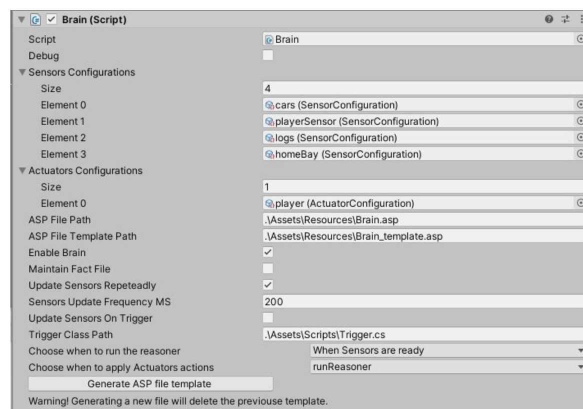


Figure 9.7: Configuration of the brain.

The brain component can be configured via the inspector tab as in Figure 9.7. Sensors, actuators and some other additional features can be attached to a brain via the visual interface. In our example, we set up the values that *a)* specify the update pace of the sensors; *b)* tell when it is triggered the ASP solver execution; and *c)* tell when it is

triggered the application of the actuators actions. The trigger conditions are boolean functions that can be customized by the developer.

During the game run-time, whenever proper conditions are met, sensors values are fed to the ASP solver, together with a knowledge base describing the desired AI. The ASP solver runs in a separate thread, and produces answer sets (i.e. a set of output logical assertions), which will be mapped to actuators by means of the actuators manager, thus influencing the game world. In the setting of the Frogger game, the solver's output encodes the next direction in which the frog should move or alternatively if it has to stay still.

ASP-ENCODED AI. The decisions taken by the AI player impersonating the frog are driven by an ASP declarative specification KB_T based on the *Guess/Check/Optimize* paradigm [22]. It is worth remarking that the goal of this work was not the development of a state-of-the-art artificial Frogger player, but rather showing the viability of our approach using a declaratively specified player strategy. The idea is to guess, at each time step, the next player's move. Considering each step independently from another, it can happen that a movement leading to a future unsafe position for the frog is chosen. The moves that lead the frog in a not admissible future position are excluded, and the optimal combination among the remaining candidates is chosen. For the sake of simplicity the optimality criterion looks only for positions that are considered safe in the current and future step and, with lesser priority, it is preferable to move the frog forward instead of backward. In this way, we ensure that the frog will be closer to an home bay at each time step. In this simple showcase AI, no planning on multiple moves is taken in consideration, although this is in principle possible.

The *guess* phase is expressed by means of the following choice rule

$$possibleMoves(still).$$
$$possibleMoves(up).$$
$$possibleMoves(left).$$
$$possibleMoves(right).$$
$$possibleMoves(down).$$
$$\{moveTo(X) : possibleMoves(X)\} = 1.$$

where the `possibleMoves` predicate describes all the admissible moves that the player could perform. Thanks to the implicit constraint included in the choice rule, we also assure that each model produced by the solver will contain exactly just one possible instance of the `moveTo` predicate.

The next positions that will result occupied or safe in the game's grid after each shot are evaluated as follows. The strategy used for the river zone is purposely simpler than the one adopted for the street zone. Therefore the movement direction of logs is not considered.

$\% \ Future\ car\ positions$

$r_1: \quad carNextPos(1,X1,Y,true) \text{ :- } car(X,Y,true), X1 = X+1.$

$r_2: \quad carNextPos(1,X1,Y,false) \text{ :- } car(X,Y,false), X1 = X-1.$

$r_3: \quad carNextPos(T1,X1,Y,true) \text{ :- } carNextPos(T,X,Y,true),$
$T1 = T+1,\ X1 = X+1,\ T1 < MT, lookAhead(MT).$

$r_4: \quad carNextPos(T1,X1,Y,false) \text{ :- } carNextPos(T,X,Y,false),$
$T1 = T+1,\ X1 = X-1,\ T1 < MT, lookAhead(MT).$

$\% \ Position\ occupied\ by\ cars\ and\ logs$

$r_5: \quad occupiedByCar(X,Y) \text{ :- } xCoord(X),\ yCoord(Y),\ carNextPos(T,X,Y,_).$

$r_6: \quad occupiedByCar(X,Y) \text{ :- } xCoord(X),\ yCoord(Y),\ car(X,Y,_).$

$r_7: \quad occupiedByLog(X,Y) \text{ :- } xCoord(X),\ log(X1,X2,Y,_),\ X \geq X1,$
$X \leq X2.$

$\% \ Safe\ positions$

$r_8: \quad safe(X,Y) \text{ :- } occupiedByLog(X,Y).$

$r_9: \quad safe(X,Y) \text{ :- } not occupiedByCar(X,Y),\ xCoord(X),\ yCoord(Y),$
$Y < 6.$

$r_{10}: \quad safe(X,Y) \text{ :- } xCoord(X),\ Y = 0.$

$r_{11}: \quad safe(X,Y) \text{ :- } xCoord(X),\ Y = 6.$

$\% \ Final\ positions\ are\ also\ safe\ (home\ positions)$

$r_{12}: \quad safe(X,Y) \text{ :- } xCoord(X),\ home(X1,X2,Y,false),\ X \geq X1,$
$X \leq X2.$

More in detail, an atom in the form `carNextPos(T,X,Y,IsRight)` is used to describe the next position in which a car will be in. The space occupied by a car is encoded by an assertion in the form `occupiedByCar(X,Y)`; similarly, the positions occupied by a log are encoded by means of the predicate `occupiedByLog`. Finally, the `safe` predicate encodes all the positions `X`, `Y` that are considered to be safe at the next step while the `home` predicate encodes the *(goal)* position that the frog has to reach in order to complete the level.

Rules from r_1 to r_4 describe which are the positions in which the cars will move in the future step. Thanks to rules r_5 and r_6, these positions are considered occupied and no longer available for the frog. Similarly, rule r_7 derives the portion of space occupied by the logs.

Finally, rules from r_8 to r_{12} describe which positions are safe for the frog. A position is considered to be safe if it is not occupied by a car or if it is occupied by a log (rules

r_8 and r_9). Moreover, also the starting position of the frog and the goal's positions are considered to be safe (rules from r_{10} to r_{12}).

The next set of rules are used to derive the X and Y coordinates of the next player position (rules from r_{13} to r_{17}). Then, we assure that the player will not move accordingly with the car in the upper line (rules r_{18} and r_{19}) otherwise the frog will never be able to cross the street. Finally, we check that the frog will not move in an unsafe position (rule r_{20}).

$$
\begin{aligned}
&\%\ Evaluating\ the\ future\ player\ position \\
r_{13}:&\quad nextPlayerPos(X,Y) \text{ :- } playerPos(X,Y), \\
&\quad moveTo(still). \\
r_{14}:&\quad nextPlayerPos(X,Y1) \text{ :- } playerPos(X,Y),\ moveTo(up), \\
&\quad Y1 = Y+1. \\
r_{15}:&\quad nextPlayerPos(X,Y1) \text{ :- } playerPos(X,Y),\ moveTo(down), \\
&\quad Y1 = Y-1. \\
r_{16}:&\quad nextPlayerPos(X1,Y) \text{ :- } playerPos(X,Y),\ moveTo(left), \\
&\quad X1 = X-1. \\
r_{17}:&\quad nextPlayerPos(X1,Y) \text{ :- } playerPos(X,Y),\ moveTo(right), \\
&\quad X1 = X+1.
\end{aligned}
$$

$$
\begin{aligned}
&\%\ Do\ not\ move\ accordingly\ with\ the\ car\ in\ the\ upperline \\
r_{18}:&\quad \text{:- } moveTo(right),\ playerPos(_,Y),\ car(_,Y1,true),\ Y1 = Y+1. \\
r_{19}:&\quad \text{:- } moveTo(left),\ playerPos(_,Y),\ car(_,Y1,false),\ Y1 = Y+1.
\end{aligned}
$$

$$
\begin{aligned}
&\%\ No\ suicide\ (move\ is\ admissible\ only\ if\ the\ position\ is\ considered\ \text{"}safe\text{"}) \\
r_{20}:&\quad \text{:- } nextPlayerPos(X,Y),\ notsafe(X,Y).
\end{aligned}
$$

$$
\begin{aligned}
&\%\ Do\ not\ move\ outside\ the\ game\ board \\
r_{21}:&\quad \text{:- } moveTo(left),\ playerPos(0,Y).
\end{aligned}
$$

The last fragment of declarative code represent optimization criteria, which are expressed in terms of *weak constraints*. Roughly speaking, a weak constraint is a condition that, if met, increases the cost of a possible frog's move. Higher cost moves are less preferred.

$\%\ Make\ a\ move\ preferring\ up.$

$\%\ Go\ back\ only\ if\ it's\ the\ unique\ possible\ move.$

$r_{22}: \quad :\sim moveTo(left).\ [1@1,2]$

$r_{23}: \quad :\sim moveTo(right).\ [2@1,3]$

$r_{24}: \quad :\sim moveTo(still).\ [3@1,0]$

$r_{25}: \quad :\sim moveTo(down).\ [4@1,4]$

Weak constraints from r_{22} to r_{25} are used to ensure that a unique best answer set is chosen. The preferred move is "up", while the "down" one is the least desiderable. It is worth noting that the left and right moves respectively have different priorities. However, when both moves lead to an optimum model, they are completely equivalent. In order to obtain a unique answer set, in this case, we arbitrarily prefer the left move over the right one.

Note that this artificial player, although not optimal, can be easily modified by either changing: *(i)* the heuristic associated to the weak constraints, or *(ii)* introducing new weak constraints expressing other desiderata, or *(iii)* changing the priority level of the constraints and so on. The above artificial player, including its declarative code, is available within the *ThinkEngine* repository reachable at `https://github.com/DeMaCS-UNICAL/ThinkEngine-Public`.

9.7.2 The Tetris showcase

We herein report about our version of the classic game Tetris. We started from a public available open-source project[6], inspired from the original game and, just like in Frogger showcase discussed in the previous Subsection, we modified this project to obtain an automated player whose artificial intelligence is managed by an ASP program. Note that we are not proposing a state-of-the-art Tetris player, rather a demonstration of how an AI can be easily developed by means of logical rules and then deployed in Unity.

Sensors and Actuators Configuration. Developers can access to a list of the GO*s* used in the game scene via a custom Unity window editor [7] as in Figure 9.8. It is possible to browse objects and select which properties are mapped on the reasoning side. We will use next some of the typical terminology used to describe our infrastructure and the Tetris game, as recalled here:

ARENA: as shown in Figure 9.8, the arena is a GO that contains all the properties relative to the playable game scene (i.e. a matrix of tiles, the properties *maxTileX*, *maxTileY* etc.);

6 `https://github.com/MaciejKitowski/Tetris`

7 I.e. a window similar to the Unity inspector. The inspector displays detailed information about the currently selected *Game Object*, including all attached components.

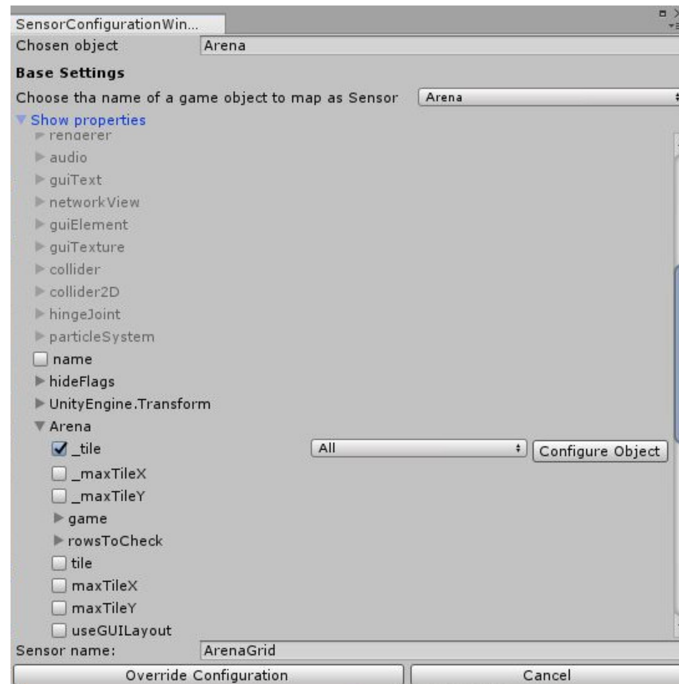


Figure 9.8: Editor window for the sensor configuration

_TILES: a matrix of GO*s* of type ArenaTile. This matrix can be expanded by the user in order to configure some extra properties;

TETROMINO: a geometric shape composed of four squares;

CURRENTTETROMINO: in the Tetris game it represents the tetromino that is currently dropping in the Arena;

SPAWNER: a GO that manages the generation of a new Tetromino when the previously created one can not drop further down in the Arena.

We bound to the reasoning side, as sensors, the *Arena*, the *currentTetromino* and the *Spawner*, and, in a similar way, we configured the actuators. By means of the *Actuator Configuration Window* one can select the AI script that is needed to be mapped within the ASP module. The single selected actuator, called *player*, contains the properties: *nMove*, *nLatMove*, *nRot*, *typeLatMove*. The meaning of these properties will be explained in the next paragraph.

BRAIN COMPONENT. After configuring the sensors (*arenaGrid*, *tetromino* and *spawner*) and the actuator (*player*), we added to the GO hierarchy a new GO with an attached component of type brain. The brain consists in a standard script belonging to the *ThinkEngine* asset that will coordinate sensors, the actuator and the solver executor. The brain component can be configured via the inspector tab. Sensors, actuators and some other additional features can be attached to a brain via the visual

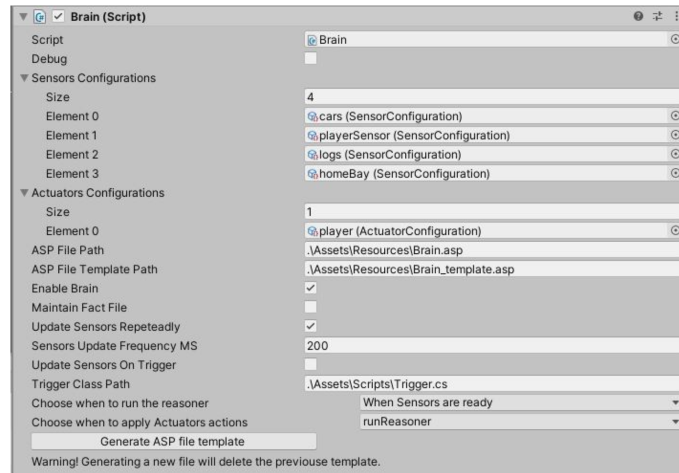


Figure 9.9: Configuration of the brain.

interface. In our example, we setup the conditions[8] to meet in order to *a)* update the sensors and run the ASP Solver; *b)* let the *Actuators Manager* apply the actuators actions.

When the game starts, thus at run-time, the brain will start updating sensors and will also run an external thread that will execute the ASP solver if sensors have data to share and the solver is not already running. Every time that it is necessary to invoke the solver, the Sensor Mapper produces a representation, in the form of logical assertions, of the filtered sensors values attached to the brain. Then, the ASP solver is invoked by passing it this sensor representation and a knowledge base KB_T expressing the desired brain AI. After the ASP solver ends its execution, its answer sets (i.e. a set of output logical assertions), will be mapped to actuators by means of the actuators manager, thus influencing the game world.

In the setting of the Tetris game, the solver's output encodes the position and orientation in which the current tetromino should be dropped. This is then translated to the corresponding number of rotations and lateral moves of the tetromino. In turn, a corresponding number of simulated swipes is commanded via Unity procedural code and the tetromino is eventually dropped.

ASP-ENCODED AI. The ASP declarative specification KB_T driving the brain decision is based on the *Guess/Check/Optimize* paradigm [22]. The idea is to range in the search space of columns of the Tetris grid and of rotations of the tetromino; to exclude combinations of columns and rotations such that the piece cannot be geometrically placed; choose the optimal combination among the remaining candidates. For the sake of simplicity the optimality criterion looks for positions not leaving holes in the grid, and with lesser priority, lower dropping positions in the grid are preferred. The

8 Conditions are selected from a set of boolean functions customized by the developer.

reader can refer to [46] for a detailed illustration of syntax and semantics of answer set programming.

The *guess* phase is expressed in the rule

$$bestSol(X,Y,C) \mid notBestSol(X,Y,C) \text{ :- } col(C),\ availableConfig(X,Y).$$

where the `availableConfig(X,Y)` predicate keeps track of all the possible rotations for the current tetromino. This assertion, combined with the strong constraints

$$\text{:- } \#count\{Y,C:\ bestSol(X,Y,C)\} \ >\ 1.$$
$$\text{:- } \#count\{Y,C:\ bestSol(X,Y,C)\} \ =\ 0.$$

assures that each model produced by the solver will contain exactly one `bestSol`.

The lowest row that the tetromino can reach when positioned with a given rotation in a chosen column is described as follows.

$r_1:$ $free(R,C,C1) \text{ :- } tile(R,C,true),\ C1 = C+1.$

$r_2:$ $free(R,C,C2) \text{ :- } free(R,C,C1),\ tile(R,C1,true), C2 = C1+1.$

$r_3:$ $firstEmpty(R) \text{ :- } nCol(C),\ \#max\{R1:\ free(R1,0,C)\} = R.$

$r_4:$ $canPut(R) \text{ :- } bestSol(X,Y,C),\ free(R,C,C1),\ firstEmpty(R),$
$confMaxW(X,Y,W),\ C1 = C+W.$

$r_5:$ $canPut(R) \text{ :- } bestSol(X,Y,C),\ canPut(R1),\ free(R,C,C1),$
$confMaxW(X,Y,W),\ C1 = C+W,\ R = R1+1.$

$r_6:$ $freeUpTo(R) \text{ :- } canPut(R),\ not\ canPut(R1),\ R1 = R+1.$

$r_7:$ $oneMore(R1) \text{ :- } bestSol(X,Y,C),\ botSpace(X,Y,I,J),\ freeUpTo(R),$
$R1 = R+1,\ free(R1,C1,C2),\ C1 = C+I,\ C2 = C+J.$

$r_8:$ $twoMore(R1) \text{ :- } bestSol(X,Y,C),\ oneMore(R),\ extraRow(X,Y),$
$botSpace(X,Y,I,J),\ free(R1,C1,C2),\ R1 = R+1,\ C1 = C+I,$
$C2 = C+J.$

$r_9:$ $bestRow(R) \text{ :- } freeUpTo(R),\ not\ oneMore(R2),\ botSpace(X,Y,0,0),$
$R2 = R+1,\ bestSol(X,Y,_).$

$r_{10}:$ $bestRow(R1) \text{ :- } freeUpTo(R),\ not\ oneMore(R2),$
$not\ extraRow(X,Y),\ bestSol(X,Y,_),\ not\ botSpace(X,Y,0,0),$
$R1 = R-1,\ R2 = R+1.$

$r_{11}:$ $bestRow(R1) : -bestSol(X,Y,_),\ not\ oneMore(R2),\ freeUpTo(R),$
$extraRow(X,Y),\ not\ botSpace(X,Y,0,0),\ R1 = R-2,\ R2 = R+1.$

$r_{12}:$ $bestRow(R) \text{ :- } oneMore(R),\ not\ twoMore(R1),\ bestSol(X,Y,_),$
$R1 = R+1,\ not\ extraRow(X,Y).$

$r_{13}:$ $bestRow(R) \text{ :- } twoMore(R).$

$r_{14}:$ $\text{:- } \#count\{R:\ bestRow(R)\} = 0.$

The `tile` predicate is used to derive in which rows the tetromino can be placed.

The space occupied by a tetromino is encoded by a number of assertions, like e.g., `confMaxW(x,y,w)` which expresses that the maximum horizontal amount of cells occupied by the tetromino `x` on which it has been applied the rotation `y` is `w`; other similar assertions are `botSpace(x,y,c,c1)`, `topSpace(x,y,h)`, `leftSpaceWrtSpawn(x,y,l)`, `extraRow(x,y)`.

Rules r_1 and r_2 describe, for each row of the arena, all the sequences of free slots of the matrix ($0-2, 0-3...0-10, 1-2, ..., 1-10...$, note that the second index is exclusive). Rule r_3 derives the highest row in the arena completely empty, thus the first row in which the tetromino can be placed in whatever column. Starting from this row, rules from r_4 to r_8 describe in which row the tetromino, in the chosen rotation configuration, is allowed to be placed, according also with the current tetromino shape. Finally, rules from r_9 to r_{13}, describe the lowest line that the tetromino will drop to.

The next set of rules describe which row the tetromino will reach (in height) once it is placed (rule r_{15}) and how many holes will remain in the row immediately below (rules from r_{16} to r_{20}).

$$
\begin{aligned}
r_{15}: \quad & reach(R) \text{ :- } bestSol(X,Y,_),\ bestRow(R1),\ topSpace(X,Y,W),\\
& R = R1 - W.\\
r_{16}: \quad & hole(R,C1) : -bestSol(X,Y,C),\ bestRow(R1),\ tile(R,C1,true),\\
& confMaxW(X,Y,W),\ R = R1 + 1,\ C1 \geq C,\ C < W1,\\
& W1 = C + W.\\
r_{17}: \quad & hole(R,C1) \text{ :- } bestSol(X,Y,C),\ botSpace(X,Y,I,J),\\
& tile(R,C1,true),\ L = I + J,\ L > 0,\ C1 \geq C,\ C1 < C2,\\
& C2 = C + I,\ oneMore(R).\\
r_{18}: \quad & hole(R,C1) \text{ :- } bestSol(X,Y,C),\ botSpace(X,Y,I,J),\\
& tile(R,C1,true),\ L = I + J,\ L > 0,\ C1 \geq C2,\ C2 = C + J,\\
& C1 < C3,\ C3 = C + W,\ oneMore(R),\ confMaxW(X,Y,W).\\
r_{19}: \quad & hole(R,C1) \text{ :- } bestSol(X,Y,C),\ botSpace(X,Y,I,J),\\
& tile(R,C1,true),\ L = I + J,\ L > 0,\ C1 \geq C,\ C1 < C2,\\
& C2 = C + I,\ twoMore(R).\\
r_{20}: \quad & hole(R,C1) \text{ :- } bestSol(X,Y,C),\ botSpace(X,Y,I,J),\\
& tile(R,C1,true),\ L = I + J,\ L > 0,\ C1 \geq C2,\ C2 = C + J,\\
& C1 < C3,\ C3 = C + W,\ twoMore(R),\ confMaxW(X,Y,W).
\end{aligned}
$$

The last fragment of declarative code represent optimization criteria, which are expressed in terms of *weak constraints*. Roughly speaking, a weak constraint is a condition that, if met, increases the cost of a possible tetromino drop configuration.

$$r_{21}: \quad :\sim \#count\{R,C: \ hole(R,C)\} = N, \ \#int(N1), \ \#int(N), \ N1 = 3*N. \ [N1:4]$$
$$r_{22}: \quad :\sim bestRow(R), numOfRows(N), D = N - R. \ [D:4]$$
$$r_{23}: \quad :\sim reach(R), numOfRows(N), D = N - R. \ [D:3]$$
$$r_{24}: \quad :\sim bestSol(X,Y,C). \ [C:2]$$
$$r_{25}: \quad :\sim bestSol(X,Y,C). \ [Y:1]$$

Weak constraints r_{21} and r_{22} have been assigned to the same priority (4) since we want, at the same time, to minimize the number of holes and to maximize the lowest line that the tetromino will drop to. However, since we want to give a bit more importance to the holes, we decided to assign a triple weight with respect to the lowest row optimization criterion. At a lower priority level, we find the minimization of the row reached in height by the tetromino (r_{23}). The last two constraints, r_{24} and r_{25}, are used to assure that no more than one answer set is produced. Indeed, when having two answer sets with the same costs for respectively the number of holes criterion, for the lowest line criterion and for the top most row criterion, we will choose the solution occupying the leftmost column and requiring the lowest number of rotations.

Note that this artificial player, although not optimal, can be easily modified by changing the heuristic associated to the weight of constraint in 21.; introducing new weak constraints expressing other desiderata; changing the priority level of the constraints and so on. The above artificial player, including both the declarative code and all the procedural code, can be downloaded at `https://github.com/DeMaCS-UNICAL/Tetris-AI4Unity`.

9.8 Benchmark

One of the most common measuring indicators used for assessing the performance of a videogame is the framerate, i.e., the number of frames that can be displayed in a second. Furthermore, it appears to be a good measure even for the evaluation of the *ThinkEngine* impact on the game performance, measured as the number of screen updates per second that can be achieved given the computational burden of the game implementation at hand. In both Frogger and Tetris showcases described in Subsections 9.7.1 and 9.7.2, we compared the framerate of the game when played by a human agent and the one obtained using the *ThinkEngine* asset. Recalling that we are using these games as a mere proof of concept of the viability of the *ThinkEngine* approach, we were instead not interested in comparing the score of the AI with respect to human players. However, when a human agent controls the game, the performance is expected to be higher, since the thinking phase is absent and substituted by a quick keyboard reading. On the other hand, if the game is controlled by *ThinkEngine*, some impact on performance is expected.

Videogames work customarily on fixed target framerate settings, where the typical default ones are 30 or 60fps. In the case of Unity games, this means that the update cycle shown in Figure 9.1 is run at a pace of 30 or 60 times per second respectively. The target rate will not be respected if an update cycle takes more than the allotted frame time.

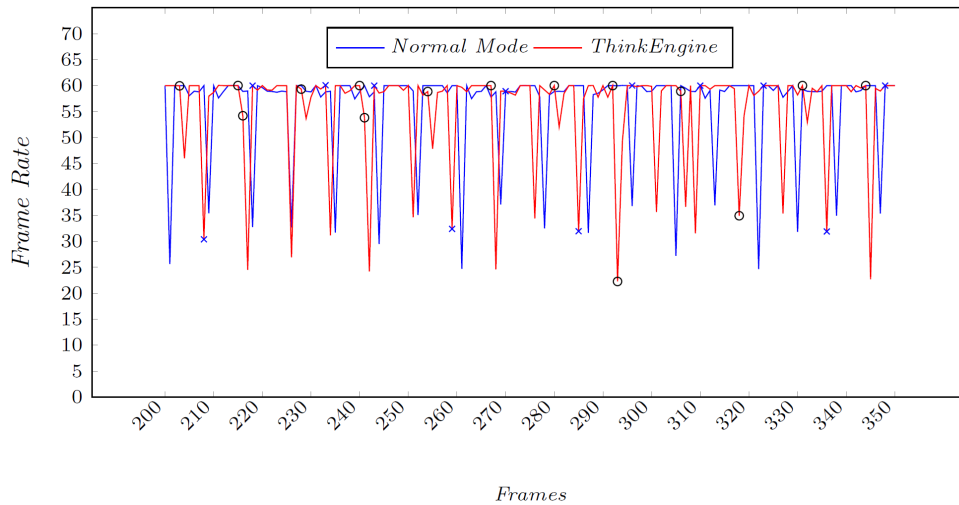


Figure 9.10: Framerate evaluation on Frogger game.

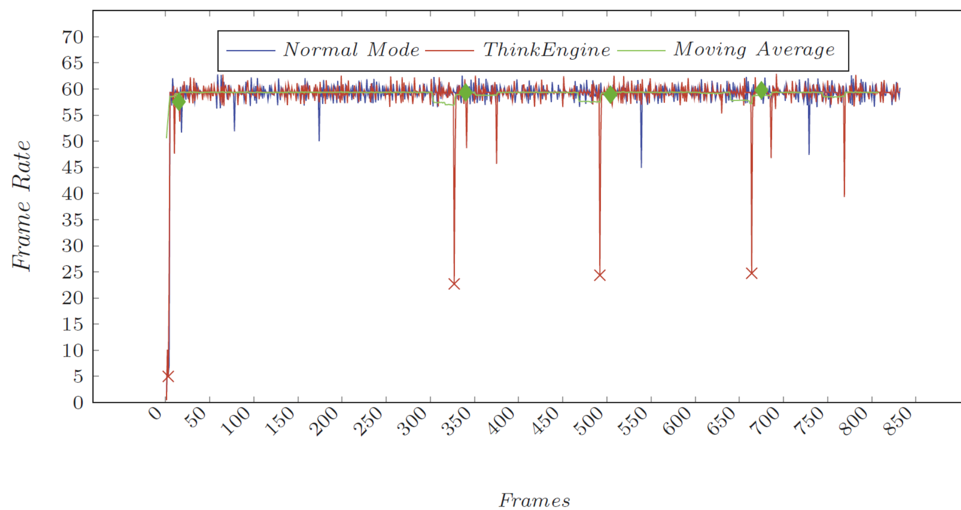


Figure 9.11: Frame rate evaluation on Tetris game.

In this section, we illustrate qualitatively the performance of our Frogger in Figure 9.10 and Tetris engines in Figure 9.11, which reports a snippet of a single run, where we set the target framerate to 60fps.

The overall framerate was almost identical in the two settings (56.03 fps in “normal” mode or also “human player” mode and 55.08 fps in *ThinkEngine* mode). Tests were

performed on a Intel CPU i7-4700MQ with 16GB of RAM and equipped with a Nvidia GeForce GPU GT740M with 1.5GB of RAM.

We show how, for our settings, the framerate is not constant, but it can vary on each frame. The two curves represent, respectively, the instantaneous framerate measured when a human is playing (blue curves) and when the game is controlled by the *ThinkEngine* (red curves). The two curves generally keep the target framerate, although they share some periodical negative spikes.

In particular, in Figure 9.10, which shows the Frogger framerate evaluation, the circles and crosses represent respectively sensor reading and actuator update events. Although the sensors track down a number of average sized list data structures, the frame rate seems to be not affected by the sensors update phase. In the same way, the actuators update phase does not influence the frame rate. The sensor reading events, which in turn trigger reasoning tasks, were purposely set to a pace of 200ms (i.e. about each 12 frames) in order to tick the decisions of the AI player at a human-like, visible speed. Concerning the generation of the logical assertions encoding the sensor values, in the Frogger setting we have a really low timing: 1ms in average. In the same way the ASP solver generates an answer set in 80ms in average. Note that, since solving tasks run in a separate thread, the impact of reasoning tasks on framerate is supposedly almost negligible.

The *ThinkEngine* framerate evaluation on Tetris game in Figure 9.11 has specific negative spikes that are caused by the overhead introduced by the sensors update phase (red crosses in the figure). These spikes do not have a visible impact on the graphical update as they are sufficiently isolated and the moving average (light green curve in figure 9.11) over 25 frames is almost constant. This analysis can be used as an indication for how often one should update the sensors: the game would stall if this is done too often. The actuators update step, instead, has no appreciable impact on the performance of the game (green diamonds in the figure). Obviously, the sensors update needs more time with respect to actuators since they have to track down an entire matrix of values on the game board.

In both cases, our expectations are met. Indeed, these spikes do not have a visible impact on the graphical update as they are sufficiently isolated and the moving average is almost constant.

Another aspect that is interesting to look at, is the time that the *ThinkEngine* needs to auto-generate the input facts for the ASP solver and how fast is this latter in producing a solution. These two measures cannot be tracked in the framerate analysis since the two operations are performed in a separate thread and the main one which is in charge of updating the graphics. However, an intuition of the amount of time elapsed between a sensors update and a solution generation can be spotted in the figure 9.11. Indeed, the number of frames between a sensors update and an actuators update is

really low. Table 9.2 shows the time needed on average on a single Tetris match for both facts and answer set generation: the last row is the overall average.

Run	**Facts (ms)**	**AnswerSets (ms)**
1	537.17	628.60
2	548.86	623.00
3	609.00	785.86
4	310.60	444.20
5	228.00	421.00
6	426.20	607.80
7	342.25	544.75
8	493.60	596.25
9	435.80	522.00
AVG	**436.83**	**574.83**

Table 9.2: Generation time

10 DECLARATIVE CONTENT SPECIFICATION VIA ANSWER SET PROGRAMMING

Declarative tools are useful also for other typical necessities of the game development world, such as content generation. Indeed, *Procedural Content Generation* (PCG) [119] is an important tool for modern videogame development, and commonly used in both triple-A (i.e., high-budget games) and indie games. A good PCG framework allows the creation of new game content without the specific need to create it by hand: instead, a program is run and the produced output is used into the game itself in the form of landscapes, playable levels, open worlds, i.e., of what we will in general call *game artifacts*.

The game production workflow usually involves a number of professional profiles, whose jobs can partially overlap. In particular, the game *designer* is in charge of manually designing and combining game artifacts, while the *programmer* writes general purpose code, *Artificial Intelligence* (AI) code, and, in our context, content generation code. Designers and programmers usually have a strict collaboration, and interact in order to produce the right content generation code. There can be cases in which the designer has little inspiration and cannot converge to a concrete game world or to suitable new levels, nor explore novel ideas, etc. In this setting *PCG* might be very beneficial. However, finding a good content generation scheme might be a big burden on programmers' shoulders. In other development contexts, designers extensively collaborate with a programmer in order to modify the content creation algorithm, so that the generated game artifacts fit to the original ideas and description.

In both settings above, input ideas about the game world come from designers in terms of high-level rules and constraints, and such information is then used by programmers to encode algorithms which should generate the content specified. This, however, means that the programmer must devise not only what is supposed to be generated, but also the procedural algorithms in charge of the generation task.

It turns then out that a general technique, reusable and decoupled from the specific game domain and visual appearance, and results accessible to non-programmers and suitable for rapidly prototyping game content, can be significantly of help. In this respect, logic-based declarative tools, such as Answer Set Programming, can be a game changer, as they limit, if not eliminate, the need for imperative code, thus achieving the above benefits in several respects. On the one hand, a skilled game designer can declaratively express quantitative and qualitative desiderata in terms of ASP code; on the other hand, programmers themselves can define content generation

strategies without the burden of programming detailed algorithms via imperative languages.

In other words, ASP can be used for evolving traditional PCG techniques to the notion of what might be better defined as Declarative Content Specification (DCS); in this respect, declarative specifications can be easily modified and incremented with new knowledge at will.

The first application of ASP in the vast literature concerning procedural content generation[1] can be found in [121, 96], with particular focus on the maze generation problem, where promising results have been achieved. In particular, the work in [121] adopts a tile-by-tile generation model, and encouraging performance results are reported over 6×6 mazes.

In this scenario, we investigated over the usage of a partition-based generation technique [118] in ASP.Approaches relying on partition-based generation are generally efficient in terms of processing time and, if mixed with ASP, can benefit from its declarative properties.

For these reason, we propose a multiple step-generation approach, set in the context of the 2-D caves generation domain and where each step is declaratively controlled by an ASP specification, as described in Section 10.2.

With respect to existing literature [121, 96], our approach promises to be better scalable to real contexts with higher size mazes. Experiments aimed at confirming that are currently ongoing.

Furthermore, in Section 10.3 we will report about the two plugins we developed based on our generation technique, which were respectively deployed as an asset available in the Unity development[2] and in the GVGAI [102] frameworks, respectively;

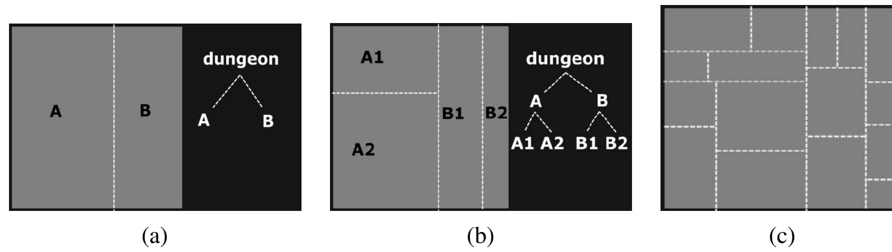
Finally, some experiments will be shown in Section 10.4 and some possible improvements will be, also, analyzed.

10.1 PROCEDURAL CONTENT GENERATION: AN OVERVIEW

Procedural Content Generation (PCG) can be defined as the algorithmic creation of game content with limited/indirect user input [128]: roughly, one can think of a software that automatically generates game contents, that might be possibly refined by a designer afterwards. There are several reasons for the use of PCG techniques in actual game development, such as fast content prototyping and improved design tasks: indeed, a software can be much faster than a designer [119], and, in general, it

1 The reader can refer to the last edition of [119] for a comprehensive survey of generation techniques and related research.

2 https://unity3d.com/unity



(a) (b) (c)

Figure 10.1: Steps of Space Partitioning. (a) First step: the level is divided by a vertical line and two sub zones are defined, A and B. (b) Further steps divide each zone into smaller areas. Here A has been divided by a horizontal line, while B has been split by a vertical line. (c) After n steps, a given criterion is met, and the level will be subdivided into sectors.

might lead to comparatively better results in less time. Furthermore, the procedural generation of game content allows to tailor the gaming experience to several extents: different challenges can be proposed according to the game session, or even depending on the player profile that is currently playing the game. In general, PCG helps to obtain a game that can be played an arbitrary number of times with always new original content, while also improving the development phases.

Game content that can be automatically created ranges from game levels to music, textures, entire worlds. We focus here on the generation of 2-D levels and caves; in particular, we focus on a PCG technique called *space partitioning*, as we found it particularly suitable when combined with the ASP approach. Space partitioning is typically used in PCG to create dungeons, both 2-D and 3-D. It works by recursively dividing the level area into smaller zones, until all meet a certain criterion, such as a specific size. Once the partitioning is done, monsters and other game objects can be placed into each "room"; eventually, rooms can be connected. One of the most popular space partitioning algorithm is the Binary Space Partitioning (BSP), which recursively divides a given "space" into two subspaces. By splitting the space into two sub-zones, the algorithm creates a binary tree. Figure 10.1 illustrates the principle of this technique. As shown in Figure 10.2, the binary space partitioning algorithm guarantees that no areas will be overlapping, and the result is very structured and uniform. Once the space has been partitioned, proper policies are adopted to create the areas and directly affect the structure of the dungeon; for instance, one can decide to randomly assign each final zone either the "room" or the "empty" property, thus creating a very symmetric dungeon. Room connections and additional level content can hence be created and placed either purely randomly, or using other techniques that don't rely on the space partitioning itself. A pseudo-code for an implementation of the binary space partitioning technique can be found in [119].

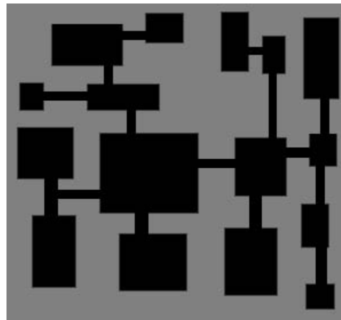


Figure 10.2: Each obtained sector is filled with a room, which will be connected afterwards using the BSP tree.

10.2 Declarative content generation with space partitioning in ASP

We will use next some typical map generation terminology, as recalled here:

MAP: a rectangular grid composed of square *tiles*;

PARTITION: the outcome of sectioning the map at hand obtained by dividing the map itself into non-overlapping parts;

AREA: a component of a partition, usually a rectangle-shaped area made of tiles;

STRUCTURE: how areas composing partitions are connected to each other, for instance by means of a door object;

TYPE: the actual kind of an area, which can be either a room, a corridor or a filled zone, i.e., an area filled with walls;

TILE: a tile can be either a wall, i.e., a cell which is not accessible by a game character, a floor cell, i.e., a cell that can be walked on by a game character. A floor cell can be possibly occupied by a game object;

OBJECT: a game object (such as keys, treasures, enemies, etc.); game objects fit in general in one or more tiles; we will assume to deal with single tile objects;

In this setting, given an empty map M of given size, the goal of a game content generation framework is to assign each tile of M an appropriate value and to properly put game objects in "floor" tiles. One could explore the search space of all the possible tile assignments: however, as soon as the required map size grows to common values for commercial
videogames, such an exhaustive search approach is not viable anymore. This is the main reason for the choice of a space partitioning approach, as it should scale better on larger maps, in principle.

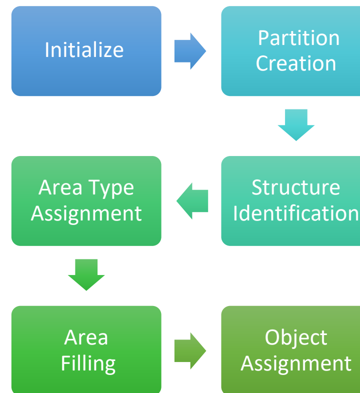


Figure 10.3: Declarative Content Generation: work-flow of the proposed approach.

Our approach is conceived on a multiple step basis: each step is implemented using a different declarative specification, written in ASP; the output of each step is processed and glued to next stages using the EmbAPS library. This tight, step-by-step, mix of imperative and declarative programming allows to overcome the limitations of both paradigms, and to cut the search space of each computation, achieving better results in terms of performance and a higher scalability, as shown in Section 10.4.
Our content generation strategy is illustrated in Figure 10.3. The behaviour of each step is summarized next:

INITIALIZE: Initial step to create the basic structure of the map, i.e., a "base" Partition;

PARTITION CREATION: Partitions are recursively created according to requirements, such as minimum size of areas, so that a raw structure is obtained similar to the one of Figure 10.4. In this step we also place the doors connecting the areas according to specified desiderata, such as minimum allowed distance from corner walls, etc.;

STRUCTURE IDENTIFICATION: A graph model of the raw map is built, where nodes represent areas and arcs represent connections between areas; note that placed doors induce connections between areas;

AREA TYPE ASSIGNMENT: Each area is assigned a type according to specified qualitative requirements, i.e., the desired density of rooms with respect to corridors, etc.;

AREA FILLING: According to the assigned area type, each tile of an area is assigned a floor type or a wall type. Three different specifications, each given by means of a logic program, are defined for rooms, corridors, and filled partitions, respectively; the three logic programs can be modified, thus allowing to tune rooms and corridor shapes according to design wishes. For instance, one can

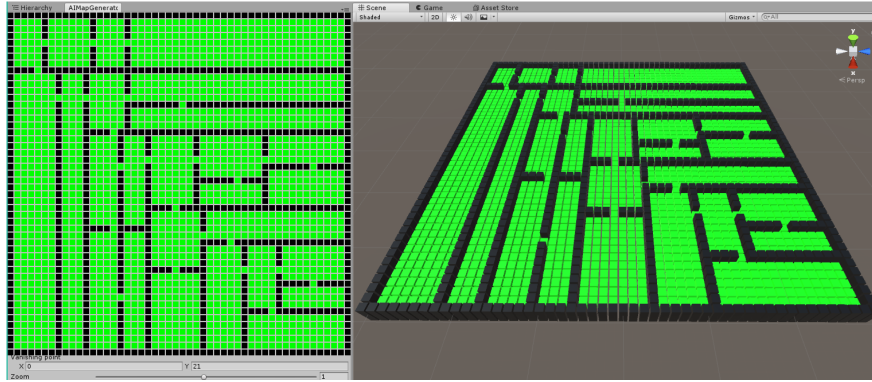


Figure 10.4: A semi-finished map obtained after the **Partition Creation** step, shown in the Unity plugin.

build square rooms instead of rounded caves by changing the corresponding specification;

OBJECT ASSIGNMENT: Eventually, game objects are assigned to selected rooms, and put in selected tiles. Note that this task is not trivial, in principle, given that some objects, such as keys, might be needed in order to enter some rooms and therefore they cannot be simply placed randomly.

We outline next some of the ASP logical specifications we used to develop the strategy described above. The **Partition Creation** step is repeatedly invoked; at each step, given a rectangular area, a partition is chosen consisting of two new rectangular areas, which share a wall having a door. The following rules, expressed in ASP, are used to achieve this result:

$$\begin{aligned}
1 = \{new_door(X,Y,D_{type}) \quad &: \quad free_cells(X,Y)\} = 1 \leftarrow door_type(D_{type}). \\
cell(X,Y,D_{type}) \quad &\leftarrow \quad new_door(X,Y,_), door_type(D_{type}). \\
\\
free_cells(X,Y) \quad &\leftarrow \quad row(X), col(Y), notunavailable_cells(X,Y). \\
\\
cell(X,Y_2,\text{“}wall'') \quad &\leftarrow \quad new_door(X,Y_1,_), col(Y_2), Y_1 \neq Y_2, \\
& \qquad orientation(horizontal), \\
& \qquad Y_2 > Min, Y_2 < Max, \\
& \qquad min_col(Min), max_col(Max). \\
cell(X_2,Y,\text{“}wall'') \quad &\leftarrow \quad new_door(X_1,Y,_), row(X_2), X_1 \neq X_2, \\
& \qquad orientation(vertical), \\
& \qquad X_2 > Min, X_2 < Max, \\
& \qquad min_row(Min), max_row(Max).
\end{aligned}$$

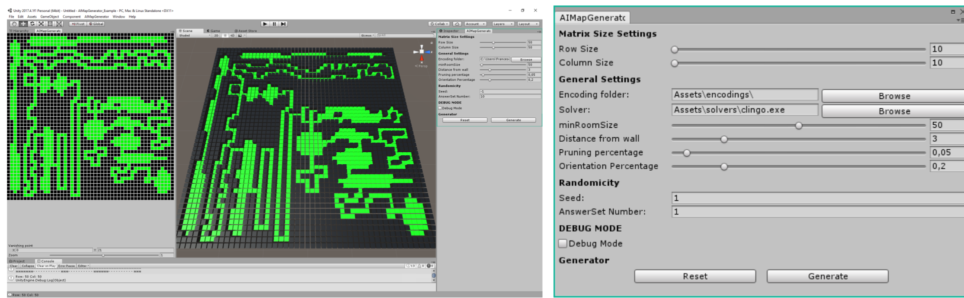


Figure 10.5: The full Unity interface. A complete map where areas are transformed in rooms and corridors is shown. On the right, it is highlighted our Unity asset.

These rules specify how the input area at hand must be sectioned by expressing where it is possible to place a door with a surrounding wall. Intuitively, the first two rules express the requirement that a new door of type D_{type} must be positioned at some point (X,Y) in the range of *free_cells*; *free_cells* are the cells that are not considered as *unavailable_cells* (there is no wall, door or a general object in position X,Y); the following rules enforce that an horizontal or a vertical (depending on the *orientation* variable) line of "walls" should start from both sides of the newly placed door. Note that the value assigned to the *orientation* variable change from one execution to another depending on the parameter *same orientation percentage* previously set. When run along with proper input data, the above specification produces a set of logical assertions in the form $cell(x,y,t)$, each telling that the tile at position (x,y) must be assigned type t (for t either a $wall$, a $vertical_door$, etc.)

10.3 Implementation: an overview

In this section we report about our prototype. It applies the techniques described above in order to generate dungeons by means of declarative languages. We deployed our application both in the Unity, as a Unity Asset, and in the GVGAI-framework, as an extension of the GVGAI-framework.

10.3.1 Unity Asset

As described in Section 9.1, Unity is a cross-platform game engine primarily used to develop videogames or simulations for more than 25 different platforms, such as mobile, computers and consoles. Its community offer a wide range of asset in the store; among them, a lot of general level generator (both for 2-D and 3-D games) exist; nevertheless, none of them give the developers the possibility to generate the game content describing it in terms of rules and constraints.

It is worth observing that the common approach to content generation is usually built on a per-game, per-level basis, with little or no opportunity of reusing the same

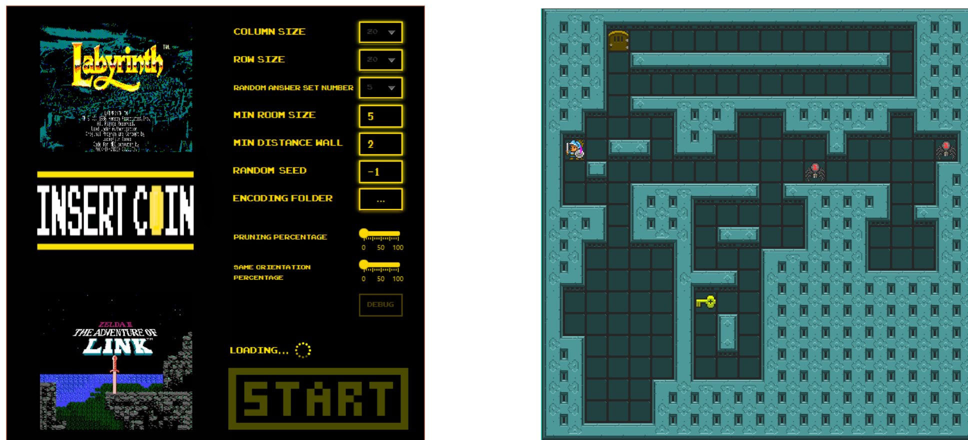


Figure 10.6: GVGAI integration.

content generation module across different games. This is reflected by the Unity Asset Store, in which almost any level generator is very tailored to specific game domains, specific graphic game elements, etc. Our framework, instead, is capable of producing general purpose content, which has very few features depending from a specific game context. As shown in Figure 10.5 our tool provides an easy-to-use graphical interface: the developer is free to set some specific parameters using the menu integrated into the Unity Editor. Among the parameters that can be set we have map size, number and minimum size of rooms, and also location of the ASP encodings; the developer can hence change the default style of the generated map. On the left side little 2-D preview of the generated map is presented, while in the middle of the Unity Editor the 3-D scene view is showed, ready to be integrated in the whole game code.

10.3.2 GVGAI PLUG-IN

GVGAI [102] is a Java framework that can be easily used to play any game described using the standard *Video Game Description Language* (VGDL) [97]. It is used for different purposes, such as an AI benchmark to test intelligent agents and as a framework for general level generation for any game. The framework is currently used for hosting the *General Video Game Player* (GVGP) competition. We deployed our application on top of the GVGAI-framework, in order to gain the possibility to reuse both games and controllers created by other participants of the GVGP competition [83]. Figure 10.6 shows Zelda game built using the GVGAI-framework and our level generator. When the application starts, the user is free to fill some fields (as in the Unity asset) and then specify the encoding folder he wants to use for the map generation. Eventually, the program generates a new example map and runs the previously selected game on the top of the generated level.

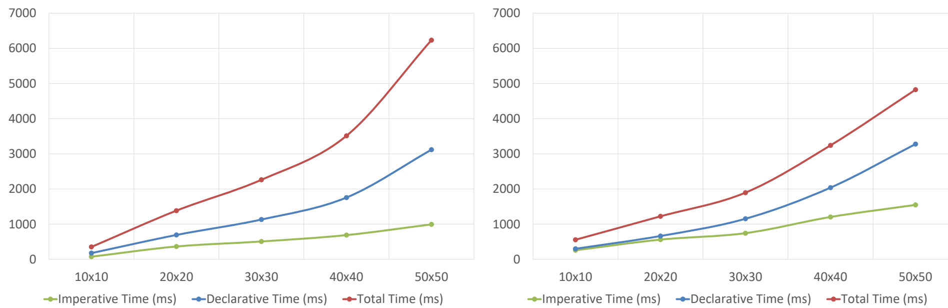


Figure 10.7: Benchmark test on scalability over Setting 1 (left side) and Setting 2 (right side). Times are reported in milliseconds.

10.4 Performance considerations and conclusions

We performed a set of experiments with a two-fold aim: on the one hand, to have a first qualitative idea of the order of magnitude of execution times, in order to prove the practical viability of the approach; on the other hand, to assess scalability with respect to maze sizes. We report here about two different cases: in the first one (Setting 1) we required the generation of mazes featuring few rooms of big size, while in the second (Setting 2) we gave input specifications for having more rooms of smaller size. We considered sizes of 10×10, 20×20, 30×30, 40×40 and 50×50; for each size we performed 3 runs for both settings 1 and 2. This led to a total of 15 executions for each setting, that produced 30 random mazes. Results are reported in Figure 10.7, where times have been averaged over the maze size. The green line shows the execution time taken by the imperative code to connect the different declarative specification written in ASP. The blue one shows the execution time of the declarative specification while the red one shows the total execution time given by the sum of imperative time and declarative time. First of all, we notice that executions take a few seconds, even for significantly large mazes. Furthermore, performance in the two settings are almost the same, even though number and shape of generated rooms are significantly different; such difference and quality of the result can be appreciated in Figure 10.8. Results confirm the effectiveness of the approach, and its capability of scaling when dealing with generation tasks of significant size.

Compared to imperative generators, the performance of our ASP-based generator can be considered good; indeed, it is sufficient enough for runtime content generation during game, and it is especially well suited for generation and manual refinement at design time. It is worth noting that, in this latter case, ASP can be of great help in the design phase, by shortening the distance between designers and programmers, especially when fast prototyping is needed.

Both versions of our prototypes, together with logic program specifications and source code are fully available online at

https://github.com/DeMaCS-UNICAL/DCS-Maze_Generator-GVGAI and https://github.com/DeMaCS-UNICAL/DCS-Maze_Generator-Unity.

Figure 10.8: Two sample maps obtained with Setting 1 (left-hand side) and with Setting 2 (right-hand side).

11 CONCLUSIONS

In this thesis, we focused on ASP-based solutions for highly dynamic environments. In particular, we addressed the realization of a novel incremental ASP grounder, able to operate over fast-paced event flows, i.e., in contexts where a high reactivity of response is required. The initial questions that inspired is are: *Which of the problems of the Stream Reasoning field can be represented using the current systems? Is it possible to use a pure* ASP *semantic to represent them?* In order to answer these questions, we investigated the current state of the art. In particular, the currently available systems were: (i) *clingo* which introduces ample flexibility but requires a non-negligible knowledge of solver-specific internal algorithms due to its extended ASP semantic; (ii) *Ticker* which extends the LARS framework trying to restore tractability but thus featuring a language which is still not quite expressive and (iii) *Laser*, another extension of the LARS framework whose implementation targets high performance, but in turn limits the LARS language to positive and stratified programs.

In order to overcome these limits, we introduced a new advanced incremental technique to promote efficient and easy to use declarative reasoning under the answer set semantic in the *Stream Reasoning* context, called *overgrounding*. Overgrounding is attractive since it fully preserves declarativity as the control of the incremental process is purposely hidden from the final user who does not need detailed knowledge of the system. Another advantage of our overgrounded programs is that, after some update iterations, they converge to a propositional theory general enough to be reused together with possible future inputs, with no further update required and virtually eliminating grounding activities in later iterations. In other words, such an overgrounded program becomes more and more general while moving from a shot to the next, increasingly adding potentially useful rules, enabling an enhancement of the time performance.

Although this technique is promising in terms of time performance, the efficiency of solvers can decrease because of larger input programs. To this end, we optimized our incremental grounding strategy to make it able to overcome the limitations of the classical overgrounding approach. This technique, called *overgrounding with tailoring*, limits the number of generated rules and reduces their size by applying known simplification methods for ground logic programs.

However, extending the overgrounding technique in such direction has not been a trivial matter. Namely, simplification criteria are, generally, applied based on specific inputs. Consequently, there is the possibility that the simplifications previously applied

could be subsequently invalidated in later shots, e.g., , consider the case in which one simplifies a ground program by properly removing a rule containing an atom which are known to be false in all answer sets at a fixed shot. In the case in which the atom that caused the deletion of the previous rule will become true in a future shot, this simplification must be invalidated in order to make the program compatible with the new set of input facts. In order to overcome these nonobvious technical obstacles, our tailored overgrounding strategy, not only makes a ground program P "reusable" with a family $\mathscr{F}$ of different inputs, but also shows how to modify P in a way such that $\mathscr{F}$ can be enlarged with small computational cost. More in detail, this techniques lets the generation of OPTs by alternating *desimplification* steps, taking care of restoring previously deleted and reduced rules and *incremental* grounding steps, which add and simplifies new generated rules. The maintained program becomes more and more general (i.e., the family of "compatible" input facts becomes increasingly larger) while moving from a shot to the next, and the update activity becomes progressively lighter.

Finally, our system was tested in order to assess the practical impact of our approach. In particular, we decided to apply our strategy in videogames since this field let the developers to "test" their techniques in a controlled, yet reproducible, environment. The results of our experiments showed that it pays off in terms of performance, by reducing grounding times and keeping solving times within more than reasonable bounds.

Despite the many advantages offered by our system, there is still a broad room for improvement. In particular, we discuss some of the possible extensions of the system in order to improve its overall performance.

(i) **Introducing some constraints and optimization techniques to reduce the memory consumption during the grounding step**. This can be fulfilled implementing some different approaches depending on the user's needs. A very simple approach could consist in setting an upper bound to memory consumption. When an overgrounded program size exceeds the memory limit one can use a custom memory trimming policy which aims to remove the less triggered rules or simply the oldest ones. Alternatively, a more complex strategy which requires the implementation of a particular heuristic, could include the possibility of marking some rules as *core* rules and the others as *non-core* rules. The *core* rules are kept in memory, whereas the others can be removed at any time and, if necessary, regenerated at the next iteration. A possible heuristic for labeling a rules as *core* or *non-core* could be based on the following parameters: (a) rule instantiation time and (b) its recent instantiation frequency.

Consider the case in which we have two rules r_1 and r_2 requiring, respectively, $t_{r_1} = 1$ second and $t_{r_2} = 0.02$ seconds to be instantiated from scratch. Rule r_1 recently fired on average every $f_{r_1} = 5$ shots, whereas rule r_2 fired recently almost at every shot ($f_{r_2} = 1$). r_1 could be given better priority, and thus be tagged as a *core* rule, in that it features a ratio t_{r_1} / f_{r_1} higher than t_{r_2} / f_{r_2}.

Moreover, a similar strategy could also offer the possibility of forgetting the extension of a selected predicate instead of rules, i.e., a heuristics could also choose to completely forgot all the extension relative to a specific predicate p. By applying this strategy, also all the rules which contain p in the head or in the body must be deleted. This operation affects (positively), also on the size of the ground program in terms of simplifications. Indeed, when a previously forgot predicate p_f comes back, all the new rules generated can be simplified also on the base of p_f.

(ii) **Implementing a memory caching strategy**. This technique could use also mass memory, allowing to efficiently cache and swap to disk those data computed by our grounding algorithm. It could be particularly useful in situations when the quantity of memory used by stored rules exceed a certain limit. An external agent working in background, could intelligently determine, on the basis of particular heuristics similar to the ones described above, which rules must be saved on the disk and which ones need to be kept on the main memory. This strategy could reduce not only the main memory footprint and the grounding time, but also the loading time from disk between one shot to another.

(iii) **Developing an incremental solver considering also the model generation phase**. The solving times over overgrounded programs are obviously greater than solving times over simplified input instances. This time could be reduced if a solver could be fed at each shot with only the newly computed ground rules. In this case it is not necessary to read at each iteration the entire ground program but only the new incremental rules. This optimization can be improved even further: in order to allow reactive reasoning tasks lasting in the order of tenths of milliseconds, and to also make repeated non-reactive tasks considerably more performant, our overgrounding techniques could be broadened by extending the overgrounding approach to the solving step. A solver can update its internal state incrementally depending on modified inputs. This result can be achieved exploiting existing benefits of our incremental evaluation strategy, i.e., the absence of the manual development effort necessary for controlling differential computations.

(iv) **Introducing the possibility to *stop and restart* the computation of partial answer set**. This extension could be reached formalizing the notion of intermediate solver state: based on this formalization, we aim to devise and implement an appropriate *stop and restart* scheme for reasoning tasks. It is important to point out that, the strategy we are proposing here differs from the canonical solving process on a single fixed input. Namely, in our approach, we aim for adding the possibility to stop and restart the computation of the answer sets when the system understands that the current knowledge base has been changed by adding or removing assertions, so the set of candidate solutions is no more the same and it is needed to restart the computation over the new set of facts.

It is important to note that the properties of overgrounded programs could facilitate the use of *stops and restarts* techniques.

The $\mathscr{I}^2$-*DLV* binaries can be downloaded from the official repository, where a detailed documentation about the system usage and the reproducibility of the experiments presented in this thesis work is also available [34].

BIBLIOGRAPHY

[1] Black & white. game. https://bit.ly/2SF4mS0, 2001.

[2] Halo. game. https://bit.ly/2RTBghN, 2001.

[3] Newzoo global games market report. https://bit.ly/3vIixop, 2019.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] Z. A. Algfoor, M. S. Sunar, and H. Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *Int. J. Comput. Games Technol.*, 2015:736138:1–736138:11, 2015.

[7] M. Alviano, C. Dodaro, N. Leone, and F. Ricca. Advances in WASP. In F. Calimeri, G. Ianni, and M. Truszczynski, editors, *LPNMR 2015*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015.

[8] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[9] K. Arulkumaran, A. Cully, and J. Togelius. Alphastar: an evolutionary computation perspective. In M. López-Ibáñez, A. Auger, and T. Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 314–315. ACM, 2019.

[10] M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The usa-advisor: A case study in answer set planning. In T. Eiter, W. Faber, and M. Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*, pages 439–442. Springer, 2001.

[11] C. Baral, T. H. Nam, and L. Tuan. Reasoning about actions in a probabilistic setting. In R. Dechter, M. J. Kearns, and R. S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 507–512. AAAI Press / The MIT Press, 2002.

[12] H. R. Bazoobandi, H. Beck, and J. Urbani. Expressive stream reasoning with laser. In *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, pages 87–103, 2017. Prototype available at https://github.com/karmaresearch/Laser.

[13] 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 *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–6. IEEE, 2017.

[14] H. Beck, M. Dao-Tran, and T. Eiter. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.*, 261:16–70, 2018.

[15] H. Beck, M. Dao-Tran, T. Eiter, and C. Folie. Stream reasoning with LARS. *Künstliche Intell.*, 32(2-3):193–195, 2018.

[16] H. Beck, T. Eiter, and C. Folie. Ticker: A system for incremental asp-based stream reasoning. *TPLP*, 17(5-6):744–763, 2017.

[17] L. E. Bertossi and L. Bravo. Consistent query answers in virtual data integration systems. In L. E. Bertossi, A. Hunter, and T. Schaub, editors, *Inconsistency Tolerance [result from a Dagstuhl seminar]*, volume 3300 of *Lecture Notes in Computer Science*, pages 42–83. Springer, 2005.

[18] J. Bomanson, T. Janhunen, and A. Weinzierl. Enhancing lazy grounding with lazy normalization in answer-set programming. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 2694–2702. AAAI Press, 2019.

[19] J. Bonastre. Why should i use threads instead of coroutines. *Dosegljivo: https://support. unity3d. com/hc/en-us/articles/208707516-Why-should-I-use-Threads-instead-of-Coroutines*, 2016.

[20] L. Bravo and L. E. Bertossi. Logic programs for consistently querying data integration systems. In G. Gottlob and T. Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 10–15. Morgan Kaufmann, 2003.

[21] G. Brewka, T. Eiter, and M. Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[22] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In *LPNMR*, pages 2–17. Springer, 1997.

[23] F. Buccafurri, N. Leone, and P. Rullo. Enhancing disjunctive datalog by constraints. *IEEE Trans. Knowl. Data Eng.*, 12(5):845–860, 2000.

[24] A. Calì, G. Gottlob, and T. Lukasiewicz. Tractable query answering over ontologies with datalog+/-. In *Description Logics*, volume 477 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[25] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP: theory and implementation. In M. G. de la Banda and E. Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424. Springer, 2008.

[26] 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, 2015. https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf.

[27] F. Calimeri, D. Fuscà, S. Perri, and J. Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.

[28] F. Calimeri, M. Gebser, M. Maratea, and F. Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231:151–181, 2016.

[29] F. Calimeri, S. Germano, G. Ianni, F. Pacenza, S. Perri, and J. Zangari. Integrating rule-based AI tools into mainstream game development. In *RuleML+RR 2018*, volume 11092 of *LNCS*, pages 310–317, 2018.

[30] F. Calimeri, G. Ianni, F. Pacenza, S. Perri, and J. Zangari. Incremental answer set programming with overgrounding. *TPLP*, 19(5-6):957–973, 2019.

[31] F. Calimeri, G. Ianni, S. Perri, and J. Zangari. The eternal battle between determinism and nondeterminism: preliminary studies in the sudoku domain. *20th RCRA International Workshop. 2013*, 2013.

[32] F. Calimeri, G. Ianni, and F. Ricca. The third open answer set programming competition. *Theory Pract. Log. Program.*, 14(1):117–135, 2014.

[33] F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: Preliminary report of the system competition track. In J. P. Delgrande and W. Faber, editors, *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2011.

[34] F. Calimeri, S. Perri, D. Fuscà, J. Zangari, and F. Pacenza. $\mathscr{I}$-DLV repository, since 2016. https://github.com/DeMaCS-UNICAL/I-DLV/wiki/Incremental-IDLV.

[35] F. Calimeri, S. Perri, and J. Zangari. Optimizing answer set computation via heuristic-based decomposition. *TPLP*, 19(4):603–628, 2019.

[36] T. J. Cerexhe, M. Gebser, and M. Thielscher. Online agent logic programming with oclingo. In D. N. Pham and S. Park, editors, *PRICAI 2014: Trends in Artificial Intelligence - 13th Pacific Rim International Conference on Artificial Intelligence, Gold Coast, QLD, Australia, December 1-5, 2014. Proceedings*, volume 8862 of *Lecture Notes in Computer Science*, pages 945–957. Springer, 2014.

[37] M. Certický, D. Churchill, K. Kim, M. Certický, and R. Kelly. Starcraft AI competitions, bots, and tournament manager software. *IEEE Trans. Games*, 11(3):227–237, 2019.

[38] A. Colmerauer and P. Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. 1996.

[39] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.

[40] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity, Ulm, Germany, June 24-27, 1997*, pages 82–101. IEEE Computer Society, 1997.

[41] D. Dell'Aglio, E. D. Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. *Data Sci.*, 1(1-2):59–83, 2017.

[42] T. M. Do, S. W. Loke, and F. Liu. Answer set programming for stream reasoning. In C. J. Butz and P. Lingras, editors, *Advances in Artificial Intelligence - 24th Canadian Conference on Artificial Intelligence, Canadian AI 2011, St. John's, Canada, May 25-27, 2011. Proceedings*, volume 6657 of *Lecture Notes in Computer Science*, pages 104–109. Springer, 2011.

[43] J. Doyle. A truth maintenance system. *Artif. Intell.*, 12(3):231–272, 1979.

[44] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem-solving using the dlv system. In *Logic-based artificial intelligence*, pages 79–103. Springer, 2000.

[45] T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In S. Tessaris, E. Franconi, T. Eiter, C. Gutiérrez, S. Handschuh, M. Rousset, and R. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[46] T. Eiter, G. Ianni, and T. Krennwallner. *Answer Set Programming: A Primer*, pages 40–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[47] T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In S. Tessaris, E. Franconi, T. Eiter, C. Gutiérrez, S. Handschuh, M. Rousset, and R. A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[48] T. Eiter, P. Ogris, and K. Schekotihin. A distributed approach to LARS stream reasoning (system paper). *TPLP*, 19(5-6):974–989, 2019.

[49] E. Erdem, M. Gelfond, and N. Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.

[50] E. Erdem, V. Patoglu, and P. Schüller. A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks. *AI Commun.*, 29(2):319–349, 2016.

[51] W. Faber, N. Leone, and S. Perri. The intelligent grounder of DLV. In E. Erdem, J. Lee, Y. Lierler, and D. Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2012.

[52] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *JELIA*, volume 3229 of *LNCS*, pages 200–212. Springer, 2004.

[53] W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011.

[54] F. Fages. Consistency of clark's completion and existence of stable models. *Meth. of Logic in CS*, 1(1):51–60, 1994.

[55] A. A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, and E. C. Teppan. Industrial applications of answer set programming. *Künstliche Intell.*, 32(2-3):165–176, 2018.

[56] O. Febbraro, N. Leone, G. Grasso, and F. Ricca. JASP: A framework for integrating answer set programming with java. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*, 2012.

[57] 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 the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom*, pages 38–49, 2016.

[58] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Stream reasoning with answer set programming: Preliminary report. In G. Brewka, T. Eiter, and S. A. McIlraith, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press, 2012.

[59] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. Reactive answer set programming. In J. P. Delgrande and W. Faber, editors, *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2011.

[60] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In M. G. de la Banda and E. Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer, 2008.

[61] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub. Progress in clasp series 3. In *LPNMR 2015*, volume 9345 of *LNCS*, pages 368–383. Springer, 2015.

[62] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[63] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-shot ASP solving with clingo. *TPLP*, 19(1):27–82, 2019.

[64] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In *LPNMR*, volume 6645 of *LNCS*, pages 345–351. Springer, 2011.

[65] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, pages 15–19, 2008.

[66] M. Gebser, M. Maratea, and F. Ricca. What's hot in the answer set programming competition. In D. Schuurmans and M. P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 4327–4329. AAAI Press, 2016.

[67] M. Gebser, M. Maratea, and F. Ricca. The design of the seventh answer set programming competition. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 3–9. Springer, 2017.

[68] M. Gebser, M. Maratea, and F. Ricca. The sixth answer set programming competition. *J. Artif. Intell. Res.*, 60:41–95, 2017.

[69] M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In C. Baral, G. Brewka, and J. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483, pages 266–271, Tempe, Arizona, May 2007. Springer Verlag.

[70] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, WA, Aug 15-19, 1988 (2 Volumes)*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[71] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[72] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[73] I. P. Gent, C. Jefferson, and P. Nightingale. Complexity of n-queens completion (extended abstract). In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5608–5611. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[74] S. Germano, T. Pham, and A. Mileo. Web stream reasoning in practice: On the expressivity vs. scalability tradeoff. In B. ten Cate and A. Mileo, editors, *Web Reasoning and Rule Systems - 9th International Conference, RR 2015, Berlin, Germany, August 4-5, 2015, Proceedings*, volume 9209 of *Lecture Notes in Computer Science*, pages 105–112. Springer, 2015.

[75] B. Kaufmann, N. Leone, S. Perri, and T. Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.

[76] C. Lefèvre, C. Béatrix, I. Stéphan, and L. Garcia. Asperix, a first-order forward chaining approach for answer set computing. *TPLP*, 17(3):266–310, 2017.

[77] D. B. Lenat, R. V. Guha, K. Pittman, D. Pratt, and M. Shepherd. CYC: toward programs with common sense. *Commun. ACM*, 33(8):30–49, 1990.

[78] N. Leone, T. Eiter, W. Faber, M. Fink, G. Gottlob, L. Granata, G. Greco, E. Kalka, G. Ianni, D. Lembo, M. Lenzerini, V. Lio, B. Nowicki, R. Rosati, M. Ruzzi, W. Staniszkis, and G. Terracina. Data integration: a challenging ASP application. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 379–383. Springer, 2005.

[79] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, Jul 2006.

[80] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. 7(3):499–562, July 2006.

[81] N. Leone and F. Ricca. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In W. Faber and A. Paschke, editors, *Reasoning Web. Web Logic Rules - 11th International Summer School 2015, Berlin, Germany, July 31 - August 4, 2015, Tutorial Lectures*, volume 9203 of *Lecture Notes in Computer Science*, pages 308–326. Springer, 2015.

[82] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135(2):69–112, 1997.

[83] D. P. Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas. General video game AI: competition, challenges and opportunities. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 4335–4337, 2016.

[84] Y. Lierler, M. Maratea, and F. Ricca. Systems, engineering environments, and competitions. *AI Magazine*, 37(3):45–52, 2016.

[85] V. Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[86] V. Lifschitz. What is answer set programming? In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597. AAAI Press, 2008.

[87] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Trans. Comput. Log.*, 2(4):526–541, 2001.

[88] M. Manna, F. Ricca, and G. Terracina. Consistent query answering via ASP from different perspectives: Theory and practice. *Theory Pract. Log. Program.*, 13(2):227–252, 2013.

[89] V. W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. In K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm - A 25-Year Perspective*, Artificial Intelligence, pages 375–398. Springer, 1999.

[90] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial intelligence*, 13(1-2):27–39, 1980.

[91] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth. Streamrule: A nonmonotonic stream reasoning system for the semantic web. In W. Faber

and D. Lembo, editors, *Web Reasoning and Rule Systems - 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013. Proceedings*, volume 7994 of *Lecture Notes in Computer Science*, pages 247–252. Springer, 2013.

[92] A. Mileo, D. Merico, and R. Bisiani. Non-monotonic reasoning supporting wireless sensor networks for intelligent monitoring: The SINDI system. In E. Erdem, F. Lin, and T. Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 585–590. Springer, 2009.

[93] A. Mileo, D. Merico, S. Pinardi, and R. Bisiani. A logical approach to home healthcare with intelligent sensor-network support. *Comput. J.*, 53(8):1257–1276, 2010.

[94] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[95] B. Motik, Y. Nenov, R. Piro, and I. Horrocks. Maintenance of datalog materialisations revisited. *Artificial Intelligence*, 269:76–136, 2019.

[96] M. J. Nelson and A. M. Smith. *ASP with Applications to Mazes and Levels*, pages 143–157. Springer International Publishing, Cham, 2016.

[97] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson. Towards generating arcade game rules with vgdl. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 185–192, Aug 2015.

[98] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

[99] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A prolog decision support system for the space shuttle. In A. Provetti and T. C. Son, editors, *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, CA, USA, March 26-28, 2001*, 2001.

[100] J. Orkin. Three states and a plan: the ai of fear. In *Game developers conference*, volume 2006, page 4, 2006.

[101] M. Ortiz. Ontology based query answering: The story so far. In *Proc. of AMW*, 2013.

[102] D. Pérez-Liébana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas. General video game AI: a multi-track framework for evaluating agents, games and content generation algorithms. *CoRR*, abs/1802.10363, 2018.

[103] D. Perez-Liebana, S. M. Lucas, R. D. Gaina, J. Togelius, A. Khalifa, and J. Liu. *General Video Game Artificial Intelligence*, volume 3. Morgan & Claypool Publishers, 2019. https://gaigresearch.github.io/gvgaibook/.

[104] D. Pérez-Liébana, S. Samothrakis, J. Togelius, T. Schaul, and S. Lucas. General video game AI: competition, challenges and opportunities. In *AAAI 2016*, pages 4335–4337, 2016.

[105] D. Pérez-Liébana, S. Samothrakis, J. Togelius, T. Schaul, and S. Lucas. General video game AI: competition, challenges and opportunities. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 4335–4337, 2016.

[106] T. Pham, A. Mileo, and M. I. Ali. Towards scalable non-monotonic stream reasoning via input dependency analysis. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1553–1558. IEEE Computer Society, 2017.

[107] É. Piette, C. Browne, and D. J. N. J. Soemers. Ludii game logic guide. *CoRR*, abs/2101.02120, 2021.

[108] T. C. Przymusinski. Stable semantics for disjunctive programs. *New Gener. Comput.*, 9(3/4):401–424, 1991.

[109] R. Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1-2):81–132, 1980.

[110] J. Renz, X. Ge, S. Gould, and P. Zhang. The angry birds AI competition. *AI Mag.*, 36(2):85–87, 2015.

[111] F. Ricca. The DLV Java Wrapper. In M. de Vos and A. Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 305–316, Messina, Italy, Sept. 2003. Online at http://CEUR-WS.org/Vol-78/.

[112] F. Ricca, W. Faber, and N. Leone. A backjumping technique for disjunctive logic programming. *AI Commun.*, 19(2):155–172, 2006.

[113] 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. *Theory Pract. Log. Program.*, 12(3):361–381, 2012.

[114] S. Risi and M. Preuss. Behind deepmind's alphastar AI that reached grandmaster level in starcraft II. *Künstliche Intell.*, 34(1):85–86, 2020.

[115] Z. G. Saribatur, V. Patoglu, and E. Erdem. Finding optimal feasible global plans for multiple teams of heterogeneous robots using hybrid reasoning: an application to cognitive factories. *Auton. Robots*, 43(1):213–238, 2019.

[116] T. Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, August 11-13, 2013*, pages 1–8. IEEE, 2013.

[117] P. Schüller and A. Weinzierl. Answer set application programming: a case study on tetris. In M. D. Vos, T. Eiter, Y. Lierler, and F. Toni, editors, *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015*, volume 1433 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[118] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra. *Constructive generation methods for dungeons and levels*, pages 31–55. Springer International Publishing, Cham, 2016.

[119] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Springer, 2016.

[120] 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. *Nat.*, 529(7587):484–489, 2016.

[121] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, Sept 2011.

[122] T. Syrjänen. Lparse 1.0 User's Manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[123] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[124] G. Tesauro. Practical issues in temporal difference learning. *Mach. Learn.*, 8:257–277, 1992.

[125] G. Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[126] M. Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, 2014.

[127] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design (ICED'03)*, pages 1290–1299, 2003.

[128] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[129] R. Tommasini, M. Balduini, and E. D. Valle. Towards a benchmark for expressive stream reasoning. In J. Calbimonte, M. Dao-Tran, D. Dell'Aglio, D. L. Phuoc, M. Saleem, R. Usbeck, R. Verborgh, and A. N. Ngomo, editors, *Joint Proceedings of the 2nd RDF Stream Processing (RSP 2017) and the Querying the Web of Data (QuWeDa 2017) Workshops co-located with 14th ESWC 2017 (ESWC 2017), Portoroz, Slovenia, May 28th - to - 29th, 2017*, volume 1870 of *CEUR Workshop Proceedings*, pages 26–36. CEUR-WS.org, 2017.

[130] M. Truszczynski and S. Woltran. Relativized hyperequivalence of logic programs for modular programming. *TPLP*, 9(6):781–819, 2009.

[131] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*, volume 14 of *Principles of computer science series*. Computer Science Press, 1988.

[132] E. D. Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intell. Syst.*, 24(6):83–89, 2009.

[133] I. Vlachopoulos, S. Vassos, and M. Koubarakis. Flexible behavior for worker units in real-time strategy games using STRIPS planning. In A. Likas, K. Blekas, and D. Kalles, editors, *Artificial Intelligence: Methods and Applications - 8th Hellenic Conference on AI, SETN 2014, Ioannina, Greece, May 15-17, 2014. Proceedings*, volume 8445 of *Lecture Notes in Computer Science*, pages 555–568. Springer, 2014.

[134] A. Zarrad. Game engine solutions. *Simulation and Gaming*, pages 75–87, 2018.