UNIVERSITÀ DELLA CALABRIA

UNIVERSITA' DELLA CALABRIA

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in

Matematica e Informatica

Con il contributo del MIUR

XXVII Ciclo

Answer Set Programming: Development Tools and Applications to Tourism.

Settore Scientifico Disciplinare INF/01 – INFORMATICA

Coordinatore: Ch.mo Prof. Nicola Leone

Supervisore/Tutor: Prof. Giorgio Terracina

Dottorando: Dott.ssa Barbara Nardi

UNIVERSITÀ DELLA CALABRIA

UNIVERSITA' DELLA CALABRIA

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in

Matematica e Informatica

Con il contributo del MIUR

XXVII Ciclo

Answer Set Programming: Development Tools and Applications to Tourism.

Settore Scientifico Disciplinare INF/01 – INFORMATICA

Coordinatore: Ch.mo Prof. (Nicola Leone)
Firma [illegible]

Supervisore/Tutor: Prof. (Giorgio Terracina)
Firma Terracina

Dottorando: Dott./ssa (Barbara Nardi)
Firma Barbara Nardi

A Giulia e Francesco,
i grandi amori della mia vita!

Sommario

L'Answer Set Programming (ASP) è un paradigma di programmazione dichiarativo che è basato sulla semantica dei modelli stabili. L'idea di base dell'ASP è quella di codificare un problema computazionale in un programma logico i cui modelli stabili (answer set) corrispondono alle soluzioni del problema. Negli ultimi anni sono state sviluppate numerose applicazioni di ASP che testimoniano un crescente interesse per questo paradigma di programmazione sia in ambito accademico che in ambito industriale. Lo sviluppo di tali applicazioni ha fornito e fornisce tuttora indicazioni importanti sulle reali potenzialità di tale paradigma di programmazione per la soluzione pratica di problemi complessi, e contemporaneamente consente di evidenziare alcuni aspetti critici da affrontare per rendere più efficace ed agevole l'utilizzo di ASP in contesti reali.

Questa tesi offre vari contributi in questo contesto che possono essere riassunti come segue:

(i) sviluppo di due applicazioni di ASP in uno specifico ambito industriale;

(ii) progettazione ed implementazione di nuovi strumenti di sviluppo per ASP.

Per quanto riguarda il primo punto, la tesi affronta due problemi molto sentiti nel settore turistico. Il primo è noto in letteratura come il problema dell'allotment (semi-)automatico di pacchetti turistici; ed il secondo riguarda la gestione intelligente di un servizio newsletter personalizzata per i clienti di un'agenzia viaggio. Tali soluzioni confermano le buone qualità di ASP come strumento efficace per la risoluzione di complessi problemi reali.

Per quanto riguarda il secondo punto, la tesi descrive due nuovi strumenti di sviluppo che estendono ASPIDE, uno dei più noti ambienti integrati di sviluppo per ASP. Il primo strumento ha come obiettivo quello di agevolare la scrittura di programmi logici per gli utenti meno esperti della sintassi di ASP, ed è particolarmente indicato per quegli utenti che prediligono l'utilizzo di interfacce grafiche. Si tratta di un nuovo strumento di programmazione visuale che consente di "disegnare" un programma ASP componendone le regole in modo completamente grafico. Il secondo strumento di sviluppo descritto nella tesi è stato ispirato da una necessità particolarmente sentita in quelle comunità scientifiche in cui si studia l'utilizzo della programmazione logica, e delle sue estensioni, per il ragionamento e l'interrogazione di ontologie. L'ostacolo da superare consiste nell'integrare l'editing di ontologie con lo sviluppo/la generazione di programmi logici. A tale scopo nella tesi si propone uno strumento che estende due fra i più

noti ambienti di sviluppo nei due campi, ASPIDE e *Protégé*, consentendone un agevole utilizzo congiunto.

I contributi principali di questa tesi hanno dato origine alle seguenti pubblicazioni scientifiche:

- Barbara Nardi, Kristian Reale, Francesco Ricca, Giorgio Terracina: *An Integrated Environment for Reasoning over Ontologies via Logic Programming*. Web Reasoning and Rule Systems - 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013. (LNCS – Vol. 7994 – Springer – Pg. 253-258).

- Barbara Nardi: *A Visual Syntax for Answer Set Programming*. Web Reasoning and Rule Systems - 8th International Conference, RR 2014, Athens, Greece, September 15-17, 2014. (LNCS – Vol. 8741 – Springer – Pg.249-250).

- Carmine Dodaro, Nicola Leone, Barbara Nardi, Francesco Ricca: *Allotment Problem in Travel Industry: A Solution Based on ASP*. Web Reasoning and Rule Systems - 9th International Conference, RR 2015, Berlin, Germany, August 4-5, 2015. (LNCS – Vol. 9209 – Springer – Pg. 77-92).

Abstract

Answer Set Programming (ASP) is a declarative rule–based programming paradigm for knowledge representation and declarative problem–solving. The idea of ASP is to represent a given computational problem by using a logic program, i.e., a set of logic rule, such that its answer sets correspond to solutions, and then, use an answer set solver to find such solutions.

Logic programming paradigms have received renewed interest in recent years, as demonstrated by emerging applications in many different areas of computer science, as well as industry. Due to this renewed interest an increased level of activity in the area has been registered which involved new partitioners both from academia and industry.

The development of such applications has provided important information on the real potentials of this programming paradigm, especially concerning the capability of solving complex problems in practice; moreover, application developers highlighted some critical issues to be addressed to make ASP more effective and easy to use ASP in real-world.

This thesis offers several contributions in this context can be summarized as follows:

(i) The development of two applications of ASP in a specific industrial field;

(ii) The design and implementation of new development tools for ASP.

Concerning point (i), the thesis addresses two issues considered relevant in the tourism industry. The first is known in the literature as the problem of (semi-)automatic allotment of package tours; and the second is the intelligent management of personalized newsletters for customers of travel agency. The ASP-based solutions presented in the thesis confirm that ASP is an effective tool for solving complex real-world problems.

Concerning point (ii), the thesis describes two new development tools that extend ASPIDE, a well-known integrated development environment for ASP. The first tool aims at making easier the writing of logic programs for novice programmers and is particularly suitable for those who prefer visual programming tools. In particular, the user can “ draw ” an ASP program composing graphically the logic rules. The second development tool described in the thesis answers a need arising in the scientific communities that study the usage of logic programming, and its extensions, for reasoning and querying ontologies. The goal is to integrate editing tools for ontologies with tools for the development/generation of logic

programs. To this end, the thesis proposes a tool that connects two well-known development environments in the two fields,ASPIDE and *Protégé*, in an integrated environment.

The main contributions presented in this thesis have been published in the following research papers:

- Barbara Nardi, Kristian Reale, Francesco Ricca, Giorgio Terracina: *An Integrated Environment for Reasoning over Ontologies via Logic Programming*. Web Reasoning and Rule Systems - 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013. (LNCS – Vol. 7994 – Springer – Pg. 253-258).

- Barbara Nardi: *A Visual Syntax for Answer Set Programming*. Web Reasoning and Rule Systems - 8th International Conference, RR 2014, Athens, Greece, September 15-17, 2014. (LNCS – Vol. 8741 – Springer – Pg.249-250).

- Carmine Dodaro, Nicola Leone, Barbara Nardi, Francesco Ricca: *Allotment Problem in Travel Industry: A Solution Based on ASP.* Web Reasoning and Rule Systems - 9th International Conference, RR 2015, Berlin, Germany, August 4-5, 2015. (LNCS – Vol. 9209 – Springer – Pg. 77-92).

Contents

Chapter 1

Introduction

Answer Set Programming (ASP) [38, 26, 51, 55, 48, 39, 4] is a declarative rule-based language for knowledge representation and reasoning that has been developed in the field of logic programming and nonmonotonic reasoning. The idea of ASP is to represent a given computational problem by using a logic program such that its answer sets correspond to solutions, and then, use an answer set solver to find such solutions [48]. The high knowledge-modeling power [4, 26], and the availability of efficient systems [14], make ASP suitable for implementing complex knowledge-based applications. Indeed, ASP has been applied in several fields ranging from Artificial Intelligence [3, 5] to Knowledge Management [4], and Information Integration [50], as well as industry [40].

The development of such applications has provided important information on the real potentials of this programming paradigm, especially concerning the capability of solving complex problems in practice; moreover, application developers highlighted some critical issues to be addressed to make ASP more effective and easy to use ASP in real-world [46].

This thesis offers several contributions in this context can be summarized as follows:

(i) A tool for automatic allotment of package tours;

(ii) An intelligent newsletter service for travel agencies;

(iii) A new visual editor for ASP programs;

(iv) An integrated environment for ontology reasoning and querying via logic programming.

Automatic Allotment. In the travel industry it is common for tour operators to pre-book for the next season blocks of package tours, which are called allotments in jargon [20, 23]. This practice is of help for both tour operators and service suppliers. Indeed, the first have to handle possible market demand changes, whereas the seconds are subject to the possibility that some package tours remain unsold, e.g., the rooms of a hotel can remain empty in a given season. Therefore, service suppliers and tour operators agree on sharing the economic risk of a potential low market demand by signing allotment contracts [23]. The effectiveness of this form of supplying has been studied in the economics literature under a number of assumptions on the behavior of the contractors [20, 41]. These studies, however, do not approach the problem of providing a tool that helps travel agents in the act of selecting package tours to be traded with service suppliers in the future market. Basically, given a set of requirements on the properties of packages to be brought, budget limits, and an offer of packages from several suppliers, the problem from the perspective of the travel agent is to select a set of offers to be brought (or pre-booked) for the next season so that the expected earnings are maximized [20]. Despite allotment is –de facto– one of the most commonly-used supplying practices in the tourism industry, the final selection of packages offered by travel suppliers is often done in travel agencies more or less manually.

In this thesis we approach the problem of automatic allotment of package tours, and we formalize and solve it by using ASP, and in particular

- We abstract the requirements of a real travel agent that needs to solve an allotment problem, and we solve it by using an ASP program.
- We model in ASP a number of additional preference criteria on packages to be selected that, according to a travel agent advise, allow one to further optimize the selection process by taking into account additional knowledge of the domain.
- We report on the results of a preliminary experimental analysis using real-word data that validates our approach.

Intelligent Newsletter. Newsletters delivered electronically via email (also called e-newsletters) have gained rapid acceptance in several business areas since the Internet technologies, and e-mail in particular become of common usage. The goal of a e-newsletter is to inform the subscribed customers about some news concerning specific topics they may be interested in. A newsletter can be used for commercial and marketing purposes, or as a very informative support tool for the users of web sites and portals. An effective implementation of a newsletter service, hence, should bring important news to the attention of the user and leave other news on the bottom of a message. Moreover, it is important to control the length of messages, as well as the frequency they are sent, in order to avoid the newsletter messages are considered spam to be trashed. However different users may have different preferences and different interests, so the need for personalizing messages depending on the user they are addressed is very important for increasing the effectiveness of this communication media.

In this thesis we present a solution to the problem of scheduling and organizing messages for the newsletter service of a travel agent, that was devised according to the requirements of the iTravelPlus project developed by the Tour Operator Top Class s.r.l. and the University of Calabria.

It is worth pointing out that the newsletter of iTravelPlus is not intended to be a recommender system [70], which sends customized messages via email containing commercial proposals. Instead the idea is to provide each registered customer with a number of news it can be interested in regarding several topics, such as information on the place(s) he/she is visiting, suggested destinations for the next trip, as well as administrative news, availability of travel document, meteo, safety news (e.g., alerts on whether the travel destination was affected by an earthquake or an other natural phenomenon that will be dangerous for the personal safety); up to news about local traditions, festivals, concerts and so on. The customization system should organize the news by selecting and by ordering them in the best possible and attractive way to each user. This means that also the length and frequency of emails should be controlled and customized. The desired intelligent behavior is obtained in this work by devising a proper ASP program that generates the plan of messages to send by personalizing the experience of each user while trying to maximize the effectiveness of messages.

A new Visual Editor for *ASPIDE*. In order to facilitate the design of ASP applications a rich set of tools for ASP-program development were proposed in the last few years, which are nowadays collected in rich integrated development environments, such as *ASPIDE* [32] and SeaLion [11]. Nonetheless, the task of designing a logic program consists of writing text files (more or less computer-assisted) for the majority of ASP programmers. Although the basic syntax of ASP is not particularly difficult, writing ASP programs might be uncomfortable for novices as well as for users who prefer graphic tools. To face with a similar problem in the field of databases, a number of tools and graphical user interfaces were proposed [79, 62, 60, 72] starting from the 70s for facilitating the specification of queries. Today many commercial and free relational database query tools offer fully graphical Query By Example (QBE) interfaces for facilitating the end approach of users to systems and languages. The practical relevance of graphic tools is now well-recognized: a QBE interface is, indeed, the default in the user-oriented Microsoft Access. Following this idea, *ASPIDE* was equipped with a QBE-like editor for logic rules (see Chapter 3.1). However, the *ASPIDE* visual editor resulted to be not immediate and intuitive for non-expert users. The main drawback of the *ASPIDE* QBE-like editor is that it can only represent the body of a rule as a query, and the interface does not provide a clear intuitive image of an entire rule and of the entire program.

After analyzing the limits of this QBE-like proposal, we devised a new visual interface for *ASPIDE* that supports all the powerful language constructs of ASP, and overcomes the limits of the original visual editor. The user does not have to edit text files, or know the details of a specific ASP dialect, but he/she can exploit a fully graphic environment that immediately recalls the structure of programs and rules, and allows the user to create rules by almost always clicking and dragging graphical elements on the main drawing area.

Integrated environment for ontologies and logic programs. Ontology-based reasoning is becoming more and more a relevant task [12, 16] in the area of knowledge representation and reasoning. New Semantic Web repositories are continuously built either from scratch or by translation of existing data in ontological form and are made publicly available. These repositories are often encoded by using W3C [77] standard languages like RDF(S), and OWL, and query answering on such repositories can be carried out with specific reasoners, supporting SPARQL as the query language.

In this context, the interest in approaches that resort to logic programming for implementing various reasoning tasks over ontologies is growing. Consider for instance that recent studies have identified large classes of queries over ontologies that can be Datalog-rewritable (see [42] for an overview) or First-Order Rewritable [17]. Approaches dealing with such fragments usually rely on query reformulation, where the original query posed on the ontology is rewritten into an equivalent set of rules/queries that can be evaluated directly on the ontology instances. Many query rewriters that are based on this idea exist [22, 1, 57, 71, 73] producing SQL queries or stratified Datalog programs. Moreover, even considering a setting where SPARQL queries are posed on RDF repositories, translations to ASP were proposed [59] and implemented [43].

However, if we look at this scenario from a developer point of view, one can notice that different families of tools are required. On the one hand, one needs a good environment for designing and editing ontologies. On the other hand one would like to design, execute and test ASP programs for ontology reasoning. Unluckily specific tools for these tasks are currently developed independently and miss a common perspective. We face with this issue proposing the integration of two major development environments for ASP programs and Ontology editing, respectively: *ASPIDE* [32] and *protégé* [76]. *Protégé* being one of the most diffused environments for the design and the exploitation of ontologies; and *ASPIDE* being the most comprehensive IDE for ASP. In particular, *protégé* is an open source ontology editor and knowledge-base framework, which (i) provides an environment for ontology-based prototyping and application development; (ii) supports several common formats (such as RDF(S), OWL); (iii) supports several querying and reasoning engines; and (iv) can be extended with user-defined plugins. *ASPIDE* supports the entire life-cycle of ASP programs development, and can be extended with user-defined plugins [30] to support: (i) new input formats, (ii) program rewritings, and even (iii) the customization of solver results.

In this thesis we present an extension of both editors with specific plugins that enable a synergic interaction between them. The user can, thus, handle both ontologies and ASP-based reasoning by exploiting specific tools integrated to work together. Note that, our solution has to be considered as a first step towards the development of a general platform, which can be personalized and extended (also with the help of the research community) by integrating additional rewriters/reasoners. The aim is to provide an environment for developing, running and testing ASP-based ontology reasoning tools and their applications.

Thesis Structure. This thesis is structured as follows: Chapter 2 overviews ASP syntax and semantics, and shows how ASP can be used as a tool for knowledge representation and reasoning; Chapter 3 presents the main developer tools for ASP programmers that have been either extended or employed in this thesis; Chapter 4 is devoted to the description of two applications of ASP to the tourism domain; Chapter 5 presents the two new developer tools developed in this work that extend *ASPIDE* with a new visual editor and a plug-in for interacting with *protégé*; and, eventually, Chapter 6 concludes the thesis summarizing the obtained results.

Chapter 2

Answer Set Programming

In this Chapter we overview Answer Set Programming (ASP) [38, 26, 51, 55, 48, 39, 4], a declarative rule-based programming paradigm for knowledge representation and declarative problem-solving. The idea of ASP is to represent a given computational problem by using a logic program, i.e., a set of logic rule, such that its answer sets correspond to solutions, and then, use an answer set solver to find such solutions. The high knowledge-modeling power [4, 26], and the availability of efficient systems [15] make ASP suitable for implementing complex knowledge-based applications.

2.1 Basic Language

Syntax. Let V be a set of variables, C be a set of constants, and S be a set of predicates symbols. We assume variables to be strings starting with uppercase letters and constants to be non-negative integers or strings starting with lowercase letters. Predicates (represented by strings starting with lowercase letters) have each one an associated arity (non-negative integer) representing the number of terms contained in the predicate.

A *term* is either a variable or a constant. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *literal* is a *positive literal* p or a *negative literal* $\text{not}\ p$, where p is an atom.

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n \text{ :- } b_1, \cdots, b_k, \text{not}\ b_{k+1}, \cdots, \text{not}\ b_m.$$

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

The disjunction $a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n$ is called *head* of r, while the conjunction $b_1, \cdots, b_k, \text{not}\ b_{k+1}, \cdots, \text{not}\ b_m$ is the *body* of r.

We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set of the body literals. In particular, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$.
A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*, and we usually omit the " :- " sign, while rules with empty head are called *integrity constraints* (or simply *constraint*):

$$:\!-\ b_1, \cdots, b_k, \text{not } b_{k+1}, \cdots, \text{not } b_m.$$

A *disjunctive logic program* $\mathcal{P}$ is a finite set of rules (possibly including integrity constraints), and $Rules(P)$ denotes the set of rules (including integrity constraints) in $\mathcal{P}$. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in p : B^-(r) = \emptyset$) is called *positive*, and a v-free program $\mathcal{P}$ (i.e., such that $\forall r \in p : |H(r)| \leq 1$) is called *normal logic program*.

A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. A program is safe, if each of its rules is safe, and in the following we will only consider safe programs.

A term (an atom, a rule, a program, etc.) is called *ground*, if no variable appears in it. A ground program is also called a *propositional* program.

Given a literal ℓ, let $\text{not}.l = a$ if $l = \text{not } a$, otherwise $\text{not}.l = \text{not } l$, and given a set L of literals, $\text{not}.L = \{\text{not}.l \mid l \in L\}$.

Example 2.1 *For example consider the following program:*

r_1*:* $a(X) \text{ v } b(X) :\!- c(X,Y), d(Y), \text{not } e(X).$
r_2*:* $:\!- c(X,Y), k(Y), e(X), \text{not } b(X).$
r_3*:* $m :\!- n, o, a(1).$
r_4*:* $c(1,2).$

r_1 *is a disjunctive rule s.t.* $H(r_1) = \{a(X), b(X)\}$*,* $B^+(r_1) = \{c(X,Y), d(Y)\}$*, and* $B^-(r_1) = \{e(X)\}$*;*
r_2 *is a constraint s.t.* $B^+(r_2) = \{c(X,Y), k(Y), e(X)\}$*, and* $B^-(r_2) = \{b(X)\}$*;*
r_3 *is a ground positive (non-disjunctive) rule s.t.* $H(r_3) = \{m\}$ $B^+(r_3) = \{n, o, a(1)\}$*, and* $B^-(r_3) = \emptyset$*;* r_4 *is a fact (note that* $:\!-$ *is omitted).*

The *aggregate functions*, are functions of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*.
Aggregate functions map multisets of constants to a constant. The most common functions implemented in ASP systems are the following:

- `#min`, minimal term, undefined for the empty set;
- `#max`, maximal term, undefined for the empty set;
- `#count`, number of terms;
- `#sum`, sum of integers.

An *aggregate atom* is of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq\}$ is a comparison operator, and T is a term called guard. An aggregate atom $f(S) \prec T$ is ground if T is a constant and S is a ground set.

Semantics. The semantics of a disjunctive logic program is given by its answer set [63].

Let $\mathcal{P}$ be an ASP program. The *Herbrand universe*, denoted by U_P, is the set of all constants appearing in $\mathcal{P}$ and the *Herbrand base*, denoted by B_P, of $\mathcal{P}$ is the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants in U_P (see e.g.,[4]).
Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of U_P. Similarly, given a program $\mathcal{P}$, the *ground instantiation* of $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

For every program $\mathcal{P}$, we define its answer set using its ground instantiation $\mathcal{P}$ in two steps:

- to define the answer set of positive programs;
- to give a reduction of general programs to positive ones and use this reduction to define answer set of general programs.

A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\mathrm{not}\ \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in B_P. A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I.

Let r be a ground rule in $\mathcal{P}$. The head of r is *true* w.r.t. I if exists $a \in H(r)$ s.t. a is true w.r.t. I (i.e., some atom in $H(r)$ is true w.r.t. I). The body of r is *true* w.r.t. I if $\forall \ell \in B(r)$, ℓ is true w.r.t. I (i.e. all literals on $B(r)$ are true w.r.t I). The body of r is *false* w.r.t. I if $\exists \ell \in B(r)$ s.t. ℓ is false w.r.t I (i.e., some literal in $B(r)$ is false w.r.t. I). 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.

An interpretation I is *total* if, for each atom A in B_P, either A or $\text{not}.A$ is in I (i.e., no atom in B_P is undefined w.r.t. I). A total interpretation M is a *model* for $\mathcal{P}$ if, for every $r \in \mathcal{P}$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M. X is a *answer set* for a positive program $\mathcal{P}$ if its positive part is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

Example 2.2 *Consider the positive programs:*

$$\mathcal{P}_1 = \{a \text{ v } b \text{ v } c.;\ :\!-\, a.\}$$
$$\mathcal{P}_2 = \{a \text{ v } b \text{ v } c.; b :\!-\, c.; c :\!-\, b.\}$$

The answer set of $\mathcal{P}_1$ are $\{b, \text{not } a, \text{not } c\}$ and $\{c, \text{not } a, \text{not } b\}$, while $\{b, c, \text{not } a\}$ is the only answer set of $\mathcal{P}_2$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by (i) deleting all rules $r \in \mathcal{P}$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules.

An answer set of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is an answer set of $\mathcal{P}^X$.

X is an answer set of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^X$.

We can observe that any answer set A of $\mathcal{P}$ is also a model of $\mathcal{P}$ because $Ground(\mathcal{P})^A \subseteq Ground(\mathcal{P})$, and rules in $Ground(\mathcal{P}) - Ground(\mathcal{P})^A$ are satisfied w.r.t. A.

Example 2.3 *Given the (general) program*

$$\begin{aligned}
&\mathcal{P}_3 = \{ \\
&\qquad a \text{ v } b :\!-\, c.; \\
&\qquad b :\!-\, \text{not } a, \text{not } c.; \\
&\qquad a \text{ v } c :\!-\, \text{not } b. \\
&\qquad \}
\end{aligned}$$

and the interpretation $I = \{b, \text{not } a, \text{not } c\}$, *the reduct* $\mathcal{P}_3^I$ *is* $\{a \text{ v } b :\!-\, c., b.\}$. I *is an answer set of* $\mathcal{P}_3^I$, *and for this reason it is also an answer set of* $\mathcal{P}_3$. *Now consider* $J = \{a, \text{not } b, \text{not } c\}$. *The reduct* $\mathcal{P}_3^J$ *is* $\{a \text{ v } b :\!-\, c.\ ;\ a \text{ v } c.\}$ *and it can be easily verified that* J *is an answer set of* $\mathcal{P}_3^J$, *so it is also an answer set of* $\mathcal{P}_3$.

2.2 ASP construct for optimization problems

Optimization problems are modeled in ASP using *weak constraint* [10]. A *weak constraint* ω is of the form:

$$:\sim b_1, \ldots, b_k, \mathrm{not} b_{k+1}, \ldots, \mathrm{not} b_m. \quad [w@l]$$

where w and l are the weight and level of ω. (Intuitively, $[w@l]$ is read "as weight w at level l"). An ASP program with weak constraints is $\Pi = \langle P, W \rangle$, where P is a program and W is a set of weak constraints.

Given a program with weak constraints $\Pi = \langle P, W \rangle$, the semantics of Π extends from the basic case defined above. Thus, let $G_\Pi = \langle G_P, G_W \rangle$ be the instantiation of Π; a constraint $\omega \in G_W$ is violated by an interpretation I if $B(\omega)$ is true in I. An *optimum answer set* O for Π is an answer set of G_P that minimizes the sum of the weights of the violated weak constraints in G_W (called cost) in a prioritized way (i.e., respecting levels).

Example 2.4 *Given the following program with weak constraints*

$$a \,\mathrm{v}\, b.\ c \,\mathrm{v}\, d.\ e \,\mathrm{v}\, f.$$
$$:\sim a, c. \quad [1@1]$$

an optimum answer sets is $\{\mathrm{not}\ a, b, \mathrm{not}\ c, d, e, \mathrm{not}\ f\}$*, which has cost zero.*

2.3 Knowledge Representation in ASP

Answer Set Programming is employed as a tool for knowledge representation and common sense reasoning in several application domains, ranging from classical deductive databases to artificial intelligence. ASP is particular suitable for handling incomplete knowledge and non-monotonic reasoning, and allows for encoding problems in a declarative fashion. Thanks to this approach, writing an ASP program is as easy as describing the problem domain, while the complexity of the reasoning task is hidden by using a dedicated ASP system. In addition, the (optional) separation of a fixed non-ground program from an input database allows one to obtain uniform solutions over varying instances.

ASP is a powerful formalisms, and allows complex problems to be expressed; its expressive power captures all problems belonging to the second level of the polynomial hierarchy (the complexity class Σ_2^P). This high expressive power is significantly relevant for approaching hard problems; for example, in solving planning and diagnosis problems, or, in the field of Artificial Intelligence, for solving problems not reducible to SAT instances.

ASP allows the encoding of problems in an intuitive and concise fashion following a "Guess&Check" programming methodology (originally introduced in [25] and refined in [45]). According to this approach a program $\mathcal{P}$ which encodes a problem **P** consists of the following parts:

Input Instance: An instance F of the problem **P** is specified in input using a database of facts.

Guess Part: A set of disjunctive rules $G \subseteq \mathcal{P}$, referred to as the "guessing part", is used the define the search space .

Check Part: The search space is then pruned by the "checking part", consisting of a set of constraints $C \subseteq \mathcal{P}$ which impose some properties to be verified.

Basically, the first two parts of the program, that is, the input instance and the guessing part, represent the "candidate solutions" to the problem. By adding the check part those solutions are filtered in order to guarantee that the answer sets of the resulting program represent exactly the admissible solutions for the input instance. The following example represents the typical application of the Guess&Check methodology.

Example 2.5 *Suppose that we want to partition a set of people into two groups, but we also know that some pairs of people dislike each other, thus we have to keep those two in different groups. Assume that the input instance consists of the following facts:*

$$person(bob).\ person(eve).\ dislike(bob, eve).$$

So as, applying the guess&check methodology, the guess part would model the possible assignments of persons to groups:

$$group(P, 1) \lor group(P, 2)\ \text{:-}\ person(P).$$

The resulting program (input instance + guess) produces the following answer set:

$\{person(bob), person(eve), dislike(bob, eve), group(bob, 1), group(eve, 1)\}$
$\{person(bob), person(eve), dislike(bob, eve), group(bob, 1), group(eve, 2)\}$

$\{person(bob), person(eve), dislike(bob, eve), group(bob, 2), group(eve, 1)\}$
$\{person(bob), person(eve), dislike(bob, eve), group(bob, 2), group(eve, 2)\}$

However, we want to discard assignments in which people that dislike each other belong to the same group. To this end, we add the checking part by writing the following constraint:

$$:\!-\ group(P1, G),\ group(P2, G),\ dislike(P1, P2).$$

Now, adding the constraint to the original program allows us to obtain the intended answer sets, as the checking part acted as a sort of filter:

$\{person(bob), person(eve), dislike(bob, eve), group(bob, 1), group(eve, 2)\}$
$\{person(bob), person(eve), dislike(bob, eve), group(bob, 2), group(eve, 1)\}$

In the following, we illustrate the use of ASP as a tool for knowledge representation by using some well-known examples, including classic deductive database applications, and hard problems that can be solved applying the “Guess&Check” programming style.

Reachability. Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. In different terms, the problem amounts to computing the transitive closure of the relation A.

The input graph is encoded by assuming that A is represented by the binary predicate $arc(X, Y)$, where a fact $arc(a, b)$ means that G contains an arc from a to b, i.e., $(a, b) \in A$; whereas, the set of nodes V is not explicitly represented, since the nodes appearing in the transitive closure are implicitly given by these facts.

The following program then defines a predicate $reachable(X, Y)$ containing all facts $reachable(a, b)$ such that b is reachable from a through the arcs of the input graph G:

$$r_1\colon reachable(X, Y) :\!- arc(X, Y).$$
$$r_2\colon reachable(X, Y) :\!- arc(X, U),\ reachable(U, Y).$$

The first rule states that node Y is reachable from node X if there is an arc in the graph from X to Y, whereas the second rule represents the transitive closure by stating that node Y is reachable from node X if there is a node U such that U is directly reachable from X (there is an arc from X to U) and Y is reachable from U.

Hamiltonian Path. Given a finite directed graph $G = (V, A)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once?

This is a classical NP-complete problem in graph theory. Suppose that the graph G is specified by using facts over predicates $node$ (unary) and arc (binary), and the starting node a is specified by the predicate $start$ (unary). Then, the following program $\mathcal{P}_{hp}$ solves the *Hamiltonian Path* problem:

$$
\begin{array}{l}
r_1\text{: } inPath(X,Y) \ \text{v} \ outPath(X,Y) \text{ :- } arc(X,Y).\\
r_2\text{: } reached(X) \text{ :- } start(X).\\
r_3\text{: } reached(X) \text{ :- } reached(Y), inPath(Y,X).\\
r_4\text{: } \text{:- } inPath(X,Y),\ inPath(X,Y1), Y <> Y1.\\
r_5\text{: } \text{:- } inPath(X,Y),\ inPath(X1,Y), X <> X1.\\
r_6\text{: } \text{:- } node(X),\ \text{not } reached(X),\ \text{not } start(X).
\end{array}
$$

The disjunctive rule (r_1) guesses a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary predicate $reached$ is defined, which specifies the set of nodes which are reached from the starting node. Doing this is very similar to reachability, but the transitivity is defined over the guessed predicate $inPath$ using rule r_3. Note that $reached$ is completely determined by the guess for $inPath$, no further guessing is needed. In the checking part, the first two constraints (namely, r_4 and r_5) ensure that the set of arcs S selected by $inPath$ meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S.

Traveling Salesperson. The Traveling Salesperson Problem (TSP) is a well-known optimization problem, widely studied in Operation Research.

Given a weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph, find a minimum-cost cycle (closed path) in G starting at a and passing through each node in V exactly once.

A DLP encoding for the Traveling Salesperson Problem (TSP) can be easily obtained from an encoding of Hamiltonian Cycle by adding optimization: each arc in the graph carries a weight, and a tour with minimum total weight is selected.

Suppose that the graph G is specified by predicates $node$ (unary) and arc (ternary), and that the starting node is specified by the predicate $start$ (unary).

The ASP program with weak constraints solving the TSP problem is thus as follows:

r_1: $inPath(X, Y, C)$ v $outPath(X, Y, C)$:- $edge(X, Y, C)$.
r_2: $reached(X)$:- $start(X)$.
r_3: $reached(X)$:- $reached(Y), inPath(Y, X, \_)$.
r_4: :- $inPath(X, Y, \_), inPath(X, Y1, \_), Y <> Y1$.
r_5: :- $inPath(X, Y, \_),\ inPath(X1, Y, \_),\ X <> X1$.
r_6: :- $vertex(X)$, not $reached(X)$.
r_7: :∼ $inPath(X, Y, C)$ $[C@1]$.

The last weak constraint (r_7) states the preference to avoid taking arcs with high cost in the path, and has the effect of selecting those answer sets for which the total cost of arcs selected by $inPath$ (which coincides with the length of the path) is the minimum (i.e., the path is the shortest) .

The TSP encoding provided above is an example of the "guess, check and optimize" programming pattern [45] by adding an additional "optimization part" which mainly contains weak constraints. In the example above, the optimization part contains only the weak constraint r_7.

Team Building. A project team has to be built from a set of employees according to the following specifications

(p_1) The team consists of a certain number of employees.

(p_2) At least a given number of different skills must be present in the team.

(p_3) The sum of the salaries of the employees working in the team must not exceed the given budget.

(p_4) The salary of each individual employee is within a specified limit.

(p_5) The number of women working in the team has to reach at least a given number.

Suppose that our employees are provided by a number of facts of the form $emp(EmpId, Sex, Skill, Salary)$; the size of the team, the minimum number of different skills, the budget, the maximum salary, and the minimum number of women are specified by the facts $nEmp(N)$, $nSkill(N)$, $budget(B)$, $maxSal(M)$, and $women(W)$. We then encode each property p_i above by an aggregate atom A_i , and enforce it by an integrity constraint containing $\mathrm{not} A_i$.

r_1: $in(I) \; \mathtt{v} \; out(I) \;\text{:-}\; emp(I, Sx, Sk, Sa).$
r_2: $\text{:-}\; nEmp(N), \mathrm{not}\ \#\mathtt{count}\{I : in(I)\} = N.$
r_3: $\text{:-}\; nSkill(M), \mathrm{not}\ \#\mathtt{count}\{Sk : emp(I, Sx, Sk, Sa), in(I)\} \geq M.$
r_4: $\text{:-}\; budget(B), \mathrm{not}\ \#\mathtt{sum}\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\} \leq B.$
r_5: $\text{:-}\; maxSal(M), \mathrm{not}\ \#\mathtt{max}\{Sa : emp(I, Sx, Sk, Sa), in(I)\} \leq M.$
r_6: $\text{:-}\; women(W), \mathrm{not}\ \#\mathtt{count}\{I : emp(I, f, Sk, Sa), in(I)\} \geq W.$

Intuitively, the disjunctive rule "guesses" whether an employee is included in the team or not, while the five constraints correspond one-to-one to the five requirements p_1-p_5 . Thanks to the aggregates the translation of the specifications is clear and intuitive.

Chapter 3

Development Tools for ASP

In order to facilitate the design of ASP applications, a rich set of tools for ASP-program development were proposed in the last few years, including editors [58, 74] and debuggers [9, 8, 27].

In the next sections we present two advanced development tools for ASP, namely *ASPIDE* [32] and JDLV [28]. *ASPIDE* is an extensible integrated development environment for ASP, which will be extended in Chapter 5. JDLV is a plugin for Eclipse, supporting a hybrid language that transparently enables a bilateral interaction between ASP and Java. JDLV will be used in Chapter 4 for implementing real-world applications of ASP.

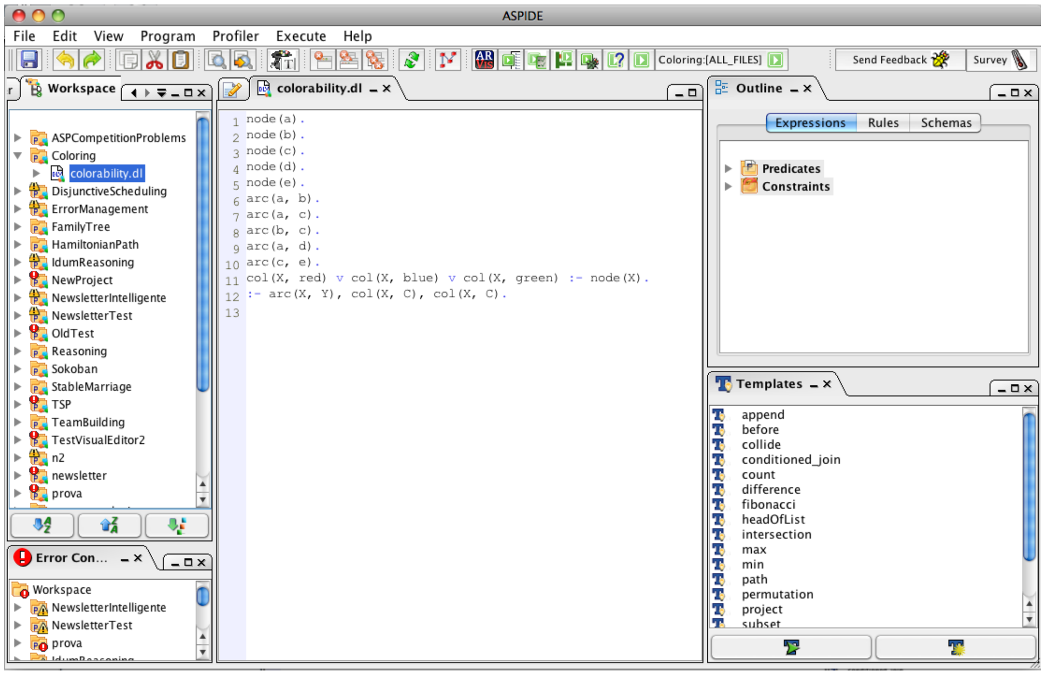
3.1 *ASPIDE*

ASPIDE [33] is a comprehensive IDE for ASP programs, which integrates an advanced editing tool with a collection of user-friendly graphical tools for program composition and execution.

ASPIDE offers a textual mode to edit the ASP programs using the DLV syntax, and support all the main language extensions (i.e. disjunction, aggregates and constraints). The editing of ASP files is semplified by an advanced text editor which provides text highlighting, auto completion, refactoring, token pair highlighter and others advanced functionalities below described.

The user interface of *ASPIDE* is depicted in Figure 3.1.

In the upper part of the interface a toolbar allows the user to quickly access some common operations. In the center of the interface there is the main editing area where it is possible to open several files organized in a tabbed panel. The left part of the interface is dedicated to the workspace explorer, which list projects,



```
node(a).
node(b).
node(c).
node(d).
node(e).
arc(a, b).
arc(a, c).
arc(b, c).
arc(a, d).
arc(c, e).
col(X, red) v col(X, blue) v col(X, green) :- node(X).
:- arc(X, Y), col(X, C), col(X, C).
```

Figure 3.1: The user interface of *ASPIDE*.

and to the error console, which organizes errors and warnings according to the project and files where they are localized. On the right, there are the outline panel and the template panel. The layout of the IDE is customizable, indeed the user can rearrange components the way he/she likes best.

In the following paragraphs we overview all the main functionalities that are available in *ASPIDE*.

Workspace organization. *ASPIDE* allows for organizing logic programs in projects à la Eclipse, which are collected in a workspace. Projects collect either different parts of an encoding or several equivalent encodings solving the same problem.

Advanced text editor. The editing of ASP files is simplified by an advanced text editor which provides *keyword outlining* (such as “ :– ” and “not”); *text highlighting* of predicate names, variables, strings, and comments; *auto completion*, predicate names are both learned while writing, and extracted from the files belonging to the same project, variables are suggested by taking into account the rule we are currently writing; *refactoring* to modify programs in a guided way;

and others advanced functionalities.

Dynamic syntactic and semantic checking. This functionality checks the program syntactical and semantical correctness during its development, highlighting the code that contains errors or warnings.

Quick fix. The editor suggests quick fixes to reported errors or warnings, and applies them (on request) by automatically changing the affected part of code.

Dynamic code template. *ASPIDE* provides support for assisted writing of rules (guessing patterns, etc.), as well as automated writing of entire subprograms (e.g., transitive closure rules) by means of code templates, which can be instantiated while writing.

Outline navigation. *ASPIDE* creates an outline view which graphically represents program elements. Each item in the outline can be used to quickly access the corresponding line of code (a very useful feature when dealing with long files).

Dependency graph. Another tool to navigate the code is the dependency graph. *ASPIDE* creates automatically a graphical representation of several variants of the (non-ground) dependency graphs associated with a project, and can be used for analyzing rule dependencies and browsing the program.

Debugger and Profiler. This functionalities allow the user to interact with ASP solver in order to understand the reason why a program does not produce the expected output.

Unit Testing. In software engineering, the task of testing and validating programs is a crucial part of the life-cycle of software development process and a test conceived for verifying the behavior of a specific part of a program is called unit testing. The testing feature consists on a unit testing framework for logic programs in the style of JUnit. The developer can specify rules by composing one or several units, specify one or more inputs and assert a number of conditions on both expected outputs and the expected behavior of sub-programs. For an exhaustive description of the testing language and of the graphical tool we refer the reader to [29].

Configuration of the execution. This feature allows to select the solver executable and the options for the program execution.

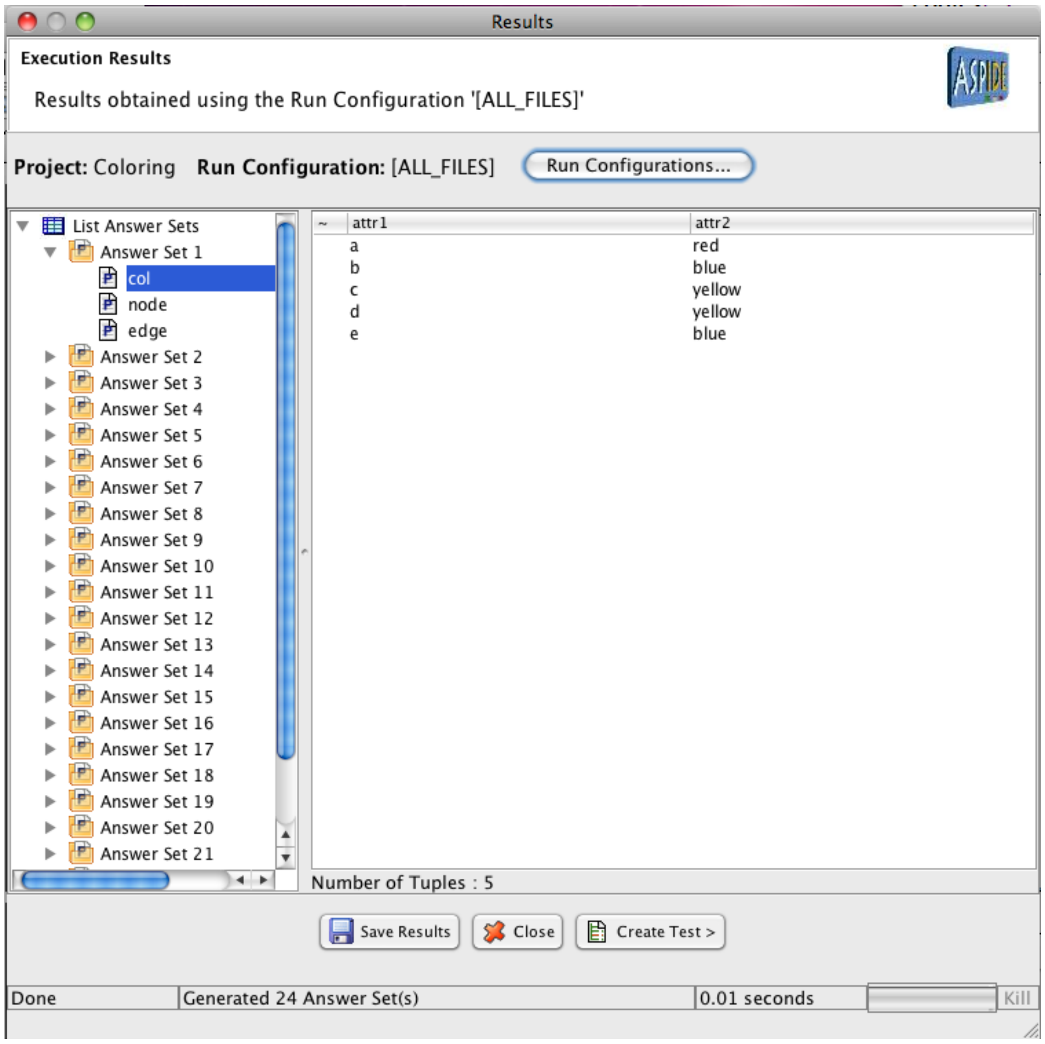
Presentation of results. The output of the program (either its answer sets, or the database table contents) can be visualized within the same environment in textual mode or using a table view of models.

Visual Editor. Using the Visual Editor, the users can draw logic programs by exploiting a full graphical environment that offers a QBE-like style for building the logic rules [31]. Note that this feature is different from the new visual programmin interface presented in next chapter, that uses a radically-different graphical language. In the remainder of this work we refer to the original visual editor of *ASPIDE* by *Visual ASP*, and we name *New Visual ASP* the new visual editor described in the next chapter.

Reverse engeenearing. The user can switch, every time he needs, from the Text Editor to the Visual Editor (and vice-versa) allowing textual (visual) refreshing during the switching phase.

Configuration of the execution. The execution of ASP programs is fully customizable by using the RunConfiguration Dialog that allows one to set the system executable, setup invocation options and input files. A number of shortcuts and drop down menus allows one for a quick execution of single files or selection of files within a project.

User-defined Plugins. An important feature of *ASPIDE* is the possibility to extend it with user defined plugins. Developers can create libraries for extending *ASPIDE* with: (i) new input formats, (ii) program rewritings, and even (iii) customizing the visualization/format of results. An input plugin can take care of input files that appear in *ASPIDE* as a logic program, and an output plugin can handle the external conversion of the computed results. A rewriting plugin may encode a procedure that can be applied to rules in the editor (e.g., disjunctive rule shifting can be applied on the fly by selecting rules in the editor and applying the mentioned rewriting). An SDK available from the *ASPIDE* web site allows one to develop new plugins.


Results

Execution Results

Results obtained using the Run Configuration '[ALL_FILES]'

Project: Coloring **Run Configuration:** [ALL_FILES]

- List Answer Sets
  - Answer Set 1
    - col
    - node
    - edge
  - Answer Set 2
  - Answer Set 3
  - Answer Set 4
  - Answer Set 5
  - Answer Set 6
  - Answer Set 7
  - Answer Set 8
  - Answer Set 9
  - Answer Set 10
  - Answer Set 11
  - Answer Set 12
  - Answer Set 13
  - Answer Set 14
  - Answer Set 15
  - Answer Set 16
  - Answer Set 17
  - Answer Set 18
  - Answer Set 19
  - Answer Set 20
  - Answer Set 21

attr1	attr2
a	red
b	blue
c	yellow
d	yellow
e	blue

Number of Tuples : 5

Done | Generated 24 Answer Set(s) | 0.01 seconds

Figure 3.2: Results

Results window. The ASP solvers, in general, show to the users the results of the execution in textual mode and sometimes those results are not so much readable by the users. A useful way to have a more user-friendly visualization is to organize them in tables, showing each predicate as a table of tuples. *ASPIDE* allows to wrap the results generated by DLV and to organize the answer-set in useful data structures. Exploiting these data structures, the results are visualized to the users using tables.

The figure 3.2 shows the result window of *ASPIDE*: the answer-sets are organized in a tree and each answer-set present the list of predicates obtained. Just by clicking on a predicate, the window will show a table of contents of that predicate as a set of tuples. Using the window, the user can choose to save the results in some file as sets of facts.

Database connectivity. *ASPIDE* simplifies access to external databases by a graphical tool connecting to DBMSs via JDBC. The database management feature of *ASPIDE* supports the creation of both *#import/#export* directives of DLV, and fully-graphical composition of TYP files [75]. Imported sources are emphasized also in the program editor by exploiting a specific color indicating the corresponding predicates. Database oriented applications can be run by setting DLV^{DB} as engine in a run configuration. A data integration scenario [47] can be implemented by exploiting these features.

3.2 $\mathcal{JASP}$: Combining Java and ASP

ASP is a convenient development framework that offers advantages from a Software Engineering viewpoint, in flexibility, readability, extensibility, ease of maintenance, etc. However, to develop real-world ASP-based application it is necessary to integrate ASP technologies (i.e., ASP programs and solvers) in the well–assessed software–development processes and platforms, which are tailored for imperative/object-oriented programming languages. Effective programming-tools were conceived to ease the usage and the integration of ASP-based technologies in the existing programming environments. The development of APIs, which offer methods for interacting with an ASP system from an embedding program, was a necessary step in accommodating he use of ASP-based solutions within large software systems. Among the first proposals we mention the DLV Wrapper [65], a library that allows to embed ASP programs and control the execution of the DLV system from a Java program, and the ONTODLV API [36], a richer API that allows to embed ontologies and reasoning modules developed using the ONTODLP language [67]. In APIs, however, the burden of the integration between ASP and Java is still in the hands of the programmer, who must take care of the (often repetitive and) time-consuming development of scaffolding code that executes the ASP system and gets data back and forth from logic-based to imperative representations.

These observations inspired the development of a hybrid language, called $\mathcal{JASP}$ [28], that transparently supports a bilateral interaction between ASP and Java. $\mathcal{JASP}$ introduces minimal syntax extensions both to Java and ASP. Its specifications are both easy to learn by programmers and easy to integrate with other existing Java technologies. The programmer can simply embed ASP code in a Java program without caring about the interaction with the underlying ASP system. In the following we describe $\mathcal{JASP}$ and present the main features of the language that were employed in this thesis, for a complete description of $\mathcal{JASP}$ we direct the reader to [28].

$\mathcal{JASP}$ **language.** In $\mathcal{JASP}$ the programmer can simply embed ASP code in a Java program without caring about the interaction with the underlying ASP system. The logical ASP program can access Java variables, and the answer sets, resulting from the execution of the ASP code, are automatically stored in Java objects, possibly populating Java collections, transparently. A key ingredient of $\mathcal{JASP}$ is the mapping between (collections of) Java objects and ASP facts. In $\mathcal{JASP}$, Java Objects are mapped to logic facts (and vice versa) by adopting a structural mapping strategy. $\mathcal{JASP}$ exploits the same ideas of modern Object-Relational Mapping (ORM) frameworks, such as Hibernate and TopLink, where objects are saved/loaded from/to relational databases. $\mathcal{JASP}$ supports both a default mapping strategy, which fits the most common programmers' requirements, and custom ORM specifications that comply with the Java Persistence API (JPA) [56] to suit enterprise application development standards.

The $\mathcal{JASP}$ code is very natural and intuitive for a programmer skilled in both ASP and Java. A monolithic block of plain ASP code (called *module*) is embedded in the Java method, which is executed "in-place", i.e., the solving process is triggered at the end of the module specification.

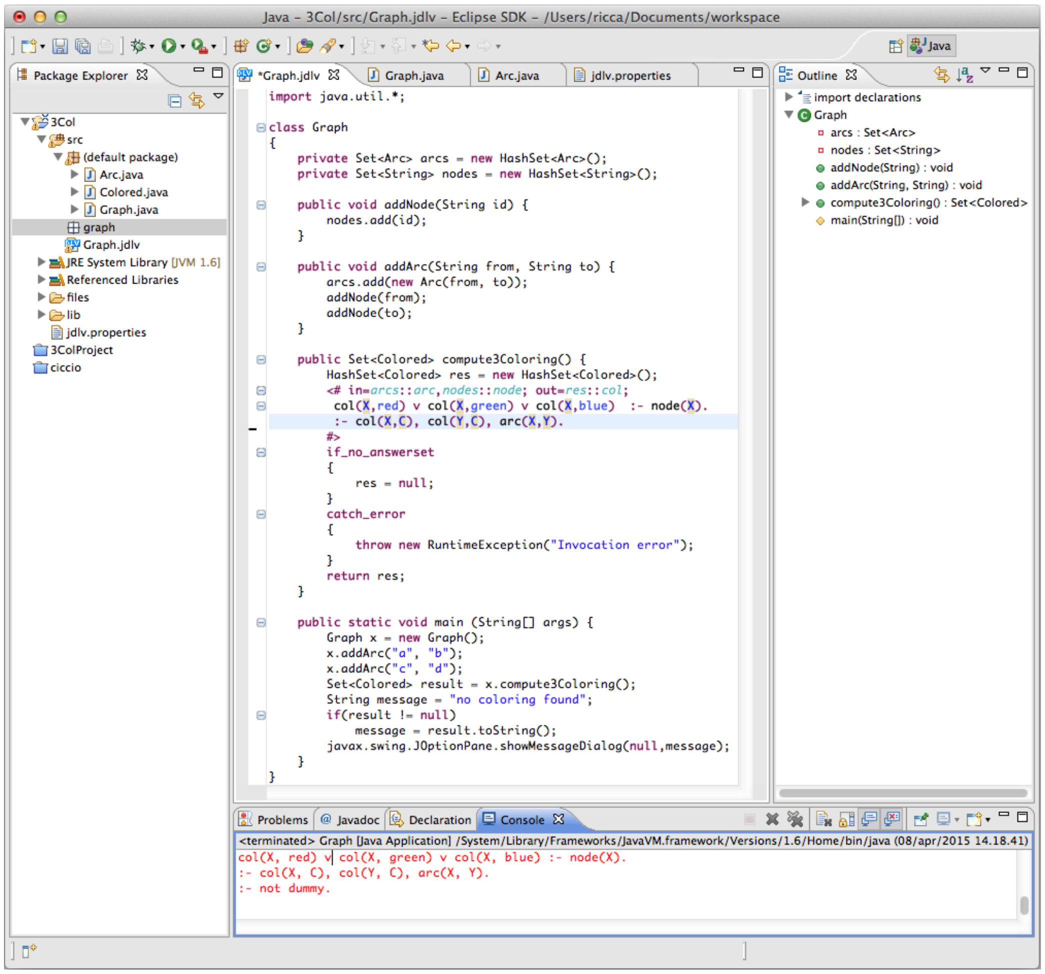
As an example consider the NP-complete problem known as *3-Colorability*. Given a graph $G = (V, A)$, *3-Colorability* amounts to assign to each node of G one of three colors (say, red, blue or green) such that adjacent nodes always have distinct colors. The input graph G is represented by facts of the form *node(v)* $\forall v \in V$, and *arc(a,b)* $\forall (a, b) \in A$.

The solutions is the ASP program made by only two rules:

$$\begin{array}{l} col(X, red) \texttt{ v } col(X, blue) \texttt{ v } col(X, green) \mathrel{:\!-} node(X). \\ \mathrel{:\!-} arc(X, Y), col(X, C), col(Y, C). \end{array}$$

The disjunctive rule can be read "if X is a node then it is either colored *red* or *blue* or *green*". The constraint can be read "discard solutions where an arc connects two nodes, namely X and Y, which have both color C".

In Figure 3.3 is depicted the simple $\mathcal{JASP}$ code solving the *3-Colorability* problem. The program in Figure 3.3 defines a *Graph* class with a method *com-*



```
import java.util.*;

class Graph
{
    private Set<Arc> arcs = new HashSet<Arc>();
    private Set<String> nodes = new HashSet<String>();

    public void addNode(String id) {
        nodes.add(id);
    }

    public void addArc(String from, String to) {
        arcs.add(new Arc(from, to));
        addNode(from);
        addNode(to);
    }

    public Set<Colored> compute3Coloring() {
        HashSet<Colored> res = new HashSet<Colored>();
        <# in=arcs::arc,nodes::node; out=res::col;
         col(X,red) v col(X,green) v col(X,blue)  :- node(X).
         :- col(X,C), col(Y,C), arc(X,Y).
        #>
        if_no_answerset
        {
            res = null;
        }
        catch_error
        {
            throw new RuntimeException("Invocation error");
        }
        return res;
    }

    public static void main (String[] args) {
        Graph x = new Graph();
        x.addArc("a", "b");
        x.addArc("c", "d");
        Set<Colored> result = x.compute3Coloring();
        String message = "no coloring found";
        if(result != null)
            message = result.toString();
        javax.swing.JOptionPane.showMessageDialog(null,message);
    }
}
```

```
<terminated> Graph [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home/bin/java (08/apr/2015 14.18.41)
col(X, red) v col(X, green) v col(X, blue) :- node(X).
:- col(X, C), col(Y, C), arc(X, Y).
:- not dummy.
```

Figure 3.3: The *JDLV* Eclipse plugin.

pute3Coloring(), which computes a 3-coloring of the given graph. The ASP code is embedded in a module statement enclosed within special tags ($< \# \ldots \# >$). When *compute3Coloring()* is invoked, Java objects are transformed into logic facts, by applying an ORM strategy as specified in the module parameters. In the example Fields (containing collections of Java objects), such as *arcs* and *nodes*, are mapped to corresponding predicates *arc* and *node*, respectively:

```
<# in = arcs::arc, nodes::node, out = res::col
```

The local variable *res* is mapped as output variable corresponding to predicate *col*. This row encoding the mapping between Java objects and ASP predicates.

The meaning of this program is the following: when *compute3Coloring()* is called, the set *nodes* is transformed in a series of unary facts of the form *node(x)*, one fact for each string x in *nodes*; similarly, each instance of *arc* stored in the variable *arcs* is transformed in a binary fact, e.g., *arc(from,to)*. These facts are input of the logic program, which is evaluated "in-place". In case no 3-coloring exists, variable *res* is set to *null*, else, when the first answer set is computed, for each fact *col* contained in the solution a new object of the class *Colored* is created and added to *res*, which, in turn, is returned by the method.

```
    if_no_answerset $res$ = null
    return $res$;
```

The $\mathcal{JASP}$'s default ORM strategy is applied to map one object per logic fact, which compound keys, i.e., keys made of all basic attributes, and *embedded values* for one to one associations, which naturally fits the usual way of representing information in ASP, e.g., in the example, one fact models one node. Such a mapping is inverted to obtain Java objects from logic facts, and ensures the safe creation of new Java objects without requiring value invention in logic programs. Although this strategy poses (very few) restrictions such as non-recursive type definition (e.g., tree-like structures are not admitted in $\mathcal{JASP}$*-core*), based on our experience, it is sufficient to handle common use cases. On the other hand, full $\mathcal{JASP}$ language allows for custom ORM strategies specified by JPA [56] annotations. It is now clear that, $\mathcal{JASP}$ directly extends the syntax of Java such that $\mathcal{JASP}$ module statements are allowed in Java *block statements*. Concerning the syntax allowed within modules, $\mathcal{JASP}$ is compliant with the language of DLV, and also supports a number of advanced features that will be described later on.

Object-Relational Mapping in $\mathcal{JASP}$. In order to give a more formal account on the way objects are translated back an forth from Java to ASP we detail the ORM mechanism that is used in $\mathcal{JASP}$. We limit our definitions to class names and fields, since other language features (e.g., modifiers, methods) do not play a role in object-relational mappings. We always assume that Java statements are *correct* w.r.t. both syntax and typing rules; and, it is given the set of admissible (Java) *identifiers*. We consider only local scope name conventions, since the definitions introduced in the following can be extended to the general case considering fully qualified names.

A class (schema) $\mathcal{K}$ is a tuple $\mathcal{K} = \langle N, \mathcal{F} \rangle$, where N is the identifier denoting the class name, and $\mathcal{F} = \langle f_1, \ldots, f_m \rangle$ ($m \geq 0$) is the tuple of *declared* fields of N. A field f_i ($0 \leq i \leq m$) is a pair $f_i = (n_i, t_i)$ where n_i is the field name and t_i is the field type. With a little abuse of notation we refer to the type of a variable/field and to the name of the corresponding class interchangeably. $\mathcal{F}^N$ denote the tuple of fields of class N, and $f_i^N = (n_i^N, t_i^N)$ denote the i-th field of $\mathcal{F}^N$. The set of basic types $\mathcal{B}$ contains *String* and all Java primitive types and corresponding boxing reference types, e.g., both *int* and *Integer* are basic types. The set of collection types $\mathcal{C}$ contains all valid capture conversions of $Collection <? >$, $Set <? >$ and $List <? >$ and the corresponding raw types, e.g., both $Set < String >$ and Set are collection types. The *actual type argument* of a collection $c \in \mathcal{C}$ is denoted by $\mathcal{A}ctual(c)$, e.g., *Actual*($Set < String >$) is *String*.

Now we illustrate the ORM strategy that is transparently applied in $\mathcal{JASP}$-*core*. Any Java class that is mapped to an ASP representation is required to have no-arguments constructor, and non-recursive type definition (e.g., tree-like structures are not admitted in $\mathcal{JASP}$-*core*); moreover, both array fields and collections fields are not allowed. Otherwise the $\mathcal{JASP}$-*core* program is not *correct* and, an implementation is required to issue an error. The $\mathcal{JASP}$-*core* ORM strategy tries to map one object per logic fact. The number of predicate arguments needed for representing an object of type N, is given by function $\mathcal{A}(\cdot)$ defined as follows:

$$\mathcal{A}(N) = \begin{cases} 1 & \text{if } N \in \mathcal{B} \\ \mathcal{A}(\mathcal{A}ctual(N)) & \text{if } N \in \mathcal{C} \\ 0 & \text{if } N \notin \mathcal{C} \cup \mathcal{B} \wedge |\mathcal{F}^N| = 0 \\ \sum_{f^N=(n^N,t^N)} \mathcal{A}(t^N) & \text{if } |\mathcal{F}^N| > 0. \end{cases}$$

Basically, to represent a field of a basic type it is necessary one argument, to represent a collection it is necessary as many arguments as are needed by representing his actual type, and to represent a field of a non-basic type it is necessary as many arguments as required for representing the basic fields of the included type.

Given a Java variable of name V and type T, and a predicate name N the schema mapping function $\mathcal{M}(N, V)$ associates to V a set of predicates containing a predicate of name N and arity $\mathcal{A}(T)$.

For instance, the class *Arc* in the previous example contains two fields of type *String*. In the Figure 3.3, the variable $arcs$ is mapped to predicate name arc; in this case, $\mathcal{M}(arc, arcs)$ is applied associating the binary predicate arc to variable $arcs$. The ASP-Core language, as in ASP, does not support a predicates with variable arguments. In the case in which the predicates employed in the rules of a $\mathcal{JASP}$-*core* module have a different arity w.r.t. the one produced by the mapping then the specification is not correct, and an implementation is expected to issue an error.

ASP facts are created from Java objects by properly filling predicate attributes of the schema defined by $\mathcal{M}(\cdot,\cdot)$.

Basic types are mapped to logic terms by exploiting the *toString()* method, if the resulting string does not match a *symbolic constant* or a *number* of ASP-Core it is surrounded by quotes. Predicate attributes are filled according to the declaration order of fields. The mapping can be inverted to obtain Java objects from ASP facts. Basically, a new Java object is created for each collection of facts matching the schema associated to an output variable mapping. In the cases in which a basic attribute is filled, in the ASP program, by a term that cannot be converted back to the expected Java type, a Java exception is thrown at runtime.

The mapping strategy defined above, in term of the naming conventions for structural ORM patterns defined in [35, 7], corresponds to mapping classes to relations with a *compound key* made of all class attributes, combined with *embedded value* for one to one associations; The choice of using a compound key (made of all basic attributes) fits the usual way of representing information in ASP, e.g., in the example, one fact models one node. Moreover, it ensures the safe creation of new Java objects without requiring value invention in ASP programs [13]. This strategy has the side effect of discarding both duplicates in a *Collection* and the object position in a *List*, since ASP has the "set semantics". Specific ORM strategies are employed by commercial tools to handle such a scenario in relational databases (see [7]). This strategy is sufficient to handle common use cases; nonetheless custom ORM strategies can be specified with JPA annotations in the full language.

JPA Mappings. $\mathcal{JASP}$ spouses the work done in the field ORM [35, 7], and complies with enterprise application development standards for customizing the $\mathcal{JASP}$*-core* mapping strategy. $\mathcal{JASP}$ supports JPA [56] standard Java annotations for defining how Java classes map to relations (logic predicates). Note that, although ORM frameworks address different behavioral problems w.r.t. $\mathcal{JASP}$ (e.g, object persistence, transaction control, etc.), they are based on a mechanisms to describe/map Java classes into relational data. The full description of JPA's mapping features is out of the scope of this thesis (see [7, 56] for a full account).

An important issue to be considered in JPA mappings is the usage of *surrogate keys* that are *generated values*. Persistence frameworks generate new identifiers according to custom algorithms when persistent objects are saved, whereas $\mathcal{JASP}$ might require to create new ids also when answer sets are transformed in objects. In $\mathcal{JASP}$, it is up to the programmer ensuring that objects have a valid id before being transformed into facts, whereas, for the other direction, there are two possible strategies: (i) embed an ASP dialect supporting value invention [13], so that new ids can be created in the ASP part, e.g., exploiting an *id* function symbol holding all basic attributes. In this case, the programmer has to be aware of termination problems; (ii) give the programmer the possibility of not specifying a value for the id field by exploiting either non-positional notation or a placeholder term "*generated value*", so that the buildFacts() procedure becomes in charge of creating new ids. The id generation function can be also shared with the actual persistence framework. The latter is the approach currently supported by the implementation. It is also require that the generated id fields cannot be joined in rule bodies (in this case, the system issues a warning), which is a compromise for overcoming the fact that ASP traditionally is a function-free language. Note that, this issue does not occur in the $\mathcal{JASP}$*-core* mapping strategy, since the key is the natural one containing all predicate attributes.

Named Non-positional Notation. $\mathcal{JASP}$ introduces an alternative compact notation for logic atoms modeling Java objects borrowed from [69], that can be implemented by rewriting in plain ASP. For instance, considering a class *Person* which has seven fields, but we want to select the names of those how are taller than 2 meters, we write the rule

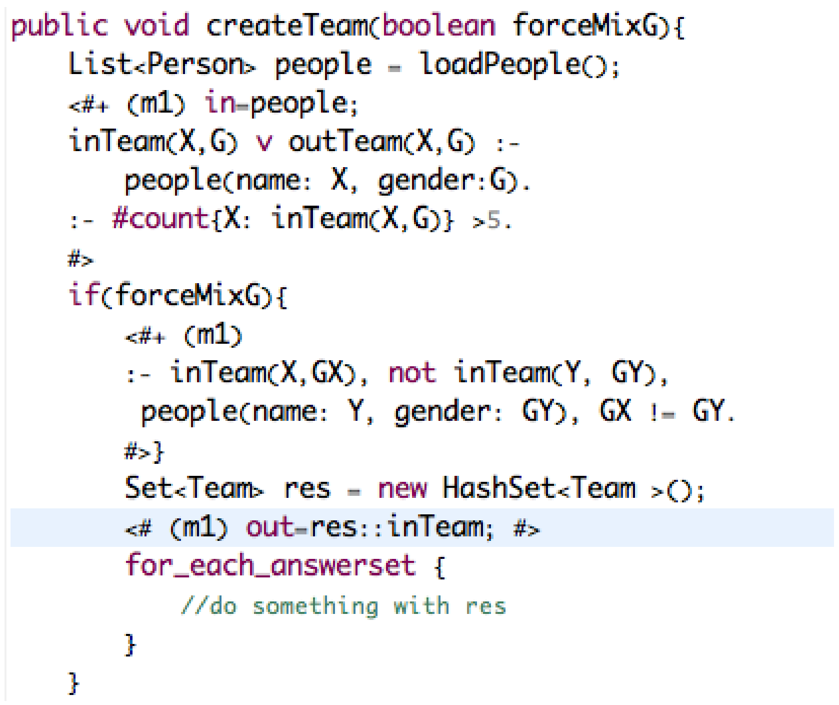
$$veryTall(X) :- person(name : X, height : H), H > 2.$$

instead of

$$veryTall(X) :- person(X, \_, \_, \_, \_, H, \_), H > 2.$$

This notation improves readability, and is less error-prone.

Dynamic Composition of $\mathcal{JASP}$ Modules. $\mathcal{JASP}$*-core* modules are monolithic blocks of ASP rules forming a program, that are executed "in-place". To give more flexibility, it is introduced *module increments*, that enable building ASP programs incrementally throughout the application. Syntactically, module increments start by "$< +$", and, semantically, correspond to accumulating additional rules and facts to the (possibly new) module at hand, without triggering the solving process. Since modules are interpreted as Java variables/fields, the usual Java scope rules apply to module increments.



```
public void createTeam(boolean forceMixG){
    List<Person> people = loadPeople();
    <#+ (m1) in=people;
    inTeam(X,G) v outTeam(X,G) :-
        people(name: X, gender:G).
    :- #count{X: inTeam(X,G)} >5.
    #>
    if(forceMixG){
        <#+ (m1)
        :- inTeam(X,GX), not inTeam(Y, GY),
         people(name: Y, gender: GY), GX != GY.
        #>}
        Set<Team> res = new HashSet<Team >();
        <# (m1) out=res::inTeam; #>
        for_each_answerset {
            //do something with res
        }
    }
```

Figure 3.4: Dynamic Module Composition and Invocation.

As an example consider the code in Figure 3.4. Module "m1", which generates all teams of at most five people, is defined incrementally. In particular a boolean the flag *forceMixG* is checked that indicates whether teams composed only of people of same gender are allowed, an additional constraint that is added to "m1" only in this case.

Accessing the Host Environment. $\mathcal{JASP}$ allows the programmer to include arbitrary Java expressions in logic rules that are evaluated at runtime. Syntactically $\mathcal{JASP}$ uses the operator ${javaExpr} that is expanded in the string obtained evaluating the Java expression $<<\ ''''+javaExpr >>$ corresponding to a call to the method $toString()$. For instance,

```
for (int i = 0; i<10; i++)
  <#+ (dyn)
     a(${i},${i+1}). #>
```
dynamically adds ten facts to module "*dyn*" (i.e., $a(1,2).a(2,3),...a(10,11)$).

The JDLV plugin. $\mathcal{JASP}$ is implemented in a prototype development system, called JDLV as an Eclipse plugin. JDLV includes *Jdlvc*, a compiler to generate

Annotation	Summary
@Entity	Indicates a class with mapping. Class name is the predicated name.
@Table (name="pred-name")	In conjunction with @Entity, to rename the default predicate name.
@Column	Identifies a class member, to be included in the mapping.
@Id	Marks a class member as identifier (key) of the relative table.
@OneToMany @ManyToOne @ManyToMany @OneToOne	On class members to denote associations multiplicity
@JoinTable (name="pred-name")	In conjunction with @OneToMany or @ManyToOne to specify a mapping realized through an associative predicate

Table 3.1: Main JPA Annotations supported by JDLV.

plain Java classes from $\mathcal{JASP}$ files. The *Jdlvc* compiler produces plain Java classes which manage the generation of logic programs and control statements for the underlying solver DLV.

The $\mathcal{J}$DLV plugin extends Eclipse with the possibility of editing files in the $\mathcal{JASP}$ syntax in a friendly environment featuring:

(i) *automatic completion* for assisted editing logic program rules;

(ii) *dynamic code checking and errors highlighting* (producing descriptive error messages and warnings);

(iii) *outline view*, a visual representation and indexing $\mathcal{JASP}$ statements, and

(iv) *automatic generation of Java code*, by means of our $\mathcal{J}$DLV compiler.

Given $\mathcal{JASP}$ files as input, the *Jdlvc* compiler produces plain Java classes which manage the generation of logic programs and control statements for the underlying ASP solver. *Jdlvc* is written in Java, and uses Java Reflection to analyze mappings (compile-time) and actual object types (runtime). An enhanced version of the DLV Java Wrapper library [66] is used to implement the solving process trough a call to the DLV system [45]. Figure 3.1 summarizes the most common JPA annotations supported by JDLV. $\mathcal{J}$DLV offers a seamless integration of ASP-based technologies within the most popular development environment for Java.

Chapter 4

Applications of ASP: Tools for Travel Agencies

This chapter is devoted to the description of two applications of ASP to the tourism domain. The applications model and solve real-world problems arising in the applied research project iTravelPlus involving the Tour Operator Top Class s.r.l. and the University of Calabria. In particular, in Section 4.1 we formalize an allotment problem that abstracts the requirements of a real travel agent, and we solve it using Answer Set Programming; and, in Section 4.2, we formalize the problem of managing a personalized newsletter considering the preference of the user to inform a potential traveler about news, events, and other information regarding destinations of interest for the user.

4.1 Automatic Allotment of Package Tours

In the travel industry it is common for tour operators to pre-book for the next season blocks of package tours, which are called allotments in jargon [23, 20]. This practice is of help for both tour operators and service suppliers. Indeed, the first have to handle possible market demand changes, whereas the seconds are subject to the possibility that some package tours remain unsold, e.g., the rooms of a hotel can remain empty in a given season. Therefore, service suppliers and tour operators agree on sharing the economic risk of a potential low market demand by signing allotment contracts [23]. Basically, given a set of requirements on the properties of packages to be brought, budget limits, and an offer of packages from several suppliers, the problem from the perspective of the travel agent is to select a set of offers to be brought (or pre-booked) for the next season so that the expected earnings are maximized [20].

In this Section we approach the problem of automatic allotment of package tours, and we formalize and solve it by using ASP. We first abstract the requirements of a real travel agent that needs to solve an allotment problem, and we solve it by using an ASP program. Then we model in ASP a number of additional preference criteria on packages to be selected that, according to a travel agent advise, allow one to further optimize the selection process by taking into account additional knowledge of the domain. Moreover we report on the results of a preliminary experimental analysis using real-word data that validates our approach. A Java library implementing our automatic allotment tool suitable to be integrated as a WEB service of the iTravel+ system is also presented.

4.1.1 Travel Agent Requirements

In this section we informally describe the requirements of a common problem in the tourism industry, i.e. the problem of booking in advance blocks of package tours for the next season. A travel agency usually selects a block of package tours from several travel suppliers, which may apply several discounts if predetermined amounts of their package tours are bought. In general, a critical requirement is that *the sum of prices of selected package tours must not exceed a limited budget*. This means that travel agencies are not allowed to buy all the package tours they need. Thus, their goal is to *select package tours in order to maximize the expected earnings*. Moreover, depending on the specific needs, travel agencies might specify other preferences among the selected package tours. Those preferences are not general. On the contrary, two different travel agencies usually have different priorities among selected packages according to their experience and customer base. In the following we detail several preferences that travel agencies might specify according to their needs.

Preference of suppliers according to destination. Travel agencies might specify a preference of suppliers for package tours involving particular destinations. For instance, a supplier can be considered highly reliable for travels in Europe and unreliable for travels in other countries.

Preference of suppliers according to the type of holidays. A supplier can be considered also preferable for particular types of holidays. For instance, some suppliers are specialized in holidays involving cruises, while others are specialized in holidays involving sports activities.

Preference on package tours with the highest rating. After a trip, travelers usually evaluate their holidays by assigning a numerical score. A package tour is evaluated by looking at the rating assigned by the travelers. Thus, travel agencies give priority to the package tours with the highest ratings.

Preferences on the number of package tours to buy. According to their typology of customers travel agencies also express a preference on the number of package tours to buy. In particular, in case travel agencies obtain the same expected earnings from two or more package tours then they can maximize or minimize the number of bought package tours according to their customer base. For instance, travel agencies working with wealthy customers may prefer to buy few package tours with highest earnings, while travel agencies working with many customers may prefer to maximize the number of package tours to buy.

Preference on the amount of money to pay. Another important preference concerns the amount of money to pay. In particular, in case travel agencies obtain the same expected earnings from two or more package tours it is preferable to select package tours with the lowest prices.

4.1.2 Basic Allotment Solution via ASP

This section illustrates the ASP program which solves the allotment problem specified in the previous section. First, the input data is described, then, the ASP rules solving the allotment problem are presented. Finally, preferences that can be specified by travel agencies are described.

Data Model. The input of the process is specified by means of the predicates described in this section. The predicates representing the facts of our encoding are the following:

- Instances of the predicate *availablePackage(pkId, supplier, destination, type, sellingPrice, purchasePrice, rating, availableQuantity)* represent stocks of available package tours in the market, where *pkId* is the identifier of the tour package, *supplier* is the identifier of the supplier selling the package tour, *destination* is the destination of the package tour, *type* is the type of holiday, *sellingPrice* is the price applied by the travel agency to their customers, *purchasePrice* is the price applied by the supplier to the travel agency, *rating* is a numerical score associated to the package tour representing the appreciation of customers for this package tour, and *availableQuantity* corresponds to the quantity of available package tours of this kind in the market.

- Instances of the predicate *requiredPackage(destination, type, minPrice, maxPrice, requiredQuantity)* represent package tours required by a travel agency, where *destination* is destination of the package tour required, *type* is the type of holiday required, *minPrice* and *maxPrice* represents the range of prices the travel agency is willing to pay for a given destination and type of holiday, and *requiredQuantity* corresponds to the quantity of required package tours of this kind.

- Instances of the predicate *discount(supplier, quantity, percentageDiscount)* represent the discount applied by suppliers if a given amount of their package tours is bought, where *supplier* represents the identifier of the supplier, *quantity* is the minimum quantity of bought package tours for applying the discount, and *percentageDiscount* is the percentage discount applied.

- The only instance of the predicate *budget(b)* represents the maximum amount of money the travel agency is willing to pay.

- Instances of the predicate *evalSupplierDestination(supplier, destination, score)* represent the evaluations of suppliers according to destination, where *supplier* is the identifier of the supplier, *destination* is the destination of the package tour, and *score* is a numerical score representing the reliability of the supplier for package tours involving the destination.

- Instances of the predicate *evalSupplierType(supplier, type, score)* represent the evaluations of suppliers according to type of holiday, where *supplier* is the identifier of the supplier, *type* is the type of holiday, and *score* is a numerical score representing the reliability of the supplier for package tours concerning the type of holiday.

Encoding The Allotment Problem. We now describe the ASP rules used for solving the allotment problem. We follow the *Guess&Check&Optimize* programming methodology [25, 45]. In particular, the following disjunctive rule guesses a quantity to buy for each required package:

$$\begin{aligned} buy(P,Q) \texttt{ v } nBuy(P,Q) \text{ :- } & availablePackages(P,\_,D,T,SP,PP,\_,AvQ),\\ & requiredPackages(D,T,MinP,MaxP,ReqQ),\\ & 0 \leq Q \leq ReqQ,\\ & Q \leq AvQ,\\ & MinP \leq SP \leq MaxP. \end{aligned} \tag{4.1}$$

The guess of the quantity is limited to available package tours which are requested and their selling price is in the requested range. Then, assignments buying different quantities of the same package tour are filtered out by the following constraint:

$$\text{:- } \#count\{Q,P : buy(P,Q)\} > 1, availablePackages(P,\_,\_,\_,\_,\_,\_,\_). \tag{4.2}$$

Suppliers may apply one or more discounts if predetermined amounts of their package tours are bought. In general several discounts are offered depending on the volume of booked packages. In this case the maximum applicable discount among them must be applied. All applicable discounts and the maximum discount among them are computed by the following rules:

$$\begin{aligned} allDiscounts(S,D) \text{ :- } & discount(S,Q1,D),\\ & \#sum\{Q,P : buy(P,Q)\} \geq Q1.\\ maxDiscount(S,Disc) \text{ :- } & discount(S,\_,\_),\\ & \#max\{D : allDiscounts(S,D)\} = Disc. \end{aligned} \tag{4.3}$$

The predicate *allDiscounts(supplier, discount)* stores the association between the supplier and all the applicable discounts, while *maxDiscount(supplier, discount)* stores the association between the supplier and the corresponding maximum applicable discount. Then, the prices of the package tours are updated according to the above-computed discounts. This behavior is achieved by employing the following rule:

$$\begin{aligned} discountPrices(P,SP,PPD) \text{ :- } & availablePackages(P,S,\_,\_,SP,PP,\_,\_),\\ & maxDiscount(S,MD),\\ & PPD = PP - (PP * MD)/100. \end{aligned} \tag{4.4}$$

The predicate *discountPrices* stores the original selling price and the purchase price after the application of the discount for each package tour. This predicate is then used to handle a critical requirement on the budget, i.e. the sum of prices of selected package tours must not exceed a limited budget. This is expressed in ASP by the following rule:

$$\begin{aligned}&{:\!-}\ \#sum\{PP * Q, P : buy(P, Q), discountPrices(P, \_, PP)\} > B,\\ &budget(B).\end{aligned} \tag{4.5}$$

Finally, the last requirement is to maximize the earnings. This is obtained in our encoding by means of the following weak constraint:

$$:\sim discountPrices(P, SP, PP), buy(P, Q), E = (SP - PP) * Q.[-E@\ell] \tag{4.6}$$

Intuitively, when a stock of package tours is bought the solution is associated with a cost depending on the earnings obtained by buying those packages. The weight of weak constraint is negative since weak constraints expresses the minimization of the cost associated to a solution.[1]

4.1.3 Additional Preferences on Allotment

In this section, we describe preferences travel agencies might specify among the selected package tours depending on their specific needs. Different travel agencies usually have different priorities among selected package tours which are expressed in our framework by means of weak constraints. In the weak constraints we use numerical values $\ell_1, \ldots, \ell_5$ representing the levels of weak constraints. Then, an order on the preferences can be specified by properly assigning a value to those levels. The only requirement is that the level ℓ of the constraint that maximizes earnings (4.6) is greater than all the other weak constraints that are specified in the following.

Preference of suppliers according to destination. A travel agency might specify a preference of suppliers according to the destination of a travel. The following weak constraint expresses this preference:

$$\begin{aligned}&:\sim evalSupplierDestination(S, D, SC), availablePackages(P, S, D, \_, \_, \_, \_, \_),\\ &\quad nBuy(P, Q).[SC * Q@\ell_1]\end{aligned}$$

[1]ASP solvers may have undefined behaviors in presence of negative weights. A work-around is to augment the weight of the weak constraint by the maximum possible earnings.

(4.7)

Intuitively, when a stock of package tours is not bought a numerical penalty is associated to the solution. For each package tour which is not selected the cost of the solution is increased by the score associated to the corresponding supplier for the destination.

Preference of suppliers according to the type of holidays. Similarly, the following weak constraint expresses a preference among suppliers according to the type of holidays:

$$\begin{aligned}&:\sim evalSupplierType(S, T, SC), availablePackages(P, S, \_, T, \_, \_, \_, \_),\\ &\quad nBuy(P, Q).[SC * Q@\ell_2]\end{aligned} \tag{4.8}$$

For each package that is not selected the solution cost is increased by the score associated to the corresponding supplier for the type of holiday. The effect is to maximize the number of package tours in the solution that are provided by preferred suppliers.

Preference on package tours with the highest rating. Travel agencies give priority to the package tours with the highest ratings. This preference is expressed by the following weak constraint:

$$:\sim availablePackages(P, \_, \_, \_, \_, \_, R, \_), nBuy(P, Q).[R * Q@\ell_3] \tag{4.9}$$

Here, the cost of the solution is given by the sum of ratings of package tours which are not bought. Thus we maximize the ratings of selected package tours.

Preferences on the number of package tours to buy. In case a travel agency is willing to minimize the number of packages to buy we apply the following weak constraint:

$$:\sim buy(P, Q).[Q@\ell_4] \tag{4.10}$$

The cost of the solution is increased by the quantity of package tours which are not bought. Otherwise, if a travel agency is willing to maximize the number of packages to buy we apply the following weak constraint:

$$:\sim nBuy(P, Q).[Q@\ell_4] \tag{4.11}$$

Here, the cost of the solution is increased by the quantity of package tours which are bought. Note that weak constraints (4.10) and (4.11) are never applied together since travel agencies either maximize or minimize the number of package tours to buy.

Preference on the amount of money to pay. Finally, travel agencies may also want to minimize the amount of money to pay. Note that this is different from the earnings, since in this case travel agency minimizes the purchase prices without considering their selling prices. This behavior is employed by the following weak constraint:

$$:\sim discountPrices(P, \_, PP), buy(P, Q).[PP * Q@\ell_5] \tag{4.12}$$

Intuitively, the cost of the solution depends on sum of prices of package tours which are bought. Hence, this has the effect to minimize the price of package tours in the solution.

Specification of preferences. As stated in Section 4.1.1, the preferences depends on the specific needs of travel agencies, and can be applied selectively by simply adding or ignoring some of the weak constraints described in Section 4.1.3. Moreover, a travel agent must also specify a layering of preferences by properly assigning values to $\ell_1, \ldots, \ell_5$. As an example, consider a travel agent that wants to give highest priorities on package tours with the highest ratings; and then maximize the number of packages to buy. In the encoding those preferences are specified by considering weak constraints (4.9) and (4.11) and by assigning integer values to the levels such that $\ell_3 > \ell_5$, e.g., $\ell_3 = 2$, and $\ell_5 = 1$.

4.1.4 Empirical Validation

We validated our ASP-based solution running a preliminary experiment on real-world data provided by the partners of the iTravelPlus project. In particular, we obtained an instance of a database of package tours querying the database of the iTravel+ system and properly encoding it by means of ASP facts. Moreover, we generated a specification of the requested package tours by running a mining services of the same system that generates a prediction based on the package tours sold in the past. Finally, we randomly generated a number of additional requirements to test the effects of the optional preferences of our solution. The system

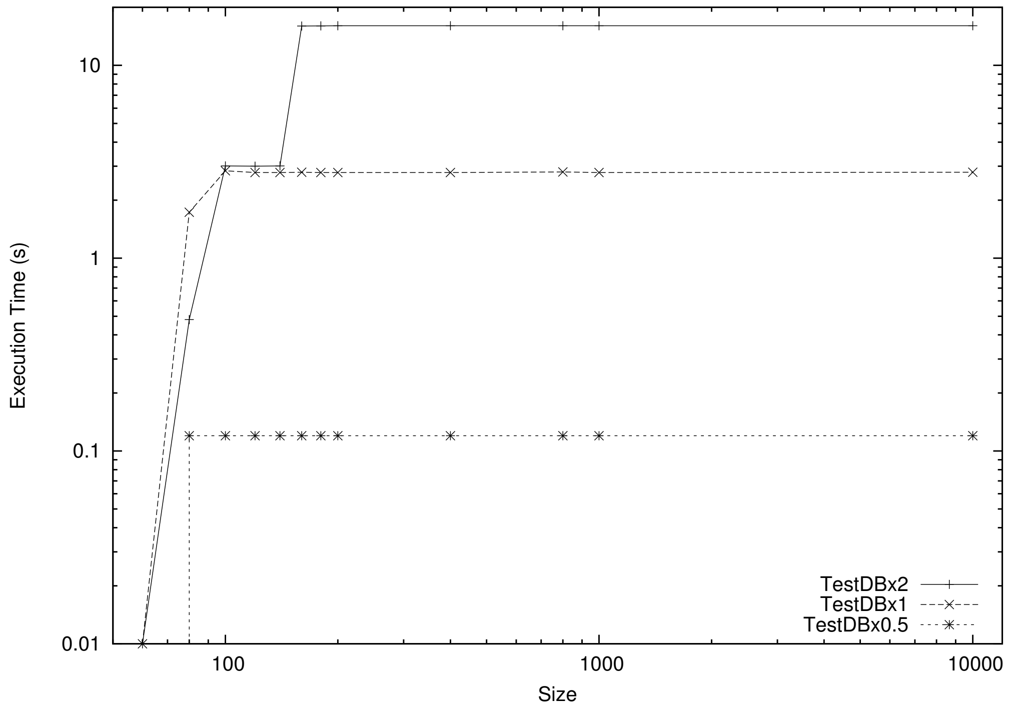


Figure 4.1: Scalability w.r.t. the number of available package tours.

was run on a four core Intel Xeon CPU X3430 2.4 GHz, with 16 GB of physical RAM, each execution was limited to 600 seconds.

The performance of the system for different sizes of the available packages is reported in Table 4.1. In particular, the first column reports the sizes of the considered DBs. We considered the original database (labeled DBx1) and then we consider two more settings containing the first half of the same database (labeled DBx0.5), a generated instance (labeled DBx2) having twice of the facts from DBx1 obtained adding more suppliers. The second column reports the minimum and the maximum available package tours among the instances considered. The number of considered instances is reported in the third column together with the

Table 4.1: Performance of the system for different available package tours.

	Available Pkgs (Min-Max)	**#inst**	**#solved**	**#optima**	**Time (no pref.)**	**Time (all pref.)**
DBx0.5	216-291	30	30	30	0.6	0.8
DBx1	445-584	30	30	26	48.5	147.8
DBx2	963-1093	30	30	14	11.7	33.41

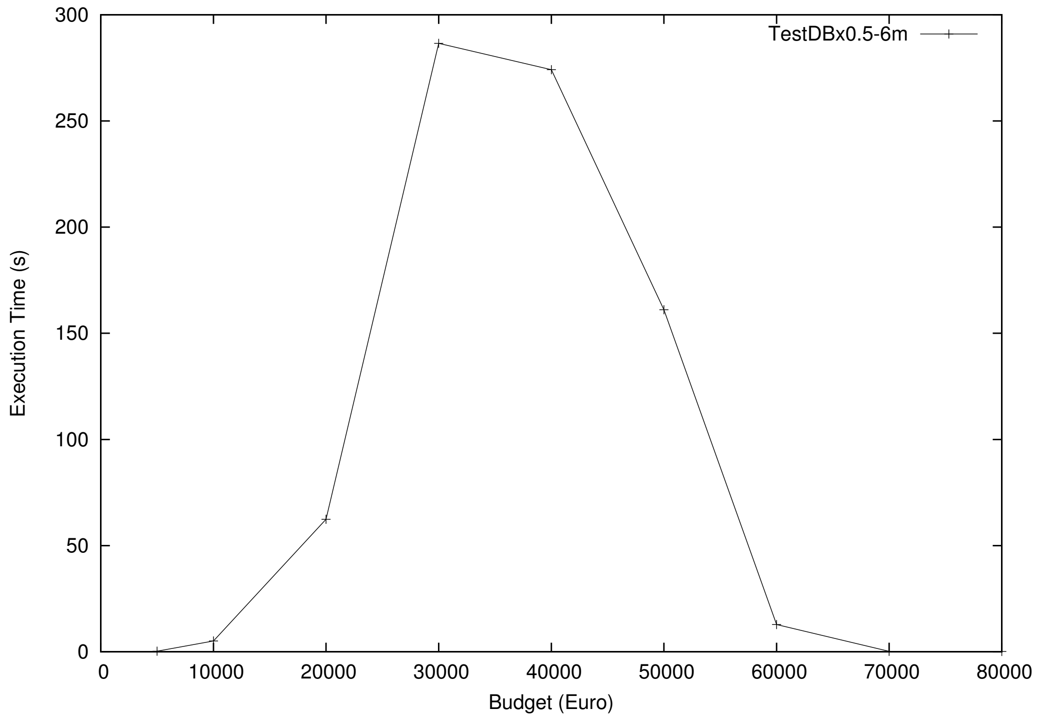


Figure 4.2: Performance of the system depending on the budget.

number of instances in which the system found a (sub-optimal) solution (fourth column) and the number of instances in which the system found the optimum solution (fifth column). The sixth column reports the sum of execution times (in seconds) elapsed for finding the optimum solution of the allotment problem without optional preferences. The seventh column reports the sum of time in seconds of finding the optimum solution of the allotment problem where all preferences are enabled. As first observation we note that the system provides a solution for all the instances considered within 10 minutes. The provided solutions are usually either close to the optimum ones or are optimal but the system is not able to proof their optimality within the timeout. Moreover, when we consider half size of the

Table 4.2: Performance of the system for different periods (in months).

Period	**Required Pkgs (Min-Max)**	**#inst**	**#solved**	**#optima**	**Search Space (avg)**
2m	79-94	30	30	30	2^{27}
4m	174-182	30	30	24	2^{54}
6m	345-365	30	30	16	2^{110}

DB size the system finds always the optimum solution, while for the original size of the DB this is the case in the 87% of the instances, which is a fairly acceptable performance for an off-line process. The performance is still good in the case we double the size of the original DB, since the system is able to find the optimum for about half of the instances. In addition, we also observe that adding the preference does not reduce either the number of solved instances nor the number of instances in which the optimum solution is found. As expected, we observe a constant slow down in the solving time, which is approximately three times higher than the one measured with no preferences.

It is worth pointing out that the performance of the system does not heavily depend on the quantity of available packages. In fact, rule (1) in Section 4.1.2 filters out all the package tours which are not required. In order to confirm this observation, we increased the available quantities for each package tours in the database by several factors. The result is reported in Figure 4.1, in which, for a particular size of the DB, a point (x, y) represents the solving time y where the availability of package tours is x percent of the original size. The graph shows that the execution times grow until the offered packages are 120% more than in the original DB, and then performance has a constant trend that is not dependent on the quantity of available packages.

Table 4.2 reports on the performance of the system for different periods of request packages. In particular, the first column reports the considered period expressed in months. The second column reports the minimum and the maximum required package tours among the instances considered. The number of considered instances is reported in the third column together with the number of instances in which the system found a solution (fourth column) and the number of instances in which the system found the optimum solution (fifth column). The last column reports the average search space for the considered instances. Also in this case, the system provides a solution for all the instances considered. Moreover, when we consider a period of 2 months the system also finds always the optimum solution, while if we considered a period of 4 months this is the case in the 80% of the instances. It is worth pointing out that, travel agencies usually book package tours for one season, thus they consider a period of at most 3-4 months. Nonetheless, the performance is still good in the case we consider a period of 6 months, since the system is able to find the optimum for about half of the instances.

Finally, another observation concerns the budget allowed by the travel agency, since the hardness of the instances depends on this parameter. In fact, is it easy to see that instances with very low (resp. high) budgets w.r.t. to the one needed to fulfill the request are likely easy, since they correspond to over-constrained (resp. under-constrained) problems where the solution is to buy no package tours (resp. to buy all required package tours). Thus, we also analyzed the behavior of the system in case we consider different budgets. The result is reported for DBx0.5 and a request of 6 months in Figure 4.2, in which a point (x, y) represents the solving time y if the budget is limited to x euro. The trend of the system confirms our expectations, since the instance is trivially solved when the budget is enough either to buy nothing or to buy everything. The maximum hardness is reached when the allotted budget can cover about 40% of the request in our experiment, a setting that in real-world instances is not that common, since the budget is usually enough to cover most of the requests.

4.1.5 Implementation in $\mathcal{JASP}$

Despite the specification of the allotment problem provided in the previous sections can be executed using an ASP system, it is not ready to be integrated in the iTravel+ system. In this section we describe the development of a Java API that provides all the interfaces that are needed to integrate our ASP program in a service of the iTravel+ system. The first step that was accomplished is to provide a specification in Java of the data model presented in Section 4.1.2, resulting in the class diagram is depicted in the Figure 4.3. In particular we have the classes: *Budget, AvailablePackage, RequiredPackage, Discount, EvalSupplierDestination, EvalSupplierType, Buy*. Note that the Java Classes have the same name of the logic predicates so that $\mathcal{JASP}$ makes automatic the conversion of Java objects representing the domain model and the corresponding predicates of our data model. More in detail, *Budget* has an attribute to represent the available budget to buy stocks. *AvailablePackage* has the attribute to identify the package, an attribute to identify the supplier, an attribute to identify the destination, an attribute to identify the type of travel, a numeric attribute to represent the selling price, i.e., the price to the client, a numeric attribute to represent the purchase price, i.e., the price to the travel agency, a numeric attribute to express the rating and a numeric attribute to express the package availability. *RequiredPackage* has an attribute to identify the destination, an attribute to identify the type of package, two attribute to the max and min price that the clients would pay to the package and a required quantity, i.e., an evaluation of the package quantity of a type that an agency would buy. *Discount* has an attributes to represent the supplier, and some values starting by the supplier apply some discount to the travel acency.

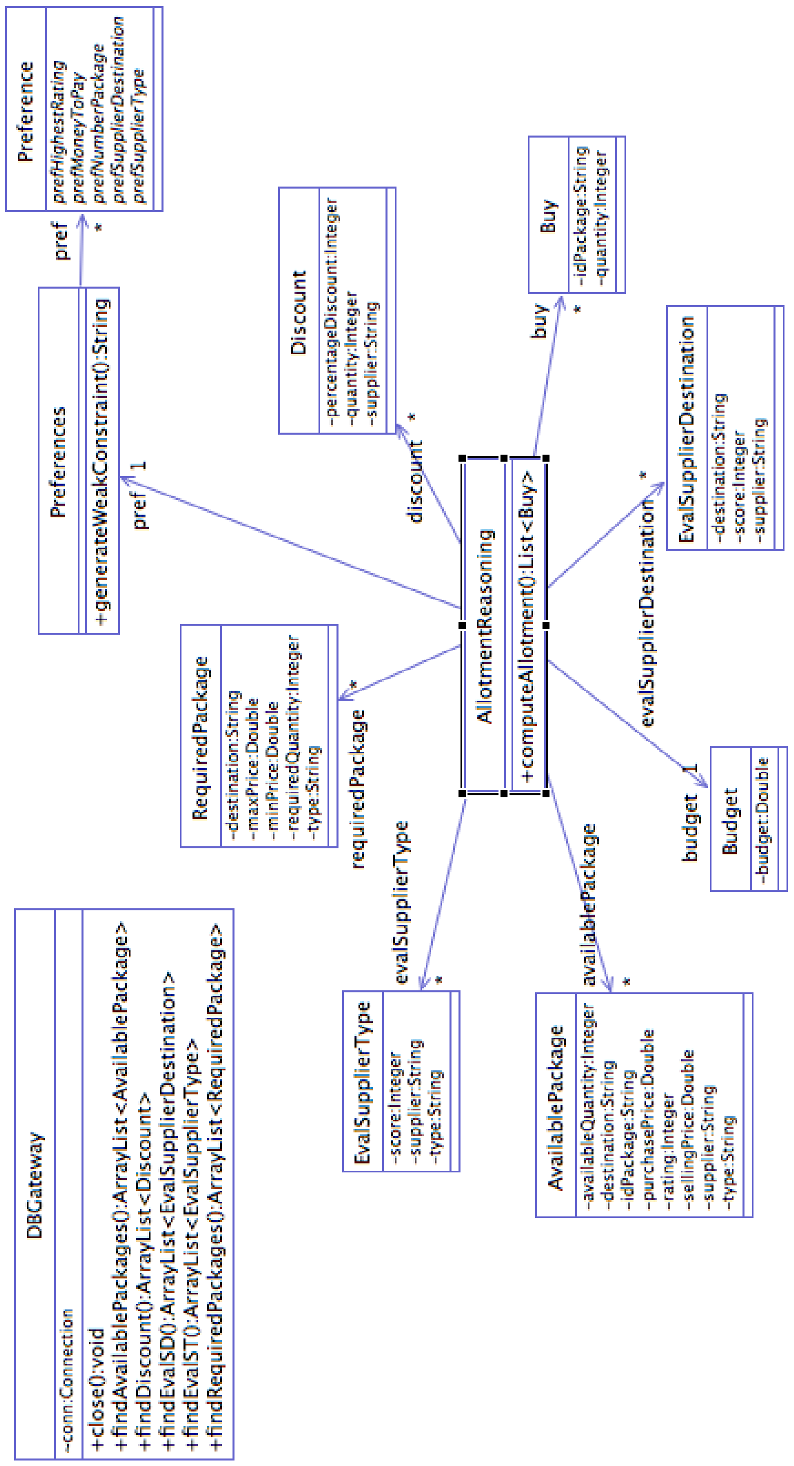
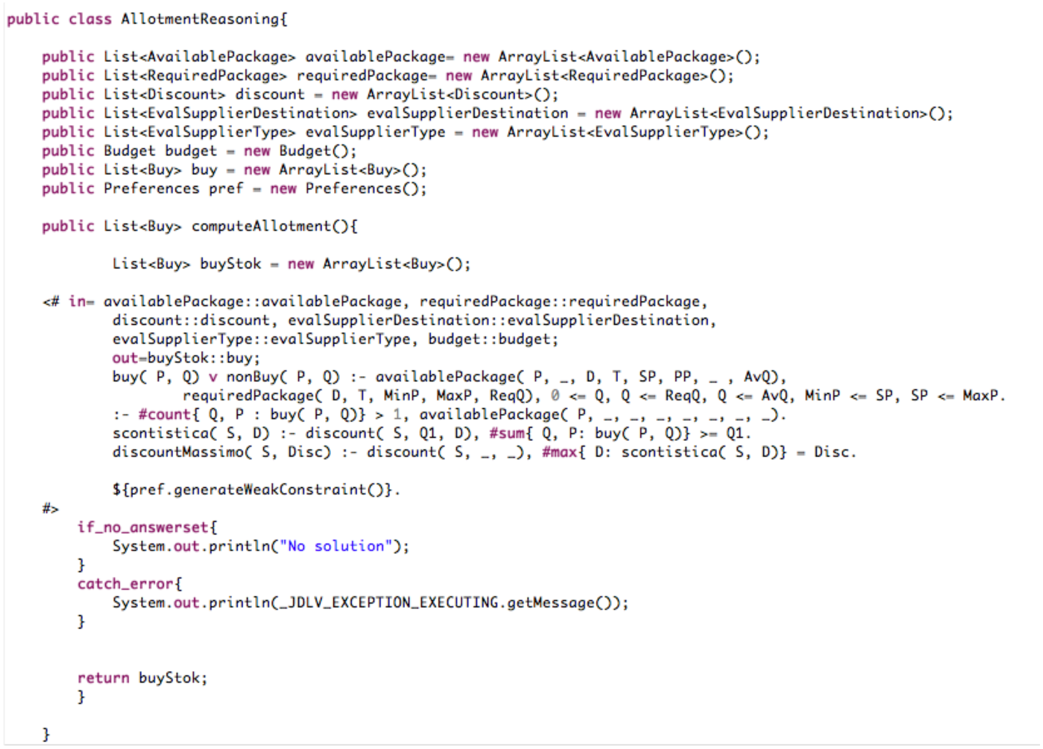


Figure 4.3: Class Diagram of the Allotment API.



```
public class AllotmentReasoning{

    public List<AvailablePackage> availablePackage= new ArrayList<AvailablePackage>();
    public List<RequiredPackage> requiredPackage= new ArrayList<RequiredPackage>();
    public List<Discount> discount = new ArrayList<Discount>();
    public List<EvalSupplierDestination> evalSupplierDestination = new ArrayList<EvalSupplierDestination>();
    public List<EvalSupplierType> evalSupplierType = new ArrayList<EvalSupplierType>();
    public Budget budget = new Budget();
    public List<Buy> buy = new ArrayList<Buy>();
    public Preferences pref = new Preferences();

    public List<Buy> computeAllotment(){

            List<Buy> buyStok = new ArrayList<Buy>();

    <# in= availablePackage::availablePackage, requiredPackage::requiredPackage,
            discount::discount, evalSupplierDestination::evalSupplierDestination,
            evalSupplierType::evalSupplierType, budget::budget;
            out=buyStok::buy;
            buy( P, Q) v nonBuy( P, Q) :- availablePackage( P, _, D, T, SP, PP, _ , AvQ),
                    requiredPackage( D, T, MinP, MaxP, ReqQ), 0 <= Q, Q <= ReqQ, Q <= AvQ, MinP <= SP, SP <= MaxP.
            :- #count{ Q, P : buy( P, Q)} > 1, availablePackage( P, _, _, _, _, _, _, _).
            scontistica( S, D) :- discount( S, Q1, D), #sum{ Q, P: buy( P, Q)} >= Q1.
            discountMassimo( S, Disc) :- discount( S, _, _), #max{ D: scontistica( S, D)} = Disc.

            ${pref.generateWeakConstraint()}.
    #>
        if_no_answerset{
            System.out.println("No solution");
        }
        catch_error{
            System.out.println(_JDLV_EXCEPTION_EXECUTING.getMessage());
        }


        return buyStok;
        }

    }
```

Figure 4.4: $\mathcal{JASP}$ code for *AllotmentReasoning* class.

EvalSupplierDestination has an attribute to identify the supplier, an attribute to identify the destination of travel and an evaluation expressed about the supplier on the destination. *EvalSupplierType* has an attribute to identify the supplier, an attribute to identify the type of travel and an evaluation expressed about the supplier on the type of travel.

In the diagram of Figure 4.3 are also present the classes *DBGateway*, *Preference*, *Preferences* and *AllotmentReasoning*. The first handles the relational database where the input of the allotment module is stored, whereas the reasoning facility is provided by class *AllotmentReasoning*.

The *AllotmentReasoning* class has a unique method, named *computeAllotment()* that executes the logic program and returns the list of packages that the reasoning model suggest to buy. The source code of the reasoning method, which has been developed in $\mathcal{JASP}$, is depicted in Figure 4.4. The main method *computeAllotment()* includes a block of ASP code containing the fixed part of the encoding presented in Section 4.1.2. Since preference order can be decided depending on the user needs, weak constraints modeling preferences are added dynamically by using the feature of $\mathcal{JASP}$ that allows one to acces the host environment (see Chapter 3). Looking at the last line of the $\mathcal{JASP}$ block we see that this feature is exploited to add the weak constraints that are generated by calling *generateWeakConstraints()*. In more details weak constraints are represented by the values of the enumeration *Preference*, and they are stored in an instance of the class *Preferences*. This latter models a list in which so that the position of a preference object in that list corresponds to the level of the corresponding weak constraint in the ASP program. The Java code for the *AllotmentReasoning* class is then automatically generated by the JDLV plugin.

4.2 Intelligent Newsletter

The iTravel+ system has among the goals the one of providing an intelligent newsletter service for travel agencies customers.

Newsletters delivered electronically via email (e-Newsletters) have gained rapid acceptance in several business areas since the Internet technologies –in general–, and e-mails –in particular– become a widely-adopted communication media. The goal of the newsletter is to inform the subscribed customers about some news concerning specific topics they may be interested in. A newsletter can be used for commercial and marketing purposes, or as a very informative support tool for the users of web sites and portals. The email, in textual or HTML format, usually offer a summary of the news of a web site that may be of interest for a customer/user. An effective implementation of a newsletter service, hence, should bring important news to the attention of the user and leave other news on the bottom of a message. Moreover, it is important to control the length of messages, as well as the frequency they are sent, in order to avoid the newsletter messages are considered spam to be trashed. However different users may have different preferences and different interests, so the need for personalizing messages depending on the user they are addressed is very important for increasing the effectiveness of this communication media.

In this section we present a solution to the problem of scheduling and organizing messages for the newsletter service of a travel agent, that was devised according to the requirements of the iTravelPlus project.

4.2.1 Requirements of the iTravel+ Newsletter

The newsletter of iTravel+ was not specified as the typical recommender system. Indeed, it is not intended to send customized messages via email containing commercial proposals, discounts or special offers or to encourage customers to buy something. Instead the idea is to provide each registered customer with a number of news it can be interested in regarding several topics, such as information on the place(s) he/she is visiting, suggested destinations for the next trip, as well as administrative news, availability of travel document, meteo, safety news (e.g., alerts on whether the travel destination was affected by an earthquake or an other natural phenomenon that will be dangerous for the personal safety); up to news about local traditions, festivals, concerts and so on. Basically, the news service is not specialized on a single topic, but regards many possible ones. As a consequence, a mailing system the just sends news as soon as they are available, without personalizing the content can result in a annoying service where the user is flooded by many possibly irrelevant messages. The customization system should organize the news by selecting and by ordering them in the best possible and attractive way to each user. This means that also the length of the email should be customized. The user can decide how many messages he/she prefers to receive for every email and the topics he/she is interested in a specific period of time. Some of these preferences can be specified during the registration or can be determined by applying well-known data mining techniques, some others should be deduced by applying some form of reasoning during the composition of messages.

The system may have a large quantity of news, which are classified by category. Each news is marked as *Urgent* or *Regular*. For each user, a news can be interesting if it is about a selected category (preferences expressed by the user during the registration step) or if a news is about an interesting travel destination for the user. The interesting destination could be the destination of the next booked travel, just about a place where it is expected to travel. Obviously, the user should not receive the same news multiple times. Once the relevant news for a user are determined, they must be grouped and sent by respecting constraints on message length and frequency, trying to maximize the useful information. This corresponds to determining the right schedule for messages to be sent in a fixed period. In particular the requirement is to build a weekly scheduling of messages to distribute. In this schedule the urgent news are sent as soon as possible. Each news has an expiration date, so, if the current date is over the expiration date, the expired news are trashed. During the scheduling task we try to maximize the number of dispatched news before the expiration date (or, that is the same, we try to minimize the number of news that are not dispatched before their expiration date). Since there is a continuous flow of new news, the scheduling is re-executed every day by looking ahead seven days so to full fill requirements.

In the following we present an ASP program that is able to select the news and schedule their composition in one or more email to be sent according to the user preferences.

4.2.2 Specification in ASP

This section illustrates the ASP program which implements the customized newsletter service specified in the previous section. First, the input data is described, then, the ASP rules solving the problem are presented.

Data Model. The input of the process is specified by means of the predicates described in the following:

- Instances of predicate *news(News, Place, Category, Urgent, Expiration_Date)* represent the set of the news in input. Concerning the attributes, *New* is the news identifier, *Place* is the identifier of the place which the news refers, *Category* specifies the category of the news (i.e., ”security”, ”roads condition”, etc.), *Urgent* is a flag that specifies if the news is an urgent news or not, *Expiration_Date* allows to discard expired news (e.g., sending the news concerning a concert is not useful after the concert took place).

- Instances of predicate *user(User, NewsForEmail, EmailForWeek)* represent the user registered to the newsletter. Concerning the attributes, *User* represents the user to which send the emails, *NewsForEmail* is the number of news that he/she would receive in a week, *EmailForWeek* is the max number of email that he/she would receive in a week.

- Instances of predicate *userTravelDestination(User, Place, End_Date)* model places where a given user is traveling or he/she is expected to travel. Concerning the attributes, *User* is the identifier of the user, *Place* is the destination, *End_Date* is the end date of the travel.

- Instances of predicate *userDestination(User, Destination)* specify destinations that are of interest for users.

- Instances of the predicate *userCategory(User, Category)* specify the category of news that can be of interest for users.

- Instances of the predicate *sent(News, User, Day)* model the set of news already sent to a given user in a specific day of the week.

- The predicate *today(Today)* has an unique instance. It represents the current day for which the program is expected to compute the messages to be sent.
- The predicate *currentWeek(startDay, endDay)* has a unique instance, which identifies in current year the current week. Concerning attributes, *startDay* and *endDay* are respectively the first day of the week and the last day of the week.
- The predicate *nextWeek(startDay, endDay)* has a unique instance, which identifies the next week for which we are planning the emails. The next week is also taken into account to enlarge the horizon of our reasoning. Concerning attributes, *startDay* and *endDay* are respectively the first day of the week and the last day of the next week.
- *maxNewsForDay(1..maxNews)* is an auxiliary predicate used to define the max number of news for a day.
- *day(1..365)* is an auxiliary predicate used to define the days of the year.

Newsletter Encoding. In this section we describe the ASP rules for solving the newsletter problem. In particular, the goal is to compute the set of news to be sent in a week from today.[2] To this end, we adopt to the *Guess&Check&Optimize* programming methodology [25, 45], but before presenting the guessing rule, we report some auxiliary ones that prepare the input for the next steps. The rule:

$$urgent(N) :\!-\, news(N, \_, \_, urgent, \_). \tag{4.13}$$

identifies the urgent news having priority over the others.

We consider of interest for a user all the news concerning a travel destination he/she is visiting or he/she has planned to visit in the future. Thus, with the following rules the news are selected for a user if: (i) the destination is about a place of interest for the user, (ii) it is about a category specified by the user, (iii) if it is not expired, and (iv) if it was not previously sent.

[2]Since this program is expected to be run every day to obtain the expected behavior for the system, “today” indicates the day the scheduling is run.

$$\begin{aligned} interestingNews(U, N, ExpDate, Urg, L) \text{ :- } & news(N, L, Urg, ExpDate), \\ & userCategory(U, C), \\ & userDestination(U, L), \\ & today(O), ExpDate > O, \\ & notsent(N, U, GP), \\ & day(GP), GP < O. \end{aligned} \tag{4.14}$$

$$\begin{aligned} userDestination(U, L) \text{ :- } & userTravelDestination(U, L, D), D > O, \\ & today(O). \end{aligned} \tag{4.15}$$

The next rule guesses the news to be sent today and in the following seven days:

$$\begin{aligned} send(User, News, Day) \text{ v } notSend(User, News, Day) \text{ :- } & \\ & interestingNews(User, News, End, Urg, L), \\ & weekDays(Day), End \geq Day. \end{aligned} \tag{4.16}$$

where rule:

$$weekDays(X) \text{ :- } day(X), today(O), X \geq O, X < Ops, Ops = O + 7. \tag{4.17}$$

is used to calculate the days of the week that starts from today.

Urgent news are sent immediately:

$$\begin{aligned} send(User, News, Day) \text{ :- } & interestingNews(User, News, End, Urg, L), \\ & toDay(Today). \end{aligned} \tag{4.18}$$

To ensure that a news is not sent several times, we write the following constraint:

$$\text{:- } send(User, News, Day1), send(User, News, Day2), Day1! = Day2.$$

(4.19)

This constraint expresses the condition that is not possible to send the same news to a user in two different days.

We use also a constraint to set the maximum number of news to collect in the same email for each user, as expected the urgent news are not counted:

$$\begin{aligned} &:\!-\ \#count\{(N : send(U,N,G), noturgent(N)\} > NewsForEmail,\\ &\quad user(U, NewsForEmail, EmailForWeek), weekDays(G). \end{aligned} \tag{4.20}$$

An other constraint is used to impose the maximum number of emails per week for each user:

$$\begin{aligned} &:\!-\ \#count\{(Day : sentNotUrgentInDate(User, Day), Day \geq I,\\ &\quad Day \leq F, currentWeek(I,F)\} > EmailForWeek,\\ &\quad user(User, NewsForEmail, EmailForWeek). \end{aligned} \tag{4.21}$$

$$\begin{aligned} &:\!-\ \#count\{(Day : sentNotUrgentInDate(User, Day), Day \geq I,\\ &\quad Day \leq F, nextWeek(I,F)\} > EmailForWeek,\\ &\quad user(User, NewsForEmail, EmailForWeek). \end{aligned} \tag{4.22}$$

$$\begin{aligned} &sentNotUrgentInDate(User, Day) :\!-\ sent(User, News, Day),\\ &\qquad\qquad \text{not } urgent(News).\\ &sent(News, User, Day) :\!-\ send(User, News, Day). \end{aligned} \tag{4.23}$$

Among the possible ways of collecting news and sending messages in the next week we select some that satisfy a number of optimization criteria.

We start by minimizing the number of emails not sent for respecting the expiration dates:

$$\begin{aligned} &:\sim\ lostNews(N,U).[1@2]\\ &lostNews(N,U) :\!-\ interestingNews(U,N,Date,Urg,L),\\ &weekDays(PrecG), PrecG < Date, \text{not } sent(N,U,PrecG). \end{aligned} \tag{4.24}$$

Then we try to maximize the news sent to a user in a given day:

$$\begin{aligned}
&:\sim\#count\{N : send(U,N,G), noturgent(N)\} = newsInADay,\\
&\quad Pay = NewsForMail - NewsInADay,\\
&\quad maxNewsForDay(newsInADay),\\
&\quad newsInADay < NewsForMail,\\
&\quad user(U, NewsForMail, NewsForWeek),\\
&\quad weekDays(G).[Pay@3]
\end{aligned} \tag{4.25}$$

Moreover news concerning current destinations of a user are preferred:

$$\begin{aligned}
&:\sim userTravelDestination(U,L,Date), notsentToUser(N,U),\\
&\quad insterstingNews(U,N,EndDate,,L).[1@4]
\end{aligned} \tag{4.26}$$

The number of days in which the emails are send is minimized:

$$\begin{aligned}
&:\sim\ sentInDate(User,Day).[1@5]\\
&sentInDate(User,Day) \text{ :- } send(User,News,Day).
\end{aligned} \tag{4.27}$$

Eventually the number of interesting news is maximized:

$$\begin{aligned}
&:\sim interestingNews(U,N,S,notUrgent,L),\\
&\quad \text{not } sentToAUser(N,U).[1@10]\\
&\quad sentToAUser(News,User) \text{ :- } send(User,News,Day).
\end{aligned} \tag{4.28}$$

4.2.3 Implementation in $\mathcal{JASP}$

The ASP program provided in previous section was specified having in mind that it will be integrated in the iTravel+ system. In this section we describe the development of a Java API that provides all the interfaces that are needed to integrate our ASP program in a service of the iTravel+ system. As we did in case of the allotment problem in previous section, we provided a specification in Java of the data model, and we included additional Java classes for handling the input from a database and used $\mathcal{JASP}$ to embed ASP in the implementation of a facade class. The resulting class diagram is depicted in the Figure 4.5. The classes *Sent, News, Today, CurrentWeek, LastWeek, User, UserCategory, UserDestination, UserTravelToDestination* represent the mapping with the logic predicates in the domain,

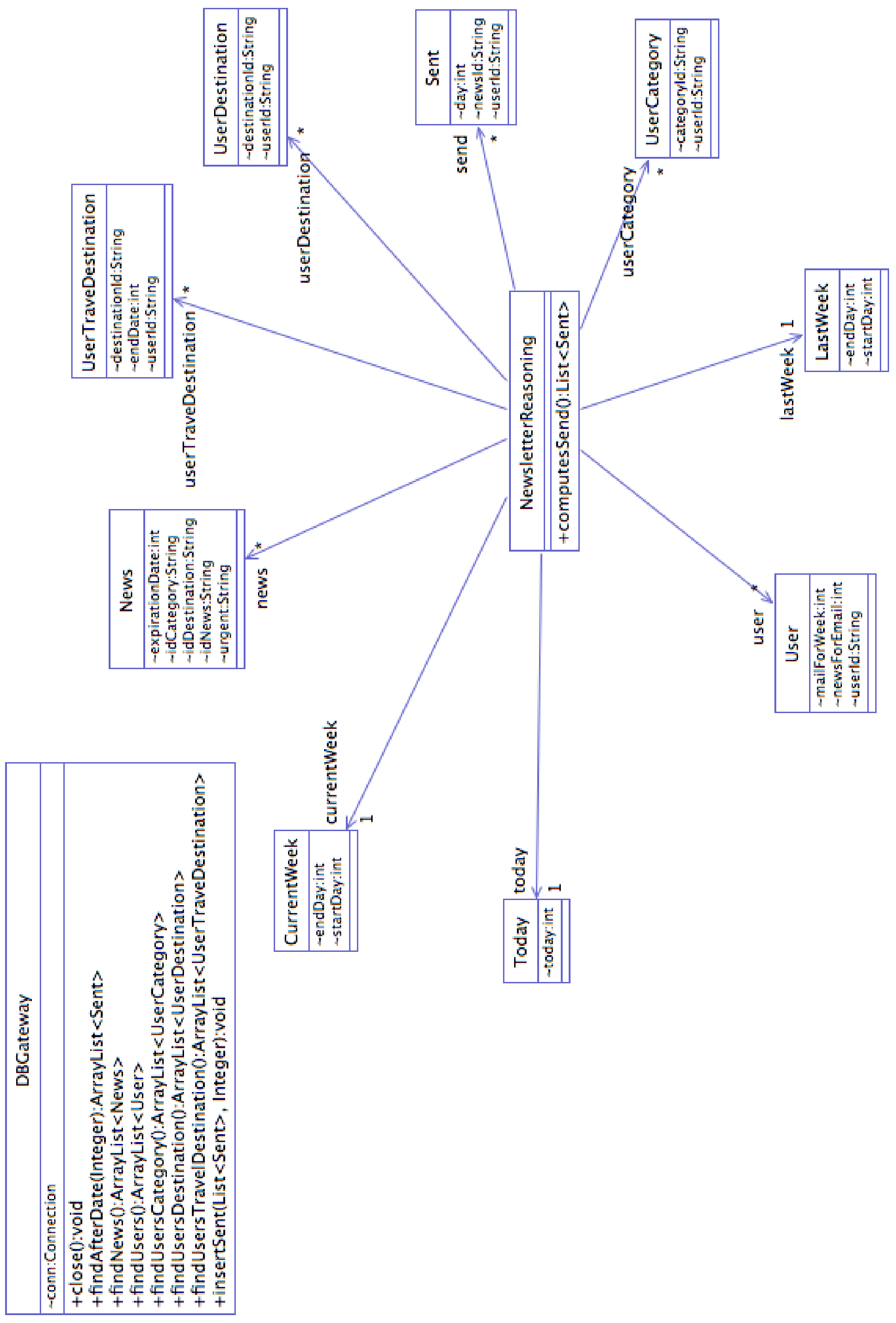


Figure 4.5: Class Diagram of the newsletter API.

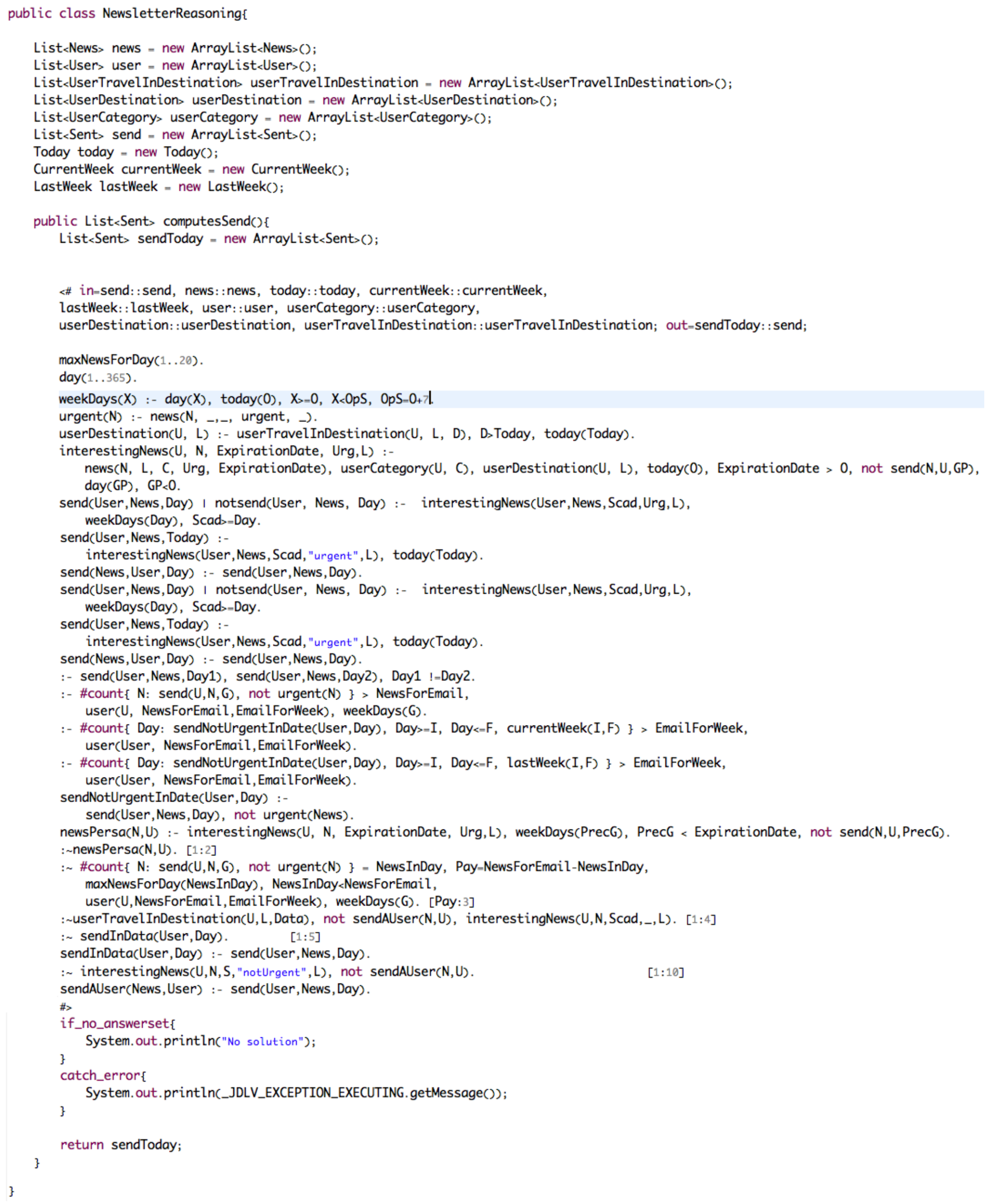
and as before Java classes have the same name of the corresponding logic predicates. The Class *DBGateway* manage the mapping between the database and the java objects. Finally, the main class is named *NewsletterReasoning* and provides a method *computeSend()* to be called each time we want to call an ASP solver to produce as output the set of news to be sent. The implementation of this method has been obtained by using the JDLV plugin and is depicted in Figure 4.6. Also in this case, the ASP encoding presented in this section is embedded in a $\mathcal{JASP}$ module, and the solver is automatically called whenever the corresponding method is called.

4.3 Related work

In the literature there are solution to many e-tourism systems challenges including: package tours search and assemblage, automatic holiday advisors, modeling of general purpose ontologies of the touristic domain [49, 18, 19, 52, 53, 61, 24, 44], etc. These studies do not focus –to the best of our knowledge– on helping travel agents in the act of selecting package tours to be traded with service suppliers in the future market. Concerning the applications of ASP, we mention that it has been used to develop several industrial applications [40, 68] and, in particular, it has already been exploited in an e-tourism system [37]. Nonetheless, the problem considered in [37] was to identify the package tours that best suit the needs of a customer of an e-tourism platform; thus, [37] approaches a different problem of the one considered here.

For the sake of completeness, we also mention a different way of dealing with the problem of allotment [78]. This approach aims at acquiring directly and on-demand from hotel management services the information about the hotel rooms and facilities that suit the request of a tour operator, so to avoid the allotment problem using agent technologies [78]. This is clearly a radically different approach from ours that aims at optimizing the pre-booking of allotments for an entire period of time.

Summarizing, this chapter presents the first attempt to exploit ASP for assisting tour operators in the allotment of packages, as well as the fist attempt to employ ASP for modeling the business logic of an intelligent newsletter.



```
public class NewsletterReasoning{

    List<News> news = new ArrayList<News>();
    List<User> user = new ArrayList<User>();
    List<UserTravelInDestination> userTravelInDestination = new ArrayList<UserTravelInDestination>();
    List<UserDestination> userDestination = new ArrayList<UserDestination>();
    List<UserCategory> userCategory = new ArrayList<UserCategory>();
    List<Sent> send = new ArrayList<Sent>();
    Today today = new Today();
    CurrentWeek currentWeek = new CurrentWeek();
    LastWeek lastWeek = new LastWeek();

    public List<Sent> computesSend(){
        List<Sent> sendToday = new ArrayList<Sent>();


        <# in=send::send, news::news, today::today, currentWeek::currentWeek,
        lastWeek::lastWeek, user::user, userCategory::userCategory,
        userDestination::userDestination, userTravelInDestination::userTravelInDestination; out=sendToday::send;

        maxNewsForDay(1..20).
        day(1..365).
        weekDays(X) :- day(X), today(O), X>=O, X<OpS, OpS=O+7.
        urgent(N) :- news(N, _,_, urgent, _).
        userDestination(U, L) :- userTravelInDestination(U, L, D), D>Today, today(Today).
        interestingNews(U, N, ExpirationDate, Urg,L) :-
            news(N, L, C, Urg, ExpirationDate), userCategory(U, C), userDestination(U, L), today(O), ExpirationDate > O, not send(N,U,GP),
            day(GP), GP<O.
        send(User,News,Day) | notsend(User, News, Day) :-  interestingNews(User,News,Scad,Urg,L),
            weekDays(Day), Scad>=Day.
        send(User,News,Today) :-
            interestingNews(User,News,Scad,"urgent",L), today(Today).
        send(News,User,Day) :- send(User,News,Day).
        send(User,News,Day) | notsend(User, News, Day) :-  interestingNews(User,News,Scad,Urg,L),
            weekDays(Day), Scad>=Day.
        send(User,News,Today) :-
            interestingNews(User,News,Scad,"urgent",L), today(Today).
        send(News,User,Day) :- send(User,News,Day).
        :- send(User,News,Day1), send(User,News,Day2), Day1 !=Day2.
        :- #count{ N: send(U,N,G), not urgent(N) } > NewsForEmail,
            user(U, NewsForEmail,EmailForWeek), weekDays(G).
        :- #count{ Day: sendNotUrgentInDate(User,Day), Day>=I, Day<=F, currentWeek(I,F) } > EmailForWeek,
            user(User, NewsForEmail,EmailForWeek).
        :- #count{ Day: sendNotUrgentInDate(User,Day), Day>=I, Day<=F, lastWeek(I,F) } > EmailForWeek,
            user(User, NewsForEmail,EmailForWeek).
        sendNotUrgentInDate(User,Day) :-
            send(User,News,Day), not urgent(News).
        newsPersa(N,U) :- interestingNews(U, N, ExpirationDate, Urg,L), weekDays(PrecG), PrecG < ExpirationDate, not send(N,U,PrecG).
        :~newsPersa(N,U). [1:2]
        :~ #count{ N: send(U,N,G), not urgent(N) } = NewsInDay, Pay=NewsForEmail-NewsInDay,
            maxNewsForDay(NewsInDay), NewsInDay<NewsForEmail,
            user(U,NewsForEmail,EmailForWeek), weekDays(G). [Pay:3]
        :~userTravelInDestination(U,L,Data), not sendAUser(N,U), interestingNews(U,N,Scad,_,L). [1:4]
        :~ sendInData(User,Day).        [1:5]
        sendInData(User,Day) :- send(User,News,Day).
        :~ interestingNews(U,N,S,"notUrgent",L), not sendAUser(N,U).                    [1:10]
        sendAUser(News,User) :- send(User,News,Day).
        #>
        if_no_answerset{
            System.out.println("No solution");
        }
        catch_error{
            System.out.println(_JDLV_EXCEPTION_EXECUTING.getMessage());
        }

        return sendToday;
    }

}
```

Figure 4.6: $\mathcal{JASP}$ code for *NewsletterReasoning* class.

Chapter 5

Extensions of ASPIDE

The design of logic programs has been made more comfortable to users since the proposal of the first advanced editors for ASP. There are however special categories of users with specific needs that still need more specialized tools to develop ASP programs comfortably. In this chapter we describe two new development tools extending the ASPIDE environment for ASP that cope with two separate but relevant categories of users. The first one is a a new system that allows for *drawing* an ASP-program on the screen, so that a user does not have to edit text files or know the details of the ASP syntax. As a consequence, it is expected to reduce the difficulty of producing ASP programs for both novice and inexperienced programmers, but also making more comfortable the encoding tasks for experts who prefer graphic tools. The second one was inspired by the growing community interested in approaches that resort to Datalog (and its extensions) for implementing various reasoning tasks over ontologies. We noticed that in this specific field, the editing environments for ontologies –on the one hand– and logic programming –on the other hand– are often developed independently and miss a common perspective. To facing with this issue we worked on the integration of *ASPIDE* with the ontology specification tool *protégé*. We extended both systems with specific plugins that enable a synergic interaction between the two development environments. The developer can then handle both ontologies and logic-based reasoning over them by exploiting specific tools integrated to work together.

5.1 Visual Editor

In order to facilitate the design of ASP applications, a rich set of tools for ASP-program development were proposed in the last few years, including editors [58, 74] and debuggers [9, 8, 27]. However, the task of designing a logic program consists of writing text files (more or less computer-assisted). Although the basic syntax of ASP is not particularly difficult, writing ASP programs might be uncomfortable for novices and error-prone. Extending the idea of Query By Example (QBE) interfaces[21] proposed in the area of databases for facilitating the approach of users to systems and languages, ASPIDE features a QBE-like editor for logic rules (see Chapter 3.1). However, the ASPIDE visual editor resulted to be not immediate and intuitive for non-expert users.

After analyzing the limits of this proposal we devised a new visual interface that supports all the powerful language constructs of ASP, like disjunction, recursion, unstratified negation, constraints, and aggregates. The use of *New Visual ASP* can encourage novice and unexperienced ASP programmers. This new editor is able to load and store ASP programs in the syntax of the ASP system DLV [45].

In the following we refer to the ASPIDE QBE-like interface as *Visual ASP* and we name the new tool introduced in this thesis as *New Visual ASP*.

5.1.1 *New Visual ASP*

New Visual ASP is conceived to the task of writing ASP programs by offering the possibility to draw single programs using a fully graphical environment. The goal is to support under a common visual syntax, and without editing the text files, all constructs of a modern ASP implementation as well as more recent extensions of logic languages for ontology-reasoning, such as rules with existential quantification in the head [15]. *New Visual ASP* should provide a new way of developing logic programs and rule-based reasoning, which we expect it to be more appreciated by users that like graphical environments as well as by non-expert users. Indeed, the user does not have to edit text files, or know the details of a specific implementation, but he can exploit a fully graphic tool for designing programs.

The description of a visual language in formal terms is usually complex and difficult to understand, thus we rather we adopt in this section a more direct description style based on running examples. Since we developed a prototype that we have integrated into an extended version of the ASPIDE environment, we present step by step the development of the examples in the environment by providing a number of snapshots of the running system, and a complete list of available commands. The *New Visual ASP* is not intended to replace the textual editor of ASPIDE, but to complement it, so that the user can choose to edit the ASP programs in the textual way or in the visual way. Basically, the user can

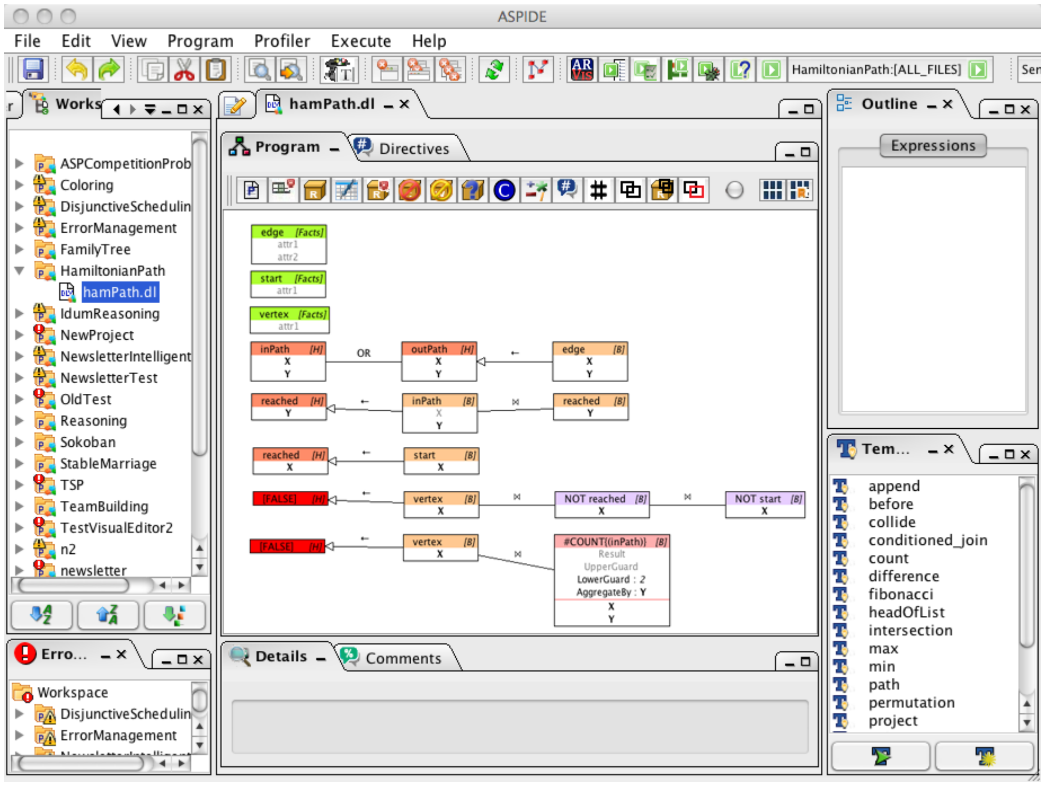


Figure 5.1: The New Visual Editor embedded in *ASPIDE*.

draw the program using the visual editor and then can switch to the textual editor to visualize the encoding of the problem in the ASP syntax and vice versa.[1]

In the following, we first provide an overview of the tool, and then we present the features available also by means of an use-case.

5.1.2 Overview of the new editor

In the new visual language rules are represented by means of a graph (see Figure 5.1) where atoms/literals are boxes connected by edges symbolizing logical connectives. Predicate arguments can be specified in apposite boxes, and are immediately visible. Joins between variables can be created by dragging and dropping variables. Each element is represented by a block coloring in a different way, i.e, facts are represented by a block with the topper coloring green, the predicates in the head of the rule are orange meanwhile the predicates in the body are light or-

[1]To switch between the different editors we can use the icon with a green double arrows in the toolbar of ASPIDE (see Figure 5.1).

ange, and so on. When the user draws an ASP program the syntactic correctness is imposed by design (e.g., the editor will not allow to add negation as failure in the head). However, in case some semantic condition is not verified, such as safety of rules, arities of predicates, the error is identified and the box that contains the error is marked with a small red icon.

The central region in 5.1 is where ASP programs are composed. On the top of this area there is a toolbar with the main operations needed to write the rules, namely from left to right:

- **New Predicate** allows to add a new atom in an existing rule. Selecting an atom/literal the new one will be inserted immediately after it;
- **Remove Entity** removes the selected atom/literal from the rule;
- **New Rule** creates a new rule, where both head and body are empty;
- **New Facts Table** allows to create a block of facts;
- **Remove Rules** removes the selected rules;
- **New Constraint** allows to create a new integrity constraint;
- **New Weak Constraint** allows to create a new weak constraint;
- **New Query** allows to create a new query;
- **New Operation** allows to insert a comparison operator or an a arithmetic operator after the selected entity;
- **New Directive** allows to define the directives of DLV (maxint, const, include, import, export);
- **Aggregate** allows to include an aggregate operation in a rule;
- **Collapse** allows to collapse two or more rule in a unique box to make more readable the program in the case it become more bigger graphically;
- **Collapse Rule** allows to collapse one or more predicate in a single box:
- **Expand** allows to expand the selected collapsed entity;
- **Zoom slider** allows to increase and decrease the size the drawing area;

- **Relayout All** automatically disposes rules in the editing area;
- **Relayout Expression** recomposes the layout of a rule.

Many of the previous operation are also available via drop-down menus (activated by clicking the right button of the mouse) or by clicking on specific icons appearing whenever the mouse is over a selected graphic representation.

On the right part of the main window of the *New Visual ASP*, there is an outline panel that is associated with the visual program. Into the outline can be visualized a list of the predicates allowed in the program. For each predicate, it is possible to expand the corresponding node and we can visualize the list of rule in which it is used.

5.1.3 A use case for *Visual ASP*

In the following paragraphs we show how *New Visual ASP* can be used to design ASP programs by exploiting an example.

Running Example. We consider the well-known problem called "Hamiltonian Path". Given a finite directed graph $G = \langle V, E \rangle$ and a node $X \in V$ of this graph, does there exist a path in G starting at X and passing through each node in V exactly once? This is a classical NP-complete problem in graph theory. Suppose that the graph G is specified by using facts over predicates $vertex$ (unary) and $edge$ (binary), and a starting node X is specified by the predicate $start$ (unary). HAMILTONIAM PATH can be encoded as follows:

% Facts
f_1: $vertex(v).$ $\quad\forall v \in V$
f_2: $edge(i, j).$ $\quad\forall (i, j) \in E$

% Guess arcs of the path
r_1: $inPath(X, Y) \text{ v } outPath(X, Y) \text{ :- } edge(X, Y, \_).$

% Auxiliary rules
r_2: $reached(X) \text{ :- } start(X).$
r_3: $reached(X) \text{ :- } reached(Y), inPath(Y, X).$

% Checking part:specify constraints on solution

$\%$ All vertexes must be in the path
r_4: $:\!-\ vertex(X), \text{not } reached(X), \text{not } start(X).$

$\%$ Each vertex in the path must have at most one
$\%$ incoming and one outgoing edge
r_5: $:\!-\ vertex(X), 2 <= \#count\{Y : inPath(X,Y)\}.$
r_6: $:\!-\ vertex(X), 2 <= \#count\{Y : inPath(Y,X)\}.$

The first two lines introduce suitable facts, representing the input graph G. The disjunctive rule r_1 guesses a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary predicate $reached$ is defined, which specifies the set of nodes which are reached from the starting node. In the checking part, the constraint r_4 enforces that all nodes in the graph are reached in the subgraph induced by S. The two constraints r_5 and r_6 ensure that the set of $arcs\ S$ selected by $inPath$ meets the following requirements, which any Hamiltonian Path must satisfy: (i) a vertex must have at most one incoming edge; (ii) a vertex must have at most one outgoing edge.

System Usage. We show how to employ *New Visual ASP* for *drawing* that encoding. Note that the system supports many different ways of creating and modifying rules and constraints, we mention only some of the possible combinations of commands and shortcuts that can be exploited for the considered program. The development of our Hamiltonian path encoding starts as usual in *ASPIDE*, i.e., one has to create a new Project, and a new file into the project. *ASPIDE* visualizes a blank text editor by default (see Figure 5.2). Thus, we switch to the graphical editor by clicking on the switch editor button (the one with green arrows in the main toolbar, see Figure 5.3). Now we can start “drawing” our ASP program. First of all we add the input facts: $vertex$, $edge$ and $start$. More in detail, there are two possibilities to add facts. The first one is to create a *New Facts Table*. In that case a box with a light green topper is added in the visual editor; and by clicking on this box, the details of the fact table are shown in the panel in the bottom of the window, labeled *Details* (see Figure 5.4). In that panel we specify the name of the predicate, and the attributes, and we can add or remove facts populating a table (see Figure 5.5). This solution is preferable when several instances have to be inserted, as it is probably the case for *edge* and *vertex* predicates. Indeed for *edge* we repeat the same operations made for creating the facts of *vertex*. The second options for creating a fact is to create a new rule, by clicking on *New Rule*

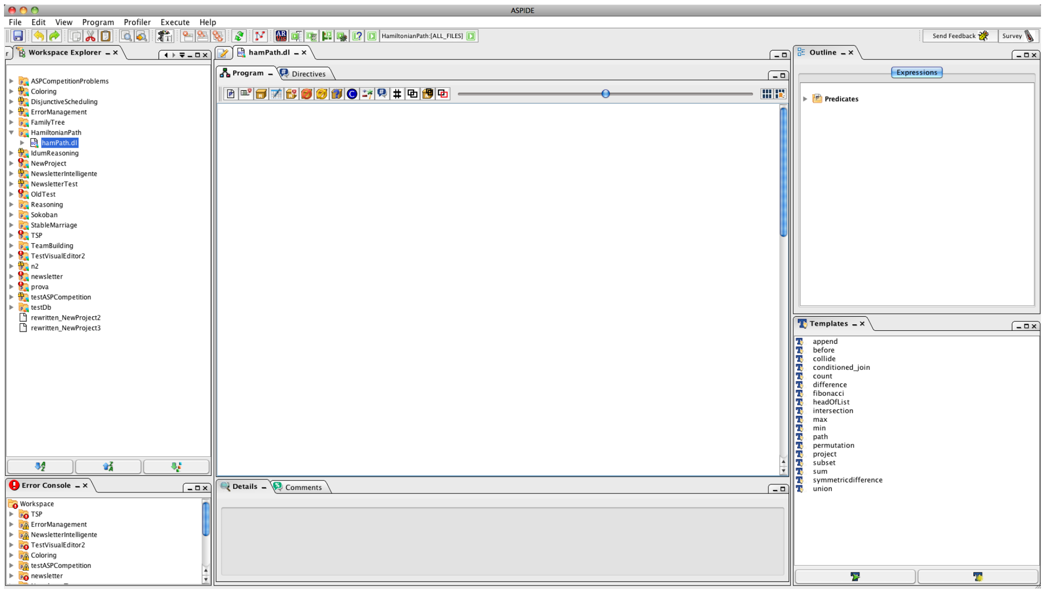


Figure 5.2: New File in *ASPIDE*.

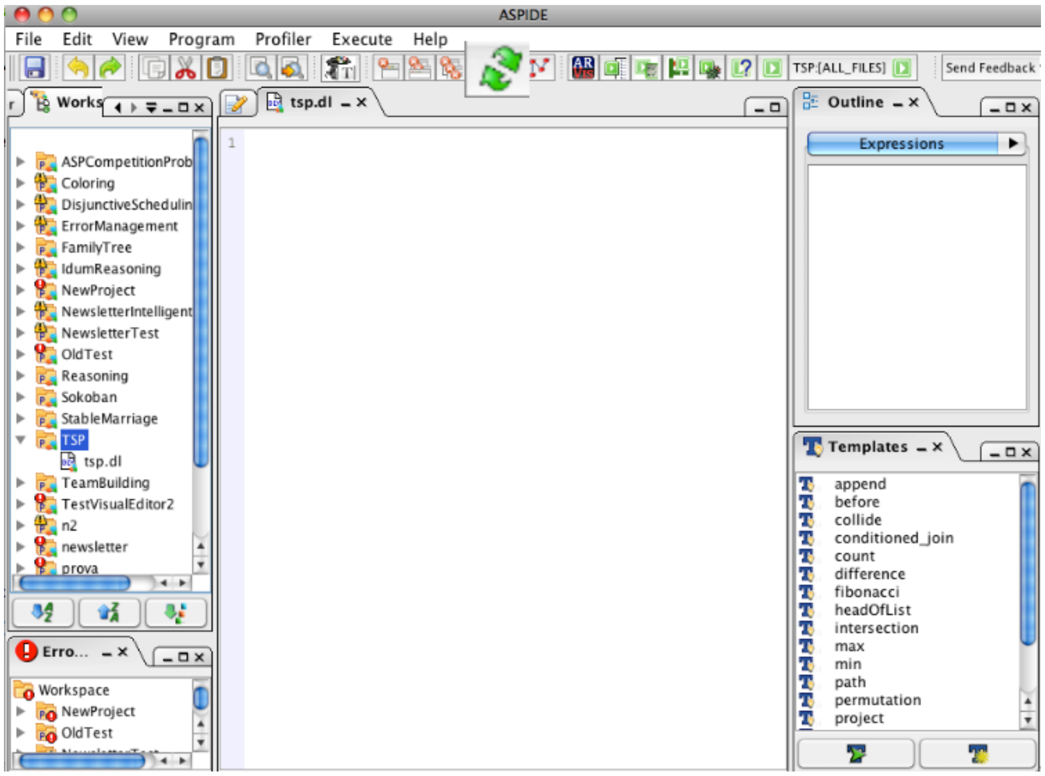


Figure 5.3: Open visual editor.

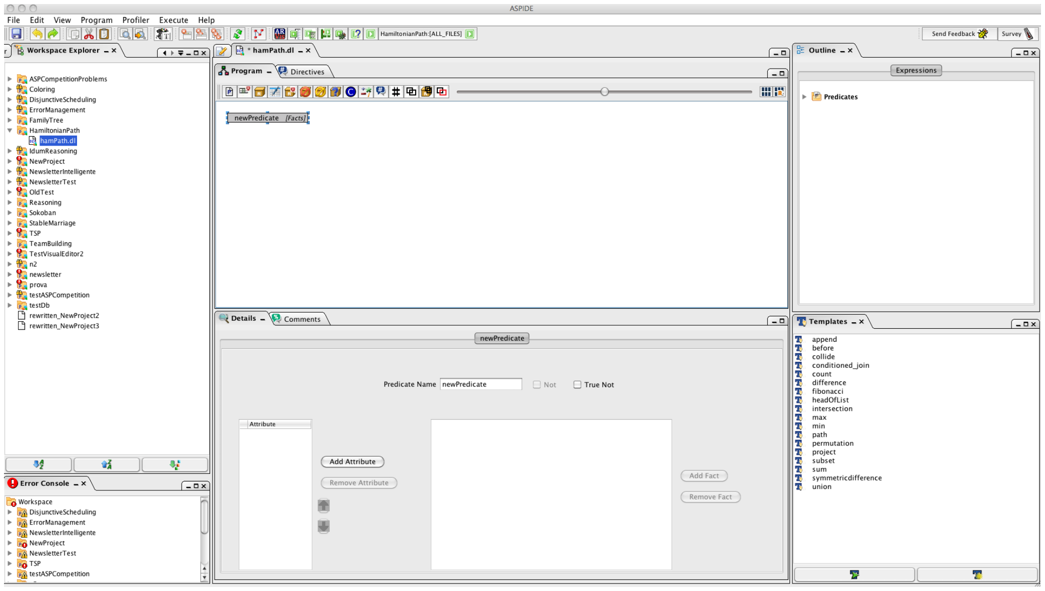


Figure 5.4: Modifying the details of the head atom.

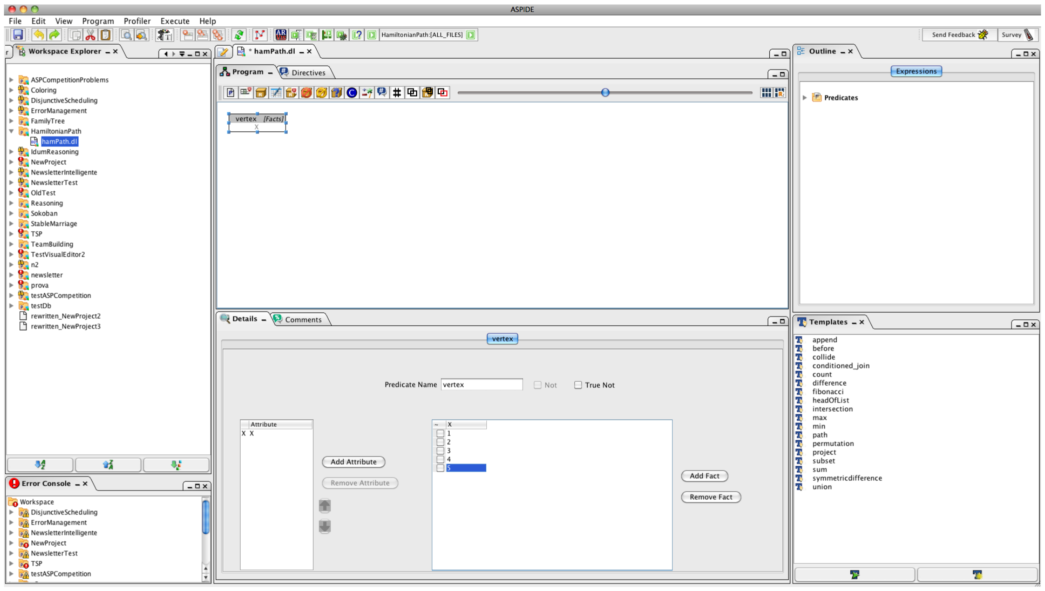


Figure 5.5: Vertex Facts.

and leaving empty the rule body. This one is more convenient when one single fact or a few facts have to be inserted. We use this second option for creating the predicate *start*. In particular, we create a new rule by clicking on *“New Rule”* in the toolbar, so we add a new empty rule, and then we adjust its content, i.e., we add one attribute and we fill it with “1” by double clicking in the box in the point corresponding to the attribute and typing the constant. The result of the complete insertion of the predicates *vertex*, *edge* and *start* is depicted in Figure 5.6.

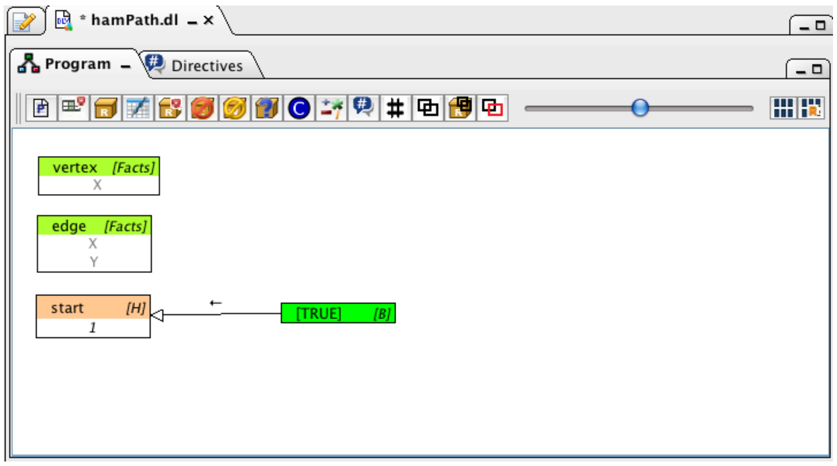


Figure 5.6: Add fact as a rule with empty body.

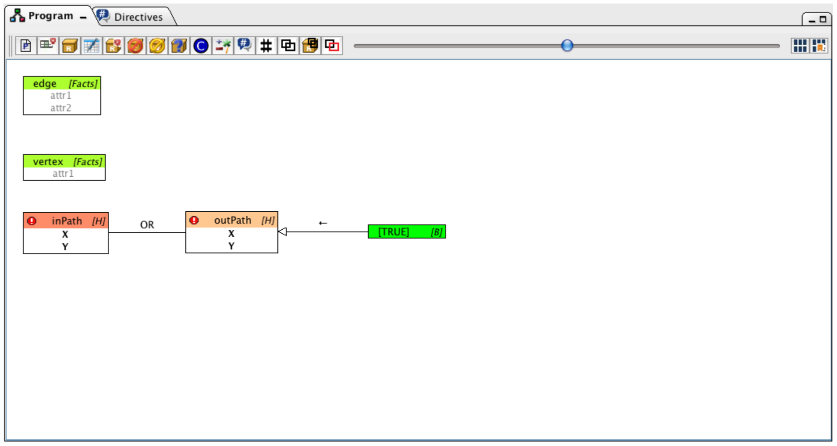


Figure 5.7: Unsafe disjunctive rule.

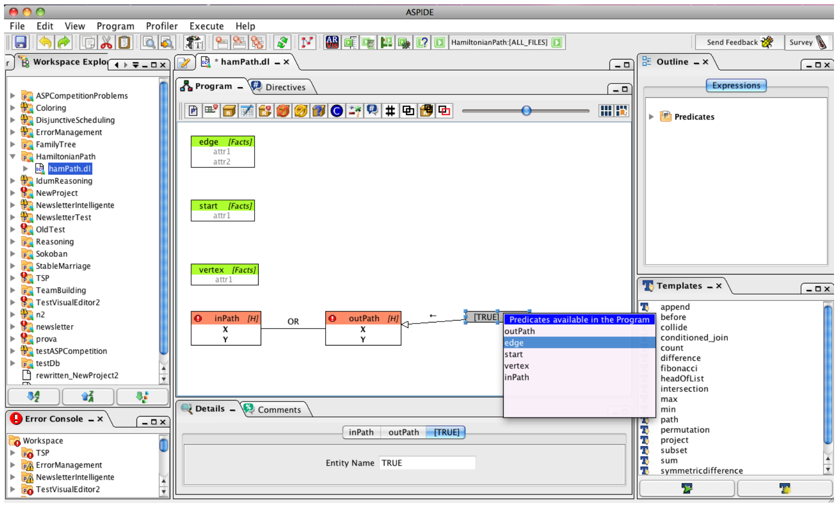


Figure 5.8: Create body.

Now we draw the disjunctive rule r_1 by clicking on the *New Rule* button on the toolbar, as we have done for *start* predicate, and we add atom *inPath*. *inPath* is binary, so we add two attributes and properly name them. Since r_1 is a disjunctive rule, we now add the atom *outPath* in the head of the rule. One possibility is to select the predicate *inPath* and click on the *New Predicate* button on the toolbar and then add the attributes. The connection between this two atoms is labeled with OR to indicate that the two are in a disjunction. At that point the rule is not safe (the body is missing), and the rule is marked as incorrect with a small red icon on the predicate box, as we can see in the Figure 5.7. Thus we go on and add the atom *edge* in the body of the rule. The first possibility to do that is to click on the head of the rule and select the icon *New Predicate*, depicted as a table icon with green add operator. In alternative one could select the head of the rule and click on the *New Predicate* button on the toolbar, or one could select *New Predicate* from the drop-down menu obtained by right-clicking on atoms of the head, or one could select *Insert Existing Predicate* from the drop-down menu obtained by right-clicking on atoms of the body (see Figure 5.8). In each case the tool adds a predicate with a generic name *newPredicate*, which has to be modified as we did for the head atoms. In particular we name it *edge*, and we perform the join with the atoms in the head by dragging each attribute of the head into the corresponding one of the body. After joining the attributes the rule becomes safe and no error icon is present in the rule now (see Figure 5.9).

We apply a similar process to write also the rules r_2 and r_3 (see Figure 5.10). Note that in the rule r_2, the arch in the body is labeled with the join symbol to indicate that they are in join relation between them. In this case, to create the second predicate in the body of r_2 we might use the *Join with New Predicate* icon. We ask to join *reached* with *start* on their attributes. Each selection is done acting on a list of possible matches that is generate automatically by the tool (see in sequence Figure 5.11, 5.12, Figure 5.13).

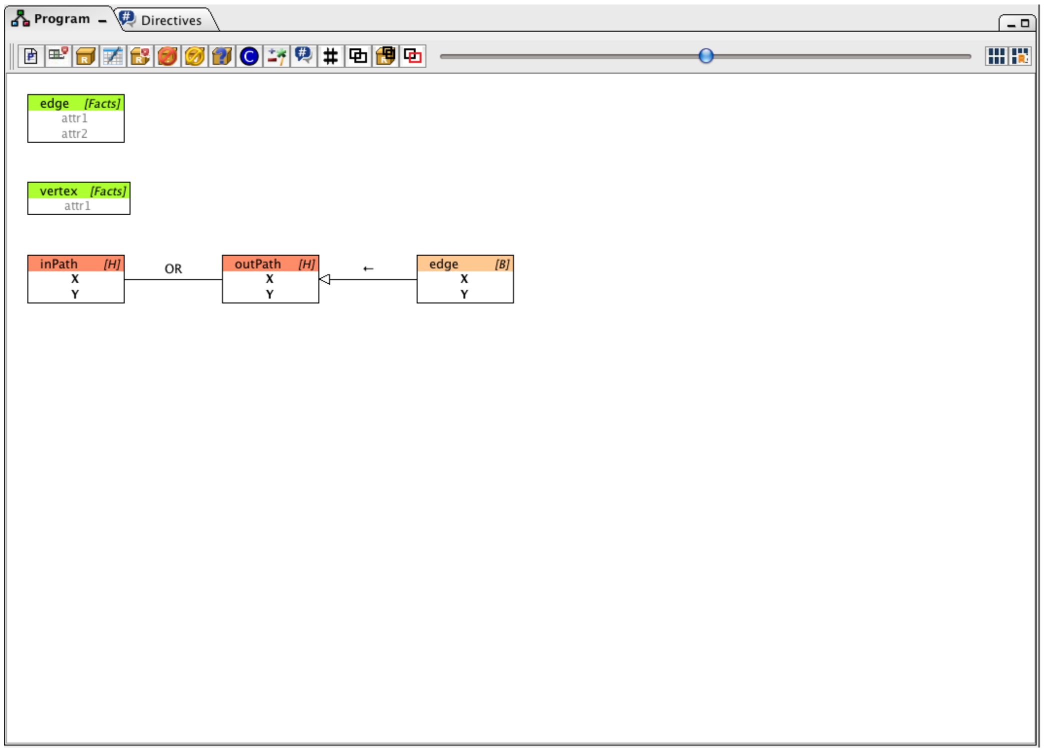


Figure 5.9: Complete disjunctive rule.

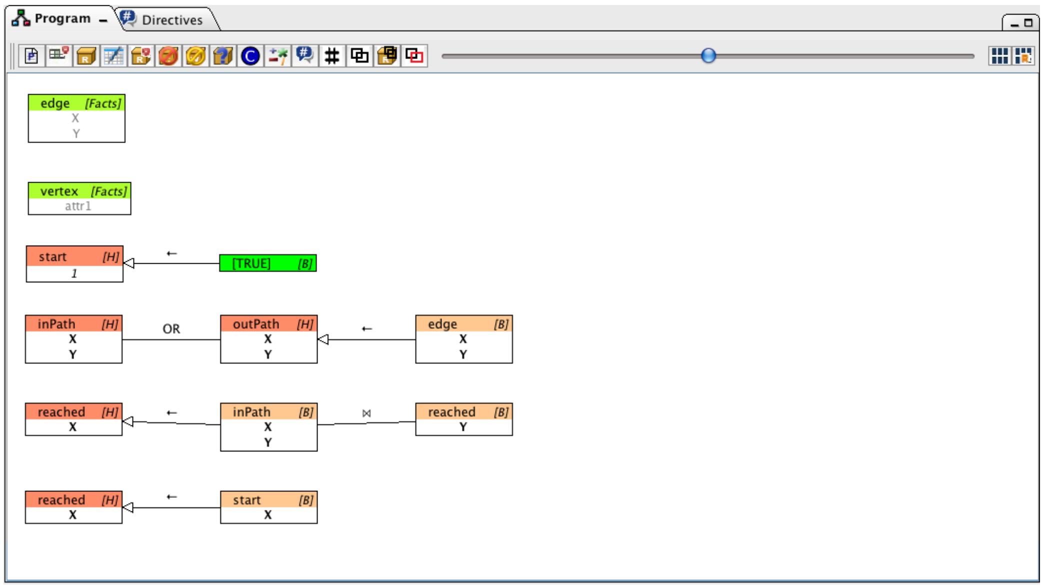


Figure 5.10: The first three rules of the program.

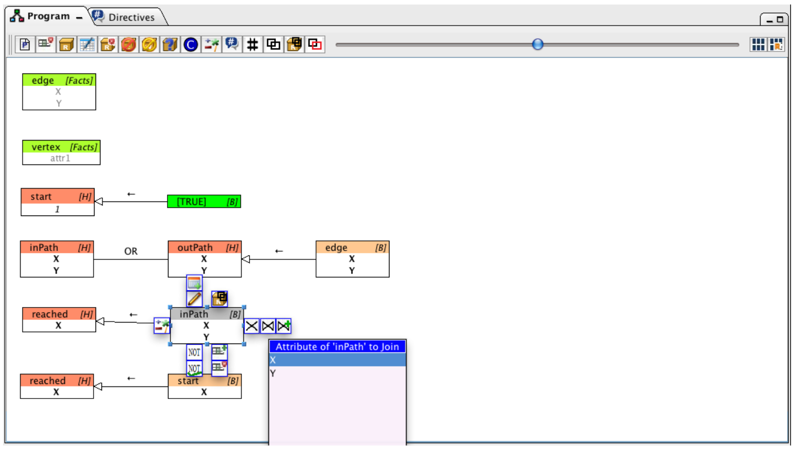


Figure 5.11: Select attribute to join.

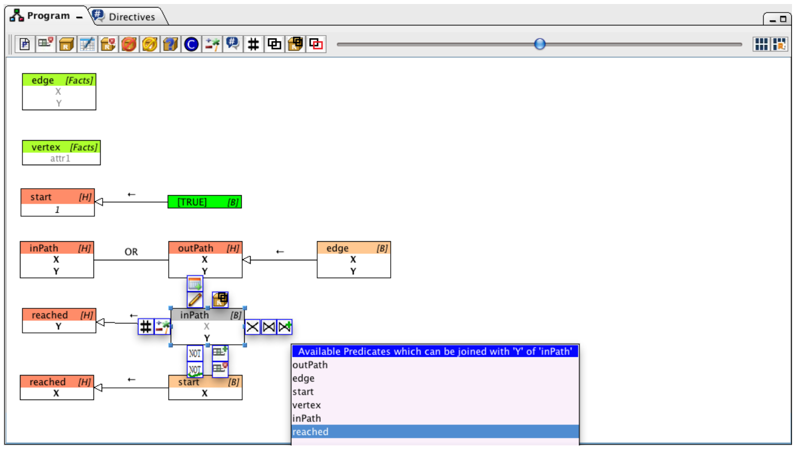


Figure 5.12: Select predicate to join.

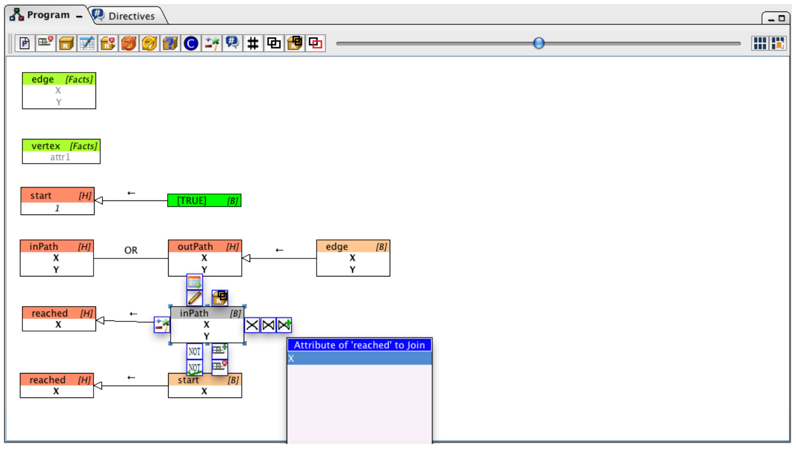


Figure 5.13: Select attribute of the predicate to join.

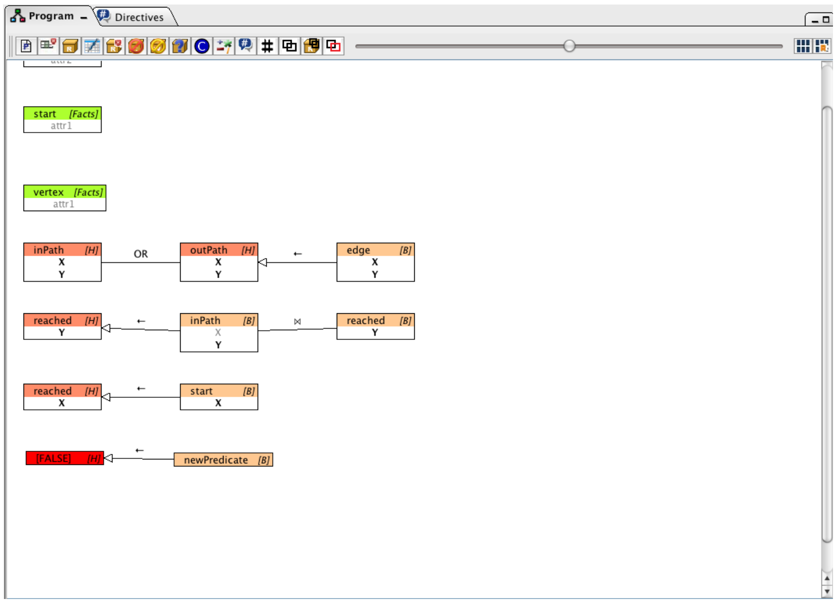


Figure 5.14: Adding new constraint.

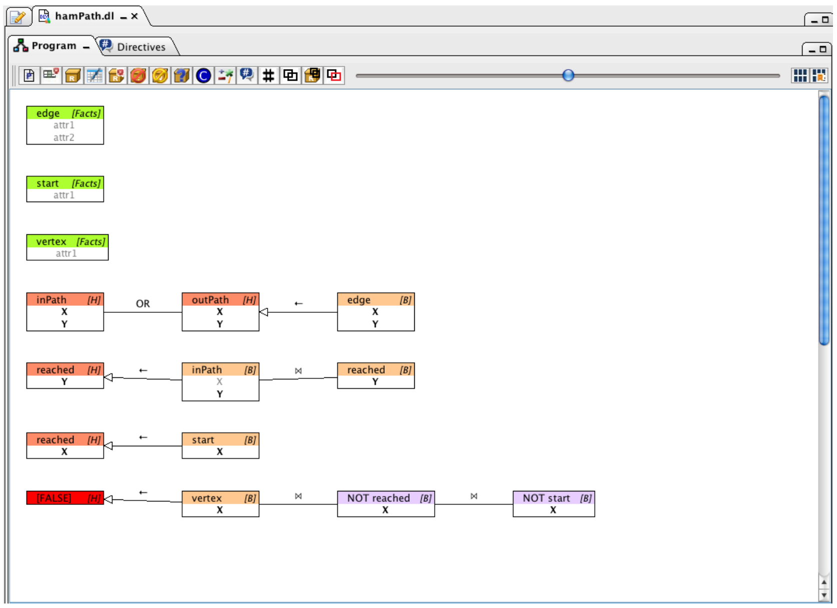


Figure 5.15: Program with the first constraint.

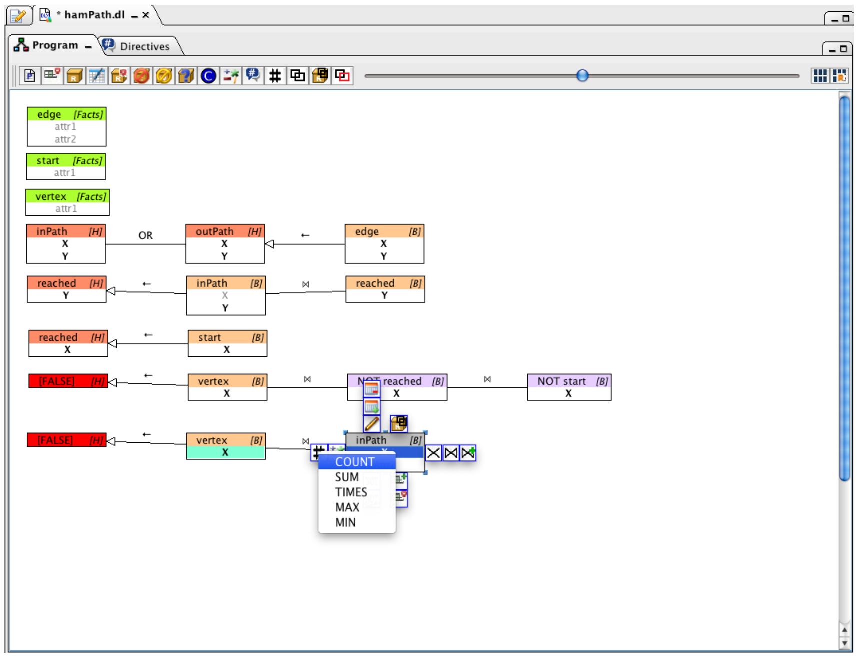
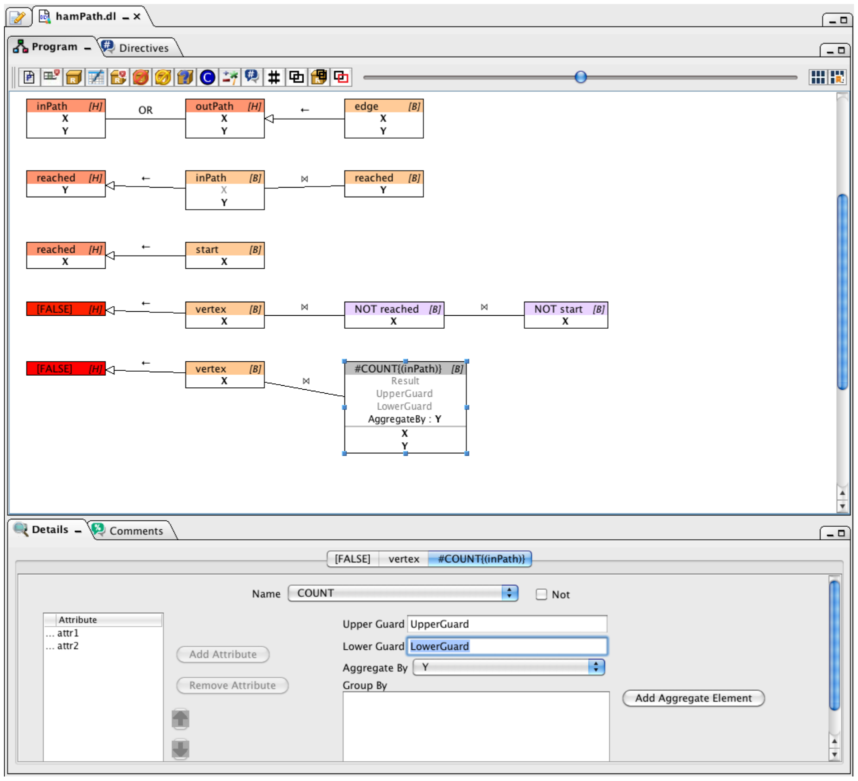


Figure 5.16: Adding aggregate atom.



[FALSE] | vertex | #COUNT{inPath}

Name: COUNT ☐ Not

Upper Guard: UpperGuard

Lower Guard: LowerGuard

Aggregate By: Y

Group By:

Figure 5.17: Adding guard to aggregate atom.

To create the constraint defined by the rule r_4, we click on the toolbar button *New Constraint*, which provides a rule with empty head and a literal in the body with a generic name, that we properly update (see Figure 5.14). Then we select the literal in the body to be negated and click on the NOT icon available around the box when it is selected. The resulting program is depicted in Figure 5.15.

The last rules, r_5 and r_6, are constraints with aggregate atoms. We create the constraint as we seen by adding *vertex* and *inPath* as before. Then we select *inPath* and we click the *Aggregate* icon around the box predicate or the *Aggregate* button on the toolbar, and select *COUNT* from a drop-down menu (see Figure 5.16). To complete the rule r_5 we add a lower guard to the aggregate with value equals to '2'. To do this, we double click on the box in the text field corresponding to *LowerGuard* or we can use the Detail tab and set the value of the field labeled *Lower Guard* (see Figure 5.17). Finally we repeat a similar procedure to insert the remaining rule, r_6.

The entire graphical representation of our program solving the Hamiltonian

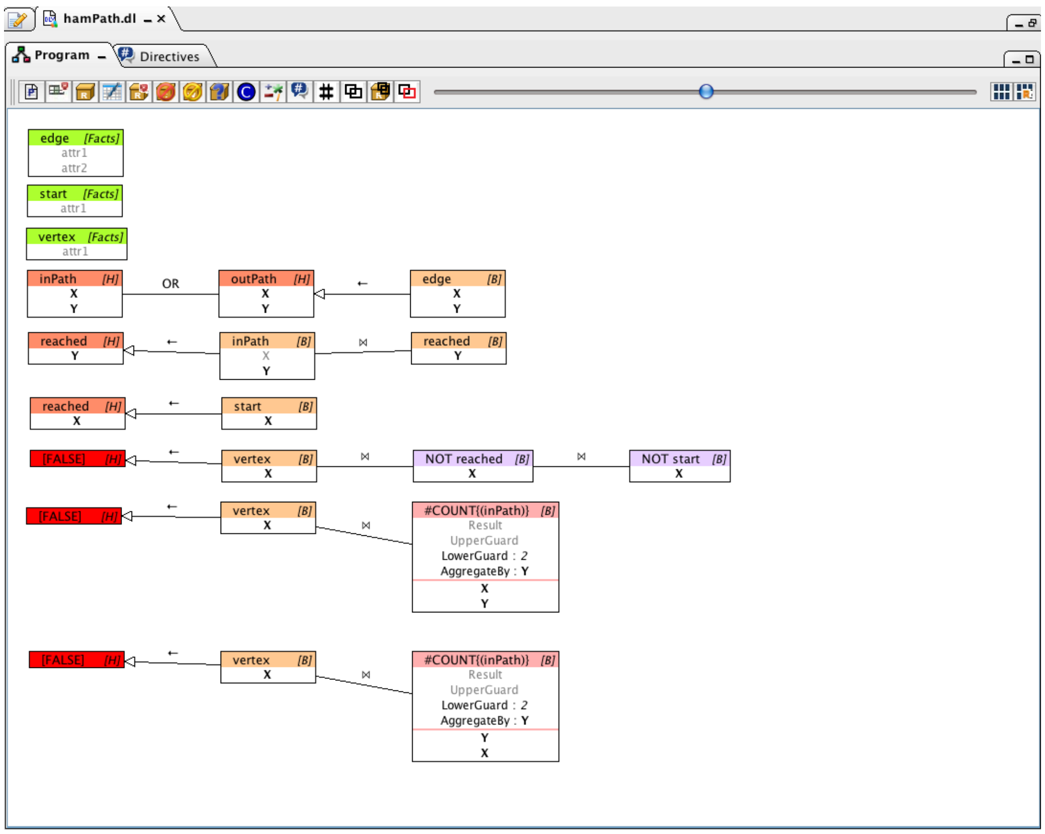
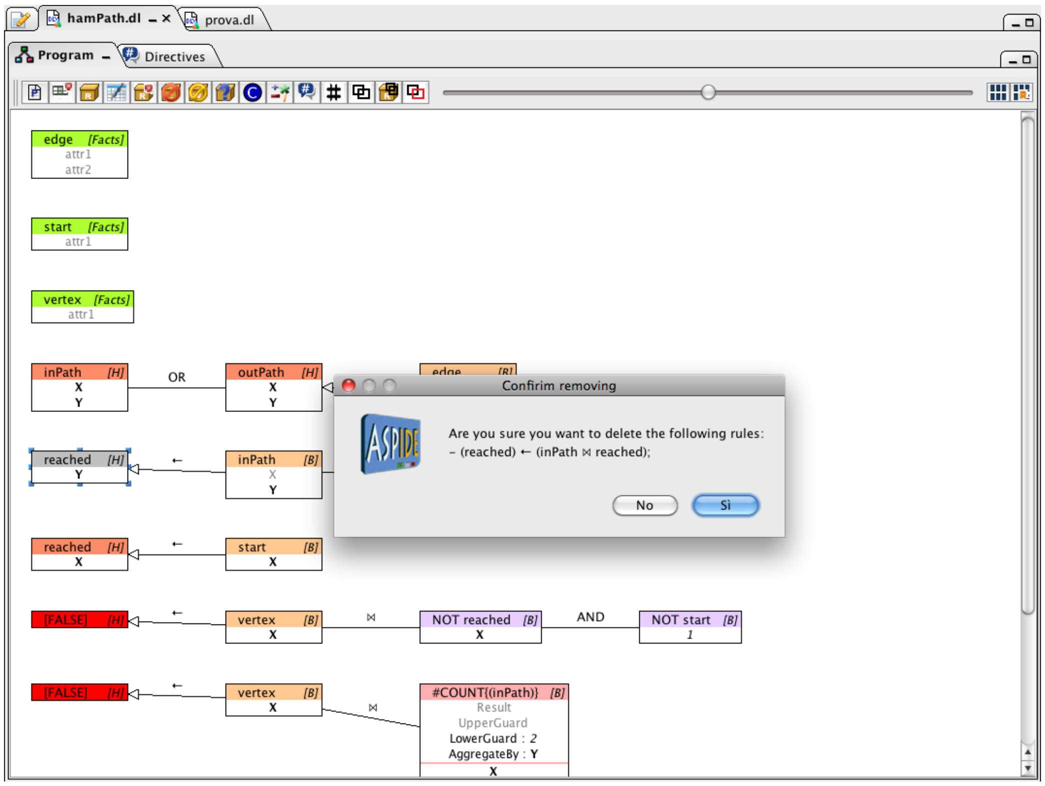


Figure 5.18: Program Hamiltonian Path.

Path problem is reported in Figure 5.18.

There are many other functionality and alternative ways of creating rules in *New Visual ASP*. In the following we describe the ones that we did not exploit in the example presented above namely:

- The **Remove** button in the toolbar can be used to remove one or more literals from a rule. The same operation is applicable both by selecting the literal to remove and by clicking on the icon of a red minus.
- The **Remove Rule** button of the toolbar allows to remove the entire rule that is selected in the main window. The same command can be acted by the drop-down menu that can be activated by right clicking on the selected rule (see see Figure 5.19).
- The **New Weak Constraint** adds a new weak constraint. Contrary to strong constraints, weak constraints allow for expressing conditions that should be satisfied, but not necessarily have to be. Associated to the weak constraint there is a weight and a priority level. A new weak constraint is visualized in Figure 5.20. The head of the weak constraint contains the fields of *weight* and *level*, and the body is built as for normal rules.
- The **New Operation** button allows to introduce comparisons ($=$, $!=$, $\leq$, $\geq$,$<$, $>$) and arithmetic operations ($+$, $-$, $*$, $\div$) into the visual ASP program. To insert one of this operation we select a predicate and we choose the specific one. Into the program will be added a box labeled with the selected operator. An example is in Figure **??**. The same behavior can be obtained also via drop-down menu and trough dedicated icons shown when a body literal is selected and the mouse is over its box.
- The **New Directive** button allows to set some directives offered by DLV, such as to set the *maxint* value or import tables from external databases. Suppose that, in our example of the *Hamiltonian Path*, instead of using facts for the predicate *edge* we want to import values contained in some external database table named *edge*, we select *Import* from the menu *New Directive*. A new *Import* directive will be shown where we can specify our directive parameters (see Figure 5.22).
- The **Collapse** (**Collapse Rule**) button allows to collapse two or more entities (respectively, one or more rules, each one will be collapsed separately).



Confirim removing

Are you sure you want to delete the following rules:
– (reached) ← (inPath ⋈ reached);

Figure 5.19: Remove rule.

Only the head of a collapsed rule is visible so to gain space in the drawing window. We could also compact the body of a rule by collapsing some of the literals in the body. We collapsed the two literals of a reachability encoding in Figure 5.23. An other possibility to Collapse two or more entities is to use the icon around one of the selected entities, or to use the right mouse button and selecting *Collapse*.

- The **Expand** button allows to expand collapsed specifications.

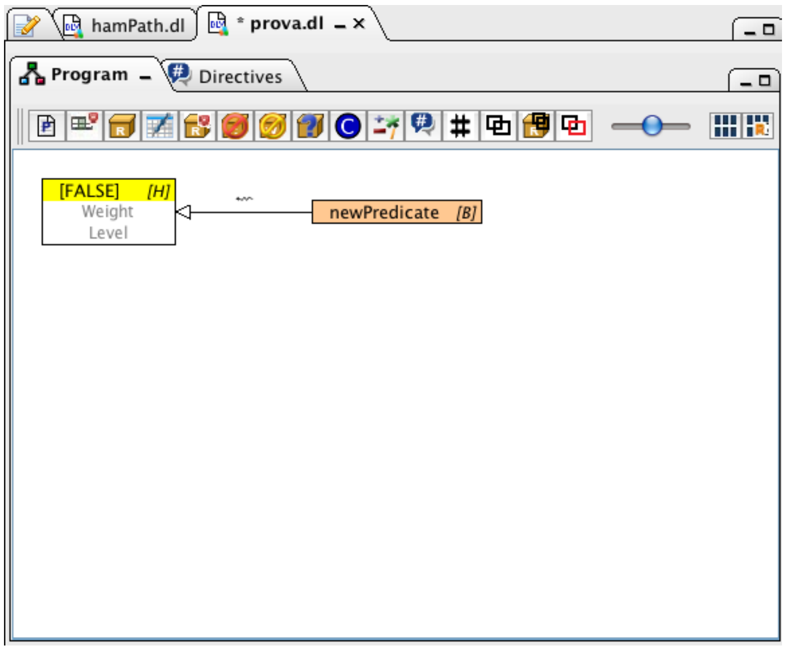


Figure 5.20: Weak constraint.

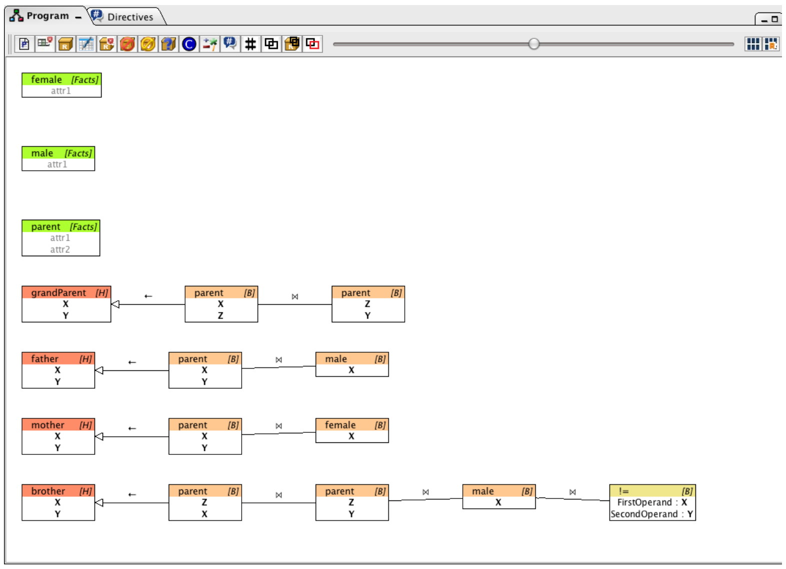

Program
Directives
female [Facts]
attr1
male [Facts]
attr1
parent [Facts]
attr1
attr2
grandParent [H] X Y ← parent [B] X Z ⋈ parent [B] Z Y
father [H] X Y ← parent [B] X Y ⋈ male [B] X
mother [H] X Y ← parent [B] X Y ⋈ female [B] X
brother [H] X Y ← parent [B] Z X ⋈ parent [B] Z Y ⋈ male [B] X ⋈ != [B]
FirstOperand : X
SecondOperand : Y

Figure 5.21: Rule with a comparison operator.

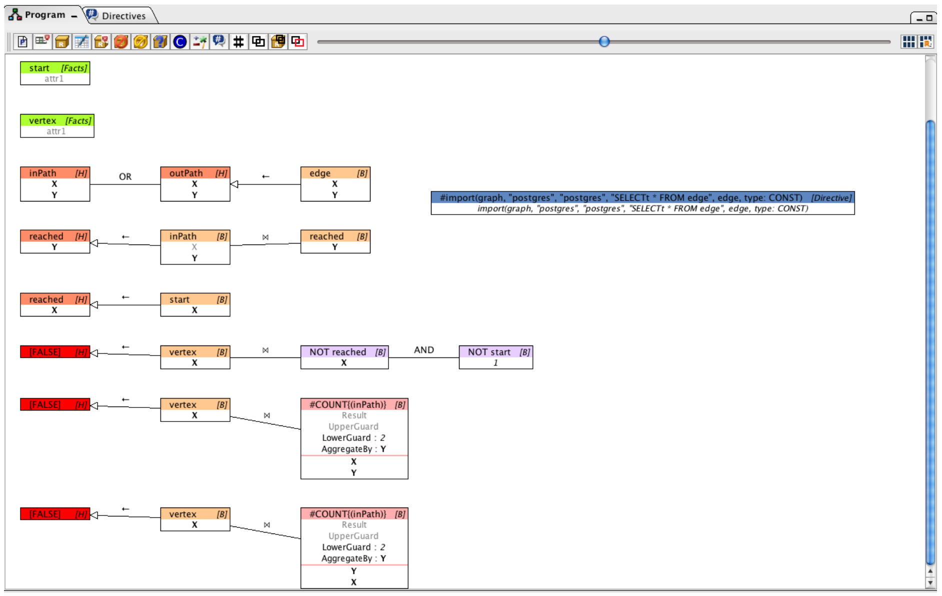


Figure 5.22: Import directive.

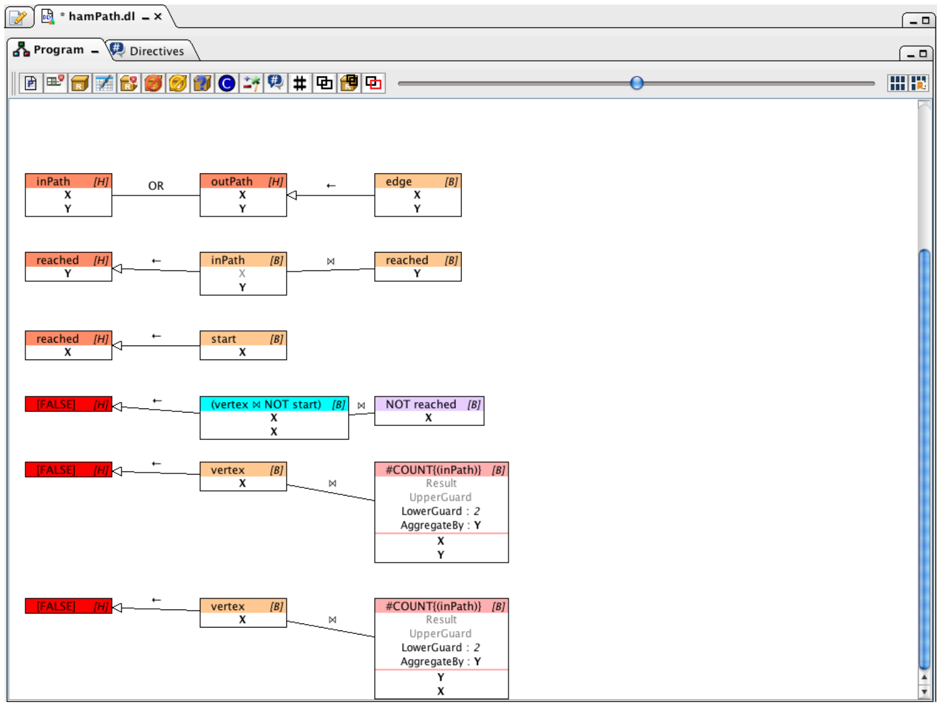


Figure 5.23: Effect of Collapse.

5.2 Integration of *ASPIDE* and *protégé*

Ontology-based reasoning is considered a crucial task in the area of knowledge management [12, 16]. New Semantic Web repositories are continuously built either from scratch or by translation of existing data in ontological form and are made publicly available. These repositories are often encoded by using W3C [77] standard languages like RDF(S), and OWL, and query answering on such repositories can be carried out with specific reasoners, supporting SPARQL as the query language.

In this context, the interest in approaches that resort to logic programming (mainly Datalog and its extensions) for implementing various reasoning tasks over ontologies is growing. Consider for instance that recent studies have identified large classes of queries over ontologies that can be Datalog-rewritable (see [42] for an overview) or First-Order Rewritable [17]. Approaches dealing with such fragments usually rely on query reformulation, where the original query posed on the ontology is rewritten into an equivalent set of rules/queries that can be evaluated directly on the ontology instances. Many query rewriters that are based on this idea exist [22, 1, 57, 71, 73] producing SQL queries or stratified programs. Moreover, even considering a setting where SPARQL queries are posed on RDF repositories, translations to Datalog with negation as failure were proposed [59] and implemented [43].

However, if we look at this scenario from a developer point of view, one can notice that different families of tools are required. On the one hand, one needs a good environment for designing and editing ontologies. On the other hand one would like to design, execute and test Datalog programs for ontology reasoning. Unluckily specific tools for these tasks are currently developed independently and miss a common perspective. We face with this issue proposing the integration of two major development environments for ASP programs and Ontology editing, respectively: *ASPIDE* [32] and *protégé* [76].

Protégé being one of the most diffused environments for the design and the exploitation of ontologies; and *ASPIDE* being the most comprehensive IDE for ASP extended with non monotonic negation and disjunction under the stable model semantics [39].

We extended both systems with specific plugins that enable a synergic interaction between the two development environments. The developer can then handle both ontologies and logic-based reasoning over them by exploiting specific tools integrated to work together. Note that, our solution has to be considered as a first

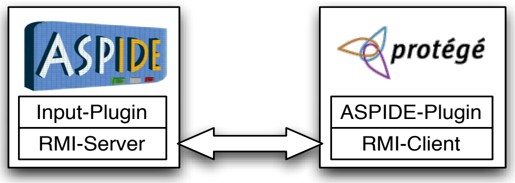


Figure 5.24: Integration of *ASPIDE* and *protégé*.

step towards the development of a general platform, which can be personalized and extended (also with the help of the research community) by integrating additional rewriters/reasoners. The aim is to provide an environment for developing, running and testing ASP-based ontology reasoning tools and their applications.

5.2.1 Integrating *ASPIDE* and *protégé*

The integration between *ASPIDE* and *protégé* is obtained by developing two separate plugins respectively extending these systems, see Figure 5.24. Both plugins are developed according to the following principle: simple modifications to ontologies and logic programs should be possible in both environments, but the user can switch to the most specific editor seamlessly.

As far as *ASPIDE* is concerned, we developed an input plugin [30] that recognizes and takes care of the ontology file types.The *ASPIDE* plugin offers two editing modalities for ontology files, which can be selected by clicking on the standard “switch button” of *ASPIDE*. The first modality opens a simple text editor embedded in *ASPIDE*, the second modality automatically opens the selected file in *protégé*. The plugin can also associate ontology files with some specific query rewriter, which is available in *ASPIDE* in the form of a rewriting plugin. Clearly new rewriters can be added to the system by developing additional rewriting plugins. *ASPIDE* is equipped with run configurations. A run configuration allows to setup a Datalog engine with its invocation options, to select input files and to (possibly) specify the associated rewriters.

Concerning the *protégé* side, we developed a plugin for *protégé* that displays the currently open *ASPIDE* workspace[2] in the usual tree-like structure. The idea is that the *ASPIDE* workspace acts as a common repository for Datalog programs and ontologies. The user can browse, add, remove or modify files in the workspace from *protégé*. Datalog programs can be modified in *protégé* by using a simple text editor in the plugin panel, whereas ontology files in the workspace are open and

[2]The workspace of *ASPIDE* is a directory collecting programs and files organized in projects.

displayed in *protégé* as usual. The user can also: require to open specific files in *ASPIDE*, execute one of the available rewriters, and invoke a Datalog reasoning engine by setting up and executing an *ASPIDE* run configuration.

The two plugins are connected each other via Java RMI. In particular, the *ASPIDE* plugin acts as a server. It publishes a remote interface that allows the *protégé* plugin to access the *ASPIDE* workspace and to require the execution of available commands. The same remote interface is exploited by *ASPIDE* to open ontologies in *protégé*.

5.2.2 Some use cases

In this section we consider some use cases possibly involving three kind of users interacting with our platform, namely:

(i) an ontology engineer,

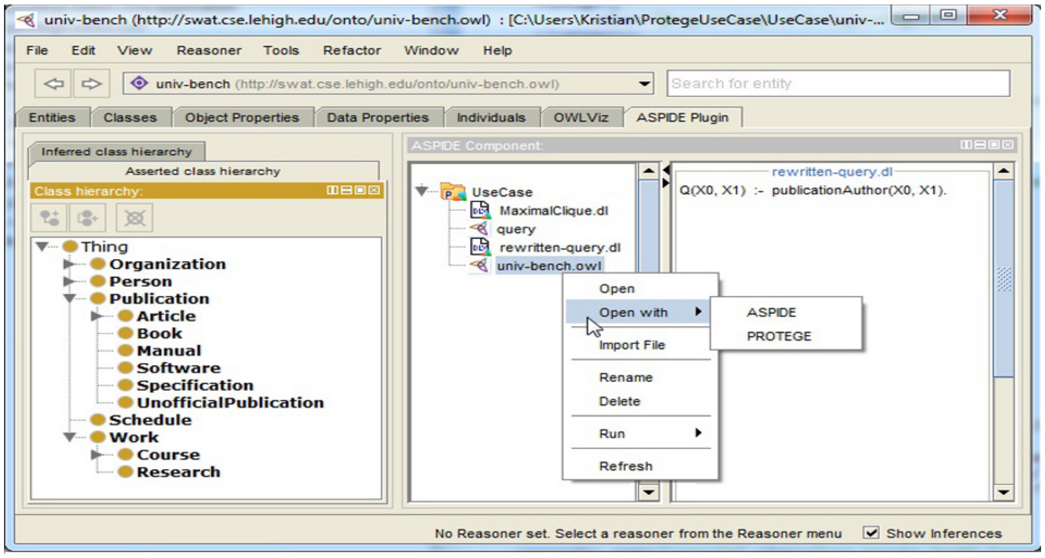
(ii) an engineer of query rewritings for ontologies,

(iii) a Datalog specialist using ontologies for reasoning tasks.

In our description we refer to the well known Datalog-based rewriters, Requiem [57] and Presto [71]. The first has been already integrated in our platform as a rewriting plugins of *ASPIDE*, and we are already working to integrate also Presto. Moreover, for the sake of presentation, we refer to some instantiation of the well known LUBM ontology. First of all, consider an ontology engineer, whose main objective is the design/update of ontologies. In this case the user starts his session from *protégé* by opening the ontology and modifying it with standard *protégé* tools. Then, in order to check the result of some reasoning task on the ontology, to be carried out with Requiem or Presto, the *ASPIDE* plugin can be opened inside *protégé* and used to select a run configuration choosing the desired rewriter and Datalog engine. The plugin allows also to inspect the produced rewriting before/after the execution, as well as the query result in the output panel of *ASPIDE*.

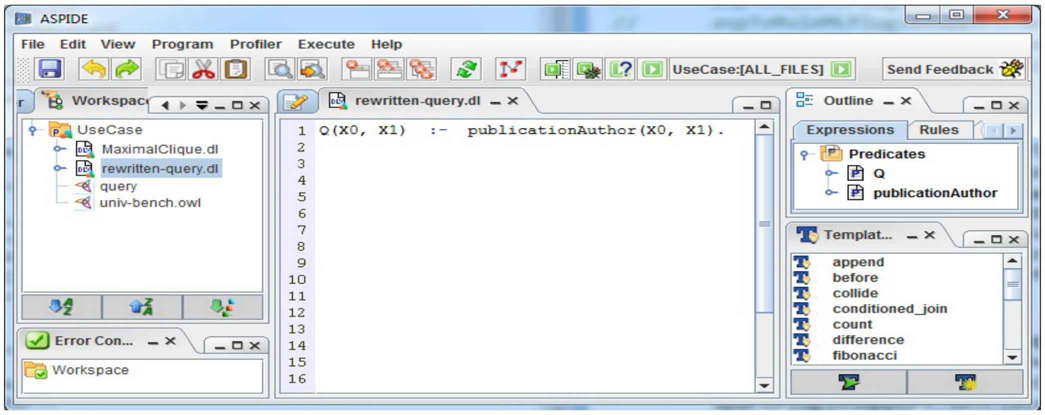
Figure 5.25(a) shows LUBM opened in *protégé* and a Requiem query rewriting shown in the ASPIDE panel inside *protégé*.

Let us now consider a rewriting engineer, whose main objective is the design/improvement of Datalog-based rewritings for ontology reasoning. In this case, his session can start either from *protégé* or from *ASPIDE*. In fact, the *ASPIDE* plugin for *protégé* allows to open ontology files directly in *ASPIDE*, in order to perform



```
Q(X0, X1) :- publicationAuthor(X0, X1).
```

(a) ASPIDE plugin for Protégé.



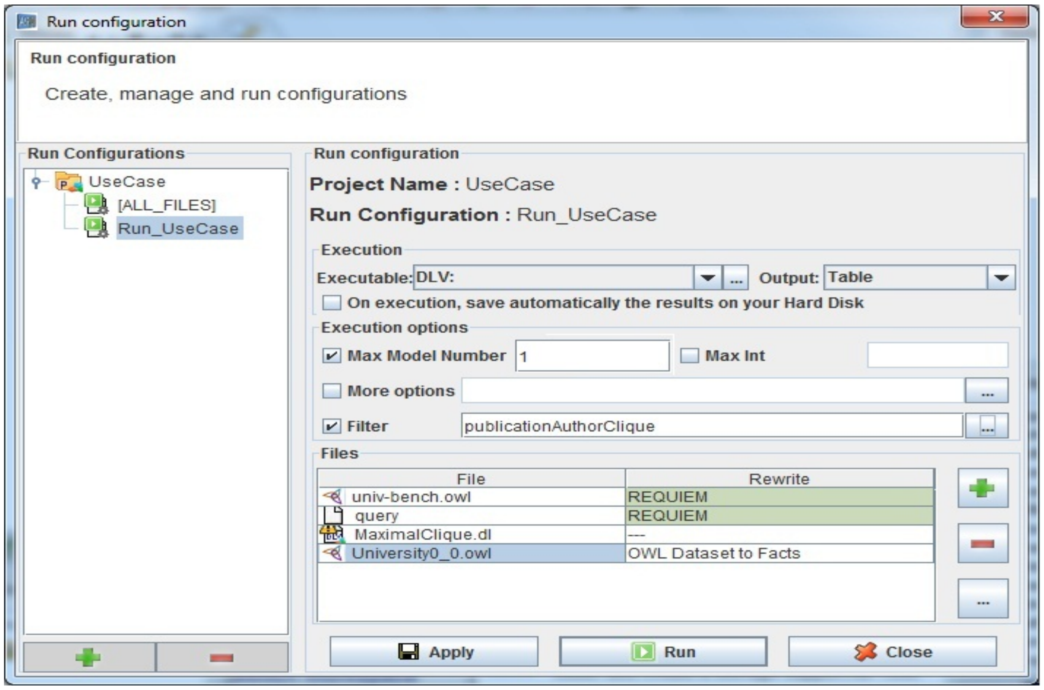
```
Q(X0, X1) :- publicationAuthor(X0, X1).
```

(b) Protégé input plugin for ASPIDE, and Requiem rewriting.

Figure 5.25: *ASPIDE* and *protégé* plugins at work.

basic inspection/modification tasks. Now, provided that an *ASPIDE* rewriting plugin is available, the user can activate the corresponding algorithm, inspect the resulting Datalog program, possibly modify it and run the Datalog engine to check the result. As a simple example, one could be interested in studying performance improvements possibly obtained on query answering by the application of magic-sets or unfolding strategies[3] applied in cascade, as a post-processing step, to a Requiem output program. This kind of analysis can be easily carried out with the proposed platform by properly setting two run configurations, one including and one excluding the post-processing step. As another example the rewriting en-

[3]These are standard query optimization strategies for Datalog programs.



Run configuration

Run configuration

Create, manage and run configurations

Run Configurations

- UseCase
  - [ALL_FILES]
  - Run_UseCase

Run configuration

Project Name : UseCase

Run Configuration : Run_UseCase

Execution

Executable: DLV: — Output: Table

☐ On execution, save automatically the results on your Hard Disk

Execution options

☑ Max Model Number: 1 — ☐ Max Int

☐ More options

☑ Filter: publicationAuthorClique

Files

File	Rewrite
univ-bench.owl	REQUIEM
query	REQUIEM
MaximalClique.dl	---
University0_0.owl	OWL Dataset to Facts

Figure 5.26: Configuring the execution of query and ontology rewritings.

gineer can be interested in checking the correctness and the performance of his brand new rewriting w.r.t. existing ones; again, this can be simply done by setting up different run configurations working on the same ontology.

Figure 5.25(b) illustrates the *ASPIDE* plugin at work on the LUBM ontology, with the Datalog program resulting from the activation of a rewriting plugin.

Finally, consider a Datalog specialist that needs to carry out complex reasoning tasks over ontologies. As an example, assume that the user is interested in identifying maximal cliques of coauthors in LUBM. LUBM provides both the *Person* and *Publication* concepts and the *publicationAuthor* role, which specifies the Authors (i.e. *Persons*) of each *Publication*. Finding a maximal clique of Authors is something that can be easily expressed with disjunctive Datalog extended with weak constraints,[4] provided the output of an ontology reasoner querying the ontology. Observe that, without our platform, the user should first infer authors and co-authorship relations from LUBM, then he should translate obtained results in a Datalog compliant format, and finally he should run a maximal clique encoding based on Datalog. To the contrary, by using the *protégé* plugin inside *ASPIDE* and a single *ASPIDE* run configuration it is sufficient to specify: the input on-

[4]We refer the reader to [4, 45] for more details on these extensions of Datalog.

tology file, the queries needed to infer data of interest, the rewriting plugin to activate, the program for computing maximal cliques, and the Datalog evaluation engine supporting needed language extensions. Figure 5.26 shows the run configuration implementing the above example, where the DLV system [45] is selected as Datalog engine.

Implementations Availability. The two plugins are available in beta version for *ASPIDE* v. 1.35 and *protégé* v. 4.3. They can be downloaded from the *ASPIDE* website `http://www.mat.unical.it/ricca/aspide`, and installed acting on the *ASPIDE* menu "File→Plugins→Manage Plugins".

5.3 Related Work

We presented a graphical interface for designing ASP programs, that is able to support all the powerful language constructs of ASP, like disjunction, recursion, unstratified negation, constraints, and aggregates. In the literature different formalisms were proposed that use a visual approach to logic programming. The articles [64, 2] describe a visual logic programming language based on a topological diagrammatic notation which combines Venn/Euler-like diagrams and DAGs (directed acyclic graphs). This formalism allows to represent, by basic syntactic elements (square boxes, rounded boxes, circles, arrows, lines) the constructs used in logic programming, including function symbols. Comparing the approach of [64, 2] with the one proposed in this paper we note that the latter does not directly support aggregates that are widely used in real applications, so this visual language can serve only as a source of inspiration for developing a visual language for ASP. The ASPIDE[34] environment for ASP also features a graphical interface for designing ASP programs, that is able to support all the powerful language constructs of ASP, like disjunction, recursion, unstratified negation, constraints, and aggregates. Nonetheless, this interface is not based on a formally-defined visual syntax, rather it disperses the definition of rules in several panels, the central one contains a graphical view of the body of a single rule. This interface was inspired by QBE interfaces, and the body of a rule is seen as a conjunctive query. The ASPIDE visual interface thus recalls the very well known and widely adopted QBE formalism for relational databases, which should be an advantage in familiarity for users that already uses this kind of visual languages. Nonetheless, the ASPIDE interface is not immediate and intuitive for non-expert users, since it does not provide a uniform view on rules and logic programs. We make this intuition more clear by providing direct comparison of the two editors in the same example. In that way we show the different representation of the same program into the two visual editor available for ASPIDE. Consider an encoding of the 3-COLORING problem: given an undirected graph $G = (V, E)$, assign each vertex one of three colors – say, red, green, or blue – such that adjacent vertices always have distinct colors. 3-COLORING can be encoded in ASP as follows:

node(v). arc(a,b). $\forall$v $\in V$ $\forall$(a,b) $\in A$

```
col(X,red) v col(X,blue) v col(X,green) :- node(X).
:- arc(X,Y), col(X,C), col(Y,C).
```

The first line asserts facts (these are rules with true body that is omitted) representing the input graph G, the second line states that each vertex needs to have some color. The last line contains a rule with false head that acts as an integrity constraint, since it disallows situations in which two connected vertexes are associated with the same color. The Figure 5.27 reports a snapshot of graphical representation into the new visual editor. In contrast, in the Figure 5.28, is depicted the same problem with the old ASPIDE editor. It is immediate to see that the new representation is more compact integrated and intuitive. Indeed, the *New Visual ASP* the complete logic program is visible and modifiable in the main window, is not the same in the *Visual ASP*. The shape of the rules is immediately recognizable, and there is a one to one mapping with the textual syntax. The new interface result also in a more friendly impact for programmers that are already expert of logic.

Concerning other systems that feature a full graphic tool for creating logic programs, we mention OntoDLV [69] and OntoStudio (*http://www.ontoprise.de*) that allow for specifying conjunctive queries and rules respectively and are strongly dependent on the features of the underlying logic-based ontology language; contrarily, *New Visual ASP* supports all the major language features of ASP, maintaining a clear separation between the rule editor and the ontology editor on the lines of [54]. Related works are also Visual Query Systems (VQS) and Visual Query Languages (VQL) that were developed in the Database Community [21]. Many graphical tool for querying Databases were presented. [6] proposes a taxonomy, based on expressive power, usability and classes of potential users, of Visual Query System. Nonetheless VQS systems were conceived for querying relational databases, thus they can only serve as a basis for devising a visual language as expressive as ASP.

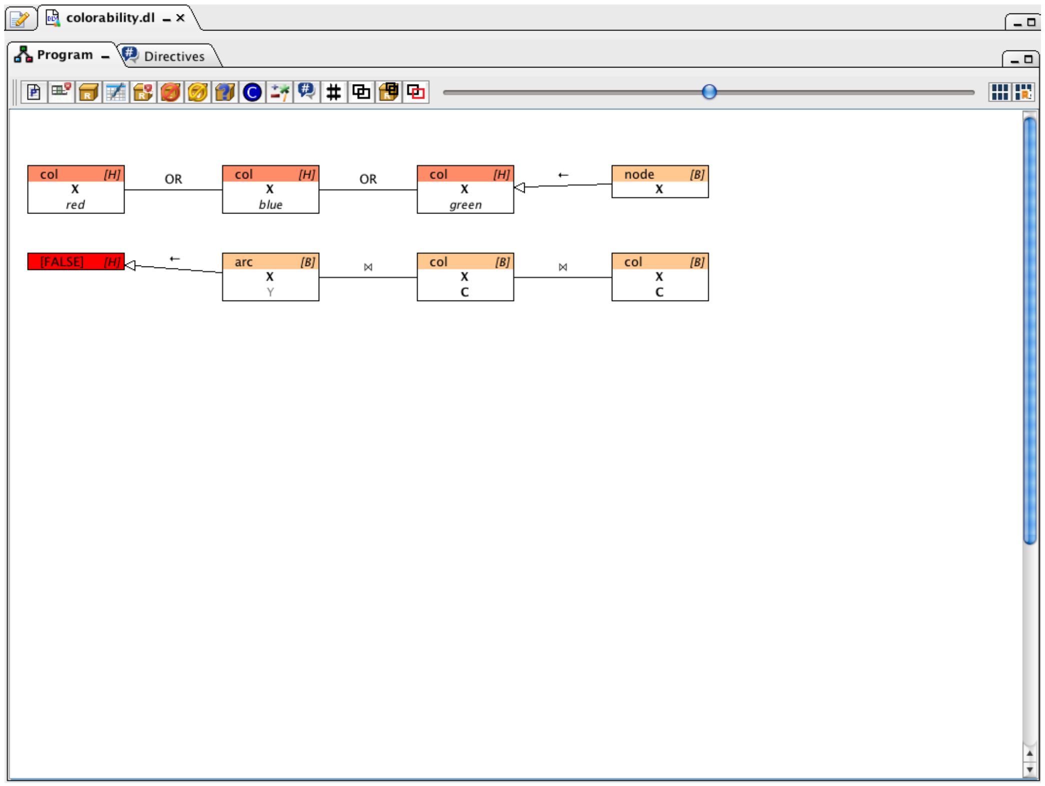
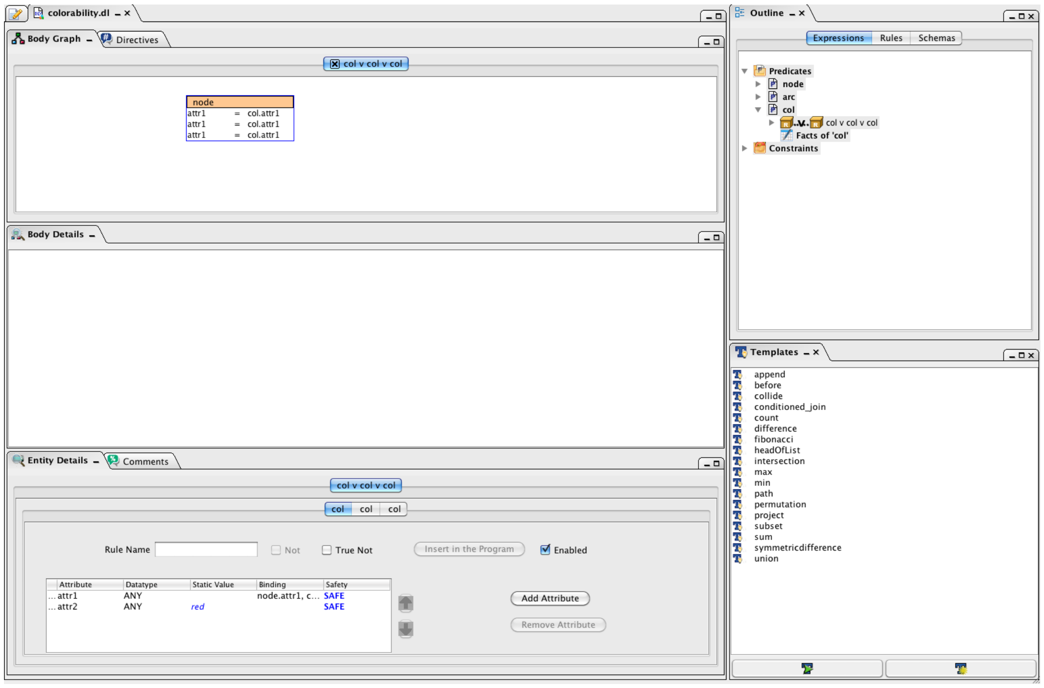


Figure 5.27: The new visual editor



Rule Name ☐ Not ☐ True Not ☑ Enabled

Attribute	Datatype	Static Value	Binding	Safety
...attr1	ANY		node.attr1, c...	SAFE
...attr2	ANY	red		SAFE

Figure 5.28: The old visual editor

Chapter 6

Conclusion

In the recent years ASP has been successfully applied for developing industrial applications. The development of applications confirmed the applicability of ASP-based technologies for solving complex real-world applications, moreover, application developers highlighted the need for more powerful programming tool to make ASP more effective and easy to use ASP in real-world development scenarios. This thesis provides several contributions in this context, and in particular is about:

(i) The development of two real-world applications of ASP in the domain of tourism;

(ii) The design and implementation of two new extensions of the ASPIDE IDE.

Concerning point (i), an application of ASP to the problem of allotment in travel industry is proposed, and an intelligent newsletter service for the customers of a travel agency is provided. Both problems were formalized in ASP by abstracting the requirements of a real travel agency. Since ASP programs are executable specifications we easily obtained using JDLV the implementations of a tool for supporting a travel agent in selecting the packages to be traded for next season, and of a newsletter service. The implementation were assessed on real-word data provided by the travel agency Top Class s.r.l. The preliminary results that were obtained are promising. The development of such applications has confirmed that ASP can be used for solving complex problems in practice, indeed, the ASP solutions described in this thesis will be included as an advanced reasoning service of the e-tourism platform developed under the iTravelPlus project by the Tour Operator Top Class s.r.l. and the University of Calabria.

Concerning point (ii), the thesis describes two new development tools that extend the well-known ASPIDE development environment: a visual programming environment and an environment for developing logic-programming-based ontology reasoning tools. The visual programming environment supports all the powerful language constructs of ASP, like disjunction, recursion, unstratified negation, constraints, aggregates, and weak constraints, and makes easier the writing of logic programs for novice programmers and targets those users prefer graphic tools. The ontology-orinted tool consists on an extension of ASPIDE and *Protégé* with specific plugins that enables a synergic interaction between the two development environments. The developer can then handle both ontologies and logic-based reasoning over them by exploiting specific tools integrated to work together. The presented solution moves the first step towards the development of a general environment for developing, running and testing logic-programming-based ontology reasoning tools.

The main contributions presented in this thesis have been published in the following research papers:

- Barbara Nardi, Kristian Reale, Francesco Ricca, Giorgio Terracina: *An Integrated Environment for Reasoning over Ontologies via Logic Programming*. Web Reasoning and Rule Systems - 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013. (LNCS – Vol. 7994 – Springer – Pg. 253-258).

- Barbara Nardi: *A Visual Syntax for Answer Set Programming*. Web Reasoning and Rule Systems - 8th International Conference, RR 2014, Athens, Greece, September 15-17, 2014. (LNCS – Vol. 8741 – Springer – Pg.249-250).

- Carmine Dodaro, Nicola Leone, Barbara Nardi, Francesco Ricca: *Allotment Problem in Travel Industry: A Solution Based on ASP.* Web Reasoning and Rule Systems - 9th International Conference, RR 2015, Berlin, Germany, August 4-5, 2015. (LNCS – Vol. 9209 – Springer – Pg. 77-92).

Bibliography

[1] Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. QUONTO: querying ontologies. In *Proc. of the 20th national conference on Artificial intelligence*, volume 4, pages 1670–1671. AAAI Press, 2005.

[2] Jaume Agust, Jordi Puigsegur, Dave Robertson, and W. Marco Schorlemmer. Visual logic programming through set inclusion and chaining. In *IN CADE-13 WORKSHOP ON VISUAL REASONING*, 1996.

[3] Marcello Balduccini, Michael Gelfond, Richard Watson, and Monica Nogeira. The USA-Advisor: A Case Study in Answer Set Planning. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *LNCS*, pages 439–442. Springer, 2001.

[4] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[5] Chitta Baral and Michael Gelfond. Reasoning Agents in Dynamic Domains. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.

[6] Carlo Batini, Tiziana Catarci, Maria Francesca Costabile, and Stefano Levialdi. Visual query systems: A taxonomy. In Elöd Knuth and Lutz Michael Wegner, editors, *VDB*, volume A-7 of *IFIP Transactions*, pages 153–168. North-Holland, 1991.

[7] Christian Bauer and Gavin King, editors. *Java Persistence with Hibernate*. Manning, 2006.

[8] Martin Brain and Marina De Vos. Debugging Logic Programs under the Answer Set Semantics. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation*, Bath, UK, July 2005.

[9] Martin Brain, Martin Gebser, Jorg Puhrer, Torsten Schaub, Hans Tompits, and Stefan Woltran. That is Illogical Captain! The Debugging Support Tool spock for Answer-Set Programs: System Description. In Marina De Vos and Torsten Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 71–85, 2007.

[10] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[11] Paula-Andra Busoniu, Johannes Oetsch, Jörg Pührer, Peter Skocovsky, and Hans Tompits. Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support. *TPLP*, 13(4-5):657–673, 2013.

[12] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 77–86, New York, NY, USA, 2009. ACM.

[13] Francesco Calimeri, Susanna Cozza, and Giovambattista Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50(3–4):333–361, 2007.

[14] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *TPLP*, 14(1):117–135, 2014.

[15] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *TPLP*, 14(1):117–135, 2014.

[16] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[17] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. *Artif. Intell.*, 195:335–360, 2013.

[18] Jorge Cardoso. Combining the semantic web with dynamic packaging systems. In *AIKED'06*, pages 133–138, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[19] Jorge Cardoso. Developing an owl ontology for e-tourism. In *Semantic Web Services, Processes and Applications*, pages 247–282. Springer, 2006.

[20] Massimiliano Castellani and Maurizio Mussoni. An economic analysis of tourism contracts: Allotment and free sale*. In *Advances in Modern Tourism Research*, pages 51–85. Springer, 2007.

[21] Tiziana Catarci, Maria Francesca Costabile, Stefano Levialdi, and Carlo Batini. Visual query systems for databases: A survey. *J. Vis. Lang. Comput.*, 8(2):215–260, 1997.

[22] Alexandros Chortaras, Despoina Trivela, and Giorgos B. Stamou. Optimized query rewriting for owl 2 ql. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803, pages 192–206. Springer, 2011.

[23] Chris Cooper, John Fletcher, Alan Fyall, David Gilbert, and Stephen Wanhill. *Tourism: Principles and Practice*. Financial Times Management, 4 pap/pas edition.

[24] A. Dogac, Y. Kabak, G. Laleci, S. Sinir, A. Yildiz, S. Kirbas, and Y. Gurcan. Semantically enriched web services for the travel industry. *SIGMOD Rec.*, 33(3):21–27, 2004.

[25] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[26] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[27] Omar El-Khatib, Enrico Pontelli, and Tran Cao Son. Justification and debugging of answer set programs in ASP. In Clinton Jeffery, Jong-Deok

Choi, and Raimondas Lencevicius, editors, *Proceedings of the Sixth International Workshop on Automated Debugging*, California, USA, September 2005. ACM.

[28] Onofrio Febbraro, iGiovanni Grasso, Nicola Leone, and Francesco Ricca. JASP: a framework for integrating Answer Set Programming with Java. In *Proc. of KR2012*. AAAI Press, 2012.

[29] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Unit testing in aspide. *CoRR*, abs/1108.5434, 2011.

[30] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Extending aspide with user-defined plugins. In *CILC*, volume 857, pages 236–240. CEUR-WS.org, 2012.

[31] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. A Visual Interface for Drawing ASP Programs. In *Proc. of CILC2010*, Rende(CS), Italy, July 2010.

[32] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. Aspide: Integrated development environment for answer set programming. In *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 317–330, 2011.

[33] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: integrated development environment for answer set programming. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, pages 317–330, 2011.

[34] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. Aspide: Integrated development environment for answer set programming. In *LPNMR*, pages 317–330, 2011.

[35] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[36] Lorenzo Gallucci and Francesco Ricca. Visual querying and application programming interface for an ASP-based ontology language. In *Proc. of SEA 2007*, pages 56–70, 2007.

[37] Alfredo Garro, Luigi Palopoli, and Francesco Ricca. Exploiting agents in e-learning and skills management context. *AI Communications – The European Journal on Artificial Intelligence*, 19(2):137–154, 2006.

[38] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[39] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[40] Giovanni Grasso, Nicola Leone, Marco Manna, and Francesco Ricca. *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, volume 6565 of *Lecture Notes in AI (LNAI)*. Springer Verlag, 2011.

[41] Tugba Gurcaylilar-Yenidogan, Alp Yenidogan, and Josef Windspergerc. Antecedents of contractual completeness: the case of tour operator-hotel allotment contracts. *Procedia - Social and Behavioral Sciences*, 24(0):1036 – 1048, 2011. The Proceedings of 7th International Strategic Management Conference.

[42] Stijn Heymans, Thomas Eiter, and Guohui Xiao. Tractable reasoning with dl-programs over datalog-rewritable description logics. In *Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 35–40, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

[43] Giovambattista Ianni, Thomas Krennwallner, Alessandra Martello, and Axel Polleres. A rule system for querying persistent rdfs data. In *ESWC*, volume 5554 of *LNCS*, pages 857–862. Springer, 2009.

[44] Bullock Joe and Goble Carole. Tourist: the application of a description logic based semantic hypermedia system for tourism. In *HYPERTEXT '98: Proceedings of the ninth ACM conference on Hypertext and hypermedia*, pages 132–141. ACM, 1998.

[45] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 7(3):499–562, 2006.

[46] Nicola Leone and Francesco Ricca. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In *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.

[47] N. Leone et al. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proc. of SIGMOD'05*, pages 915–917, New York, NY, USA, 2005. ACM.

[48] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[49] Alexander Maedche and Steffen Staab. Applying semantic web technologies for tourism information systems. In Karl Weber, Andrew Frew, and Martin Hitz (eds.), editors, *"Proceedings of the 9th International Conference for Information and Communication Technologies in Tourism ENTER 2002 Innsbruck Austria"*. Springer, 2002.

[50] Marco Manna, Francesco Ricca, and Giorgio Terracina. Consistent query answering via ASP from different perspectives: Theory and practice. *Theory and Practice of Logic Programming*, 13(2):277–252, 2013.

[51] V. Wiktor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032, 1998.

[52] Hepp Martin, Siorpaes Katharina, and Bachlechner Daniel. Towards the semantic web in e-tourism: can annotation do the trick? In *Proceedings of the 14th European Conference on Information System (ECIS 2006)*, 2006.

[53] Hepp Martin, Siorpaes Katharina, and Bachlechner Daniel. Towards the semantic web in e-tourism: Lack of semantics or lack of content? In *Poster Proceedings of the 3rd Annual ESWC (2006)*, 2006.

[54] Barbara Nardi, Kristian Reale, Francesco Ricca, and Giorgio Terracina. An integrated environment for reasoning over ontologies via logic programming. In *RR*, pages 253–258, 2013.

[55] Ilkka Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In Ilkka Niemelä and Torsten Schaub, editors, *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, Trento, Italy, May/June 1998.

[56] Oracle. JSR 317: JavaTM Persistence 2.0, 2009. `http://jcp.org/en/jsr/detail?id=317`.

[57] H. Pérez-Urbina, B. Motik, and I. Horrocks. A comparison of query rewriting techniques for dl-lite. In *Proceedings of the 22st International Workshop on Description Logics*, volume 477 of *DL '09*. CEUR-WS.org, 2009.

[58] Simona Perri, Francesco Ricca, Giorgio Terracina, D. Cianni, and P. Veltri. An integrated graphic tool for developing and testing DLV programs. In Marina De Vos and Torsten Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 86–100, 2007.

[59] Axel Polleres. From sparql to rules (and back). In *WWW*, pages 787–796. ACM, 2007.

[60] S. Polyviou and G. Samaras P. Evripidou. Query by Browsing: A Visual Query Language Based on the Relational Model and the Desktop User Interface Paradigm. University of Cyprus, Department of Computing, 2004.

[61] Katrine Prantner, Ying Ding, Michael Luger, Zhixian Yan, and Christoph Herzog. Tourism ontology and semantic management system: State-of-the-arts analysis. In *Proceedings of IADIS International Conference WWW/Internet 2007 Vila Real, Portugal, October (2007)*. IADIS, 2007.

[62] H. A. Proper. Interactive Query Formulation using Query by Navigation. *Asymetrix Research Report 94-4, Asymetrix Research Laboratory*, 1994.

[63] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[64] Jordi Puigsegur and Jaume Agust. Visual logic programming by means of diagram transformations. In *In Proc. of APPIA-GULP-PRODE Joint Conference in Declarative Programming, La Coru na*, pages 311–328, 1998.

[65] Francesco Ricca. A Java wrapper for DLV. In *Proc. of ASP 2003*, volume 78 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[66] Francesco Ricca. The DLV Java Wrapper. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 305–316, Messina, Italy, September 2003. Online at `http://CEUR-WS.org/Vol-78/`.

[67] Francesco Ricca, Lorenzo Gallucci, Roman Schindlauer, Tina Dell'Armi, Giovanni Grasso, and Nicola Leone. OntoDLV: An ASP-based system for enterprise ontologies. *J. Log. Comput.*, 19(4):643–670, 2009.

[68] Francesco Ricca, Giovanni Grasso, Mario Alviano, Marco Manna, Vincenzino Lio, Salvatore Iiritano, and Nicola Leone. Team-building with answer set programming in the gioia-tauro seaport. *TPLP*, 12(3):361–381, 2012.

[69] Francesco Ricca and Nicola Leone. Disjunctive Logic Programming with types and objects: The DLV$^+$ System. *Journal of Applied Logics*, 5(3):545–573, 2007.

[70] Francesco Ricci. Recommender systems: Models and techniques. In *Encyclopedia of Social Network Analysis and Mining*, pages 1511–1522. 2014.

[71] R. Rosati and A. Almatelli. Improving Query Answering over DL-Lite Ontologies. In *Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR 2010)*, KR '10, pages 290–300, Toronto, Ontario, Canada, 2010. AAAI Press.

[72] G. Santucci and P. A. Sottile. Query by Diagram: a Visual Environment for Querying Databases. Dipartimento di Informatica e Sistemistica, Universitá degli Studi di Roma $'$La Sapienza$'$, 1993.

[73] Markus Stocker and Michael Smith. Owlgres: A scalable owl reasoner. In Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, editors, *5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008)*, volume 432. CEUR-WS.org, 2008.

[74] Adrian Sureshkumar, Marina De Vos, Martin Brain, and John Fitch. APE: An AnsProlog* Environment. In Marina De Vos and Torsten Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 101–115, 2007.

[75] Giorgio Terracina, Nicola Leone, Vincenzino Lio, and Claudio Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming*, 8:129–165, 2008.

[76] Stanford University. The Protégé Ontology Editor and Knowledge Acquisition System, 2012. `http://protege.stanford.edu`.

[77] W3C. The World Wide Web Consortium, 2012. `http://www.w3.org/`.

[78] Josef Withalm, Eibel Karl, and Micheal Fasching. Agents solving strategic problems in tourism. In DanielR. Fesenmaier, Stefan Klein, and Dimitrios Buhalis, editors, *Information and Communication Technologies in Tourism 2000*, pages 275–282. Springer Vienna, 2000.

[79] D. Young and B. Shneiderman. A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation. *Human-Computer Interaction Laboratory & Department of Computer Science*, 1993.