UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

*Departamento de Computación*

Traballo de Fin de Grao en Enxeñaría Informática

# A Diagrammatic Reasoning Tool for Logic Programming

*Author:* Carlos Pérez Ramil

*Advisors:* José Pedro Cabalar Fernández

Gilberto Pérez Vega

A Coruña, June 20, 2017

# Especificación

|  |  |
|---|---|
| *Título*: | Unha Ferramenta de Razoamento Diagramático para Programación Lóxica |
| *Clase*: | Proxecto clásico de enxeñaría |
| *Autor*: | Carlos Pérez Ramil |
| *Director*: | José Pedro Cabalar Fernández<br>Gilberto Pérez Vega |
| *Tribunal*: |  |
| *Fecha de lectura*: |  |
| *Calificación*: |  |

# Agradecimientos

Quisiera antes de nada dar las gracias a todas las personas que han contribuido directamente a la realización de este proyecto.

A los desarrolladores de todas las herramientas software que aquí se utilizan, no sólo por su excelente trabajo, sino también por publicarlo bajo licencias libres que fomentan la cultura de la cooperación y protegen el interés común.

A los directores de este trabajo, Pedro Cabalar y Gilberto Pérez, por todo el tiempo, esfuerzo e interés que han depositado en él.

A Nuria, por su apoyo constante e incondicional durante todos estos meses, animándome a continuar frente a las adversidades y evitando que pierda de vista el horizonte.

Y sobre todo, a mis padres, por haberme enseñado a apreciar el valor de la ciencia y la cultura, por alimentar cada día mis ganas de aprender, y porque son ellos los que han hecho posible que hoy cuente con una educación de calidad.

# Abstract

In this project we will explore a new approach to problem modeling with Answer Set Programming (ASP), a declarative logic programming paradigm used in the field of Knowledge Representation. Unlike the usual symbolic representation of logic programs, we aim to develop a visual tool that allows the user to write actual ASP with diagrams instead of code. Our main inspiration will be the work of philosopher and mathematician Charles S. Peirce, who in the late 19th Century introduced a diagrammatic system equivalent to First-Order Logic.

## Keywords

Answer Set Programming, Knowledge Representation, Diagrammatic Reasoning, Existential Graphs, Equilibrium Graphs

# Contents

# List of Figures

# List of Algorithms

# Chapter 1

# Introduction

Knowledge Representation and Reasoning (KR) is a branch of Artificial Intelligence (AI) dedicated to study how information about the world can be represented in a form that a computer can use to solve complex tasks without human intervention. Common applications of KR are systems for computer-aided diagnosis, planning and constraint satisfaction. Answer Set Programming (ASP) [1, 2, 3] is a declarative logic programming paradigm widely used in this field, focused on solving difficult (primarily NP-hard) search problems. The usual workflow in ASP is to first write a formal specification of some problem through a logic program, and then use general problem solving tools known as "answer set solvers" to find the "stable models" of the given program, which correspond to the problem's solutions. For instance, a logic program could represent the relations and constraints of the Travelling Salesman problem for a given city-road layout, and each stable model would correspond to each of the possible routes that visits each city exactly once and ends again in the origin. The syntax of ASP is similar to Prolog: a logic program is a set of rules composed of predicates, variables and logical connectives.

Most efforts in Knowledge Representation (KR) have been traditionally focused on symbolic manipulation and, in particular, on logical formulation. The use of a formal representation is surely convenient for automated reasoning, since computer languages provide nowadays excellent tools for symbolic representation and processing. Unfortunately, something that is simpler for computational treatment is not always necessarily better for human understanding. Educational experiences show that learning logical notation takes some time and effort to novel students. Even for an experienced student, reading a formula that nests different quantifiers, connectives or parentheses may become a difficult task and lead to errors in formal specification.

One alternative to formal languages that is probably closer to human's intuition is the use of graphical or diagrammatic representations. Diagrammatic Reasoning is the act of finding relationships between concepts and ideas to obtain some conclusion, by means of visual representations instead of using linguistic or algebraic ones. Diagrammatic KR has been explored and used in different fields of Artificial Intelligence – a prominent example is, for instance, Sowa's *conceptual graphs* [4, 5]. But the use of diagrams for logical representation is older than KR and AI, and actually comes from the very origins of modern philosophical logic. As commented by Sowa in [6], the use of diagrams in logic was something common before the introduction of the current notation, conceived by Peano[1] in 1889 [7]. In fact, Frege's original formulation of Predicate Calculus already included some diagrammatic component. But it was Charles Sanders Peirce who first introduced[2] a full-blown non-symbolic system for first-order logic: Existential Graphs (EGs) [8]. This graphical system allows a complete characterisation of First-Order Logic in diagrammatic terms, without using logical connectives. However, save few exceptions (like their influence in Sowa's conceptual graphs [6]), the truth is that EGs did not gain the same popularity as the symbolic notation for classical logic, even though they provide an elegant and simple representation that seems very suitable for educational purposes. Perhaps one of the difficulties for their consolidation has to do with their strong dependence on classical logic. Existential graphs take conjunction, negation and existential quantifiers as primitive constructors, building all the rest (disjunction, implication or universal quantification) as derived operations. This approach leaves no room for other non-classical logics such as intuitionistic or intermediate logics, where we may need keeping all these connectives independently.

In this project, we want to combine the power of answer set solvers with Diagrammatic Reasoning. To do so, an extension of Existential Graphs will be used as an alternative diagrammatic notation for ASP and, in particular, for its logical formalization in terms of *Equilibrium Logic* [9]. Proposed by David Pearce, Equilibrium Logic has allowed the application of the *stable model* semantics [10], originally defined for the syntax of logic programs, to the case of arbitrary propositional formulas. Moreover, the extension to the first-order case, known as Quantified Equilibrium Logic (QEL) [11], provides nowadays a general logical notion of stable models for arbitrary theories expressed in the syntax of First-Order Logic. Equilibrium Logic is defined by imposing a model selection criterion on top of a monotonic intermediate logic known as the logic of *Here-and-There* [12] (HT). In this logic, implication is a primitive operation and,

[1]Peano's quantifiers $\exists, \forall$ correspond to the inverted letters E and A, whereas $\vee$ comes from Latin *vel* ("or") and conjunction $\wedge$ from its inversion.

[2]Peirce's first proposals of existential graphs date back to 1882, even earlier than Peano's publication of the modern symbolic notation.

although disjunction can be defined in terms of the former plus conjunction, its representation as a derived operator is rather cumbersome. Something similar happens in *Quantified HT* [11] where, again, the existential quantifier is definable in terms of the universal one, but it is much more convenient to treat both of them as primitive connectives.

The main focus of this project is educationanl. Mathematical Logic could be itroduced to young students in a more friendly manner. At first sight, a person not familiar with logical formulas is less reluctant towards diagrams than to traditional symbolic representations. Moreover, for a person who has never written a computer program, it seems more natural to start drawing with a user-friendly visual tool than to start coding textual ASP rules in a particular syntax that must be learned first. But this project could also be useful for experienced logic programmers who want to try a different approach to their daily tasks. After someone has spent several years with a particular workflow, it might be an enriching experience to switch from one to another for a moment, and see how different reasoning schemes affect effectiveness and efficiency. We want to let the programmer not only to think with diagrams (something that can actually be done without any software tool), but also to "encode" a problem using diagrams, so that a computer would be able to process what the programmer draws.

The main goal of this work is to provide a diagrammatic alternative to the traditional symbolic representation of ASP programs. This will be achieved by introducing Equilibrium Graphs (EqGs), a variation of Charles S. Peirce's Existential Graphs [8], and developing `Grasp`, a visual tool that allows the logic programmer to encode and solve problems in a diagrammatic fashion. As we want to cover the whole problem solving process, not only the formalization stage, `Grasp` must be integrated with state-of-the-art ASP solvers, and it must be able to present results also in diagrammatic form. A tool like this should completely hide the symbolic nature of the underlying solving process, and let users see problems as diagrams, as well as solutions.

This document is organized as follows:

- Chapter 2 provides an overview of the theoretical background needed for the development of this work. We will take a quick look at Answer Set Programming, Existential Graphs and Quantified Equilibrium Logic.

- Chapter 3 gives a thorough explanation of the system's design and implementation. First, the concept of Equilibrium Graphs is introduced in Section 3.1; then the transformations for translating a diagram into a logic program are discussed in Section 3.2; finally our software tool is analyzed step by step in Section 3.3.

- Chapter 4 presents other works related in some manner with this one.
- Chapter 5 collects some insights about the developed work and closes this document.

In addition, as Appendix content, we provide an Installation Guide, a User's Manual, a collection of UML diagrams, a set of problem examples and several file format samples.

# Chapter 2

# Theoretical Background

## 2.1 Answer Set Programming

In this chapter, we give a brief introduction to Answer Set Programming (ASP) along with some examples. ASP is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. ASP is based on the *stable model* (answer set) semantics of logic programming [10], with ideas from autoepistemic logic and default logic.

Search problems in ASP are reduced to compute the stable models of the program that encodes the problem. Answer Set solvers, programs for generating the *stable models* are used to perform the search.

The search algorithms used in the design of many answer set solvers are enhancements of the Davis-Putnam-Logemann-Loveland procedure, unlike the resolution algorithm employed in Prolog, the computation of the *stable models* of an ASP program, in principle, always terminates. Some lightweight introductions to ASP and its applications can be found in [13] and [3].

Nowadays the use of ASP has extended to many areas of application with different requirements. In order to address this issue, many syntactic constructions have been introduced, making the formalisation process more comfortable and easier to understand. Anyway, we will focus on the most basic features, rather than introducing new ones that would fall out of the scope of this project.

### 2.1.1 Basic syntax and semantics

The building blocks of Answer Set programs are atoms, literals and rules. An atom is an elementary proposition that may be either true or false. A literal is an atom $a$ and its negation *not a*. A rule is an expression of the form:

$$a \leftarrow b_1,\ b_2,\ \ldots,\ b_n,\ not\ c_1,\ not\ c_2,\ \ldots,\ not\ c_m.$$

where all $a$, $b$ and $c$ are atoms. The *head* of the rule is the part on the left of the arrow. The *body* of the rule is the conjunction of the atoms on the right of the arrow. A *rule* in ASP can be seen as the logical implication between the body of the rule and the head. Intuitively, we can "derive" that the literals on the head are true if every literal on the body is true in the following sense: a non-negated literal $b_i$ is true if it has a derivation; a negated one, $not\ c_j$ , is true if the atom $c_j$ does not have one.

For instance, the rule

$$go\_to\_the\_beach \leftarrow summer,\ holiday,\ not\ raining. \tag{2.1}$$

means that we will go to the beach if we are on summer holiday and it is not raining. More formally, *go_to_the_beach* will be true if we can derive *summer* and *holiday*, but not *raining*. Note that here the *not* operator does not correspond to a standard negation. Rather, it represents the impossibility to derive its operand (in this case, *raining*). Thus, in ASP one can obtain conclusions because of the absence of evidences, and not only because of their presence.

A rule whose body is empty is called a *fact*, and is unconditionally true as it does not depend on other atoms to be derived. The arrow is usually omitted.

$$summer. \tag{2.2}$$

$$holiday. \tag{2.3}$$

A rule whose head is empty is called a *constraint*, and satisfying the rule body would result in a contradiction, so constraints are used to specify situations that cannot happen, which in practice is to exclude some specific models from the answer set. For example, the rule

$$\leftarrow summer,\ winter. \tag{2.4}$$

says that a model where both *summer* and *winter* atoms are true cannot be included in the answer set.

A program in ASP is a finite collection of rules. The solution to an Answer Set program is the set of *stable models* of the program. We will provide a formal definition of *stable model* later, but in order to understand it intuitively, to compute an *stable model* consists of iteratively deriving atoms from a set of rules.

If a program has no *stable models* when solved, we say that the program is unsatisfiable. If it has at least one *stable model* (it can have more than one solution), then we say that the program is satisfiable.

Let us consider a program $P_1$ defined by 2.1, 2.2, 2.3 and 2.4. Its unique *stable model* is $\{$*summer, holiday, go_to_the_beach*$\}$, so it is satisfiable. If we add to $P_1$ the fact

$$winter. \tag{2.5}$$

it becomes unsatisfiable, because 2.4 now leads to a contradiction.

One important remark is that ASP is a non-monotonic reasoning system: if from a program $P$ we can derive a set of conclusions $C$, the addition of new rules to $P$ may lead to a retraction of some of the conclusions in $C$. As an example of this behavior, consider again the program $P_1$. If we add the following fact:

$$raining. \tag{2.6}$$

then the new stable model is $\{$*summer, holiday, raining*$\}$ and does not contain *go_to_the_beach*, as it can no longer be derived.

### 2.1.2 Variables

Until now we have assumed that Answer Set programs are sets of rules over propositional atoms. However, the real potential of ASP comes from the use of logical variables within the rules. A variable is a symbol that can be ranged over the domain of discourse in a program. Variables are a useful feature for representing knowledge in ASP.

However, the task of calculating the *stable models* of a program is defined to work only in programs with no variables. The process of converting a program into a variable free one is known as *grounding*, and it is done by replacing each variable on a program with every possible atom from the problem domain.

For instance, let $P_2$ be the program denoted by

$$p(a). \tag{2.7}$$

$$p(b). \tag{2.8}$$

$$q(b). \tag{2.9}$$

$$r(X,\ Y)\ \leftarrow\ p(X),\ q(Y). \tag{2.10}$$

By convention, atom names start with a lowercase letter, while variable names start with an uppercase.

After grounding $P_2$, we get the following variable free program:

$$
\begin{array}{cc}
 & r(a,\ a)\ \leftarrow\ p(a),\ q(a). \\
p(a). & \\
 & r(b,\ b)\ \leftarrow\ p(b),\ q(b). \\
p(b). & \\
 & r(a,\ b)\ \leftarrow\ p(a),\ q(b). \\
q(b). & \\
 & r(b,\ a)\ \leftarrow\ p(b),\ q(a).
\end{array}
$$

whose unique *stable model* is $\{$*p(a), p(b), q(b), r(b, b), r(a, b)*$\}$.

Finally, we introduce here an additional concept used in modern grounding software that will be needed later.

**Definition 1** (Rule safety)**.** *We say that an ASP rule is safe if every variable in it occurs in some of the rule's positive body literals (i.e., not prefixed with not) whose predicate is not "=" or any another built-in comparison predicate.*

A program is safe if every rule occurring in it is safe. State-of-the-art grounders like `gringo` require programs to be safe in order to guarantee termination.

### 2.1.3 Stable models

So far we have used the concept of *stable model* to identify the set of solutions to an Answer Set program, but we have not given any definition. In order to do so we reproduce the definition given by Gelfond and Lifschitz [10].

#### Definition

Let $P$ be a set of rules of the form

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$$

Where $a, b_1, \ldots, b_m, c_1, \ldots, c_n$ are ground atoms. If $P$ does not contain negation ($n = 0$ in every rule of the program) then, by definition, the only *stable model* of $P$ is the one which is minimal in relation to set inclusion (any program without negation has exactly one minimal model). To extend this definition to the case of programs with negation, we need the auxiliary concept of the reduct, defined below.

For any set I of ground atoms, the reduct of $P$ relative to I is the set of rules without negation obtained from $P$ by first dropping every rule such that at least one of the atoms $c_i$ in its body

$$b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$$

belongs to I, and then dropping the parts not $c_1, \ldots,$ not $c_n$ from the bodies of all remaining rules.

We say that I is a *stable model* of $P$ if I is the *stable model* of the reduct of $P$ relative to I. Since the reduct does not contain negation, its *stable model* has been already defined. As the term *stable model* suggests, every *stable model* of $P$ is a model of $P$.

**Example**

To illustrate these definitions, let us check that $\{p,\ s\}$ is a *stable model* of the program

$$p \tag{2.11}$$
$$r \leftarrow p,\ q \tag{2.12}$$
$$s \leftarrow p,\ \text{not } q. \tag{2.13}$$

The reduct of this program relative to {p, s} is

$$p \tag{2.14}$$
$$r \leftarrow p,\ q \tag{2.15}$$
$$s \leftarrow p. \tag{2.16}$$

Indeed, since $q \notin \{p,\ s\}$, the reduct is obtained from the program by dropping the part not $q$. The *stable model* of the reduct is $\{p,\ s\}$. This set of atoms satisfies every rule of the reduct, and it has no proper subsets with the same property. Thus after computing the *stable model* of the reduct we arrived at the same set $\{p,\ s\}$ as the one we started with. Consequently, that set is a *stable model.*

Checking in the same way the other 15 sets consisting of the atoms $p$, $q$, $r$, $s$ shows that this program has no other *stable models*. For instance, the reduct of the program relative to {p, q, r} is:

$$p \tag{2.17}$$
$$r \leftarrow p,\ q. \tag{2.18}$$

The *stable model* of the reduct is $\{p\}$, which is different from the set $\{p,\ q,\ r\}$ that we started with.

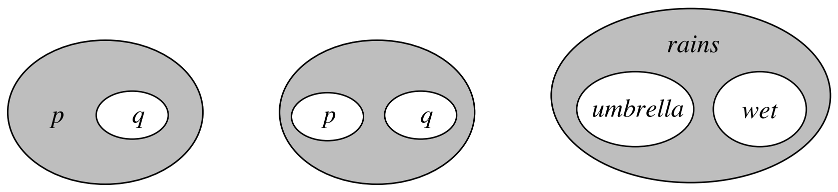
## 2.2 Existential Graphs

### 2.2.1 Alpha System

We recall next the essential components of existential graphs. Peirce classified EGs into three types, *alpha*, *beta* and *gamma*, that respectively correspond to Propositional Calculus, First-Order Logic with equality and (a kind of) normal modal logic. We start defining alpha graphs as follows. A *diagram* in alpha graphs is recursively defined as one of the following:

- the main page (when empty, it represents truth)
- atomic propositions
- a region encircled by a closed curve (called *cut*), which denotes the negation of the subdiagram inside the region. An empty cut represents falsity.
- finally, although it is not a drawing in itself, the inclusion of several elements inside the same region or cut (including the full page) is implicitly understood as their conjunction

As an example, Fig. 2.1(c) explicitly represents the formula $\neg(\mathit{rains} \wedge \neg \mathit{umbrella} \wedge \neg \mathit{wet})$ which can also be seen as the implications $\mathit{rains} \wedge \neg \mathit{umbrella} \rightarrow \mathit{wet}$ or $\mathit{rains} \wedge \neg \mathit{wet} \rightarrow \mathit{umbrella}$ or the disjunction $\neg \mathit{rains} \vee \mathit{umbrella} \vee \mathit{wet}$, since all these representations are equivalent in classical propositional logic. Using conjunction and negation as primitive operators, we can easily represent an implication $p \rightarrow q$ as $\neg(p \wedge \neg q)$ (Fig. 2.1(a)) and a disjunction $p \vee q$ as $\neg(\neg p \wedge \neg q)$ (Fig. 2.1(b)). Another common feature shown in these examples is that areas encircled by an odd number of cuts (negative areas) are sometimes shaded to facilitate the visualisation.



(a) $p \rightarrow q$ (b) $p \vee q$ (c) $\mathit{rains} \wedge \neg \mathit{umbrella} \rightarrow \mathit{wet}$

Figure 2.1: Some alpha graphs.

The alpha system was accompanied by a set of inference and equivalence (diagram redrawing) rules that was proved to be sound and complete with respect to Propositional Calculus. In this preliminary proposal, we will focus on the representation itself, leaving graphical inference in the logic of HT or even in (non-monotonic) Equilibrium Logic for a future study.

## 2.2.2 Beta System

For representing first-order expressions, Peirce extended alpha graphs to *beta* graphs by the inclusion of a new type of component in the diagram, *lines of identity.* A line of identity is an open line that connects one or more atom names. When it is used to connect more than two atom names, the identity line may bifurcate as many times as needed, getting the shape of a tree or a spider with several ramifications. The reading for an identity line is an existential quantifier: "there exists some individual such that ..." Figure 2.2 shows several examples. Fig. 2.2(a) asserts that there is a red car parked at a street: $\exists x \exists y(car(x) \wedge red(x) \wedge parkedAt(x, y) \wedge street(y))$. Fig. 2.2(b) means that there is some person that loves herself, $\exists x(person(x) \wedge loves(x, x))$. Fig. 2.2(c) says that every man is mortal, $\neg\exists x(man(x) \wedge \neg mortal(x))$ or, if preferred, $\forall x(man(x) \rightarrow mortal(x))$. Finally, Fig. 2.2(d) specifies that there is a woman adored by every catholic: $\exists x(woman(x) \wedge \forall y(catholic(y) \rightarrow adores(y, x)))$.

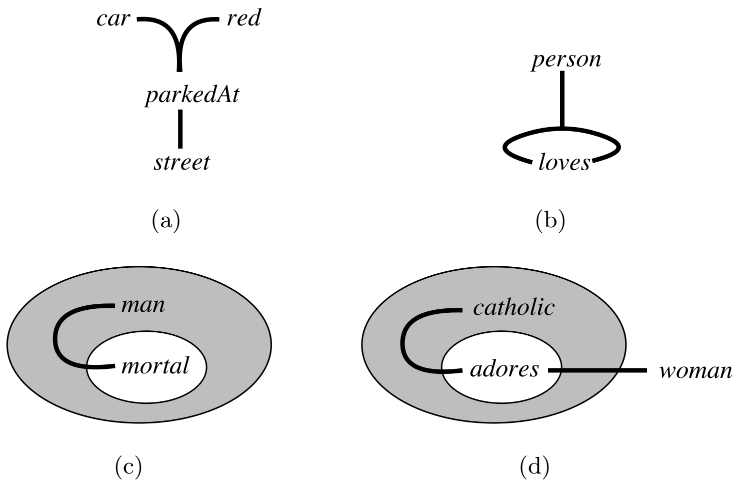


Figure 2.2: Examples of beta graphs.

As we can see, identity lines introduce a subtle difference in the role of atom names in beta graphs. Atoms represent now $n$-ary predicates whose arguments correspond to imaginary place holders surrounding the atom name that are used as endpoints of identity lines. In the case of unary predicates, such as *man* or *car*, the position of this place holder is irrelevant. However, when the predicate arity is greater than one, the argument location becomes relevant: for instance, in Fig. 2.2, predicate *adores* has a left argument that corresponds to the adorer and a right argument corresponding to the adored person.

Another important observation is that beta graphs do not provide a specific method for representing constants. For instance, there is no way for expressing that every catholic adores (Virgin) Mary other than using a unary predicate *Mary* to designate that specific person instead of some abstract *woman.*

One final remark on identity lines is that they can be actually seen as an implicit equality predicate. Some representations even introduce a label "is" for the identity line to emphasize this feature. Following this interpretation, when an identity line runs through a cut we get a convenient way to represent an inequality of the form $x \neq y$. Fig. 2.3 illustrates this case.

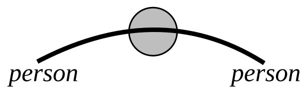
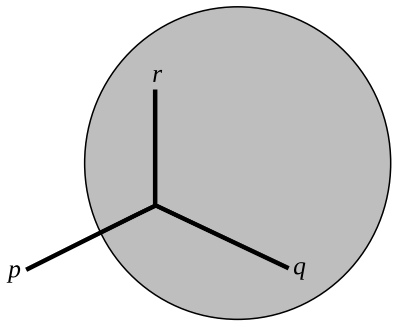


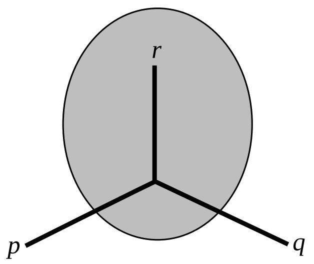
Figure 2.3: $\exists x \exists y (person(x) \wedge person(y) \wedge x \neq y)$

There is no unanimous agreement about how inequality should be treated when it affects an identity line with ramifications. However, we will follow the interpretation presented in [14], which we find coherent with the whole system. The author's vision is better understood with some examples. Fig. 2.4 shows how many quantified variables are defined depending on how the identity line intersects the cut: one (a), two (b) and three (c). Note that the cut does not need to be empty in order to represent this relation, but on these first examples we use empty cuts for clarity. In 2.4(a) only one variable is defined and shared among the three predicates. In 2.4(b) we have two variables, corresponding with the two portions of the identity line at the outermost level. In addition, the line going through the cut means that these two variables cannot be replaced with the same term. Analogously, the diagram in 2.4(c) has three variables which cannot be replaced with the same term in any case.

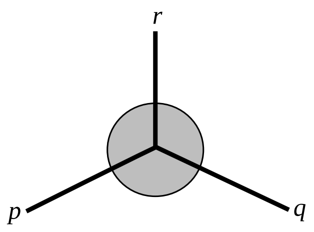
To continue with the love example, the diagram in Figure 2.5 represents that there are two different people such that one of them loves the other.



(a) $\exists x(p(x) \wedge \neg(q(x) \wedge r(x)))$



(b) $\exists x \exists y(p(x) \wedge q(y) \wedge \neg(r(x) \wedge x = y))$



(c) $\exists x \exists y \exists z(p(x) \wedge q(y) \wedge r(z) \wedge \neg(x = y \wedge x = z \wedge y = z))$

Figure 2.4: Examples of beta graphs.

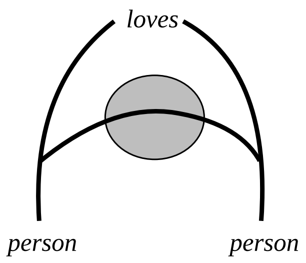


Figure 2.5: $\exists x \exists y(person(x) \wedge person(y) \wedge loves(x, y) \wedge x \neq y)$

## 2.3 Quantified Equilibrium Logic

For the sake of completeness, we provide in this section the basic definitions of Quantified Equilibrium Logic for function-free theories and Herbrand domains, since this is the most frequent situation in ASP. We consider first-order languages $\mathcal{L} = \{D,P\}$ built over a set of *constant* symbols, $D$ (the Herbrand domain), and a set of *predicate* symbols, $P$. The sets of $\mathcal{L}$-formulas, $\mathcal{L}$-sentences and atomic $\mathcal{L}$-sentences are defined in the usual way. If $D$ is a non-empty set, we denote by $At(D, P)$ the set of ground atomic sentences of the language $\langle D, P\rangle$. By an $\mathcal{L}$-interpretation $I$ over a set $D$ we mean a subset of $At(D, P)$. A *classical* Herbrand $\mathcal{L}$-structure can be regarded as a tuple $\mathcal{M} = \langle D, I\rangle$ where $I$ is an $\mathcal{L}$-interpretation over $D$.

A *here-and-there $\mathcal{L}$-structure* is a tuple $\mathcal{M} = \langle D, I_h, I_t\rangle$ where $\langle D, I_h\rangle$ and $\langle D, I_t\rangle$ are classical Herbrand $\mathcal{L}$-structures such that $I_h \subseteq I_t$. We say that the structure is *total* when $I_h = I_t$. We can think of a here-and-there structure $\mathcal{M}$ as similar to a first-order classical model, but having two parts, or components, $h$ and $t$, that correspond to two different points or “worlds”, ‘here’ and ‘there’, in the sense of Kripke semantics for intuitionistic logic, where the worlds are ordered by $h \leq t$.

We assume that $\mathcal{L}$ contains the constants $\top$ and $\bot$ and regard $\neg\varphi$ as an abbreviation for $\varphi \rightarrow \bot$. Satisfaction of formulas is defined as follows. Given some world $w \in \{h, t\}$:

- $\mathcal{M}, w \models \top$, $\mathcal{M}, w \not\models \bot$
- $\mathcal{M}, w \models p$ iff $p \in I_w$ for any atom $p \in At(D, P)$
- $\mathcal{M}, w \models c = d$ iff $c$ and $d$ denote the same constant from $D$
- $\mathcal{M}, w \models \varphi \wedge \psi$ iff $\mathcal{M}, w \models \varphi$ and $\mathcal{M}, w \models \psi$.
- $\mathcal{M}, w \models \varphi \vee \psi$ iff $\mathcal{M}, w \models \varphi$ or $\mathcal{M}, w \models \psi$.
- $\mathcal{M}, t \models \varphi \rightarrow \psi$ iff $\mathcal{M}, t \not\models \varphi$ or $\mathcal{M}, t \models \psi$.
- $\mathcal{M}, h \models \varphi \rightarrow \psi$ iff $\mathcal{M}, t \models \varphi \rightarrow \psi$ and $\mathcal{M}, h \not\models \varphi$ or $\mathcal{M}, h \models \psi$.
- $\mathcal{M}, w \models \forall x\varphi(x)$ iff $\mathcal{M}, w \models \varphi(d)$ for all $d \in D$.
- $\mathcal{M}, w \models \exists x\varphi(x)$ iff $\mathcal{M}, w \models \varphi(d)$ for some $d \in D$.

We say that $\mathcal{M}$ is a *model* of a sentence $\varphi$ iff $\mathcal{M}, h \models \varphi$. The resulting logic is called *Quantified Here-and-There Logic with static domains and decidable equality* (QHT, for short).

**Definition 2** (Equilibrium model)**.** *Let $\varphi$ be an $\mathcal{L}$-sentence. An* equilibrium *model of $\varphi$ is a total model $\mathcal{M} = \langle D, I_t, I_t \rangle$ of $\varphi$ such that there is no model of $\varphi$ of the form $\langle D, I_h, I_t \rangle$ where $I_h$ is a proper subset of $I_t$.*

When $\{D, I_t, I_t\}$ is an equilibrium model of $\varphi$ we say that the classical (Herbrand) interpretation $\{D, I_t\}$ is a *stable model* of $\varphi$.

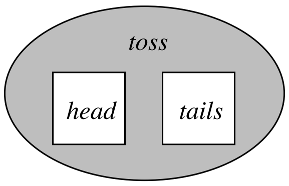
# Chapter 3

# Main Work

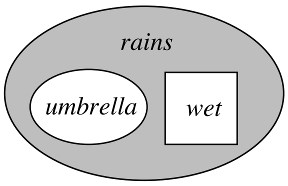
## 3.1 Equilibrium Graphs

### 3.1.1 Alpha System

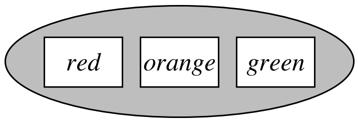
Let us begin considering the use of alpha graphs to represent Equilibrium Logic theories (or ASP logic programs). A first difficulty we face is that implication is a primitive operator in HT, and cannot be represented in terms of conjunction and disjunction (see Theorem 4 in [15]). This generates a conflict with the use of material implication in alpha graphs, defined in terms of negation and conjunction. To overcome this problem, we replace the cut component (negation) by a new diagrammatic construction we will simply call *conditional*. A conditional has the form of a closed curve (or ellipse) and may contain inside a number $n \geq 0$ of rectangles we call *consequents*. Intuitively, when all the elements inside the ellipse (but not in the rectangles) hold then one of the rectangles must hold (that is, we implicitly have a disjunction of consequents). As an example, Fig. 3.1(a) represents the implication $toss \rightarrow head \vee tails$. The case of 0 rectangles corresponds to an implication with $\perp$ (the empty disjunction) as a consequent. In other words, a conditional without consequents is just read as a negation, as happens in Peirce's alpha diagrams. As an example, Fig. 3.1(b) represents now the implication $rains \wedge \neg umbrella \rightarrow wet$, that is, $rains \wedge (umbrella \rightarrow \perp) \rightarrow wet$. It is perhaps worth to compare to Fig. 2.1(c) where, as we commented before, there was no way to differentiate between a negative condition in the antecedent and a positive condition in the consequent (*wet* and *umbrella* played the same role). This reflected the non-directional nature of material implication. Under our new notation, Fig. 3.1(b) allows now distinguishing the elements in the consequent (*wet* is inside a rectangle) from those in the antecedent, either negated (*umbrella*) or not (*rains*).



(a) $toss \rightarrow head \vee tails$



(b) $rains \wedge \neg umbrella \rightarrow wet$



(c) $red \vee orange \vee green$

Figure 3.1: Examples of conditionals.

As we have seen, when the conditional has no consequents, it corresponds to a negation. In an analogous way, when the conditional has an an empty antecedent (it only contains rectangles) it obviously represents a disjunction. Fig. 3.1(c) represents the disjunction $red \vee orange \vee green$ for the possible colors of a traffic light.

A disjunction $p \vee q$ in HT can be defined in terms of conjunction and implication, as it is equivalent to the expression

$$((p \rightarrow q) \rightarrow q) \wedge ((q \rightarrow p) \rightarrow p)$$

whose diagrammatic representation is shown in Figure 3.2. However, the only "advantage" we would gain using this representation (as primitive for disjunction) is that we would not need more than one rectangle in each conditional, while we would clearly lose readability.

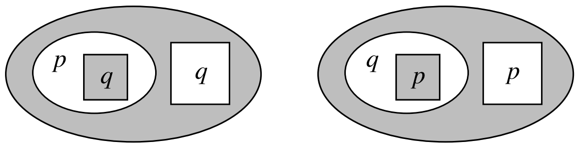
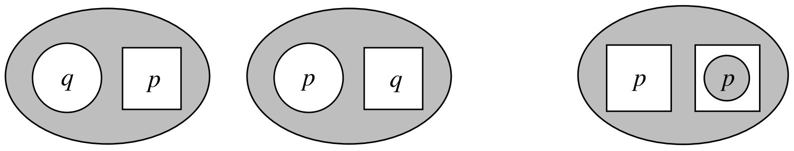


Figure 3.2: $((p \rightarrow q) \rightarrow q) \wedge ((q \rightarrow p) \rightarrow p)$

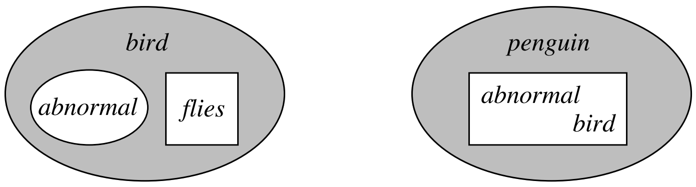
An interesting construction in ASP is the use of choice rules. The original way to build a choice that causes the non-deterministic addition of an atom $p$ in ASP was using some auxiliary predicate $q$ and building an even negative cycle as the one shown in Figure 3.3(a). A second possibility that does not require an auxiliary predicate is using the formula $p \vee \neg p$ (which is not a tautology in HT) represented in Figure 3.3(b).



(a) $(\neg q \rightarrow p) \wedge (\neg p \rightarrow q)$ (b) $p \vee \neg p$

Figure 3.3: Choice rules.

To conclude this section, we illustrate a typical example from Non-Monotonic Reasoning. Fig. 3.4 encodes a propositional program with two rules respectively asserting that a bird normally flies and that a penguin is an abnormal bird.



(a) *bird* $\wedge \neg$ *abnormal* $\rightarrow$ *flies* (b) *penguin*$\rightarrow$*abnormal*$\wedge$*bird*

Figure 3.4: Birds and penguins.

### 3.1.2 Beta System

As happened with implication in the propositional case, the universal quantifier is a primitive operator in QEL and cannot be represented in terms of existential quantifiers and the other connectives[1]. Therefore, introducing identity lines would not suffice to cover the expressive power of QEL if they were always read as existential quantifiers. Fortunately, since we count with a new conditional connective, whose expressiveness is richer than the simple cut, we can use it to cover both existential and universal quantifiers. Any identity line that satisfies the following conditions corresponds to an universal quantifier:

[1]It is actually the other way around. Any existentially quantified formula $\exists x P(x)$ is QHT equivalent to $\forall x \forall y((P(x) \rightarrow P(y)) \rightarrow P(y))$.

1. Is encircled by a conditional (cut).
2. Has some portion inside a consequent (rectangle).
3. Has some portion outside any consequent (rectangle).

Figure 3.5 shows some examples combining conditionals and identity lines. Fig. 3.5(a) corresponds to a universal quantifier, saying that all men are mortal:

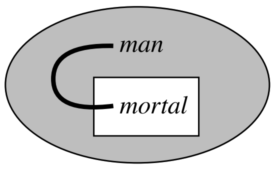
$$\forall x(\mathit{man}(x) \rightarrow \mathit{mortal}(x)) \tag{3.1}$$

Note the difference with respect to the version in Fig. 2.2(c), where we had a cut (negation) instead of the rectangle. This version is still a correct equilibrium beta graph, but its reading corresponds to:

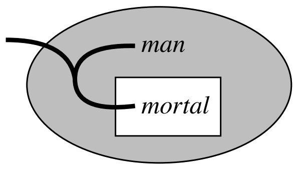
$$\forall x(\mathit{man}(x) \wedge \neg \mathit{mortal}(x) \rightarrow \bot) \tag{3.2}$$

which has a quite different meaning from (3.1): the latter is a rule that allows deriving $\mathit{mortal}(x)$ from any $\mathit{man}(x)$, whereas (3.2) acts as a constraint, forbidding stable models where some man is not known to be mortal. Another equivalent reading of a constraint like Fig. 2.2(c) (i.e. a conditional with existential lines but no consequents) is just as a negation of an existential quantifier:

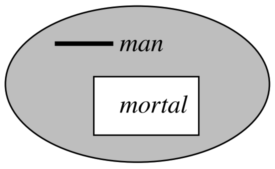
$$\neg \exists x(\mathit{man}(x) \wedge \neg \mathit{mortal}(x))$$



(a) $\forall x(\mathit{man}(x) \rightarrow \mathit{mortal}(x))$



(b) $\exists x(\mathit{man}(x) \rightarrow \mathit{mortal}(x))$



(c) $(\exists x\ \mathit{man}(x)) \rightarrow \mathit{mortal}$

Figure 3.5: Some conditionals with identity lines.

Fig. 3.5(b) corresponds to an existential quantifier: it contains an identity line which is not encircled by the conditional (it "comes from outside"). As for Fig. 3.5(c), it represents an existential quantifier: it is encircled by the conditional but it has no portion inside a consequent, so the quantifier scope is limited to the antecedent. However, a case like this can also be read as a universal quantifier in the outermost level, since $(\exists x\ man(x)) \rightarrow mortal$ is equivalent to $\forall x(man(x) \rightarrow mortal)$.

Identity lines in EqGs have also an implicit equality predicate, just as happened in Peirce's EGs. Fig 2.3 is also a valid Equilibrium Graph, though in this case its symbolic representation would be $\exists x \exists y\ (person(x)\ \wedge\ person(y)\ \wedge\ (x = y\ \rightarrow\ \bot))$. The case where an identity line goes through a rectangle should be read as $G \rightarrow (x = y) \wedge F$, where $G$ is a conjunction of the atoms in the antecedent, and $F$ is a disjunction of the atoms in the consequent. As an example of this, Fig. 3.6 means that a person can only have one name, and Eq. 3.3 is its corresponding symbolic formula:

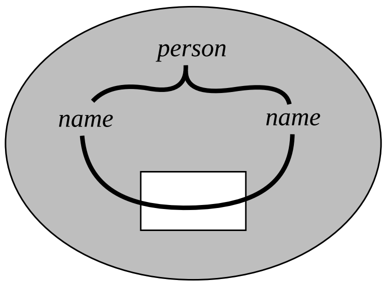


Figure 3.6: People and names.

$$\forall x \forall y\ (\exists z\ person(z)\ \wedge\ name(z,x)\ \wedge\ name(z,y)\ \rightarrow\ (x = y)) \tag{3.3}$$

One interesting property of our system is that given an Equilibrium Beta Graph $G$ we obtain its corresponding Peirce's Beta Graph $G^*$ by replacing all rectangles in $G$ by ellipses[2].

Furthermore, the symbolic representations of $G$ and $G^*$ in classical First-Order Logic are equivalent. This last property is formalized in Proposition 1.

**Proposition 1.** *Let $\Phi(G)$ denote the first-order formula associated to G under the interpretation in the current paper and $\Phi(G^*)$ the formula associated to $G^*$ under Peirce's beta graphs interpretation. Then $\Phi(G)$ and $\Phi(G^*)$ are equivalent in classical First-Order Logic.* □

[2]We could alternatively say that $G^*$ is just "Peirce's reading" of $G$ without need of any transformation, since a rectangle is also a case of closed curve, and Peirce's original approach would make no real distinction between ellipses and rectangles.

Let us consider now the diagram in 3.6 and its Peirce's symbolic interpretation defined in Eq. 3.4.

$$\neg(\exists x \exists y \exists z \ person(z) \ \wedge \ name(z,x) \ \wedge \ name(z,y) \ \wedge \ \neg(x=y)) \tag{3.4}$$

One can easily see that Eq. 3.4 and Eq. 3.3 are equivalent.

### 3.1.3 How to read the diagrams

At first glance, it may seem difficult to quickly understand a formula using this visual system, especially for someone who has been using symbolic reasoning for an entire career. Because of that, we proceed to explain a procedure that will allow the reader to inspect formulas in a recursive manner, and thus being able to abstract pieces of information from the inner levels. This is similar to how parentheses work in a symbolic formula, but without the drawback of being constantly searching for a matching brace.

Let $G$ be the diagram in Fig. 3.7 shows a formula a bit more complex than the previously presented ones. There are only two identity lines in it, but they actually represent six variables in QEL, all of them existentially quantified. Now we will parse the diagram step by step in order to get its symbolic representation. In order to do it we will use the concept of *subgraph.*

**Definition 3** (Subgraph)**.** *Let $G$ be an Equilibrium Beta Graph. A subgraph $G_i$ is an Equilibrium Beta Graph within $G$ that is fully enclosed by a cut or rectangle $C$, such that $C$ does not contain any other element than the ones in $G_i$.*

We begin at the outermost level (the main page) encapsulating everything within a cut into a *subgraph.* Then, $G$ can be rewritten as in Fig. 3.8 (or Eq.3.5).

$$\begin{aligned} G \equiv \ & \exists x_1 \exists x_2 \exists x_3 \exists x_4 \\ & (p(x_1) \wedge G_1(x_1,x_2) \wedge G_2(x_2) \wedge q(x_3) \wedge r(x_4) \wedge G_3(x_2,x_4,x_3)) \end{aligned} \tag{3.5}$$

We proceed then to step into each of the subgraphs, and repeat the process. Dashed lines are used to represent the extents of an identity line that has its end in an outer level. A grey background is used when the subgraph is in an odd level.

$$G_1 \ \equiv \ s(x_1) \ \rightarrow \ G_{1,1}(x_1,x_2) \tag{3.6}$$

$$G_2 \ \equiv \ (\exists x_5 \ m(x_5) \ \wedge \ G_{2,1}(x_5,x_2)) \rightarrow \bot \tag{3.7}$$

$$G_3 \ \equiv \ (x_2 = x_3) \wedge (x_2 = x_4) \wedge (x_3 = x_4) \rightarrow G_{3,1}(x_2) \tag{3.8}$$

Note that quantifiers are omitted for identity lines that have a dashed end. That is because they are already defined in an upper level of the formula, as well as their quantified variables. Variable names are also "inherited" from the upper level.

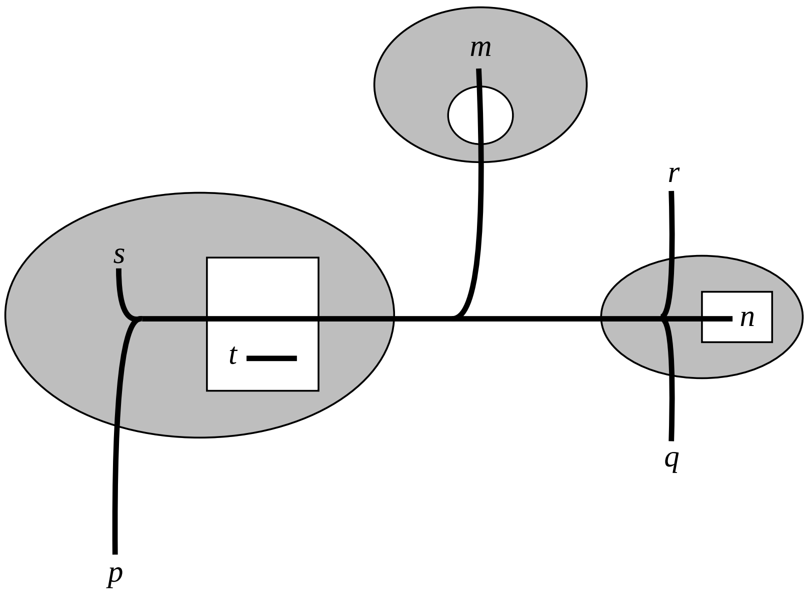


Figure 3.7: A more complex example.

Now we will focus for a moment on Fig. 3.9(c). It is a special case, because the identity line has more than one dashed end. This must be interpreted as an implicit equality predicate meaning that all the “inherited” variables are in fact the same logical object. Thus, in Eq. 3.8 we have three equality predicates (actually, one of them is redundant) and the subject of $G_{3,1}$ could be any of the three variables, as they are all the same.

Again, we recursively repeat the process for every new subgraph in Fig. 3.10.

$$G_{1,1} \equiv \exists x_6\ t(x_6) \ \wedge \ (x_1 = x_2) \tag{3.9}$$

$$G_{2,1} \equiv (x_5 \ = \ x_2) \ \rightarrow \ \bot \tag{3.10}$$

$$G_{3,1} \equiv n(x_2) \tag{3.11}$$

Finally, replacing every subgraph in Eq. 3.5 we get the full symbolic representation. If Fig. 3.7 is more readable than Eq. 3.12 or not, that is left to the reader’s judgement.

$$\begin{aligned} G \equiv\ & \exists x_1 \exists x_2 \exists x_3 \exists x_4 \exists x_5 \exists x_6 \\ & (p(x_1) \wedge q(x_3) \wedge r(x_4) \\ & \wedge (s(x_1) \ \rightarrow \ (\exists x_6\ t(x_6) \ \wedge \ (x_1 = x_2))) \\ & \wedge ((\exists x_5\ m(x_5) \ \wedge \ ((x_5 \ = \ x_2) \rightarrow \bot)) \rightarrow \bot) \\ & \wedge ((x_2 = x_3) \wedge (x_2 = x_4) \wedge (x_3 = x_4) \rightarrow n(x_2))) \end{aligned} \tag{3.12}$$

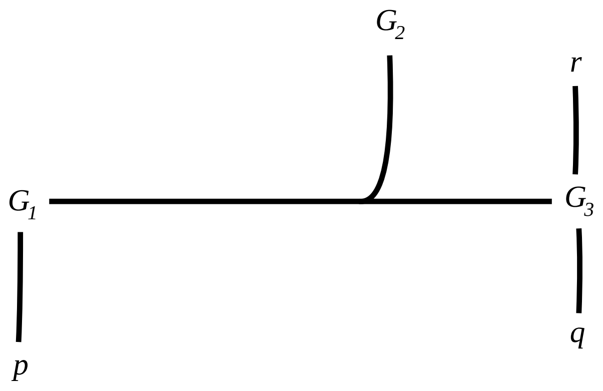
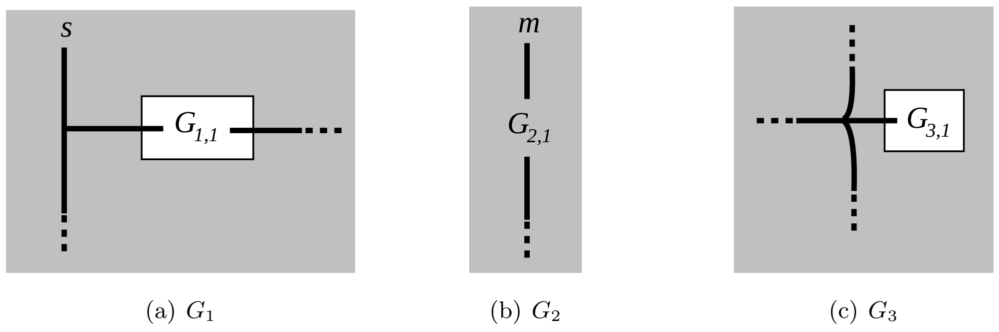
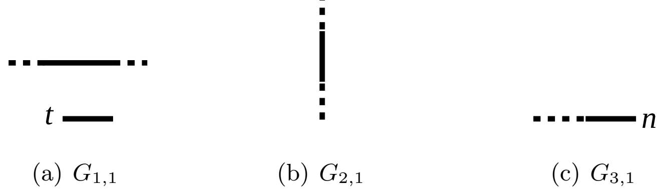


Figure 3.8: Step 1.



(a) $G_1$ (b) $G_2$ (c) $G_3$

Figure 3.9: Step 2.



(a) $G_{1,1}$ (b) $G_{2,1}$ (c) $G_{3,1}$

Figure 3.10: Step 3.

### 3.1.4 A real world problem

To conclude this section, we provide an example encoding the well-known Hamiltonian cycle problem: given a graph $G$, find a cyclic path that visits each node in $G$ exactly once. We assume that the graph $G$ is provided in terms of facts for the binary predicate *edge*, related to node names. The Hamiltonian cycle is encoded using a binary predicate *in*, meaning that the corresponding edge is included in the cycle, for a given stable model.

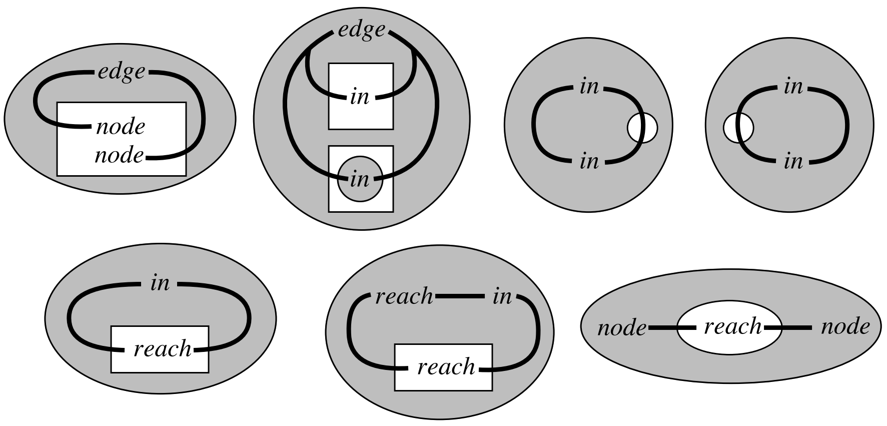


Figure 3.11: Graphical encoding of the Hamiltonian cycles problem.

Figure 3.11 shows a possible diagrammatic representation of this problem. The corresponding formulas, reading all the conditionals from left to right and from up to down would respectively be:

$$\forall x \forall y \quad \big(\ edge(x,y) \rightarrow node(x) \wedge node(y)\ \big) \tag{3.13}$$

$$\forall x \forall y \quad \big(\ edge(x,y) \rightarrow in(x,y) \vee \neg in(x,y)\ \big) \tag{3.14}$$

$$\neg \exists x \exists y \exists z \quad \big(in(x,y) \wedge in(x,z) \wedge y \neq z\ \big) \tag{3.15}$$

$$\neg \exists x \exists y \exists z \quad \big(x \neq y \wedge in(x,z) \wedge in(y,z)\ \big) \tag{3.16}$$

$$\forall x \forall y \quad \big(\ in(x,y) \rightarrow reach(x,y)\ \big) \tag{3.17}$$

$$\forall x \forall y \forall z \quad \big(\ reach(x,y) \wedge in(y,z) \rightarrow reach(x,z)\ \big) \tag{3.18}$$

$$\neg \exists x \exists y \quad \big(node(x) \wedge node(y) \wedge \neg reach(x,y)\ \big) \tag{3.19}$$

All of them can be easily rewritten as program rules in standard ASP syntax. (3.13) asserts that the two arguments of predicate *edge* are nodes. (3.14) is a non-deterministic choice to include any edge in the stable model or not. (3.15) and (3.16) are constraints respectively forbidding that two edges in the cycle with the same origin go to two different targets, and vice versa, that two different origin nodes go to a common target. Formulas (3.17),(3.18) define the transitive closure *reach* of predicate *in*. Finally, (3.18) is a constraint forbidding that a node $y$ cannot be reached from another node $x$.

Figure 3.12 shows three diagrams respectively depicting a possible example of input graph (facts for predicate *edge*) plus the two corresponding stable models that represent the Hamiltonian cycles of the input graph.

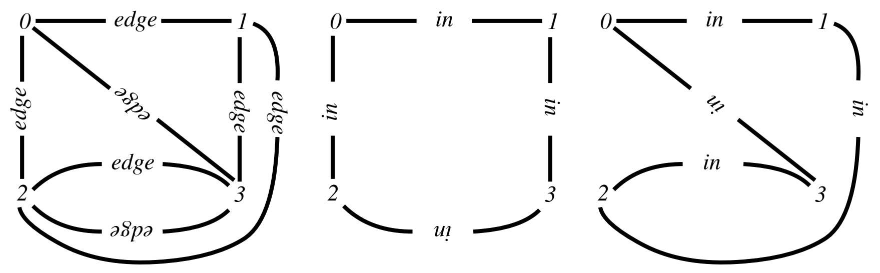


Figure 3.12: A graph and its two stable models corresponding to the Hamiltonian paths.

## 3.2 From diagrams to programs

We have introduced so far the concept of Equilibrium Graphs, as well as its equivalence to a Quantified Equilibrium Logic formula. Now if we want to apply this concept to computer-aided problem solving, we must find an automated method to translate a diagram into a symbolic formula, since computers are pure symbolic machines. This method is composed of four main steps.

1. Obtain the diagram's equivalent symbolic formula in QEL.
2. Reduce the QEL formula obtained in the previous step to Prenex Normal Form.
3. Reduce the propositional part of the formula obtained in the previous step to Negation Normal Form.
4. Reduce the propositional formula obtained in the previous step to a logic program.

The procedure to reduce an arbitrary propositional formula in Equilibrium Logic to a logic program can be found in [16]. This can be easily extended to Quantified Equilibrium Logic by adding a previous step that reduces the formula to PNF, and then reducing the propositional part treating predicates as plain literals.

### 3.2.1 Obtaining the equivalent symbolic representation

A high-level explanation of this procedure was already provided in Section 3.1.3. It highly depends on implementation, so it will be analyzed in depth later.

### 3.2.2 Prenex Normal Form

A formula is said to be in Prenex Normal Form (PNF) if it is written as

$$Q_1 x_1 Q_2 x_2 ... Q_n x_n \varphi$$

where each $Q_i$ $(i = 1, 2, ..., n)$ is an existential or universal quantifier and $\varphi$ is a quantifier-free formula. The first part is usually known as the *prefix*, and the second as the *matrix*.

#### Transformation rules

Let $\varphi(x)$ and $\psi$ be arbitrary formulas in Quantified Equilibrium Logic, such that the variable $x$ appears free in $\varphi(x)$ but not in $\psi$. We can obtain the equivalent formula in

Prenex Normal Form by applying the following transformation rules in no particular order:

$$\neg\exists x\ \varphi(x) \ \Leftrightarrow\ \forall x\ \neg\varphi(x) \tag{1.1}$$

$$\neg\forall x\ \varphi(x) \ \Leftrightarrow\ \exists x\ \neg\varphi(x) \tag{1.2}$$

$$Qx\ \varphi(x)\ \wedge\ \psi \ \Leftrightarrow\ Qx\ (\varphi(x)\ \wedge\ \psi) \tag{2}$$

$$Qx\ \varphi(x)\ \vee\ \psi \ \Leftrightarrow\ Qx\ (\varphi(x)\ \wedge\ \psi) \tag{3}$$

$$\psi\ \rightarrow\ Qx\ \varphi(x) \ \Leftrightarrow\ Qx\ (\psi\ \rightarrow\ \varphi(x)) \tag{4}$$

$$\exists x\ \varphi(x)\ \rightarrow\ \psi \ \Leftrightarrow\ \forall x\ (\varphi(x)\ \rightarrow\ \psi) \tag{5.1}$$

$$\forall x\ \varphi(x)\ \rightarrow\ \psi \ \Leftrightarrow\ \exists x\ (\varphi(x)\ \rightarrow\ \psi) \tag{5.2}$$

Note that if a variable name appears more than once in a formula, it should be renamed in order to properly apply this rules. Since in `Grasp` the names of variables are automatically generated to be unique, we don't have to deal with renaming issues.

### 3.2.3 Negation Normal Form

A formula is said to be in Negation Normal Form (NNF) when negation operator is only applied to literals.

#### Transformation rules

Let $\varphi$ and $\psi$ be arbitrary propositional formulas in Equilibrium Logic. We can obtain the equivalent formula in Negation Normal Form by applying the following transformation rules in no particular order:

$$\begin{array}{rlrl}
\neg\top \Leftrightarrow \bot & (1) & \neg(\varphi\wedge\psi) \Leftrightarrow \neg\varphi\vee\neg\psi & (4)\\
\neg\bot \Leftrightarrow \top & (2) & \neg(\varphi\vee\psi) \Leftrightarrow \neg\varphi\wedge\neg\psi & (5)\\
\neg\neg\neg\varphi \Leftrightarrow \neg\varphi & (3) & \neg(\varphi\rightarrow\psi) \Leftrightarrow \neg\neg\varphi\wedge\neg\psi & (6)
\end{array}$$

### 3.2.4 Logic Program

A logic program is a set of rules of the form

$$q_1, \ldots, q_k \leftarrow p_1, \ldots, p_m,\ not\ p_{m+1}, \ldots, not\ p_n$$

where all $p$ and $q$ are atoms. Commas on the head of the rule represent disjunction, while on the body represent conjunction.

## Transformation rules

Let $\varphi$ and $\psi$ be arbitrary propositional formulas in Equilibrium Logic. Let $\alpha$ be a conjunction of literals and $\beta$ a disjunction. We can obtain the equivalent logic program by applying the following transformation rules in no particular order:

Left side rules:

$$\top \wedge \alpha \rightarrow \beta \ \Leftrightarrow\ \{\ \alpha \rightarrow \beta\ \} \tag{L1}$$

$$\bot \wedge \alpha \rightarrow \beta \ \Leftrightarrow\ \varnothing \tag{L2}$$

$$\neg\neg\varphi \wedge \alpha \rightarrow \beta \ \Leftrightarrow\ \{\ \alpha \rightarrow \neg\varphi \vee \beta\ \} \tag{L3}$$

$$(\varphi \vee \psi) \wedge \alpha \rightarrow \beta \ \Leftrightarrow\ \left\{ \begin{array}{ccc} \varphi \wedge \alpha & \rightarrow & \beta \\ \psi \wedge \alpha & \rightarrow & \beta \end{array} \right\} \tag{L4}$$

$$(\varphi \rightarrow \psi) \wedge \alpha \rightarrow \beta \ \Leftrightarrow\ \left\{ \begin{array}{rcl} \neg\varphi \wedge \alpha & \rightarrow & \beta \\ \psi \wedge \alpha & \rightarrow & \beta \\ \alpha & \rightarrow & \varphi \vee \neg\psi \vee \beta \end{array} \right\} \tag{L5}$$

Right side rules:

$$\alpha \rightarrow \bot \vee \beta \ \Leftrightarrow\ \{\ \alpha \rightarrow \beta\ \} \tag{R1}$$

$$\alpha \rightarrow \top \vee \beta \ \Leftrightarrow\ \varnothing \tag{R2}$$

$$\alpha \rightarrow \neg\neg\varphi \vee \beta \ \Leftrightarrow\ \{\ \neg\varphi \wedge \alpha \rightarrow \beta\ \} \tag{R3}$$

$$\alpha \rightarrow (\varphi \wedge \psi) \vee \beta \ \Leftrightarrow\ \left\{ \begin{array}{ccc} \alpha & \rightarrow & \varphi \vee \beta \\ \alpha & \rightarrow & \psi \vee \beta \end{array} \right\} \tag{R4}$$

$$\alpha \rightarrow (\varphi \rightarrow \psi) \vee \beta \ \Leftrightarrow\ \left\{ \begin{array}{rcl} \varphi \wedge \alpha & \rightarrow & \psi \vee \beta \\ \neg\psi \wedge \alpha & \rightarrow & \neg\varphi \vee \beta \end{array} \right\} \tag{R5}$$

Note that rules (L5) and (R5) are the only ones to introduce new negations and that they both result in $\neg\varphi$ and $\neg\psi$ for the inner implication $\varphi \rightarrow \psi$. Thus, if the original propositional formula was in NNF, the computation of NNF in each intermediate step is needed for these newly generated $\neg\varphi$ and $\neg\psi$. Note also that these transformations do not guarantee the absence of redundant, subsumed or tautology formulas.

## 3.3 `Grasp`

In this section we present `Grasp` (Graphical Reasoning for Answer Set Programming), a visual tool for diagrammatic problem-solving with Equilibrium Graphs.

`Grasp` is a tool made for viewing and editing Equilibrium Graphs fully integrated with the Potsdam Answer Set Solving Collection (POTASSCO) [17], a system to ground and solve ASP logic programs.

### 3.3.1 Development Methodology

The nature of this project claimed for an agile development methodology from the very beginning. The theoretical framework of Equilibrium Graphs was going to be developed along with `Grasp`, so we opted for an approach based on rapid prototyping and testing. In agile software development, requirements and solutions evolve through time as new problems arise and new features are implemented. We have picked ideas from both SCRUM and Kanban for the development process of `Grasp`, so as to provide rapid and flexible response to change from the very beginning of the project to its end. Relying on a robust version control system is also a key point in any software development project. In our case, this was carried out with `Git` version control.

The development process has been divided in two main stages, each of them composed of several iterations. The main goal of **Stage 1** was to develop a fully functional Alpha-System prototype; the main goal of **Stage 2** was to develop a fully functional Beta-System application, now focusing also on usability and user experience, and not only on diagram creation and edition.

Below we give a brief description of the work done in each iteration. Iterations 1 to 7 correspond to **Stage 1**, and iterations 8 to 12 correspond to **Stage 2**. Except for background and design iterations, each of them produced a working prototype built on top of the previous one.

**Iteration 1**: Background study of Peirce's EGs.

**Iteration 2**: Design of the Alpha System prototype.

**Iteration 3**: Implementation of Drawing module.

**Iteration 4**: Background study of ASP.

**Iteration 5**: Implementation of Normalization module and `Clingo` integration.

**Iteration 6**: Implementation of Save/Load diagram functionality.

**Iteration 7**: Design and Implementation of a Graphical User Interface (GUI).

**Iteration 8**: Design of the Beta System prototype.

**Iteration 9**: Extension of Drawing module for Beta graphs.

**Iteration 10**: Extension of Normalization module for Beta graphs.

**Iteration 11**: Extension of Save/Load data functionality for Beta graphs.

**Iteration 12**: `Graphviz` integration.

**Iteration 13**: Drafting of the present document.

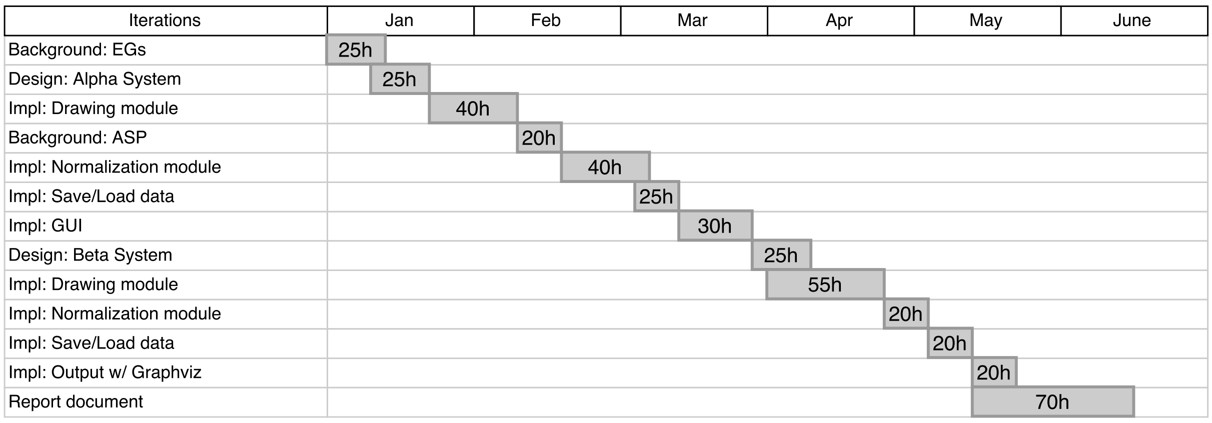


Figure 3.13: Gantt chart.

Now we present an estimation of the project's economic budget. Assuming a salary of 10 euros per hour/person in an average development team (one single person in this case), and with a total of 410 work hours, the development cost of this project would be 4100 euros. We should also add to this the cost the supervisors' validation time, which was approximately two hours/person per iteration with a salary of 28 euros/hour, resulting in a total amount of 13 iterations $\times$ 2 hours $\times$ 28 euros/hour $\times$ 2 supervisors $=$ 1456 euros. This sums up to 5556 euros. Production cost is exactly the same as development cost, since all additional software we use is distributed with no payments required.

### 3.3.2 Design and Main Features

A software like `Grasp` can be divided in two main parts:

- A diagrammatic front-end that serves as Human-Computer Interface, used for the visualisation and edition of EqGs, i.e. the modeling of a problem.
- A symbolic back-end (POTASSCO), used for the solving process.

Because of the fact that POTASSCO works only with symbolic formulas, we are forced to translate the user's input into an equivalent symbolic formula. This, however, must be done in a way that is completely hidden to the user.

The main design challenge of `Grasp` was to decide how the communication between this two main parts would be. In first place, the translation from a diagrammatic formula to its symbolic equivalent is not a trivial task. After that, additional computation is needed in order to reduce the symbolic QEL formula to a logic program. At the initial phase of the design process, two approaches were considered:

- Digital Image Processing approach: The software would be composed of a simple *image editor* module, where the user could draw freely, and an *image parser* module, which receives an image as input and performs Digital Image Processing operations on it in order to obtain a meaningful specification.
- Semantic approach: The software would be composed of one main *diagram editor* module. Note the use of the word diagram instead of image. In this approach, the user is only allowed to draw meaningful images (diagrams), so that the system is able to keep an internal representation of the formula *with no additional processing.*

The selected approach was finally the semantic one, for the reasons explained below. Semantic approach has no need for a complex DIP analysis, while the other one does. Moreover, no syntax errors are possible (overlapping shapes, closed lines, etc.), only errors of a semantic kind. This means that every diagram created with `Grasp` is in fact a well-formed QEL formula. The increased complexity of the drawing module is balanced with the absence of an image parser module (although we will see later that an additional normalization module is needed in any case).

Fig. 3.14 shows the data pipeline of `Grasp`: first of all, the user introduces a diagram formula in the Drawing module; then, the formula is passed to the Normalization module in Reverse Polish Notation (RPN); this module reduces the formula to a logic program as described in [16] and gives it to `Clingo` so as to start the solving process; when `Clingo` finishes, the stable models are given to the `Graphviz` module in order to present results again in diagrammatic form.

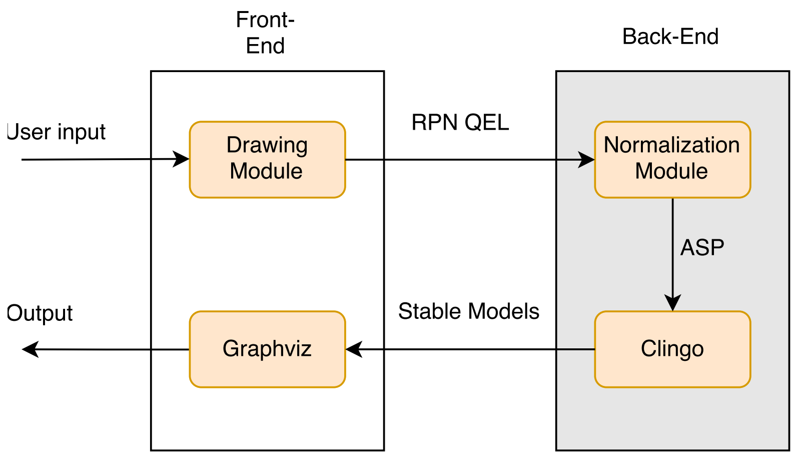


Figure 3.14: The data pipeline (showing only high-level modules).

### 3.3.3 Implementation

The `Python` language [18] was our first choice in order to turn `Grasp` into an actual executable application; and `CPython`, its reference implementation, was our main development tool. `Python` is a high-level interpreted programming language with a design philosophy that emphasizes code readability. It features a dynamic type system, automatic memory management and supports multiple programming paradigms, such as object-oriented, imperative, functional (partially), and procedural styles. It has a large and comprehensive standard library, and an active community with a great collaborative philosophy. All of this features provide a very suitable environment for rapid prototyping and easy maintenance.

The front-end of `Grasp` is powered by the `Kivy Library` [19], a cross-platform graphical user interface framework for `Python`. `Kivy` has been chosen mainly because of its flexibility and ease of use. The decision to use a general purpose *graphics* library rather than a *graph* library was made because of the multiple specific restrictions we face when manipulating the diagrams. We preferred to implement it ourselves in order to avoid eventual limitations of using a library designed for working with graphs, but not Existential Graphs.

The back-end of `Grasp` is powered by `Clingo` (part of the POTASSCO suite [17]), a software system that combines `gringo` (an ASP grounder) and `clasp` (an ASP solver) into one bundled application. It was chosen because of its acceptance among the ASP community, its great performance, and also because of the availability of bindings for `Python`.

In addition, `Graphviz` [20] was used as the main tool for output results. `Graphviz` is a package of tools for drawing graphs specified in DOT language scripts (a sample script is provided in Appendix E.0.2). Its main feature is to automatically generate a layout for an arbitrary graph description, trying to minimize edge crossings with nodes and other edges. This makes a lot easier the task of translating the output of `Clingo` back into diagrammatic form. Since this output (*stable models*) consists exclusively of a conjunction of predicates, there is no need to handle nested graphs with cuts and rectangles. Moreover, as it does not make much sense to edit an *stable model*, we opted for a non-interactive output style.

The whole development process was done under a GNU/Linux environment. However, the combination of `Python` + `Kivy` makes instantaneous the task of porting an application to any operating system in which `Python` can be run). So, although our back-end (`Clingo`) only runs on Unix-like environments, an eventual port announcement from POTASSCO developers would let us distribute `Grasp` for the new supported platform without additional work on the front-end side. This advantage was another reason that made us prefer this set of tools over the rest.

### Stage 1: Alpha System

Our first goal was to have a minimal vertical slice[3] of the system as soon as possible. The needed components are a simple *drawing module* (consisting only of an interactive canvas, similar to many general-purpose drawing applications), a *propositional normalization module*, and `Clingo`. Output (*stable models*) is directly printed to command line. A self-imposed restriction is to maximize cohesion and minimize coupling between modules. Thus, the two mentioned modules will remain *totally independent of each other*. The communication between them two will be done with Reverse Polish Notation (RPN) formulas implemented as standard `Python` strings, so that any of the modules can be completely replaced without affecting the other.

We will talk first about the *drawing module*. The core elements of this module are what we call diagram *widgets*. A *widget* is a graphical entity with logical meaning. So atoms, cuts (called ellipses on the implementation) and rectangles are *widgets*. As they have some common properties, they have been implemented through *object inheritance* as described in Appendix D. One important thing to mention about cut and rectangle *widgets* is that they have their own canvas, they are indeed *subgraphs* where more *widgets* can be added recursively. Then, a parent-child relationship exists between a cut/rectangle and the *widgets* on its canvas.

[3]A vertical slice is a cross-sectional demonstration through the layers that form the structure of a software program.

The *drawing module* exposes to the user an interactive canvas, which is the only user interface at this stage. Based on user input, we distinguish between two types of actions that modify the current diagram on the canvas: *semantic* actions and *presentation* actions. Both types' actions have a direct effect on the appearance of diagrams, but only those of the first type do alter the meaning of the formula. Actions that fall into the semantic category are: addition of an atom, cut or rectangle; deletion of an atom, cut or rectangle; renaming of an atom. Actions that fall into the presentation category are: translation of an atom, cut or rectangle through the diagram space; scaling of a cut or rectangle.

Presentation actions could break the diagram meaning if not implemented carefully. If the user was allowed to apply tanslation and scaling operations freely, she might alter a problem specification with an action that was only intended to have aesthetic effects. Even worse, the resulting diagram may have structural errors, such as overlapping shapes. For this reason, in `Grasp` presentation actions are constrained so as to maintain the meaning of the formula unaltered. The constraints present at this stage are the following:

1. The position of a *widget* must be within the area of its parent.
2. The whole area of a *widget* must be strictly within the area of its parent.
3. The area of a *widget* must not overlap with the area of any other *widget* than its parent.

These constraints have some interesting side effects worth mentioning. A *widget* that is added to a cut or rectangle becomes part of its scope, being permanently attached until an eventual deletion.

Following the design approach described in Section 3.3.2, we want to build a system where no image processing techniques are necessary in order to obtain the symbolic equivalent of a diagram. Thus, *widgets* data is arranged in the computer memory as a tree structure, in a way that is easily translatable into a symbolic formula using Alg. 3.1. For instance, the diagram in Fig. 3.15 would be represented internally as in Fig. 3.16.

We will now focus on the *propositional normalization module.* Its main task is to reduce an arbitrary propositional Equilibrium Logic formula to a logic program. First of all, we shall define the data structures used in this module. The most important one is the `Node` class (see Fig. D.2), which is used for building logic formulas. Indeed, the internal representation of a logic formula is conceptually equivalent to a *parse tree*

**Algorithm 3.1** Obtain an RPN formula from a widget tree

```
function GET_FORMULA_RPN(T: WidgetTree): String
    s ← Empty String
    if is_atom(T) then
        s ← T.as_string()
    else
        if is_ellipse(T) then
            conjunction ← Empty Stack
            disjunction ← Empty Stack
            for all child ∈ T do
                if is_square(T) then
                    conjunction.push(GET_FORMULA_RPN(child))
                else
                    disjunction.push(GET_FORMULA_RPN(child))
                end if
            end for
            s.add_operator(LIST_TO_BINARY_OP(conjunction, and, ⊤))
            s.add_operator(LIST_TO_BINARY_OP(disjunction, or, ⊥))
            s.add_operator(implies)
        else
            conjunction ← Empty Stack
            for all child ∈ T do
                conjunction.push(GET_FORMULA_RPN(child))
            end for
            s.add_operator(LIST_TO_BINARY_OP(conjunction, and, ⊤))
        end if
    end if
    return s
end function
```

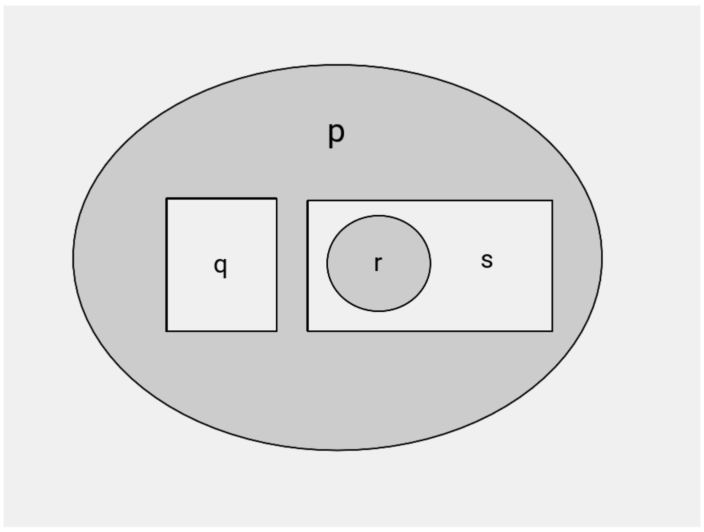


Figure 3.15

built from `Node` instances. For example, the parse tree of diagram in Fig. 3.15 would be as the one in Fig. 3.17.

Formula manipulation is done according to the transformation rules introduced in Section 3.2. Algorithms 3.2 and 3.4 perform NNF and logic program reduction, respectively.

Algorithm 3.2 takes a `Node` instance as input and returns another `Node` as output. The computation is pretty straightforward. First, iterate over the set of transformation rules (denoted by $\mathcal{R}$) in search of an applicable rule for the current node. If one is found, apply it. Then, a recursive call is made in order to get the NNF formula for *left* and *right* nodes. This manner of visiting the nodes of a parse tree is broadly known as *depth-first pre-order* traversal.

Algorithms 3.3 and 3.4 require a bit more explanation. The first one operates at *rule level*, while the second does at *program level*. In this context, a rule is represented as a 4-tuple $(\alpha_t,\ \alpha_n,\ \beta_t,\ \beta_n)$ where $\alpha_t$ is a set of head literals for which the normalization process has already terminated, $\alpha_n$ is a set of head subformulas for which the process has not terminated, $\beta_t$ is a set of body literals for which the normalization process has already terminated, and $\beta_n$ is a set of body subformulas for which the normalization process has not terminated. A program is represented as a 2-tuple $(S_t,\ S_n)$ where $S_t$ are normalized rules and $S_n$ are rules to be normalized. By the application of transformation rules showed in Section 3.2, any QEL formula will converge into a logic program, so the termination condition $S_n = \emptyset$ will eventually become true.

The computation is performed as follows. If $S_n$ is not empty, get any rule from it

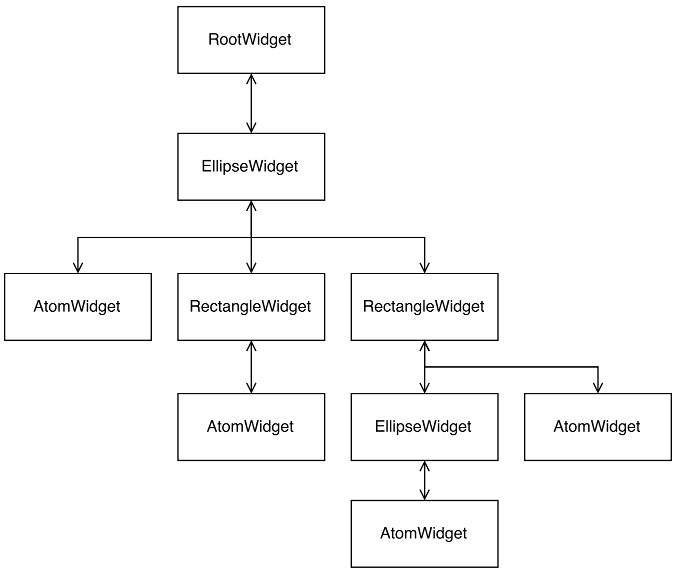


Figure 3.16: Memory layout of Fig. 3.15 (bidirectional arrows represent double links).

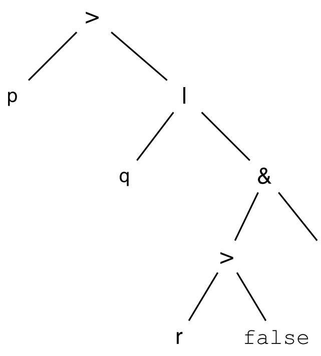


Figure 3.17: Parse tree of diagram in Fig. 3.15.

and look at its $\alpha_n$ and $\beta_n$. If any of them two is not empty then a substitution can be applied. After applying the substitution, add the new formulas to $S_n$ again in order to normalize them. If $\alpha_n$ and $\beta_n$ were both empty, then the formula is removed from $S_n$ and added to $S_t$.

Finally, translation into ASP syntax is trivial, since it is only needed to replace lexical symbols. A collection of unit tests has been created for each rule's representative use cases.

**Algorithm 3.2** Reduce a propositional Equilibrium Logic formula to NNF

```
1:  function NNF(N: Node): Node
2:      $n \leftarrow$ copy(N)
3:      for all $r \in \mathcal{R}$ do
4:          if is_applicable($r$, $n$) then
5:              apply($r$, $n$)
6:          end if
7:      end for
8:      if not is_literal($n$) then
9:          $n.left \leftarrow$ NNF($n.left$)
10:         $n.right \leftarrow$ NNF($n.right$)
11:     end if
12:     return $n$
13: end function
```

**Algorithm 3.3** Search for a suitable transformation rule and apply it

```
1: function APPLY_SUBSTITUTION($f$: 4-tuple, $side$: [left, right]): set of 4-tuple
2:     for all $r \in \mathcal{R}(side)$ do
3:         if is_applicable($r$, $f$) then
4:             return apply($r$, $f$)
5:         end if
6:     end for
7: end function
```

**Algorithm 3.4** Reduce a propositional Equilibrium Logic formula to a logic program

```
1: function NORMALIZE($S_t$: set of 4-tuple, $S_n$: set of 4-tuple): set of 4-tuple
2:     while $S_n \neq \emptyset$ do
3:         $G \leftarrow \emptyset$
4:         $f \leftarrow (x \in S_n)$
5:         if $\beta_n \neq \emptyset$ then
6:             $G \leftarrow G \cup$ APPLY_SUBSTITUTION($f$, right)
7:         else if $\alpha_n \neq \emptyset$ then
8:             $G \leftarrow G \cup$ APPLY_SUBSTITUTION($f$, left)
9:         else
10:            $G \leftarrow G \cup \{f\}$
11:        end if
12:        if $G = \{f\}$ then
13:            $S_t \leftarrow S_t \cup G$
14:        else
15:            $S_n \leftarrow S_n \cup G$
16:        end if
17:    end while
18:    return $S_t$
19: end function
```

### Stage 2: Beta System

As we have already explained, the Beta System is a superset of the Alpha System, so we have built the new set of features on top of the Alpha prototype. The growing complexity of the system calls for new modules to be implemented: first, an actual GUI is an essential part of any desktop application; second, we want to generate output in diagrammatic form instead of printing *stable models* to command line; and last but not least, a *name manager* module is required in order to handle atom and predicate names properly. Of course, the previous *drawing* and *normalization* modules must be extended to support new features.

Identity line drawing will be done through the use of new entities called *anchor points* and *nexus*. Anchor points are the place to start a line from, and they are always attached to some predicate. A predicate can have a maximum of four anchor points. That means we are limited to use up to four arguments for every predicate (enough for the kind of problems we will be modeling). Nexus are a special type of anchor point not attached to any predicate. They serve as intermediate points from which the user can start new segments to create "spider" identity lines. Fig. 3.18 shows how anchor points and nexus look like in a diagram with 3 anchor points and 1 nexus.

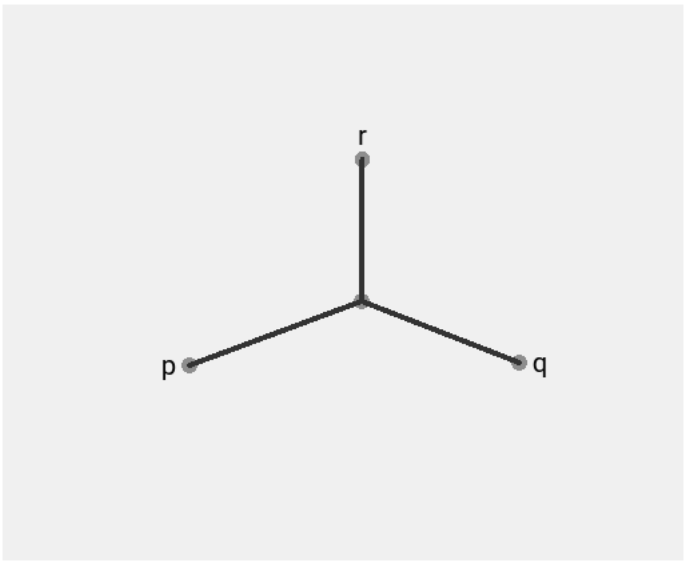


Figure 3.18: $\exists x\ (p(x) \wedge q(x) \wedge r(x))$

We have previously mentioned that in ASP, programs with variables must be grounded in order to get a solution. So an special type of predicate is defined in order to represent constants. An identity line attached to a constant no longer represents a variable: it is replaced by the constant value. Some examples with constants are provided in Appendix C.

One single identity line in EqGs may represent more than one variable in QEL, so correctly identifying variables can be a challenging task for the newcoming user. Because of this, a variable-highlighting feature has been implemented, like is shown in Fig. 3.19. Each color represents a new variable.

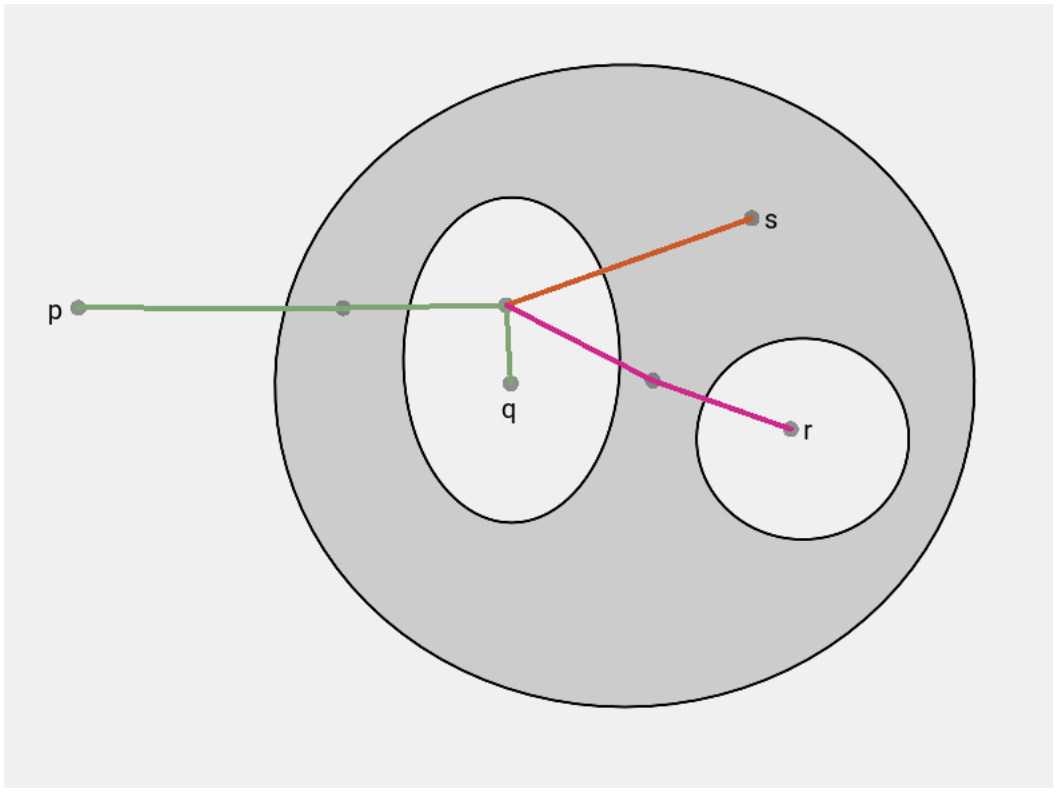


Figure 3.19: $\exists x$
$(p(x) \wedge ((\exists y \exists z\ (s(y) \wedge (x = y \wedge x = z \wedge q(x) \rightarrow \bot) \wedge (r(z) \rightarrow \bot)))) \rightarrow \bot))$

We have previously commented that any propositional Equilibrium Logic formula can be reduced to a logic program. So does a QEL formula, but the generated logic program may not be safe (see Definition 1). If the program is not safe, `Clingo` will not be able to search for *stable models*, so `Grasp` notifies the user consequently.

### 3.3.4 Software Architecture

The architecture of `Grasp` is illustrated in Fig. 3.20. It consists of several independent modules that communicate with each other in order to process data.

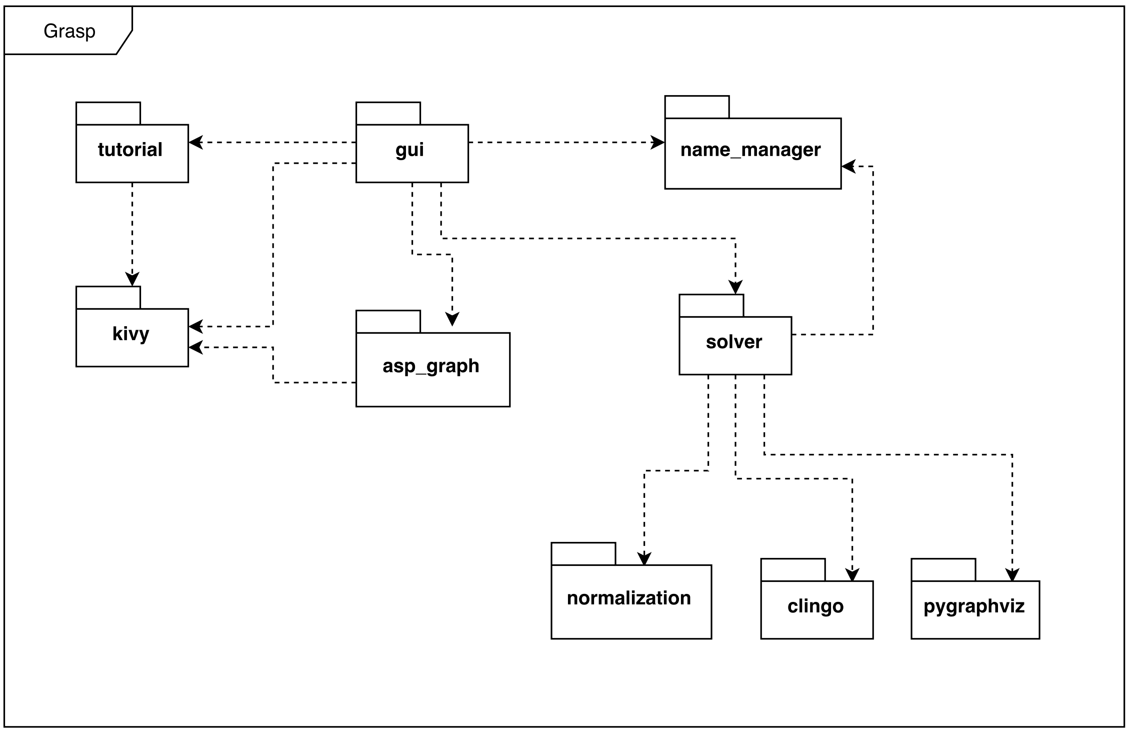


Figure 3.20: UML Package Diagram of `Grasp`.

### 3.3.5 The `Grasp` file format

An important feature of any document edition software is to be able to capture the state of a document and save it for a later use. In order to implement this, we define a new markup syntax, based on the `kv` language. A `Grasp` file is composed of two main sections:

1. The **header**, where names and lines are declared.
2. The **body**, where the diagram layout is described.

A sample file is provided in Appendix E.0.1

# Chapter 4

# Related Work

To the best of our knowledge, there is no other software tool for logic-based diagrammatic non-monotonic reasoning. Some tools have been developed over the past years with the aim of helping with visualisation of ASP problem encodings and solutions. The most relevant ones are cited below.

- `ASPIDE` [21], an ASP Integrated Development Environment that features a visual node editor for ASP. `ASPIDE` allows building rules in a graphical way, but the diagrams are closer to database Entity-Relationship scheme rather a full-fledged logical notation.

- `ASPVIZ` [22] and `Kara` [23] are tools for visualising answer sets in which ASP itself is used to define how visualisations are constructed. However, the ASP problem encodings must be written in the traditional symbolic notation.

- `ARVis` [24] (available as a plugin for ASPIDE), a tool for visualising relations between answer sets. Again, this tool is focused on producing an easily understandable output and not on the representation of problems in diagrammatic form.

Very few efforts have been done to develop Existential Graphs related software. The only relevant work found is `Peirce-Logic`, a web app for working with Peirce's Alpha Graphs proofs [25].

Some attempts have been made to build diagrammatic software tools based on Sowa's Conceptual Graphs, but most of them are not maintained anymore. However, we provide here a reference for the sake of completeness [26].

Another tool related to our project is `f2lp` [27]. Although it has nothing to do with Diagrammatic Reasoning, it performs a similar reduction from Quantified Equilibrium

Logic to logic programs, as the one we presented in Section 3.2.

# Chapter 5

# Conclusions and Future Work

In the previous sections we have introduced Equilibrium Graphs (EqGs), a diagrammatic logic system with an expressive power equivalent to Quantified Equilibrium Logic (QEL), and we have brought it into reality through `Grasp`, a visual software tool for creating diagrammatic logic programs within the Answer Set Programming (ASP) paradigm. `Grasp` allows the user to model computational problems through the creation and manipulation of EqGs, as well as saving and loading diagram data. Moreover, ASP solver `Clingo` and graph visualization library `Graphviz` are fully integrated with the application, so the user is able to work with diagrams from the beginning of the process until its end.

`Grasp` has proved to be a comfortable tool to draw Equilibrium Graphs, and even Existential Graphs. At least more comfortable than general purpose graph editors, though much work can still be done if we want to improve the system. Developing the software tool has also helped us to improve the diagrammatic formalism of EqGs in several ways. For instance, it has forced us to clarify the treatment of equality/inequality, something poorly specified in the original versions of EqGs Beta System.

`Grasp` was not designed taking into account efficiency issues, but rather focusing on the EqGs formalism, and how to translate it into an actual usable application. For this reason, the most interesting research that could be done would be to test `Grasp` with users not familiar with mathematical logic at all. Then, the learning curve of Equilibrium Graphs should be compared with that of symbolic First Order Logic, in order to reveal the main differences between diagrammatic and symbolic reasoning.

From a theoretical point of view, the next step should be to define *stable models* in diagrammatic terms, as well as a set of axioms and inference rules for diagrammatic reasoning with EqGs. Peirce already did this for EGs, and his work could again serve as inspiration for our non-monotonic version of his system.

Another interesting direction to go from here is to explore the possibility of extending it in order to support **functions**. Some research has already been done by F. Dau in [28]. Functions would be very useful for modeling mathematical concepts intuitively, and would simplify a lot some of the formalizations that have been tested.

While problem encodings are easy to model in EqGs and correspond to simple, clear diagrams, it is not the same for problem data. For example, the graph instance in the Hamiltonian cycles example (see Appendix C.1) is quite readable with 4 nodes, but if we extend it to 8 or 10 it becomes a confusing network of identity lines and predicates. A possible solution for this would be to allow the user to associate special **line rendering styles** to an specific predicate. Thus, the predicate would be implicit in the line, and the total number of lines in a program would be dramatically reduced. For example, the *edge* predicate in Hamiltonian Cycles and Graph Coloring problems could be represented as a dashed line that connects nodes directly. Such representation would lead to a simple, readable diagram, much closer to the layout of an actual graph, even for large problem instances.

With the same goal of improving readability, a method for handling **line crossings** should be considered. Although this feature does not appear usually in Existential Graphs, it may have practical benefits in our application. It would be nice to have also a **better output presentation**. `Graphviz` is a powerful tool, but with large problem instances it sometimes fails to place atoms and identity lines correctly.

Other very important aspect is to test `Grasp` with more complex, real-world problems. We know that our diagrammatic system is suitable for many quite simple use cases, but we cannot imagine to what extent this could be a useful tool in other scenarios.

A huge amount of work that can also be done on the interface side. There is no point in mentioning here every little detail, so we will briefly enumerate the main improvements that can be done. First, implement a Tab system, with the purpose of allowing the user to work on more than one document at the same time. Even more, the user could choose to select more than one file to generate the ASP program. Then, an essential feature of a software editor of any kind is a proper Undo/Redo system. It was not implemented yet because of its complexity for graphical applications. Finally, Multiple Selection and Copy/Paste functionalities would highly improve usability and would allow users to draw faster.

Another useful feature would be to **import ASP code**. As we already have a system that translates diagrams into logic programs, it would be very interesting to have a second tool for the reverse process, namely, obtaining diagrammatic representations from existing textual ASP encodings. `Graphviz` library could be used to optimize the layout of the generated graph, but additional checks would probably be required. This idea seems feasible for programs without variables (i.e. modeled in Alpha System), but further research would be needed in order to design a robust ASP importer for non-ground programs.

# Appendix A

# Installation

**Disclaimer**: Grasp was only tested on a Debian GNU/Linux environment.

In order to get the software running on your maching, please follow these steps:

1. Install dependencies: Python language (version 2.7 w/ Standard Library), Kivy library, pygraphviz module. For example, with apt package manager:

   ```
   $ sudo apt-get install python
   $ sudo apt-get install python-kivy python-pygraphviz
   ```

2. Run (assuming that current path is the project's root directory):

   ```
   $ python src/main.py
   ```

3. A simple interactive tutorial can be started from Main Menu > Help > Quick Tutorial. Some examples of problem formalizations can be found under /examples directory.

# Appendix B

# User's Manual

The interface has three main parts:

- Main Menu Bar: Traditional menu bar with all available functions.
- Canvas: The place to draw diagrams.
- Side Panel: Used for managing names and predicate properties.

The canvas has two operating modes: SELECT and INSERT. Graph creation is done entirely with the mouse. Keyboard is only used for shortcuts.

| Mode | Left-Click | Right-Click | Middle Button | Scroll |
|---|---|---|---|---|
| `INSERT` | Add item | Delete item | Pan | Zoom |
| `SELECT` | Move item | Resize item | Pan | Zoom |

### B.0.1 Shortcuts

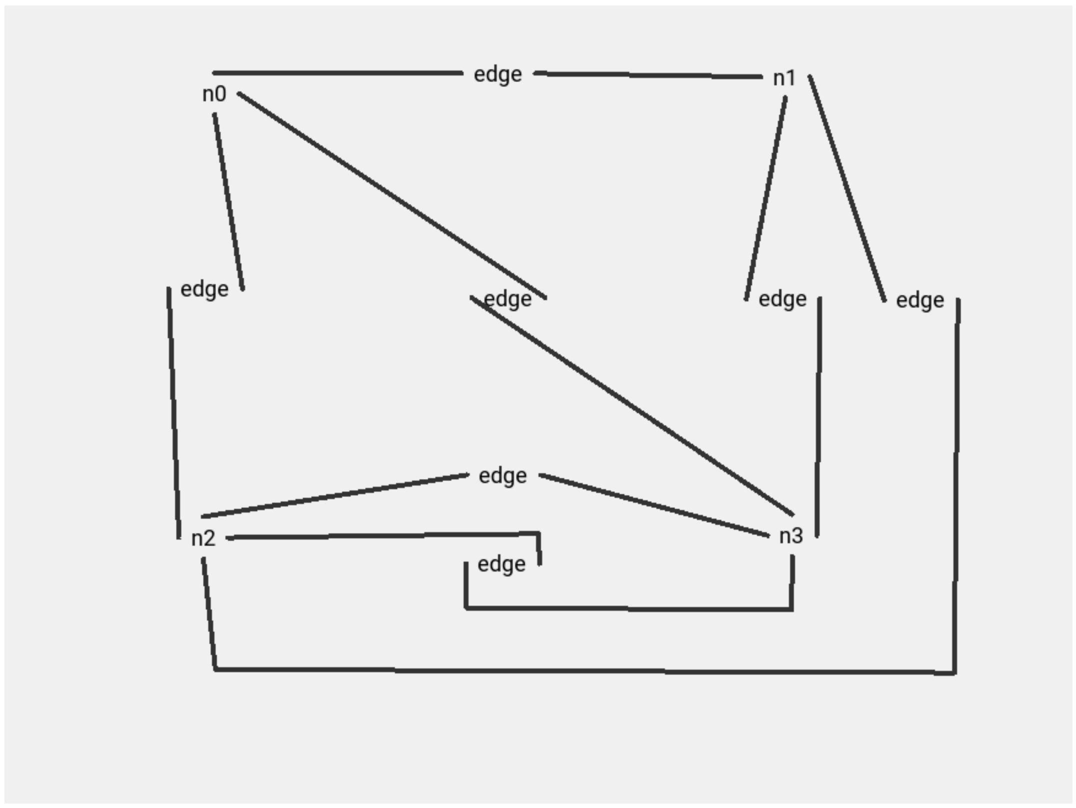
| Key | Action |
|---|---|
| `D` | `INSERT` Mode |
| `S` | `SELECT` Mode |
| `W` | Atom |
| `E` | Ellipse |
| `R` | Rectangle |
| `T` | Toggle side panel |
| `TAB` | Focus name input |
| `ESC` | Quit |

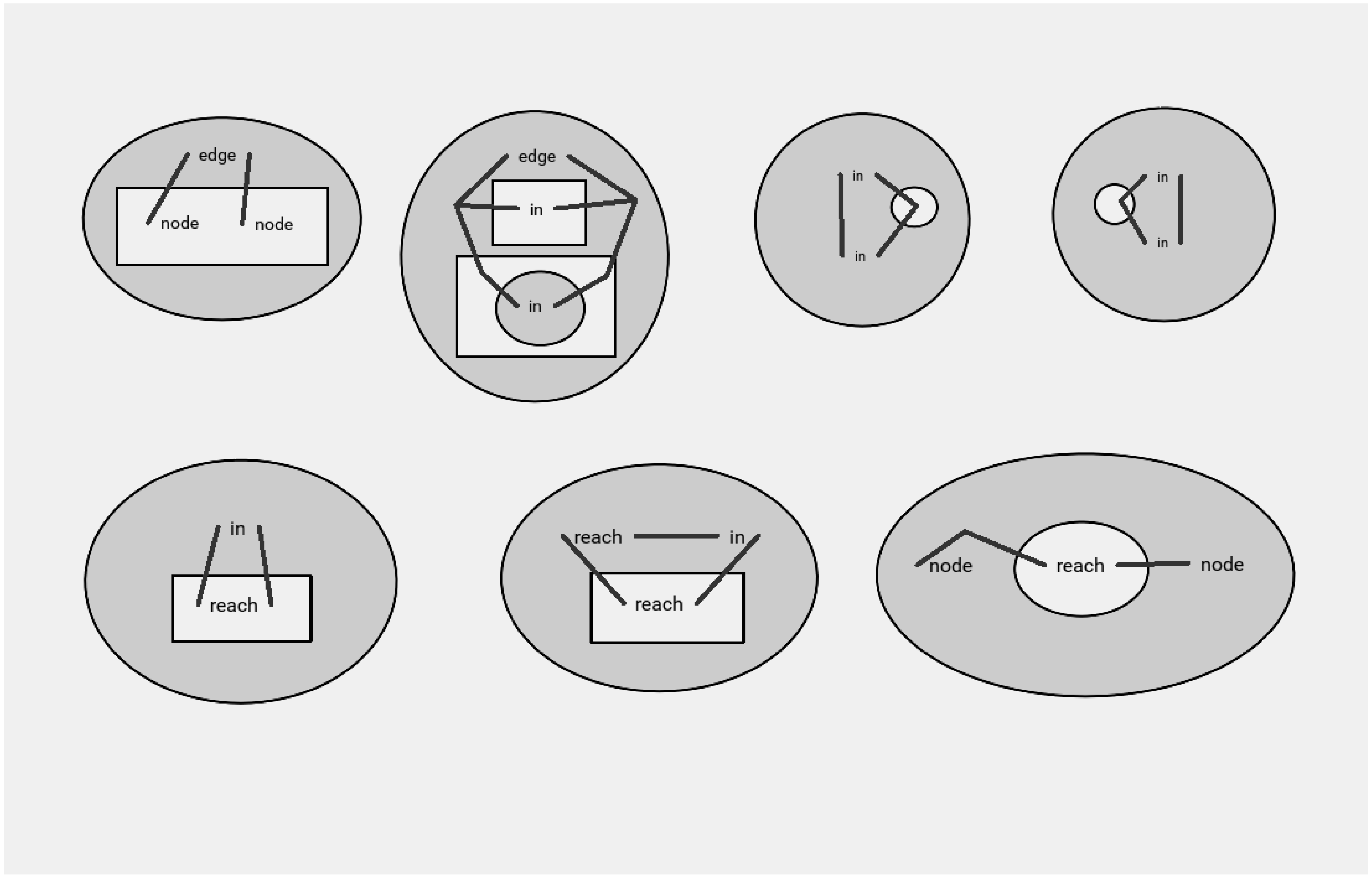
# Appendix C

# Examples

## C.1 Hamiltonian Cycles

### C.1.1 Equilibrium Graph



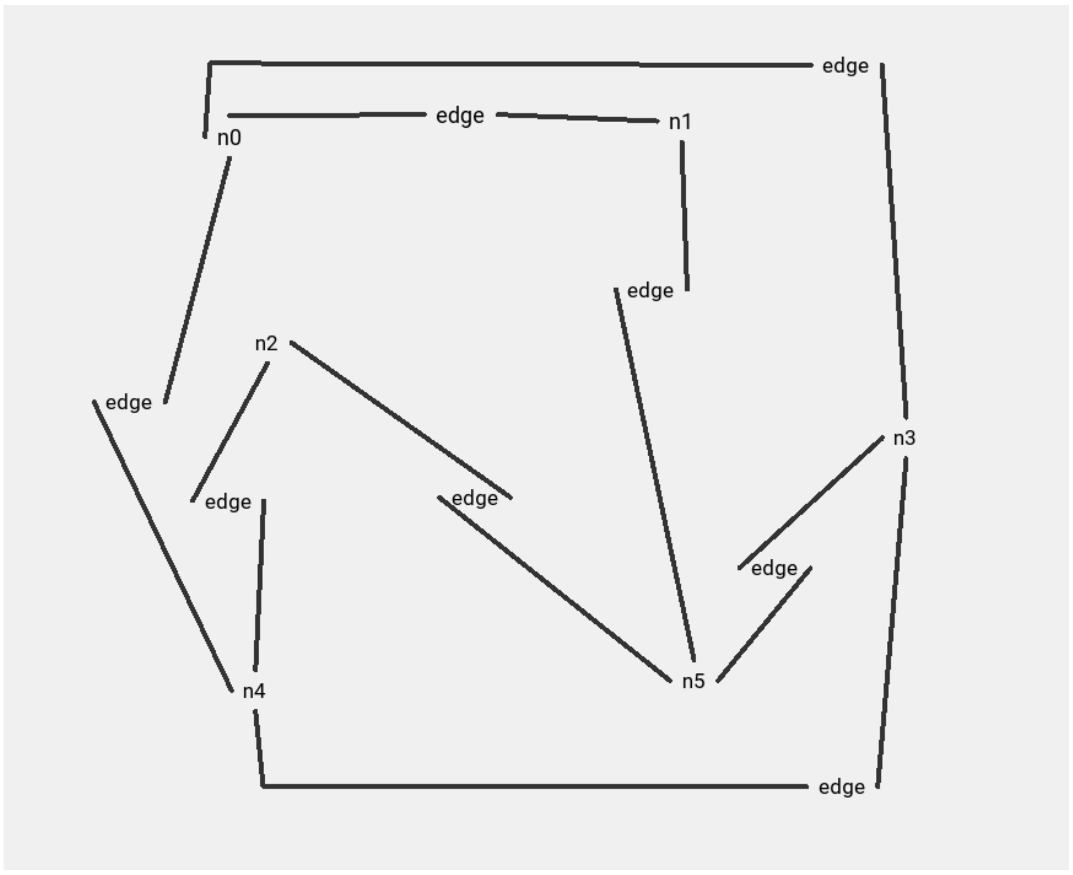

edge
node
node
edge
in
in
in
in
in
in
in
reach
reach
in
reach
node
reach
node

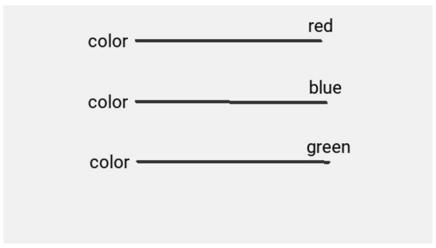
### C.1.2 ASP program

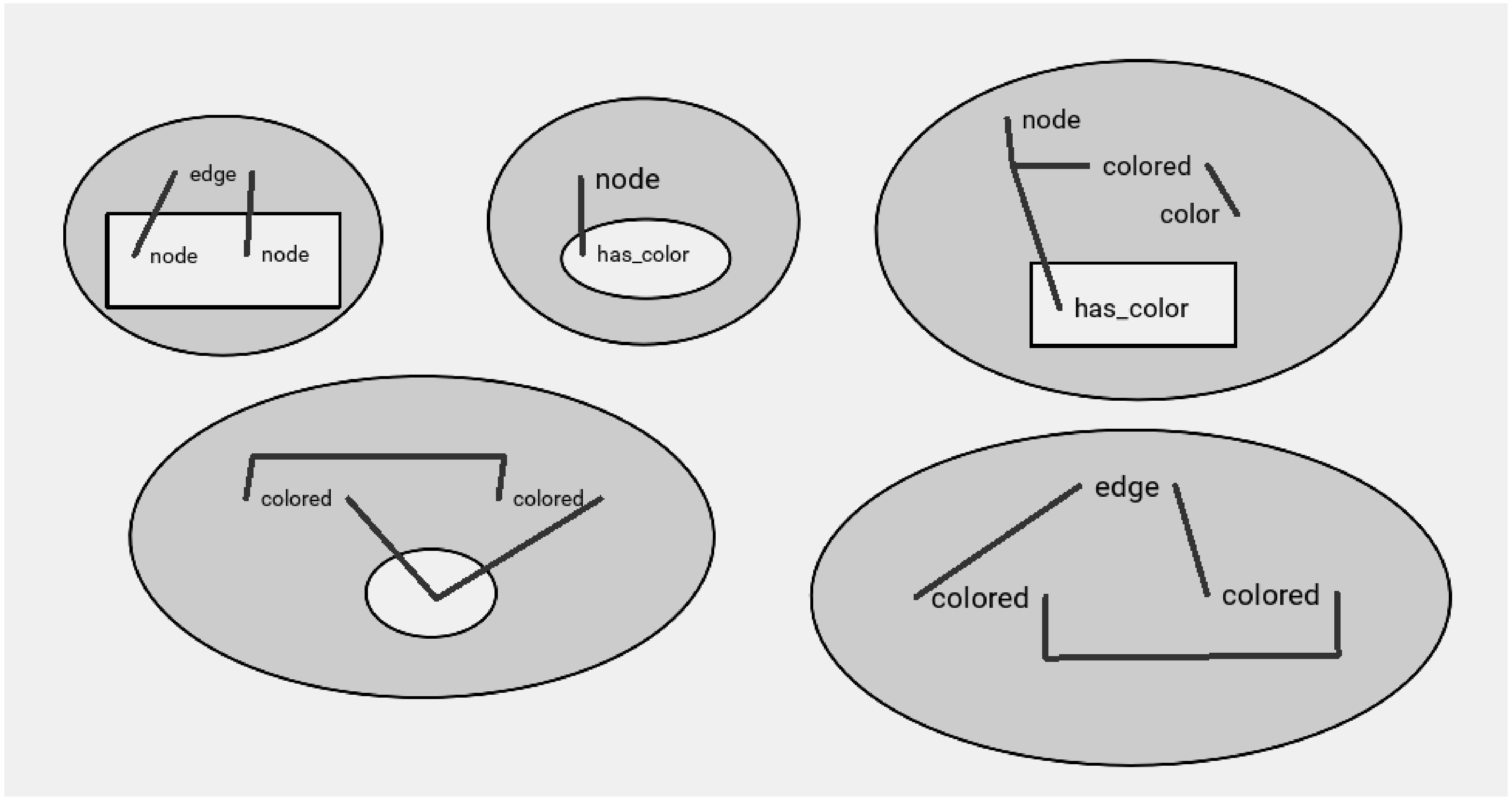
```
edge(n2,n0).
edge(n1,n3).
edge(n3,n2).
edge(n1,n2).
edge(n3,n0).
edge(n0,n1).
edge(n2,n3).
node(X1_0) :- edge(X0_0,X1_0).
in(X2_0,X3_0) :- edge(X2_0,X3_0), in(X2_0,X3_0).
 :- in(X5_0,X4_0), in(X5_0,X4_1), not X4_0=X4_1.
 :- in(X6_0,X7_0), in(X6_1,X7_0), not X6_0=X6_1.
reach(X10_0,X12_0) :- in(X11_0,X12_0), reach(X10_0,X11_0).
reach(X8_0,X9_0) :- in(X8_0,X9_0).
 :- node(X14_0), node(X13_0), not reach(X13_0,X14_0).
node(X0_0) :- edge(X0_0,X1_0).
in(X2_0,X3_0), not in(X2_0,X3_0) :- edge(X2_0,X3_0).
```

Hamiltonian Cycles auto-generated code

## C.2 Graph Coloring





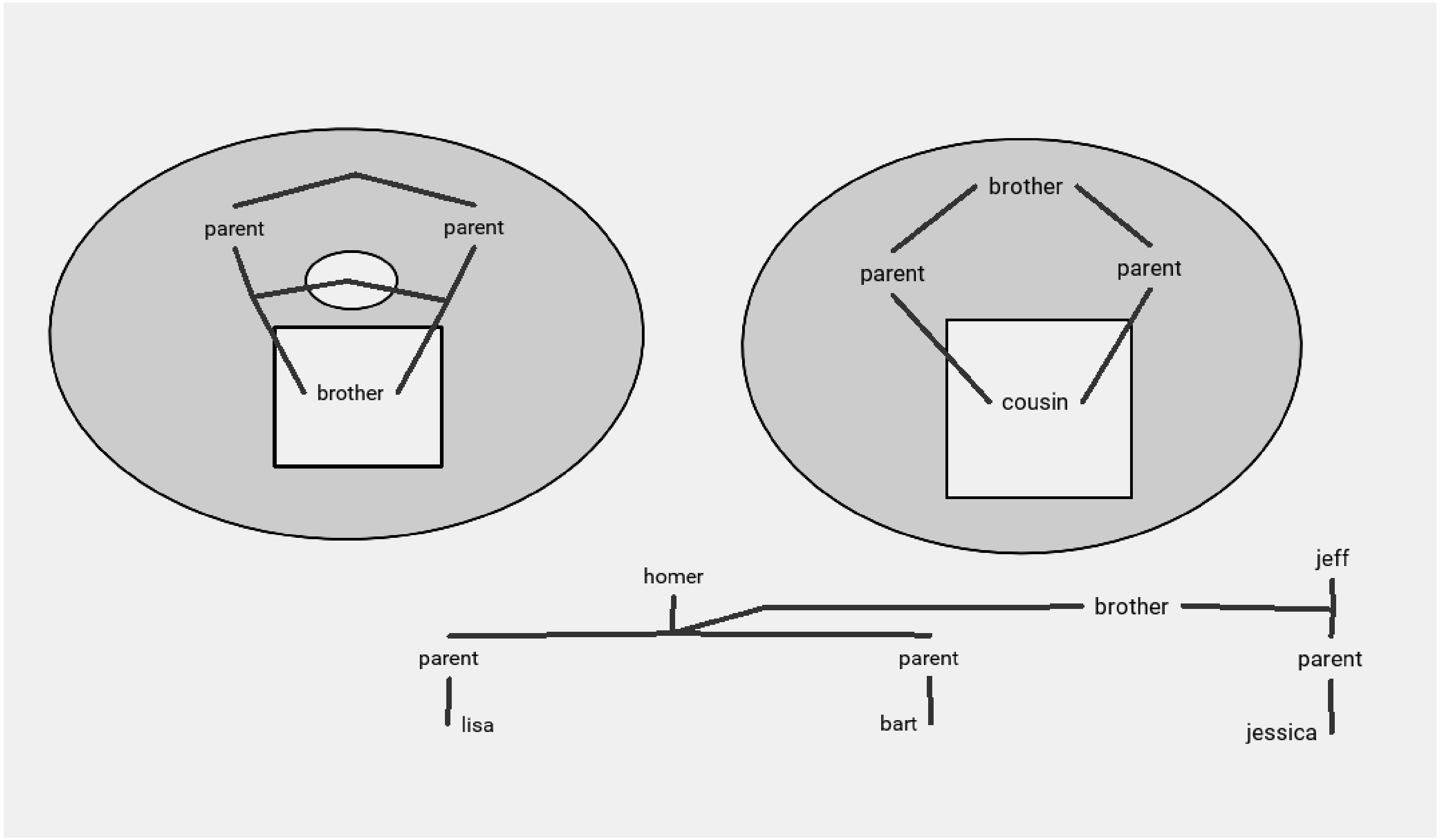

edge
node
node
node
has_color
node
colored
color
has_color
colored
colored
edge
colored
colored

### C.2.1 ASP program

```
color(red).
color(green).
color(blue).
edge(n3,n5).
edge(n2,n4).
edge(n0,n1).
edge(n4,n0).
edge(n0,n3).
edge(n5,n2).
edge(n4,n3).
edge(n5,n1).
node(X7_0) :- edge(X7_0,X6_0).
 :- node(X9_0), not has_color(X9_0).
 :- edge(X14_0,X15_0), colored(X15_0,X16_0), colored(X14_0,X16_0).
has_color(X10_0) :- colored(X10_0,X11_0), color(X11_0), node(X10_0).
colored(X12_0,X13_0) :- colored(X12_0,X13_0), node(X12_0), color(
    X13_0).
node(X6_0) :- edge(X7_0,X6_0).
colored(X12_0,X13_0), not colored(X12_0,X13_0) :- node(X12_0), color(
    X13_0).
 :- colored(X5_0,X8_1), colored(X5_0,X8_0), not X8_0=X8_1.
```

Graph Coloring auto-generated code

## C.3 Family Relations

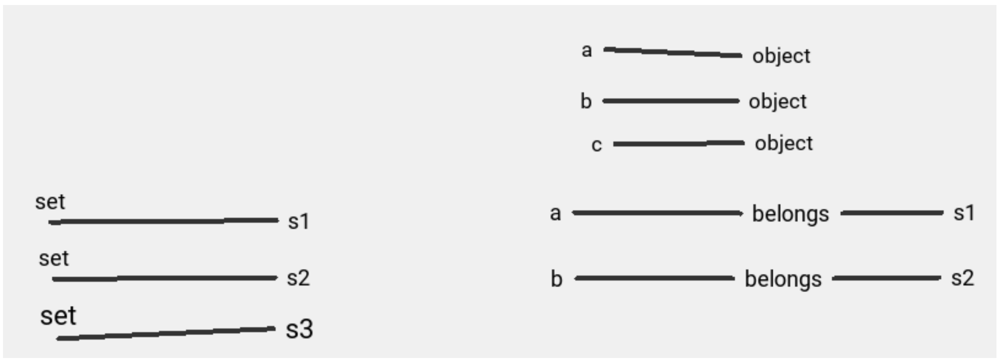

parent
parent
brother
brother
parent
parent
cousin
homer
jeff
brother
parent
parent
parent
lisa
bart
jessica

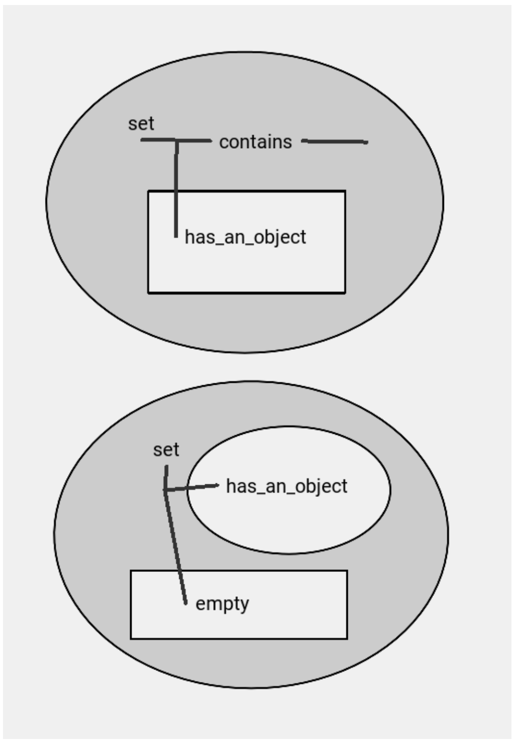
### C.3.1 ASP program

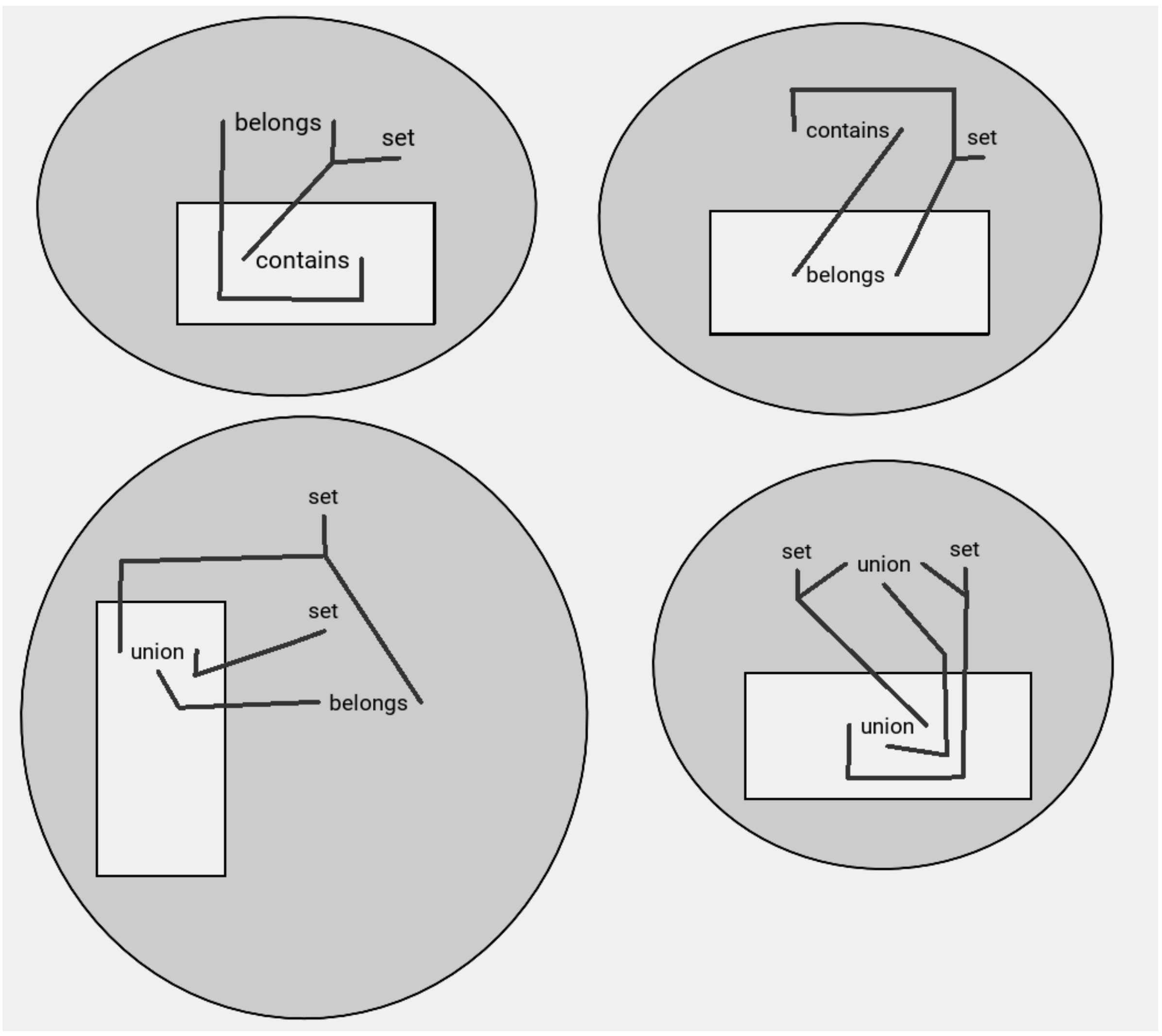
```
parent(homer,bart).
brother(homer,jeff).
parent(homer,lisa).
parent(jeff,jessica).
brother(X10_0,X10_1) :- parent(X0_0,X10_1), parent(X0_0,X10_0), not
    X10_0=X10_1.
cousin(X6_0,X4_0) :- brother(X7_0,X5_0), parent(X5_0,X4_0), parent(
    X7_0,X6_0).
```

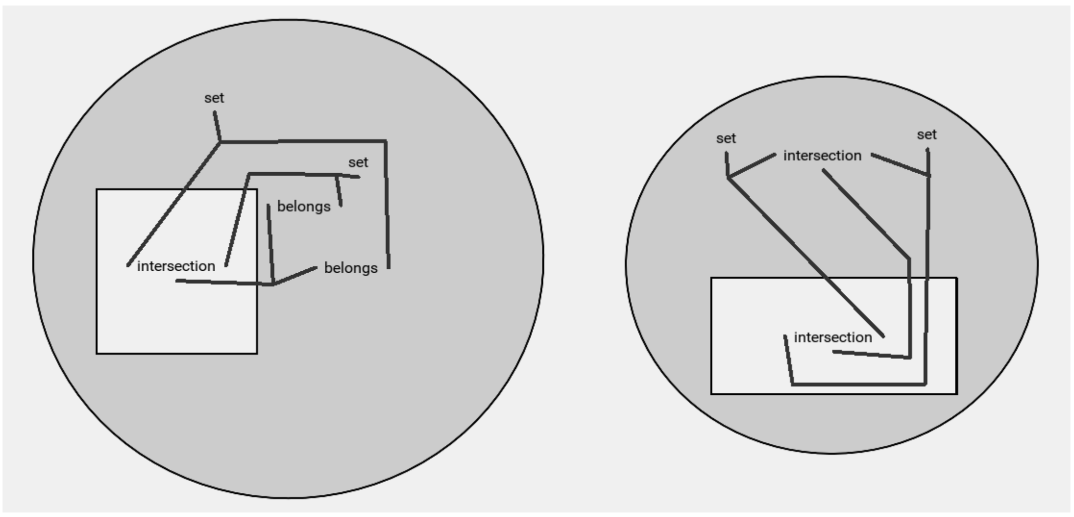
Family Relations auto-generated code

## C.4 Set Theory






belongs
set
contains
contains
set
belongs
set
set
union
belongs
set
union
set
union


set
set
belongs
intersection
belongs
set
intersection
set
intersection

### C.4.1 ASP program

```
object(a).
object(b).
object(c).
set(s1).
set(s2).
set(s3).
belongs(a,s1).
belongs(b,s2).
union(X19_0,X18_0,X17_0) :- belongs(X17_0,X19_0), set(X18_0), set(
    X19_0).
intersection(X24_0,X25_0,X23_0) :- set(X24_0), belongs(X23_0,X24_0),
    belongs(X23_0,X25_0), set(X25_0).
belongs(X9_0,X8_0) :- set(X8_0), contains(X8_0,X9_0).
has_an_object(X14_0) :- contains(X14_0,X13_0), set(X14_0).
contains(X6_0,X7_0) :- set(X6_0), belongs(X7_0,X6_0).
union(X20_0,X21_0,X22_0) :- set(X21_0), set(X20_0), union(X21_0,X20_0
    ,X22_0).
empty(X15_0) :- set(X15_0), not has_an_object(X15_0).
intersection(X27_0,X26_0,X28_0) :- set(X26_0), intersection(X26_0,
    X27_0,X28_0), set(X27_0).
```

Set Theory auto-generated code

# Appendix D

# UML Diagrams

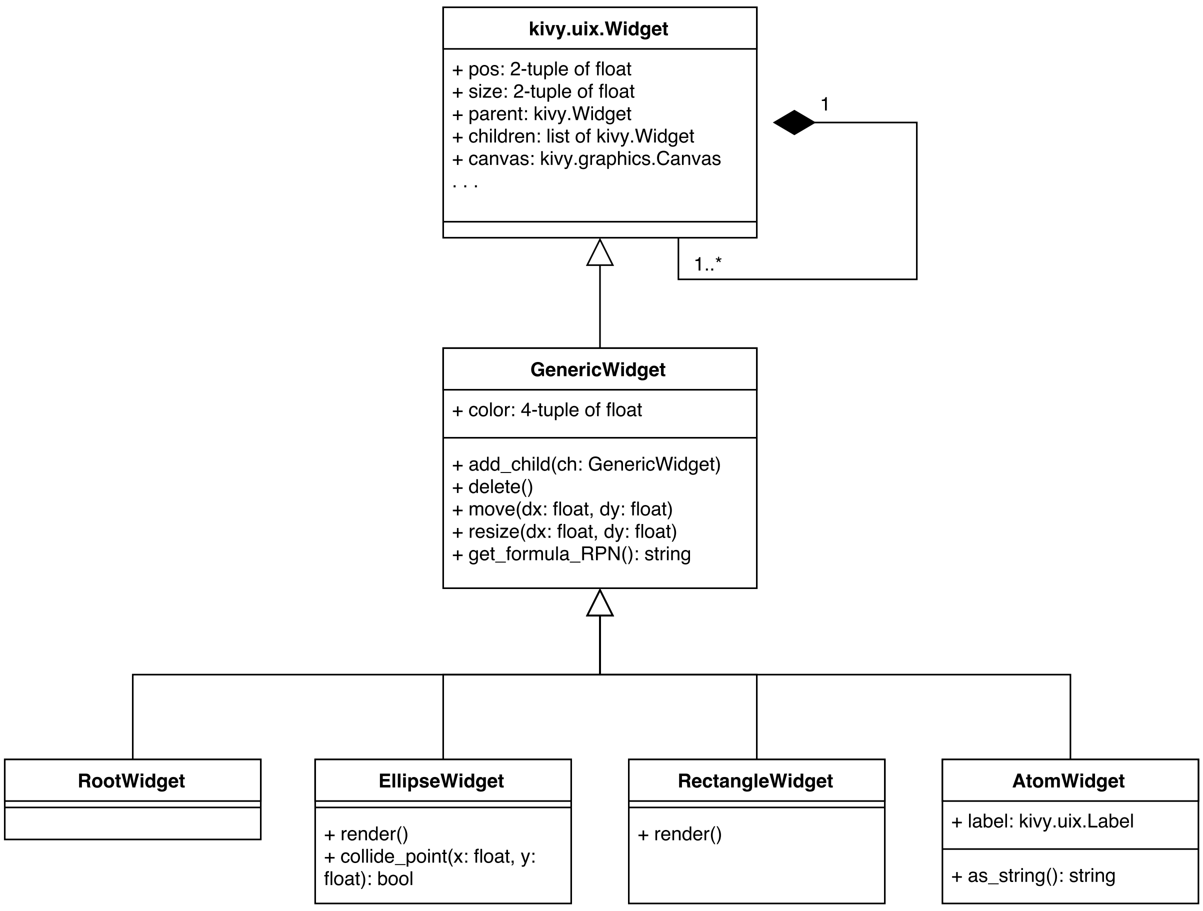

kivy.uix.Widget
+ pos: 2-tuple of float
+ size: 2-tuple of float
+ parent: kivy.Widget
+ children: list of kivy.Widget
+ canvas: kivy.graphics.Canvas
. . .
1
1..*
GenericWidget
+ color: 4-tuple of float
+ add_child(ch: GenericWidget)
+ delete()
+ move(dx: float, dy: float)
+ resize(dx: float, dy: float)
+ get_formula_RPN(): string
RootWidget
EllipseWidget
+ render()
+ collide_point(x: float, y: float): bool
RectangleWidget
+ render()
AtomWidget
+ label: kivy.uix.Label
+ as_string(): string

Figure D.1: Widget class diagram. Only relevant attributes and methods are showed.

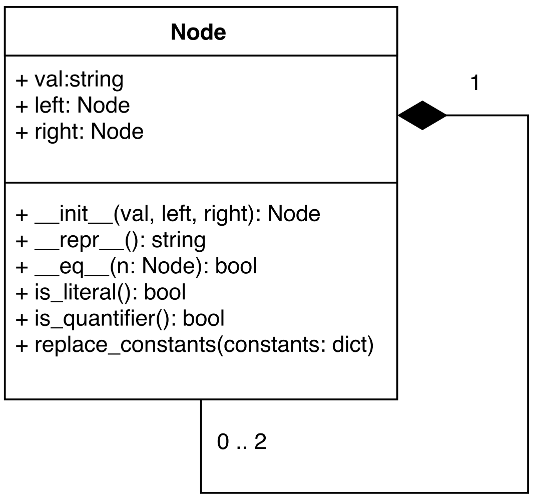

Node
+ val:string
+ left: Node
+ right: Node
+ __init__(val, left, right): Node
+ __repr__(): string
+ __eq__(n: Node): bool
+ is_literal(): bool
+ is_quantifier(): bool
+ replace_constants(constants: dict)
1
0 .. 2

Figure D.2

# Appendix E

# File formats

### E.0.1 Grasp sample file

```
#:kivy 1.0.9

#name: 'node', [True, False, False, False]
#name: 'edge', [True, True, False, False]
#name: 'reach', [True, True, False, False]
#name: 'in', [True, True, False, False]
#line: 0, {0: [1], 1: [0]}
#line: 1, {0: [1], 1: [0]}
#line: 2, {0: [1], 1: [0, 2, 3], 2: [1], 3: [1, 4], 4: [3]}
#line: 3, {0: [1], 1: [0, 2, 3], 2: [1], 3: [1, 4], 4: [3]}
#line: 4, {0: [1], 1: [0, 2], 2: [1]}
#line: 5, {0: [1], 1: [0]}
#line: 6, {0: [1], 1: [0, 2], 2: [1]}
#line: 7, {0: [1], 1: [0]}
#line: 8, {0: [1], 1: [0]}
#line: 9, {0: [1], 1: [0]}
#line: 10, {0: [1], 1: [0]}
#line: 11, {0: [1], 1: [0]}
#line: 12, {0: [1], 1: [0]}
#line: 13, {0: [1], 1: [0, 2], 2: [1]}
#line: 14, {0: [1], 1: [0]}

RootWidget:
    pos: [0, 0]
    size: [1008.95, 685.0]
    EllipseWidget:
        pos: [657.8585961700002, 142.59717790000002]
        size: [336.65452065000005, 196.0317628200001]
        NexusWidget:
            pos: [723.0999999999999, 269.71]
            size: [12, 12]
            line_info: [(13, 1)]
        AtomWidget:
            pos: [919.1199999999999, 242.17000000000002]
            size: [35, 16.2]
            text: 'node'
            line_info: [(14, 1)]
        AtomWidget:
            pos: [700.42, 240.55]
            size: [35, 16.2]
            text: 'node'
            line_info: [(13, 0)]
        EllipseWidget:
            pos: [769.27, 207.33999999999997]
            size: [107.73, 76.14000000000001]
            AtomWidget:
                pos: [803.29, 240.55]
                size: [39, 16.2]
                text: 'reach'
                line_info: [(13, 2), (14, 0)]
    EllipseWidget:
        pos: [355.04129497000014, 146.69249821000008]
        size: [254.77140843000015, 183.1536392400001]
        SquareWidget:
            pos: [427.45000000000005, 186.27999999999992]
            size: [123.12000000000006, 55.89000000000003]
            AtomWidget:
                pos: [463.9000000000001, 209.77]
                size: [39, 16.2]
                text: 'reach'
                line_info: [(10, 1), (12, 1)]
        AtomWidget:
            pos: [539.23, 264.04]
            size: [13, 16.2]
            text: 'in'
            line_info: [(11, 1), (12, 0)]
        AtomWidget:
            pos: [414.49, 264.04]
            size: [39, 16.2]
            text: 'reach'
            line_info: [(10, 0), (11, 0)]
```

Portion of a `Grasp` save file for Hamiltonian Cycles problem

### E.0.2 `Graphviz` sample file

```
graph {
  graph [pad=2 size=10]
  node [margin=0 fontsize=16 shape=none width=0 height=0]
  in1 [label="in"]
  in2 [label="in"]

  n1 -- in1:w
  in1:e -- n2
  n2 -- in2:w
  in2:e -- n1
  n3 -- in2:n
  in2:s -- n4
}
```

Graphviz sample file

# Glossary

***parse tree*** A data structure that consists of an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. 34

***stable model*** A solution to an ASP program. 5, 7–10, 15, 33, 34

**ASP** Answer Set Programming. v, vi, 3, 5–7, 9, 17, 30, 33, 34, 49, 52, 55, 59, 67

**EGs** Existential Graphs. v, 3, 10, 11, 13, 21, 33

**EqGs** Equilibrium Graphs. v, 3, 17, 19, 21, 23, 25, 27, 30, 31

**HT** *Here-and-There*. 17

**KR** Knowledge Representation. 1

**NNF** Negation Normal Form. 27, 29

**PNF** Prenex Normal Form. 27, 28

**POTASSCO** Potsdam Answer Set Solving Collection. 30, 31, 33

**QEL** Quantified Equilibrium Logic. v, 14, 15, 23, 27

**RPN** Reverse Polish Notation. 34

**satisfiable** A program that has one or more stable models as solution. 7

**UML** Unified Modeling Language. 61

**unsatisfiable** A program that has no stable models. 7

# Bibliography

[1] V. Marek and M. Truszczyński, *Stable models and an alternative logic programming paradigm*, pp. 169–181. Springer-Verlag, 1999.

[2] I. Niemelä, "Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3-4, pp. 241–273, 1999.

[3] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Communications of the ACM*, vol. 54, no. 12, pp. 92–103, 2011.

[4] J. F. Sowa, "Conceptual graphs for a data base interface," *IBM Journal of Research and Development*, vol. 20, no. 4, pp. 336–357, 1976.

[5] J. F. Sowa, "Conceptual graphs," in *Handbook of Knowledge Representation* (F. van Harmelen, V. Lifschitz, and B. W. Porter, eds.), vol. 3 of *Foundations of Artificial Intelligence*, pp. 213–237, Elsevier, 2008.

[6] J. F. Sowa, "From existential graphs to conceptual graphs," *IJCSSA*, vol. 1, no. 1, pp. 39–72, 2013.

[7] G. Peano, "Aritmetices principia nova methoda exposita," 1889. Torino: Bocca.

[8] C. S. Peirce, "Manuscripts on existential graphs," in *Collected Papers of Charles Sanders Peirce*, vol. 4, pp. 320–410, Harvard University Press, Cambridge, MA, 1906.

[9] D. Pearce, "A New Logical Characterisation of Stable Models and Answer Sets," in *Proc. of Non-Monotonic Extensions of Logic Programming (NMELP'96)*, (Bad Honnef, Germany), pp. 57–70, 1996.

[10] M. Gelfond and V. Lifschitz, "The Stable Model Semantics For Logic Programming," in *Proc. of the 5th International Conference on Logic Programming (ICLP'88)*, (Seattle, Washington), pp. 1070–1080, 1988.

[11] D. Pearce and A. Valverde, "Quantified equilibrium logic and foundations for answer set programs," in *Proc. of the 24th Intl. Conf. on Logic Programming, ICLP 2008, (Udine, Italy, December 9-13)*, vol. 5366 of *Lecture Notes in Computer Science*, pp. 546–560, Springer, 2008.

[12] A. Heyting, "Die formalen Regeln der intuitionistischen Logik," *Sitzungsberichte der Preussischen Akademie der Wissenschaften. Physikalisch-mathematische Klasse*, 1930.

[13] V. Lifschitz, "What is answer set programming?," in *AAAI*, vol. 8, pp. 1594–1597, 2008.

[14] S.-J. Shin, *The Iconic Logic of Peirce's Graphs.* Bradford Book, 2002.

[15] F. Aguado, P. Cabalar, D. Pearce, G. Pérez, and C. Vidal, "A denotational semantics for equilibrium logic," *Theory and Practice of Logic Programming*, vol. 15, no. 4-5, pp. 620–634, 2015.

[16] P. Cabalar, D. Pearce, and A. Valverde, "Reducing propositional theories in equilibrium logic to logic programs," *Lecture Notes in Computer Science*, vol. 3808, pp. 4–17, 2005.

[17] U. of Potsdam, "The Potsdam Answer Set Solving Collection." `https://potassco.org/`. Accessed: 2017-07-06.

[18] Python-Foundation, "The `Python` language." `https://www.python.org/`. Accessed: 2017-15-06.

[19] Kivy-Organization, "The `Kivy` library." `https://kivy.org/#home`. Accessed: 2017-15-06.

[20] Graphviz-Developers, "`Graphviz`." `http://www.graphviz.org/`. Accessed: 2017-15-06.

[21] U. of Calabria, "`ASPIDE`." `https://www.mat.unical.it/ricca/aspide/index.html`. Accessed: 2017-15-06.

[22] O. Cliffe, M. de Vos, M. Brain and J. Padget, "ASPVIZ: Declarative Visualisation and Animation Using Answer Set Programming," *Lecture Notes in Computer Science*, vol. 5366, 2008.

[23] C. Kloimüllner, J. Oetsch, J. Pührer and H. Tompits, "Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs," *Lecture Notes in Computer Science*, vol. 7773, 2013.

[24] T. U. of Wien, "`ARVis`." `http://www.dbai.tuwien.ac.at/research/project/arvis/index.html`. Accessed: 2017-15-06.

[25] B. van Heuveln, "`Peirce-Logic`." `http://peirce-logic.appspot.com/`. Accessed: 2017-15-06.

[26] "Collection of Conceptual Graphs software tools." `https://conceptualgraphs.org/`. Accessed: 2017-15-06.

[27] J. Lee and R. Palla, "`F2LP` - Computing Answer Sets of First Order Formulas," in *Proc. of International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-09*, 2009.

[28] F. Dau, "Constants and Functions in Peirce's Existential Graphs," in *Conceptual Structures: Knowledge Architectures for Smart Applications* (S. P. U. Priss and R. Hill, eds.), vol. 4604, pp. 429–442, 2007.