Modularization of Logic Programs

Alexandre Miguel Pinto[1] and Luís Moniz Pereira[2]

[1]Outra Limited UK, ORCID: 0000-0003-0577-0939 and
[2]NOVA LINCS, Universidade Nova de Lisboa, Portugal,
ORCID: 0000-0001-7880-4322

Abstract. Standard software and knowledge engineering best practices advise for modularity because, amongst other benefits, it facilitates development, debugging, maintenance, composition and interoperability. Knowledge bases written as Logic Programs are no exception, and their corresponding semantics should enable such modularity. In this paper we formally define several new syntactical notions and semantics properties that capture the notions of modularity and separation of concerns applied to the LPs domain. Furthermore, we set forth other notions necessary for top-down, call-graph oriented existential query answering with 2-valued semantics for LPs with Integrity Constraints.

Keywords: Modularization, Logic Programs, Credulous Reasoning, Properties, Semantics

1 Introduction

1.1 Context

Both in the academia and in the industry, development of intelligent software systems is becoming increasingly more frequent due to the need to offer systems and services that deliver more value to the end user. Larger and more distributed teams collaborate in the development of such systems, including the Knowledge Bases (KBs) they are built upon. In this paper we focus on the usage of Logic Programs (LPs) as the means to encode the KBs and the usual credulous and sceptical reasoning tasks as the mechanisms to solve the computational problem the system is intended to. LPs have been used successfully to represent and solve several kind of problems including combinatorial search, planning, abduction, diagnosis, constraint solving and many others.

The different kinds of problems and the respectively distinct intended usages of the LP-based systems require different reasoning mechanisms. Whenever the LP-based system is intended to allow the user to explore alternative scenarios a 2-valued semantics, e.g., Stable Models (SMs) [5], is the adequate choice for the LP as, in general, these allow for more than one model. Under this setting the individual models can represent the different scenarios. Credulous reasoning can then be used to find one, or more, of the individual models that satisfy a user's query. On the other hand, if the intended usage of the LP-base system is to provide irrefutable answers, and warranted knowledge to the user, then a

3-valued semantics, like the Well-Founded Semantics (WFS) [4], may be more adequate as these usually provide exactly one model — sceptical reasoning allows the user to find out what consequences necessarily follow from the KB and is commonly implemented as checking if the user's query is entailed by the single 3-valued (sceptical) model, and where Integrity Constraints (ICs), in the form of denials, can then be satisfied when their bodies are false but also if undefined [7].

Basic notions : We consider here the usual notions of alphabet, language, atom, literal, rule, and (logic) program. A literal is either an atom A or its default negation $not\ A$. We dub default literals (or default negated literals — DNLs, for short) those of the form $not\ A$. Without loss of generality we consider only ground Normal Logic Programs (NLPs), which are sets of Normal Logic Rules (NLRs) of the form $H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m,$ (with $m, n \geq 0$ and finite) where H, the B_i and the C_j are ground atoms. In conformity with the standard convention, we write rules of the form $H \leftarrow$ also simply as H (known as "facts"). An NLP P is called definite if none of its rules contain default literals. If r is a rule we denote its head H by $head(r)$, and $body(r)$ denotes the set $\{B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m\}$ of all the literals in its body. We write $\mathcal{H}_P$ to denote the Herbrand Base of P.

Besides containing normal rules as above, LPs may also include rules with a non-empty body and where the head is the special symbol $\bot$ which are known as a type of Integrity Constraints (ICs), specifically *denials*, and they are normally used to prune out unwanted models of the normal rules part. An LP is thus the union of a set of normal rules with a (possibly empty) set of ICs.

1.2 Motivation

The development of LP-based intelligent systems are software engineering projects and as these, their teams, and the KBs developed grow larger, the adoption of the best practices and principles of the software engineering discipline become indispensable if one wishes to guarantee certain qualities of the overall intelligent system. In particular, the development and usage of the LP-based KB part must itself be subject to compliance with those guidelines. In this paper we focus on the assurance of the principles of Modularity and Separation of Concerns in LP-based KBs. We will see in the sequel that adopting these principles and ensuring those qualities in an LP-based KB has a number of implications regarding the properties the particular LP semantics must comply with, depending on whether it is a 3-valued or a 2-valued one. Our main goal within this paper is precisely to contribute with the definitions of those formal properties of semantics for LPs which ensure Modularity and Separation of Concerns, and to provide with mechanisms to define such semantics.

The approach we follow in the remainder of the paper goes as follows. First, we recap the most common properties of semantics in the literature [3] that are related to the principles of Modularity and Separation of Concerns. Since the concept of Modularity is intrinsically related to the notion of modules, or components, and their interdependencies, we translate these dependencies notions to the domain of LPs and in that regard we recap the definitions in the literature, define two new notions of syntactic structure of an LP and compare them to the standard ones. We translate the concept of Separation of Concerns into the LP domain by making explicitly distinct the role of Normal Logic Rules and Integrity Constraints and show how this explicit distinction implies certain properties of the semantics for the Normal Logic Rules part. Then we characterize the family of semantics complying with the properties we defined, compare them to SMs, and show an algorithm to compute models of these semantics. Final remarks and future work conclude the paper.

2 Background Review

2.1 Modularity and Separation of Concerns

Modularity and Separation of Concerns (henceforth abbreviated as Mod. and SoC, respectively) are two of the central qualities required of software systems developed according to the best practices of software engineering. Amongst other benefits, a modular, i.e., a component-based, system is easier to develop, test, debug, maintain, to compose, and to interoperate with others. According to [8]

> *Component-based software engineering is a reuse-based approach to defining, implementing, and composing loosely coupled independent components into systems. A component is a software unit whose functionality and dependencies are completely defined by a set of public interfaces. Components can be composed with other components without knowledge of their implementation and can be deployed as an executable unit.*

From this definitions we can infer a module, or component, in such a "component-based" system, should be easily replaceable by another with the same functionality, as long as its interface and externally observable behavior remain the same; modules can also be independently developed and later put together to form the entire system. When translating these notions to LP-based KBs we need to define what the modules are so that they exhibit these high (internal) cohesion and low (external) coupling [6] characteristics. In section 3 we present new semantical properties and syntactic structure notions that will allows us to define such modules in LPs.

Also from [8] we learn that

> *The separation of concerns is a key principle of software design and implementation. It means that you should organize your software so that each element in the program (. . .) does one thing and one thing only. You can then focus on that element without regard for the other elements in the program. You can understand each part of the program by knowing its concern, without the need to understand other elements. When changes are required, they are localized to a small number of elements.*

From this definition it follows immediately that, in the LP domain (remember we are considering a LP to be the union of a NLP with a set of ICs), the normal rules in the NLP and the ICs are reifications of two very distinct Concerns: that of generating alternative scenarios, and that of filtering out the undesired candidates, respectively. Since it is the job of the ICs part to reject the bad candidates, by the SoC principle, the NLP part, or any subset of it, must not be allowed to prevent the existence of said candidates. Thus, compliance with the SoC principle implies the semantics for the NLP must guarantee model existence; to allow otherwise is to violate the SoC.

2.2 Semantics and Models

Taking the classical notions of (Herbrand) interpretation and model, [2] defines (def. 2.4) a semantics of LPs as follows:

> *A semantics SEM is a mapping from the class of all programs into the powerset of the set of all 3-valued Herbrand structures. SEM assigns to every program a set of 3-valued Herbrand models of P*

and also a sceptical entailment relation (def. 2.5) as:

> *Let P be a program and U a set of atoms. Any semantics SEM induces a sceptical entailment relation SEM^{scept} as follows:*

$$SEM^{scept}(U) := \bigcap_{\mathcal{M} \in SEM_P(U)} \{L : L \textit{ is a pos. or neg. literal with } \mathcal{M} \models L\}$$

where $SEM_P(U) = SEM_{P \cup U}$, the set of models of $P \cup U$ according to SEM. In the following we write $SEM(P)$ to denote the set of all models of P according to SEM, whereas $SEM^{scept}(P)$ still denotes the intersection of all such models. For LPs including ICs, every model $M \in SEM(P)$ is such that $\bot \notin M$.

In [2], and its subsequent paper [3], the author defines several properties of semantics, including Relevance, Cumulativity, Modularity, and many others, but all of these regard SEM^{scept}, i.e., the intersection of all models of $P \cup U$ according to SEM. When SEM is a 3-valued semantics, e.g., the WFS, SEM already provides a single model, so in that case the intersection of all models coincides with the unique model. When SEM is a 2-valued semantics this means those properties pertain to the literals in the intersection of all 2-valued models of the semantics; not to each individual 2-valued model. However, when we are interested in using the individual 2-valued-models, e.g. for answering existential queries, we need properties analogous to that of Relevance and Cumulativity, but pertaining to individual models and not to their intersection. We have found no such properties in the literature and so we provide them below as part of our contribution. In [1] the authors stress the importance of the Cumulativity property and define an alternative more credulous version of this property (dubbing it Extended Cumulativity, ECM for short). They also show that the SM semantics enjoys ECM although it does not enjoy cumulativity.

2.3 Syntactic Dependencies

In [3] the author introduced a notion of Modularity (def. 5.7) as a formal property of semantics for LPs. We recap it here for self-containment, but first we need to include other auxiliary syntactic notions.

Definition 1. ***Dependencies in a program.*** *In a LP P, a rule r_2 directly depends on r_1 (written as $r_2 \leftarrow r_1$) iff the head of r_1 appears, possibly negated, in the body of r_2; we say r_2 depends on r_1 ($r_2 \twoheadleftarrow r_1$) iff either r_2 directly depends on r_1 or r_2 directly depends on some other rule r_3 which in turn depends on r_1.*

We also consider the other combinations of (direct) dependencies amongst atoms and rules, and use the same graphical notation ($\leftarrow, \twoheadleftarrow$) to denote (direct, indirect) dependency. Rule r directly depends on an atom a iff a appears, possibly negated, in the body of r; and r depends on a iff either r directly depends on a or r depends on some rule r' which directly depends on a. An atom a directly depends on rule r iff $head(r) = a$; and a depends on r iff either a directly depends on r or a directly depends on some rule r' such that r' depends on r. An atom b directly depends on atom a iff a appears (possibly default negated) in the body of a rule with head b, and b depends on a iff either b directly depends on a, or b directly depends on some rule r which depends on a.

In [3] Dix introduces the notion of *relevant rules*, which we restate here adapted to our notation.

Definition 2. ***Sub-program Relevant for Atom.*** *Let P be a NLP and a an atom of P. We write $Rel_P(a)$ to denote the set of rules of P which are relevant and enough for determining a's truth value. Formally, $Rel_P(a) = \{r \in P : a \text{ depends on } r\}$.*

Also in [3] we find the notion of Program Reduction (def. 3.8) which is similar, but not exactly equal, to the Gelfond-Lifschitz program division, and which will be necessary to the Modularity notion.

Definition 3. ***P reduced by M (def. 3.8 of [3], adapted to our notation).*** *Let P be a program and M be a set of literals. "P reduced by M" is the program $P^M := \{r^M : r \in P \text{ and } (body(r) \cup M) \text{ is a consistent set of literals}\}$, where $body(r^M) = body(r) \setminus M$.*

Now that we have the notions of Relevant Part and P reduced by M we can recap the notion of Modularity from [3].

Definition 4. ***Modularity (def. 5.7 of [3] adapted to meet our notation).*** *Let $P = P_1 \cup P_2$ be instantiated and for every $A \in \mathcal{H}_2 : Rel_P(A) \subseteq P_2$. The principle of Modularity is: $SEM^{scept}(P) = SEM^{scept}(P_1^{SEM^{scept}(P_2)} \cup P_2)$.*

These syntactical and semantical notions do not capture all the various aspects of the Modularity and Separation of Concerns of software engineering principles applied to LPs. For this reason we now introduce, as part of our contribution, the new ones we find necessary for that purpose.

3 New notions and properties

The concept of Modularity is intrinsically related to the notion of modules, or components, and their interdependencies, and in order to provide a rendering of that concept in the LP domain we need to translate these dependencies into syntactic features of LPs. The relevant rules (def. 2) syntactic notion does part of this job but it still does not capture all the characteristics of a module. We introduce below the formal notion of a Module as well as some of its syntactic properties. Also, the semantical notion of modularity in def. 4, besides being insufficient to fully grasp the Modularity concept applied to LPs, and like all other semantical properties in [2] and [3], regards only the intersection of models, and for that reason is suitable only for sceptical reasoning purposes. Since in our work we are especially interested in existential query answering with 2-valued semantics, we also provide new definitions of credulous reasoning oriented semantical properties that capture the various aspects of Modularity.

As stated before, regarding the concept of Separation of Concerns, it translates into the LP domain by making explicitly distinct the role of Normal Logic Rules from that of Integrity Constraints, and noticing this explicit distinction entails the property of guarantee of model existence for the semantics for the Normal Logic Rules part.

Finally, we introduce new notions supporting existential query answering with LPs, both syntactic and semantical, which allow us to formally compose a comprehensive framework for credulous reasoning with LPs (including ICs) with semantics that comply with both the Modularity and Separation of Concerns principles.

3.1 Modularity in Logic Programs

In a modular system, the components, or modules, have high internal cohesion (the elements inside the module are tightly related), and low external coupling (the elements from two distinct modules are lightly, if at all, related). The modularity semantical property in def. 4 does not capture all these requirements associated with the Modularity principle.

In LPs we only have logic rules and the only dependency notion we can find is a syntactical one. By taking the transitive closure over this syntactic dependency, the *relevance* in def. 2 captures a part of the *module* concept according to the description above, but not all of it. For this reason, we set forth a more encompassing notion of *module* and examine some of its properties.

Definition 5. ***Modules of a Logic Program.*** *Let P and P_1 be LPs such that $P_1 \subseteq P$. P_1 is said to be a* module *of P iff $\forall_{a \in \mathcal{H}_{P_1}} Rel_P(a) \subseteq P_1$. I.e., a module of P is any subset of rules of P that contains all, and only, the rules relevant to the atoms inside the module.*

Let P_1 and P_2 be modules of P. We say P_1 is nested inside P_2 iff $P_1 \subseteq P_2$. In this case we also say P_1 is a sub-module *of P_2.*

We say two modules P_1 and P_2 of P are independent *iff they do not share any atoms, i.e., $\mathcal{H}_{P_1} \cap \mathcal{H}_{P_2} = \emptyset$.*

It follows from this definition that if modules P_1 and P_2 are independent, then every sub-module of P_1 is independent from every sub-module of P_2. From a system-wide analysis perspective, it might be of interest to identify the unique set of maximal (w.r.t. set-inclusion) independent modules of a given program P — we denote this set by $MIM(P)$.

With the above definition, the "components" inside modules (the individual rules) are necessarily highly correlated, by virtue of syntactic dependency, thus embodying the high internal cohesion demanded of modules. On the other hand, the independent modules notion fully captures the low external coupling by virtue of their syntactical independence.

Example 1. **Modules in a program.** Let P be

$$\begin{aligned} c &\leftarrow not\ a \\ a &\leftarrow not\ b \\ b &\leftarrow not\ a \\ x &\leftarrow y \end{aligned}$$

The pair of rules $a \leftarrow not\ b$ and $b \leftarrow not\ a$ form a module P_1 of P. $P_2 = P_1 \cup \{c \leftarrow not\ a\}$ is another module, $P_3 = \{x \leftarrow y\}$ is yet another module, and the whole program is also considered to be a module. P_1 is nested inside P_2, and every module of P is nested inside P. In this example, P_2 and P_3 are *independent*, and so are necessarily P_1 and P_3 as well.

As stated above, the Modularity property defined in [3] pertains to the sceptical entailment of a semantics SEM, i.e., when taking a 2-valued semantics, this property is defined only over the intersection of all its models for a given program. In our work, since we are intent on performing credulous reasoning with a 2-valued semantics, we need a corresponding credulous version of modularity, one that concerns each indvidual 2-valued-model, and not just the intersection of all models. Hence, we introduce now several new semantical properties that will be used to build our credulous modularity property.

Definition 6. ***Credulous Module Replaceability.*** *Let P_1, P_2 and P_x be LPs such that P_1 is a module of $P_x \cup P_1$ and P_2 is a module of $P_x \cup P_2$, with $\mathcal{H}_{P_1} = \mathcal{H}_{P_2}$, and let SEM be a 2-valued semantics for LPs. When $SEM(P_1) = SEM(P_2)$ — in which case we say P_1 and P_2 are $SEM-$equivalent — we say SEM enjoys* Credulous Module Replaceability *iff $SEM(P_x \cup P_1) = SEM(P_x \cup P_2)$.*

Intuitively this means one can replace one module of a program with another as along as they have exactly the same models, all the while preserving the models of the global program. This notion intends to capture the idea of functional implementation independence of modules as far as interface and meaning are preserved, which is characteristic of modular systems.

The following two notions (Credulous Monotony and Cartesian Product) capture the black-box view on modules which allows the rapid composition of a prototypical system by knowing the possible behaviors of its composing modules.

Definition 7. ***Credulous Monotony.*** *Let P be an LP and P_1 a module of P. A 2-valued semantics SEM is said to enjoy* Credulous Monotony *iff*

$$\forall_{M_1 \in SEM(P_1)} \{M : M \in SEM(P) \wedge M \supseteq M_1\} = SEM((P \setminus P_1) \cup M_1)$$

Intuitively this means one can replace one module of the program by any one of its models, and rest assured that the models of the resulting program are exactly those models of the original program that set-included the model which was used to replace the module.

The Stable Models semantics fails this property as the following example shows.

Example 2. **Stable Models fail Credulous Monotony.** Let P be

$$\begin{array}{l} a \leftarrow not\ b \\ b \leftarrow not\ a \\ c \leftarrow not\ c, not\ a \end{array}$$

P has $\{a\}$ as its unique SM. The rules for a and b form a module P_1 of P which has two SMs: $\{a\}$, and $\{b\}$. If we replace P_1 by its model $\{b\}$ we obtain the program $P' =$

$$\begin{array}{l} b \\ c \leftarrow not\ c, not\ a \end{array}$$

which has no SMs at all, thus showing the failure of SM semantics regarding Credulous Monotony.

Definition 8. ***Cartesian Product.*** *Let P_1 and P_2 be independent modules of $P_1 \cup P_2$, and SEM a 2-valued semantics for LPs. SEM is said to enjoy the* Cartesian Product *property iff*

$$SEM(P_1 \cup P_2) = \{M_1 \cup M_2 : M_1 \in SEM(P_1) \wedge M_2 \in SEM(P_2)\}$$

I.e., models of unions of independent modules are unions of models of the individual modules. If $\#SEM(P_1) = n$ and $\#SEM(P_2) = m$ then $\#SEM(P_1 \cup P_2) = nm$, hence the name *Cartesian Product.*

Definition 9. ***Credulous Modularity.*** *SEM is said to enjoy Credulous Modularity iff it enjoys all four properties of Credulous Module Replaceability, Credulous Monotony, Cartesian Product, and Model Existence (i.e., $\#SEM(P) \geq 1$ for any given NLP P).*

This definition considers the concept of Credulous Modularity as including the notion of Separation of Concerns (by demanding Model Existence for NLPs) as one of its characteristics.

3.2 A framework for credulous reasoning with LPs

Credulous reasoning with LPs amounts to finding if there is some model M of the program P at hand that satisfies some user-specified criteria Q. This can either take the form of finding/computing one such model (if it exists), or finding/computing one sub-model of it (i.e. subset of a model) sufficient to answer the user's query. The former is a common way to, e.g., address combinatorial search problems, while the latter is more commonly used in top-down query-answering *a la* Prolog.

In our work we focus on the latter approach which is only realizable with semantics where the truth value of atoms in any given model depends only on their relevant rules. The Relevance notion in def. 2 pertains only to the atoms in the intersection of all models. What we need here is a "per-model" version of the Relevance notion. We put if forward now.

Definition 10. ***Credulous Relevance.*** *Let P be an NLP. SEM is Credulously Relevant iff*

$$\forall_{a\in\mathcal{H}_P}\Big(\forall_{M\in SEM(P)} a\in M \Rightarrow \big(\exists_{M_a\in SEM(Rel_P(a))} M_a\subseteq M \wedge a\in M_a\big)\Big)$$
$$\wedge$$
$$\Big(\forall_{M_a\in SEM(Rel_P(a))}\ \exists_{M\in SEM(P)} M_a\subseteq M\Big)$$

I.e., in a Credulously Relevant semantics, an atom is *true* in *some* model of the whole program iff it is *true* in *some* sub-model of the part of the program *relevant* to the atom, where that sub-model is a subset of a model for the whole program where the atom is *true.*

This notion, however, is applicable only to NLPs, but not whole LPs (which may include ICs). When finding an existential answer to a query in a LP, it might be the case that a candidate answer found may turn out to be rejected by some IC. This means we need another notion of relevance that is applicable to LPs with ICs. We now present this notion, preceded by other auxiliary ones.

Definition 11. ***Sub-program Influenced by Atom.*** *Let P be a LP. We say atom $a\in\mathcal{H}_P$* influences *rule $r\in P$ iff r* depends *on a. We write $Infl_P(a)$ to denote the set of such r, i.e., $Infl_P(a)=\{r\in P : r \twoheadleftarrow a\}$.*

Definition 12. ***Constraint Directly Relevant Atoms.*** *Let $P=NLP\cup ICs$ be a LP composed of the set of normal rules NLP and the set of ICs ICs, and $S\subseteq\mathcal{H}_P$ a subset of atoms of P. The set of atoms of P which are Constraint Directly Relevant for S contains exactly all the atoms relevant for the ICs in P which are influenced by the atoms in the Relevant part of P for any atom in S.*

Due to its complexity, we breakdown this definition in intermediate steps as follows. First we take each atom a of S and obtain the Relevant part of P for it. Taking the union over all such atoms of S we obtain all the atoms of P relevant for any atom in S, i.e.,

$$\bigcup_{a\in S} Rel_P(a)$$

Let us abuse notation are denote this set by $Rel_P(S)$. Next we take all the atoms in $Rel_P(S)$, i.e., $\mathcal{H}_{Rel_P(S)}$, and for each we find the rules of P which it influences, thus obtaining

$$\bigcup_{b \in \mathcal{H}_{Rel_P(S)}} Infl_P(b)$$

We abuse notation again and denote this set by $Infl_P(S)$, and now we find which ICs are included in this set of influenced rules, i.e., $Infl_P(S) \cap ICs$, denoted by $ICInfl_P(S)$. Finally, we take all atoms in the rules of P which are relevant to the atoms in $ICInfl_P(S)$, to obtain the set of Constraint Directly Relevant Atoms*,*

$$ICDirRel_P(S) = \mathcal{H}_{(\bigcup_{c \in \mathcal{H}_{ICInfl_P(S)}} Rel_P(c))}$$

Definition 13. ***Constraint Relevant Atoms.*** *Let $P = NLP \cup ICs$ be a LP composed of the set of normal rules NLP and the set of ICs ICs, and $S \subseteq \mathcal{H}_P$ a subset of atoms of P. The set of atoms of P which are Constraint Relevant for S, denoted by $ICRel_P(S)$ is S^ω, where*

$$\begin{aligned} S^0 &= S \\ S^{i+1} &= S^i \cup ICDirRel_P(S^i) \\ S^\alpha &= \textstyle\bigcup_{\beta<\alpha} S^\beta \end{aligned}$$

Definition 14. ***Credulous Constraint Relevance.*** *Let P be a LP. SEM is Credulously Constraint Relevant iff*

$$\forall_{\substack{a \in \mathcal{H}_P \\ M \in SEM(P)}} a \in M \Rightarrow \Big(\exists_{M_a \in SEM(Rel_P(ICRel_P(\{a\})) \cup ICInfl_P(\{a\}))} M_a \subseteq M \wedge a \in M_a \Big)$$

I.e., in a Credulously Constraint Relevant semantics, if an atom is *true* in *some* model of the whole program then it is *true* in *some* sub-model of the part of the program *constraint relevant* to the atom, where that sub-model is a subset of a model for the whole program where the atom is *true*.

These definitions set forth a theoretical framework upon which formal existential query answering methods can be developed for LPs, including ICs, with a 2-valued semantics.

Definition 15. ***Credulous Constraint Relevant Knowledge Existential Answer to a Query.*** *Let $P = NLP \cup ICs$ be a LP composed of the set of normal rules NLP and the set of ICs ICs, and Q a set of literals formed with atoms from $\mathcal{H}_P$ dubbed the* user's query*.*

M_Q is a credulous constraint relevant knowledge existential answer to query Q according to SEM iff

$$M_Q \in SEM(Rel_P(ICRel_P(|Q|)) \cup ICInfl_P(|Q|)) \text{ and } M_Q \supseteq Q$$

where $|Q|$ denotes the set of atoms in the literals in Q, i.e.,

$$|Q| = \{q : q \in Q \vee not\ q \in Q\}$$

The existence of a Credulous Constraint Relevant Knowledge Existential Answer to a Query M_Q does not necessarily guarantee the existence of a model M of P such that $M \supseteq M_Q$, but only because independent ICs might prevent model existence at all. However, the credulous constraint relevance property still ensures yet another local degree of modularity which might be used to focus the scope of rules considered when answering an existential query.

4 Conclusions and Future Work

We have taken the concepts of Modularity and Separation of Concerns from the software engineering discipline and applied them to the Logic Programs domain. As a result we devised a set of semantical properties, and auxiliary syntactical notions, that a 2-valued semantics for LPs must comply with in order to ensure the LP respects those Modularity and SoC principles. We have provided new notions of relevance and modularity that extend the ones in the literature in two ways: by being applicable to individual models of a 2-valued semantics instead of just to their intersection, and by taking into account also the possible ICs in the LP. Future work includes the definition of a 2-valued semantics complying with all these properties, and respective implementations.

5 Acknowledgements

L.M.P. is supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT- Fundação para a Ciência e a Tecnologia, Portugal, through national funds.

References

1. Stefania Costantini, Gaetano Aurelio Lanzarone, and Giuseppe Magliocco. Layer supported models of logic programs. In Michael Maher, editor, *Procs. 1996 Joint International Conference and Symposium on Logic Programming (JICSLP 1996)*, pages 438–452, Cambridge, USA, 1996. MIT Press.
2. Jürgen Dix. A Classification Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundamenta Informaticae*, 22(3):227–255, 1995.
3. Jürgen Dix. A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, 22(3):257–288, 1995.
4. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
6. M. Papazoglou and J. Yang. Design methodology for web services and business processes. *Technologies for E-Services*, 2444:175–233, 2002.
7. L. M. Pereira, J. N. Aparicio, and J. J. Alferes. Hypothetical reasoning with well founded semantics. In B. Mayoh, editor, *Scandinavian Conference on Artificial Intelligence: Proc. of the SCAI'91*, pages 289–300. IOS Press, Amsterdam, 1991.
8. Ian Sommerville. *Software Engineering 9.* Pearson Education, 2011.