A P2P framework for modular disjunctive knowledge bases with negation and constraints and its applications to the Semantic Web

Anastasia Analyti

Institute of Computer Science, FORTH-ICS, Greece
analyti@ics.forth.gr

Abstract. We present a P2P framework based on asynchronous messaging for collaborative reasoning over modular disjunctive knowledge bases with weak negation, strong negation, and constraints. In particular, we present algorithms running upon receival from a knowledge base of a corresponding *Ask_pred*, *Tell_pred*, *Ask_KB*, or *Tell_KB* message, which are needed for query answering. In particular, these messages collect in the knowledge base that received the query the corresponding part of the modular knowledge base that is needed for query answering. This process is done in polynomial time with respect to the size of the modular knowledge base. We present several applications of our theory, including Restricted Modular ERDF Ontologies, SPARQL 1.1 seen as a Rule Language, the Maximal Weak Model Semantics, the Minimal Weak Model Semantics, Multi-context Reasoning, the Contextually Closed Semantics, and Taxonomy-based Sources connected by Articulations. Several complexity results are provided.
Keywords: modular disjunctive knowledge bases with negation and constraints, P2P asynchronous messaging, applications to the Semantic Web

1 Introduction

In this paper, we consider a framework of knowledge bases, each associated with an IRI (its name) and a disjunctive logic program with weak ($\sim$) and strong ($\neg$) negation, as well as constraints. These knowledge bases are distributed over the web and interconnected through qualified literals in the bodies of their logic program rules. Each qualification of a literal corresponds to the IRI of a remote knowledge base and qualified literals correspond to literals of remote knowledge bases. A set of such knowledge bases forms a *modular knowledge base* $\mathcal{K}$, assuming that all qualifications appearing in $\mathcal{K}$ identify a knowledge base in this set. We define the semantics of a modular knowledge base and provide algorithms for querying answering based on an exchange of messages in a P2P fashion.

The *dependencies* of a knowledge base KB, denoted by $D^{\mathcal{K}}_{KB}$, is the minimal set of knowledge bases such that (i) $KB \in D^{\mathcal{K}}_{KB}$ and (ii) if a qualification appears in a knowledge base in $D^{\mathcal{K}}_{KB}$ then this identifies a knowledge base in $D^{\mathcal{K}}_{KB}$. The *semantics* of a knowledge base KB is defined as the union, $P^{\mathcal{K}}_{KB}$, of all logic programs of the knowledge bases in $D^{\mathcal{K}}_{KB}$, where predicates with a common name in the different knowledge bases are appropriately renamed to become distinct and qualifications are removed from the qualified literals. In particular, when a query q is posed to a knowledge base KB, only relevant parts from the knowledge bases in $D^{\mathcal{K}}_{KB}$ that are needed for query answering based on the Answer Set Semantics [26] are retrieved. These consist of (i) the parts relevant to the query and (ii) the parts that are needed to derive that $P^{\mathcal{K}}_{KB}$ is inconsistent (that is, that $P^{\mathcal{K}}_{KB}$ does not have a desired model, or in other words an Answer Set [26]).

We would like to note that a weakly negated literal $\sim L$, appearing in $P^{\mathcal{K}}_{KB}$, holds iff literal L is not satisfied by any Answer Set of $P^{\mathcal{K}}_{KB}$. On the other hand, a strongly

negated literal $L = \neg A$, appearing in $P^{\mathcal{K}}_{KB}$, holds iff L is satisfied by all Answer Sets of $P^{\mathcal{K}}_{KB}$. If a constraint $\texttt{false} \leftarrow L_1, ..., L_n, \sim L'_1, ..., \sim L'_m$, appears in $P^{\mathcal{K}}_{KB}$ then, according to Answer Set Semantics, *only* the Answer Sets that do not satisfy the formula $L_1 \wedge ... \wedge L_n \wedge \sim L'_1 \wedge ... \wedge \sim L'_m$ are considered in defining the semantics of $P^{\mathcal{K}}_{KB}$. In other words, all considered Answer Sets should either (i) do not satisfy at least one of the literals L_i, for $i = 1, ..., n$, or (ii) satisfy at least one of the literals L'_i, for $i = 1, ..., m$.

For query answering, all communication between the distributed knowledge bases is based on *asynchronous messaging*. In particular, *Ask_pred* messages are send and the corresponding *Tell_pred* messages are returned all in asynchronous time. Through these messages, the related to query answering, sub-part of $P^{\mathcal{K}}_{KB}$ is received by our main algorithm in *polynomial time* with respect to the size of $P^{\mathcal{K}}_{KB}$. Additionally, we use the asynchronous messages *Ask_KB* and *Tell_KB*. Through these messages, the parts of $P^{\mathcal{K}}_{KB}$ that are needed for deciding if $P^{\mathcal{K}}_{KB}$ is inconsistent are retrieved from the knowledge bases in $D^{\mathcal{K}}_{KB}$. Again, these parts are received by our main algorithm in *polynomial time* with respect to the size of $P^{\mathcal{K}}_{KB}$.

Our framework has applications on the Semantic Web, where datasets are expressed in the web ontology language RDFS [29] and published in RDF graph stores. Through the use of the Semantic Web query language SPARQL 1.1 [27], information from several RDF graph stores can be combined and queried based on monotonic and non-monotonic constructs. However, SPARQL 1.1 does not define the notion of recursive RDF graph view whose advantages have been defended in Network Graphs [39] and SPARQL 1.1 seen as a Rule Language [36]. RDF graph views are defined through named SPARQL 1.1 *construct* statements and allow one to be constructed from a query to another RDF graph view creating possible recursive statements. In Network graphs [39], the semantics of recursive RDF graph views is defined based on the well-founded semantics [22], while in SPARQL 1.1 seen as a Rule Language [36], the semantics is defined using the stable model semantics [24]. Note that both frameworks [39] and [36] do not faithfully extend RDFS semantics [29]. Instead, they only consider the RDF triples in the RDF stores.

In [2], the *simple modular ERDF ontologies* are defined as a set of distributed simple `r`-ERDF ontologies. A simple `r`-ERDF ontology O contains: (i) import statements, (i) an ERDF graph G_O, and (iii) a set of derivation rules P_O with qualified literals and $\sim$, $\neg$, $\wedge$ in the bodies of the rule. An ERDF graph G_O is an extension of an RDF graph with possible negative information through the strong negation ($\neg$) operator. Note that qualifications in the derivation rules are IRIs of other simple `r`-ERDF ontologies and allow accessing of remote information. In simple modular ERDF ontologies, query answering under the modular ERDF stable model semantics reduces to query answering under the Answer Set Semantics [25]. Modular ERDF stable model semantics faithfully extends RDFS [30, 28].

In this paper, we define the *restricted modular ERDF ontologies*, which is a special case of the simple modular ERDF ontologies, by imposing the restriction that if $[\neg]\texttt{rdf:type}(o, c)$ appears in a rule of a knowledge base then c should not be a variable but only an IRI.

The recursive RDF graph views, as defined in [36] and the framework of restricted modular ERDF ontologies can be implemented using the algorithms presented in this paper. This way only the relevant to the query parts will be transferred to the query node for query answering under the Answer Set Semantics. Similar is the case, for the frameworks of the Maximal Weak Model Semantics [9, 11], the Minimal Weak Model Semantics [10, 11], Multi-context Reasoning [7], the Contextually Closed Semantics of modular knowledge bases [35], and Taxonomy-based Sources connected by Articulations [33]. All these applications are reviewed in Section 4.

The rest of the paper is organized as follows: In Section 2, we define the modular disjunctive knowledge bases and the answers to a query sent to a knowledge base which is a member of a modular disjunctive knowledge base. In Section 3, we present the algorithms of our theory, providing query answering. In Section 4, we present several applications of our theory, including the frameworks of Restricted Modular ERDF Ontologies, SPARQL 1.1 seen as a Rule Language, the Maximal Weak Model Semantics, the Minimal Weak Model Semantics, Multi-context Reasoning, the Contextually Closed Semantics, and Taxonomy-based Sources connected by Articulations. Section 5 reviews other related work and Section 6 concludes the paper. Appendix A provides the proofs of some Theorems and Propositions (for the rest, intuitions are provided). Appendix B provides a List of Symbols. The electronic Appendix:
`http://users.ics.forth.gr/~analyti/Papers_2/Appendix_Modular_KBs.pdf`
provides an overview of required definitions of the simple modular ERDF ontologies (provided also in [2]) for helping the reader.

2 Modular Disjunctive Knowledge Bases

In this section, we define the modular disjunctive knowledge bases and the answers to a query sent to a knowledge base which is a member of a modular disjunctive knowledge base.

We denote by *IRI* the set of IRI references. Additionally, we denote by $\mathcal{KB}_{\texttt{nam}} \subseteq \mathcal{IRI}$ the set of knowledge base names. A term t is a variable or a constant. In our definitions, we consider a set of predicate names.

Below, we provide the necessary definitions.

Definition 1 (literal). A *literal* is a qualified or non-qualified literal defined as follows:

- A *non-qualified literal* is an expression $L = [\neg]pred(\bar{t})$, where *pred* is a predicate name and $\bar{t}$ is a sequence of terms.
- A *qualified literal* is an expression $L = [\neg]pred(\bar{t})@qual$, where *pred* is a predicate name, $\bar{t}$ is a sequence of terms, and $qual \in \mathcal{KB}_{\texttt{nam}}$. The knowledge base name *qual* is called the *qualification* of L. □

Definition 2 (disjunctive rule). A *disjunctive rule* r has the form:

$$L'_1 \vee ... \vee L'_n \leftarrow [\sim]L_1 \wedge ... \wedge [\sim]L_m, \text{ where}$$

- L'_i, for $i = 1, ..., n$, are non-qualified literals, or $n = 1$ and $L'_1 = \texttt{false}$, and
- L_i, for $i = 1, ..., m$, are literals, or $m = 1$ and $L_1 = \texttt{true}$

We define $Concl(r) = L'_1 \vee ... \vee L'_n$ (called *conclusion*) and $Cond(r) = [\sim]L_1 \wedge ... \wedge [\sim]L_m$ (called *condition*).

The disjunctive rule r is called *safe* iff in the case that a literal L (i) appears in $Cond(r)$ weakly negated or (ii) appears in $Concl(r)$ then every variable of L appears in a non-weakly negated literal of $Cond(r)$. □

Definition 3 (constraint). A disjunctive rule r is also called *constraint* if it is of the form:

$$\texttt{false} \leftarrow [\sim]L_1 \wedge ... \wedge [\sim]L_m \ \square$$

Definition 4 (disjunctive logic program). A *disjunctive logic program* P is a set of *safe* disjunctive rules. □

Definition 5 (knowledge base). A *knowledge base* is a pair $KB = \langle Nam_{KB}, P_{KB} \rangle$, $Nam_{KB} \in \mathcal{KB}_{\mathtt{nam}}$ and P_{KB} is a disjunctive logic program. □

Definition 6 (modular knowledge base). A *modular knowledge base* $\mathcal{K}$ is a set of knowledge bases with the constraint: $\forall KB \in \mathcal{K}$ and $\forall r \in P_{KB}$, if $[\neg]p(\bar{t})@Nam_{KB'}$ appears in r then $KB' \in \mathcal{K}$. □.

In particular, Nam_{KB} is the IRI through which the knowledge base is accessible. Additionally, we would like to note that safety of the rules appearing in the knowledge bases of $\mathcal{K}$ is needed for the correctness of our algorithms.

Example 1. Consider the modular knowledge base $\mathcal{K} = \{KB_1, KB_2, KB_3, KB_4\}$, shown in Figure 1[1]. Knowledge base KB_1, with Nam_{KB_1} =`<http://www.person_info>`, provides the sex and age of person, as well as other personal information. Knowledge base KB_2, with Nam_{KB_2} =`<http://www.health_conditions>`, provides information about health conditions of persons. Knowledge base KB_3, with Nam_{KB_3} = `<http://www.possible_illness>`, provides rules that based on the sex, age and health conditions of persons identify possible illnesses of these persons. Similarly, knowledge base KB_4, with Nam_{KB_4} =`<http://www.recommended_doctors>`, associates illnesses with doctors and provides, among others, a rule that based of the possible illnesses of a person suggests doctors that he/she should visit. □

Convention: In rest of the paper by $\mathcal{K}$, we will denote a modular knowledge base as MKB.

Let $KB \in \mathcal{K}$. Below we define the dependencies of KB w.r.t. $\mathcal{K}$. These are the knowledge bases in $\mathcal{K}$ on which KB depends, because KB rules query information from them directly or indirectly.

Definition 7 (dependencies of an knowledge base w.r.t. an MKB). Let $KB \in \mathcal{K}$. The *dependencies* of KB w.r.t. $\mathcal{K}$, denoted by $D^{\mathcal{K}}_{KB}$, is the minimum set of knowledge bases s.t.: (i) $KB \in D^{\mathcal{K}}_{KB}$ and (ii) if $KB' \in D^{\mathcal{K}}_{KB}$ and there exists a qualified literal $[\neg]pred(.)@Nam_{KB''}$ in $Cond(r)$, for an $r \in P_{KB'}$, then $KB'' \in D^{\mathcal{K}}_{KB}$. □

Example 2. Consider the modular knowledge base $\mathcal{K}$ of Example 1. It holds that: (i) $D^{\mathcal{K}}_{KB_2} = \{KB_2\}$, (ii) $D^{\mathcal{K}}_{KB_1} = \{KB_1, KB_2\}$, and (iii) $D^{\mathcal{K}}_{KB_3} = \{KB_1, KB_2, KB_3\}$. Item (i) is true because knowledge base KB_2 does not use any predicate from any other knowledge base, item (ii) is true because knowledge base KB_1 uses predicates `Male` and `Female` from knowledge base KB_2, and finally item (iii) is true because knowledge base KB_3 uses knowledge bases KB_1 and KB_2 in the body of its rules. □

Before we define the logic program of a knowledge base KB w.r.t a modular knowledge base $\mathcal{K}$, we provide an auxiliary definition.

Let $KB \in \mathcal{K}$ and let $r \in P_{KB}$. We denote by $tr(r)$ the rule derived from r if:

1. we replace each predicate p appearing in a non-qualified literal in r by p_Nam_{KB},
2. we replace each predicate appearing in a qualified literal in r with qualification *qual* by p_qual, and
3. we remove all qualifications from the qualified literals in r.

Below, we define the logic program of KB w.r.t. $\mathcal{K}$ which will be used for defining the *stable answers of a query q* to a knowledge base KB.

[1] Following usual convention, we have replaced $\wedge$ by "," in the program rules. Additionally, for simplification, we have removed the namespaces from the IRIs.

Knowlege Base KB_1

$\langle http://www.person\_info \rangle$

P_{KB_1} =

Female(?x) ← Female(?x)@⟨*http://health.conditions*⟩.
Male(?x) ← Male(?x)@⟨*http://health.conditions*⟩.
Person(?x) ← Female(?x).
Person(?x) ← Male(?x).
Female(?x) ← ¬Male(?x).
Male(?x) ← ¬Female(?x).
Male(Peter).
age(Peter, 40).
age(Mary, 65).
false ← Male(?x), Female(?x).
... /* KB rules defining other properties */

Knowlege Base KB_2

$\langle http://www.health\_conditions \rangle$

P_{KB_2} =

Female(?x) ← Menopausal(?x).
Female(?x) ← years_at_menopause(?x, ?y).
Has_Bad_Mood(Mary).
years_at_menopause(Mary, 11).
has_health_condition(Mary, body_weakness).
has_health_condition(Peter, body_weakness).
Menopausal(?x) ← years_at_menopause(?x, ?y), $?y > 0$.
¬ Happy(?x)← Has_Bad_Mood(?x).
... /* KB rules defining other properties */

Knowlege Base KB_3

$\langle http://www.possible\_illness \rangle$

P_{KB_3} =

has_possible_illness(?x, anemia) ← Male(?x)@⟨*http://www.person_info*⟩,
has_health_condition(?x, body_weakness)@⟨*http://www.health_conditions*⟩,
age(?x, ?y)@⟨*http://www.person_info*⟩, $?y < 45$.

has_possible_illness(?x, heart_disease) ← Male(?x)@⟨*http://www.person_info*⟩,
has_health_condition(?x, body_weakness)@⟨*http://www.health_conditions*⟩,
age(?x, ?y)@⟨*http://www.person_info*⟩, $?y \geq 45$.

has_possible_illness(?x, depression) ← years_at_menopause(?x, ?y)@⟨*http://www.person_info*⟩,
$y < 10$, $y > 1$, has_health_condition(?x, body_weakness)@⟨*http://www.health_conditions*⟩,
Has_Bad_Mood(?x)@⟨*http://www.person_info*⟩.

has_possible_illness(?x, heart_disease) ← years_at_menopause(?x, ?y)@⟨*http://person.info*⟩,
$y > 10$, has_health_condition(?x, body_weakness)@⟨*http://www.health_conditions*⟩,
∼Has_Bad_Mood(?x)@⟨*http://www.person_info*⟩.
... /* KB rules defining other properties */

Knowledge Base KB_4

$\langle http://www.recommended\_doctors \rangle$

P_{KB_4} =

recommended_doctor(anemia, pathologist).
recommended_doctor(heart_disease, cardiologist).
recommended_doctor(depression, psychiatrist).
possible_doctor(?x, ?z) ← has_possible_illness(?x, ?y)@⟨*http://www.possible_illness*⟩,
recommended_doctor(?y, ?z).
... /* KB rules defining other properties */

Fig. 1. A modular knowledge base

Definition 8 (logic program of a knowledge base w.r.t. an MKB). Let $KB \in \mathcal{K}$. The *logic program* of KB w.r.t. $\mathcal{K}$ is defined as:

$$P^{\mathcal{K}}_{KB} = \bigcup\{tr(r) \mid r \in KB', KB' \in D^{\mathcal{K}}_{KB}\} \ \square$$

We denote all the constants appearing in $P^{\mathcal{K}}_{KB}$ by $V_{KB,\mathcal{K}}$.

Definition 9 (query). A *query* q to a knowledge base KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$ is a formula $[\sim]L_1 \wedge ... \wedge [\sim]L_n$, where L_i, $i = 1, ...n$, are literals with qualification, in case of qualified literals, in $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$.

Below, we define the *stable answers of a query* q to a knowledge base KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. First, we provide an auxiliary definition.

Let $KB \in \mathcal{K}$ and let q be a query to a knowledge base KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. We denote by $tr(q)$ the ELP query derived from q if:

1. we replace each predicate p appearing in a non-qualified literal in q by p_Nam_{KB},
2. we replace each predicate appearing in a qualified literal in q with qualification *qual* by p_qual, and
3. we remove all qualifications from the qualified literals in q.

Definition 10 (answers to a query to a knowledge base w.r.t. an MKB). Let $KB \in \mathcal{K}$ and let q be a query to KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. The *stable answers* of q w.r.t. KB and $\mathcal{K}$ according to mode $mode \in \{cautious, brave\}$ are defined as:

$$Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q) = Ans^{AS,mode}_{P^{\mathcal{K}}_{KB}}(tr(q)) \ \square.$$

In particular, if an answer $v \in Ans^{AS,brave}_{P^{\mathcal{K}}_{KB}}(tr(q))$, it holds that $v(tr(q))$ holds in an Answer Set of $P^{\mathcal{K}}_{KB}$. In contrast, if an answer $v \in Ans^{AS,cautious}_{P^{\mathcal{K}}_{KB}}(tr(q))$, it holds that $v(tr(q))$ holds in *all* Answer Sets of $P^{\mathcal{K}}_{KB}$. In the case that the keyword *mode* is omitted, *cautious* reasoning is assumed.

For answering a query q to a knowledge base $KB \in \mathcal{K}$, only the definition programs of (i) the predicates in $tr(q)$ and (ii) the predicates in $P^{\mathcal{K}}_{KB}$ that can cause *inconsistency* are needed. Once these are fetched through asynchronous P2P messages, query answering can be achieved through Answer Set Programming using the *Splitting Set Theorem* [32]. Replacing in $P^{\mathcal{K}}_{KB}$, the literals $\neg p(\bar{t})$ by $\neg\_p(\bar{t})$, the predicates whose definition programs are required for deriving possible *inconsistency* are:

1. the predicates p and $\neg\_p$ if each one appears in the conclusion of rules,
2. the predicates p if they appear weakly negated in the condition of rules, and
3. the special predicate `false`.

Assumption: To simplify our Algorithms, presented in Section 3, without loss of generality, if in a knowledge base $KB \in \mathcal{K}$ appears a weakly negated qualified literal $\sim[\neg]p(\bar{t})@Nam_{KB'}$ in the body of of rule in P_{KB} then (i) we replace it with a new weakly negated literal $\sim p'(\bar{t})$, where p' is a new predicate name, and (ii) introduce the rule $p'(\bar{t}) \leftarrow [\neg]p(\bar{t})@Nam_{KB'}$. The semantics of the (new) knowledge base KB will remain the same.

Example 3. To understand how weakly negated literals can cause inconsistency, consider the modular knowledge base $\mathcal{K} = \{KB_1, KB_2\}$. Let

$P_{KB_1} = \{p(?x) \leftarrow q(?x)@Nam_{KB_2}\}$ and
$P_{KB_2} = \{q(?x) \leftarrow q'(?x), \sim p(?x)@Nam_{KB_1}. \quad q'(a)\}$.

Then, $P^{\mathcal{K}}_{KB_1}$ and $P^{\mathcal{K}}_{KB_2}$ do not have any answer set, that is knowledge bases KB_1 and KB_2 are inconsistent.

Regarding our previous assumption, we will replace P_{KB_2} by

$$P_{KB_2} = \{q(?x) \leftarrow q'(?x), \sim p_{new}(?x). \quad p_{new}(x) \leftarrow p(?x)@Nam_{KB_1}. \quad q'(a)\}.$$

in order to make our algorithms work, where p_{new} is a new predicate name (not appearing in old P_{KB_2}). The semantics of knowledge bases KB_1 and KB_2 will be the same, that is inconsistency is still derived. □

We now define the definition program of a predicate name appearing in a disjunctive logic program without strong negation.

Definition 11 (definition program of a predicate). Let P be a disjunctive logic program without strong negation. Let *pred* be a predicate name appearing in P or `false`. The *definition program* of *pred* is the minimum set of rules which:

1. includes the rules $r \in P$ where *pred* appears in $Concl(r)$,
2. if r is in the definition program of *pred* and p is a predicate of a literal appearing in $Cond(r)$ then, it includes the rules $r \in P$ where p appears in $Concl(r)$. □

Example 4. Consider the modular knowledge base $\mathcal{K}$ of Example 1. It holds that the definition program of $\texttt{Female\_}Nam_{KB_1}(?\texttt{x})$ (after replacing all literals $\neg p(\bar{t})$ by $\neg\_p(\bar{t})$) is:

$\{\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \texttt{Female\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \neg\_\texttt{Male\_}Nam_{KB_1}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}).$
$\texttt{years\_at\_menopause\_}Nam_{KB_2}(\texttt{Mary}, \texttt{11}).$
$\texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}),\ ?y > 0.\}$.

and the definition program of `false`, after replacing all literals $\neg p(\bar{t})$ by $\neg\_p(\bar{t})$, is:

$\{\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \texttt{Female\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \neg\_\texttt{Male\_}Nam_{KB_1}(?\texttt{x}).$
$\texttt{Male\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \neg\_\texttt{Female\_}Nam_{KB_1}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}).$
$\texttt{years\_at\_menopause\_}Nam_{KB_2}(\texttt{Mary}, \texttt{11}).$
$\texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}),\ ?y > 0.$
$\texttt{Male\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \texttt{Male\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Male\_}Nam_{KB_1}(\texttt{Peter}).$
$\texttt{false} \leftarrow \texttt{Male\_}Nam_{KB_1}(?\texttt{x}), \texttt{Female\_}Nam_{KB_1}(?\texttt{x}).\}$

Thus, only, the union of these two definition programs is needed for sound and complete answering of the query $q = \texttt{Female\_}Nam_{KB_1}(?x)$ on $P^{\mathcal{K}}_{KB}$ according to Answer Set Programming. This is because, according to Splitting Set Theorem [32], the answer set of the rest of the rules in $P^{\mathcal{K}}_{KB}$ will be combined with the answer set of the previous extended logic program (derived after replacing all literals $\neg\_p(\bar{t})$ by $\neg p(\bar{t})$) to provide the unique, in this case, answer set of $P^{\mathcal{K}}_{KB}$.

3 Algorithms

In this section, we present the algorithms of our theory, providing query answering. In the algorithms, query answering is based on asynchronous messaging, where *Ask_pred* and *Ask_KB* messages are send and the corresponding *Tell_pred* and *Tell_KB* messages are returned, all in asynchronous time. Through these messages the related to query answering, sub-part of $P^{\mathcal{K}}_{KB}$ is received by our main algorithm in *polynomial time* with respect to the size of $P^{\mathcal{K}}_{KB}$.

Our main Algorithm 3.1 *Answers_to_query*$(q, Nam_{KB}, mode)$ provides the answers of a user query q to knowledge base $KB \in \mathcal{K}$. The parameter *mode* indicates the mode for answering the query and takes the values "*cautious*" or "*brave*". This algorithm, calls Algorithm 3.2 *Get_query_program*$(q,\ Nam_{KB})$ which returns in *polynomial time* the sub-part of $P^{\mathcal{K}}_{KB}$ that is needed for sound and complete answering of q through Answer Set Programming. In particular, the result will be $Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q)$.

Specifically, in Algorithm 3.2, we define q' as query q after replacing:

1. all literals $\neg p(.)$ and $\neg p(.)@qual$ by $\neg\_p(.)$ and $\neg\_p(.)@qual$, respectively,
2. all predicates p appearing in a non-qualified literal by p_Nam_{KB}, and
3. all predicates p appearing in a qualified literal with qualification *qual* by p_qual.

Let the derived query q' have the form:

$$q' = [\sim]pred_1(\overline{t_1}) \wedge ... \wedge [\sim]pred_n(\overline{t_n}) \wedge [\sim]pred'_1(\overline{t_1})@qual_1 \wedge ... [\sim]pred'_m(\overline{t_m})@qual_m$$

First, we get a new (query) transaction ID, *TID*. Then, for each $i = 1, ..., n$, the asynchronous message $Ask\_pred(TID, pred_i, Nam_{KB})$ is sent to knowledge base KB asking for the definition program of $pred_i$. Additionally, for each $i = 1, ..., m$, the asynchronous message $Ask\_pred(TID, pred'_i, Nam_{KB})$ is sent to knowledge base $qual_i$ asking for the definition program of $pred'_i$. Then the asynchronous message $Ask\_KB(TID, Nam_{KB})$ is sent to knowledge base KB asking for the definition of all predicates in $P^{\mathcal{K}}_{KB}$ that can cause inconsistency. Then, KB waits for the corresponding *Tell* messages with the requested definition programs. These are received in *polynomial time* w.r.t. the size of $P^{\mathcal{K}}_{KB}$. The union of these definitions programs is combined with the query to form a disjunctive logic program that will be returned for query answering.

The rest of the Algorithms correspond to responses to messages *Ask_pred*, *Ask_KB*, *Tell_pred*, and *Tell_KB* received by a knowledge base $KB' \in \mathcal{K}$. To support these responses, each knowledge base $KB' \in \mathcal{K}$ holds five caches. These are:

1. The *predicate program cache*, which holds tuples of the form $(TID, pred, Nam_{KB}, P)$, where P will eventually be the definition program of predicate *pred*, requested by knowledge base KB.

2. The *foreign program cache*, which holds tuples of the form $(TID, pred, Nam_{KB}, C)$, where C is the set of foreign predicates whose definition is needed for completing the definition of local predicate *pred* (requested by knowledge base KB).

3. The *KB program cache*, which holds triples of the form (TID, Nam_{KB}, P), where P will eventually be the union of the definitions of the required, for deriving possible inconsistency, predicates holded by the knowledge bases in $D^{\mathcal{K}}_{KB'}$. Logic program P is requested by knowledge base KB.

4. The *required predicate cache*, which holds triples of the form $(TID, Nam_{KB}, pred)$, where *pred* is a required (for deriving possible inconsistency) local predicate, whose definition is expected by knowledge base KB.

5. The *KB qualification cache*, which holds triples of the form (TID, Nam_{KB}, D), where D indicates the knowledge bases from which the definitions of their local, required for deriving possible inconsistency, predicates will be asked through an Ask_KB message, sent by knowledge base KB'.

Below we describe the rest of the Algorithms.

In Algorithm 3.3, the message $Ask\_pred(TID, Nam_{KB}, pred)$ is sent by knowledge base KB to another knowledge base holding the predicate *pred* and it asks for the definition program of *pred*. The knowledge base receiving the message is indicated by *Self*. First, *Self* checks if *pred* has already been processed. If not, it finds all the local and foreign predicates involved in its definition that have not been processed, yet. The local predicates are put in set A and the rules that define them, after removing qualifications, are put in logic program Π. The foreign predicates (qualified with the name of the knowledge base KB' that are defined) are initially put in set B. From these, the ones that have not been processed yet are put in set C. If $C \neq \{\}$ then Π is stored in the *predicate program cache* and C is stored in *foreign predicate cache*. Then, an $Ask\_pred(TID, Nam_{Self}, p')$ message is sent to the corresponding knowledge bases KB' asking the definition of each of the foreign predicates p' in C. If $C = \{\}$ then the definition of *pred* provided in Π is complete and is *telled* through a *Tell_pred* message to the requesting knowledge base KB.

In Algorithm 3.4, the message $Ask\_KB(TID, Nam_{KB})$ is sent by knowledge base KB to another knowledge base (which is indicated by *Self*). The message asks for the definition programs of all predicates in the knowledge bases in $D^{\mathcal{K}}_{Self}$ that are required for deriving possible *inconsistency*. If such a message has already been sent to *Self* before, then the empty logic program is *telled* to knowledge base KB. If not, then the KB logic program is initialized to empty and stored in the *KB program cache*. The predicates in *Self* whose definition is required for deriving possible *inconsistency* are: (i) the predicates p and $\neg\_p$ if each one appears in the conclusion of rules in *Self*, (ii) the predicates p if they appear weakly negated in the condition of rules in *Self*, and (iii) the special predicate `false`. All these predicates are put in a set A. Each predicate $pred \in A$, that has not been processed yet, is put in the *required predicate cache* and the knowledge base *Self* sends an $Ask\_pred(TID, Nam_{Self}, pred)$ to itself asking for the definition program of *pred*. Then, the set of all qualifications appearing in *Self* is put in a set D, which if it is not empty, it is stored in the *KB qualification cache*. For each qualification $Nam_{KB'} \in D$, an $Ask\_KB(TID, Nam_{Self})$ message is sent by *Self* to knowledge base KB' asking again for the definition programs of all predicates in the knowledge bases in $D^{\mathcal{K}}_{KB'}$, required for deriving possible *inconsistency*. If sets A and D are empty, the empty logic program is *telled* to the requesting knowledge base KB.

In Algorithm 3.5, the message $Tell\_pred(TID, Nam_{KB}, pred, P)$ is sent by knowledge base KB to another knowledge base (which is indicated by *Self*). The message *tells* a disjunctive logic program P to knowledge base *Self* with the definition of predicate *pred*. The knowledge base *Self* checks the *foreign predicate cache* to see if there is in it a tuple $t = (TID, p, Nam_{KB'}, S)$ with $pred@Nam_{KB} \in S$. This indicates that the definition of *pred* in logic program P is required for the definition of a predicate p in *Self* requested by a knowledge base KB'. If $S \neq \{pred@Nam_{KB}\}$ then remove $pred@Nam_{KB}$ from S in the *foreign predicate cache* and replace the logic program Π defining p in *predicate program cache* by $\Pi \cup P$. If $S = \{pred@Nam_{KB}\}$ then delete t from the *foreign predicate*

cache. If p is not found in the *required predicate cache* then find in the *predicate program cache* the logic program Π that defines p and remove the related tuple from the latter cache. In this case, the message $Tell\_pred(TID, Nam_{Self}, p, \Pi \cup P)$ is sent by *Self* to the requesting knowledge base KB', providing the definition program $\Pi \cup P$ for p.

If $S = \{pred@Nam_{KB}\}$ and p is found in the *required predicate cache* asked by a knowledge base KB' then delete the corresponding tuple from the *required predicate cache*. Assume that (i) there is no in *required predicate cache*, another predicate p' asked by KB' and (ii) there is no in *KB qualification cache*, a set D of qualifications stored there through the receival of an $Ask\_KB(TID, KB')$ message by *Self*. Then, retrieve the definition program Π of p requested by knowledge base KB' from the *predicate program cache* and delete the corresponding tuple from this cache. Additionally, retrieve the KB program Π' with the so far definition of required predicates, requested by knowledge base KB', from the *KB program cache*. Finally, send the message $Tell\_KB(TID, Nam_{Self}, \Pi \cup \Pi' \cup P)$ to the knowledge base KB', providing the requested logic program $\Pi \cup \Pi' \cup P$. In the case that (i) or (ii) do not hold then retrieve the definition program Π of p requested by knowledge base KB' from the *predicate program cache* and delete the corresponding tuple from this cache. Additionally, retrieve the KB program Π' with the so far definition of required predicates, requested by knowledge base KB', from the *KB program cache* and replace it by $\Pi \cup \Pi' \cup P$ in this cache.

In Algorithm 3.6, the message $Tell\_KB(TID, Nam_{KB}, P)$ is sent by knowledge base KB to another knowledge base (which is indicated by *Self*). The message *tells* a disjunctive logic program P to knowledge base *Self* with the definition of all required predicates in the knowledge bases in $D^{\mathcal{K}}_{KB}$. The knowledge base *Self* retrieves from the *KB qualification cache*, the set D of qualifications with $Nam_{KB} \in D$, stored there through the receival of an $Ask\_KB(TID, KB')$ message by *Self*. Assume that $D = \{Nam_{KB}\}$ and that it does not exist any required predicate in the *required predicate cache*. Then, retrieve the logic program P' stored in the *KB program cache* and delete it from the cache. Finally, send the message $Tell\_KB(TID, Nam_{Self}, P \cup P')$ to knowledge base KB', providing the requested logic program $P \cup P'$. Assume that $D = \{Nam_{KB}\}$ and that it exists a required predicate in the *required predicate cache*. Then, retrieve the logic program P' stored in the *KB program cache* and replace it by $P \cup P'$. Finally, if $\{Nam_{KB}\} \subset D$ then replace D in *KB qualification cache* by $D - \{Nam_{KB}\}$. Additionally, retrieve the logic program P' stored in the *KB program cache* and replace it by $P \cup P'$.

Algorithm 3.1 $Answers\_to\_query(q, Nam_{KB}, mode)$

Input:
q: is a user query sent to the knowledge base $KB \in \mathcal{K}$,
Nam_{KB}: the name of the knowledge base KB receiving the query,
mode: the mode for answering the query which takes that values "*cautious*" or "*brave*"
Output: $Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q)$

(1) P=$Get\_query\_program(q, Nam_{KB}, flag)$;
/* It gets the logic program required for answering q according to *mode* */
(2) If P is inconsistent then return($\bot$);
else
(3) Evaluate query $\mathtt{Answer}(\overline{x})$ on P through ASP under mode *mode* and
let S be the set of solutions;
(4) *return*(S);

Algorithm 3.2 *Get_query_program*(q, Nam_{KB})

Input:
q: is a user query sent to the knowledge base $KB \in \mathcal{K}$,
Nam_{KB}: the name of the knowledge base KB receiving the query,
Output: A disjunctive logic program which will be used for the evaluation of q

(1) Let q' be q after replacing:
(i) all literals $\neg p(.)$ and $\neg p(.)@qual$ by $\neg\_p(.)$ and $\neg\_p(.)@qual$, respectively,
(ii) all predicates p appearing in a non-qualified literal by p_Nam_{KB}, and
(iii) all predicates p appearing in a qualified literal with qualification $qual$ by p_qual;
(2) Let the derived query q' have the form:
$q' = [\sim]pred_1(\overline{t_1}) \wedge ... \wedge [\sim]pred_n(\overline{t_n}) \wedge [\sim]pred'_1(\overline{t_1})@qual_1 \wedge ... [\sim]pred'_m(\overline{t_m})@qual_m$
(3) Get a new (query) transaction ID, TID;
(4) For $i = 1, ..., n$ do
(5) $\quad Nam_{KB}.Ask\_pred(TID, pred_i, Nam_{KB})$;
(6) EndFor
(7) For $i = 1, ..., m$ do
(8) $\quad qual_i.Ask\_pred(TID, pred'_i, Nam_{KB})$;
(9) EndFor
(10) $Nam_{KB}.Ask\_KB(TID, Nam_{KB})$;
(11) The knowledge base KB waits until receive all messages below:
(i) $Tell\_pred(TID, Nam_{KB}, pred_i, P_i)$, for $i = 1, ..., n$,
(ii) $Tell\_pred(TID, Nam_{KB}, pred'_i, P'_i)$, for $i = 1, ..., m$, and
(iii) $Tell\_KB(TID, Nam_{KB}, P)$.
(12) Let $P' = \{P_i \mid i = 1, ..., n\} \cup \{P'_i \mid i = 1, ..., n\} \cup P$;
(13) Let P'' be P' after replacing all literals $\neg\_pred(.)$ by $\neg pred(.)$;
(14) Let q'' be q' after:
(i) replacing all literals $\neg\_pred(.)$ and $\neg\_pred(.)@qual$ by
$\neg pred(.)$ and $\neg pred(.)@qual$, respectively, and
(ii) after removing all qualifications;
(15) $P'' = P'' \cup \{\texttt{Answer}(\overline{\texttt{x}}) \leftarrow q'' \mid \overline{x}$ are the variables of $q\}$;
(16) $return(P'')$;

Algorithm 3.3 *Ask_pred*$(TID, Nam_{KB}, pred)$

TID: a transanction ID,
Nam_{KB}: the name of a requesting knowledge base KB,
pred: a predicate name whose definition program is requested by KB or `false`

(1) Let *Self* denote the knowledge base receiving the *Ask_pred* message;
(2) If $T(TID, pred) = processed$ then
$Nam_{KB}.Tell\_pred(TID, Nam_{Self}, pred, \{\})$;
/* if a knowledge base has already requested the definition program
of predicate *pred* for a user query with transaction ID, *TID*, then send a *Tell*
message to the requesting knowledge base KB with the empty logic program */
(3) else
(4) Let P be P_{Self} after replacing:
(i) all literals $\neg p(.)$ and $\neg p(.)@qual$ by $\neg\_p(.)$ and $\neg\_p(.)@qual$, respectively,
(ii) all predicates p appearing in a non-qualified literal by p_Nam_{Self}, and
(iii) all predicates p appearing in a qualified literal with qualification *qual* by p_qual;
(5) Let $A = \{pred\}$;
(6) Let $B = \{\}$;
(7) Let $\Pi = \{\}$;
(8) For each $p \in A$ s.t. $T(TID, p) \neq processed$ do
(9) $T(TID, p) = processed$;
(10) For each $r \in P$ s.t. p appears in $Concl(r)$ do
(11) $\Pi = \Pi \cup \{r$ after removing qualifications$\}$;
(12) $A = A \cup \{p' \mid p'$ is a predicate of a non-qualified literal appearing in $Cond(r)$
or $Concl(r)\}$;
(13) $B = B \cup \{p'@Nam_{KB'} \mid p'$ is a predicate of a qualified literal
$p'(.)@Nam_{KB'}$ in $Cond(r)\}$;
(14) EndFor
(15) $C = \{\}$;
(16) For each $p@Nam_{KB'} \in B$ s.t. $T(TID, p@Nam_{KB'}) \neq processed$ do
(17) $T(TID, p@Nam_{KB'}) = processed$;
(18) $C = C \cup \{p@Nam_{KB'}\}$;
(19) EndFor
(20) If $C \neq \{\}$ then
(21) Put in *predicate program cache* the tuple $(TID, pred, Nam_{KB}, \Pi)$;
/* Π is the logic program derived so far that defines *pred* and is cached*/
(22) Put in *foreign predicate cache* the tuple $(TID, pred, Nam_{KB}, C)$;
(23) For each $p'@Nam_{KB'} \in C$ do
(24) $Nam_{KB'}.Ask\_pred(TID, Nam_{Self}, p')$;
/* knowledge base *Self* requests the definition of predicate p' from
knowledge base KB' in order to complete the definition of *pred* */
(25) EndFor
(26) else /* $C = \{\}$ */
(27) $Nam_{KB}.Tell\_pred(TID, Nam_{Self}, pred, \Pi)$;
/* if $C = \{\}$ then the definition of *pred* provided in Π is complete
and is *telled* through a *Tell_pred* message
to the requesting knowledge base KB */

Algorithm 3.4 $Ask\_KB(TID, Nam_{KB})$

TID: a transanction ID,
Nam_{KB}: the name of a knowledge base KB requesting the definition
of all predicates in the knowledge bases in $D^{\mathcal{K}}_{Self}$, that are required
for deriving possible inconsistency

(1) Let *Self* denote the knowledge base receiving the Ask_KB message;
(2) If $T(TID, \text{“}KB\text{”}, Nam_{Self}) = processed$ then
(3) $Nam_{KB}.Tell\_KB(TID, Nam_{Self}, \{\})$;
/* if a knowledge base has already requested from *Self* the required
predicate definitions then send a $Tell\_KB$ message to the requesting
knowledge base KB with the empty logic program */
(4) else
(5) $T(TID, \text{“}KB\text{”}, Nam_{Self}) = processed$;
(6) Store in *KB_program cache* $(TID, Nam_{KB}, \{\})$;
/* Initialize the KB logic program, which will contain the definitions of
the required predicates, to empty */
(7) Let P be P_{Self} after replacing:
(i) all literals $\neg p(.)$ and $\neg p(.)@qual$ by $\neg\_p(.)$ and $\neg\_p(.)@qual$, respectively,
(ii) all predicates p appearing in a non-qualified literal by p_Nam_{Self}, and
(iii) all predicates p appearing in a qualified literal with qualification $qual$ by p_qual;
(8) $A = \{p, \neg\_p \mid p, \neg\_p$ are predicates each one appearing in the conclusions of rules of $P\}$;
(9) $A = A \cup \{p \mid p$ is a predicate of a literal L in P that appears weakly negated$\}$;
(10) $A = A \cup \{\texttt{false}\}$;
(11) For each $pred \in A$ s.t. $T(TID, pred) \neq processed$ do
(12) $T(TID, pred) = processed$;
(13) Store in *required predicate cache* $(TID, Nam_{KB}, pred)$;
/* $pred$ is a required predicate */
(14) $Nam_{Self}.Ask\_pred(TID, Nam_{Self}, pred)$;
(15) EndFor
(16) Let $D = \{Nam_{KB'} \mid Nam_{KB'}$ is a qualification appearing in $P\}$;
(17) If $D \neq \{\}$ then store in *KB qualification cache* (TID, Nam_{KB}, D);
(18) For $Nam_{KB'} \in D$ do
(19) $Nam_{KB'}.Ask\_KB(TID, Nam_{Self})$;
/* knowledge base *Self* asks knowledge base KB' for the definition
of all predicates defined in the knowledge based in $D^{\mathcal{K}}_{KB'}$, that
are required for deriving possible inconsistency */
(20) EndFor
(21) If $A = \{\}$ and $D = \{\}$ then
(22) $Nam_{KB}.Tell\_KB(TID, Nam_{Self}, \{\})$;
/* If there are not required predicates in knowledge base *Self* and there are not
knowledge bases to be accessed by *Self* then *tell* the empty logic program
to the requesting knowledge base KB */

Algorithm 3.5 *Tell_pred*$(TID, Nam_{KB}, pred, P)$

TID: a transanction ID,
Nam_{KB}: the name of the knowledge base KB sending the requested logic program P defining predicate $pred$,
$pred$: a predicate name or `false` whose definition is found in logic program P,
P: a disjunctive logic program *Telled* by knowledge base KB

(1) Find in *foreign predicate cache* tuple $t = (TID, p, Nam_{KB'}, S)$, for a $p, Nam_{KB'}$, and S s.t. $pred@Nam_{KB} \in S$;
(2) If $S \neq \{pred@Nam_{KB}\}$ then
(3) Replace in *foreign predicate cache*, tuple t by $(TID, p, Nam_{KB'}, S - \{pred@Nam_{KB}\})$;
(4) Find in *predicate program cache*, tuple $(TID, p, Nam_{KB'}, \Pi)$, for a logic program Π, and replace it with $(TID, p, Nam_{KB'}, \Pi \cup P)$;
/* the logic program defining p that is requested by knowledge base KB' is updated to $\Pi \cup P$ */
(5) else /* $S = \{pred@Nam_{KB}\}$ */
(6) Delete tuple t from *foreign predicate cache*;
(7) If the tuple $(TID, Nam_{KB'}, p)$ does not exist in *required predicate cache*,
/* if p is not a required predicate */
(8) then
(9) Find in *predicate program cache* tuple $(TID, p, Nam_{KB'}, \Pi)$, for a logic program Π, and remove it from the cache;
(10) $Nam_{KB'}.Tell\_pred(TID, Nam_{Self}, p, \Pi \cup P)$;
/* the logic program $\Pi \cup P$ corresponds to the definition of predicate p and it is *Telled* to the requesting knowledge base KB' */
(11) If $S = \{pred@Nam_{KB}\}$ and the tuple $t = (TID, Nam_{KB'}, p)$ is in *required predicate cache*
(12) then
(13) Delete t from the *required predicate cache*;
(14) If there is no tuple $(TID, Nam_{KB'}, p')$, for a predicate p', in *required predicate cache* and there is no tuple $(TID, Nam_{KB'}, D)$, for a D, in *KB qualification cache*
(15) then
(16) Retrieve tuple $(TID, p, Nam_{KB'}, \Pi)$, for a Π, from *predicate program cache* and delete it from this cache;
(17) Retrieve tuple $(TID, Nam_{KB'}, \Pi')$, for a Π', from *KB program cache* and delete it from this cache;
(18) $Nam_{KB'}.Tell\_KB(TID, Nam_{Self}, \Pi \cup \Pi' \cup P)$;
/* *Tell* to the knowledge base KB', the logic program $\Pi \cup \Pi' \cup P$ */
(19) else
(20) Retrieve tuple $(TID, p, Nam_{KB'}, \Pi)$, for a Π, from *predicate program cache* and delete it from this cache;
(21) Retrieve tuple $(TID, Nam_{KB'}, \Pi')$, for a Π', from *KB program cache* and replace it with $(TID, Nam_{KB'}, \Pi \cup \Pi' \cup P)$;
/* Update the logic program in *KB program cache* to be *telled* when completed to knowledge base KB' */

Algorithm 3.6 *Tell_KB*(TID, Nam_{KB}, P)

TID: a transaction ID,
Nam_{KB}: the name of the knowledge base KB sending the requested logic program P defining the required predicates,
P: a disjunctive logic program *Telled* by knowledge base KB

(1) Retrieve and delete from the KB *qualification cache*, the tuple $(TID, Nam_{KB'}, D)$, for a D, with $Nam_{KB} \in D$;
(2) If $D = \{Nam_{KB}\}$ then
(3) If it does not exist tuple $(TID, Nam_{KB'}, p)$, for a p, in *required predicate cache* then
(4) Retrieve tuple $(TID, Nam_{KB'}, P')$, for a P', from KB *program cache* and delete it from the cache;
(5) $Nam_{KB'}.Tell\_KB(TID, Nam_{Self}, P \cup P')$;
(6) else
(7) Retrieve tuple $(TID, Nam_{KB'}, P')$, for a P', from KB *program cache* and replace it by $(TID, Nam_{KB'}, P \cup P')$;
(8) else /* $\{Nam_{KB}\} \subset D$ */
(9) Store in KB *qualification cache*, the tuple $(TID, Nam_{KB'}, D - \{Nam_{KB}\})$;
(10) Retrieve tuple $(TID, Nam_{KB'}, P')$, for a P', from KB *program cache* and replace it with $(TID, Nam_{KB'}, P \cup P')$;

Example 5. Consider the modular knowledge base $\mathcal{K}$ of Example 1. Assume that the call *Answers_to_query*(`Female`$(?x), Nam_{KB_1}$, "*cautious*") is issued by the user with the query `Female`$(?x)$. Then, the call *Get_query_program*(`Female`$(?x), Nam_{KB_1}$) is made. This call will generate a new query transaction ID, TID_1, and send the messages *Ask_pred*(TID_1, Nam_{KB_1}, `Female_`Nam_{KB_1}) and *Ask_KB*(TID_1, Nam_{KB_1}) to knowledge base KB_1.

When KB_1 receives the message *Ask_pred*(TID_1, Nam_{KB_1}, `Female_`Nam_{KB_1}), it sets $A = \{$`Female_`$Nam_{KB_1}, \neg$`_Male_`$Nam_{KB_1})\}$ and $B = C = \{$`Female_`$Nam_{KB_2}@Nam_{KB_2}\}$. Additionally,

$$\Pi = \{\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \texttt{Female\_}Nam_{KB_2}(?\texttt{x}).$$
$$\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \neg\texttt{\_Male\_}Nam_{KB_1}(?\texttt{x}).\}$$

The tuple $(TID_1, \texttt{Female\_}Nam_{KB_1}, Nam_{KB_1}, \Pi)$ is stored in the *predicate program cache* and the tuple $(TID_1, \texttt{Female\_}Nam_{KB_1}, Nam_{KB_1}, C)$ is stored in the *foreign predicate cache*. Then, the message *Ask_pred*(TID_1, Nam_{KB_1}, `Female_`Nam_{KB_2}) is sent to KB_2.

When KB_2 receives the message *Ask_pred*(TID_1, Nam_{KB_1}, `Female_`Nam_{KB_2}), it sets $A = \{$`Female_`Nam_{KB_2}, `Menopausal_`Nam_{KB_2}, `years_at_menopause_`$Nam_{KB_2}\}$ and $B = C = \{\}$. Additionally,

$$\Pi_1 = \{\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}).$$
$$\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}).$$
$$\texttt{years\_at\_menopause\_}Nam_{KB_2}(\texttt{Mary}, \texttt{11}).$$
$$\texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}),\ ?y > 0.\}.$$

Since $C = \{\}$, the message *Tell_pred*(TID_1, Nam_{KB_2}, `Female_`Nam_{KB_2}, Π_1) is sent to KB_1. When KB_1 receives this message, the tuple $t = (TID_1, \texttt{Female\_}Nam_{KB_1}, Nam_{KB_1}, S)$, with $S = \{$`Female_`Nam_{KB_2}`@_`$Nam_{KB_2}\}$ is retrieved from the *foreign predicate cache* and delete it from the cache. Since no tuple (TID_1, Nam_{KB_i}, `Female_`Nam_{KB_1}) exists in the *required predicate cache*, the tuple $(TID_1, \texttt{Female\_}Nam_{KB_1}, Nam_{KB_1}, \Pi)$ is

retrieved from the *predicate program cache* and removed from the cache. Then, the message $Tell\_pred(TID_1, Nam_{KB_1}, \texttt{Female\_}Nam_{KB_1}, \Pi \cup \Pi_1)$ is sent to the queried, by the user, knowledge base KB_1 which waits for it.

Now the message $Ask\_KB(TID_1, Nam_{KB_1})$ which has been sent to knowledge base KB_1, it sets $A = \{\texttt{false}\}$. Then, the tuple $(TID_1, Nam_{KB_1}, \texttt{false})$ is stored in the *required predicate cache* and the message $Ask\_pred(TID_1, Nam_{KB_1}, \texttt{false})$ is sent to KB_1. Additionally, $D = \{Nam_{KB_2}\}$ and the tuple (TID_1, Nam_{KB_1}, D) is stored in the *KB qualification cache*. Then, the message $Ask\_KB(TID_1, Nam_{KB_1})$ is sent to knowledge base KB_2.

When KB_1 receives the message $Ask\_pred(TID_1, Nam_{KB_1}, \texttt{false})$, the set $A - \{p \mid T(TID_1, p) = processed\} = \{\texttt{Male\_}Nam_{KB_1}, \neg\texttt{Female\_}Nam_{KB_1}\}$. Note that the predicate $\texttt{Female\_}Nam_{KB_1}$ has already been processed. Additionally, the set
$C = \{\texttt{Male\_}Nam_{KB_2}@\ Nam_{KB_2}\}$ and the logic program

$\Pi_2 = \{\texttt{Male\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \texttt{Male\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Male\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \neg\texttt{Female\_}Nam_{KB_1}(?\texttt{x}).$
$\texttt{Male\_}Nam_{KB_1}(\texttt{Peter}).$
$\texttt{false} \leftarrow \texttt{Male\_}Nam_{KB_1}(?\texttt{x}), \texttt{Female\_}Nam_{KB_1}(?\texttt{x}).\}.$

The tuple $(TID_1, \texttt{false}, Nam_{KB_1}, \Pi_2)$ is stored in the *predicate program cache* and the tuple $(TID_1, \texttt{Male\_}Nam_{KB_1}, Nam_{KB_1}, C)$ is stored in the *foreign predicate cache*. Then, the message $Ask\_pred(TID_1, Nam_{KB_1}, \texttt{Male\_}Nam_{KB_2})$ is sent to KB_2. When KB_2 receives the latter message, it is derived $A = C = \{\}$ and thus the message $Tell\_pred(TID_1, Nam_{KB_2}, \texttt{Male\_}Nam_{KB_2}, \{\})$ is sent to KB_1.

When KB_1 receives this message, the tuple $t = (TID_1, \texttt{false}, Nam_{KB_1}, S)$, with $S = \{\texttt{Male\_}Nam_{KB_2}@Nam_{KB_2}\}$ is retrieved from the *foreign predicate cache* and is deleted from the cache. Note that the tuple $(TID_1, Nam_{KB_1}, \texttt{false})$ exists in the *required predicate cache* and is deleted from the cache. Now, the tuple $(TID_1, \texttt{false}, Nam_{KB_1}, \Pi_2)$ is retrieved from the *predicate program cache* and removed from the cache. Additionally, the tuple $(TID, Nam_{KB_1}, \{\})$ is retrieved from the *KB program cache* and replaced with $(TID_1, Nam_{KB_1}, \Pi_2)$.

When KB_2 receives the message $Ask\_KB(TID_1, Nam_{KB_1})$, it holds $A = D = \{\}$ and the message $Tell\_KB(TID_1, Nam_{KB_2}, \{\})$ is sent to KB_1. When KB_1 receives this message, the tuple (TID_1, Nam_{KB_1}, D), with $Nam_{KB_2} \in D$, is retrieved from the *KB qualification cache* and delete it from the cache. Since $D = \{Nam_{KB_2}\}$ and it does not exist tuple (TID_1, Nam_{KB_1}, p), for a p, in the *required predicate cache*, the tuple $(TID_1, Nam_{KB_1}, \Pi_2)$ is retrieved from the *KB program cache* and deleted from the cache. Then, message $Tell\_KB(TID_1, Nam_{KB_1}, \Pi_2)$ is sent to the queried by the user, knowledge base KB_1 which waits for it.

Thus, $Get\_query\_program(\texttt{Female}(?\texttt{x}), Nam_{KB_1})$ will return the extended logic program:

$\Pi_3 = \Pi\ \cup\ \Pi_1\ \cup\ \Pi_2\ \cup\ \{\texttt{Answer}(?\texttt{x}) \leftarrow \texttt{Female\_}Nam_{KB_1}(?\texttt{x})\} =$
$\{\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \texttt{Female\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \neg\texttt{Male\_}Nam_{KB_1}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Female\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}).$
$\texttt{years\_at\_menopause\_}Nam_{KB_2}(\texttt{Mary}, \texttt{11}).$
$\texttt{Menopausal\_}Nam_{KB_2}(?\texttt{x}) \leftarrow \texttt{years\_at\_menopause\_}Nam_{KB_2}(?\texttt{x}, ?\texttt{y}),\ ?y > 0.$
$\texttt{Male\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \texttt{Male\_}Nam_{KB_2}(?\texttt{x}).$
$\texttt{Male\_}Nam_{KB_1}(?\texttt{x}) \leftarrow \neg\texttt{Female\_}Nam_{KB_1}(?\texttt{x}).$
$\texttt{Male\_}Nam_{KB_1}(\texttt{Peter}).$

$\texttt{false} \leftarrow \texttt{Male\_}Nam_{KB_1}(?\texttt{x}), \texttt{Female\_}Nam_{KB_1}(?\texttt{x}).$
$\texttt{Answer}(?\texttt{x}) \leftarrow \texttt{Female\_}Nam_{KB_1}(?\texttt{x}).\}$

The Algorithm *Answers_to_query*($\texttt{Female}(x), Nam_{KB_1}$, "*cautious*") will evaluate Π_3 through ASP in *cautious* mode and it will return to the user the answer $\{(?\texttt{x}, \texttt{Mary})\}$. □

Let $KB \in \mathcal{K}$ and let q be a query over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. Recall that the call *Get_query_program*(Nam_{KB}, q), made by the *Answers_to_query*($Nam_{KB}, q, mode$) call, returns a disjunctive logic program Π to be evaluated through ASP. As we will see in Theorem 1 below, the evaluation of Π provides the *stable answers* of q w.r.t. KB and $\mathcal{K}$ according to *mode*, that is $Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q)$.

It is easy to see that following Proposition holds, which shows that:
$\Pi \subseteq P^{\mathcal{K}}_{KB} \cup \{Answers(\overline{x}) \leftarrow tr(q)\}$. This is because *Get_query_program*(Nam_{KB}, q) retrieves only the parts of $P^{\mathcal{K}}_{KB}$ that are required for answering the query q.

Proposition 1. Let $KB \in \mathcal{K}$ and let q be a query to KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. Additionally, let v be (i) "yes", if $Var(F) = \emptyset$, or (ii) a mapping $v : Var(F) \rightarrow V_{KB,\mathcal{K}}$, if $Var(F) \neq \emptyset$. Then, *Get_query_program*(Nam_{KB}, q) $\subseteq P^{\mathcal{K}}_{KB} \cup \{Answers(\overline{x}) \leftarrow tr(q)\}$. while there are cases where the equality holds.

The following Theorem 1 shows that the *stable answers* of a query q w.r.t. KB and $\mathcal{K}$ according to *mode* can be computed through the call *Answers_to_query*($Nam_{KB}, q, mode$). Its proof is provided in Appendix A.

Theorem 1. Let $\mathcal{K}$ be a modular knowledge base. Let $KB \in \mathcal{K}$ and let q be a query to KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. Then, the following hold:

1. If *Answers_to_query*($q, Nam_{KB}, mode$) $= \perp$ then $P^{\mathcal{K}}_{KB}$ is inconsistent.
2. If *Answers_to_query*($q, Nam_{KB}, mode$) $\neq \perp$ then

$$Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q) = Answers\_to\_query(q, Nam_{KB}, mode).$$

Let $KB \in \mathcal{K}$. We define:

$$size(KB, \mathcal{K}) = \sum\{\text{size of } P_{KB} \mid KB' \in D^{\mathcal{K}}_{KB}\} = \text{size of } P^{\mathcal{K}}_{KB}$$
$$fact\_size(KB, \mathcal{K}) = \sum\{\text{fact size of } P_{KB} \mid KB' \in D^{\mathcal{K}}_{KB}\} = \text{fact size of } P^{\mathcal{K}}_{KB}$$

Below, we provide the complexities of query answering on $P^{\mathcal{K}}_{KB}$ as derived from [17], [13], and [31]. Note that $P^{\mathcal{K}}_{KB}$ is a general disjunctive logic program with weak and strong negation, as well as constraints. References [17], [13], and [31] provide the complexities of these logic programs as well as their special case of extended logic programs (that they do not include disjunction in the head of their rules).

Proposition 2. Let $KB \in \mathcal{K}$ and let q be a query to KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. Additionally, let v be (i) "yes", if $Var(q) = \emptyset$, or (ii) a mapping $v : Var(F) \rightarrow V_{KB,\mathcal{K}}$, if $Var(F) \neq \emptyset$. The problem of establishing whether $v(q)$ is a *cautious* consequence of $P^{\mathcal{K}}_{KB}$ (i.e. $v(q) \in Ans^{\mathtt{st},cautious}_{KB,\mathcal{K}}$) is:

1. co-NEXP$^{\text{NP}}$-complete w.r.t. $size(KB, \mathcal{K})$,
2. co-NP$^{\text{NP}}$-complete w.r.t. $fact\_size(KB, \mathcal{K})$,
3. co-NEXP-complete w.r.t. $size(KB, \mathcal{K})$, if $P^{\mathcal{K}}_{KB}$ is an extended logic program, and
4. co-NP-complete w.r.t. $fact\_size(KB, \mathcal{K})$, if $P^{\mathcal{K}}_{KB}$ is an extended logic program.

Proposition 3. Let $KB \in \mathcal{K}$ and let q be a query to KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. Additionally, let v be (i) "yes", if $Var(q) = \emptyset$, or (ii) a mapping $v : Var(F) \rightarrow V_{KB,\mathcal{K}}$, if $Var(F) \neq \emptyset$. The problem of establishing whether $v(q)$ is a *brave* consequence of $P^{\mathcal{K}}_{KB}$ (i.e. $v(q) \in Ans^{\mathtt{st},brave}_{KB,\mathcal{K}}$) is:

1. NEXP$^{\text{NP}}$-complete w.r.t. $size(KB, \mathcal{K})$ and
2. NP$^{\text{NP}}$-complete w.r.t. $fact\_size(KB, \mathcal{K})$.
3. NEXP-complete w.r.t. $size(KB, \mathcal{K})$, if $P^{\mathcal{K}}_{KB}$ is an extended logic program, and
4. NP-complete w.r.t. $fact\_size(KB, \mathcal{K})$, if $P^{\mathcal{K}}_{KB}$ is an extended logic program.

We want to emphasize that though the complexities for query answering are high, the number of exchange messages *Ask_pred*, *Tell_pred*, *Ask_KB*, *Tell_KB* is polynomial and in fact linear w.r.t. the $size(KB, \mathcal{K})$. Note that for each *Ask_pred* message, there is in return a unique *Tell_pred* and for each *Ask_KB* message, there is in return a unique *Tell_KB* message.

Based on the above, the following Proposition holds whose proof is provided in Appendix A.

Proposition 4. Let $\mathcal{K}$ be a modular knowledge base. Let $KB \in \mathcal{K}$ and let q be a query to KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$. The complexity of the algorithm *Get_query_program*(q, Nam_{KB}) is polynomial w.r.t. $size(KB, \mathcal{K})$.

4 Applications

In this section, we define applications of our framework. Specifically, applications where our algorithms can be used.

4.1 Query Answering on Restricted Modular ERDF Ontologies

In this subsection, we will define the restricted modular ERDF ontologies, which is a special case of the simple modular ERDF ontologies, defined in [2]. Efficient P2P query answering on restricted modular ERDF ontologies can be achieved through our algorithms. Since restricted modular ERDF ontologies is a special case of the simple modular ERDF ontologies, for helping the reader with the notations that we necessarily use from the latter framework, we provide a summary of the simple modular ERDF ontologies (from [2]) in the electronic Appendix `http://users.ics.forth.gr/~analyti/Papers_2/Appendix_Modular_KBs.pdf`. There, we also provide an Extended Logic Program $\Pi_{O,\mathcal{R}}$, where $\mathcal{R}$ is a simple modular EDF ontology and $O \in \mathcal{R}$, defined also in [2], based on which query answering can be achieved through Answer Set Programming [25].

In [2], we have shown that reasoning on simple modular ERDF ontologies *faithfully* extends reasoning on RDFS ontologies [28], while the framework adds rules, weak and strong negation, and modularity to RDFS. Similarly, reasoning on restricted modular ERDF ontologies *faithfully* extends reasoning on RDFS ontologies (according to the latest RDFS semantics [29]). This framework also adds rules, weak and strong negation, and modularity to RDFS, imposing, however, certain restrictions.

Reasoning on restricted modular ERDF ontologies can be *translated* to reasoning on modular knowledge bases and thus our algorithms can be applied. Note that reasoning on simple modular ERDF ontologies can also be translated to reasoning on modular knowledge bases, but there our algorithms will be inefficient. This is because, in this case, each formed knowledge base will have different binary predicates that are not equal to `type`, but a single predicate `type` for the instances of classes. Thus, applying our

algorithms, when the definition program of predicate `type` is requested, all local classes and their instances will have to been transferred through the P2P network.

In contrast, in restricted modular ERDF ontologies, we impose the restriction that if $[\neg]\texttt{type}(o, c)$ appears in a rule then c should not be a variable but only an IRI. This allows us to translate restricted modular ERDF ontologies to modular knowledge bases where knowledge bases may have, along with different binary predicates, different unary predicates, one for each class. Thus, in this case, only the required classes and their instances in the corresponding knowledge bases will have to the transferred through the P2P network and, consequently, our algorithms can be efficiently applied.

Below we provide the related definitions.

Definition 12 (restricted r-ERDF ontology). A *restricted* r*-ERDF ontology* O is a simple r-ERDF ontology that satisfies the following constraints:

1. For all $\texttt{subClassOf}(c_1, c_2) \in G_O$, it holds that $c_2 \notin \{\texttt{TotalClass}, \texttt{TotalProperty}, \texttt{Datatype}, \texttt{ContainerMembershipProperty}\}$.
2. For all $\texttt{subPropertyOf}(p_1, p_2) \in G_O$, it holds that $p_2 \notin \{\texttt{domain}, \texttt{range}, \texttt{subClassOf}, \texttt{subPropertyOf}\}$.
3. For all $r \in P_O$, if $Concl(r) = p(x, y)$ then $p \notin \{\texttt{domain}, \texttt{range}, \texttt{subClassOf}, \texttt{subPropertyOf}\}$.
4. For all $r \in P_O$, if $[\neg]\texttt{type}(o, c)$ appears in r then $c \notin Var$.
5. For all $r \in P_O$, if $Concl(r) = \texttt{type}(o, c)$ then $c \notin \{\texttt{TotalClass}, \texttt{TotalProperty}, \texttt{Datatype}, \texttt{ContainerMembershipProperty}\}$.
6. $\forall p \in Imported_O^{\texttt{cl}}$, it holds that $p \notin \{\texttt{Datatype}, \texttt{ContainerMembershipProperty}\}$.
7. $\forall p \in Imported_O^{\texttt{pr}}$, it holds that $p \notin \{\texttt{type}, \texttt{domain}, \texttt{range}, \texttt{subClassOf}, \texttt{subPropertyOf}\}$.
8. (*SAFETY*): $\forall r \in P_O$, if variable $?x$ appears in $Concl(r)$ or in a weakly negated ERDF triple in $Cond(r)$ then $?x$ appears in a non-weakly negated ERDF triple in $Cond(r)$. □

From the above restrictions, the only strong restriction is Condition 4 because it does not allow to reason on classes that they are variables and their values are taken from other triples in the condition of the rules.

Definition 13 (restricted modular ERDF ontology). A *restricted modular ERDF ontology* $\mathcal{G}$ is a simple modular ERDF ontology s.t. each $O \in \mathcal{G}$ is a restricted r-ERDF ontology. □

Note that Example 6 provides an example restricted modular ERDF ontology.

Example 6. Consider the restricted modular ERDF ontology $\mathcal{R} = \{O_1, O_2\}$, shown in Figure 2. Ontology O_1, with Nam_{O_1} =`<http://www.person_info>`, provides the sex and age of person. Ontology O_2, with Nam_{O_2} =`<http://www.health_conditions>`, provides information about health conditions of persons. □

Convention: In the rest of the paper by $\mathcal{G}$*, we will denote a restricted modular ERDF ontology.*

Let $\mathcal{D}$ be the chosen set of *recognized* datatype IRIs for the application [29]. It holds that: $\{\texttt{rdf:langString}, \texttt{xsd:string}\} \subseteq \mathcal{D}$.

Let $O \in \mathcal{G}$. Below, we define the ELPs Π_O and $\Pi_O^{\mathcal{G}}$, which will be used for query answering through Answer Set Programming (ASP). First, we define:

Ontology O_1

$\langle http://www.person\_info \rangle$

`imports class Male, Female from` $\langle http://health.conditions \rangle$.

$G_{O_1} =$

`subClassOf(Female, Person).`
`subClassOf(Male, Person).`
`Female(?x)` $\leftarrow \neg$ `Male(?x).`
`Male(?x)` $\leftarrow \neg$ `Female(?x).`
`type(Peter, Male).`
`age(Peter, 40).`
`age(Mary, 65).`

$P_{O_1} =$

`false` $\leftarrow$ `type(?x, Male), type(?x, Female).`
... /* ERDF graph triples and rules defining other properties */

Ontology O_2

$\langle http://www.health\_conditions \rangle$

$G_{O_2} =$

`subClassOf(Menopausal, Female).`
`domain(years_at_menopause, Female).`
`type(Mary, Has_Bad_Mood).`
`years_at_menopause(Mary, 11).`
`has_health_condition(Mary, body_weakness).`
`has_health_condition(Peter, body_weakness).`

$P_{O_2} =$

`type(?x, Menopausal)` $\leftarrow$ `years_at_menopause(?x, ?y),` $?y > 0$.
$\neg$ `type(?x, Happy)` $\leftarrow$ `type(?x, Has_Bad_Mood).`
... /* ERDF graph triples and rules defining other properties */

Fig. 2. A restricted modular ERDF ontology

Π_{P_O}=$\{r'$ resulting from r after replacing all $\texttt{type}(o,c)$ appearing in r by $c(o)|\ r \in P_O\}$.

Π_{G_O}=$\{p(x,y) \leftarrow \texttt{true} \mid p(x,y) \in sk(G_O)$ and $p \neq \texttt{type}\} \cup \{c(o) \leftarrow \texttt{true} \mid \texttt{type}(o,c) \in sk(G_O)\}$.

We denote by Π_O^{rERDF} the ELP that consists of the following sets of rules:

Importing Rules
If $p \in Imported_O^{\texttt{pr}}$ and $Nam_{O'} \in Import_O^{\texttt{pr}}(p)$ then:
$p(?x,?y) \leftarrow p(?x,?y)@Nam_{O'}$.
$\neg p(?x,?y) \leftarrow \neg p(?x,?y)@Nam_{O'}$.

If $c \in Imported_{O'}^{\texttt{cl}}$ and $Nam_{O'} \in Import_O^{\texttt{cl}}(c)$ then:
$c(?x) \leftarrow c(?x)@Nam_{O'}$.
$\neg c(?x) \leftarrow \neg c(?x)@Nam_{O'}$.

Restricted ERDF Interpretation Rules
For all $[\neg]p(x,y)$ appearing non-qualified in O or $p \in Imported_O^{\texttt{pr}}$:
$\texttt{Property}(p) \leftarrow p(?x,?y)$.
$\texttt{Property}(p) \leftarrow \neg p(?x,?y)$.

For each $[\neg]\texttt{type}(o,c)$ appearing non-qualified in $sk(G_O)$ or P_O or $c \in Imported_O^{\texttt{cl}}$:
$\texttt{Class}(c) \leftarrow c(?x)$.
$\texttt{Class}(c) \leftarrow \neg c(?x)$.

$\texttt{subClassOf}(?x,\texttt{Resource}) \leftarrow \texttt{Class}(?\texttt{x})$.

$\texttt{Class}(?x) \leftarrow \texttt{subClassOf}(?x,?y)$.
$\texttt{Class}(?y) \leftarrow \texttt{subClassOf}(?x,?y)$.
$\texttt{subClassOf}(?x,?x) \leftarrow \texttt{Class}(?x)$.
$\texttt{subClassOf}(?x,?z) \leftarrow \texttt{subClassOf}(?x,?y), \texttt{subClassOf}(?y,?z)$.

$\texttt{Property}(?x) \leftarrow \texttt{subPropertyOf}(?x,?y)$.
$\texttt{Property}(?y) \leftarrow \texttt{subPropertyOf}(?x,?y)$.
$\texttt{subPropertyOf}(?x,?x) \leftarrow \texttt{Property}(?x)$.
$\texttt{subPropertyOf}(?x,?z) \leftarrow \texttt{subPropertyOf}(?x,?y), \texttt{subPropertyOf}(?y,?z)$.

$\texttt{Property}(?x) \leftarrow \texttt{domain}(?x,?y)$.
$\texttt{Class}(?y) \leftarrow \texttt{domain}(?x,?y)$.
$\texttt{Property}(?x) \leftarrow \texttt{range}(?x,?y)$.
$\texttt{Class}(?y) \leftarrow \texttt{range}(?x,?y)$.

For $d \in \mathcal{D}$:
$\texttt{Datatype}(d) \leftarrow \texttt{true}$.

$\texttt{subClassOf}(?x,\texttt{Literal}) \leftarrow \texttt{Datatype}(?x)$.
$\texttt{subPropertyOf}(?x,\texttt{member}) \leftarrow \texttt{ContainerMembershipProperty}(?x)$.

For all $\texttt{subClassOf}(c_1,c_2) \in sk(G_O)$: $c_2(?x) \leftarrow c_1(?x)$.
For all $\texttt{subPropertyOf}(p_1,p_2) \in sk(G_O)$: $p_2(?x,?y) \leftarrow p_1(?x,?y)$.
For all $\texttt{domain}(p,c) \in sk(G_O)$: $c(?x) \leftarrow p(?x,?y)$.
For all $\texttt{range}(p,c) \in sk(G_O)$: $c(?y) \leftarrow p(?x,?y)$.

For all d s.t. $\texttt{type}(d,\texttt{Datatype}) \in sk(G_O)$ or $d \in \mathcal{D}$:

$\texttt{Literal}(?x) \leftarrow d(?x)$.
$\neg d(?x) \leftarrow \neg\texttt{Literal}(?x)$.

For all $\texttt{type}(p, \texttt{ContainerMembershipProperty}) \in sk(G_O)$:
$\texttt{member}(?x, ?y) \leftarrow p(?x, ?y)$.
$\neg p(?x, ?y) \leftarrow \neg\texttt{member}(?x, ?y)$.

For all $1 \leq i \leq N_O$:
$\texttt{member}(?x, ?y) \leftarrow \texttt{rdf:\_i}(?x, ?y)$.
$\neg\texttt{rdf:\_i}(?x, ?y) \leftarrow \neg\texttt{member}(?x, ?y)$.

For all $o \in V_O^{N_O}$: $\texttt{Resource}(o) \leftarrow \texttt{true}$.
For all $1 \leq i \leq N_O$: $rdf\_number\_term(\texttt{rdf:\_i}) \leftarrow \texttt{true}$.

For each $o =$ "s"^^$d \in V_O^{N_O}$ s.t. $d \in \mathcal{D}$ and s is well-typed according to datatype d:
$well\_typed\_d(o) \leftarrow \texttt{true}$.

For each $o =$ "s"^^$d \in V_O^{N_O}$ s.t. $d \in \mathcal{D}$ and s is not well-typed according to datatype d:
$not\_well\_typed\_d(o) \leftarrow \texttt{true}$.

For each $Nam_{O'}$ appearing as qualification of a triple in P_O or $Nam_{O'} \in Imported_O^{\texttt{cl}} \cup Imported_O^{\texttt{pr}}$:

(1) $\texttt{Resource}(?x) \leftarrow \texttt{Resource}(?x)@Nam_{O'}$.

(2) For all $d \in \mathcal{D}$:
$well\_typed\_d(?x) \leftarrow well\_typed\_d(?x)@Nam_{O'}$.
$not\_well\_typed\_d(?x) \leftarrow not\_well\_typed\_d(?x)@Nam_{O'}$.

(3) $rdf\_number\_term(?x) \leftarrow rdf\_number\_term(?x)@Nam_{O'}$.

For all $d \in \mathcal{D}$:
$d(?x) \leftarrow well\_typed\_d(?x)$.
$\texttt{false} \leftarrow d(?x), not\_well\_typed\_d(?x)$.

$\texttt{ContainerMembershipProperty}(?x) \leftarrow rdf\_number\_term(?x)$.
$\texttt{domain}(?x, \texttt{Resource}) \leftarrow rdf\_number\_term(?x)$.
$\texttt{range}(?x, \texttt{Resource}) \leftarrow rdf\_number\_term(?x)$.

For all RDF, RDFS, ERDF axiomatic triples $p(x, y)$ with terms in $V_{RDF}^1 \cup V_{RDFS} \cup V_{ERDF}$:

$p(x, y) \leftarrow \texttt{true}$.

If $p(x, y) = \texttt{subClassOf}(c_1, c_2)$: $c_2(?x) \leftarrow c_1(?x)$.
If $p(x, y) = \texttt{subPropertyOf}(p_1, p_2)$: $p_2(?x, ?y) \leftarrow p_1(?x, ?y)$.
If $p(x, y) = \texttt{domain}(p', c)$: $c(?x) \leftarrow p'(?x, ?y)$.
If $p(x, y) = \texttt{range}(p', c)$: $c(?y) \leftarrow p'(?x, ?y)$.

For all $\texttt{domain}(p, c)$ in the RDFS axiomatic triples with $p \neq \texttt{rdf:\_}i$, for $i \in \mathbb{N}$:
$\texttt{Property}(p) \leftarrow \texttt{true}$.
$\texttt{Class}(c) \leftarrow \texttt{true}$.

$\texttt{Class}(\texttt{TotalClass}) \leftarrow \texttt{true}$.
$\texttt{Class}(\texttt{TotalProperty}) \leftarrow \texttt{true}$.

For all $\texttt{type}(c, \texttt{TotalClass}) \in sk(G_O)$:
$\neg c(?x) \leftarrow \texttt{Resource}(?x), {\sim}c(?x)$.

$c(?x) \leftarrow \texttt{Resource}(?x), {\sim}\neg c(?x)$.

For all $\texttt{type}(p, \texttt{TotalProperty}) \in sk(G_O)$:
$\neg p(?x, ?y) \leftarrow \texttt{Resource}(?x), \texttt{Resource}(?y), {\sim}p(?x, ?y)$.
$p(?x, ?y) \leftarrow \texttt{Resource}(?x), \texttt{Resource}(?y), {\sim}\neg p(?x, ?y)$.

Let $\texttt{subClassOf}^*$ be the transitive closure of the relationship $\texttt{subClassOf}$ appearing in $sk(G_O)$ and in the RDFS or ERDF axiomatic triples.
For all $\texttt{subClassOf}^*(c_1, c_2)$: If
(i) $\neg\texttt{type}(o, c_2)$ appears in $sk(G_O)$, or
(ii) $\neg\texttt{type}(o, c_2)$ appears in $Concl(r)$, for an $r \in P_O$, or
(iii) $\texttt{type}(c_2, \texttt{Datatype}) \in sk(G_O)$, or
(iv) $\texttt{type}(c_2, \texttt{TotalClass}) \in sk(G_O)$, or
(v) $c_2 \in Imported_O^{\texttt{cl}}$, or
(vi) $c_2 \in \mathcal{D}$:

$$\neg c_1(?x) \leftarrow \neg c_2(?x). \qquad (\star)$$

Let $\texttt{subPropertyOf}^*$ be the transitive closure of the relationship $\texttt{subPropertyOf}$ appearing in $sk(G_O)$ and in the RDFS axiomatic triples.
For all $\texttt{subPropertyOf}^*(p_1, p_2)$: If
(i) $\neg p_2(x, y)$ appears in $sk(G_O)$, or
(ii) $\neg p_2(x, y)$ appears in $Concl(r)$, for an $r \in P_O$, or
(iii) $\texttt{type}(p_2, \texttt{ContainerMembershipProperty}) \in sk(G_O)$, or
(iv) $\texttt{type}(p_2, \texttt{TotalProperty}) \in sk(G_O)$, or
(v) $p_2 \in Imported_O^{\texttt{pr}}$, or
(vi) $p_2 = \texttt{rdf:\_}i$, for $0 \leq i \leq N$:

$$\neg p_1(?x, ?y) \leftarrow \neg p_2(?x, ?y). \qquad (\star)$$

Note that rules $(\star)$ are introduced because $\neg c_2(x)$ (resp. $\neg p_2(x, y)$) appears in the head of a rule in the rest of Π_O^{rERDF} or in the head of a rule in $\Pi_{G_O} \cup \Pi_{P_O}$.

We define the *logic program derived from* O as follows:

$$\Pi_O = \Pi_{G_O} \cup \Pi_{P_O} \cup \Pi_O^{rERDF}.$$

Note that the logic programs P_{KB_1} and P_{KB_2} of the knowledge bases KB_1 and KB_2 of Figure 1, contain part of Π_{O_1} and Π_{O_2}, for the restricted r-ERDF ontologies O_1 and O_2 of Figure 2, respectively.

Let $tr(\Pi_O)$ derived from Π_O after replacing all non-qualified literals $[\neg]p(.)$ in Π_O by $[\neg]p\_Nam_O(.)$ and all qualified literals $[\neg]p(.)@Nam_{O'}$ in Π_O by $[\neg]p\_Nam_{O'}(.)$.

Finally, we define:

$$\Pi_O^{\mathcal{G}} = \bigcup\{tr(\Pi_{O'}) \mid O' \in D_O^{\mathcal{G}}\}.$$

Note that $\Pi_O^{\mathcal{G}}$ is an Extended Logic Program (ELP).

Definition 14 (restricted r-ERDF formula). Let F be a simple r-ERDF formula. We say that F is a *restricted* r*-ERDF formula* if for all $[\neg]\texttt{type}(o, c)$ appearing in F, it holds that $c \notin Var$.

Below, we provide an auxiliary definition, where we define the transformation of a restricted r-ERDF formula F to an ELP formula $tr_O(F)$, where $O \in \mathcal{G}$.

Definition 15 (transformation of a restricted r-ERDF formula to an ELP formula). Let $O \in \mathcal{G}$ and let F is be a restricted r-ERDF formula. We define $tr_O(F)$ to be the ELP formula derived from F after:

1. replacing all non-qualified literals $[\neg]\texttt{type}(o, c)$ in F by $[\neg]c\_Nam_O(o)$ and all qualified literals $[\neg]\texttt{type}(o, c)@Nam_{O'}$ in F by $[\neg]c\_Nam_{O'}(o)$ and
2. replacing all non-qualified literals $[\neg]p(x, y)$ in F, for $p \neq \texttt{type}$, by $[\neg]p\_Nam_O(x, y)$ and all qualified literals $[\neg]p(x, y)@Nam_{O'}$, for $p \neq \texttt{type}$, in F by $[\neg]p\_Nam_{O'}(x, y)$.

It is easy to see that the following proposition holds which shows equivalence of ASP reasoning [25] between the ELPs $\Pi_{O,\mathcal{G}}$ and $\Pi^{\mathcal{G}}_O$. Note that the symbols $\Pi_{O,\mathcal{G}}$ and L^O_F are defined in the electronic Appendix and in [2]. Additionally, note that ELP $\Pi_{O,\mathcal{G}}$ is grounded, whereas ELP $\Pi^{\mathcal{G}}_O$ may have variables. The proof is based on the definitions of $\Pi_{O,\mathcal{G}}$ and $\Pi^{\mathcal{G}}_O$, based on the fact that O is a restricted r-ERDF ontology (Def. 12) and $\mathcal{G}$ is a restricted modular ERDF ontology (Def. 13). We brought logic program $\Pi_{O,\mathcal{G}}$, which is defined for the more general case of simple modular ERDF ontologies and it is grounded, into an equivalent logic program $\Pi^{\mathcal{G}}_O$ with variables that can be efficiently evaluated by our algorithms.

Proposition 5. Let $O \in \mathcal{G}$ and let F be a restricted r-ERDF formula over $\{Nam_{O'} \mid O' \in D^{\mathcal{G}}_O\}$. Then, $Ans^{AS}_{\Pi_{O,\mathcal{G}}}(L^O_F) = Ans^{AS}_{\Pi^{\mathcal{G}}_O}(tr_O(F))$.

For self-containment, we state below Proposition 1 in the electronic Appendix:
`http://users.ics.forth.gr/~analyti/Papers_2/Appendix_Modular_KBs.pdf`
(also Proposition 11 in [2]).

Proposition 6 ([2]). Let $\mathcal{R}$ be a simple modular ERDF ontology, let $O \in \mathcal{R}$, and let F be a simple r-ERDF formula over $\{Nam_{O'} \mid O' \in D^{\mathcal{R}}_O\}$. Then, $Ans^{\texttt{st}}_{O,\mathcal{R}}(F) = Ans^{AS}_{\Pi_{O,\mathcal{R}}}(L^O_F)$.

From Proposition 5 and Proposition 6, it follows immediately the following Corollary. Recall that a restricted modular ERDF ontology is a special case of a simple modular ERDF ontology. The Corollary indicates that the *modular stable answers* of a restricted r-ERDF formula F w.r.t. O and $\mathcal{G}$ [2], denoted by $Ans^{\texttt{st}}_{O,\mathcal{G}}(F)$, can be computed through Answer Set Programming on $\Pi^{\mathcal{G}}_O$. Note that a restricted r-ERDF formula is a simple restricted r-ERDF formula.

Corollary 1. Let $O \in \mathcal{G}$ and let F be a restricted r-ERDF formula over $\{Nam_{O'} \mid O' \in D^{\mathcal{R}}_O\}$. Then,

$$Ans^{\texttt{st}}_{O,\mathcal{G}}(F) = Ans^{AS}_{\Pi^{\mathcal{G}}_O}(tr_O(F)).$$

Let $O \in \mathcal{G}$. We define the *knowledge base derived from* O as follows:

$$KB_O = \langle Nam_O, \Pi_O \rangle$$

Additionally, we define the *modular knowledge base derived from* $\mathcal{G}$ as follows:

$$MKB_{\mathcal{G}} = \{KB_O \mid O \in \mathcal{G}\}$$

Now, we provide an auxiliary definition, where we define the transformation of a restricted r-ERDF formula F to a KB query $tr'(F)$.

Definition 16 (transformation of a restricted r-ERDF formula to a KB query). Let F is be a restricted r-ERDF formula. We define $tr'(F)$ to be the query derived from F after replacing all $[\neg]\texttt{type}(o, c)$ appearing in F by $[\neg]c(o)$ and all $[\neg]\texttt{type}(o, c)@Nam_{O'}$ appearing in F by $[\neg]c(o)@Nam_{O'}$. □

Let $O \in \mathcal{G}$ and let F be a restricted r-ERDF formula over $\{Nam_{O'} \mid O' \in D_O^{\mathcal{G}}\}$. Recall that the call $Get\_query\_program(Nam_O, tr'(F))$, made by the $Answers\_to\_query($ $Nam_O, tr'(F),$"$cautious$") call, returns an ELP Π to be evaluated through ASP. As we will see in Theorem 2 below, the evaluation of Π provides the *modular stable answers* of F w.r.t. O and $\mathcal{G}$, denoted by $Ans^{\texttt{st}}_{O,\mathcal{G}}(F)$ [2].

It is easy to see that following Proposition holds, which shows that:
$\Pi \subseteq \Pi_O^{\mathcal{G}} \cup \{Answers(\overline{x}) \leftarrow tr(tr'(F))\}$.This is because $Get\_query\_program(Nam_O, tr'(F))$ retrieves only the parts of $\Pi_O^{\mathcal{G}}$ that are required for answering the query $tr'(F)$ or deriving inconsistency of $\Pi_O^{\mathcal{G}}$.

Proposition 7. Let $O \in \mathcal{G}$ and let F be a restricted r-ERDF formula over $\{Nam_{O'} \mid O' \in D_O^{\mathcal{G}}\}$. Additionally, let v be (i) "yes", if $Var(F) = \emptyset$, or (ii) a mapping $v : Var(F) \rightarrow V_{O,\mathcal{G}}$, if $Var(F) \neq \emptyset$. Consider the modular knowledge base $MKB_{\mathcal{G}}$. Then,
$Get\_query\_program(Nam_O, tr'(F)) \subseteq \Pi_O^{\mathcal{G}} \cup \{Answers(\overline{x}) \leftarrow tr(tr'(F))\}$, while there are cases where the equality holds.

From Corollary 1 and Proposition 7, it follows Theorem 2, provided below. Theorem 2 shows that the *modular stable answers* of a restricted r-ERDF formula F w.r.t. O and $\mathcal{G}$ can be computed through the call $Answers\_to\_query(Nam_O, tr'(F),$"$cautious$").

Theorem 2. Let $O \in \mathcal{G}$ and let F be a restricted r-ERDF formula over $\{Nam_{O'} \mid O' \in D_O^{\mathcal{G}}\}$. Consider the modular knowledge base $MKB_{\mathcal{G}}$. Then, the following hold:

1. If $Answers\_to\_query(Nam_O, tr'(F),$"$cautious$"$)= \bot$ then there is no modular stable model of O w.r.t. $\mathcal{G}$.
2. If $Answers\_to\_query(Nam_O, tr'(F),$"$cautious$"$)\neq \bot$ then

$$Ans^{\texttt{st}}_{O,\mathcal{G}}(F)=Answers\_to\_query(Nam_O, tr'(F),\text{"}cautious\text{"}).$$

Concerning the proof of Theorem 2, note that in the case that $Answers\_to\_query(Nam_O,$ $tr'(F),$ "$cautious$"$)= \bot$ then the logic program $\Pi_O^{\mathcal{G}}$ is inconsistent. Therefore, based on Corollary 1, there is no modular stable model of O w.r.t. $\mathcal{G}$. In the case that $Answers\_to\_query(Nam_O,$ $tr'(F),$ "$cautious$"$)\neq \bot$, then $Answers\_to\_query(Nam_O, tr'(F),$ "$cautious$"$)=Ans^{AS}_{\Pi_O^{\mathcal{G}}}(tr_O(F))$. Thus, based on Corollary 1:

$$Ans^{\texttt{st}}_{O,\mathcal{G}}(F)=Answers\_to\_query(Nam_O, tr'(F),\text{"}cautious\text{"}).$$

Let $O \in \mathcal{G}$. The following proposition gives the complexity of query answering on ELP $\Pi_O^{\mathcal{G}}$ through Answer Set Programming.

We define:

$$size(O, \mathcal{G}) = \sum\{\text{size of } (G_{O'} \cup P_{O'}) \mid O' \in D_O^{\mathcal{G}}\}$$

Additionally, we define:

$$graph\_size(O, \mathcal{G}) = \sum\{\text{size of } G_{O'} \mid O' \in D_O^{\mathcal{G}}\}$$

The following Proposition holds whose proof is provided in Appendix A.

Proposition 8. Let $O \in \mathcal{G}$ and let F be a restricted $\mathtt{r}$-ERDF formula over $\{Nam_{O'} \mid O' \in D_O^{\mathcal{G}}\}$. Additionally, let v be (i) "yes", if $Var(F) = \emptyset$, or (ii) a mapping $v : Var(F) \to V_{O,\mathcal{G}}$, if $Var(F) \neq \emptyset$. The problem of establishing whether $v(tr_O(F))$ is a cautious consequence of $\Pi_O^{\mathcal{G}}$ is:

1. co-NP$^{\text{NP}}$-complete w.r.t. $size(O, \mathcal{G})$ and
2. co-NP-complete w.r.t. $graph\_size(O, \mathcal{G})$.

In the subsequent subsections, we briefly describe further applications of our framework.

4.2 SPARQL 1.1 seen as a Rule Language

Networked Graphs [39] allow to define RDF graph views through SPARQL 1.0 CONSTRUCT queries [37], which can be recursive. The authors provide a well-founded based semantics [23] to these recursive graphs. A distributed implementation is reported in the paper based on the alternating fixpoint [21], where computations are iterated through the network until alternating fixpoints are computed.

In [36], the same idea of network graphs is extended to SPARQL 1.1 CONSTRUCT statements [27]. In this work, the authors provide a translation of SPARQL 1.1 queries to extended logic programs that can be evaluated through answer set programming. They indicate that SPARQL 1.1 can be seen as a rule language where named construct statements can be seen as rules. In fact, through their translation of SPARQL 1.1 CONSTRUCT queries to extended logic programs, a modular knowledge base can be constructed and our algorithms can be used for query evaluation. For our translation, the constraints that have to be imposed is that no variable should be used in the property position of an RDF triple, blank nodes should not exist in the construct template and the assignment pattern BIND that creates new values is not allowed. The details of the translation of recursive named SPARQL 1.1 CONSTRUCT statements to a modular knowledge base will be given in a subsequent paper.

4.3 Maximal Weak Model Semantics and Minimal Weak Model Semantics

The work in [9] considers distributed consistent P2P deductive databases with integrity constraints which are interacting by means of mapping rules. In the framework, weak negation is allowed only in the integrity constraints. A deductive database imports maximal sets of atoms from the indicated, according to the mapping rules, deductive databases, as long as local integrity constraints are not violated. The proposed semantics, called *maximal weak model semantics*, can be computed by means of a prioritized logic program. However, no distributed algorithms are provided.

The authors in [10] consider inconsistent P2P deductive databases with integrity constraints interrelated by means of mapping rules. Again weak negation is allowed only in the integrity constraints. The authors develop a semantics, called *minimal weak model semantics*, where each peer just imports minimal set of atoms, from the indicated, according to the mapping rules, peer in order to restore consistency. A prioritized logic program is defined which computes the minimal weak model semantics. However, no distributed algorithms are provided.

In [11], the above two frameworks and their corresponding semantics are considered but these are computed through *centralized disjunctive logic programs*. Each such disjunctive logic program can be straightforwardly transformed to a modular knowledge base and thus, our algorithms can be used for query answering, providing results to [9] and [10].

4.4 Multi-context Reasoning

In [7], a general framework of knowledge bases of different logics which are associated though possibly non-monotonic bridge rules is considered. Then, the *ground equilibriums* for certain such multi-context systems are defined, including knowledge bases of normal logic programs under stable model semantics. The following definition relates our work with ground equilibriums.

Proposition 9. Let $\mathcal{K}$ be a modular knowledge base consisting of knowledge bases that do not contain disjunction and strong negation. Let $KB \in \mathcal{K}$ and let $D^{\mathcal{K}}_{KB} = \{KB_1, ..., KB_m\}$. Let the bridge rules br_i of KB_i be the rules in KB_i that contain qualified literals. Let the *multi-context system* $W = (C_1, ..., C_m)$, where $C_i = (NLP, KB_i, br_i)$ and NLP is the logic of normal logic programs under stable model semantics [24]. Then, the following hold:

– A valuation v is an certain or brave answer to a query q posed to a knowledge base KB_i based on the *belief states* of W [7] iff v is a centain or brave stable answer to a query q on KB_i over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$.

The proof of Proposition 9 follows easily from Definitions 12 and 13 of [7].

4.5 Contextually Closed Semantics of Modular Knowledge Bases

We would like also to mention a framework for modular knowledge bases $\mathcal{K}$, proposed in [35], under the *Contextually Closed Semantics*. There, a knowledge base $KB \in \mathcal{K}$ is a normal logic program with atoms possibly qualified over other knowledge bases in $\mathcal{K}$. It is assumed that weakly negated atoms in the body of the rules are always qualified. The *contextually closed* `SMS` and *contextually closed* `WFS` semantics of a knowledge base $KB \in \mathcal{K}$ are defined, through the stable model semantics [24] and well-founded semantics [23] of a normal logic program KB_{CC}, generated from KB and the knowledge bases in $D^{\mathcal{K}}_{KB}$, respectively. In particular, all *simple atoms* appearing in the program of a knowledge base $KB' \in D^{\mathcal{K}}_{KB}$ are qualified by the name of the knowledge base KB'. The resulting rules union the original rules of the program of the knowledge base $KB \in \mathcal{K}$ form the logic program KB_{CC}.

It holds that the stable answers to a query q to a knowledge base KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$ under our framework coincide with the answers to a query q to KB over $\{Nam_{KB'} \mid KB' \in D^{\mathcal{K}}_{KB}\}$ under the *contextually closed* `SMS` semantics. Thus, our algorithms can be used for providing answers to queries according to the *contextually closed* `SMS` semantics.

4.6 Taxonomy-based Sources connected by Articulations

In [33], the authors consider taxonomy-based sources connected by articulations. Taxonomy-based sources can be modeled by datalog rules with all atoms having a single common variable. Similarly, articulations can be modeled by mapping rules from a peer to a peer where the head atom and the qualified atoms in the body of the rules have a single common variable. The authors present top-down, asynchronous, distributed algorithms for query answering which, however, result in an exponential time complexity for query answering with respect to the size of the framework. Using our algorithms in the resulting, after transformations, modular knowledge base, query answering can by achieved in *polynomial time* with respect to the size of the framework. This is because the program computed by the *Get_query_program*(q, Nam_{KB}) algorithm, where KB is the knowledge base that received the query q, is a positive logic program, where all atoms appearing in

the rules have a single common variable. Evaluation of such a logic program has polynomial time complexity w.r.t. the size of the program [13]. Recall also Proposition 4, which indicates that the complexity of the algorithm *Get_query_program*(q, Nam_{KB}) is polynomial w.r.t. the $size(KB, \mathcal{K})$.

5 Related work

In this paper, we studied the problem of evaluating queries in an pure (unstuctured) P2P network, where each peer holds a logic program (possibly with disjunction, strong and weak negation, and qualified literals - mappings).

Below, we review related work, which have hot been mentioned in Section 4 (Applications). There, we described direct applications of our work, whereas below we describe works that they are just related to our framework of distributed, collaborating knowledge bases, communicating in a P2P fashion for query answering.

In [8], a P2P framework is considered where each peer $\mathcal{S}$ holds a local database and an exported peer schema which is defined based on a set of rules from the local database. Peers are related through mapping rules having conjunctive (possibly existentially quantified) queries on both sides of the rule, each expressed between the exported schema of a peer. For this system, in order to achieve decidability, the authors adopt a new semantics based on epistemic logic. In particular, a peer poses subqueries to related peers and receives, from each one, a datalog program (based on appropriate transformations) that contains the full extensions of the related peer predicates. The returned to the user datalog program is used for providing the answers to the user query. Though the algorithm for a term subquery visits the corresponding peer just once, it is based on synchronous messaging which implies strong delays. Note that our algorithms have been inspired by [8] but they are based on *asynchronous messaging* and support a different framework (with disjunction in the head of the rules of logic programs and negation in the head and body of the rules under the Answer Set Semantics but not existential variables in the heads of the rules, as in [8]).

The framework in [40] considers (i) rules with no negation but possibly qualified atoms and (ii) relations (also suggested in [41, 38]), mapping the objects of one peer to the objects of another peer[2]. A distributed query answering algorithm is presented based on synchronous messaging. However, the algorithm performs poorly compared to our framework. This is because when a peer sends subqueries to the appropriate peers it waits for their answers. These peers send further subqueries to related peers waiting for their answers and an iteration is created, until a fixpoint on the answers is reached. The fist answer is send based on the local data. These iterations, according to [18], have EXPTIME node complexity. In our case, no iteration until fixpoint is required. Additionally, asynchronous messaging is supported.

In [20, 19], the authors consider a framework where each peer is associated with a local database and P2P mapping rules between two peers, which contain (possibly existentially quantified) conjunctive queries in both sides of the rule. Epistemic logic [18] is used for defining the semantics of the framework. The algorithm is based on asynchronous messaging and starts at a peer which sends subqueries to neighbor peers according to the corresponding mapping rules. When a peer receives a subquery, it replies back based on its own data and propagates sub-queries similarly to neighbor nodes. When a peer receives an answer, it stores the answer locally and evaluates the involved mapping rule. Then, it sends the result to the requesting peer. This process stops when no new answer tuples are coming to the peer, that is until a fixpoint is reached. However, the proposed

[2] In our framework, these relations correspond to the identity relation.

update algorithm have 2EXPTIME node complexity. Note that in our case, no iteration until fixpoint is required and that our node complexity is linear.

As was already mentioned in Section 4.3, in [11], the *maximal weak model semantics*, originally defined in [9], are computed through a centralized disjunctive logic program P. In [4], it has been shown that if P is *head-cycle-free* (HCF), that is, if there are no two distinct atoms in the head of the rules of P mutually dependent by positive recursion then P can be transformed into an equivalent normal logic program P' that has the same stable models as P. Assuming P' is HCF, and the mapping rules between the knowledge bases are acyclic, the authors provide a distributed algorithm based on *synchronous* messaging that computes the *well-founded model* [22] of P' in polynomial data complexity time. It is well known that the well-founded model of P' is a subset of every stable model of P'. Thus, in this case the authors provide an approximation of the *maximal weak model semantics*.

The work in [12], extends the works in [9, 10] described in Section 4.3, by allowing weak negation in the local rules and by considering both types of of mapping rules considered in [9, 10]. The semantics, called *Max-Min weak model semantics*, are computed through a centralized prioritized logic program with two levels of priorities. As in [9, 10], no distributed algorithms are provided.

In [6], a multi-context system framework that is based on propositional defeasible logic is proposed. In particular, the authors develop a P2P distributed algorithm for query evaluation, where each query is a propositional literal l issued to some peer. Each peer is a defeasible theory and peers are connected by defeasible rules. It also exists a total preference ordering of peers for resolving conflicts when conflicting information is imported. The proposed P2P distributed algorithm is based on synchronous messaging. In our case, we support disjunctive logic rules with negation and n-ary (possibly qualified) predicates based on the Answer Set Semantics and provide answers, in the case of no inconsistency of $P^{\mathcal{K}}_{KB}$, to a user query posed to a knowledge base KB. Our theory is also based on asynchronous messaging with obvious advantages.

In [14], the authors present P2P algorithms for model building in heterogeneous non-monotonic multi-context systems, where P2P mappings are non-monotonic rules. The models correspond to *equilibria* as defined in [7] which (as shown in [7]) they do not support groundedness. In contrast our model is based on the Answer Set Semantics. Moreover the complexity of for computing the equilibria through the provided distributed algorithms in [14] is exponential.

In [15], an *RDF Peer System* is considered which consists of (i) a set of peers, each holding an RDF graph [30], (ii) a set of *graph mapping assertions*, mapping *graph pattern queries*[3] on the RDF graph of a peer to the RDF graph of another peer, and (iii) a set of equivalence mappings between the IRIs appearing in two peers. The graph mapping assertions are translated into rules where the body and the head are existential conjunctive queries over the two peers. Additionally, the equivalence mappings between IRIs are translated into rules where the head and the body are atoms of ternary predicates. The authors identify conditions on the mapping rules (linear, sticky, or sticky-join) under which an original conjunctive SPARQL query on an RDF Peer System can be equivalently converted into a union of conjunctive SPARQL queries on the original sources (that is, SPARQL queries that do not take account of the mappings). In this case, the latter query can be used for sound and complete query answering. However, the authors show that this equivalent SPARQL query translation is not possible for all kinds of graph mapping assertions. In contrast, in our framework, sound and complete answering is always achieved.

[3] A *graph pattern query*, as defined in [15], can always be translated to a *conjunctive SPARQL query* [37], that is a SPARQL query containing only the AND operator.

In [1], a P2P-based architecture for publishing and querying RDF data, called *PIQNIC*, is proposed. In PIQNIC, an RDF graph is broken into *fragments* based on common predicates of its RDF triples. These fragments are distributed and replicated in the P2P network. Query processing proceeds are follows: A SPARQL query q is received at a node n which evaluates the q's triple patterns starting from its local database and sending corresponding queries to its neighbors which may in turn forward them to theirs neighbors. The user querying node n receives partial results directly from the queried nodes and computes the final answer based on these results and the operations in the query q. In our case, we do not consider SPARQL queries in a fragmented RDF graph but queries on a knowledge base of a modular knowledge base.

In [43], the authors propose an approach based on *structured P2P networks* that publish shareable ontologies on different peers and automatically discover the ontologies useful for a SPARQL query posed to a peer. In particular, when the peer receives a SPARQL query, it breaks it into subqueries whose results are to be unioned, i.e. it converts the query into a logical expression and brings it in disjunctive nornal form (DNF). Then it finds out all indexed ontologies that can provide answers to all triple patterns of each subquery and sends the subquery to each one of the identified ontologies, separateley. When it gets the results of each subquery, it reorganizes them based on the break of the SPARQL query into subqueries and sends the output to the requestor. In our case, we do not consider SPARQL queries on published shareable ontologies but queries on a knowledge base of a modular knowledge base.

In [5], a *peer data exchange system* (PDES) consists of a set of peers, each of them with a local database. Peers may be related through formulas, called *data exchange constraints* (DECs). Additionally, there may be integrity constraints local to each peer. When a query q is posed to a peer P, the decision on what data to consider from the neighbors does not depend only on the DECs but also on the *trust relationships* (*same* or *less*) that P has with its neighbor peers, used for resolving conflicts. A limitation is that the graph of trust relationships *same* or *less* should be *acyclic*. The solutions to a query posed on a peer P may be found through the solution instances found by a sequence of disjunctive logic programs under the answer set semantics. In particular, under the assumption that we have computed the solution instances of the neighbors of P, the logic program for P allows to compute its own solution instances. In our case, we do not handle inconsistency and in the case of an inconsistent $P^{\mathcal{K}}_{KB}$, where $KB \in \mathcal{K}$ is the knowledge base that receives the query, we return $\perp$. Additionally, query answering is achieved through the answer set semantics of *single* disjunctive logic program that is formed based an asynchronous messaging between knowledge bases.

In [42], a distributed approach is proposed to directly publish and share axioms of OWL ontologies on a *structured P2P network*. In particular, if a node has sharable OWL ontologies, it can publish all the axioms in these ontologies. If a node receives a SPARQL query q (note that an OWL ontology is essentially an RDF graph) then all the distributed OWL axioms related to entities appearing in the query q are fetched. Then, at the next step all the distributed OWL axioms of the entities appearing in the OWL axioms fetched at the previous step are also fetched. The steps continue until closure of the fetched OWL axioms is computed, called *Relevance Axiom Closure* of q ($RAC(q)$). Then, query q at the recipient node can be answered using only the axioms in $RAC(q)$ instead of the union of the published OWL ontologies. Similarly, in our case we fetch only the required parts of the logic programs in a modular knowledge base $\mathcal{K}$, for answering a query to a knowledge base $KB \in \mathcal{K}$. The parts are only a subset of $P^{\mathcal{K}}_{KB}$.

6 Conclusions

In this paper, we defined modular disjunctive knowledge bases $\mathcal{K}$ with strong and weak negation, and constraints. Additionally, we defined query answering on a query q posed to a knowledge base $KB \in \mathcal{K}$. The purpose of our work is using asynchronous polynomial-time P2P algorithms based on messaging, to collect from the distributed knowledge bases only the part that is needed for query answering. This includes the parts that are relevant to the query and the parts that are needed to derive inconsistency. For this, we use four messages Ask_pred, $Tell\_pred$, Ask_KB, and $Tell\_KB$ which upon receival by a knowledge base appropriate actions are taken. To support asynchronous messaging several caches are used. When the relevant to the query disjunctive logic program is computed, this is evaluated through Answer Set Programming and for this purpose the DLV system [31] may be used.

We presented several applications of our theory, indicated in Section 4, including the Restricted Modular ERDF Ontologies, SPARQL 1.1 seen as a Rule Language, the Maximal Weak Model Semantics, the Minimal Weak Model Semantics, Multi-context Reasoning, and Contextually Closed Semantics, and Taxonomy-based Sources connected by Articulations. More details are provided to the Restricted Modular ERDF Ontologies framework. The application of our framework to the rest of the applications is more straightforward and is considered for future work.

Obviously, the complexity of query answering does not depend on the polynomial time, asynchronous P2P algorithms but on the supported features of the distributed knowledge bases, namely disjunction in the heads of the rules and negation. These complexity results are provided in the paper as derived from [17], [13], and [31], in Propositions 2 and 3.

Obviously, in the case that the modular knowledge base does not support negation, the messages Ask_KB, and $Tell\_KB$ (which collect the relevant to inconsistency part of $P^{\mathcal{K}}_{KB}$) are not required, since there is no case that inconsistency arises.

We would like to recall that our approach though it has been inspired by [8], it is based on asynchronous messaging and supports a different framework (with disjunction in the head of the rules of logic programs and negation in the head and body of the rules under the Answer Set Semantics but it does not support existential variables in the heads of the rules).

Future work also includes the implementation of our modular knowledge base framework and the restricted modular ERDF ontologies framework.

Appendix A: Proofs

In this Appendix, we provide the proofs of some Theorems and Propositions.

Proof of Theorem 1:
According to Definition 10, $Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q) = Ans^{AS,mode}_{P^{\mathcal{K}}_{KB}}(tr(q))$. Algorithm 3.1 *Answers_to_query*($q, Nam_{KB}, mode$), calls Algorithm 3.2 *Get_query_program*(q, Nam_{KB}). In particular, in Algorithm 3.2, we define q' as q after replacing:

1. all literals $\neg p(.)$ and $\neg p(.)@qual$ by $\neg\_p(.)$ and $\neg\_p(.)@qual$, respectively,
2. all predicates p appearing in a non-qualified literal by p_Nam_{KB}, and
3. all predicates p appearing in a qualified literal with qualification $qual$ by p_qual.

Let the derived query q' have the form:

$$q' = [\sim]pred_1(\overline{t_1}) \wedge ... \wedge [\sim]pred_n(\overline{t_n}) \wedge [\sim]pred'_1(\overline{t_1})@qual_1 \wedge ... [\sim]pred'_m(\overline{t_m})@qual_m$$

For each $i = 1, ..., n$, the asynchronous message $Ask\_pred(TID, pred_i, Nam_{KB})$ is sent to knowledge base KB asking for the definition program of $pred_i$. Additionally, for each $i = 1, ..., m$, the asynchronous message $Ask\_pred(TID, pred'_i, Nam_{KB})$ is sent to knowledge base $qual_i$ asking for the definition program of $pred'_i$. Further, the asynchronous message $Ask\_KB(TID, Nam_{KB})$ is sent to knowledge base KB asking for the definition of all predicates in $P^{\mathcal{K}}_{KB}$ that can cause inconsistency. Then, KB waits for the corresponding *Tell* messages with the requested definition programs. The union of these definitions programs is combined with the query to form a disjunctive logic program P that will be returned to main Algorithm 3.1 for query answering.

1) It is easy to see that since P is safe, the set of literals of $[P]$, denoted by U, is a *splitting set* [32] of $[P^{\mathcal{K}}_{KB}]$. Additionally, P contains the definition of all literals in $P^{\mathcal{K}}_{KB}$ that can cause inconsistency. Thus, if P is inconsistent then $Answers\_to\_query(Nam_{KB}, q, mode) = \bot$.

2) Recall that since P is safe, the set of literals of $[P]$[4], denoted by U, is a *splitting set* [32] of $[P^{\mathcal{K}}_{KB}]$. Thus, according to the *Splitting Set Theorem* [32], A is an answer set of $[P^{\mathcal{K}}_{KB}]$ iff $A = X \cup Y$, where X is an answer set of $[P]$ and Y is an answer set of $e_U([P^{\mathcal{K}}_{KB}] - [P], X)$. We note that $e_U([P^{\mathcal{K}}_{KB}] - [P], X)$ is the set of rules in $P' = [P^{\mathcal{K}}_{KB}] - [P]$ after replacing each literal $L \in U$ appearing in P' either non-weakly negated or weakly negated with `true` if $L \in X$ and with `false` if $L \notin X$ (for more details, see [32]). Thus, $Ans^{AS,mode}_{P}(q') = Ans^{AS,mode}_{P^{\mathcal{K}}_{KB}}(q')$. Note that $q' = tr(q)$. Thus, the algorithm *Answers_to_query*($q, Nam_{KB}, mode$) will return $Ans^{AS,mode}_{P^{\mathcal{K}}_{KB}}(tr(q)) = Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q)$. □

Proof of Proposition 4:
In Algorithm 3.2 *Get_query_program*(q, Nam_{KB}), we define q' as q after replacing:

1. all literals $\neg p(.)$ and $\neg p(.)@qual$ by $\neg\_p(.)$ and $\neg\_p(.)@qual$, respectively,
2. all predicates p appearing in a non-qualified literal by p_Nam_{KB}, and
3. all predicates p appearing in a qualified literal with qualification $qual$ by p_qual.

Let the derived query q' have the form:

$$q' = [\sim]pred_1(\overline{t_1}) \wedge ... \wedge [\sim]pred_n(\overline{t_n}) \wedge [\sim]pred'_1(\overline{t_1})@qual_1 \wedge ... [\sim]pred'_m(\overline{t_m})@qual_m$$

[4] Let P be a logic program. By $[P]$ we denote the instantiation of all rules in P by the constants appearing in P.

For each $i = 1, ..., n$, the asynchronous message $Ask\_pred(TID, pred_i, Nam_{KB})$ is sent to knowledge base KB asking for the definition program of $pred_i$. Additionally, for each $i = 1, ..., m$, the asynchronous message $Ask\_pred(TID, pred'_i, Nam_{KB})$ is sent to knowledge base $qual_i$ asking for the definition program of $pred'_i$. Further, the asynchronous message $Ask\_KB(TID, Nam_{KB})$ is sent to knowledge base KB asking for the definition of all predicates in $P^{\mathcal{K}}_{KB}$ that can cause inconsistency. Then, KB waits for the corresponding *Tell* messages with the requested definition programs. The union of these definitions programs is combined with the query to form a disjunctive logic program P that will be returned to main Algorithm 3.1 for query answering.

Each message $Ask\_pred(TID, pred, Nam_{KB})$, for a predicate name $pred$, sent to knowledge base KB', where $KB' \in D^{\mathcal{K}}_{KB}$, will eventually generate a polynomial number of Ask_pred messages w.r.t. $size(KB, \mathcal{K})$. Similarly, the $Ask\_KB(TID, Nam_{KB})$ message sent to knowledge base KB will eventually generate a polynomial number of Ask_pred messages w.r.t. $size(KB, \mathcal{K})$. For each $Ask\_pred(TID, pred, Nam_{KB''})$ message sent to knowledge base KB' from knowledge base KB'', where $KB', KB'' \in D^{\mathcal{K}}_{KB}$, there is a corresponding generated $Tell\_pred(TID, Nam_{KB'}, pred, P)$ message sent to knowledge base KB'' from knowledge base KB'. The complexity of each Ask_pred and each $Tell\_pred$ message sent to a knowledge base KB', where $KB' \in D^{\mathcal{K}}_{KB}$, is polynomial w.r.t. size of KB'.

Similarly, each message $Ask\_KB(TID, Nam_{KB})$, sent to knowledge base KB', where $KB' \in D^{\mathcal{K}}_{KB}$, will eventually generate a polynomial number of Ask_KB messages w.r.t. $size(KB, \mathcal{K})$. For each $Ask\_KB(TID, Nam_{KB''})$ message sent to knowledge base KB' from knowledge base KB'', where $KB', KB'' \in D^{\mathcal{K}}_{KB}$, there is a corresponding generated $Tell\_KB(TID, Nam_{KB'}, P)$ message sent to knowledge base KB'' from knowledge base KB'. The complexity of each Ask_KB and each $Tell\_KB$ message sent to a knowledge base KB', where $KB' \in D^{\mathcal{K}}_{KB}$, is polynomial w.r.t. size of KB'. □

Proof of Proposition 8:

1)

Hardness) Let $\mathcal{F}$ be a 2-universal quantified boolean formula. Deciding if $\mathcal{F}$ is valid is a co-NP$^{\text{NP}}$-complete problem [44, 34].

In Section 7.2 of [3], we construct a simple ERDF ontology, denoted by $O_{\mathcal{F}} = \langle G_{\mathcal{F}}, P_{\mathcal{F}} \rangle$, such that $\mathcal{F}$ is invalid iff $O_{\mathcal{F}}$ has a $\#n$-stable model, for $n \in I\!N$.

Let O' be the restricted r-ERDF ontology such that $G_{O'} = G_{O_{\mathcal{F}}}$, $P_{O'} = P_{O_{\mathcal{F}}}$, and $Int_{O'} = \{\}$. Let $\mathcal{G}' = \{\mathcal{O}'\}$. Note that $\mathcal{G}'$ is a restricted modular ERDF ontology. It is easy to see that $\Pi^{\mathcal{G}'}_{O'}$ has an answer set iff $\mathcal{F}$ is invalid.

Let $F' = p(s, o) \wedge \neg p(s, o)$. Then, $\Pi^{\mathcal{G}'}_{O'} \models F'$ iff $\Pi^{\mathcal{G}'}_{O'}$ has no answer set iff $\mathcal{F}$ is valid.

Therefore, the complexity of deciding whether $v(tr_O(F))$ is a cautious consequence of $\Pi^{\mathcal{G}}_{O}$ is co-NP$^{\text{NP}}$-hard w.r.t. $size(O, \mathcal{G})$.

Membership) It follows directly from the fact that cautious reasoning on ELP programs with predicates of bounded arity by a constant has combined complexity co-NP$^{\text{NP}}$-complete [16].

2)

Hardness) In Section 7.1 of [3], we construct a restricted r-ERDF ontology O_D. Let $\mathcal{G}_{\mathcal{D}} = \{\mathcal{O}_{\mathcal{D}}\}$. It is easy to see that D is *3-colorable* iff $\Pi^{\mathcal{G}_{\mathcal{D}}}_{O_D}$ has an answer set. Recall that *3-colorability* is an NP-complete problem.

Let $F' = p(s, o) \wedge \neg p(s, o)$. Then, $\Pi^{\mathcal{G}_{\mathcal{D}}}_{O_D} \models F'$ iff $\Pi^{\mathcal{G}_{\mathcal{D}}}_{O_D}$ has no answer set iff D is not *3-colorable*. Therefore, the complexity of deciding whether $v(tr_O(F))$ is a cautious con-

sequence of $\Pi_O^{\mathcal{G}}$ is co-NP-hard w.r.t. $graph\_size(O, \mathcal{G})$.

Membership) It follows directly from the fact that cautious reasoning on ELP programs has data complexity co-NP-complete [13]. □

Appendix B: Table of Symbols

List of Symbols	
Symbol	*Description*
$Cond(r)$	the condition of a rule r
$Concl(r)$	the conclusion of a rule r
Nam_{KB}	the name of a knowledge base KB
$\mathcal{K}$	a modular knowledge base (MKB)
$D^{\mathcal{K}}_{KB}$	the dependencies of a knowledge base KB w.r.t. an MKB
$P^{\mathcal{K}}_{KB}$	the logic program of a knowledge base KB w.r.t. an MKB
$V_{KB,\mathcal{K}}$	the constants appearing in $P^{\mathcal{K}}_{KB}$
$Ans^{\mathtt{st},mode}_{KB,\mathcal{K}}(q)$	the stable answers of query q w.r.t. KB and $\mathcal{K}$ according to $mode \in \{cautious, brave\}$
V_G	the set of IRI references and literals appearing in the ERDF graph G
V_P	the set of IRI references and literals appearing in the r-ERDF program P
$sk(G)$	the skolemization of an ERDF graph G
Nam_O	the name of an ERDF ontology O
$L_O = \langle G_O, P_O \rangle$	the logic of an ERDF ontology O
$Import^{\mathtt{t}}_O$	a (partial) function from $V \cap URI$ to $\mathcal{P}(\mathcal{O}_{\mathtt{nam}} - \{Nam_O\})$ $\quad (\mathtt{t} \in \{\mathtt{cl}, \mathtt{pr}\})$
$Imported^{\mathtt{t}}_O$	$\{x \mid Import^{\mathtt{t}}_O(x) \text{ is defined }\}$ $\quad (\mathtt{t} \in \{\mathtt{cl}, \mathtt{pr}\})$
$\mathcal{R}$	a simple modular ERDF ontology
$D^{\mathcal{R}}_O$	the dependencies of an ERDF ontology O w.r.t. $\mathcal{R}$
$\mathcal{V}^{\#n}_{RDF}$	$\mathcal{V}_{RDF} - \{rdf{:}\_i \mid i > n\}$
$V^{\#n}_O$	$V_{sk(G_O)} \cup V_{P_O} \cup Imported^{\mathtt{pr}}_O \cup Imported^{\mathtt{cl}}_O \cup V^{\#n}_{RDF} \cup \mathcal{V}_{RDFS} \cup \mathcal{V}_{ERDF}$
$V_{O,\mathcal{R}}$	$\cup\{V^{\#n_O}_{O'} \mid O' \in D^{\mathcal{R}}_O\}$
$[P]_V$	instantiation of all rules of a logic program P according to vocabulary V
$\Pi_{O,\mathcal{R}}$	$\bigcup\{\Pi^{O'}_{G_{O'}} \cup [\Pi^{O'}_{P_{O'}}]_{V_{O',\mathcal{R}}} \cup [\Pi^{H_{\mathcal{R}}}_{O'}]_{V_{O',\mathcal{R}}} \mid O' \in D^{\mathcal{R}}_O\}$
$Ans^{\mathtt{st}}_{O,\mathcal{R}}(F)$	the modular stable answers of F w.r.t. ERDF ontology O and $\mathcal{R}$
$\mathcal{G}$	a restricted modular ERDF ontology
Π_O	$\Pi_{G_O} \cup \Pi_{P_O} \cup \Pi^{rERDF}_O$
$\Pi^{\mathcal{G}}_O$	$\bigcup\{tr(\Pi_{O'}) \mid O' \in D^{\mathcal{G}}_O\}$
KB_O	the knowledge base $\langle Nam_O, \Pi_O \rangle$, where $O \in \mathcal{G}$
$MKB_{\mathcal{G}}$	the modular knowledge base $\{KB_O \mid O \in \mathcal{G}\}$

Table 1. Symbols and Description

References

1. C. Aebeloe, G. Montoya, and K. Hose. A Decentralized Architecture for Sharing and Querying Semantic Data. In *The Semantic Web - 16th International Conference, (ESWC-2019)*, volume 11503 of *Lecture Notes in Computer Science*, pages 3–18, 2019.
2. A. Analyti, G. Antoniou, C. V. Damásio, and I. Pachoulakis. A Framework for Modular ERDF Ontologies. *Annals of Mathematics and Artificial Intelligence*, 67(3-4):189–249, 2013.
3. A. Analyti, C. V. Damásio, and G. Antoniou. Extended RDF: Computability and complexity issues. *Annals of Mathematics and Artificial Intelligence*, 75(3-4):267–334, 2015.
4. R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12(1-2):53–87, 1994.
5. L. E. Bertossi and L. Bravo. Consistency and trust in peer data exchange systems. *Theory and Practice of Logic Programming.*, 17(2):148–204, 2017.
6. A. Bikakis, G. Antoniou, and P. Hassapis. Strategies for contextual reasoning with conflicts in ambient intelligence. *Knowledge and Information Systems (KAIS)*, 27(1):45–84, 2011.

7. G. Brewka and T. Eiter. Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. In *22nd AAAI Conference on Artificial Intelligence (AAAI-2007)*, pages 385–390, 2007.
8. D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Logical foundations of peer-to-peer data integration. In *23rd ACM symposium on Principles of database systems (PODS-2004)*, pages 241–251, New York, NY, USA, 2004. ACM Press.
9. L. Caroprese, C. Molinaro, and E. Zumpano. Integrating and Querying P2P Deductive Databases. In B. C. Desai and S. K. Gupta, editors, *Tenth International Database Engineering and Applications Symposium (IDEAS-2006)*, pages 285–290, 2006.
10. L. Caroprese and E. Zumpano. Restoring Consistency in P2P Deductive Databases. In *Scalable Uncertainty Management - 6th International Conference on Scalable Uncertainty Management (SUM-2012)*, pages 168–179, 2012.
11. L. Caroprese and E. Zumpano. A Logic Based Approach for Managing Incompleteness and Inconsistencies in P2P Deductive Databases. In *19th International Symposium on Database Engineering & Applications*, pages 168–173, 2015.
12. L. Caroprese and E. Zumpano. A Logic Framework for P2P Deductive Databases. *Theory Practice of Logic Programming*, 20(1):1–43, 2020.
13. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
14. M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Distributed Evaluation of Nonmonotonic Multi-context Systems. *Journal of Artificial Intelligence Research (JAIR)*, 52:543–600, 2015.
15. M. M. Dimartino, A. Calì, A. Poulovassilis, and P. T. Wood. Peer-to-Peer Semantic Integration of Linked Data. In P. M. Fischer, G. Alonso, M. Arenas, and F. Geerts, editors, *Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015*, volume 1330 of *CEUR Workshop Proceedings*, pages 213–220, 2015.
16. T. Eiter, W. Faber, M. Fink, and S. Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123–165, 2007.
17. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions in Database Systems*, 22(3):364–418, 1997.
18. E. Franconi, G. M. Kuper, A. Lopatenko, and L. Serafini. A Robust Logical and Computational Characterisation of Peer-to-Peer Database Systems. In *First International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P-2003)*, pages 64–76, 2003.
19. E. Franconi, G. M. Kuper, A. Lopatenko, and I. Zaihrayeu. A Distributed Algorithm for Robust Data Sharing and Updates in P2P Database Networks. In *EDBT'04 Intern. Workshop on Peer-to-Peer Computing and Databases (P2P&DB-2004)*, pages 446–455, 2004.
20. E. Franconi, G. M. Kuper, A. Lopatenko, and I. Zaihrayeu. Queries and Updates in the coDB Peer to Peer Database System. In *30th International Conference on Very Large Data Bases (VLDB-2004)*, pages 1277–1280, 2004.
21. A. V. Gelder. The Alternating Fixpoint of Logic Programs with Negation. *Journal of Computer and System Sciences (JCSS)*, 47(1):185–221, 1993.
22. A. V. Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
23. A. V. Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
24. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
25. M. Gelfond and V. Lifschitz. Logic programs with Classical Negation. In *7th International Conference on Logic Programming*, pages 579–597, 1990.
26. M. Gelfond and V. Lifschitz. Classical Negation in Logic programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
27. S. Harris and A. Seaborne. SPARQL 1.1 Query Language. W3C Recommendation, 21 March 2013. Available at `http://www.w3.org/TR/sparql11-query/`.
28. P. Hayes. RDF Semantics. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.

29. P. Hayes and P. F. Patel-Schneider. RDF 1.1 Semantics. W3C Recommendation, 25 February 2014. Available at `http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/`.
30. G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.
31. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
32. V. Lifschitz and H. Turner. Splitting a Logic Program. In P. V. Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 23–37. MIT Press, 1994.
33. C. Meghini and A. Analyti. Query Processing in a P2P Network of Taxonomy-based Information Sources. *Open Journal of Web Technologies (OJWT)*, 3(1):1–25, 2016.
34. C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
35. A. Polleres, C. Feier, and A. Harth. Rules with Contextually Scoped Negation. In *3rd European Semantic Web Conference (ESWC-2006)*, pages 332–347, 2006.
36. A. Polleres and J. P. Wallner. On the relation between SPARQL1.1 and Answer Set Programming. *Journal of Applied Non-Classical Logics*, 23(1-2):159–212, 2013.
37. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 15 January 2008. Available at `http://www.w3.org/TR/rdf-sparql-query/`.
38. M. A. Rahman, M. Masud, I. Kiringa, and A. El-Saddik. Bi-Level Mapping: Combining Schema and Data Level Heterogeneity in Peer Data Sharing Systems. In *3rd Alberto Mendelzon International Workshop on Foundations of Data Management*, 2009.
39. S. Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *17th International Conference on World Wide Web (WWW-2008)*, pages 585–594, 2008.
40. L. Serafini and C. Ghidini. Using Wrapper Agents to Answer Queries in Distributed Information Systems. In *Procs. of the First International Conference on Advances in Information Systems (ADVIS-2000)*, pages 331–340. Springer-Verlag, 2000.
41. L. Serafini, F. Giunchiglia, J. Mylopoulos, and P. A. Bernstein. Local Relational Model: A Logical Formalization of Database Coordination. In *Procs. of the 4th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-2003)*, pages 286–299, 2003.
42. H. Si, Z. Chen, Y. Zhao, and Y. Deng. P2P-Based Publication and Sharing of Axioms in OWL Ontologies for SPARQL Query Processing in Distributed Environment. In *14th Asia-Pacific Web Conference on Web Technologies and Applications*, pages 586–593. Springer, 2012.
43. H. Si, Y. Qi, M. Zheng, Y. Ren, and L. Yu. Structured peer-to-peer-based publication and sharing of ontologies to automatically process SPARQL query on a semantic sensor network. *International Journal of Distributed Sensor Networks*, 14(10), 2018.
44. L. J. Stockmeyer and A. R. Meyer. Word Problems Requiring Exponential Time: Preliminary Report. In *Fifth Annual ACM Symposium on Theory of Computing (STOC'73)*, pages 1–9, 1973.