Specifying and Reasoning with Underspecified Knowledge Bases Using Answer Set Programming

Vinay K. Chaudhri
AI Center
SRI International
Mento Park, California, USA

Tran Cao Son
Computer Science Department
New Mexico State University
Las Cruces, New Mexico, USA

Abstract

A large and complex knowledge base that models some aspect of the real world can rarely be fully specified. Two examples of such underspecification are that (*i*) some of the cardinality constraints are omitted; (*ii*) some properties of all individual instances of a class are specialized across a class hierarchy, but specific references to which particular values are specialized are omitted. Such knowledge bases are of great practical interest as they are the basis of an empirically tested knowledge acquisition system that has been used to construct a knowledge base from a significant portion of a biology textbook. In this paper, we formalize an underspecified knowledge base using answer set programming, and give a set of rules called UMAP that support inheritance reasoning in such a knowledge base.

Introduction

Building large and complex knowledge bases (KBs) has been an intensive topic of artificial intelligence research in general and knowledge representation and reasoning research in particular (Lenat 1995). Clark & Porter proposed a method toward this goal in which a KB can be built from reusable components that can be composed automatically in response to novel questions (Clark and Porter 1997; Clark et al. 2000). Their AAAI'97 paper on this topic won the best paper prize and this approach was implemented in a knowledge representation and reasoning system called KM (Clark and Porter 2011), which has been the foundation of two knowledge acquisition systems, SHAKEN (Barker et al. 2003) and AURA (Gunning et al. 2010). Both of these systems have been empirically tested in acquiring knowledge from domain experts. The success is attributed to two key features of the KM system: prototypes and heuristic unification. There has, however, been no published computational characterization of these features. Such characterization is of interest for at least two reasons: other representation languages could support these features for similar problems, and the substantial KBs built using this approach could be used for reasoning systems other than KM. In addition, this formalization provides a way to analyze how the representation and reasoning in AURA could be improved.

A knowledge base in KM is a collection of frames representing classes and individuals. Classes are organized into a class hierarchy. Each frame is associated with a set of slots and slot values. Each slot value can be a class, individual, or a value. For example, the statement "Every Car has an engine and a tank connected to the engine" can be represented by the following axiom in KM:

```
  (every Car has
    (has-engine ((a Engine called E1)))
    (has-tank ((a Tank with                    (1)
      connected-to (((the has-engine
        of Self) called E1)))))
```

This axiom applies to all individual instances of the class `Car` and defines two slots `has-engine` and `has-tank`. The slot `has-engine` (resp. `has-tank`) is associated with a frame that refers to a Skolem instance of class `Engine` (resp. `Tank`). The Skolem instance `Tank` has a further slot `connected-to` that has a value that is same as the Skolem instance of `Engine` that is the value of the `has-engine` slot. The use of "`called E1`" is similar to using a variable name in traditional logical syntax. KM also provides an alternative syntax called *prototypes* to encode such axioms. The axioms such as (1) are the most frequently occurring axiom pattern in the biology textbook KB. We note that KM identifies (1) with the following first-order logic formula:

$$\forall x.[instance\text{-}of(x, Car) \Rightarrow \\ \exists e,t.[instance\text{-}of(e, Engine)\wedge \\ instance\text{-}of(t, Tank) \wedge has\text{-}engine(x, e)\wedge \\ has\text{-}tank(x, t) \wedge connected\text{-}to(e, t)]]$$

While constructing a KB, it is common to specialize properties of classes along the class hierarchy. While doing so, one may need to refer to a Skolem instance that was introduced in one of the superclasses. For example, assume that we add to the KB the following axioms:[1]

$$\forall x.[instance\text{-}of(x, Suburban) \Rightarrow \\ instance\text{-}of(x, Car)] \tag{2}$$

and

$$\forall x.[instance\text{-}of(x, Suburban) \Rightarrow \\ \exists e.[instance\text{-}of(e, Engine)\wedge \\ has\text{-}engine(x, e) \wedge size(e, Large)]] \tag{3}$$

These axioms state that `Suburban` is a subclass of `Car`, and every suburban has a large engine. The axioms (1)-(3) are an example of an underspecified KB in the sense that they omit

[1]For simplicity of the presentation, we omit the KM representation of the axioms.

the relationship between an `Engine` introduced in axiom (1) and the `Engine` introduced in axiom (3). In one possible interpretation, the engines mentioned in axioms (1) and (3) refer to the same individual and in another interpretation they refer to different values.

A system can handle such underspecification in the following ways:

1. While writing (3), provide a mechanism using which a user can explicitly state that the engine in axiom (3) is a specialization of the engine introduced in (1). We refer to this approach as *explicit coreference*. For instance, the formulae (1) and (3) should be given as:
$$\begin{array}{l}\forall x.[instance\text{-}of(x, Car) \Rightarrow \\ \quad [instance\text{-}of(E_1(x), Engine) \wedge \\ \quad instance\text{-}of(T_1(x), Tank) \wedge \\ \quad has\text{-}engine(x, E_1(x)) \wedge \\ \quad has\text{-}tank(x, T_1(x)) \wedge \\ \quad connected\text{-}to(E_1(x), T_1(x))]]\end{array} \tag{4}$$
and
$$\begin{array}{l}\forall x.[instance\text{-}of(x, Suburban) \Rightarrow \\ \quad size(E_1(x), Large)]\end{array} \tag{5}$$
2. Add a cardinality constraint to the KB saying that cars have exactly one engine. We refer to this approach as the *cardinality constraint*.
3. Support a default reasoning mechanism that can draw intuitive conclusions with the axioms as they are currently written. We refer to this approach as the *underspecified knowledge base*.

The explicit coreference approach has the advantage that it leaves no ambiguity, but it has a major disadvantage: it breaks the modularity of axioms in that while writing an axiom, we must refer to other axioms. While creating the class `Suburban`, we must refer to its superclass `Car`, and explicitly say which specific engine value we are specializing. If after having created `Car` and `Suburban`, suppose we introduce a superclass `Vehicle` and we want this class to also have a value for `has-engine`, we must refer to its subclasses and make sure all of them now specialize the `Engine` value introduced in `Vehicle`. In case of multiple inheritance, for example, if a `Car` were to be a subclass of `Vehicle` and `Gas-driven` objects each of which provides an `Engine`, the knowledge base author must resolve how the multiple inheritance must be handled while defining `Car`. The approach breaks down completely when we need to answer a novel question about an object that must be an instance of multiple classes. But, there is no pre-existing class which specifies how values from multiple classes must be combined. The extra work of fully specifying correfernces starts to become onerous and is unnecessary for a large number of practical situations in which the *specificity principle* in default reasoning, which states that when conflicts arise more specific information overrides less specific one, could be adapted for answering such a question. Our adaption of this principle will, however, maintain the principle of inheritance reasoning in that it does not affect multiple inheritance of non-conflicting values. For example, since `Suburban` is a subclass of `Car`, we have that the slot specification at the class `Suburban` is more preferred than (and thus could override) the slot specification at the class `Car`, resulting in a conclusion that a `Suburban` has only one `Engine`. Furthermore, if a class inherits non-conflicting values from the multiple super classes, for example, a `Subarau` has a large engine (defined for the class `Suburban`), and that a `Subarau` has a gas driven engine (defined for the class `Gas driven Object`), our adaptation of the specificity principle will allows us to conclude that a `Subarau` has a large and gas driven engine. The disadvantage of the explicit coreference approach is especially a limiting factor when the knowledge base is to be authored by a domain expert who is not well-versed in logical knowledge representation and the knowledge base must evolve over a period of time.

In the cardinality constraint approach we can conclude that the axioms (1) and (3) musr refer the the same `Engine` value. It can work for many situations, but in some cases it is incorrect to use such constraints. For example, race cars may have more than one engine, and it is too strong to add a constraint that every car has exactly one engine. The KM system supports such reasoning using cardinality constraints when the constraints are available, but in situations where it is incorrect to add constraints such reasoning cannot be used.

In an underspecified knowledge base approach, we assume that the inherited and locally defined engines must be the same unless there is a reason to believe otherwise. Thus, the class definitions for `Car`, `Suburban`, `Vehicle`, and `Gas-driven` objects could be written without making any reference to each other. At the time of answering questions, the reasoning system is able to resolve the underspecified reference between the `Engine` values, and as a default inference, assumes that they must be the same. If the KB contained knowledge to the contrary, for example, for a `Race Car`, we introduce a new engine such as `Turbo Diesel Engine`, and we had a statement in the KB saying that the class `Turbo Engine` is disjoint from `Gas Engine`, then the system will assume that an inherited `Gas-driven` engine could not be the same as a `Turbo Engine`. By supporting such reasoning with an underspecified knowledge base, we retain the modularity of axiom definitions. The users can write their knowledge base in a modular fashion, and be confident that any obvious missing details will be filled in by the reasoning mechanism at the time of answering questions. The KM system implements heuristic unification to do such default reasoning with an underspecified knowledge base.

The focus of this paper is to define a default reasoning mechanism called unification mapping or UMAP. UMAP is motivated by the heuristic unification in KM. UMAP has a declarative specification in *Answer Set Programming (ASP)*) (Marek and Truszczyński 1999; Niemelä 1999), a declarative problem solving approach using logic programming under answer set semantics (Gelfond and Lifschitz 1990). The use of ASP also enables reasoning with disjunction and negation for which KM is not well suited. UMAP has the same behavior as heuristic unification in KM for a number of practical examples. Thus, UMAP puts an empirically useful and well-tested behavior in a rigorous formal framework.

We will start with a review of logic programming with answer set semantics. We then define a simple description language suitable for the encoding of *underspecified knowledge bases* (KB) in logic programs. The KB will contain specifications about classes, individuals, and relationships between individuals. It might not contain all explicit specializations or lack constraints. Then we discuss the four principles for the unification mapping between terms in reasoning with an underspecified knowledge base. First is the well-known specificity principle in default reasoning and the second one is specific to this application, called the *specialization principle*, which dictates that the specificity principle should be applied in a controlled manner. The third principle aims at removing redundant specification, and the fourth principle ensures that unification between terms is consistently applied across all specifications. We then present an ASP program that implements these principles. Finally, we relate our formalization to that of the KM system and discuss some potential uses of the newly developed ASP program.

Logic Programming and Answer Sets

A logic program Π is a set of rules of the form

$$c_1 \mid \ldots \mid c_k \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \quad (6)$$

where $0 \leq m \leq n$, $0 \leq k$, each a_i or c_j is a literal of a propositional language[2] and not represents *negation-as-failure*. When $n = 0$, the rule is called a *fact*. When $k = 0$, the rule is called a *constraint*. A negation as failure literal (or naf-literal) is of the form $not\ a$ where a is a literal.

For a rule r of the form (6), $head(r)$ denotes the set $\{c_1, \ldots, c_k\}$; $pos(r)$ and $neg(r)$ denote $\{a_1, \ldots, a_m\}$ and $\{a_{m+1}, \ldots, a_n\}$, respectively. For a program P, $lit(P)$ denotes the set of literals occurring in P.

Consider a set of ground literals X. X is consistent if there exists no atom a such that both a and $\neg a$ belong to X. The body of a rule r of the form (6) is *satisfied* by X if $neg(r) \cap X = \emptyset$ and $pos(r) \subseteq X$. A rule of the form (6) with nonempty head is satisfied by X if either its body is not satisfied by X or $head(r) \cap X \neq \emptyset$. A constraint is *satisfied* by X if its body is not satisfied by X.

For a consistent set of ground literals S and a program Π, the *reduct* of Π w.r.t. S, denoted by Π^S, is the program obtained from the set of all rules of Π by deleting (**i**) each rule that has a naf-literal *not a* in its body with $a \in S$, and (**ii**) all naf-literals in the bodies of the remaining rules.

S is an *answer set (or a stable model)* of Π (Gelfond and Lifschitz 1990) if it satisfies the following conditions: (**i**) If Π does not contain any naf-literal (i.e., $m = n$ in every rule of Π) then S is a minimal consistent set of literals that satisfies all the rules in Π; and (**ii**) If Π does contain some naf-literal ($m < n$ in some rule of Π), then S is an answer set of Π if S is the answer set of Π^S.

Several new extensions have been introduced to enhance the modeling capability of logic programming. In this paper, we will make use of *cardinality constraint atom* (Simons, Niemelä, and Soininen 2002) of the following form:

$$l\{b_1, \ldots, b_k\}u \quad (7)$$

[2]Rules with variables are viewed as a shorthand for the set of its ground instances.

where b_j's are literals and l and u are two integers, $l \leq u$. An atom of the form (7) is true in a set of literals X if at least l and at most u literals of the set $\{b_1, \ldots, b_k\}$ are true in X. Cardinality constraint atoms can be used anywhere a literal can be used. Using this type of atom, one can greatly reduce the number of rules of programs in answer set programming.

ASP has been successfully applied to several applications such as bioinformatics (Gebser et al. 2010); diagnosis for the space shuttle (Balduccini et al. 2001); linear temporal logic (LTL) model checking (Heljanko and Niemelä 2003); planning (Tu et al. 2011); security protocols verification (Aiello and Massacci 2001); and so on. The existence of answer set solvers, whose performance is comparable to state-of-the-art SAT solvers in many domains (e.g., CLASP (Gebser et al. 2007)), the theoretical building block results of ASP, and its recent extensions (e.g. aggregates) make ASP an appealing language for modeling of and reasoning with knowledge-intensive applications.

Underspecified Knowledge Bases

In this paper, we consider underspecified knowledge bases that are built over many sorted signatures (Manzano 1993) containing at least the three sorts *class*, *slot*, and *individual*. A sorted signature σ is a collection of constants, predicate symbols, and function symbols and is represented by a tuple $\langle \mathcal{C}, \mathcal{P}, \mathcal{F} \rangle$ where $\mathcal{C}$, $\mathcal{P}$, and $\mathcal{F}$ are pairwise disjoint and

- $\mathcal{C}$ is a collection of sorted constants, including class constants, slot constants, individual constants, and other constants;
- $\mathcal{P}$ is a set of sorted first-order predicates that contains at least the following predicates: $class$, $subclass\_of$, $instance\_of$, and $slot$ whose type[3] is $\langle class \rangle$, $\langle class \times class \rangle$, $\langle individual \times class \rangle$, and $\langle slot \times individual \times class \rangle$, respectively; and
- $\mathcal{F}$ is a set of sorted first-order functions that contains at least a set of *Skolem function* mapping individuals to classes.

We denote the set of class, slot, individual, and other constants by $\mathcal{C}_c$, $\mathcal{C}_s$, $\mathcal{C}_i$, and $\mathcal{C}_o$; the set of Skolem functions by $\mathcal{F}_s$. We define the following types of axioms.

- A *class axiom* is an atom of the form
  $$class(c) \quad (8)$$
  for $c \in \mathcal{C}_c$. This axiom states that the constant c denotes a class. For example, if $car \in \mathcal{C}_c$ then $class(car)$ is a class atom stating that *car* encodes the class of cars.
- A *constant axiom* is an axiom of the form
  $$constant(o) \quad (9)$$
  for $o \in \mathcal{C}_i \cup \mathcal{C}_o$. This axiom indicates that o is a constant.
- A *subclass axiom* is an atom of the form[4]
  $$subclass\_of(c, c') \quad (10)$$

[3]A predicate with the type $\langle s_1 \times \ldots \times s_n \rangle$ has arity n and is a relation over tuples $(x_1, \ldots, x_n)$ where x_i is a variable of sort s_i.

[4]From the knowledge representation perspective, allowing $c = c'$ in the specification results in redundancy. A simple rule $subclass\_of(X, X) \leftarrow class(X)$ can be used to enforce reflexivity of the $subclass\_of$ relation.

for $c, c' \in \mathcal{C}_c$, $c \neq c'$. This axiom states that c is a subclass of c'. E.g., the atom $subclass\_of(suburban, car)$ states that the class of Suburbans, denoted by *suburban*, is a subclass of the class of cars, denoted by *car*.

- An *instance axiom* is a rule of the form
$$instance\_of(i, c) \tag{11}$$
for some $i \in \mathcal{C}_i$, and $c \in \mathcal{C}_c$. This axiom states that the constant i is an individual of the class c. For example, to state that s_1 is a `suburban`, we write
$$instance\_of(s_1, suburban).$$
- A *descriptive axiom* is a rule of the form
$$2\left\{\begin{array}{l} slot(s, X, f_1(X)), \\ instance\_of(f_1(X), c_1) \end{array}\right\}2 \leftarrow instance\_of(X, c) \tag{12}$$
or
$$3\left\{\begin{array}{l} slot(s, f_1(X), f_2(X)), \\ instance\_of(f_1(X), c_1), \\ instance\_of(f_2(X), c_2) \end{array}\right\}3 \leftarrow instance\_of(X, c) \tag{13}$$
where f_1, f_2 are Skolem functions in $\mathcal{F}_s$, $c, c_1, c_2 \in \mathcal{C}_c$ and $s \in \mathcal{C}_s$, and X is a variable and $c \notin \{c_1, c_2\}$.
A descriptive axiom of the form (12) or (13) describes a property of individuals belonging to class c, encoded by the $slot$-atom $slot(s, X, f_1(X))$ or $slot(s, f_1(X), f_2(X))$ respectively; (12) states for each individual X in c, X is related to $f_1(X)$—an individual in class c_1—via the slot s; (13) states that for each individual X in c, $f_1(X)$—an individual in class c_1— is related to $f_2(X)$—an individual in the class c_2—via the slot s. We observe that descriptive axioms help us capture rules of form (1), and more generally, the knowledge that one would find in KM prototypes.
- A *value axiom* is a rule of the form
$$slot(s, X, v) \leftarrow instance\_of(X, c) \tag{14}$$
or
$$2\left\{\begin{array}{l} slot(s, f(X), v), \\ instance\_of(f(X), c') \end{array}\right\}2 \leftarrow instance\_of(X, c) \tag{15}$$
where $c, c' \in \mathcal{C}_c$, $c \neq c'$, $f \in \mathcal{F}_s$, and $v \in \mathcal{C}_o$. Intuitively, (14)-(15) allow for the specification of specific value of a slot.

In the following we will assume that for each Skolem function $f \in \mathcal{F}_s$ there exists at most one $c \in \mathcal{C}_c$ such that $instance\_of(f(X), c)$ appears in the axioms (12)–(15). This assumption is reasonable since the Skolemization of a formula usually creates different Skolem functions for different variables. This assumption also guarantees that for a finite signature, the ground instantiation of all axioms (8)-(15) yields a finite set of rules, a precondition for the use of answer set solvers.

We define the notion of a domain description as follows.

Definition 1 *A* domain description $D(\sigma)$ *over a signature* σ *is a logic program* $D(\sigma) = D_b \cup D_e$ *where* $D_b \cap D_e = \emptyset$ *and*

- D_b *is a set of axioms of the form (8)-(15); and*
- D_e *is set of ASP rules defined over the signature* σ.

We call D_b *and* D_e *the* basic *and* extended *part of D. We say that D is* basic *if* $D_e = \emptyset$.

For simplicity, we delay the consideration of constraints in domain descriptions to later part of the paper.

The next example shows that domain descriptions are sufficiently expressive for the representation of knowledge encoded by KM's axioms.

Example 1 Let us consider an extended version of the car domain discussed in the introduction. The signature σ_0 consists of $\mathcal{C}_c = \{car, engine, tank, suburban\}$, $\mathcal{C}_i = \{s_1\}$, $\mathcal{C}_s = \{has\_engine, has\_tank, connected\}$, $\mathcal{F}_s = \{\_Eng, \_Tank\}$. $D(\sigma_0)$ is given by the following axioms:
$$class(X) \quad \text{for } X \in \mathcal{C}_c$$
$$instance\_of(s_1, suburban)$$
$$subclass\_of(suburban, car)$$
and the following descriptive axioms (the third axiom is simplified by omitting the two instance axioms, which have already been included in the first two):
$$\begin{array}{l} 2\left\{\begin{array}{l} slot(has\_engine, X, \_Eng(X)), \\ instance\_of(\_Eng(X), engine) \end{array}\right\}2 \leftarrow instance\_of(X, car) \\ 2\left\{\begin{array}{l} slot(has\_tank, X, \_Tank(X)), \\ instance\_of(\_Tank(X), tank) \end{array}\right\}2 \leftarrow instance\_of(X, car) \\ slot(connected, \_Eng(X), \_Tank(X)) \leftarrow instance\_of(X, car) \end{array} \tag{16}$$
□

Observe that a domain description $D(\sigma)$ is a logic program that could be used to reason about properties of individuals encoded in $D(\sigma)$ under the answer set semantics represented by the $slot$-atoms. This reasoning will need to take into consideration the inheritance information encoded in $D(\sigma)$ in axioms of the form (8), (10), and (11). This can be achieved by the set Π_I, which allows for reasoning about class membership and subclass relationship in inheritance reasoning encoded in the following rules:
$$subclass\_of(C_1, C_3) \leftarrow subclass\_of(C_1, C_2), subclass\_of(C_2, C_3). \tag{17}$$
$$instance\_of(X, C_1) \leftarrow instance\_of(X, C_2), subclass\_of(C_2, C_1). \tag{18}$$
To use domain descriptions in reasoning about its individuals, we need to take Π_I into consideration. We define the notion of an underspecified knowledge base as follows.

Definition 2 *A* knowledge base *defined over a signature* σ *is the program* $D(\sigma) \cup \Pi_I$ *where* $D(\sigma)$ *is a domain description over* σ.

We observe that a knowledge base can be underspecified in the sense discussed in the introduction (see, e.g, the knowledge base in Ex. 3). A knowledge base, under the answer set semantics, will entail various $slot$-atoms about its individuals. A feasible but naive approach is to use these atoms to characterize the properties of an individual as shown in the next example.

Example 2 Let $KB_0 = D(\sigma_0) \cup \Pi_I$. It is easy to see that KB_0 has a unique answer set containing the following $slot$-atoms: $slot(connected, \_Eng(s_1), \_Tank(s_1))$ $slot(has\_engine, s_1, \_Eng(s_1))$, and $slot(has\_tank, s_1, \_Tank(s_1))$.

These atoms belong to the answer set because of the axioms in the group (16) and the fact that s_1 is an instance of *suburban* and thus is also a member of *car* (due to Π_I). □

The result in Ex. 2 suggests that $slot$-atoms can be used for reasoning about individuals in an underspecified knowledge base. The next example shows that the approach is too weak to deal with inheritance.

Example 3 Let σ_1 be the signature σ_0 extended with the class constants $\{lEngine, lTank\}$ and the Skolem functions $\{\_lEng, \_lTank\}$ (the prefix 'l' stands for 'large'). Let $D_1(\sigma_1)$ be $D_0(\sigma_0)$ extended with the following axioms:

$$class(lEngine). \quad subclass\_of(lEngine, engine).$$
$$class(lTank). \quad subclass\_of(lTank, tank).$$

and the descriptive axioms

$$2\left\{\begin{array}{l} instance\_of(\_lEng(X), lEngine), \\ slot(has\_engine, X, \_lEng(X)) \end{array}\right\}2 \leftarrow instance\_of(X, suburban). \quad (19)$$

$$2\left\{\begin{array}{l} instance\_of(\_lTank(X), lTank), \\ slot(has\_tank, X, \_lTank(X)) \end{array}\right\}2 \leftarrow instance\_of(X, suburban). \quad (20)$$

Let $KB_1 = D_1(\sigma_1) \cup \Pi_I$. Again, we can show that it has a unique answer set containing the $slot$-atoms in Fig. 1.

$$slot(has\_engine, s_1, \_Eng(s_1))$$
$$slot(has\_tank, s_1, \_Tank(s_1))$$
$$slot(connected, \_Eng(s_1), \_Tank(s_1))$$
$$slot(has\_engine, s_1, \_lEng(s_1))$$
$$slot(has\_tank, s_1, \_lTank(s_1))$$

Figure 1: Slot atoms entailed by KB_1

The presence of $slot(has\_engine, s_1, \_lEng(s_1))$ and $slot(has\_tank, s_1, \_lTank(sa))$ is due to the newly introduced descriptive axioms. □

Consider the first descriptive axioms in (16) and (19)-(20). Both describe the slot has_engine of individuals in the class $suburban$; the former via inheritance and the latter via a direct specification. This is the reason for the presence of different $slot$-atoms (e.g., $slot(has\_engine, s_1, \_Eng(s_1))$ and $slot(has\_engine, s_1, \_lEng(s_1))$)—indicating that s_1 has two engines denoted by $\_Eng(s_1)$ and $\_lEng(s_1)$—in the answer set of KB_1. Because $lEngine$ is a subclass of $Engine$, we have that $\_lEng(s_1)$ is *more specific* than $\_Eng(s_1)$. Following the specificity principle, which states that more specific information overrides less specific one, it is intuitive to conclude that s_1 has only one engine $\_LEng(s_1)$, i.e., to unify $\_LEng(s_1)$ with $\_Eng(s_1)$. Similarly, we have that $\_LTank(s_1)$ should be unified with $\_Tank(s_1)$. Our goal in the next section is to develop a declarative formalization of this type of unification.

UMAP-Atoms

We define a set of logic programming rules defining a special type of atom, called *umap*-atom, to characterize how inherited values in an underspecified knowledge bases may be combined with locally asserted values. This set of rules will define a predicate, called $umap$, whose type is the same as that of the predicate $slot$. Each ground atom $umap(s, x, y)$ encodes a relation, named s, between the individuals x and y, i.e., $umap/3$ has the same meaning as that of $slot/3$ in a descriptive axioms (12)-(13). The key difference between $umap$-atoms and $slot$-atoms lies in that one $umap$-atom might encode a relation represented by several $slot$-atoms. Our goal is to identify a set of $umap$-atoms that, given a knowledge base, (*i*) represents the expected result from the users' point of view; and (*ii*) fully characterizes individuals in the domains. For example, given KB_1 (Ex. 3), the $slot$-atoms in Fig. 1 should be represented by three $umap$-atoms:

$$umap(connected, \_lEng(s_1), \_lTank(s_1))$$
$$umap(has\_engine, s_1, \_lEng(s_1)), \text{ and}$$
$$umap(has\_tank, s_1, \_lTank(s_1)).$$

We begin with a discussion about the principles that will be used for UMAP.

Principles for Unification Mapping

As we have mentioned, a $umap$-atom unifies a set of $slot$-atoms, e.g., $umap(has\_engine, s_1, \_lEng(s_1))$ represents the set consisting of $slot(has\_engine, s_1, \_lEng(s_1))$ and $slot(has\_engine, s_1, \_Eng(s_1))$. Therefore, the key questions in defining the $umap$-atoms are

1. How can such a set of $slot$-atoms be identified and when should the atoms be unified?
2. When two or more slot atoms are unified in one rule, how should that decision be reflected in other rules that use the same atoms?

In the following, we will answer these questions by developing principles that should be applied in the unification process. First, let us define some additional notions. Let D be a domain description. In the following, by an atom, we mean a grounded atom occurring in the knowledge base $D \cup \Pi_I$.

Definition 3 *Let $D(\sigma) = D_b \cup D_e$ be a domain description over a signature σ. A* slot-atom *of D is a ground atom of the form $slot(S, X, Y)$ that occurs in D_b.*

Given a slot-atom $slot(S, X, Y)$, X and Y are called slot-term *(or* term*) of D.*

It is easy to see that by the definition of domain descriptions, a term is of the form X (or $f(X)$) for an instance X (or a Skolem function f) of some class c; furthermore, if $f(X)$ is a term then D will contain a rule stating that it is a member of some class. Due to the subclass relationship, a term can be a member of several classes. A class c is a *most specific class* of a term X if c is a minimal element (with respect to the ordering defined by the $subclass\_of$ relation) among classes having X as an instance. Two terms $f(X)$ and $g(X)$ are *compatible* if it can be proved that they belong to the same class. In Ex. 3, for any instance X of the class $suburban$, we have that $\_lEng(X)$ and $\_Eng(X)$ are compatible terms since they are an instance of the class $engine$; the most specific class of $\_lEng(X)$ and $\_Eng(X)$ is $lEngine$ and $engine$, respectively. A term X is *more specific* than a term Y if X and Y are compatible and the most specific class of X is more specific than the most specific class of Y.

The first principle that we would like to enforce in the definition of the $umap$-atoms is the well-known specificity principle in inheritance reasoning. In the context of this paper, it is as follows.

(**P1**) *Specificity principle*: in selecting terms for the construction of $umap$-atoms, more specific terms should be preferred over less specific ones.

Applying this principle to the construction of the $umap$-atoms implies that if $umap(N, X, Y)$ is entailed by the knowledge base then

- It should be obtained from an atom $slot(N, X_1, Y_1)$ by substituting X_1 and Y_1 with compatible terms X and Y, respectively; and
- X and Y are more specific than or at least as specific as, according to the inheritance principle, X_1 and Y_1, respectively.

According to (**P1**), $slot(has\_engine, s_1, \_lEng(s_1))$ and $slot(has\_engine, s_1, \_Eng(s_1))$ should be unified into a single $umap$-atom $umap(has\_engine, s_1, \_lEng(s_1))$ since $\_lEng(s_1)$ is compatible to and more specific than $\_Eng(s_1)$. This is indeed what we expect given $D_1(\sigma_1)$. However, the use of the more specific principle is in some situations too strong, as illustrated in the next example.

Example 4 Let σ_2 be the signature σ_1 (Ex. 3) extended with a Skolem function $\_eEng$ ('eEng' stands for 'extra engine') and $D_2(\sigma_2)$ be $D_1(\sigma_1)$ extended with the axiom

$$2\left\{\begin{array}{l} slot(has\_engine, X, \_eEng(X)), \\ instance\_of(\_eEng(X), engine) \end{array}\right\}2 \quad \leftarrow instance\_of(X, car) \tag{21}$$

Intuitively, (21) and (16) represent two different constraints on the slot has_engine of individuals in car, which state that each car has two engines.

Consider an instance s_0 of the class car. We have that $\_eEng(s_0)$ and $\_Eng(s_0)$ are compatible but neither is preferred over the other. As such, the two atoms $slot(has\_engine, s_0, \_Eng(s_0))$ and $slot(has\_engine, s_0, \_eEng(s_0))$ will yield two $umap$-atoms $umap(has\_engine, s_0, \_Eng(s_0))$ and $umap(has\_engine, s_0, \_eEng(s_0))$ which indicates that s_0—being a member of the class car—has two engines, the (standard) one $\_Eng(s_0)$ and the extra one $\_eEng(s_0)$.

Now, consider again the instance s_1 of $suburban$. Observe that the term $\_lEng(s_1)$ is compatible to, and more specific than, both terms $\_Eng(s_1)$ and $\_eEng(s_1)$. Thus, applying the principle (**P1**) results in a single $umap$-atom $umap(has\_engine, s_1, \_lEng(s_1))$, i.e., s_1 has a single engine. This is counter intuitive since s_1, being an instance of car, should have two engines. □

Example 4 shows that the use of the most preferred term could lead to the loss of constraints on an individual with set valued slots. We therefore introduce the next principle in the definition of $umap$-axioms.

(**P2**) *Specialization Principle*: Given a slot s and a class c, the application of the specificity principle should be limited to at most one possible value of s at c. Furthermore, if the application of (**P1**) does not violate (**P2**) then (**P1**) should be applied.

Observe that while (**P2**) limits the application of (**P1**), it also guarantees that (**P1**) is applied on as many terms that have a more specific term as possible. This can be seen in the next example.

Example 5 Returning to Ex. 4, (**P2**) should yield two possible outcomes:

- $\_lEng(s_1)$ overrides $\_Eng(s_1)$ which gives $umap(has\_engine, s_1, \_lEng(s_1))$ and $umap(has\_engine, s_1, \_eEng(s_1))$; and
- $\_lEng(s_1)$ overrides $\_eEng(s_1)$ which yields $umap(has\_engine, s_1, \_lEng(s_1))$ and $umap(has\_engine, s_1, \_Eng(s_1))$. □

Our next principle deals with the redundancy of specifications, which often occurs in a large knowledge base. A slot can have multiple specifications and some might be redundant. Consider $D_1(\sigma_1)$ with the additional descriptive axiom:

$$2\left\{\begin{array}{l} slot(has\_engine, X, \_Eng1(X)), \\ instance\_of(\_Eng1(X), engine) \end{array}\right\}2 \quad \leftarrow instance\_of(X, suburban) \tag{22}$$

This axiom says that every suburban has an engine. Intuitively, this axiom is redundant because the conclusion that a suburban has an engine follows from the fact that it is a car and every car has an engine. An alternative view is that (22) is a more specific specification of the slot has_engine in (16) and thus should override the specification in (16). We call this the *redundancy principle* as follows.[5]

(**P3**) *Redundancy Principle*: In the presence of multiple specifications of a slot for an individual, the most-specific slot specification overrides less-specific ones.

(**P1**)-(**P3**) are used to decide whether a term x should be unified with a term y of the same slot. They do not relate different slots of the same classes. For example, $\_Eng(s_1)$ occurs in both $slot(has\_engine, s_1, \_Eng(s_1))$ and $slot(connected, \_Eng(s_1), \_Tank(s_1))$. Clearly, we should require that if $\_Eng(s_1)$ is unified with a term t then t should be used in both $umap$-atoms derived from these two $slot$-atoms. This is stated in the next principle.

(**P4**) *Consistency Principle*: If a unification between x and y takes place at class c then it should be applied in every slot of class c.

Computing UMAP-Atoms using ASP

We will now specify a program, denoted by Π_R, for defining the $umap$-axioms. Π_R enforces the principles (**P1**)–(**P3**). It consists of rules for reasoning about specificity and defining the predicate $umap$. We next describe the rules of Π_R, dividing them into the set of domain-dependent rules Π^d_R and the set of domain-independent rules Π^i_R. Assume that $KB(\sigma)$ is a knowledge base over σ.

- *Domain-dependent rules*: Rules in this group declare the compatibility between terms constructed from Skolem functions over individuals. For each pair of $f_1, f_2 \in \mathcal{F}_s$ appearing in an axiom of the form (12) or (13), Π^d_R contains the rule

$$\begin{array}{l} compatible(f_1(X), f_2(X)) \\ \quad \leftarrow constant(X), instance\_of(f_1(X), D), \\ \quad\ \ class(D), instance\_of(f_2(X), D). \end{array} \tag{23}$$

[5]We observe that (**P3**) is needed only when the KB contains redundant axioms. For KB without redundant axioms, this principle (and the rules for enforcing it) will not be needed.

Furthermore, for each descriptive axiom of the form (12) Π^d_R contains the rules

$$range(f_i(X), c_i) \leftarrow constant(X) \quad (24)$$

$$dom(f_i(X), c) \leftarrow constant(X) \quad (25)$$

Atoms of the form $range(Z, c)$ ($dom(Z, c)$) encode the range (domain) of the Skolem function occurring in Z.

- *Domain-independent rules*: The set of independent rules Π^i_R defines various predicates related to the compatibility between constants and terms of the form $f(X)$, $f \in \mathcal{F}_s$, and the preference between compatible terms under the specificity principle.
  - *Terms, compatibility between terms, and type of slots*: This group includes the following rules.

$$2\{term(X), term(Y)\}2 \leftarrow slot(S, X, Y). \quad (26)$$

$$tv(Y, S, C) \leftarrow slot(S, X, Y), range(Y, C). \quad (27)$$

$$compatible(X, X) \leftarrow constant(X). \quad (28)$$

Rule (26) defines terms that will be used in defining $umap$-atoms. Rule (27) associates a value (a term) with a slot at a class for use in UMAP since UMAP is restricted to values for the same slot at the same class. For example, terms referring to instances of the class $engine$ and used in the slot has_engine can be unified with each other; they should not be unified with terms referring to the class $tank$. Rule (28), with (23), defines the compatibility between two terms X and Y.

  - *Specificity between terms*: Rules in this group define a more specific relation between terms and the most specific term of a given term.

$$inst(X, C) \leftarrow instance\_of(X, C_1), \quad (29)$$
$$C_1 \neq C, subclass\_of(C_1, C).$$

$$most\_cls(X, C) \leftarrow instance\_of(X, C), \quad (30)$$
$$not\ inst(X, C).$$

$$more\_sp(X, Y) \leftarrow compatible(X, Y), \quad (31)$$
$$subclass\_of(C_1, C_2), C_1 \neq C_2,$$
$$most\_cls(X, C_1), most\_cls(Y, C_2).$$

$$has\_ms(Y) \leftarrow compatible(X, Y), \quad (32)$$
$$more\_sp(X, Y),$$

$$mspec(X, Y, S, C) \leftarrow more\_sp(X, Y), \quad (33)$$
$$tv(X, S, C), tv(Y, S, C),$$
$$compatible(X, Y), not\ has\_ms(X).$$

$$mspec(X, Y, S, C) \leftarrow more\_sp(X, Y), \quad (34)$$
$$compatible(X, Y), subclass\_of(C, C_1),$$
$$tv(X, S, C), tv(Y, S, C_1),$$
$$not\ has\_ms(X).$$

Rule (29) says that X is an instance of a class C by inheritance ($inst(X, C)$) if it is an instance of a class C_1 that is a proper subclass of C. Rule (30) identifies the most specific class of a term X. Rule (31) defines the more specific relation between compatible terms: X is more specific than Y if X and Y are compatible terms and the most specific class of X is a subclass of the most specific class of Y. Rule (32) identifies a term that has a compatible and more specific term. $mspec(X, Y, S, C)$, defined in (33)-(34), encodes that X is a most specific term of Y and can be used in defining $umap$-atoms for the slot S at the class C. The rules state that X is a most specific term of Y if they are compatible, X is more specific than Y, and there exists no term that is more specific than X.

  - *Computing Redundancy*: Rules in this group identify redundant specifications and enforce the principle (**P3**).

$$2\{over(Y_1, Y), redundant(S, X, Y)\}2 \leftarrow \quad (35)$$
$$class(C), slot(S, X, Y), slot(S, X, Y_1),$$
$$range(Y, C), range(Y_1, C), dom(Y, D),$$
$$dom(Y_1, D_1), subclass\_of(D_1, D).$$

$$over(Y_2, Y) \leftarrow over(Y_2, Y_1), over(Y_1, Y). \quad (36)$$

$$selected\_to\_replace(Y) \leftarrow over(Y_1, Y). \quad (37)$$

(35) declares $redundant(S, X, Y)$ which encodes that $slot(S, X, Y)$ is redundant if there exists another specification $slot(S, X, Y_1)$ for the same class at a more specific class. In this case, Y should be unified with Y_1, denoted by $over(Y_1, Y)$. (36) says that $over$ is transitive and (37) says that Y will be replaced by some term if it is unified with some other term.

  - *Unification rules*: Rules in this group create a mapping between terms for the unification mapping according to the principles (**P1**), (**P2**), and (**P4**)

$$0\left\{\begin{array}{c} pick(X, Y, S, C) : \\ mspec(X, Y, S, C) \end{array}\right\}1 \leftarrow term(Y). \quad (38)$$

Intuitively, $pick(X, Y, S, C)$ states that the most specific term X of Y, w.r.t. the slot S at the class C, will be unified with Y. Rule (38) creates a mapping from the set of terms into the set of their most specific terms. To ensure that the two principles are enforced, we need the following rules.

$$override(Y, S, C) \leftarrow pick(X, Y, S, C). \quad (39)$$

$$selected\_ovd(Y) \leftarrow pick(X, Y, S, C). \quad (40)$$

$$used\_ovd(X, S, C) \leftarrow pick(X, Y, S, C). \quad (41)$$

These rules keep track of the terms that have been selected for unification mapping by projecting different elements of $pick(X, Y, S, C)$. (39) and (40) define $override(Y, S, C)$ and $selected\_ovd(X)$, respectively, indicating that the value Y of slot S at class C has been selected to be unified with some other term. These atoms are used in the enforcement of (**P1**) and (**P2**) by the following constraints.

$$\leftarrow tv(X, S, C), pick(Z, X, S, C), X \neq Y, \quad (42)$$
$$term(Z), tv(Y, S, C), pick(Z, Y, S, C).$$

$$\leftarrow mspec(X, Y, S, C), \quad (43)$$
$$not\ override(Y, S, C),$$
$$not\ used\_ovd(X, S, C).$$

(42) guarantees that one term is not unified with two different and less specific terms. (43) ensures that the specificity principle is applied whenever it is possible. The combination of these two rules enforces (**P2**).

The next rules finalize the mapping for the unification process.

$$unify(X,Y) \leftarrow pick(X,Y,S,C). \quad (44)$$

$$unify(X,Y) \leftarrow over(X,Y), \quad (45)$$
$$not\ selected\_to\_replace(X).$$

$$unify(X,X) \leftarrow not\ selected\_ovd(X), \quad (46)$$
$$term(X), not\ selected\_to\_replace(X).$$

Rules (44)-(46) enforce (**P4**) by defining the predicate $unify(X,Y)$. If $unify(X,Y)$ holds, then the term X should replace the term Y. (44) states that if X has been selected to replace Y via $pick(X,Y,S,C)$, then Y should be identified with X. (45) states that if X has been selected to be replaced by any other term Y, then Y should be identified with X. (46) indicates that if X has not been selected to be replaced by any other term then, X is identified by itself.

– *Creating umap-atoms*: Rule (47) defines the predicate $umap(S,X,V)$. It states that $umap(S,X,V)$ should be obtained from some $slot$-atom $slot(S,X_1,V_1)$ by applying the specificity and specialization principles on the terms X_1 and V_1 simultaneously.

$$umap(S,X,V) \leftarrow slot(S,X_1,V_1),$$
$$not\ redundant(S,X,V), \quad (47)$$
$$unify(X,X_1), unify(V,V_1).$$

Given an underspecified knowledge base $KB = D(\sigma) \cup \Pi_I$, by KB^r we denote the program $KB \cup \Pi_R$. Let S be a set of ground atoms in KB^r and i be an individual constant in σ, we define

$$D(i,S) = \left\{ umap(s,x,v) \middle| \begin{array}{l} umap(s,x,v) \in S, \\ x = i \vee \exists f \in \mathcal{F}_s.[x = f(i)] \\ v = i \vee \exists g \in \mathcal{F}_s.[v = f(i)] \end{array} \right\}$$

Definition 4 *Let KB be an underspecified knowledge base over σ, i be an individual of the class c in KB, and S be an answer set of KB^r. The set $D(i,S)$ is called the* description *of i w.r.t. S.*

In the next examples, we illustrate the use of KB^r in the different knowledge bases.

Example 6 Let us consider the KBs from Examples 2-3. We can check that KB_0^r has an answer set M that contains the following $umap$-atoms: $umap(has\_engine, s_1, \_Eng(s_1))$, $umap(has\_tank, s_1, \_Tank(s_1))$, $umap(connected, \_Eng(s_1), \_Tank(s_1))$.

Each of the $umap$-atoms corresponds to one $slot$-atom (Ex. 2) because none of the terms in the set $\{s_1, \_Eng(s_1), \_Tank(s_1)\}$ has a more specific term. Hence, there exists no atom of the form $mspec(V_1, V, S, C)$ in M. This implies that no atom of the form $selected\_ovd(V)$, $selected\_to\_replace(V)$, or $redundant(S,X,Y)$ exists in M. Under this condition, rule (46) identifies each term by itself. Rule (47) indicates that for each $slot(N,X,V) \in M$, $umap(N,X,Y) \in M$.

Now consider KB_1 from Ex. 3. We will show that the program KB_1^r has a unique answer set M that contains $umap(has\_engine, s_1, \_lEng(s_1))$, $umap(has\_tank, s_1, \_lTank(s_1))$, and $umap(connected, \_lEng(s_1), \_lTank(s_1))$.

First, we have that $instance\_of(s_1, suburban) \in M$ because it is a fact of KB_1^r. This, together with rule (18), implies that $instance\_of(s_1, car) \in M$. The latter, with the axioms in $D_1(\sigma_1)$ (Ex. 3), implies that $instance\_of(\_lEng(s_1), engine) \in M$ and $instance\_of(\_lTank(s_1), tank) \in M$. Similarly, we have that $instance\_of(\_lEng(s_1), lEngine) \in M$ and $instance\_of(\_lTank(s_1), lTank) \in M$.

It is easy to see that the atoms of the form $instance\_of(X,Y)$ in M imply the presence of the five atoms of the form $slot(S,X,V)$ in Fig. 1 in M.

Since there exists no class that is a proper subclass of $lEng$ or $lTank$, the most specific class of $\_lEng(s_1)$ and $\_lTank(s_1)$ is $lEngine$ and $lTank$, respectively (by (29)-(30)); this implies that the most specific term of $\_lEng(s_1)$ (resp. $\_lTank(s_1)$) is itself (by (33)-(34)). On the other hand, $\_Eng(s_1)$ (resp. $\_Tank(s_1)$) has a more specific term, $\_lEng(s_1)$ (resp. $\_lTank(s_1)$), as both are instances of $engine$ (resp. $tank$), but the former is more specific than the latter. This results in the following atoms are in M:
$mspec(\_lEng(s_1), \_Eng(s_1), has\_engine, engine)$ and
$mspec(\_lTank(s_1), \_Tank(s_1), has\_tank, tank)$
Since none of the terms $\_lEng(s_1)$, $\_Eng(s_1)$, $\_lTank(s_1)$, and $\_Tank(s_1)$ appears more than once in atoms of the form $mspec$, rules (38)-(43) imply that $pick(\_lEng(s_1), \_Eng(s_1), has\_engine, engine)$ $pick(\_lTank(s_1), \_Tank(s_1), has\_tank, tank)$ belong to M. So, $unify(\_lEng(s_1), \_Eng(s_1))$ belongs to M because of (42). Similarly, we can show that $unify(\_lTank(s_1), \_Tank(s_1))$ belongs to M. This implies that the application of rule (47) yields only three $umap$-atoms in M as described earlier. □

We observe that the principle (**P2**) might create different specializations of a slot and thus different sets of $umap$-atoms for an individual. It is easy to see that for $D_2(\sigma_2)$ (Ex. 4) and $KB_2 = D_2(\sigma_2) \cup \Pi_I$, KB_2^r has two answer sets, each corresponding to one outcome discussed in Ex. 5.

Properties of the Implementation

We now discuss some properties of the program developed in the previous subsection. In general, we say that a program P satisfies a principle (**Pi**) ($i = 1, \ldots, 4$) if none of the answer sets of P violates (**Pi**).

Theorem 1 *For a basic domain description $D(\sigma)$ over the signature σ, the program $KB^r = D(\sigma) \cup \Pi_I \cup \Pi_R$ satisfies the principles (***P1***)-(***P4***).*

Proof. (Sketch) Let M be an answer set of KB^r. M satisfies (**P1**) because of rules (44) and (45). It satisfies (**P2**) because of rules (42) and (43). Rules (35)-(37)) guarantee that M satisfies (**P3**). (**P4**) is satisfied because of the rule (47). □

To discuss the next property, we need an additional definition.

Definition 5 *Given a domain description $D(\sigma)$, a slot s is* deterministic at a class c *in $D(\sigma)$ if*

- *there exists at most one Skolem function f_c such that $slot(s,X,f_c(X))$ appears in an axiom of the form (12) in $D(\sigma)$ whose right side is $instance\_of(X,c)$; and*
- *there exists at most one pair of Skolem functions f_1, f_2 such that $slot(s, f_1(X), f_2(X))$ appears in $D(\sigma)$ whose right side is $instance\_of(X,c)$.*

A slot s is deterministic *if it is deterministic at every class in* $D(\sigma)$. *Otherwise, s is* nondeterministic.
$D(\sigma)$ *is* deterministic *if every slot in* $D(\sigma)$ *is deterministic.*

Observe that the fact that a slot s is deterministic does not imply that the slot is associated with a single value as commonly called *single-valued* slot in the literature. For example, the slot has_engine of $D_1(\sigma_1)$ (Ex. 3) is indeed a deterministic slot since for each class appearing in $D_1(\sigma_1)$, we can check that the above conditions are satisfied.

Theorem 2 *For a basic and deterministic domain description $D(\sigma)$, the program $KB^r = D(\sigma) \cup \Pi_I \cup \Pi_R$ has a unique answer set.*

Proof. We use the splitting sequence theorem (Lifschitz and Turner 1994) with respect to the sequence $\langle X_i \rangle_{i=1}^6$ where $X_i = \bigcup_{j=1}^i L_i$ and

- L_1 is the set of atoms of the form $class(X)$, $constant(X)$, $instance\_of(X,Y)$, $subclass\_of(X,Y)$, and $slot(X,Y,Z)$;
- L_2 is the set of atoms of the form $compatible(X,Y)$, $range(X,Y)$, $dom(X,C)$, $term(X)$, $tv(X,S,C)$, and $inst(X,C)$;
- L_3 is the set of atoms of the form $most\_cls(X,C)$;
- L_4 is the set of atoms of the form $more\_sp(X,Y)$;
- L_5 is the set of atoms of the form $mspe(X,Y,S,C)$; and
- L_6 is the set of atoms of the form $replace(X,Y)$, $selected\_to\_replace(X)$, or $redundant(S,X,Y)$.

It can be shown that the bottom of KB^r relative to X_i has a unique answer set S_0 and the partial evaluation of KB^r with respect this answer set has a unique answer set. Detail can be found in (Chaudhri and Son 2011). □

We note that Theorem 1 indicates that KB^r provides a declarative characterization of $umap$-atoms. Since the class of basic domain descriptions covers the most frequently occurred axioms in the biology textbook KB—which was an inspiration for our investigation—we can conclude that the proposed methodology is a viable alternative to query answering in systems that employs the knowledge base developed in AURA and SHAKEN. Theorem 2 shows that for basic and deterministic domain descriptions, the description for each individual in the domain is unique and can be computed bottom up. This, along with the recent advances in ASP solvers, allows us to expect that answering queries using KB^r can be done efficiently.

Related Work and Discussions

We first comment on our choice of using ASP for this work as opposed to a description logic formalism. Recall that the axiom (1) is the most frequently occurring axiom pattern in KM prototypes. The axiom (1) violates the tree model property. The axioms of this form can only be represented in description graphs, but that requires separating the slots that participate in such graphs from the slots in the rest of the KB (Motik et al. 2009). For this reason, we chose to first do this formalization using ASP. We do, however, believe that it is possible to express aspects of reasoning in an underspecified knowledge base in a description logic framework, and we are investigating that in our current research.

Our formalization of umap-atoms is inspired by the heuristic unification implemented in the KM system (Clark and Porter 2011). There are several differences between unification mapping as specified in this paper and the heuristic unification used in KM. First, KM's approach is procedural and is explained mostly using examples while our formalization is declarative. This also prevents an object-level comparison between KM and our formalization. Second, KM computes only one possible unification while our approach computes all possible unifications. A single default choice for unification suffices in many practical situations, but in situations where the default choice goes wrong, it is important to give the user an option of choosing amongst different alternatives. Finally, the heuristic unification in KM is destructive, i.e., when two individuals are unified, one is replaced with the other in the KB. In contrast, our approach is non-destructive. The unification decisions made by UMAP are truly non-monotonic in the sense that if additional information is added to the KB, the individuals that were unified in the initial version of the KB may no longer be unified in the new version of the KB. Due to destructive nature of unification in KM, it is not capable of such non-monotonic behavior. KM allows a user to make explicit unification assertions in the KB. We have not yet considered that aspect of heuristic unification in our formalization.

We note that even though an ability to write class definitions in a modular manner is a desirable property, it is still useful to provide a facility when a change in a class could be automatically reflected in its subclasses. The SHAKEN and AURA systems have supported a knowledge propagation mechanism for this purpose. The current paper ignores the aspect of updating an under-specified knowledge base.

Dealing with Multiple Inheritance

Multiple inheritance occurs when two values of a slot at a class are represented by Skolem functions defined in two different classes.

Example 7 Consider the following statements about cars: (*i*) A powerful car is a car that has an engine with lots of power; (*ii*) A big car is a car that has a large engine; (*iii*) Suburbans are powerful cars and are big cars; and (*iv*) s_1 is a suburban. This information can be encoded as a domain description over the signature σ_{car} containing $\mathcal{C}_i = \{s_1\}$; $\mathcal{C}_c = \{engine, big\_car, powerful\_car, suburban\}$; $\mathcal{C}_o = \{large, lots\}$; $\mathcal{C}_s = \{has\_engine, size, power\}$; and $\mathcal{C}_f = \{\_Eng1, \_Eng2, \_size, \_power\}$. For brevity, we omit the class, instance, constant, and subclass axioms of $D(\sigma_{car})$. To state that a big car has a large engine, we use[6]

$$3 \left\{ \begin{array}{l} instance\_of(\_Eng1(X), engine), \\ slot(size, \_Eng1(X), large), \\ slot(has\_engine, X, \_Eng1(X)) \end{array} \right\} 3 \qquad (48)$$
$$\leftarrow instance\_of(X, big\_car).$$

[6]We combine two descriptive axioms into one to save space.

That powerful car has a powerful engine is encoded as

$$3 \left\{ \begin{array}{l} slot(power, \_Eng2(X), lots) \\ slot(has\_engine, X, \_Eng2(X)) \\ instance\_of(\_Eng2(X), engine) \end{array} \right\} 3 \leftarrow \quad (49)$$
$$instance\_of(X, powerful\_car)$$

We can check that $D(\sigma_{car}) \cup \Pi_I \cup \Pi_R$ has a single answer set containing $umap(has\_engine, s_1, \_Eng1(s_1))$, $umap(size, \_Eng1(s_1), large)$, $umap(power, \_Eng2(s_1), lots)$, $umap(has\_engine, s_1, \_Eng2(s_1))$. □

The answer given by $KB(\sigma_{car})^r$ is only partially satisfactory in that it indicates that s_1 has an engine that is large and an engine that is powerful. It is not fully satisfactory since, unless there is a reason to believe otherwise, the two engines should be considered the same. Observe that because there is no subclass relation between big_car and $powerful\_car$, there is no preference among the slot specification for has_engine in (48) and (49). Nevertheless, it is intuitive that they should be unified. When should this unification be executed? To answer this question, we observe that a slightly different situation occurs in Ex. 4 where the two $slot$-atoms should not be unified. We observe that the specifications of the has_engine slot in Ex. 4 have the same domain while those in Ex. 7 come from unrelated domains. So, we can resolve the coreferential problem by (*i*) determining the set $S(v, s, c)$ of coreferential $slot$-atoms given each triple of a term v, a class c, and a slot s; and (*ii*) unify the set $S(v, s, c)$. The first task can be realized using the following ASP rule:

$$\begin{array}{r} coreference(X, Y, S, C) \leftarrow X \neq Y, C_1 \neq C_2, \quad (50) \\ compatible(X, Y), dom(X, C_1), dom(Y, C_2), \\ not\ has\_ms(X), not\ has\_ms(Y), dom(X, C_1), \\ tv(X, S, C), not\ subclass\_of(C_1, C_2), \\ tv(Y, S, C), not\ subclass\_of(C_2, C_1). \end{array}$$

The above rule states that two compatible terms X and Y, which are possible values of the term S at class C, are coreferential if their domains are unrelated and neither has a more specific term. To accomplish task (*ii*), we need a set of rules similar to the unification rules (38)–(46).

Dealing with Constraints

We will now develop a set of rules for dealing with cardinality constraints on the set of slot values. First, we extend our domain description language to allow the representation of cardinality constraints on a set of slot values as follows. A *constraint axiom* can be given in the following form:

$$constraint(s, c, l, u) \quad (51)$$

where $s \in \mathcal{C}_s$, $c \in \mathcal{C}_c$, and l and u are integers with $1 \leq l \leq u$. This constraint states that slot s has at least l and at most u values at class c. We will next discuss a set of rules for enforcing the constraints of the form (51). First, we need to guarantee that the set $\{umap(s, X, Y) \mid X$ is a *specified* instance of $c\}$ has at least l elements, where a specified instance of a class is a member of the class that is not an instance of any of its subclasses. This can be achieved by the following constraint:

$$\begin{array}{l} \leftarrow \{umap(s, X, Y)\}\ l - 1, instance\_of(X, c) \\ \quad not\ inst(X, c), constraint(s, c, l, u) \end{array} \quad (52)$$

Since each $umap$-atom encodes at least one $slot$-atom, the above rule guarantees that slot s has at least l values at class c. Similarly, we can add a constraint to make sure that the set $\{umap(s, X, Y) \mid X$ is a specified instance of $c\}$ has at most u elements. The following constraint achieves this:

$$\begin{array}{l} \leftarrow u + 1\ \{umap(s, X, Y)\}, instance\_of(X, c) \\ \quad not\ inst(X, c), constraint(s, c, l, u) \end{array} \quad (53)$$

We will need to add rules, similar to rules (38)–(46), to the program to make sure that, for each $constraint(s, c, l, u)$, if the set $\{slot(s, X, Y)\} \mid X$ is a specified instance of $c\}$ has more than u elements then unification needs to be executed; and if it has less than l elements then the program is inconsistent. This can be achieved using a set of rules similar to rules (38)–(46). We omit these rules for space reasons (see (Chaudhri and Son 2011)).

Conclusions and Future Work

In this paper, we considered the problem of reasoning in an under-specified knowledge base. Specifically, we considered two forms of underspecification: some of the cardinality constraints are omitted from the KB and some values are specialized across a class hierarchy but the explicit references to which values are specialized are omitted. Such underspecification is very useful in achieving modularity in a large complex KB. We presented an approach called UMAP or unification mapping to do inheritance reasoning in such an under-specified KB. UMAP is inspired by a similar reasoning mechanism called heuristic unification that is implemented in KM and has proven to be empirically useful in enabling knowledge base construction by biologists with little background in formal knowledge representation. While we have used ASP as a formal framework to present our approach, we believe that the basic ideas are general and applicable to other reasoning frameworks.

Our immediate goal in the near future is to do the performance evaluation of the proposed framework using the biology knowledge base developed as part of Project Halo (Gunning et al. 2010). Our focus will be on using the program in answering three types of questions: (*i*) What is a X? The set of UMAP-atoms represents a complete description about an individual and thus can serve as the answer for this question; (*ii*) What are the relationships between X and Y? In this type of question, we will focus on relationships that can be described by paths connecting two individuals X and Y. We expect that the efficient solvers that have been developed for ASP will help us compute relationships that cannot be computed by the current KM system. (*iii*) A third possible class of questions involves process interruption reasoning. Specifically, we plan to enhance the current formalization to include a modular action language similar to what has been done in (Inclezan and Gelfond 2011).

Acknowledgment: This work has been funded by Vulcan Inc. and SRI International. This work has greatly benefited from substantive discussions with Pete Clark, Richard Fikes, Benjamin Grosof, Michael Kifer and Michael Wessel. We are grateful to them for their input.

References

Aiello, L. C., and Massacci, F. 2001. Verifying security protocols as planning in logic programming. *ACM Trans. Comput. Log.* 2(4):542–580.

Balduccini, M.; Gelfond, M.; Watson, R.; and Nogueira, M. 2001. The USA-Advisor: a case study in answer set planning. In *Lectures Notes in Artificial Intelligence (Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'01)*, volume 2173, 439–442. Springer-Verlag.

Barker, K.; V., C.; Clark, P.; Israel, D.; Porter, B.; and Romero, P. 2003. *Halo Pilot Project*. Technical report, SRI International.

Chaudhri, V. K., and Son, T. C. 2011. *Specifying and reasoning with underspecified knowledge bases using answer set programming*. Technical report, NMSU. Technical Report.

Clark, P., and Porter, B. W. 1997. Building concept representations from reusable components. In *Proceedings of the AAAI Conference*, 369–376.

Clark, P., and Porter, B. 2011. *KM (v2.0 and later): Users Manual*.

Clark, P.; ; Thompson, J.; and Porter, B. W. 2000. Knowledge patterns. In *Proceedings of the KRR Conference*, 591–600.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. clasp: A conflict-driven answer set solver. In Baral, C.; Brewka, G.; and Schlipf, J., eds., *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*, 260–265. Springer-Verlag.

Gebser, M.; Schaub, T.; Thiele, S.; and Veber, P. 2010. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming* 10(4-6).

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation. In Warren, D., and Szeredi, P., eds., *Logic Programming: Proceedings of the Seventh International Conference*, 579–597.

Gunning, D.; Chaudhri, V. K.; Clark, P.; Barker, K.; Chaw, S.-Y.; Greaves, M.; Grosof, B.; Leung, A.; McDonald, D.; Mishra, S.; Pacheco, J.; Porter, B.; Spaulding, A.; Tecuci, D.; and Tien, J. 2010. Project Halo Update—Progress Toward Digital Aristotle. *AI Magazine* 33–58.

Heljanko, K., and Niemelä, I. 2003. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming* 3(4,5):519–550.

Inclezan, D., and Gelfond, M. 2011. Representing biological processes in modular action language $\mathcal{ALM}$. In *Logical Formalizations of Commonsense Reasoning, AAAI Spring Symposium*.

Lenat, D. 1995. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM* 38(11).

Lifschitz, V., and Turner, H. 1994. Splitting a logic program. In Van Hentenryck, P., ed., *Proceedings of the Eleventh International Conference on Logic Programming*, 23–38.

Manzano, M. 1993. Introduction to many-sorted logic. In *Many-sorted logic and its applications*. Wiley Professional Computing. 3–86.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, 375–398.

Motik, B.; Grau, B. C.; Horrocks, I.; and Sattler, U. 2009. Representing ontologies using description logics, description graphs, and rules. *Artificial Intelligence* 173:1275–1309.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3,4):241–273.

Simons, P.; Niemelä, N.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1–2):181–234.

Tu, P. H.; Son, T. C.; Gelfond, M.; and Morales, R. 2011. Approximation of action theories and its application to conformant planning. *Artificial Intelligence Journal* 175(1):79–119.