4
Creating a Knowledge Base

To reason about the world, an agent must have information about it. Because it is unreasonable to teach an agent everything there is to know, we decide on what kind of agent we are building and educate it accordingly. The collection of statements about the world we choose to give the agent is called a knowledge base. In this chapter we create several knowledge bases using the declarative approach. The emphasis is on the methodology of knowledge representation and the use of ASP. Using examples in several, very different domains, we emphasize the importance of the following:

- modeling the domain with relations that ensure a high degree of elaboration tolerance;
- the difference between knowledge representation of closed vs open domains; i.e., we need to know when we can assume that our information about a relation is complete and when we should instead reason with incomplete information, but realize that we are doing so (reasoning with incomplete information is covered much more thoroughly in Chapter 5);
- representing commonsense knowledge along with expert knowledge;
- recursive definitions and hierarchical organization of knowledge.

The first knowledge base contains various relationships within the family group, the second models electrical circuits, and the third deals with a basic taxonomic hierarchy that includes classes such as "submarine" and "vehicle." (For now we only make use of basic rules and recursion. More sophisticated knowledge bases are discussed in later chapters.)

Examples in this chapter are given using notation necessary to run programs on real systems.
To make ASP programs executable, we replace $\neg$ with `-`, $\leftarrow$ with `:-`, and *or* with `|`.
For information on these systems, please see Appendix A and B.

4.1 Reasoning about Family

In this section we consider several extensions of the simple family knowledge base (KB) from Chapter 1. The first extension familiarizes the program with such basic terms as *brother*, *sister*, and so on. The second defines a notion of orphan. The third deals with a recursive definition of the notion of ancestor. Each extension has its own assumptions about completeness of information for various relations of the domain.

4.1.1 Basic Family Relationships

To represent knowledge about family relationships we start with defining the sorts that exist in our domain. Let us use the same sorts we had in the old KB in Chapter 1 – *person* and *gender*. Their membership is represented by the following atoms:

```
1  person(john).
2  person(sam).
3  person(alice).
4
5  gender(male).
6  gender(female).
```

Next we define the relationships between the objects of our domain – binary relations *father*, *mother*, *parent*, and *child* with parameters of type *person* and relation *gender_of* with parameters *person* and *gender*.

```
7  father(john,sam).
8  mother(alice,sam).
9
10 gender_of(john,male).
11 gender_of(alice,female).
12 gender_of(sam,male).
13
14 parent(X,Y) :- father(X,Y).
15 parent(X,Y) :- mother(X,Y).
16
17 child(X,Y) :- parent(Y,X).
```

Lines of code numbered 1–17 give the complete program from Chapter 1.

In this book, lines of code that are numbered sequentially can be executed as a single program; code that is not numbered usually represents a (possibly incorrect) alternative.

We proceed by showing how this initial knowledge can be expanded. Suppose, for instance, that our family from Chapter 1 has an exciting announcement – John and Alice had another baby boy, Bill. To record this joyous event, we add new statements to our knowledge base:

```
18 person(bill).
19 father(john,bill).
20 mother(alice,bill).
21 gender_of(bill,male).
```

This may be an appropriate moment for teaching our agent a new family relation – "X is a brother of Y." Let us denote this relation by $brother(X, Y)$ where X and Y are of sort $person$.

At first glance, the following simple rule should suffice to define the new notion:

```
brother(X,Y) :- gender_of(X,male),
                father(F,X),
                father(F,Y),
                mother(M,X),
                mother(M,Y).
```

This rule says that X is a brother of Y if X is male and X and Y have the same parents. To check if the rule is properly understood, we can create a program consisting of lines 1–21 together with the above rule for $brother$ and use an implementation of STUDENT to answer queries $?brother(sam, bill)$ and $?brother(sam, X)$. Although the first query is answered correctly, the second gives a surprising answer – $brother(sam, sam)$.

The agent is not at fault – we are. Of course, every one of us knows that a boy cannot be his own brother, but STUDENT does not share this knowledge. It must be explicitly stated in the rule. To do that we will use the built-in operator `!=` that stands for $\neq$. Here is the correct definition:

```
22 brother(X,Y) :- gender_of(X,male),
23                 father(F,X),
24                 father(F,Y),
25                 mother(M,X),
26                 mother(M,Y),
27                 X != Y.
```

We can test the new program and see that now the answers are correct. We succeeded in teaching the agent a new family relation!

The problem we experienced in defining the notion of brother is symptomatic of a serious difficulty confronted by a programmer in the process of knowledge representation. *A very large part of our knowledge is so deeply engrained in us that we do not normally think about it.* Bringing this knowledge out in the open requires discipline and a well-developed power of introspection. (In this sense declarative programming is not very different from procedural programming.)

Several fascinating subareas of AI deal with discovering and codifying such hidden "commonsense" knowledge and with finding ways of its efficient and elegant representation. In the following chapters we discuss some recent advances in this field.

Even though our agent can answer a large number of questions about relationships within the family, an experienced teacher will be able to discover serious gaps in its knowledge of the domain. Let's add the name Bob to our $person$ sort and consider the questions:

$$?\ father(alice, bill)$$

$$?\ father(bill, sam)$$

$$?\ father(john, bob)$$

What are the expected answers to these queries? Well, the answer to the first two questions we expect from humans is obviously *no*. STUDENT, however, will return *unknown* to both of them. This result is again not surprising. To answer the first query correctly, the program should know that females cannot father children. This information can be easily incorporated into the program by the rule

```
28 -father(X,Y) :- gender_of(X,female).
```

The second answer is justified because we know that a person can have only one father. This is expressed by the rule

```
-father(X,Y) :- father(Z,Y),
                X != Z.
```

It is at this point that we run into an unpleasant aspect of ASP as it is currently implemented. If you run the program that incorporates the last rule with one of the currently available solvers, you will get a mysterious message about safety. The "safety" requirement comes from serious concerns about the efficiency of the solver. The general definition of this notion of safety differs from system to system. For our purpose it is sufficient to say that *a rule is* **unsafe** *if it contains an unsafe variable (i.e., one that does not occur in a literal in the body that is neither built in nor preceded by default negation).* (For a more detailed explanation of rule safety, please see your chosen ASP solver manual.)

One can easily see that the previous rule is unsafe because the only literal in the body containing variable X is formed by the built-in predicate `!=`. Hence, variable X is unsafe. We can make the rule safe by noting that, since X is a parameter of predicate $father$, it must be of the sort $person$, and then adding this information to the body of the rule:

```
-father(X,Y) :- person(X),
                father(Z,Y),
                X != Z.
```

This strategy is rather general. If you took care to define the sorts of parameters of predicates, a rule with an unsafe variable X can be turned into a safe rule by adding the sort of X to the body. The problem with the safety of rules disappears if one uses SPARC, an extension of ASP that requires an explicit definition of a program's sorts. Appendix C gives a brief introduction to this language and its corresponding software tools.

Let's return to our discussion of representing $\neg father$ and note that, strictly speaking, the given rule is not sufficient to represent the statement that a person cannot have more than one father. Our program also needs to know that Bill and John are two different people. We do not need to add anything, though, because ASP has a built-in **Unique-Name Assumption (UNA)**. This means that the objects in our program are considered distinct if their names are different unless we have specified otherwise. In other words, our agent considers "John" and "Dad" to be distinct people, even though in real life, he may answer to both. With the UNA, the first two questions are correctly answered by *no*.

Now try to use your common sense to answer the third question: "Is John the father of Bob?" Informal tests conducted with a fairly large number of students show that the responders are divided into two groups: a big one whose members respond with a definite *no*, and a smaller one with people that give a hesitant *maybe*. The difference can be explained by their varying understandings of the context of our family story. The larger group apparently makes the so-called Closed World Assumption (CWA) – they assume that the story contains *complete* information about John's family. This justifies the following simple argument: "The story does not mention that John is the father of Bob and, therefore, he is not." The members of the second group do not assume that the given information about John's family is complete (e.g., Bob can be John's son from a previous marriage); hence, the cautious answer.

It is easy to check that the reasoner associated with our program belongs to the second, more cautious and deliberate group. To see that, we simply need to expand the program by the fact

```
person(bob).
```

and ask our query. The answer will be *unknown*.

If we want to model the reasoning of the first group, we should be able to explicitly state that our information about John's fatherhood is complete. This can be done by using the default negation of ASP. The rule

```
-father(X,Y) :- person(X), person(Y),
                not father(X,Y).
```

says that if there is no reason to believe that X is the father of Y, then he is not. This is exactly the closed world assumption for fathers mentioned earlier. It is easy to see that our new program containing this rule will answer *no* to the third query.

Note that statements $person(X)$ and $person(Y)$ cannot be removed without violating the safety requirement. Note also that similar rules can be added for other relations of our program. We suggest that the reader test the additions to the program to see that they do indeed give the correct results. For more practice, experiment with closed world assumptions for $mother$, $parent$, and $brother$.

Notice that expanding our original knowledge base required only the addition of rules but no modification of them, indicating a reasonably high degree of elaboration tolerance.

4.1.2 Defining Orphans

Here is another example. Consider a collection of people represented by the sort $person$:

```
person(mary).
person(bob).
person(mike).
person(rich).
person(kathy).
person(patty).
```

Assume that we have a complete list of children represented by the relation $child(person)$:

```
child(mary).
child(bob).
```

For each child our knowledge base includes the name of the child's mother ($mother(person, person)$), and the same for the child's father.

```
father(mike,mary).
father(rich,bob).
mother(kathy,mary).
mother(patty,bob).
```

Let us also assume that the knowledge base contains a complete record of deaths, represented by $dead(person)$:

```
dead(rich).
dead(patty).
```

Completeness of information about the above relations can be expressed by closed-world assumptions:

```
-child(X) :- person(X),
             not child(X).
-father(F,C) :- person(F),
                child(C),
                not father(F,C).
-mother(M,C) :- person(M),
                child(C),
                not mother(M,C).
-dead(X) :- person(X),
            not dead(X).
```

Our knowledge base has the record of a child, Mary, whose parents are Mike and Kathy. Since their death is not recorded, they must be alive. Another child recorded in the knowledge base is named Bob. His parents, Rich and Patty, have died.

Assume now that our goal is to teach an agent the notion of an orphan. Before we try to create a mathematical definition, we must first understand what it means for someone to be an orphan. One dictionary defines an orphan as a child whose father and mother are dead. Another notes that sometimes a child who has lost only one parent can be considered an orphan. The question of which definition to use must be resolved by the "users" prior to coding.

Let's stick with the first definition of orphan. The definition can be given by the following rules:

```
25 parents_dead(P) :- father(F,P),
26                    mother(M,P),
27                    dead(F),
28                    dead(M).
29
30 orphan(P) :- child(P),
31              parents_dead(P).
32 -orphan(X) :- person(X),
33               not orphan(X).
```

Here $parents\_dead$ is an auxiliary predicate added for readability. For simplicity we assume that users of the program are not even aware of its existence. This allows us to skip defining when $parents\_dead(P)$ is false. The next rules encode the definition of the orphan. The closed world assumption is justified by completeness of our death records and the information about the names of children's parents. Let's construct a program called `orphans.lp` from lines 1–33 and see what STUDENT knows. Extension `.lp` (for logic program) is arbitrary. If we ask whether Bob is an orphan, we get *yes*. Query $orphan(mary)$ is answered by *no*. These are, of course, the expected answers. Note, however, that if we expand our program by the following,

```
person(perry).
child(perry).
mother(patty,perry).
```

and ask the new program whether Perry is an orphan, the answer is a definite *no*. Intuitively, however, this may not be a correct answer, since the unknown father of Perry could also be dead. The program is not responsible for the wrong answer, because we have violated the assumption requiring that the knowledge base contain names of both parents of each child. Nevertheless, it may be more prudent to modify our program to reject the new input (see Exercise 4).

4.1.3 Defining Ancestors

You are given Bill's family tree shown in Figure 4.1.

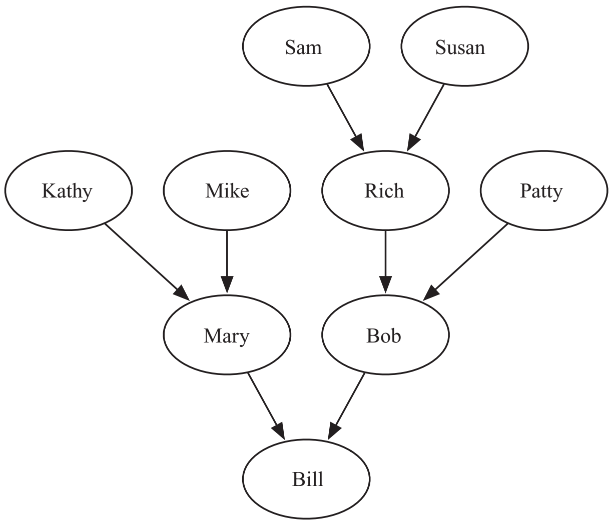

Figure 4.1. Ancestors.

We can encode it as follows:

```
person(bill).  person(mary). person(bob).
person(kathy). person(mike). person(rich).
person(patty). person(sam).  person(susan).

father(bob,bill).
father(rich,bob).
father(mike,mary).
father(sam,rich).
mother(mary,bill).
mother(patty,bob).
mother(kathy,mary).
mother(susan,rich).
```

We assume that the knowledge base contains complete information about the parents of each child in the domain, which can be expressed by the rules

```
-father(F,C) :- person(F), person(C),
                not father(F,C).
-mother(M,C) :- person(M), person(C),
                not mother(M,C).
```

(Note that by children we mean everyone except Sam, Susan, Mike, Kathy, and Patty whose parents are unknown. But because the domain contains no persons not mentioned in the family tree, this closed world assumption is justified even for them.)

Our goal is to teach the agent the notion of $ancestor(X, Y)$ – "X is an ancestor of Y" – where X and Y are of sort $person$. How would we define the notion of ancestor? Are your parents your ancestors? There are different definitions. Let us pick one and assume that they are, as are their parents' parents, and so on and so on. Unfortunately, the computer does not understand that last English expression, but with the proper language, it can be made to reason about it using **recursive definitions**.

The general structure of a recursive definition requires that the base case for a concept be defined. For example, our closest ancestors are our parents. Then we assume that we know how to define some ancestor n and concentrate on expressing what it means to be an $(n + 1)$-th ancestor.

ASP lends itself naturally to recursive definitions. This gives it a substantial advantage over knowledge representation languages that do not allow recursion (e.g., traditional relational databases). The end result is both more elegant and more economical. Here is an ASP definition of ancestor:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
-parent(X,Y) :- person(X), person(Y),
                not parent(X,Y).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y),
                 ancestor(X,Z).
-ancestor(X,Y) :- person(X), person(Y),
                  not ancestor(X,Y).
```

Note that the assumption of completeness of our information about children justifies the CWA rules for parents and ancestors. This program allows our agent to conclude that Bob is Bill's ancestor, as well as Mary, Rich, Patty, Mike, Kathy, Sam and Susan. We also conclude that Mary is not Bob's ancestor, and so on. In general the program gives a definite answer to a question $ancestor(a, b)$ for any two persons a and b.

The definition of ancestors concludes our discussion of family relationships. We hope that even these simple programs are sufficient to give some insight into the power of ASP. The process of programming in ASP is rather natural and does not require a lot of knowledge about the language. The resulting programs are clear, concise, and elaboration tolerant. Writing

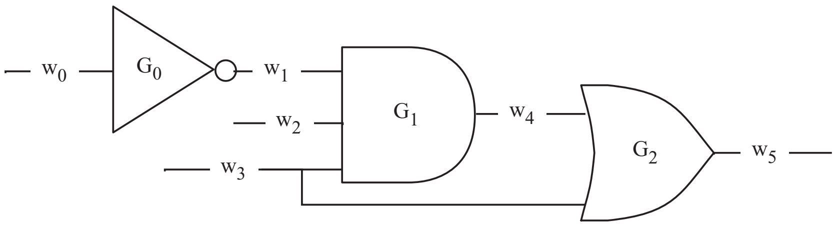

Figure 4.2. Electrical Circuit.

something comparable in traditional procedural or object-oriented programming languages such as *C* or *C++* would require good knowledge of data structures and substantially more effort. The result would be less readable and more difficult to modify. A really good programmer, however, might achieve better efficiency than that which can be achieved by existing ASP inference engines; however, this increased efficiency would hardly matter for this domain (even for a family with hundreds of thousands of members). Of course, implementations of ASP are constantly improving as well, so without extra effort by the knowledge base programmer, the efficiency of the implementation is increasing all the time, although it is unlikely to ever match a truly expert implementation in a lower level language. The choice of language for a particular task, as always, depends on the importance of program efficiency versus the price paid in developers' time.

4.2 Reasoning about Electrical Circuits

Suppose you were asked to describe a simple electrical circuit in ASP. What would be the first thing you would try to do? Desperately look for your old EE book to review what a circuit looks like? Describe gates? Wires? Connections? Just so we are all on the same footing, let's look at a picture of what we mean by a simple circuit (Fig. 4.2). G_0 is a NOT gate, G_1 is an AND gate, and G_2 is an OR gate. Let's keep it simple and limit ourselves to two-valued circuits.

So what next? When writing any logic program, a person should always ask oneself: "What are the objects and the relations that I am trying to represent?" The objects are wires and gates, and the connections are the relationships between them. Though it sounds obvious, this identification is not a trivial step. For example, it is tempting to think of a gate and its inputs and outputs as a single object. It turns out, however, that doing so would complicate the representations. Gates can have different numbers of inputs, forcing us to distinguish between gates that are otherwise similar.

Output wires of some gates can be input wires to others, but storing them with the gates does not specify their function. The identification of objects and relations in a domain greatly affects the representation and the ease and elegance of programming.

So, if we choose gates and wires as our objects, we can describe the circuit in Figure 4.2 as follows:

```
wire(w0).   wire(w1).   wire(w2).
wire(w3).   wire(w4).   wire(w5).

gate(g0).
type(g0,not_g).
input(g0,w0).
output(g0,w1).

gate(g1).
type(g1,and_g).
input(g1,w1).
input(g1,w2).
input(g1,w3).
output(g1,w4).

gate(g2).
type(g2,or_g).
input(g2,w4).
input(g2,w3).
output(g2,w5).
```

Now consider current that runs along the wire. We can represent its value by adding a new sort:

```
signal(0).
signal(1).
```

and predicate *val* that gives the value of the signal for a given wire. For example,

```
val(w0,1).
val(w2,0).
val(w3,1).
```

Given the values on input wires, the agent should be able to predict values on all other wires. To do this, it must know how NOT, AND, and OR gates function. This is accomplished by the following recursive definition of relation *val*:

```
% A NOT gate flips the value of the signal:

opposite(0,1).
opposite(1,0).

val(W1,V1) :- output(G,W1),
              type(G,not_gate),
              input(G,W0),
              val(W0,V0),
              opposite(V1,V0).

% The output of an AND gate is 0 if at least one input is 0:

val(W1,0) :- output(G,W1),
             type(G,and_gate),
             input(G,W0),
             val(W0,0).

% It is 1 otherwise:

val(W1,1) :- output(G,W1),
             type(G,and_gate),
             -val(W1,0).

% The output of an OR gate is 1 if at least one input is 1:

val(W1,1) :- output(G,W1),
             type(G,or_gate),
             input(G,W0),
             val(W0,1).

% It is 0 otherwise:

val(W1,0) :- output(G,W1),
             type(G,or_gate),
             -val(W1,1).
```

Finally, negation of the relation *val* is defined by the closed world assumption:

```
-val(W,V) :- wire(W), signal(V),
             not val(W,V).
```

We may also add the rule

```
-val(W,V1) :- signal(V1),
              val(W,V2),
              V1 != V2.
```

to avoid erroneous input that assigns both 0 and 1 to an input wire. (Note that the statements describing sorts of variables in the bodies of the last two rules guarantee the rules' safety.) It is not difficult to show that, given the proper values of these wires, the program computes the unique value for every other wire of the circuit.

Query STUDENT with $val(W, V)$. Does it predict the output values correctly? For more practice, run the program on several other sets of inputs.

Notice that it is easy to build on to the circuit. Adding objects amounts to naming them. If they are gates, we must specify their type. Hooking them up to the configuration requires stating the gate's inputs and outputs. The existing definitions do not need to be changed in any way.

What we just encoded can be viewed as a kind of expert knowledge about circuits. It is not unreasonable to imagine that this simple example could be expanded into some sort of advice-giving system for humans dealing with complex circuitry. If humans were to safely rely on it for procedural instructions, this system must have some commonsense knowledge on top of its expert knowledge. Suppose this system was smart enough to determine that one of the gates was defective. First, it might need to be able to draw the conclusion that, if a component is defective, it must be replaced. Second, it may need to know that it is dangerous to replace a component in a system if there is current running through it. ASP gives us the power to add this knowledge without changing languages, systems, and the like. A possible implementation of this idea looks like this:

```
%% Assume we have a sensor that tells us the actual
%% value of the output wire of a gate by setting the
%% value of predicate sensor_val for that wire.
%% Then if the sensor value does not match the
%% predicted value, the gate must be defective.
%% To test the program, we artificially set sensor_val:
sensor_val(w1,1).

defective(G) :- output(G,Output_wire),
                sensor_val(Output_wire, SV),
                val(Output_wire,V),
                SV != V.

needs_replacing(G) :- defective(G).

%% We can also encode the knowledge that a gate is
%% dangerous to replace if any of its input wires
%% might have the value 1; i.e., if it is not known
%% whether the value of W is 0.

dangerous_to_replace(G) :- input(G,W),
                           not val(W,0).
```

(Note that if we assume completeness of information about val, then `not val(W,0)` can be replaced by `-val(W,0)`.) Based on this information, the system might perform some prearranged function such as notifying the operator or shutting off the current.

4.3 Hierarchical Information and Inheritance

Consider how one might represent the following information in ASP.

- The *Narwhal* is a submarine.
- A submarine is a vehicle.
- Submarines are black.[1]
- The *Narwhal* is a part of the U.S. Navy.

Here is one possible solution:

```
sub(narwhal).
vehicle(X) :- sub(X).
black(X) :- sub(X).
part_of(narwhal,us_navy).
```

[1] The reader has probably noticed that our specification makes a generalization about the color of submarines. It would have been better to say that *normally* submarines are black. In the next chapter, we revisit hierarchies and show how to express such statements, known as defaults.

This program is short and encodes exactly what was given. It allows an agent to conclude that the *Narwhal* is a submarine, is a vehicle and is black. However, when a human hears the story, there is much commonsense information that is assumed. For example, suppose we ask our agent whether the *Narwhal* is a car. A human could answer this question without further information, but STUDENT would rightfully answer *unknown*, because we did not teach it that cars are not submarines. Likewise, if we wanted to know if the submarine was red, we would have the same problem. This lack of negative information could be remedied by adding the following axioms to the program:

```
-car(X) :- sub(X).
-sub(X) :- car(X).
-red(X) :- black(X).
-black(X) :- red(X).
```

Note that as soon as we decide we want to allow other types of vehicles and other colors in our program, we need to add two lines for each addition just so that the agent could correctly answer some simple commonsensical queries.

We can come up with a better representation if we exploit the hierarchical nature of the information from our story. Humans are good at organizing information about the world into tree-like structures of classes and subclasses. For example, when we are told that the *Narwhal* is a submarine, we understand that it belongs to a class of things that are called submarines and, thus, has certain properties that all submarines are assumed to have. With a submarine being a vehicle, objects that are submarines will inherit properties of vehicles as well. An **inheritance hierarchy** is a collection of classes organized into a tree formed by the subclass relation. Figure 4.3 shows the hierarchical structure of the classes in the story. Note that children of the class do not necessarily form its partition. In other words we do not exclude the existence of classes not mentioned in our representation. Thus, there is the possibility that our domain contains unknown or irrelevant classes that the designer decided not to include in the signature of our program. As a result the reasoner associated with the program will not be able to conclude that a vehicle x is either a submarine or a car – it could belong to some other class of vehicle left outside of our representation and not included in the hierarchy.

To represent this hierarchy we identify the implicit classes relevant to our story and make them objects of our domain. Once concepts become objects of the domain, we can define relations on them and explicitly reason about

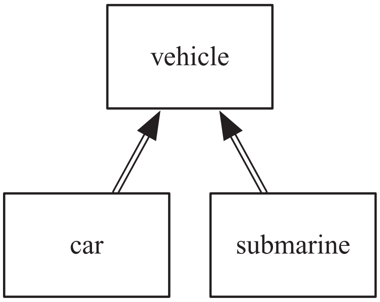

Figure 4.3. Subclasses in the Expanded Submarine Story.

whether they satisfy these relations. This process, often called **reification**, is frequently used to generalize and improve the quality of knowledge representation.

Syntactically reification of classes is accomplished by introducing a new sort $class$

```
class(sub).
class(car).
class(vehicle).
```

Relation $is\_subclass(C_1, C_2)$ corresponding to a subclass link of the hierarchy is defined as follows:

```
is_subclass(sub,vehicle).
is_subclass(car,vehicle).
```

The subclass relation is defined as the transitive closure[2] of $is\_subclass$:

```
subclass(C1,C2) :- is_subclass(C1,C2).

subclass(C1,C2) :- is_subclass(C1,C3),
                   subclass(C3,C2).
```

As usual in recursive definitions we also add

```
-subclass(C1,C2) :- class(C1),
                    class(C2),
                    not subclass(C1,C2).
```

[2] A binary relation R^* is called the transitive closure of binary relation R if $R \subseteq R^*$ and for all X, Y, Z if $R(X, Z)$ and $R(Z, Y)$ then $R(X, Y)$ and no proper subset of R^* satisfies these properties.

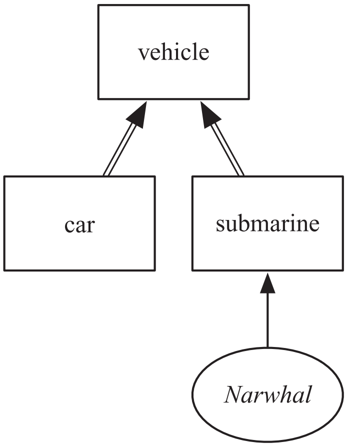

Figure 4.4. Adding an is_a Link.

To be able to talk about *objects of the classes* of the hierarchy, such as the *Narwhal*, we expand our picture by allowing a new type of link connecting objects to their corresponding classes (see Fig. 4.4).

To represent these new links we introduce a new sort, $object$:

```
object(narwhal).
```

and a new relation, $is\_a(X, C)$, where X is an object and C is a class:

```
is_a(narwhal,sub).
```

Now we define the main relation between objects and classes, called $member(X, C)$. The positive part of membership is defined as follows:

```
member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0),
               subclass(C0,C).
```

But, unlike the case of the $subclass$ relation, we do not have complete information about membership. For example, Figure 4.5 shows a vehicle called the *Mystery*, but we do not know what kind of vehicle it is.

```
object(mystery).
is_a(mystery,vehicle).
```

We allow such members in our hierarchies and expect an agent to answer *unknown* to queries about whether the *Mystery* is a car or a sub. Therefore, we do not wish to use the CWA to represent negative information about membership.

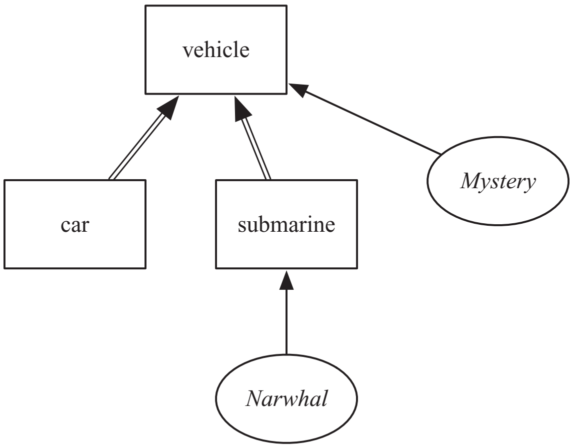

Figure 4.5. Incomplete Information about the Classification of the *Mystery*.

However, the designers of hierarchies often make a weaker assumption: *Normally, children of a class in a hierarchy are disjoint.* Thus in the absence of information to the contrary, it is reasonable to conclude that the *Narwhal* cannot be both a submarine and a car. Of course, it is possible to have exceptions to this rule – plenty of action heroes have vehicles that are difficult to categorize – but we do not discuss handling such cases until the next chapter. For now, we drop the word "normally" and assume that sibling subclasses are disjoint. This can be expressed by the following rules:

```
siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.
-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2.
```

These rules are sufficient to define the hierarchical structure.

In addition to classes of vehicles and their members, the story also talks about colors. Their representation is substantially facilitated by the same process of reification we used for classes. We make colors full-fledged citizens of our domain by introducing a sort *color*

```
color(black).
color(red).
```

The fact that the submarines are black is expressed as

```
28 has_color(X,black) :- member(X,sub).
```

Note that now we can ask what color something is, not just whether it is black. Recall that in our first implementation of the story, we wanted our program to realize that, if a submarine is black, then it is not red. In other words, we had an implicit assumption that objects can only have one color. With our new representation this assumption can be made explicit by the following axiom that, unlike the axioms we were forced to add to the first version, will work once and for all, no matter how many colors we choose to introduce:

```
29 -has_color(X,C2) :- has_color(X,C1),
30                     color(C2),
31                     C1 != C2.
```

(If we want to talk about multicolored objects, we simply introduce new colors, such as $black\_red$.)

The last line of the original program

```
32 part_of(narwhal,us_navy).
```

remains unchanged.

The new program (lines 1–32) is clearly more powerful than the first one. Anything the old program can do, this program can do better. Not only can our agent answer all previous questions correctly, including that the *Narwhal* is not a car and is not red, but it also understands some deep, general notions about the nature of hierarchies and colors. With the first program, we can only ask if a particular submarine is a vehicle, not whether submarines in general are vehicles. It is easy to formulate these questions for the second program. It correctly answers both queries $member(narwhal, vehicle)$ and $subclass(sub, vehicle)$.

Reification has made our program not only more general but also more elaboration tolerant (i.e., more easily modifiable). For instance, to conclude that the *Narwhal* is not white, we simply familiarize our agent with that color by adding $white$ to the definition of the sort $color$:

```
33 color(white).
```

If we wanted to say that vehicles are a type of machine, we could write

```
34 class(machine).
35 is_subclass(vehicle, machine).
```

Thanks to our subclass axioms, the whole tree structure would be adjusted automatically. Our agent would know that the *Narwhal* is a submarine, which is a type of vehicle, which is a type of machine. It would also know that it is not a car. Try asking $member(narwhal, X)$.

If we said that a vehicle is something that can be used for traveling, the second program could easily be expanded to allow the agent to deduce that this property would be true for all subclasses of vehicle. It would need no changes, just the following addition:

```
used_for_travel(X) :- member(X,vehicle).
```

If we wanted to say that machines are not alive, we could just add

```
-alive(X) :- member(X,machine).
```

We could then derive that machines, vehicles, cars, and submarines are not alive.

We have seen how the process of reification and the methodology of hierarchical organization can be very useful in creating elaboration-tolerant programs. The main differences between the two solutions of the original problem are their brevity, generality, and modifiability. Initially, the first program was shorter, but with minor subsequent changes, it lost this advantage. The second program is more general and more elaboration tolerant, which allows it to incorporate changes without substantial growth in size.

It is important to realize that the quest for elaboration-tolerant programming paradigms has been ongoing since the beginning of computer science and has included such developments as high-level languages, the notion of procedure and module, and object-oriented programming. However, having helpful tools does not necessarily lead to their appropriate use. When developing programs in any language, we must always strive to predict just how much a program will be expected to change and then balance compactness with elaboration tolerance.

Summary

In this chapter we have seen that the ASP-based declarative approach to knowledge representation is applicable to a wide variety of domains. We showed it to be capable of representing definitions (including recursive ones), open and closed world assumptions, and hierarchical knowledge. In practice this approach has been used successfully in circuit design, decision support systems for space shuttle controllers, team building that ensures fair scheduling of employees with necessary skills, automatic systems for software configuration management, data integration, semantic-web programming, and more, and applications are emerging in such varying fields as molecular biology, psychology, and linguistics.

We also discussed the basic methodology of representing knowledge in ASP, including the importance of the good selection of objects and relations of the domain, the power of reification, and the necessity of explicitly stating commonsense assumptions and thinking about possible extensions of the domain and the degree of elaboration tolerance of your program. In the next chapter we show how this methodology can be extended to represent another important concept of knowledge representation – defaults and their exceptions.

References and Further Reading

A good source for learning more about representing knowledge in Answer Set Prolog is *Knowledge Representation, Reasoning, and Declarative Problem Solving* by Chitta Baral (2003). The family example and various forms of inheritance hierarchies have been discussed in many books on logic programming and Prolog. These, however, assume the CWA for all relations of the domain; making this assumption for individual predicates requires explicit representation. In logic programming, such a representation became possible after the introduction of classical negation in Gelfond and Lifschitz (1991). Our representation of circuits is a special case of the more general representation from Balduccini, Gelfond, and Nogueira (2000), which deals with circuits with delays and possibly undefined signals. Reasoning about inheritance hierarchies has a long history, which probably started with Aristotle. Formalization of various forms of reasoning about hierarchies based on classical logic can be found in Baader et al. (2003).

Reasoning with inheritance hierarchies based on their graphical representation is described in Sowa (2000). For an approach to knowledge representation based on well-founded semantics of logic programs, one can consult Alferes and Pereira (1996). To get a better idea of a range of applications of ASP one can look at Nogueira et al. (2001) and Balduccini, Gelfond, and Nogueira (2006) (decision support for the space shuttle controllers), Ricca et al. (2012) (automation of the team-building process for the largest Italian seaport), Soininen and Niemela (1999) (product configuration), Manna et al. (2012) (information extraction from the WEB), Boenn et al. (2011) (music composition), Aker et al. (2011) (robotics), Inclezan (2013) (linguistics), and Balduccini and Girotto (2010) (psychology).

Exercises

1. Define and test relation
   $brothers(X, Y)$ — “X and Y are brothers”.
2. Define and test relation
   $uncle(X, Y)$ — “X is an uncle of Y”.
   To test this relation you will need to populate your world with relatives of John and/or Alice.
3. Consider the program obtained from our family knowledge base (lines 1–34) by removing rules on lines 28–31. Does this change the answers to our three queries from Section 4.1? Can you think of a reason to keep these rules? *Hint:* Consider (possibly erroneous) updates to the knowledge base.
4. Expand program `orphans.lp` from Section 4.1.2 to guarantee that input that violates the assumption concerning the completeness of information about a child’s parents causes the program to become inconsistent.
5. Can statement `person(X)` be removed from the rule on line 15 in program `orphans.lp` from Section 4.1.2? Explain your answer.
6. Add a gate (with some input and output wires) to the configuration in Section 4.2 and test the program on some values.
7. A directed graph G can be described by a set of vertices, represented by facts $vertex(a), vertex(b), \ldots$ and a set of edges, represented by facts $edge(a, b), edge(a, c), \ldots$ Use ASP to define relation $connected(X, Y)$ that holds iff there is a path in G connecting vertices X and Y.
8. Consider a directed graph represented as in the previous exercise, but assume that some of its edges can be blocked (denoted as $blocked(X, Y)$). Redefine relation $connected(X, Y)$ as follows: Two vertices X and Y of the graph are *connected* iff there a path from X to Y such that no edge of this path is blocked.
9. “Jets are faster than birds. There is an eagle that is faster than every robin. The SR-71 Blackbird is a jet. Jo is a robin.” Write a program to describe this story. Make sure that it can derive that the SR-71 is faster than Jo. *Hint:* Build a hierarchy of flying objects.
10. Modify the existing vehicle hierarchy to add the new subclasses and member shown in Figure 4.6. Add some property of water vehicles to your program and make sure that the *Narwhal* inherits this property and *Abby* does not.

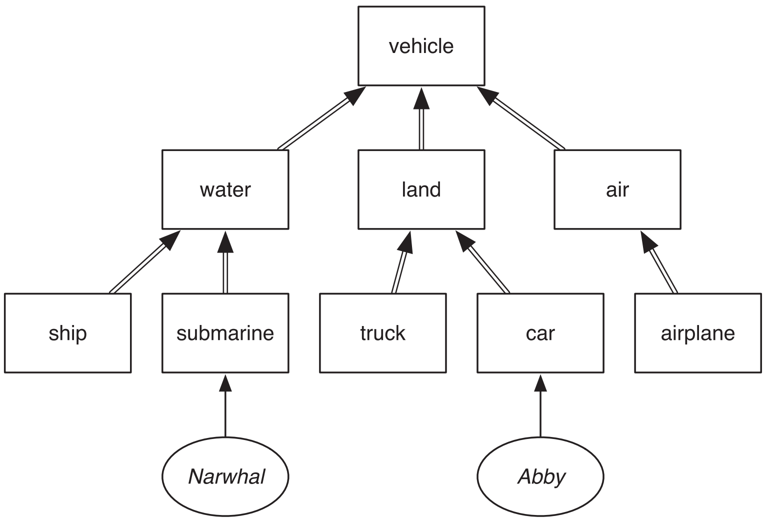

Figure 4.6. New Subclasses of Vehicles.

11. Consider a hierarchy where is_a is applicable only to members of leaf classes. Note that this would imply that information about membership is complete.
    (a) Give a recursive definition of membership using CWA. (Make sure $\neg member$ is part of your definition.)
    (b) The program in Exercise 11a still has the assumption that sibling classes are disjoint. Create a new program by removing this assumption. Compare the two programs. Do they have the same answer sets? Do they behave equivalently under updates? *Hint:* Consider the addition of an element that belongs to two subclasses of the program.

12. Consider the description of hierarchy as in Section 4.3 with the difference that we assume that we have complete information about the subdivision of subclasses (i.e., subclasses form a complete partition of the parent class, so that the sum of the parts forms a complete whole). This can be expressed by adding the following rules:

```
member(X,C) | -member(X,C) :- object(X),
                              leaf(C).
in_a_leaf(X) :- object(X),
                leaf(C),
                member(X,C).
```

```
-in_a_leaf(X) :- object(X),
                 not in_a_leaf(X).
:- object(X), -in_a_leaf(X).
```

The first rule forces the agent to consider membership for each object and each leaf class, and the last three require an object to be in at least one leaf class.

(a) Define relation $leaf(C)$ that holds iff C is a leaf class of the hierarchy. *Hint:* Define $\neg leaf(C)$ first.

(b) Compare the answers that should be given by the hierarchy program that includes vehicle *Mystery* from Figure 4.5 with and without the above four rules on the following query:

$$sub(mystery) \text{ } or \text{ } car(mystery)$$