Comparing SNePS with Topbraid/Pellet*
SNeRG Technical Note 42

Michael Kandefer and Stuart C. Shapiro
Department of Computer Science and Engineering
and Center for Cognitive Science
and National Center for Multisource Information Fusion
State University of New York at Buffalo
Buffalo, NY 14260-2000
`{mwk3|shapiro}@cse.buffalo.edu`

18th July 2008

1 Introduction

In this paper we compare the SNePS knowledge representation and reasoning system (Shapiro ; Shapiro & The SNePS Implementation Group 2008), with the Topbraid Ontology Editing Tool (Top Quadrant Inc. 2007) using the Pellet OWL DL Reasoner (Clark & Parsia, LLC 2007). To compare these two system we represent two problem domains in each system, and have them reason over the data. The two problem domains are:

- The Jobs Puzzle
- The ParentSally Example

2 The Jobs Puzzle

The Jobs Puzzle is a small logic puzzle involving constraint satisfaction. It is a version of one presented in (Wos *et al.* 1984). The puzzle involves figuring out which of eight jobs each of four people has. The following section discusses the representation of the puzzle's constraints in SNePS and Topbraid, and the results of their respective automated reasoners.

2.1 The Jobs Puzzle in SNePS

To represent the Jobs Puzzle in SNePS, we use SNePS's SNePSLOG, a logical language resembling higher-order logic. The following SNePSLOG terms are used for representing the puzzle:

- `Roberta` - The person named "Roberta"
- `Thelma` - The person named "Thelma"
- `Steve` - The person named "Steve"

*This work was supported in part by CUBRC under prime contract FA8750-06-C-0184 between CUBRC and U.S. Air Force Research Laboratory, Rome, NY.

- `Pete` - The person named "Pete"
- `chef` - The job of chef
- `guard` - The job of guard
- `nurse` - The job of nurse
- `"telephone operator"` - The job of telephone operator[1]
- `"police officer"` - The job of police officer
- `teacher` - The job of teacher
- `actor` - The job of actor
- `boxer` - The job of boxer
- `Male(x)` - The proposition that `x` is a male.
- `Female(x)` - The proposition that `x` is a female.
- `Person(x)` - The proposition that `x` is a person.
- `Job(x)` - The proposition that `x` is a job.
- `HasJob(x,y)` - The proposition that `x` has the job of `y`

The puzzle and its formalization in SNePSLOG is as follows. (Each item shows an English statement from the puzzle, and its translation into SNePSLOG.)

- Roberta, Pete, Thelma, and Steve are people.
  `Person({Roberta, Thelma, Steve, Pete}).`
- The jobs are chef, guard, nurse, telephone operator, police officer, teacher, actor, and boxer.
  ```
  Job({chef, guard, nurse, "telephone operator",
       "police officer", teacher, actor, boxer}).
  ```
- Each person has exactly two of the eight jobs.
  `all(p)(Person(p) => nexists(2,2,8)(j)(Job(j):  HasJob(p,j))).`
- Each job is held by exactly one of the four people.
  `all(j)(Job(j) => nexists(1,1,4)(p)(Person(p):  HasJob(p,j))).`
- No female has the job of nurse, actor, or telephone operator.
  ```
  all(w)(Female(w)
         => andor(0,0)
                 {HasJob(w, nurse), HasJob(w, actor),
                  HasJob(w, "telephone operator")}).
  ```
  This is formulated as "A female is not a nurse, nor an actor, nor a telephone operator", instead of as "The nurse, the actor, and the telephone operator are males" so that SNePS can perform direct reasoning to conclusions of the form `HasJob(`*p*`,`*j*`)` or `~HasJob(`*p*`,`*j*`)` without using the rule of inference of *modus tollens*, which is not implemented in SNePS.
- No male has the job of chef.
  `all(m)(Male(m) => ~HasJob(m, chef)).`

[1] A term that contains a blank is entered as a string, enclosed in quotation marks.

- No person is both the chef and the police officer.
```
all(p)(Person(p) => andor(0,1){HasJob(p, chef),
                               HasJob(p, "police officer")}).
```

- Roberta and Thelma are female.
```
Female({Roberta, Thelma}).
```

- Steve and Pete are male.
```
Male({Steve, Pete}).
```

- Roberta is neither the boxer, nor the chef, nor the police officer.
```
andor(0,0){HasJob(Roberta, boxer), HasJob(Roberta, chef),
           HasJob(Roberta, "police officer")}.
```

- Pete is neither the nurse, the police officer, nor the teacher.
```
andor(0,0) {HasJob(Pete, nurse),
            HasJob(Pete, "police officer"), HasJob(Pete, teacher)}.
```

With all the constraints specified, we query the system for the job assignments:

```
: HasJob(?p,?j)?
  wff108!:  HasJob(Thelma,boxer)
  wff107!:  ~HasJob(Thelma,guard)
  wff105!:  ~HasJob(Thelma,teacher)
  wff103!:  ~HasJob(Pete,boxer)
  wff101!:  ~HasJob(Pete,guard)
  wff98!:  HasJob(Pete,telephone operator)
  wff96!:  HasJob(Pete,actor)
  wff95!:  ~HasJob(Steve,boxer)
  wff93!:  ~HasJob(Steve,guard)
  wff91!:  ~HasJob(Steve,teacher)
  wff89!:  ~HasJob(Steve,telephone operator)
  wff87!:  ~HasJob(Steve,actor)
  wff84!:  HasJob(Steve,nurse)
  wff82!:  HasJob(Roberta,guard)
  wff80!:  HasJob(Roberta,teacher)
  wff78!:  ~HasJob(Thelma,police officer)
  wff76!:  ~HasJob(Steve,chef)
  wff75!:  ~HasJob(Pete,chef)
  wff72!:  ~HasJob(Roberta,nurse)
  wff71!:  ~HasJob(Roberta,actor)
  wff70!:  ~HasJob(Roberta,telephone operator)
  wff69!:  ~HasJob(Thelma,nurse)
  wff68!:  ~HasJob(Thelma,actor)
  wff67!:  ~HasJob(Thelma,telephone operator)
  wff32!:  HasJob(Thelma,chef)
  wff28!:  HasJob(Steve,police officer)
  wff23!:  ~HasJob(Pete,nurse)
  wff22!:  ~HasJob(Pete,police officer)
  wff21!:  ~HasJob(Pete,teacher)
  wff20!:  ~HasJob(Roberta,boxer)
  wff19!:  ~HasJob(Roberta,chef)
  wff18!:  ~HasJob(Roberta,police officer)
```

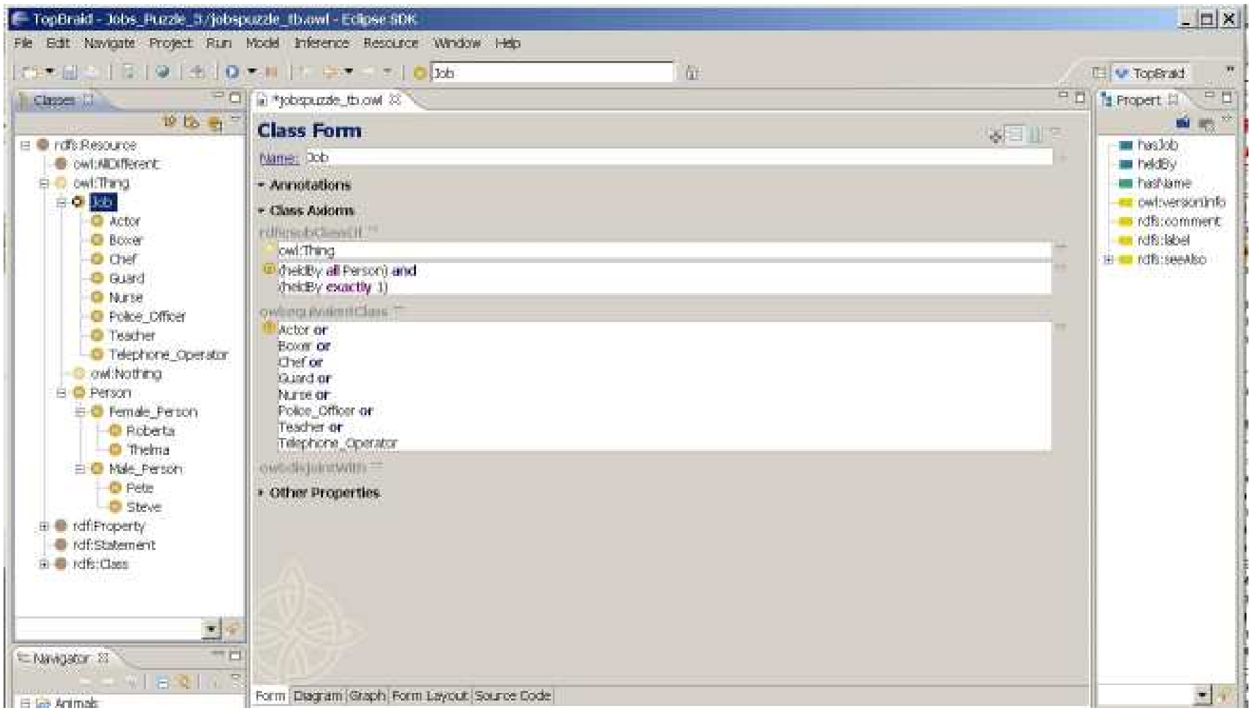


Figure 1: Topbraid - Jobs Puzzle

This is the complete solution to the puzzle: Pete is the actor and the telephone operator; Steve is the police officer and the nurse; Thelma is the boxer and the chef; and Roberta is the guard and the teacher. In addition, the $4 \times 6 = 24$ negative conclusions are found: who doesn't hold which jobs.

2.2 The Jobs Puzzle in Topbraid/Pellet

The Topbraid Ontology Editor (Top Quadrant Inc. 2007) can create and edit ontologies, load them from, and save them as RDF or OWL files. It has a user interface, depicted in Figure 1, which allows a user to organize classes into a class hierarchy (the upper left frame displays this hierarchy), and edit properties of those classes in the class form frame (the upper middle frame). Standard OWL Full RDF schema features (W3C 2004a) are provided, and new properties can be specified by the user in the properties frame (the upper right frame).

To represent the Jobs Puzzle in Topbraid, we created two subclasses of `owl:Thing`, the top-level OWL class: `Job`, and `Person`. `Job` has as mutually distinct subclasses the eight jobs (`Actor`, `Boxer`, `Chef`, `Guard`, `Nurse`, `Police_Officer`, `Teacher`, and `Telephone_Operator`). `Person` is partitioned into the two disjoint and exhaustive subclasses, `Female_Person` and `Male_Person`. `Female_Person` is partitioned into the subclasses `Thelma` and `Roberta`, while `Male_Person` is partitioned into the subclasses `Pete` and `Steve`. Each of the leaf subclasses, `Actor`, `Roberta`, *etc.* has a single instance (e.g. `Ind_Roberta` and `Actor_Pete` are the only instances of `Roberta` and `Actor`, respectively). The use of names as classes was done because one cannot place contraints on the instances in Topbraid, only the classes; and SWRL rules, which allow for reasoning to be done on instances, proved insufficient for the reasoning needed by this domain. With the hierarchy in place, the only components necessary for formalizing constraints are relationships. We've made `hasJob` a relation between a `Person` and `Job`, and `heldBy` its converse relationship (its `owl:inverseOf`).

To formalize the constraint that all jobs are held by one person, we placed the condition on the `Job rdfs:subclassOf` property that

```
owl:Thing and heldBy exactly 1 and heldBy all Person
```

and on the `Job owl:equivalentClass` property that

```
Actor or Boxer or Chef or Guard or Nurse or Police_Officer
      or Teacher or Telephone_Operator
```

To formalize the constraint that all people have two jobs and that no person holds both the job of the `Police_Officer` and `Chef`, the `Person rdfs:subclassOf` property was given the condition

```
owl:Thing
and ((hasJob some Police_Officer) and not hasJob some Chef)
or ((hasJob some Chef) and not hasJob some Police_Officer)
or (not hasJob some Chef and not hasJob some Police_Officer)
and hasJob exactly 2
and hasJob all Job
```

The constraint that the jobs of `Nurse`, `Actor` and `Telephone_Operator` are not held by a female is specified by placing on `Female_Person rdfs:subclassOf` the two restrictions

```
Person
(Person
 and not hasJob some Nurse
 and not hasJob some Actor
 and not hasJob some Telephone-Operator)
```

A similar constraint is placed on `Male_Person rdfs:subclassOf` to indicate that males cannot be the chef:

```
Person
not hasJob Chef
```

To indicate the Jobs the individuals cannot hold, similar restriction are placed on their `rdfs:subclassOf` restriction sets. For `Roberta`:

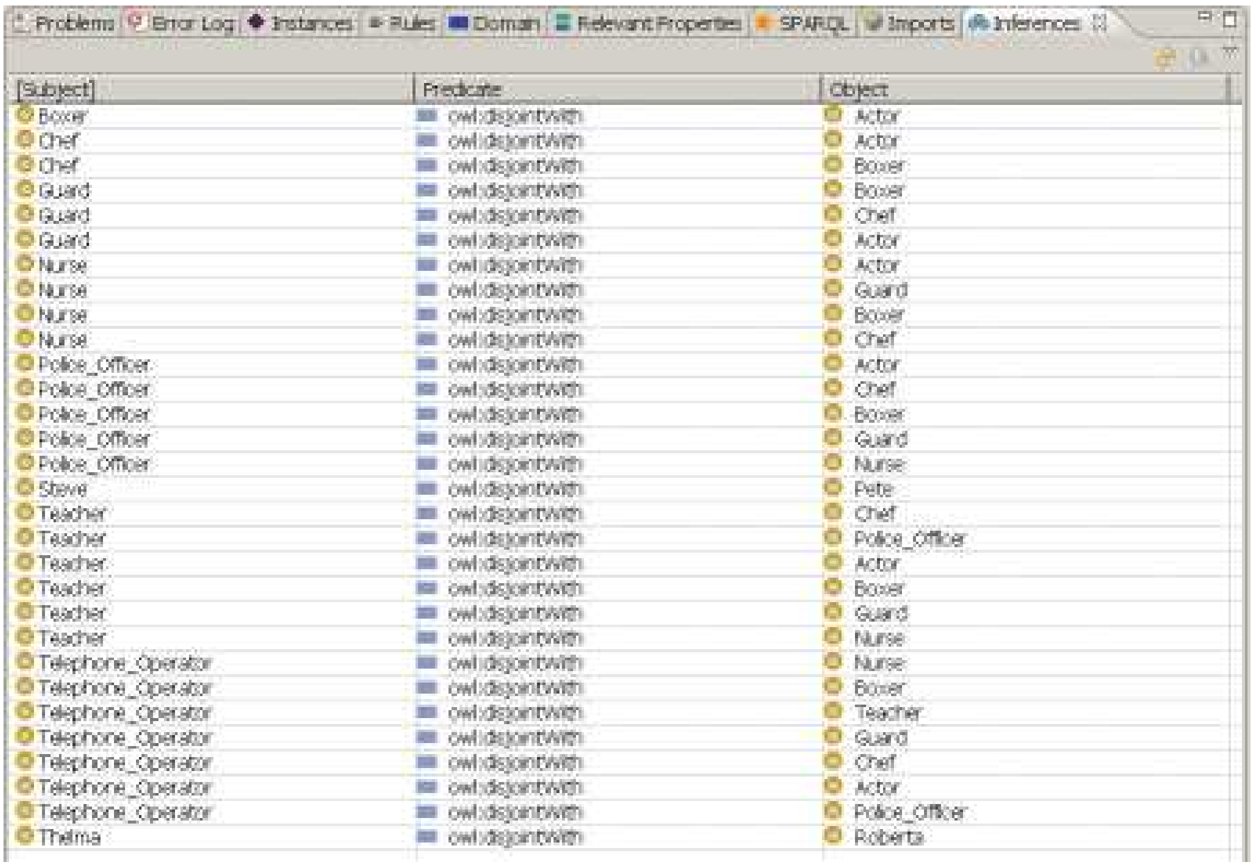
```
Female_Person
not hasJob some Boxer
and not hasJob some Chef
and not hasJob some Police_Officer
```

For `Pete`:

```
Male_Person
not hasJob some Teacher
and not hasJob some Nurse
and not hasJob some Police_Officer
```

With all the constraints represented, the Pellet reasoner is invoked. However, after finishing the reasoning process none of the individual constants have their `hasJob` or `heldBy` relationships filled. The reasoner does draw conclusions, such as several `owl:disjointWith` relationships between classes that weren't asserted directly, but these do not suffice for a solution to the puzzle. The inferred relationships are shown in Figure 2.

Though Topbraid and Pellet were unable to provide a solution to the puzzle automatically, we were able to check various combinations of solutions by running the consistency checker after entering values for the `hasJob` relationships for each individual. If a particular combination was inconsistent, the consistency checker would report it. The correct values caused no inconsistency.



[Subject]	Predicate	Object
Boxer	owl:disjointWith	Actor
Chef	owl:disjointWith	Actor
Chef	owl:disjointWith	Boxer
Guard	owl:disjointWith	Boxer
Guard	owl:disjointWith	Chef
Guard	owl:disjointWith	Actor
Nurse	owl:disjointWith	Actor
Nurse	owl:disjointWith	Guard
Nurse	owl:disjointWith	Boxer
Nurse	owl:disjointWith	Chef
Police_Officer	owl:disjointWith	Actor
Police_Officer	owl:disjointWith	Chef
Police_Officer	owl:disjointWith	Boxer
Police_Officer	owl:disjointWith	Guard
Police_Officer	owl:disjointWith	Nurse
Steve	owl:disjointWith	Pete
Teacher	owl:disjointWith	Chef
Teacher	owl:disjointWith	Police_Officer
Teacher	owl:disjointWith	Actor
Teacher	owl:disjointWith	Boxer
Teacher	owl:disjointWith	Guard
Teacher	owl:disjointWith	Nurse
Telephone_Operator	owl:disjointWith	Nurse
Telephone_Operator	owl:disjointWith	Boxer
Telephone_Operator	owl:disjointWith	Teacher
Telephone_Operator	owl:disjointWith	Guard
Telephone_Operator	owl:disjointWith	Chef
Telephone_Operator	owl:disjointWith	Actor
Telephone_Operator	owl:disjointWith	Police_Officer
Thelma	owl:disjointWith	Roberta

Figure 2: Topbraid - Jobs Puzzle Inferences

Figure 3: Topbraid - ParentSally Example Classification Hierarchy

2.3 Comparison

Its clear that both Topbraid and SNePS are capable of representing the constraints of the Jobs Puzzle. SNePS uses a higher-order logic notation with specialized logical operators (such as `nexists` and `andor`) to accomplish this task, while Topbraid uses the OWL Full restrictions. Where the two systems differ is in the reasoning. SNePS is capable of reasoning to the puzzle's solution, while the Pellet reasoner in Topbraid cannot. This is because Pellet was primarily designed to reason over OWL ontologies, which doesn't allow for inferring the values of a particular instance's relationship slots from negative information (i.e. by asserting the relationship is not of some class, the reasoner cannot conclude it is of another). Though there exists a logical notation for OWL called SWRL (W3C 2004b) that can fill in relation slots, an attempt at representing the puzzle using this import proved unsuccessful, as SWRL rules lack negation. As discussed previously, an automatic solution could be built around Pellet that would reason to the solution by trying various possibilities for the `hasJob` relationship by utilizing the Java API, but this is beyond the scope of this study. These reasoning issues with Pellet are easily handled in SNePS using `nexists`, which was created precisely to handle reasoning from negative information (Shapiro 1979).

3 The ParentSally Example

The ParentSally Example is a domain of our creation that was designed to illustrate Description Logic reasoning. The domain includes the classes of Sex and Animal. The Animal type has as its subclasses Person, Cattle, and Dog. Cattle has only one subclass, Cow, while Person is divided into Man, Woman, and Parent subclasses. All classes are considered subclasses of the class Thing, which is the top-level class. Various properties are ascribed to each subclass, and will be discussed in the following sections.

3.1 ParentSally in Topbraid/Pellet

Topbraid as a description logic framework was built to represent classification hierarchies. As such there is little effort required to represent the hierarchy and populate it with instances. Figure 3 shows the hierarchy. All the classes needed, plus the additional `PersonWithAtMost1Child`, which will be discussed later, are shown in the leftmost frame. The rightmost frame shows the relationships used; `hasChild` (and its `owl:inverseOf` relationship `isChildOf`, and `hasSex` (and its `owl:inverseOf` relationship `isSexOf`). Finally, the middle frame displays the `Woman` class, which specifies that it is `owl:disjointWith` the `Man` class. In addition to the classes: `Fred` is created as an instance of `owl:Thing`; `Elsie` as an instance of `Cattle`; `Lucy`, `Pete`, `Tom`, and `Sally` as instances of `Person`; and `male` and `female` as instances of `Sex`.

With the hierarchy established, we begin creating restrictions on the classes using the relationships. Restrictions in the `rdfs:subClassOf` and `owl:equivalentClass` restriction sets represent necessary, and necessary and sufficient properties for being members in those classes respectively. The class of `Man` is declared to be an `rdfs:subclassOf`

```
Person and hasSex has male
```

`Woman` is an `rdfs:subclassOf`

```
Person and hasSex has female
```

`Cow` is the `rdfs:subclassOf` and `owl:equivalentClass` of

```
Cattle and hasSex has female
```

We give the class `Parent` the necessary and sufficient conditions that a parent is a person with at least one child by placing a restriction in its `rdfs:subClassOf` and `owl:equivalentClass` of:

```
Person and hasChild min 1
```

and we add to `Pete`'s representation

```
Pete rdfs:type Parent
Pete hasChild Fred
```

Topbraid/Pellet concludes that `Fred IsChildOf Pete`.

Then, we set the `isSexOf` relationships for the two `Sex`es as

```
male isSexOf Fred
female isSexOf Elsie
```

Topbraid adds `Fred hasSex male` to `Fred`'s representation, and `Elsie has Sex female` to `Elsie`'s.

The results of then running the Pellet reasoner is that `Fred` remains in the category of `Thing` (Figure 4), while `Elsie` is inferred to be a `Cow` (Figure 5).

To make the reasoner conclude that Fred is a person we specify that all the children of parents are people, by adding to the `Parent` `rdfs:subclassOf` and `owl:equivalentClass` restrictions, so that they are now

```
Person and hasChild min 1 and hasChild all Person
```

Notice that this also means that any person all of whose children are people is a parent.

Running the Pellet reasoner on this allows the system to conclude Fred is a person (because Fred is Pete's child), as depicted in Figure 6.

Finally, we want to establish that `Sally` is a parent by asserting she has one child that is a person. To do this we add to Sally's `hasChild` relationship the value of `Lucy`, who is also a person. Performing Pellet reasoning on this does not conclude that Sally is a person, because Pellet operates under the open world assumption: Sally might have some as yet unknown children that aren't people. To solve this, we create a new subclass of `Person` named `PersonWithAtMost1Child` and give it a `rdfs:subClassOf` restriction of:

```
Person and hasChild max 1
```

We then add to Sally's `rdfs:type` field the class of `PersonWithAtMost1Child`, asserting that Sally is an individual of this class. Invoking Pellet reasoning now causes the reasoner to conclude that Sally is a parent, as depicted in Figure 7.

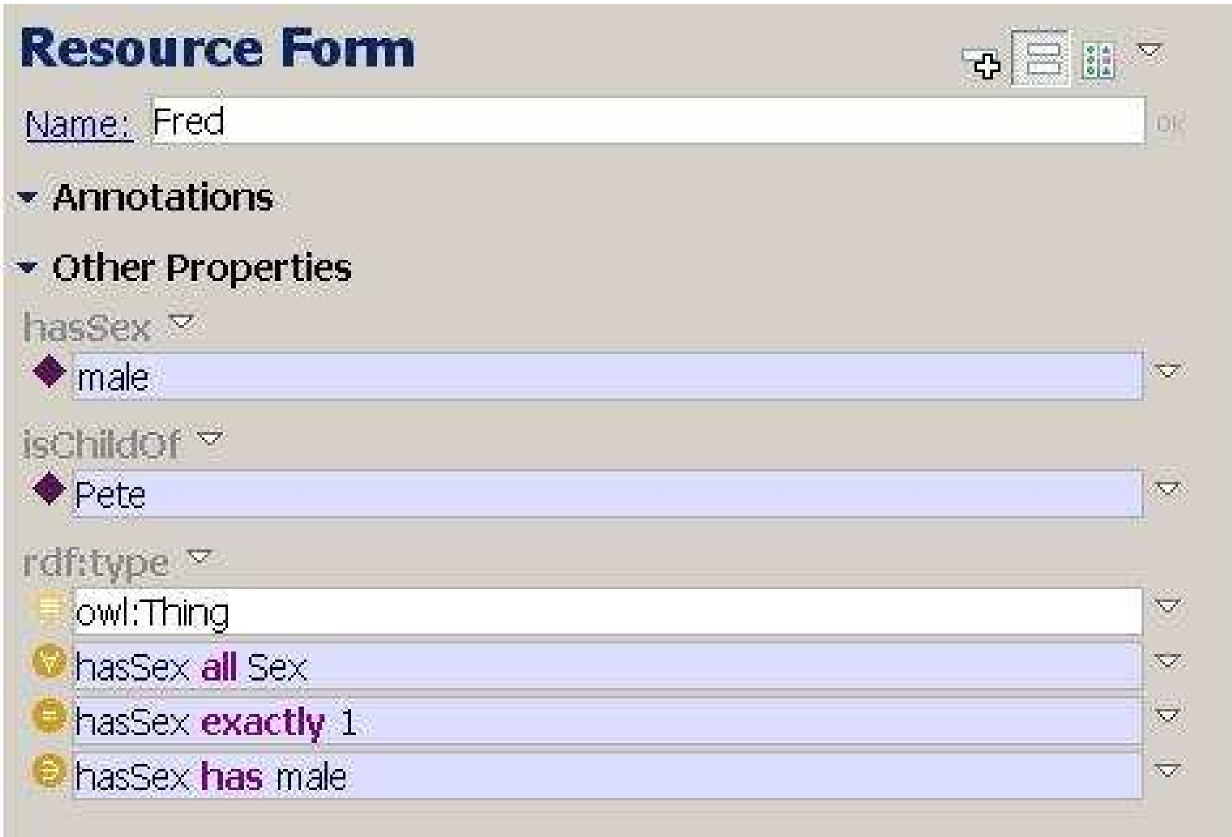


Figure 4: Topbraid - Fred's Resource Description after Pellet inference, showing that he's still just a `Thing`.

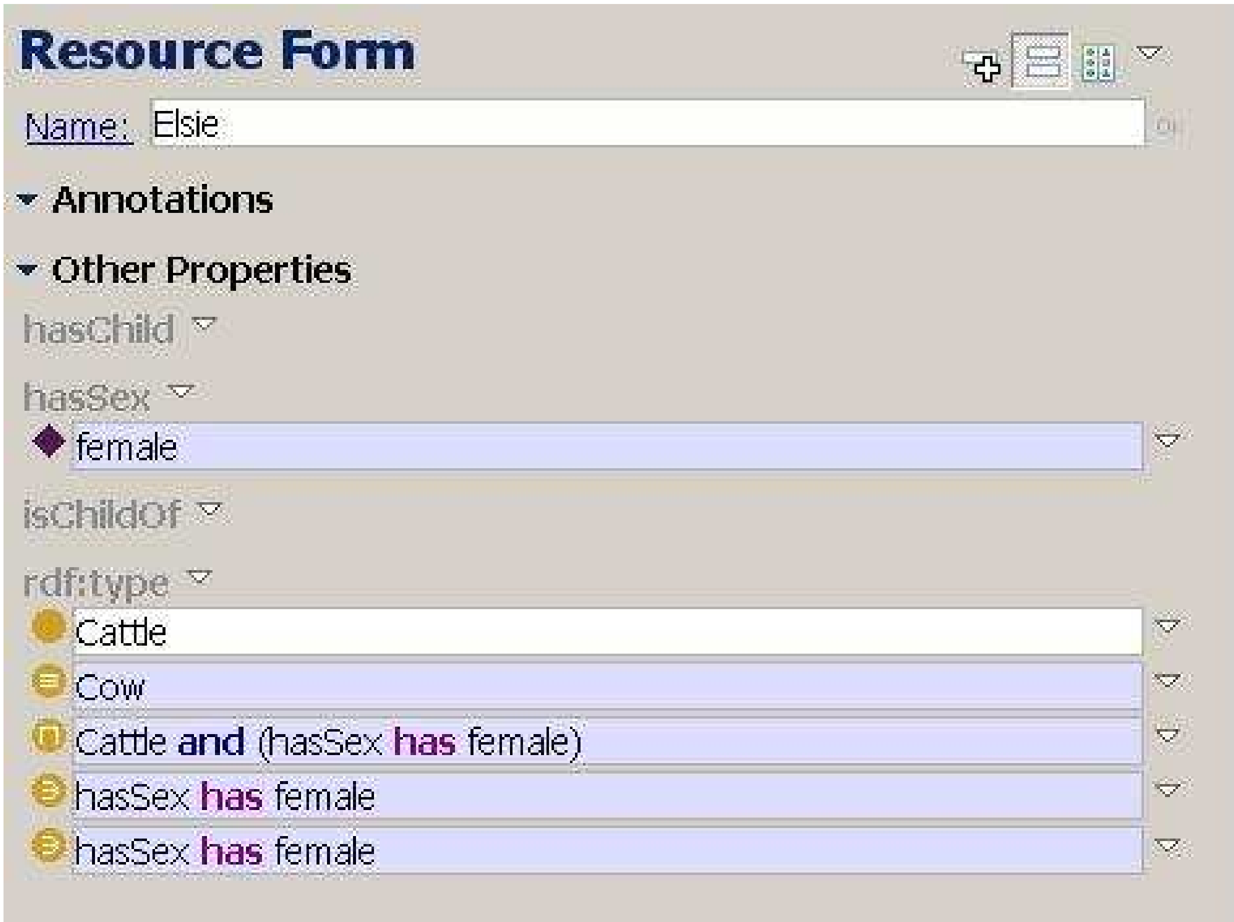


Figure 5: Topbraid - Elsie's Resource Description after Pellet inference, showing that she's a `Cow`.

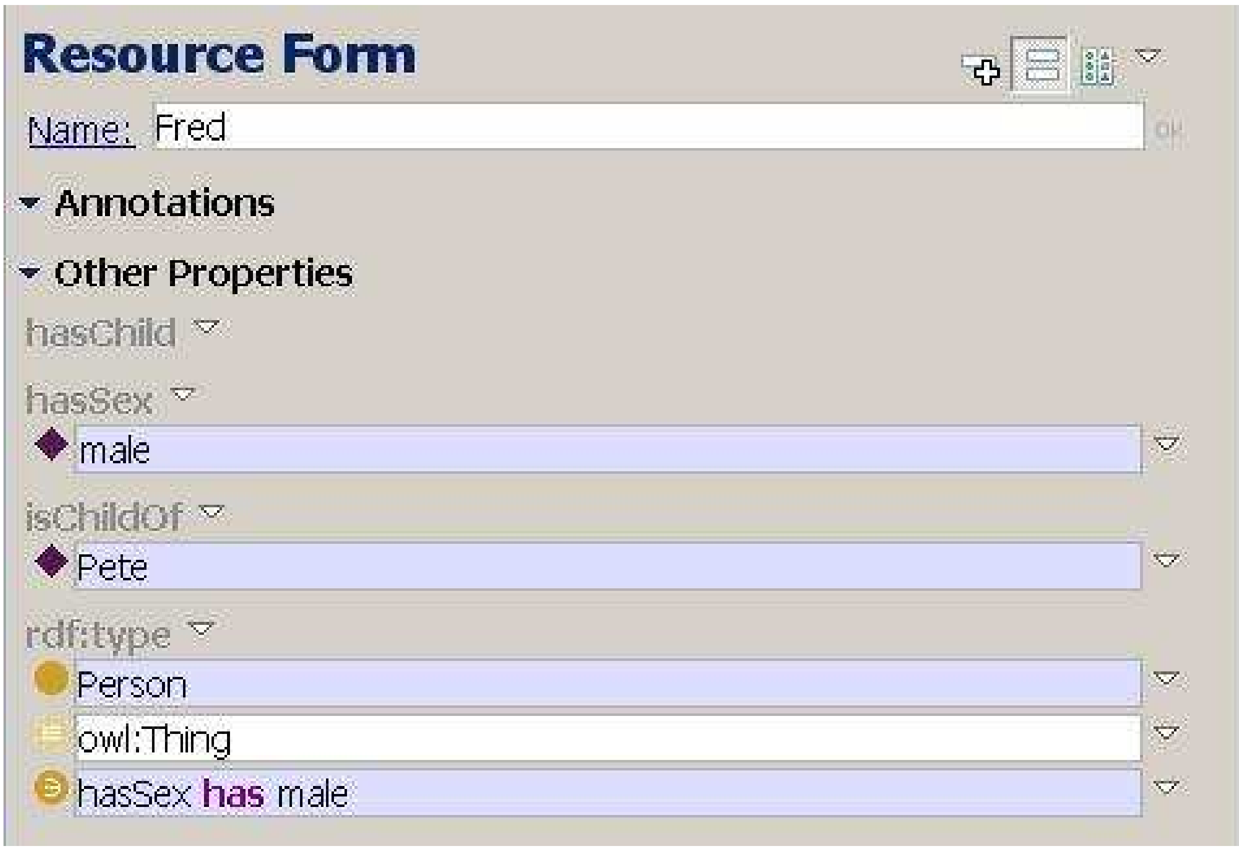


Figure 6: Topbraid - Fred's Resource Description after a second Pellet inference infers that he's a `Person`.

Figure 7: Topbraid - Sally's Resource Description after Pellet infers that she's a parent.

3.2 ParentSally in SNePS

To represent the structure of a classification hierarchy in SNePSLOG we use the two terms:

- `Isa(x,y)` - The proposition that `x` is a member of class `y`
- `Ako(x,y)` - The proposition that `x` is a subclass of `y`

To reason about the hierarchy, the following path-based rules (Shapiro 1991; Shapiro & The SNePS Implementation Group 2008) are used:

```
(a) define-path superclass
      (or superclass
          (compose ! superclass (kstar (compose subclass- ! superclass)))
          (domain-restrict ((compose arg- ! max) 0)
                           (compose superclass
                                    (kstar (compose superclass- ! subclass)))))

(b) define-path subclass
      (or subclass
          (compose ! subclass (kstar (compose superclass- ! subclass)))
          (domain-restrict ((compose arg- ! max) 0)
                           (compose subclass
                                    (kstar (compose subclass- ! superclass)))))

(c) define-path class
      (or class
          (compose ! class (kstar (compose subclass- ! superclass)))
          (domain-restrict ((compose arg- ! max) 0)
                           (compose class
                                    (kstar (compose superclass- ! subclass)))))
```

The details of SNePS' path-based inference are not important. The above essentially establishes when the system can create a new path between nodes in the SNePS network, thus, creating a believed proposition. Path-based rule (a) allows the system to reason that if some class *c1* has *c2* as a superclass, and *c2* has *c3* as a superclass, then *c1* has *c3* as a superclass. Path-based rule (b) allows the system to reason that if some class *c1* has *c2* as a subclass, and *c2* has *c3* as a subclass, then *c1* has *c3* as a subclass. In other words, (a) and (b) establish *Ako* transitivity. Path-based rule (c) allows the system to reason that some instance *i* of class *c1* is a member of *c2*, if *c1* is a subclass of *c2*. In other words, if *Isa* holds between an instance and its class, it also holds between that instance and that class' superclasses. What is significant about SNePS' path-based reasoning is that it is more efficient that using the equivalent rules

```
all(c1,c2,c3)({Ako(c1,c2), Ako(c2,c3)} => Ako(c1,c3)).
all(c1,c2,c3)({~Ako(c1,c2), Ako(c3,c2)} => ~Ako(c1,c3)).
all(i,c1,c2)({Isa(i,c1), Ako(c1,c2)} => Isa(i,c2)).
all(i,c1,c2)({~Isa(i,c1), Ako(c2,c1)} => ~Isa(i,c2)).
```

The above is a general SNePS axiomatization for reasoning about class hierarchies. Terms used for the ParentSally example are:

- `Thing` - the class of all things
- `Sex` - the class of sexes
- `Animal` - the class of animals

- `Person` - the class of people
- `Dog` - the class of dogs
- `Cattle` - the class of cattle
- `Man` - the class of men
- `Woman` - the class of women
- `Parent` - the class of parents
- `Cow` - the class of cows

- `male` - the individual male
- `female` - the individual female
- `Fred` - the individual Fred
- `Elsie` - the individual Elsie
- `Lucy` - the individual Lucy
- `Pete` - the individual Pete
- `Sally` - the individual Sally
- `Tom` - the individual Tom

- `childOf(x)` - A child of `x`

- `hasSex(x,y)` - the proposition that `x` has the sex `y`
- `hasChild(x,y)` - the proposition that `x` has the child `y`

The class hierarchy is establised by making the assertions:

```
Ako({Sex, Animal}, Thing).
Ako({Person,Dog,Cattle}, Animal).
Ako({Man,Woman,Parent}, Person).
Ako(Cow, Cattle).

Isa(Fred, Thing).
Isa(male,Sex).
Isa(female, Sex).
Isa(Elsie, Cattle).
Isa({Lucy, Pete, Sally, Tom}, Person).
```

The ParentSally example requires various constraints:

- The classes of *Man* and *Woman* are disjoint.
  `all(x)(andor(0,1)Isa(x,Man),Isa(x,Woman)).`

- Every animal has exactly one sex.[2]
  `all(x)(Isa(x,Animal) => nexists(1,1,2)(s)(Isa(s,Sex):  hasSex(x,s))).`

- Every man is male.
  `all(x)(Isa(x,Man) => hasSex(x,male)).`

- Every woman is female.
  `all(x)(Isa(x,Woman) => hasSex(x,female)).`

- Every cow is a female cattle.
  `all(x)(Isa(x,Cow) => {Isa(x,Cattle), hasSex(x,female)}).`

- Every female cattle is a cow.
  `all(x)(Isa(x,Cattle) => (hasSex(x,female) => Isa(x,Cow))).`

- Every parent is a person who has a child.[3]
  `all(x)(Isa(x,Parent) => {Isa(x,Person), hasChild(x,childOf(x))}).`

We enter the assertions that Pete is a parent whose child is Fred,

```
Isa(Pete,Parent).
hasChild(Pete,Fred).
```

and that Fred is male and Elsie is female.

```
hasSex(Fred,male).
hasSex(Elsie,female).
```

Then we ask SNePS for the classes that Fred and Elsie are instances of.

```
: Isa(Fred,?x)?
  wff5!:  Isa(Fred,Thing)

: Isa(Elsie,?x)?
  wff45!:  Isa(Elsie,Cow)
  wff38!:  Isa(Elsie,Thing)
  wff37!:  Isa(Elsie,Animal)
  wff8!:  Isa(Elsie,Cattle)
```

As in the Topbraid/Pellet run, the system still has Fred as just a thing, but infers that Elsie is a cow. So, again as we did in the Topbraid/Pellet run, we assert that

- Every child of a parent is a person.
  `all(x)(Isa(x,Parent) => all(y)(hasChild(x,y) => Isa(y,Person))).`

- Every person all of whose children are people is a parent.

```
all(x)(Isa(x,Person)
       => (all(y)(hasChild(x,y) => Isa(y,Person))
           => Isa(x,Parent))).
```

and again ask for the classes that Fred is an instance of:

---

[2] If the minimal parameter of `nexists` is given, the total parameter must also be supplied. This rule actually says, "*Every animal has exactly one of the two sexes.*"

[3] The current SNePS does not have existential quantifiers, so the `childOf(x)` is used as a Skolem function.

```
: Isa(Fred,?x)?
  wff35!:  Isa(Fred,Person)
  wff25!:  Isa(Fred,Animal)
  wff5!:  Isa(Fred,Thing)
```

As in the Topbraid/Pellet run, this is now successful.

The culmination of the ParentSally example is to assert that

- Lucy is Sally's child.
  `hasChild(Sally,Lucy).`

and ask if Sally is a parent:

```
: Isa(Sally,Parent)?
```

The lack of response indicates that SNePS can neither conclude that `Isa(Sally,Parent)` nor `~Isa(Sally,Parent)`, even though it has that both Sally and Lucy are people:

```
: Isa(Sally,Person)?
  wff60!:  Isa(Sally,Person)

: Isa(Lucy,Person)?
  wff62!:  Isa(Lucy,Person)
```

SNePS does not conclude that Sally is a parent, even though one might expect it to be able to from the rule

```
all(x)(Isa(x,Person)
       => (all(y)(hasChild(x,y) => Isa(y,Person))
           => Isa(x,Parent))).
```

There are two reasons for the absence of this inference.

1. SNePS currently does not have implemented an introduction rule that would allow it to infer universally quantified implications like

   ```
   all(y)(hasChild(Sally,y) => Isa(y,Person))
   ```

2. Even if SNePS had this rule, it does not follow logically from the current knowledge base that an arbitrary child of Sally is a person. This reason is similar to the reason that Pellet couldn't infer that Sally was a person until we said that Sally had at most one child.

To solve this problem, we employ a form of limited closed world assumption using SNeRE, the SNePS acting system (Shapiro & The SNePS Implementation Group 2008). The following assertion gives a plan for concluding that, for any person x who is known to have a child who is a person, if every child they are known to have is a person, then x is a parent.

```
all(x,y)({Isa(x,Person), hasChild(x,y), Isa(x,Person)}
   => ActPlan(parentIfAllKnownChildrenArePeople(x),
              snsequence4(believe(Maybe(Isa(x,Parent))),
                          withall(y, hasChild(x,y),
                              snif({if(Isa(y,Person), noop()),
                                    else(disbelieve(Maybe(Isa(x,Parent))))}),
                              noop()),
                          snif({if(Maybe(Isa(x,Parent)),
                                   believe(Isa(x,Parent))),
                                else(believe(~Isa(x,Parent)))}),
                          disbelieve(Maybe(Isa(x,Parent)))))).
```

This plan for a person `x` with a known child who is a person is:

```
1. Believe, as a temporary default, that x may be a parent;
2. With every y such that y is a child of x
         if y is a person do nothing
         else disbelieve that x may be a parent
   but if x has no children, do nothing (but this cannot be)
3. If it is still believed that x may be a parent
         believe that x is a parent
   else believe that x is not a parent
4. Disbelieve that x may be a parent.
```

The reason for the temporary default belief that `Maybe(Isa(x,Parent))`[4] instead of a temporary belief that `Isa(x,Parent)` is that once the system believed that `Isa(x,Parent)`, it would infer that all `x`'s children were people, defeating the check that all `x`'s children are already known to be people.

Since we want to say that Sally is a parent if all her known children are people, we perform this act on Sally:

```
: perform parentIfAllKnownChildrenArePeople(Sally).
```

and then again ask if Sally is a parent:

```
: Isa(Sally,Parent)?
  wff64!:  Isa(Sally,Parent)
```

Sally is now believed to be a parent.

To make sure that the system is not overgeneralizing, let us introduce Ted, a person with two children, only one of whom is known to be a person:

```
Isa(Ted,Person).
: hasChild(Ted,{Betty,Jean}).
: Isa(Betty,Person).
```

and see if Ted is inferred to be a person:

```
: perform parentIfAllKnownChildrenArePeople(Ted).
: Isa(Ted,Parent)?
  wff129!:  ~Isa(Ted,Parent)
```

The system infers that Ted is not a parent.

3.3 Comparison

Topbraid/Pellet and SNePS are both able to represent the ParentSally Example, and to perform the required reasoning. The techniques, of course, are different. The most noticeable difference is in the way they handle the inference that Sally is a parent. Topbraid/Pellet essentially does the following.

1. Sally is a person with a child, Lucy, who is a person.
2. Sally is an instance of `PersonWithAtMost1Child`.
3. All instances of `PersonWithAtMost1Child` have at most 1 child.
4. Therefore Lucy is Sally's only child.
5. Therefore all Sally's children are people.

[4] `Maybe(Isa(x,Parent))` is a legal proposition because all well-formed expressions in SNePSLOG are terms; some of them, such as this one, being proposition-valued terms (Shapiro 1993; Shapiro *et al.* 2007)

6. Therefore Sally is a parent.

The crucial step is (4), which is a version of circumscription (McCarthy 1980), using, specifically, a number restriction on the role, `hasChild`. Number restrictions, and reasoning according to them, are among the earliest features of Description Logics, and "are sometimes viewed as a distinguishing feature of Description Logics" (Nardi & Brachman , p. 9).

SNePS inferred that Sally is a parent by following this line of reasoning:

1. Sally is a person with a child, Lucy, who is a person.
2. So maybe Sally is a person.
3. Consider all Sally's children.
   (a) Lucy is a child of Sally's, and she is a person, so Sally still may be a person.
   (b) That's all the children of Sally that I know of.
4. Sally still may be a parent.
5. So Sally is a parent.

SNePS uses introspective acting to: consider all the children of Sally it knows; disbelieve that Sally may be a parent if one of them is not a person; believe that Sally is a parent if it still believes that she may be one after considering all her children. Unlike Topbraid/Pellet, SNePS never concludes that Lucy is Sally's only child.

Both Topbraid/Pellet and SNePS use a form of closed-world reasoning restricted to the set of Sally's children. If Topbraid/Pellet later learned that Sally had another child, that would contradict the conclusion that Sally is an instance of `PersonWithAtMost1Child`. If SNePS later learned that Sally had another child, say Dave, who was not known to be a person, it would conclude that Dave is a person, because Sally is now believed to be a person. However, if we had SNePS first `disbelieve` that Sally is a parent, when we subsequently had it perform `parentIfAllKnownChildrenArePeople(Sally)` again, it would then believe that `~Isa(Sally,Parent)`.

4 Conclusions

Topbraid/Pellet and SNePS are both knowledge representation and reasoning systems. Topbraid/Pellet uses Description Logic. SNePS uses higher-order predicate logic and an integrated acting system.

The Jobs Puzzle was originally described in (Wos *et al.* 1984) to illustrate a resolution theorem prover, but we have used it for many years to illustrate SNePS reasoning, especially the use of `andor` and `nexists`, which are particularly appropriate for the Jobs Puzzle. For example, using `nexists`, SNePS reasons that, since neither Steve, Thelma, nor Roberta is the actor, Pete must be the actor. Since Topbraid/Pellet seems incapable of this style of reasoning, we could not get it to reason directly to a solution of this puzzle.

The ParentSally example was specifically designed (by one of the authors) to illustrate Description Logic reasoning for a knowledge representation class. Topbraid/Pellet uses a number restriction on `hasChild` to reason that Sally is a parent. Reasoning with number restrictions is a fundamental feature of Description Logics. SNePS was also able to conclude that Sally is a parent, but by using its acting component to examine each currently known child of Sally, checking that they are people.

References

Clark & Parsia, LLC. 2007. Pellet: The Open Source OWL DL Reasoner. `http://pellet.owldl.com/`.

McCarthy, J. 1980. Circumscription: A form of non-monotonic reasoning. *Artificial Intelligence* 13(1–2):27–39.

Nardi, D., and Brachman, R. J. An introduction to description logics. 1–43.

Shapiro, S. C. SNePS: A logic for natural language understanding and commonsense reasoning. 175–195.

Shapiro, S. C., and The SNePS Implementation Group. 2008. *SNePS 2.7 User's Manual*. Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY. Available as `http://www.cse.buffalo.edu/sneps/Manuals/manual27.pdf`.

Shapiro, S. C.; Rapaport, W. J.; Kandefer, M.; Johnson, F. L.; and Goldfain, A. 2007. Metacognition in SNePS. *AI Magazine* 28:17–31.

Shapiro, S. C. 1979. Numerical quantifiers and their use in reasoning with negative information. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann. 791–796.

Shapiro, S. C. 1991. Cables, paths and "subconscious" reasoning in propositional semantic networks. In Sowa, J., ed., *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Los Altos, CA: Morgan Kaufmann. 137–156.

Shapiro, S. C. 1993. Belief spaces as sets of propositions. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)* 5(2&3):225–235.

Top Quadrant Inc. 2007. Topbraid Composer. `http://www.topbraidcomposer.com/`.

W3C. 2004a. OWL Web Ontology Language Overview. `http://www.w3.org/TR/2004/REC-owl-features-20040210/`.

W3C. 2004b. SWRL: A semantic web rule language. `http://www.w3.org/Submission/SWRL/`.

Wos, L.; Overbeek, R.; Lusk, E.; and Boyle, J. 1984. *Automated Reasoning: Introduction and Applications*. Englewood Cliffs, NJ: Prentice-Hall.