Cassie Reads Sci-Fi

Determining the Meaning of The Fictional Noun *maker* From Context

Jonathan Bona
CSE663: Advanced Knowledge Representation

December 14, 2003

Abstract

The Contextual Vocabulary Project is directed by William J. Rapaport and Michael W. Kibby at the State University of New York at Buffalo. The project is using SNePS to develop algorithms which can infer the meaning of an unknown word based on the context in which it is used. SNePS is a propositional semantic network-based Knowledge Representation and Reasoning System. This paper describes related work I have done as a project for the course *CSE663: Advanced Topics in Knowledge Representation and Reasoning*. My work includes selecting and representing in SNePS a passage containing an unknown noun, representing appropriate background information about other terms and ideas in the passage, and adapting those representations to improve the performance of the CVA noun definition algorithm. I also suggest some candidate areas for possible future work on my project, including short-term goals such as bug fixes as well as some more complicated long-term improvements.

1 Introduction

The phrase "Contextual Vocabulary Acquisition" (CVA) refers to one method by which cognitive agents can learn the meaning of new words or phrases. That is, when confronted by a word for which the agent has no stored definition (or possibly just an incomplete or incorrect definition), the agent derives a meaning for the word based on the context in which the word is used. This phenomenon has been widely observed in humans but little is

understood about the specific details of *how* - precisely which steps to take - to extract the meaning of a word from context. The CVA Project, directed by William J. Rapaport and Michael W. Kibby, is developing a computational theory of this process. The algorithms developed will become part of a reading curriculum to teach students how to effectively perform Contextual Vocabulary Acquisition.

The project's computational research makes use of the SNePS Knowledge Representation and Reasoning System to model a cognitive agent name Cassie. SNePS is a propositional, semantic network-based Knowledge Representation and Reasoning system. It is developed by Stuart C. Shapiro and the SNePS Research Group (SNeRG) at the University at Buffalo. SNePS represents and reasons about information as a semantic network of nodes and directed arcs. Nodes in SNePS represent concepts, while arcs represent relations between concepts. Base nodes, which represent individuals, are nodes with no arcs going out of them: the only information the system has about a base node is what is asserted about it. For example, in this paper I will be presenting a SNePS network that includes an individual base node which represents a human being named Paul. The base node that represents Paul has a number of propositions *about* it, declaring that Paul has different properties and characteristics, participates in relations with other nodes and so on. However, the base node representing the individual Paul cannot be said to be a proposition *about* anything else, and therefore has no arcs coming out of it. Propositions in SNePS can be either *asserted* or not *asserted*. Asserted propositions are things that Cassie believes to be the case. Non-asserted nodes represent propositions that Cassie does not specifically believe are true. For instance, if I tell Cassie the proposition "If proposition1 then proposition2", both proposition1 and

proposition2 will be *built* (i.e. represented) but neither of them will necessarily be asserted unless we tell Cassie to assert them. Nodes which were not explicitly asserted by the user can be asserted by Cassie as the result of logical inference.

Cassie reads narratives - often natural language passages converted to *Snepsul*(the SNePS User Language), which she understands better than ambiguous English. After reading part or all of a passage (and when prompted by the user), Cassie will attempt to give a definition for the target word. Cassie's definition combines her background knowledge about the world (the world being either the "real world", or the fictional world of the passage, or both), and contextual information about the unknown word contained in the passage.

The CVA project currently has working algorithms for extracting the meaning of nouns and verbs from context in a passage. Ongoing research continues to improve the scope and accuracy of the algorithms. One component of the research involves selecting passages which contain *difficult* or *unknown* words (unknown to Cassie, not necessarily to the human researcher), creating a knowledge base of background knowledge appropriate to the passage, and having Cassie read the passage. She is then asked to give a definition for the target word using the corresponding word definition algorithm (e.g. *defineNoun* for nouns). If the results are unsatisfactory changes may be made the the background knowledge, the representation of the passage, or the algorithm itself.

This paper describes work I have done to fulfill the project component of CSE663: "Advanced Topics In Knowledge Representation And Reasoning". In it, I discuss my results as well as some ways in which my efforts can be expanded/improved upon in the future. My work has consisted of selecting a target word and a passage containing the word, translating

the passage and appropriate background knowledge from English into Snepsul (the SNePS User Language), and experimenting with the noun definition algorithm's ability to handle my choice of representations and produce "correct" output. I have not made any modifications to the algorithm itself.

2 The Word and Passage

The target word and passage I selected for this project come from Frank Herbert's epic science fiction novel, *Dune*. The word is the noun "maker", as used in the following passage.

> *"With the whip like hook-staffs Paul knew he could mount the* maker*'s high curving back. For as long as a forward edge of the worm's ring segment was held open by a hook open to admit abrasive sand into the more sensitive interior the creature would not retreat beneath the desert." (Herbert 1965)*

3 Human Protocols

3.1 Introduction

As part of the preparation for representing the passage and background knowledge, I conducted think-aloud sessions with human subjects. Subjects were instructed to read the passage and come up with a definition for the word *maker*, and encouraged to speak aloud as they read and thought about the passage. The point of these experiments was to see what kind of information a person uses to reason about this particular word in the context of this particular passage. The rest of this section describes two of these think-aloud sessions. Rather than providing a word for word transcript of their attempts, I have summarized the ideas they mentioned, in the order they were mentioned.

3.2 Subject 1

- A *maker* is some kind of a worm.
- It is some kind of a desert worm.
- It is very large, because Paul is going to mount it - unless Paul is very small.
- It has segments which can be held open by a hook.
- Final definition: a *maker* is a segmented worm that inhabits a desert environment. It is large in comparison to humans. The segments are separable.

3.3 Subject 2

- A *maker* is some sort of large being that is rideable.
- In order to control it, you must use dangerous items.
- It might be similar to a dinosaur.
- It's a giant worm. It's violent because you need to hurt it to make it behave.
- It lives in the desert.
- Final definition: a *maker* is a large, desert-dwelling worm. Violence is required to control it.

3.4 Comments

Both of the subjects noted that the maker is a large worm, and the first explicitly stated one of the things I was expecting/hoping for, that the worm must be large because paul is riding it (and presumably, people don't ride things smaller than themselves). The first subject even uses the rule of circumscription - concluding that the maker is *large* since it must be larger than a person, but noting that this is only the case (that the maker is large) if the person is not abnormal (i.e. if the person is not very small).

They also both conclude that it lives in the a desert environment, based on the statement about it retreating beneath the desert. The underlying assumption to this conclusion seems to be something like, *if some animal is mentioned in a scenario that takes place in a specific type of environment, then presumably that is where the animal lives, barring any other information about where the animal lives.*

4 Passage Representation

4.1 Introduction

Cassie recognizes only a subset of English grammar. Therefore, it is necessary to interpret all but the simplest passages for her into some more recognizable form, which can be represented in the SNePS system (Cassie's brain). One approach is to reduce the passage to a simpler English-language version of itself, and let the SNePS Natural Language Processing System (SNALPS) convert that into SNePS's internal representation (c.f. Ehrlich, 1995). Another option is for the human knowledge engineer to translate the passage directly into Snepsul (c.f. Ahmed, 2003), and give that to Cassie as input for her to "read".

4.2 Reducing The Passage to a Simpler Form

One thing that quickly became apparent as I attempted to represent the passage in Snepsul was that it is a rather oddly worded, difficult passage. The first major issue is the fact that the second of the two sentences is a rather oddly-phrased run on sentence. Another major issue is that the same entity is referred to by three different names ("maker","worm","creature") in the passage.

As a first step I reduced the passage to a simpler series of short sentences. These were

then converted to Snepsul. For simplicity's sake I ignore the issue of tense, converting the narrative of the passage into the present tense. Thus, "Paul knew" becomes "Paul knows".

The first sentence was broken down into the following three short sentences:

- Paul knows that he is able to mount the maker's back using the hook-staffs.

- The hook-staffs are whiplike.

- The maker's back is high and curved.

These sentences seem to preserve the meaning of the original, with one possible exception: the original passage states that the maker's back is curv*ing*, not just that it is curv*ed*. The original could be interpreted as stating that the back is making some kind of curving motion, while the simplified version almost has to be interpreted as stating that the back has the property "curved", or that the shape of the back is "curved". Of course, both of these would be true if the back is involved in a curving motion, but some of the information is lost in the transition.

My general strategy for dealing with small discrepancies like this has been to go with the simplest representation as long as I believe that the overall meaning of the passage and therefore the definition of the entity in question will be preserved. For example, if the noun algorithm tells me that the maker's back has the property curved rather than that the maker's back has the ability to perform a curving action, that part of the definition will be sufficient- unless we're very much more concerned about what the maker might be doing than about what properties it has.

I reduced the second sentence to the following:

- An effect of the forward edge of the worm's ring segment being held open by the hook

is that sand can be admitted into the interior of the worm.

- The interior of worm is sensitive.

- An effect of the forward edge of the worm's ring segment being held open by the hook is that the creature will not retreat beneath the desert.

Note that the actual meaning of the sentence is slightly more complicated than this reduced version. Specifically, the sentence states that *for as long as (something1) is the case, (something2) is not the case*, where something1 is "a forward edge of the worm's ring segment is held open by a hook (and since it is open, sand can be admitted)" and something2 is "the creature retreats beneath the desert".

Clearly, the simplified "cause and effect" version of this sentence is losing some of the original meaning. There's no notion of temporal *duration*; if we had to ask questions about what happens when the forward edge is no longer being held open after having been held open for some period of time, this representation would fail to accurately portray that.

With that said, using simple cause and effect seems to capture plenty of the meaning in order for us to be able to characterize the entity whose definition is in question. The effect of the forward edge being held open *is* that the sand can be admitted and the creature will not retreat.

For further discussion of this issue, including possible solutions that would allow a more accurate representation, please see section 4, "Snepsul Representation of the Passage" below.

Another issue that needs to be overcome is the fact that, while we're trying to build a representation which will allow Cassie to infer a definition for "maker", the word "maker" only occurs in the first sentence. In the second sentence, the entity called a "maker" is called

both a "worm" and a "creature". It is fairly clear to a human reader that these three terms refer to the same entity (see section 3, "Human Protocols" for examples). For the human who is able to determine that the three nouns refer to the same thing, the fact that the entity is called a "worm" may be a large help in figuring out what kind of thing a "maker" is (assuming that "worm" is a known word). Cassie however, will not automatically be able to determine that these refer to the same thing.

My strategy for resolving that "maker" and "worm" refer to the same entity is outlined in a separate section below. However, my solution for resolving "worm" with "creature" is to simply replace the term "creature" with the term "worm" before the passage is presented to Cassie. The use of both of those terms has more to do with Frank Herbert's writing style than it has to do with the actual meaning of the sentence. Including "creature" would not really add any information to the passage considering that the class of things that are worms is a subset of the class of creatures - a piece of information that the typical reader is like to have in their background knowledge.

We're now ready to take a look at the actual Snepsul representation of the passage as it was given to Cassie.

4.3 Snepsul Representation of the Passage

4.3.1 Introduction

In converting the passage to Snepsul, I attempted to use Standard CVA case frames [1] or Standard SNePS case frames from the Case Frame Dictionary [2] in that order. In the few cases

[1] http://www.cse.buffalo.edu/ rapaport/CVA/cvaresources.html CVA Resources
[2] http://www.cse.buffalo.edu/sneps/Manuals/dictionary.pdf A Dictionary of SNePS Case Frames

where neither was possible, I was forced to create my own case frames. To conserve room in the paper itself, definitions for new case frames introduced here are given in Appendix A at the end of the paper. An informal syntax and semantics for new case frames will be given in the paper itself when necessary, and when *pretend-it's-english* semantics will not suffice.

Note that in some cases, further changes are made to the already reduced English version of the passage, to facilitate a line-by-line correspondence with the Snepsul code.

Because this paper is not intended to serve as a reference or tutorial on SNePS or Snepsul, I do not attempt to explain in any detail the Snepsul language specifics for code that appears in this section. Please refer to (Shapiro et al., 2003) for a comprehensive description of SNePS.

4.3.2 Snepsul Code With Commentary

"With the whip like hook-staffs Paul knew he could mount the maker's high curving back."

```
;; something in the story is named "Paul"
(add object #paul proper-name "paul")
```

This code instantiates a new base node for the individual named "Paul" using the "#" operator, and adds to the knowledge base the proposition that the individual is an object with the proper name "paul". Any further references to that specific base node can be made using the "*" operator, i.e. *paul.

```
;;something in the story ("the maker") is a maker
(add member #themaker class (build lex "maker"))

;; something in the story is a hookstaff
```

```
(add member #thehook
      class (build lex "hook staff"))

;; something is the back of the maker
(add object #theback
     rel (build lex "back")
     possessor *themaker)

;; the hook staff(s) have the property "whip like"
(add object *thehook
     property (build lex "whiplike"))

;; the maker's back is high
(add object *theback
     property (build lex "high"))

;; the maker's back is curved
(add object *theback
     property (build lex "curved"))
```

The preceding assertions all use standard case frames with their usual meanings. In the next, my representation uses a combination of new case frames, the agent/ability and the agent/act/action/object/instrument frame.

```
;; paul knows that he has the ability to mount the back of the maker,
;; using the hook staff as an instrument
(add agent *paul
     act (build action (build lex "know")
                object (build agent *paul
                              ability (build action (build lex "mount")
                                                    object *theback
                                                    instrument *thehook))))
```

This code asserts that the base node we're referring to by "*paul" is an agent who performs the action of "know"ing, directed toward some object. The object that *paul knows is the proposition that some agent has the ability to perform the action of "mount"ing

directed toward the base node which represents "the back of the maker", using "the hook-staffs" as an instrument. It happens to be the case that the agent who has this ability is the same one performing the knowing - "paul" himself.

Note that I'm treating the hook staffs as a single object. Its obvious that there are at least two of these things (whatever they may be), so calling them one object might not be the absolute best choice here. I decided that the fact that they're two (or more) distinct objects is not essential to the meaning of the passage. Of course, I can always justify calling "the hook staffs" an object by saying that they are an object that consists of a pair (or collection) of other smaller objects.

This sentence about what paul knows is a *de se* belief report. That is, it reports something that the agent (Paul) believes (knows) about himself. If we were to ask Paul what it was that he knew, he would most likely not reply: "I knew that with the whip like hook-staffs *'he'* could mount the maker's high curving back." Rather, the indexical would shift and he would report: "I knew that with the whip like hook-staffs *'I'* could mount the maker's high curving back."(Actually, 'he' is a *quasi*-indexical since it occurs within the scope of an agent's belief report).(Rapaport et al. 1997:4).

For this reason, representing the passage as "Paul knows that *he...*" directly would make things more complicated. I have chosen to "dereference" the thing that the indexical 'he' refers to. Since Paul is represented by the first base node created, SNePS will call that node *b1*, Therefore, my representation of the sentence says that "*[[b1]]* knows that *[[b1]]* (that is, Paul himself) has the stated ability".

As it turns out in this case, the things Paul knows about are not going to be very

interesting or helpful in reasoning about the passage unless we have some kind of rule stating that, *If someone knows something, then that thing is the case.* This is generally true. What is not generally true is that *If someone says that they know something, then that thing is the case*, because people frequently confuse their own (unjustified) beliefs with knowledge. In using this passage however, we can safely make the assumption that the narrative is not confused, deceived or attempting to deceive and that in fact Paul knows what he is said to know.

See figures 1-4 for diagrams of this sentence in the passage. Note that diagrams of the passage are split up into several parts to make them fit well on these pages. Consequently, nodes representing some of the base nodes and some of the propositions are duplicated. In SNePS, the uniqueness principle ensures that any two nodes with the same name are actually the same node. If two nodes, both labeled "b1" (e.g.), appear in two different figures, they should be considered to be the same node *unless otherwise noted.*

"For as long as a forward edge of the worm's ring segment was held open by a hook open to admit abrasive sand into the more sensitive interior the creature would not retreat beneath the desert."

```
;; something (``the worm'') is a worm
(add member #worm class (build lex "worm"))

;; something is the worm's ring segment
(add possessor *worm
     object #ring-segment
     rel (build lex "ring segment"))

;; there's some forward edge of the ring segment
(add possessor *ring-segment
     object #forward-edge
     rel (build lex "forward edge"))
```

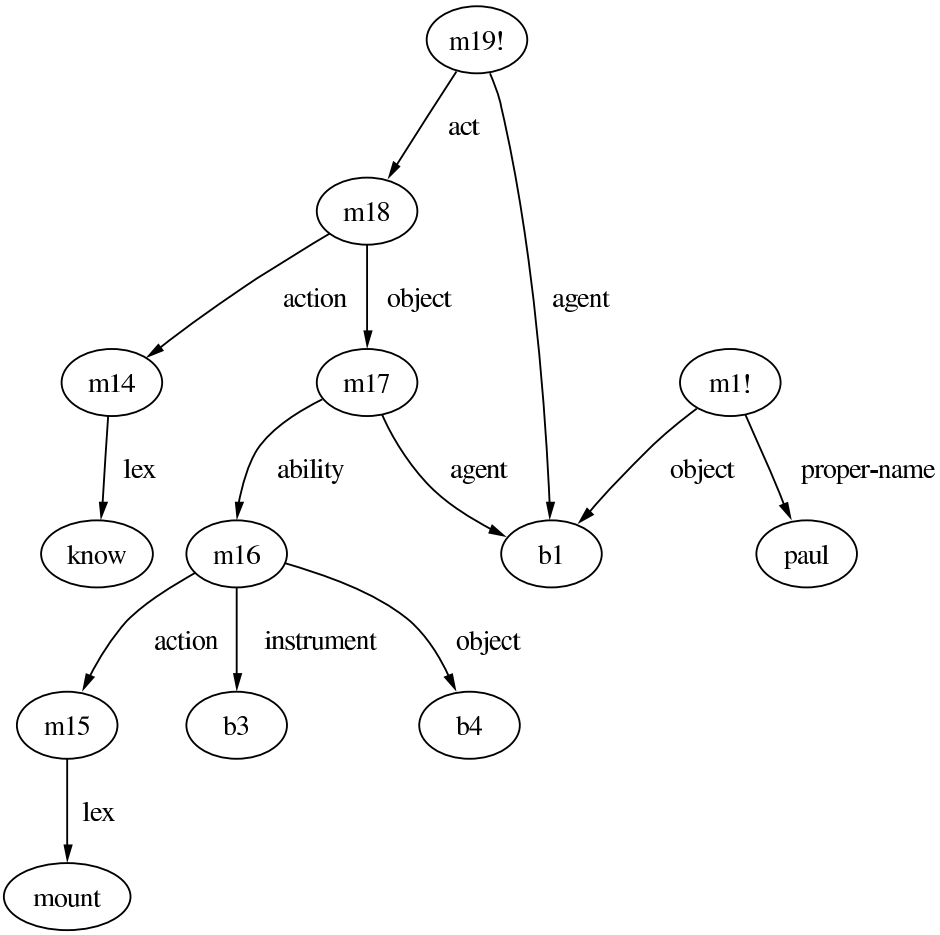

Figure 1: "b1 (Paul) knows that b1 has the ability to mount b4 using the instrument b3"

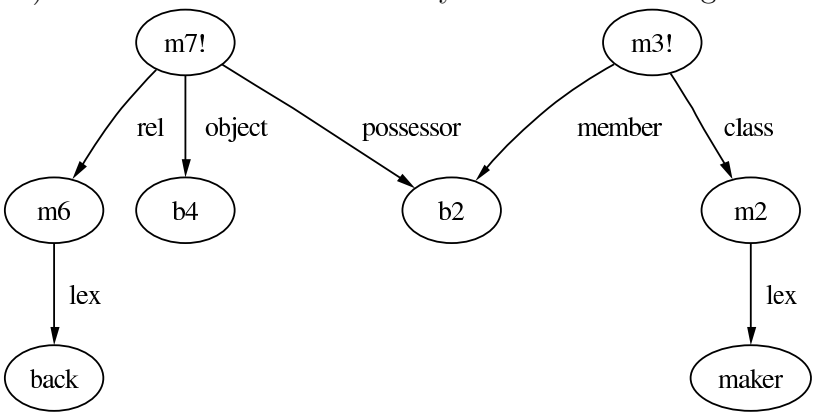

Figure 2: "b2 is a member of the class *maker* and b4 is the back of b2"

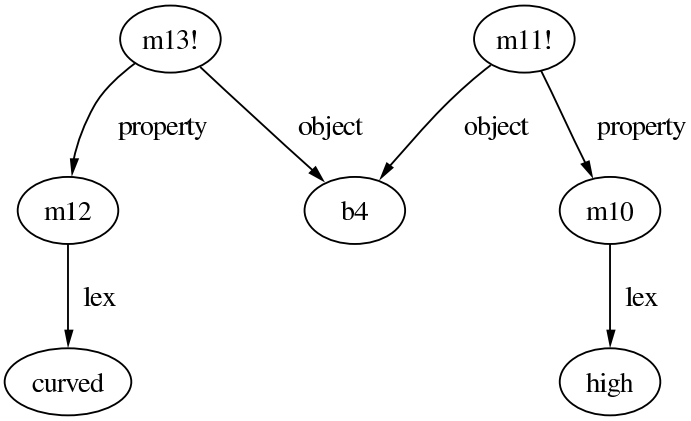

Figure 3: "b4 (the Maker's back) has the properties *high* and *curved*"

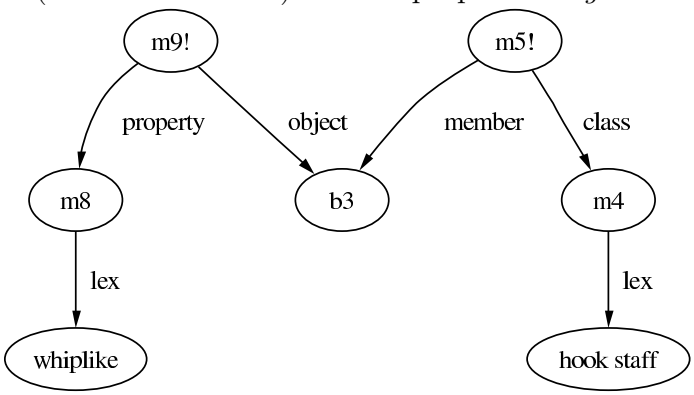

Figure 4: "b3 (the hook-staffs) is a member of the class of *hookstaffs* and has the property *whiplike* "

```
;; the interior of the worm is the worm's interior
(add possessor *worm
     rel (build lex "interior")
     object #sens-interior)

;; the interior of the worm is sensitive
(add object *sens-interior
     property (build lex "sensitive"))

;; some (a piece of/a group of) sand exists
(add member #sand
     class (build lex "sand"))

;; the sand is abrasive
(add object *sand
     property (build lex "abrasive"))


;; "the desert" is a desert
(add member #desert
     class (build lex "desert"))

;; the worm is a creature
(add member *worm
     class (build lex "creature"))


;; if the hook holds open the forward edge
;; then the forward edge admits the abrasive sand into the sensitive interior
;; and the creature(worm) will not retreat below the desert
(describe
(add forall $a
     ant (build agent *thehook
                act (build action (build lex "hold open")
                           object *forward-edge))
     cq (build agent *forward-edge
               act (build action (build lex "admit")
                         object (build object1 *sand
                                       rel (build lex "inside")
                                       object2 *sens-interior)))
     cq (build min 0
               max 0
               agent *worm
               act (build action (build lex "retreat beneath")
```

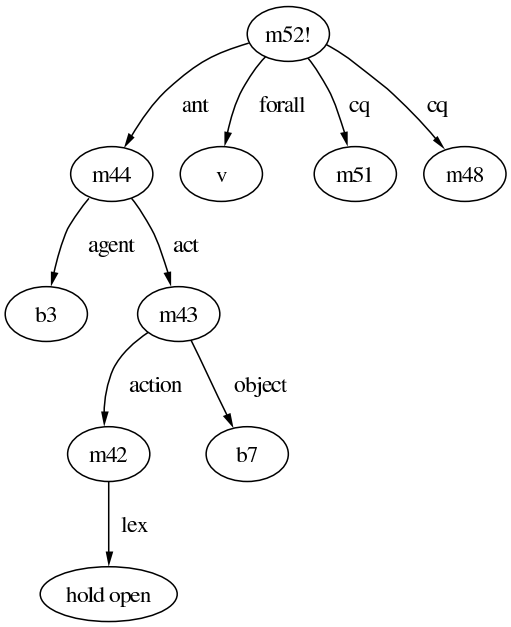

Figure 5: "if the hook (b3) holds open the forward edge (b7), there are two consequences, m44 and m48"

```
                    object *desert)))
)
```

As mentioned above, the sentence beginning with *For as long as* would probably be more accurately represented with a loop structure or with some notion of durations of time. One way of dealing with this if I didn't want to worry specifically about representing *time* might be to use a situation calculus.

Another way to more accurately represent the meaning without requiring an ontology of time would be to use one or more functions in the SNePS Rational Engine (SNeRE). SNeRE was developed to allow Cassie to perform actions based on her beliefs about the world, and to reason about actions. SNeRE provides functions such as *sniterate* and *withall*, which allow representation of conditional loops. In order to keep things simple enough for the noun algorithm to understand - it does not recognize SNeRE case frames - this path

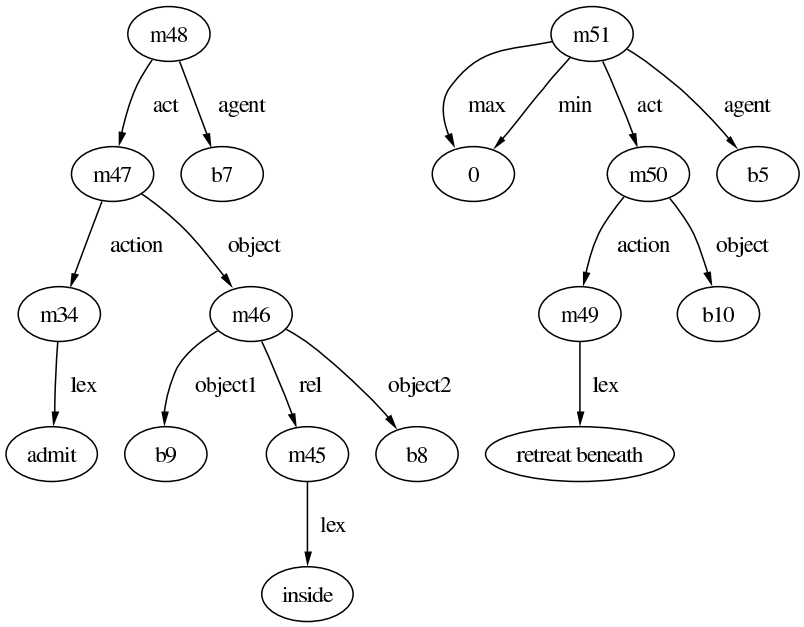

Figure 6: "m48: the forward edge (b7) admitting the sand b9 inside/into the sensitive interior b8 and m51: the worm (b5) not retreating beneath the desert (b10)"

was avoided. For more information about these options, refer to the SNePS User's Manual, Section 4 (Shapiro et al. 2002:25-43).

The other option discussed was to use the notion of cause and effect. The noun definition algorithm does not recognize the cause-effect case frame. Observe however, that if something is *the* cause of something else, then whenever the first happens to be the case, the second (the *effect*) is also the case. This does not hold for multiple causes in which more than one must be true for the effect to take place. Of course in this case, we might be able to treat the conjunction of the necessary causes as a single cause. In any case, the single cause here is the "being held open" of the forward edge. Therefore it seems appropriate given the discussion above to use the forall-ant-cq case frame to represent the sentence.

The final simplified interpretation of this part of the passage is: "If it is the case that

the hook holds open the forward edge, then it is the case that the forward edge admits the abrasive sand into the sensitive interior and it is the case that the worm does not retreat below the desert".

Figures 5 and 6 offer a picture of part of Cassies SNePS representation of the sentence. In the interest of saving space, I have not provided a diagram of the entire sentence. The rest of the Snepsul code for this part of the passage is relatively simple and easily interpreted (with the Case Frame Dictionary in hand as needed!).

5 Background Knowledge

5.1 Introduction

The amount of background knowledge in my knowledge base before Cassie reads the passage is (appropriately) quite a bit larger than the passage. Rather than presenting all the background Snepsul code and diagrams in this section, I have included all the annotated background knowledge code (as well as the passage code) in Appendix B. In this section, I list English versions of the background rules and present the details of some of the more important ones.

5.2 Background Goals

Picking an appropriate amount of background information for the passage is tricky. Clearly, there are many pieces of information which humans know that have nothing to do with the meaning of the passage. There are also probably many pieces of information that are related to concepts in the passage, but which will not find themselves into a definition for one of the words. For instance, we could make all kinds of statements about what kinds of things grow

in the desert, but those would probably not contribute to the meaning of the word "maker" in this passage.

The sorts of things the noun algorithm will infer about *maker*s are things like: what kind of properties *maker*'s have, what kinds of actions they can take, what classes of things they belong to, and so on.

My own definition for *maker* would include all sorts of information not included in the passage because I happen to have read the entire book in which the passage appears, and information about them is given elsewhere in the text. Generally, the type of definition I would like to see from this passage would include statements like "humans can ride makers", "makers can retreat beneath the surface of the desert", "makers have similar properties and possible actions as worms" - e.g. they are long and flexible - and so on. The background information is necessarily biased toward my (and my experimental subjects') understanding of the passage.

5.3 Background Rules and Facts

1. If an agent knows that they (themselves) have an ability to perform some action directed toward some object and using some instrument, then they do in fact have that ability.
2. If an agent mounts some thing, and the thing is the back of an object, then the agent mounts the object.
3. If anything mounts something else, then the first thing rides the second.
4. If something rides something else, the rider is on the ridee.
5. Something ridden is possibly an animal.
6. If something1 mounts something2 then something1 is larger than something2
7. If x is smaller than y then y is larger than x.

8. If x is larger than y, then y is smaller than x.
9. Sand is part of the desert.
10. Humans are animals.
11. Creatures are animals.
12. Worms are animals.
13. Worms are elongated and flexible.
14. Worms tunnel in the ground
15. If two classes are equivalent, then they are each subclasses of the other
16. If two things are equivalent and one of them is a member of some class, the other is also a member of that class
17. If two things are equivalent and one of them is a subclass of some superclass, then the second thing is also a subclass of the same superclass.
18. If any two agents are equivalent, and one of them performs an action, then they both perform that action.
19. If two objects are equivalent, they share all their properties.
20. If two objects are equivalent and an agent performs some action on one of them, then the agent performs that action on both of them.
21. If two objects are equivalent and an agent performs some action on one of them using an instrument, then the agent performs that action on the other of them using an instrument.
22. If something1 is the interior of something2, then something1 is part of the whole something2 and something1 is inside something2.
23. If something1 is part of something2 and something2 is part of something3, then something1 is part of something3.
24. If an agent holds something open, then that thing has the property open.
25. If a worm has a ring segment, the ring segment is part of the worm.
26. If something is the forward edge of something else, then the forward edge is part of the "something else" anyone named Paul is a male human.
27. If some agent mounts some animal (a1) using an instrument i, and the same agent uses the same instrument on part of another animal (a2), then a1 is equivalent to a2.
28. If an agent has some ability, then the agent possibly acts on that ability.

5.4 Background Commentary

The first rule is one of the more interesting ones in my background knowledge. Initially, I included a rule stating that "if an agent knows something, then that thing is the case". This rule was designed to basically cut the "paul knows that" part off of the sentence "paul knows that he has the ability to ...", leaving "paul has the ability to...". Clearly, this rule would allow that, as well as being general enough to handle any new sentences about what agents might know, and asserting those objects of knowledge to be true. However, my Snepsul attempt at representing "if an agent knows something, then that thing is the case" was unsuccessful. The system allowed me to assert rules like:

```
;; if an agent x knows something y, then y is the case
(add forall ($x $y)
     ant (build agent *x
                act (build action (build lex ‘‘know’’)
                           object *y))
     cq *y)
```

When the noun definition algorithm was run on a knowledge base containing this rule, it immediately went into an infinite loop. I tried several syntactic variants, but they all met with the same result. Whether this problem results more from a personal misunderstanding of Snepsul syntax or from a flaw in the noun algorithm remains a topic for further investigation.

While midly disappointing, this was not a major setback. I got around the problem by adding the less general rule that if an agent knows that they have a specific ability, then they have that specific ability:

```
;; if an agent knows that they (themselves) have an ability
```

```
;; to perform some action directed toward  some object using
;; some instrument, then they have that ability
(add forall ($anyone $anyobject $anyaction $anyobject $anyinstrument)
     ant (build agent *anyone
                act (build action (build lex "know")
                           object (build agent *anyone
                                        ability (build action *anyaction
                                                       object *anyobject
                                                       instrument *anyinstrument))))
     cq (build agent *anyone
               ability (build action *anyaction
                              object *anyobject
                              instrument *anyinstrument)))
```

This rule has the desired effect. When told that paul knows he has the ability, Cassie asserts that he does in fact have the ability. Figure 7 diagrams this rule. Note that node labels in the diagrams for this section may not correspond to node labels in diagrams from the "Passage" section.

Another interesting rule is the second to last one: "If an agent has some ability, then the agent possibly acts on that ability".

```
(add forall ($agent1 $act1 )
        ant (build agent *agent1
                   ability *act1)
        cq (build mode (build lex "possibly")
                  agent *agent1
                  act *act1))
```

Since the agent-ability case frame is not recognized by the noun definition algorithm, this rule (or one like it) was necessary to make the representation of the passage and the algorithm work together. See figure 8.

Note that this use of the mode arc is not the standard SNePS/CVA way of using a modal operator. The more standard way of doing this would be to use the mode/object case frame

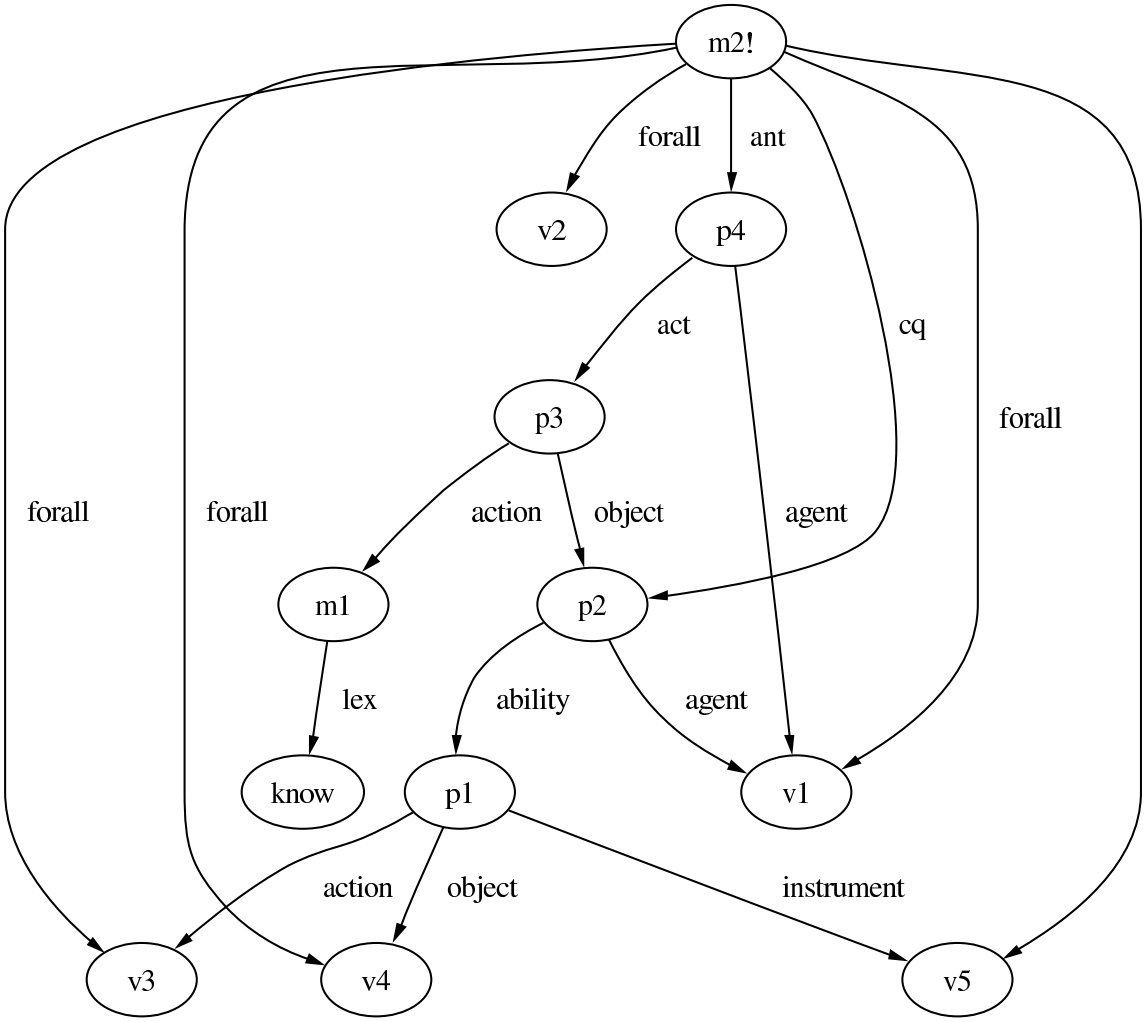

Figure 7: "If an agent knows that it has the ability to perform an action on some object with some instrument, then the agent has that ability"

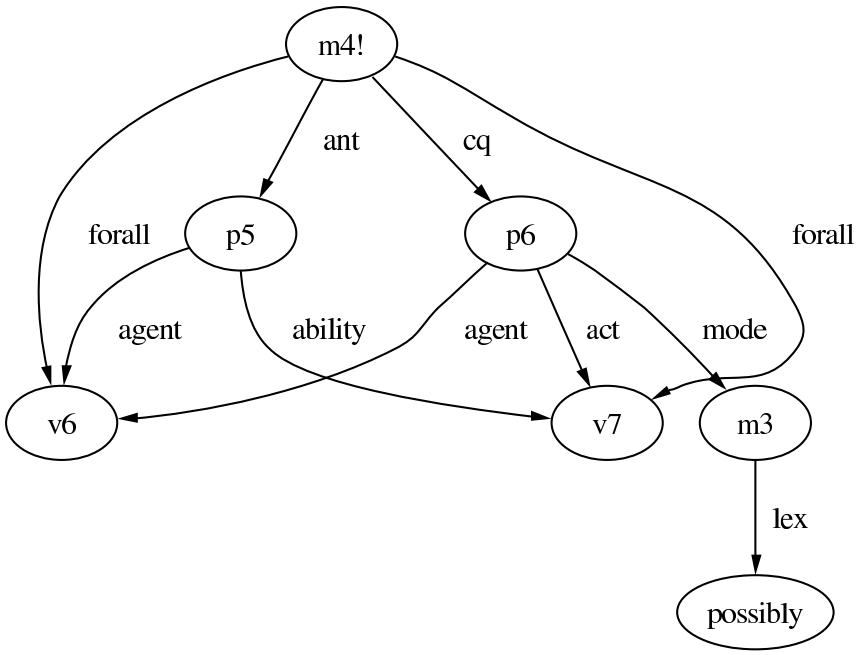

Figure 8: "If an agent has some ability, then the agent possibly acts on that ability"

(c.f. Broklawski, 2002:14), essentially moving the modal outside of the scope of the possible thing. This would change the above rule to something like:

```
(add forall ($agent1 $act1 )
        ant (build agent *agent1
                   ability *act1)
        cq (build mode (build lex "possibly")
                  object (build  agent *agent1
                                 act *act1)))
```

One disadvantage of the first representation is that the "possibly" might end up being inadvertently ignored at some point. SNePS performs *reduction inference*. If I search a knowledge base containing the first representation for this rule, and I am searching for propositions about agents and acts, but do not explicity ask about the mode, Cassie will return the agent/act part but will drop the mode arc. In this way, it is possible for the

information about an agent possibly doing some act to be interpreted as stating that the agent actually does that act.

At one point in the progress of this project, this rule simply stated that if an agent has the ability to do something, then the agent does that thing. It could be argued that in order to be *absolutely certain* that some agent can perform some act, the agent should be observed to perform the act at least once. Note that this might not necessarily apply to the narrative voice - being generally omniscient, it may not require empirical evidence to know that something is possible.

In any case, even this first use of the mode "possibly" is at least no worse than stating that if an agent has an ability it actually performs that act. For simplicity's sake (and because the second option made the definition algorithm stop working at all with my representation), the first of these two possible representations was kept in my knowledge base.

For a detailed syntax and semantics of my use of *mode*, please see Appendix A.

The final rule I will discuss in detail is number 27, "If some agent mounts some animal (a1) using an instrument i, and the same agent uses the same instrument i on part of another animal (a2), then a1 is equivalent to a2".

```
;; Rule 27
(add forall ($person $action1 $animal1 $animal2 $part-animal2 $instrument1)
     &ant ((build agent *person
                  act (build action (build lex "mount")
                            object *animal1
                            instrument *instrument1))
           (build part *part-animal2
                  whole *animal2)
           (build agent *instrument1
                  act (build action *action1
                            object *part-animal2)))
```

```
cq ( build equiv *animal1
           equiv *animal2))
```

This rule is essential to linking the two sentences, and determining that the maker is a worm. One of the few things linking the two sentences in the passage is the presence of the hook staffs. Paul and the maker are only mentioned in the first sentence. The worm and its behavior are only mentioned in the second sentence.

Recall that in the first sentence, the hook staffs appear as an instrument being used by the agent Paul as part of his act of mounting the maker. In the second sentence they appear as agents, themselves performing the act of holding open the forward edge of the ring segment. Rule 23 makes sure that the forward edge of the ring segment is considered part of the worm.

Cassie will not infer just because two things are equivalent (expressed by the equiv/equiv as the consequent of this rule), that those two things necessarily have the same properties, abilities or other features. Unfortunately, there is no way in SNePS to express a single rule that says "if two things are equivalent, then *everything* known about either of them applies to the both". Rules 15-21 are some rules of equivalence that will allow parts of the definitions for equivalent things to be copied over from one of the things to the other. These will be essential in defining what a maker is, as once it is determined that the maker and the worm are the same thing, we will want the things known about the maker to also be known of the maker.

6 Results

6.1 Introduction

The passage and background knowledge were placed in a SNePS demo file "maker.demo", which loads the noun algorithm and generally sets the system up for the algorithm to be run. The last line in the demo file is:

```
^(defineNoun ‘‘maker’’)
```

which calls the lisp function defineNoun for the word maker.

The file, "maker.demo" is included at the end of this paper as Appendix B.

This section describes the results obtained with the knowledge base as described so far and some necessary final changes made to the representation to overcome problems that came up.

6.2 First Issue: Unasserted Propositions

The first issue with the representation as presented so far has to do with the expression of the statment (derived from the passage) which states that, *"If the hook holds open the forward edge, then the forward edge admits the abrasive sand into the sensitive interior and the creature(worm) will not retreat below the desert."*

Recall that this is represented in Snepsul as a forall/antecedent/consequent/consequent rule (Fig. 5). The problem is that, since the rule deals with the hypothetical, the information about the *sand being admitted* and *the worm not retreating beneath the desert* is not asserted to the knowledge base, and so is apparently ignored by the noun definition algorithm. As

the time alloted to work on this project was ending, I came up with a rather simply but slightly unsatisfying solution: tell Cassie that it is the case that the hooks hold open the forward edge. Obviously, the passage does not directly say this. However, in the context of preceding passages (which Cassie doesn't know about, but I do) it is clear that Paul is *going to* perform the action of mounting the maker, as well as the associated actions involving the hook staffs.

Thus, in order for the information about the worm being prevented from retreating to show up in the definition, I have added this fact to the passage section of the knowledge base.

```
;; the hookstaff holds the forward edge of the ring segment open
(add agent *thehook
     act (build action (build lex "hold open")
                object *forward-edge))
```

6.3 Second Issue: Prolonged Looping

The second major issue is at first glance even worse than the first. The rule intended to tie the two sentences together (#27 in the background knowledge) sends the system into what is apparently an infinite loop - the demo doesn't even make it to the point where the definition algorithm is invoked. The demo does not however begin looping immediately after rule #27 is added. Rather, it hangs right after reaching the part of the passage which asserts "paul knows that he has the ability to mount the back of the maker, using the hook staff as an instrument" (see section 4.3.2). Recall that this assertion takes the form of a fairly large, deeply nested forall/ant/cq - in many ways similar to rule #27.

There are actually two different ways of asserting propositions in SNePS. The first is the *assert* function. The function I have used throughout the project is the *add* function. The difference between these is that *assert* simply places propositions in the knowledge base of things Cassie believes, while *add* automatically triggers forward inference, causing Cassie to discover and believe consequences of her beliefs.

My hypothesis is that I am giving Cassie too many complex, related rules, and that they are triggering either an infinite or simply *very large* inference loop that she gets caught in. I verified this suspicion by entering the offending rule *after* the rest of the background and passage had been entered. Since the rule was not initially in the knowledge base, the out-of-control forward inference did not occur while inputting the background knowledge nor during Cassie's reading of the passage. However once I entered rule #27, Cassie concludes (among other things) that the worm and the maker are equivalent! Running the definition algorithm at this point caused the same looping behavior.

Here is the situation: Rule #27 is the key to tying together the two sentences and to determining that makers are equivalent to worms - a significantly useful piece of information. If the rule is entered before most of the background and all of the passage, it exhibits a looping behavior with no promise of halting.

If the rule is entered after the background and passage representations, Cassie will correctly conclude that the maker and worm are equivalent (among quite a few other things), and the looping is (temporarily) averted because of the order in which information was added. I made absolutely sure that the rule was adding the fact that *the worm and the maker are equivalent* with the command:

```
;; ask Cassie whether any two things are equivalent
(deduce equiv $m equiv $w)
```

- to which Cassie replied with the name of a proposition that asserted this fact.

If we then ask Cassie to define "maker" for us, some inference triggered by the algorithm sends the system into the same (or an indistinguishable) inference loop.

Rule #27 is only being used to infer from both sentences that the worm and the maker are equivalent. It works for that purpose, but after that point it is an obstacle to obtaining a proper definition. If we first add the rule and then remove it, all the results of the rule - including the equivalence relation that we need - will be erased with it. My temporary solution to this problem is to leave the rule out and simply assert the equivalence that the rule was shown to produce:

```
;; tell Cassie that the maker and the worm are the same thing
(add equiv *themaker equiv *worm)
```

6.4 The Definition

The definition produced by the noun algorithm at the end of my *maker.demo* file is:

```
* ^(defineNoun "maker")

-->  Definition of maker:
 Possible Class Inclusions: animal, worm, creature,
 Possible Structure: b6, b8,
 Possible Actions: retreat beneath desert, tunnel ground,
 Possible Properties: elongated, flexible, larger than human male,
nil
```

7 Future Work

The first thing that really needs to be taken care of in order for my work on this passage to continue is the looping issue mentioned in the last section. The temporary solution is fine for now and demonstrates that the right information is being asserted into the knowledge base. However, the result (the fact that the maker is the worm) really needs to be derived by Cassie based on her background information and the passage without having a user assert it to her. An important next step would be to simplify the most complex rules in the knowledge base and see whether that cuts down on the looping. Another promising way of dealing with the situation is to convert some of the forall/ant/cq rules (of which there are many) into path definitions, thereby replacing forward inference for those rules with path-based inference, and possibly eliminating the overzealous inference loop. Other smaller issues for further work include the first problem mentioned in the last section, as well as building in some rules to allow a notion of cause and effect.

Long-term, I could work on developing a representation that preserves the notion of "for as long as"/"for the duration of" found in the second sentence of the passage (see section 4.2). Approaches to this might include developing an ontology of time, or implementing a situational calculus. Any of these long-term options would probably also involve making changes to the noun definition algorithm itself.

Right now, the method by which I am doing noun phrase resolution (besides not quite working) is very specific to the context of this passage. Another area of potential long-term work on this project is this area of resolving different noun phrases which refer to the same individual across subpassages.

Finally an idea for future work that is not directly to this passage (but which could make use of it): I recently spent some time researching and writing about Social Network Analysis, which uses the model of a network comprised of arcs and nodes to analyze various types social systems. The methodologies of SNA borrow heavily from graph theory in Mathematics and are not restricted to social networks (which seem to have a pretty broad definition anyway). It would be interesting to apply social network analysis techniques to SNePS networks, especially SNePS networks created for the CVA project. Specifically, I am interested in how the *centralities* (*degree centrality*, *betweenness centrality* and *closeness centrality*) of "unknown word" nodes in a network might relate to the effectiveness of the definition algorithms in defining the word.

8 Conclusions

This paper has presented a detailed summary of the work I have completed to represent a passage containing the unknown word "maker". Some work still remains in order for my representation to allow Cassie to derive a meaning that uses all the information in the passage. Specifically, there is one major flaw in the representation that causes the system to go into a loop that does not terminate. Overall, the representations of both the background knowledge and the passage itself are sound and allow Cassie (with a little help) to give a satisfactory definition for the word.

9 References

1. Ahmed, Adel (2003), "Contextual Vocabulary Acquisition: Inferring the Meaning of the Verb 'Cripple' from Context", [http://www.cse.buffalo.edu/ rapaport/CVA/Cripple/ahmed.pdf].

2. Broklawski, Marc K. (2002), "Computational Contextual Vocabulary Acquisition: A Noun Algorithm", [http://www.cse.buffalo.edu/ rapaport/CVA/docs/cva.wrapup.pdf].

3. Ehrlich, Karen (1995), "Automatic Vocabulary Expansion through Narrative Context", Technical Report 95-09 (Buffalo: SUNY Buffalo Department of Computer Science).

4. Herbert, Frank. (1965) "Dune" New York: Berkley.

5. William J. Rapaport, Stuart C. Shapiro, and Janyce M. Wiebe (1997), "Quasi-Indexicals and Knowledge Reports", Cognitive Science 21, 1 (January-March, 1997), 63-107.

6. Shapiro, Stuart C. and The SNePS Implementation Group (2002), "SNePS User Manual", [http://www.cse.buffalo.edu/sneps/Manuals/manual26.ps].

Appendix A: Case Frames

agent/ability

-Syntax:

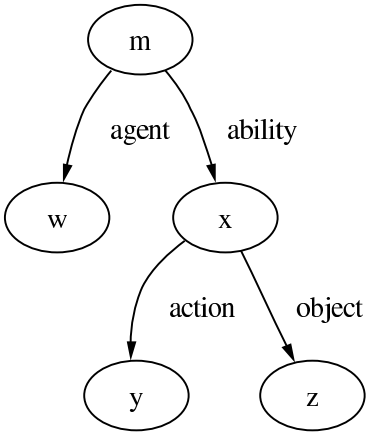

Figure 9: "A.1: agent/ability"

If w, y and z are individual nodes and 'm' and 'x' identifiers not previously used, then Figure A.1 is a network, and m and x are structured proposition nodes.

-Semantics: *[[m]]* is the proposition that *[[w]]* is an agent with the ability to perform the action *[[y]]* directed towards the object *[[z]]*.

-Sample Context: Figure 1 in section 4.3.2

agent/act/mode

-Syntax:

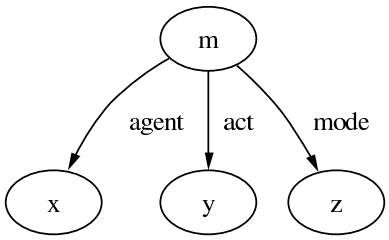

Figure 10: "A.2: agent/act/mode"

If x, y and z are individual nodes and 'm' is an identifier not previously used, then Figure A.2 is a network, and m is a structured proposition node.

-Semantics: *[[m]]* is the proposition that *[[x]]* is an agent with the ability to perform the action *[[y]]* with modality! *[[z]]*.

-Sample Context: Figure 8 in section 5.4

agent/act/action/object/instrument

-Syntax:

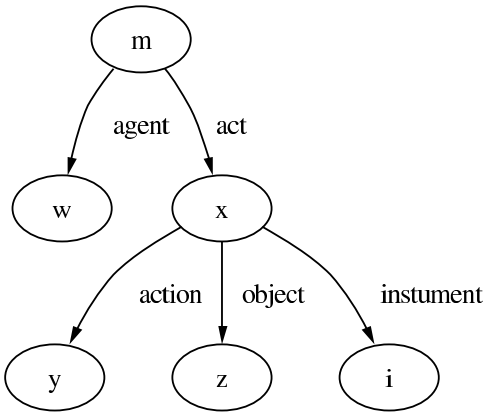

Figure 11: "A.3: agent/act/instrument/object"

If w, y, z and i are individual nodes and 'm' and 'x' are identifiers not previously used, then Figure A.3 is a network, and m and x are structured proposition nodes.

-Semantics: *[[m]]* is the proposition that *[[w]]* is an agent who performs the act consisting of performing the action *[[y]]* directed towards the object *[[z]]* using as an instrument *[[i]]*.

-Sample Context: Figure 1 in section 4.3.2

Appendix B: Background Representation

```
; BACKGROUND KNOWLEDGE:
; =====================

;; if an agent knows that they (themselves) have an ability to perform some action directed tow
;; some object and using some instrument, then they have that ability
(describe
(add forall ($anyone $anyobject $anyaction $anyobject $anyinstrument)
        ant (build agent *anyone
                   act (build action (build lex "know")
                             object (build agent *anyone
                                          ability (build action *anyaction
                                                         object *anyobject
                                                         instrument *anyinstrument))))
        cq (build agent *anyone
                  ability (build action *anyaction
                                object *anyobject
                                instrument *anyinstrument))))

;; if an agent mounts some thing, and the thing is the back of an object, then
;; the agent mounts the object
(describe
(add forall ($mounter $mountee-back $mountee)
     &ant ((build agent *mounter
                  act (build action (build lex "mount")
                            object *mountee-back))
           (build possessor *mountee
                  rel (build lex "back")
                  object *mountee-back))
     cq (build agent *mounter
               act (build action (build lex "mount")
                         object *mountee))))

;; if anything mounts something else, then the first thing rides the second
(describe
(add forall ($mounter $mountee)
        ant (build agent *mounter
                   act (build action (build lex "mount")
                             object *mountee))
        cq (build agent *mounter
                   act (build action (build lex "ride")
                             object *mountee))))

;; if something rides something else, the rider is on the ridee
(describe
```

```
(add forall ($rider $ridee)
        ant (build agent *rider
                   act (build action (build lex "ride")
                             object *ridee))
        cq (build object1 *rider
                  rel (build lex "on")
                  object2 *ridee)))

;; something ridden is possibly an animal
(describe
(add forall ($rider $ridee)
     ant (build agent *rider
                act (build action (build lex "ride")
                           object *ridee))
     cq (build mode (build lex "possibly")
               member *ridee class (build lex "animal"))))

;; if something1 mounts something2 then something1 is larger than something2
(describe
(add forall ($p2 $a2)
        ant (build agent *p2
                   act (build action (build lex "mount")
                             object *a2))
        cq (build object1 *a2
                  rel (build lex  "larger than")
                  object2 *p2)))

;; if x is smaller than y then y is larger than x
(describe
(add forall ($small $large)
        ant (build object1 *small
                   rel (build lex "smaller than")
                   object2 *large)
        cq (build object1 *large
                   rel (build lex "larger than")
                   object2 *small)))

;; if x is larger than y, then y is smaller than x
(describe
(add forall ($small2 $large2)
        ant (build object1 *large2
                   rel (build lex "larger than")
                   object2 *small2)
        cq (build object1 *small2
                   rel (build lex "smaller than")
                   object2 *large2))
)
```

```
;; sand is part of the desert
(describe
(add part (build lex "sand")
     whole (build lex "desert")))

;; humans are animals
(describe
(add subclass (build lex "human")
     superclass (build lex "animal")))

;; worms are animals
(describe
(add subclass (build lex "worm")
     superclass (build lex "animal")))

;; worms are elongated and flexible
(describe
(add forall $w
        ant (build member *w
                  class (build lex "worm"))
        cq ((build object *w
                  property (build lex "flexible"))
            (build object *w
                  property (build lex "elongated")))))

;; creatures are animals
(describe
(add subclass (build lex "creature")
     superclass (build lex "animal")))

;; worms tunnel in the ground
(describe
(add forall $w100
        ant (build member *w100 class (build lex "worm"))
         cq (build agent *w100
                  ability (build action (build lex "tunnel")
                                object (build lex "ground")))))

;; information about equivalency:
;; if two classes are equivalent, then they are each subclasses of the other
(describe
(add forall ($c1 $c2)
        ant (build equiv *c1
                  equiv *c2)
        cq ((build subclass *c1 superclass *c2)
            (build subclass *c2 subclass *c1))))
```

```
;; if two things are equivalent and one of them is a member
;; of some class, the other is also a member of that class
(describe
(add forall ($thing1 $class $thing2)
     &ant ((build member *thing1 class *class)
           (build equiv *thing1 equiv *thing2))
     cq (build member *thing2 class *class)))

;; if two things are equivalent and one of them is a subclass
;; of some superclass, then the second thing is also a subclass
;; of the same superclass
(describe
(add forall ($subclass1 $subclass2 $superclass)
     &ant((build subclass *subclass1
                 superclass *superclass)
          (build equiv *subclass1 equiv *subclass2))
     cq (build subclass *subclass2
               superclass *superclass)))

;; if any two agents are equivalent, and one of them performs an action,
;; then the other performs that action
(describe
(add forall ($agent1 $agent2 $action)
     &ant ((build equiv *agent1
                   equiv *agent2)
           (build agent *agent1
                  ability *action))
       cq (build agent *agent2
                 ability *action)))

;; if two objects are equivalent, they share all their properties
(describe
(add forall ($object1 $object2 $prop)
       &ant ((build equiv *object1
                     equiv *object2)
     ( build object *object1
                    property *prop))
        cq (build object *object2
                  property *prop)))

;; if two objects are equivalent and an agent performs some action on one of them,
;; then the agent performs that action on both of them
(describe
(add forall ($object1 $object2 $agent1 $action1)
     &ant ((build equiv *object1
                   equiv *object2)
```

```
           (build agent *agent1
                 act  (build action *action1
                            object *object1)))
     cq (build agent *agent1
               act  (build action *action1
                           object *object2))))

;; if two objects are equivalent and an agent performs some action on one of them
;; using an instrument,  then the agent performs that action on the other of them
(describe
(add forall ($object1 $object2 $agent1 $action1 $inst)
     &ant ((build equiv *object1
                  equiv *object2)
           (build agent *agent1
                  act (build action *action1
                             object *object1
                             instrument *inst)))
     cq (build agent *agent1
               act (build action *action1
                           object *object2
                           instrument *inst))))

;; anyone named Paul is a male human
(describe
(add forall $p
     ant  (build object *p proper-name "paul")
     cq ((build member *p class (build lex "human"))
         (build member *p class (build lex "male")))))

;; if something1 is the interior of something2
;; then something1 is part of the whole something2
;; then something1 is inside something2
(describe
(add forall ($x $y)
          ant (build possessor *x
                          rel (build lex "interior")
                          object *y)
          cq((build part *y whole *x)
                (build object1 *y
                           rel (build lex "inside")
                           object2 *x))))

;; if an agent holds something open, then that thing has the property open
(describe
(add forall ($thing $agent $object)
     ant (build agent *agent
               act (build action (build lex "hold open")
```

```
                          object *object))
     cq (build object *object
               property (build lex "open"))))

;; if something1 is a part of something2
;; and somehting2 is a part of something3
;; then something1 is a part of something3
(describe
(add forall ($part1 $part2 $whole)
     &ant  ((build part *part1
                   whole *part2)
            (build part *part2
                   whole whole))
    cq (build part *part1
              whole *whole)))

;; COMMENTED OUT BECAUSE IT SENDS CASSIE INTO AN INFINITE LOOP
;; if an agent mounts some animal1 using instrument1, and instrument1
;; performs some action on part of animal2, then animal1 and animal2
;; are equivalent (they are the same entity)
;(describe
;(add forall ($person $action1 $animal1 $animal2 $part-animal2 $instrument1)
;     &ant ((build agent *person
;                   act (build action (build lex "mount")
;                             object *animal1
;                             instrument *instrument1))
;           (build part *part-animal2
;                   whole *animal2)
;           (build agent *instrument1
;                   act (build action *action1
;                             object *part-animal2)))
;    cq ( build equiv *animal1
;                equiv *animal2))
;)

;; if a worm has a ring segment, the ring segment is part of the worm
(describe
(add forall ($rs $w)
     &ant ((build possessor *w
                   object *rs
                   rel (build lex "ring segment"))
           (build member *w
                   class (build lex "worm")))
     cq (build part *rs
               whole *w)))

;; if something is the forward edge of something else,
```

```
;; then the forward edge is part of the other thing
(describe
(add forall ($edge $thing)
        ant (build possessor *thing
                    object *edge
                    rel (build lex "forward edge"))
        cq (build part *edge
                  whole *thing)))

;; if an agent has some ability then the agent
;; possibly acts on that ability
(describe
(add forall ($agent1 $act1 )
        ant (build agent *agent1
                    ability *act1)
        cq (build mode (build lex "possibly")
                  agent *agent1
                  act *act1))
)
```

Appendix C: Demo Transcript

```
; ========================================================================
; FILENAME: buildmaker.demo
; DATE: 12-14-03
; PROGRAMMER: Jonathan Bona
;
; Lines beginning with a semi-colon are comments.
; Lines beginning with "^" are Lisp commands.
; All other lines are SNePS commands.
;
; To use this file: run SNePS; at the SNePS prompt (*), type:
;
; (demo "~/class/663/projects/one/code/maker.demo" :av)
;
; Make sure all necessary files are in the current working directory
; or else use full path names.
; ========================================================================

; Turn off inference tracing.
; This is optional; if tracing is desired, then delete this.
^(
--> setq snip:*infertrace* nil)
nil

 CPU time : 0.00
*
; Load the appropriate definition algorithm:
; first: on cse systems, second: on laptop
^(
--> load "/projects/rapaport/CVA/STN2/defun_noun.cl")
; Loading /projects/rapaport/CVA/STN2/defun_noun.cl
t

 CPU time : 0.17
*
; Clear the SNePS network:
(resetnet)

Net reset - Relations and paths are still defined

 CPU time : 0.03
*
; OPTIONAL:
; UNCOMMENT THE FOLLOWING CODE TO TURN FULL FORWARD INFERENCING ON:
;
```

```
;enter the "snip" package:
; ^(in-package snip)

;turn on full forward inferencing:
; ^(defun broadcast-one-report (represent)
;    (let (anysent)
;      (do.chset (ch *OUTGOING-CHANNELS* anysent)
;         (when (isopen.ch ch)
;                      (setq anysent
;                      (or (try-to-send-report represent ch)
;                           anysent)))))
;    nil)

;re-enter the "sneps" package:
; ^(in-package sneps)

; load all pre-defined relations:
(intext "/projects/rapaport/CVA/STN2/demos/rels")
File /projects/rapaport/CVA/STN2/demos/rels is now the source of input.

 CPU time : 0.00
* (a1 a2 a3 a4 after agent against antonym associated before cause class
 direction equiv etime event from in indobj instr into lex location manner
 member mode object on onto part place possessor proper-name property rel skf
 sp-rel stime subclass superclass subset superset synonym time to whole kn_cat)

 CPU time : 0.02
* End of file /projects/rapaport/CVA/STN2/demos/rels

 CPU time : 0.02
* (intext "~jpbona/class/663/projects/one/code/rels")
File ~jpbona/class/663/projects/one/code/rels is now the source of input.

 CPU time : 0.00
(a1 a2 a3 a4 after agent against antonym associated before cause class
 direction equiv etime event from in indobj instr into lex location manner
 member mode object on onto part place possessor proper-name property rel skf
 sp-rel stime subclass superclass subset superset synonym time to whole kn_cat)

 CPU time : 0.01
* End of file ~jpbona/class/663/projects/one/code/rels

 CPU time : 0.02
* (define instrument ability)
(instrument ability)
 CPU time : 0.00
* ; load all pre-defined path definitions:
```

```
(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")
File /projects/rapaport/CVA/mkb3.CVA/paths/paths is now the source of input.
 CPU time : 0.00
* before implied by the path (compose before (kstar (compose after- ! before)))
before- implied by the path (compose (kstar (compose before- ! after)) before-)

 CPU time : 0.00
* after implied by the path (compose after (kstar (compose before- ! after)))
after- implied by the path (compose (kstar (compose after- ! before)) after-)

 CPU time : 0.00
* sub1 implied by the path (compose object1- superclass- ! subclass superclass-
                           ! subclass)
sub1- implied by the path (compose subclass- ! superclass subclass- !
                           superclass object1)

 CPU time : 0.00
* super1 implied by the path (compose superclass subclass- ! superclass object1-
                             ! object2)
super1- implied by the path (compose object2- ! object1 superclass- ! subclass
                             superclass-)

 CPU time : 0.00
* superclass implied by the path (or superclass super1)
superclass- implied by the path (or superclass- super1-)

 CPU time : 0.00
*

End of file /projects/rapaport/CVA/mkb3.CVA/paths/paths

 CPU time : 0.01
*
; BACKGROUND KNOWLEDGE:
; =====================

;; if an agent knows that they (themselves) have an ability to perform some action directed tow
;; some object and using some instrument, then they have that ability
(describe
(add forall ($anyone $anyobject $anyaction $anyobject $anyinstrument)
        ant (build agent *anyone
                   act (build action (build lex "know")
                             object (build agent *anyone
                                           ability (build action *anyaction
                                                          object *anyobject
                                                          instrument *anyinstrument))))
        cq (build agent *anyone
```

```
                  ability (build action *anyaction
                                object *anyobject
                                instrument *anyinstrument))))

(m2! (forall v5 v4 v3 v2 v1)
 (ant
  (p4
   (act (p3 (action (m1 (lex know)))
         (object
          (p2 (ability (p1 (action v3) (instrument v5) (object v4)))
           (agent v1)))))
   (agent v1)))
 (cq (p2)))

(m2!)

 CPU time : 0.00

* ;; if an agent mounts some thing, and the thing is the back of an object, then
;; the agent mounts the object
(describe
(add forall ($mounter $mountee-back $mountee)
     &ant ((build agent *mounter
                  act (build action (build lex "mount")
                            object *mountee-back))
           (build possessor *mountee
                  rel (build lex "back")
                  object *mountee-back))
     cq (build agent *mounter
               act (build action (build lex "mount")
                         object *mountee))))

(m5! (forall v8 v7 v6)
 (&ant (p7 (object v7) (possessor v8) (rel (m4 (lex back))))
  (p6 (act (p5 (action (m3 (lex mount))) (object v7))) (agent v6)))
 (cq (p9 (act (p8 (action (m3)) (object v8))) (agent v6))))

(m5!)
 CPU time : 0.01

* ;; if anything mounts something else, then the first thing rides the second
(describe
(add forall ($mounter $mountee)
        ant (build agent *mounter
                   act (build action (build lex "mount")
                             object *mountee))
        cq (build agent *mounter
```

```
                  act (build action (build lex "ride")
                            object *mountee))))

(m7! (forall v10 v9)
 (ant (p11 (act (p10 (action (m3 (lex mount))) (object v10))) (agent v9)))
 (cq (p13 (act (p12 (action (m6 (lex ride))) (object v10))) (agent v9))))
(m7!)

 CPU time : 0.01
* ;; if something rides something else, the rider is on the ridee
(describe
(add forall ($rider $ridee)
        ant (build agent *rider
                   act (build action (build lex "ride")
                             object *ridee))
        cq (build object1 *rider
                  rel (build lex "on")
                  object2 *ridee))
)
(m9! (forall v12 v11)
 (ant (p15 (act (p14 (action (m6 (lex ride))) (object v12))) (agent v11)))
 (cq (p16 (object1 v11) (object2 v12) (rel (m8 (lex on))))))
(m9!)
 CPU time : 0.01

*;; something ridden is possibly an animal
(describe
(add forall ($rider $ridee)
     ant (build agent *rider
                act (build action (build lex "ride")
                           object *ridee))
     cq (build mode (build lex "possibly")
               member *ridee class (build lex "animal"))))

(m12! (forall v14 v13)
 (ant (p18 (act (p17 (action (m6 (lex ride))) (object v14))) (agent v13)))
 (cq
  (p19 (class (m11 (lex animal))) (member v14) (mode (m10 (lex possibly))))))

(m12!)
 CPU time : 0.01

*;; if something1 mounts something2 then something1 is larger than something2
(describe
(add forall ($p2 $a2)
        ant (build agent *p2
                   act (build action (build lex "mount")
```

```
                              object *a2))
        cq (build object1 *a2
                  rel (build lex  "larger than")
                  object2 *p2)))

(m14! (forall v16 v15)
 (ant (p21 (act (p20 (action (m3 (lex mount))) (object v16))) (agent v15)))
 (cq (p22 (object1 v16) (object2 v15) (rel (m13 (lex larger than))))))

(m14!)

 CPU time : 0.01

*
;; if x is smaller than y then y is larger than x
(describe
(add forall ($small $large)
        ant (build object1 *small
                   rel (build lex "smaller than")
                   object2 *large)
        cq (build object1 *large
                   rel (build lex "larger than")
                   object2 *small)))

(m16! (forall v18 v17)
 (ant (p23 (object1 v17) (object2 v18) (rel (m15 (lex smaller than)))))
 (cq (p24 (object1 v18) (object2 v17) (rel (m13 (lex larger than))))))

(m16!)

 CPU time : 0.01

*
;; if x is larger than y, then y is smaller than x
(describe
(add forall ($small2 $large2)
        ant (build object1 *large2
                   rel (build lex "larger than")
                   object2 *small2)
        cq (build object1 *small2
                   rel (build lex "smaller than")
                   object2 *large2)))

(m17! (forall v20 v19)
 (ant (p25 (object1 v20) (object2 v19) (rel (m13 (lex larger than)))))
 (cq (p26 (object1 v19) (object2 v20) (rel (m15 (lex smaller than))))))
(m17!)
```

```
 CPU time : 0.01

*;; sand is part of the desert
(describe
(add part (build lex "sand")
     whole (build lex "desert")))

(m20! (part (m18 (lex sand))) (whole (m19 (lex desert))))
(m20!)
 CPU time : 0.01

*;; humans are animals
(describe
(add subclass (build lex "human")
     superclass (build lex "animal")))

(m22! (subclass (m21 (lex human))) (superclass (m11 (lex animal))))
(m22!)
 CPU time : 0.00

*;; worms are animals
(describe
(add subclass (build lex "worm")
     superclass (build lex "animal")))

(m24! (subclass (m23 (lex worm))) (superclass (m11 (lex animal))))
(m24!)
 CPU time : 0.01

*;; worms are elongated and flexible
(describe
(add forall $w
        ant (build member *w
                   class (build lex "worm"))
        cq ((build object *w
                   property (build lex "flexible"))
            (build object *w
                   property (build lex "elongated")))))

(m27! (forall v21) (ant (p27 (class (m23 (lex worm))) (member v21)))
 (cq (p29 (object v21) (property (m26 (lex elongated))))
  (p28 (object v21) (property (m25 (lex flexible))))))

(m27!)
 CPU time : 0.04
```

```
*;; creatures are animals
(describe
(add subclass (build lex "creature")
     superclass (build lex "animal")))

(m29! (subclass (m28 (lex creature))) (superclass (m11 (lex animal))))
(m29!)
 CPU time : 0.00

* ;; worms tunnel in the ground
(describe
(add forall $w100
        ant (build member *w100 class (build lex "worm"))
         cq (build agent *w100
                   ability (build action (build lex "tunnel")
                                 object (build lex "ground")))))

(m33! (forall v22) (ant (p30 (class (m23 (lex worm))) (member v22)))
 (cq
  (p31 (ability (m32 (action (m30 (lex tunnel))) (object (m31 (lex ground)))))
   (agent v22))))

(m33!)
 CPU time : 0.01

* ;; if two classes are equivalent, then they are each subclasses of the other
(describe
(add forall ($c1 $c2)
        ant (build equiv *c1
                   equiv *c2)
        cq ((build subclass *c1 superclass *c2)
            (build subclass *c2 subclass *c1))))

(m34! (forall v24 v23) (ant (p32 (equiv v24 v23)))
 (cq (p34 (subclass v24 v23)) (p33 (subclass v23) (superclass v24))))

(m34!)
 CPU time : 0.01

* ;; if two things are equivalent and one of them is a member
;; of some class, the other is also a member of that class
(describe
(add forall ($thing1 $class $thing2)
     &ant ((build member *thing1 class *class)
           (build equiv *thing1 equiv *thing2))
     cq (build member *thing2 class *class)))
```

```
(m35! (forall v27 v26 v25)
 (&ant (p36 (equiv v27 v25)) (p35 (class v26) (member v25)))
 (cq (p37 (class v26) (member v27))))
(m35!)
 CPU time : 0.02

*;; if two things are equivalent and one of them is a subclass
;; of some superclass, then the second thing is also a subclass
;; of the same superclass
(describe
(add forall ($subclass1 $subclass2 $superclass)
     &ant((build subclass *subclass1
                 superclass *superclass)
          (build equiv *subclass1 equiv *subclass2))
     cq (build subclass *subclass2
               superclass *superclass)))

(m36! (forall v30 v29 v28)
 (&ant (p39 (equiv v29 v28)) (p38 (subclass v28) (superclass v30)))
 (cq (p40 (subclass v29) (superclass v30))))
(m29! (subclass (m28 (lex creature))) (superclass (m11 (lex animal))))
(m24! (subclass (m23 (lex worm))) (superclass (m11)))
(m22! (subclass (m21 (lex human))) (superclass (m11)))

(m36! m29! m24! m22!)
 CPU time : 0.02

*;; if any two agents are equivalent, and one of them performs an action,
;; then the other performs that action
(describe
(add forall ($agent1 $agent2 $action)
     &ant ((build equiv *agent1
                  equiv *agent2)
           (build agent *agent1
                  ability *action))
       cq (build agent *agent2
                 ability *action)))

(m37! (forall v33 v32 v31)
 (&ant (p42 (ability v33) (agent v31)) (p41 (equiv v32 v31)))
 (cq (p43 (ability v33) (agent v32))))

(m37!)
 CPU time : 0.02

* ;; if two objects are equivalent, they share all their properties
(describe
```

```
(add forall ($object1 $object2 $prop)
       &ant ((build equiv *object1
                     equiv *object2)
     ( build object *object1
                      property *prop))
        cq (build object *object2
                  property *prop)))

(m38! (forall v36 v35 v34)
 (&ant (p45 (object v34) (property v36)) (p44 (equiv v35 v34)))
 (cq (p46 (object v35) (property v36))))

(m38!)
 CPU time : 0.03

*;; if two objects are equivalent and an agent performs some action on one of them,
;; then the agent performs that action on both of them
(describe
(add forall ($object1 $object2 $agent1 $action1)
     &ant ((build equiv *object1
                  equiv *object2)
           (build agent *agent1
                 act  (build action *action1
                            object *object1)))
     cq (build agent *agent1
               act  (build action *action1
                           object *object2))))

(m39! (forall v40 v39 v38 v37)
 (&ant (p49 (act (p48 (action v40) (object v37))) (agent v39))
  (p47 (equiv v38 v37)))
 (cq (p51 (act (p50 (action v40) (object v38))) (agent v39))))

(m39!)
 CPU time : 0.03

*;; if two objects are equivalent and an agent performs some action on one of them
;; using an instrument,  then the agent performs that action on the other of them
(describe
(add forall ($object1 $object2 $agent1 $action1 $inst)
     &ant ((build equiv *object1
                  equiv *object2)
           (build agent *agent1
                  act (build action *action1
                             object *object1
                             instrument *inst)))
     cq (build agent *agent1
```

```
               act (build action *action1
                             object *object2
                             instrument *inst))))

(m41! (forall v44 v43 v42 v41)
 (&ant (p58 (act (p57 (action v44) (object v41))) (agent v43))
  (p52 (equiv v42 v41)))
 (cq (p60 (act (p59 (action v44) (object v42))) (agent v43))))
(m40! (forall v45 v44 v43 v42 v41)
 (&ant
  (p54 (act (p53 (action v44) (instrument v45) (object v41))) (agent v43))
  (p52))
 (cq (p56 (act (p55 (action v44) (instrument v45) (object v42))) (agent v43))))
(m39! (forall v40 v39 v38 v37)
 (&ant (p49 (act (p48 (action v40) (object v37))) (agent v39))
  (p47 (equiv v38 v37)))
 (cq (p51 (act (p50 (action v40) (object v38))) (agent v39))))

(m41! m40! m39!)
 CPU time : 0.06

* ;; anyone named Paul is (most likely) a male human
(describe
(add forall $p
     ant  (build object *p proper-name "paul")
     cq ((build member *p class (build lex "human"))
         (build member *p class (build lex "male")))))

(m43! (forall v46) (ant (p61 (object v46) (proper-name paul)))
 (cq (p63 (class (m42 (lex male))) (member v46))
  (p62 (class (m21 (lex human))) (member v46))))

(m43!)
 CPU time : 0.05

*;; if something1 is the interior of something2
;; then something1 is part of the whole something2
;; then something1 is inside something2
(describe
(add forall ($x $y)
          ant (build possessor *x
                         rel (build lex "interior")
                         object *y)
          cq((build part *y whole *x)
                (build object1 *y
                           rel (build lex "inside")
                           object2 *x))))
```

```
(m46! (forall v48 v47)
 (ant (p64 (object v48) (possessor v47) (rel (m44 (lex interior)))))
 (cq (p66 (object1 v48) (object2 v47) (rel (m45 (lex inside))))
  (p65 (part v48) (whole v47))))

(m46!)
 CPU time : 0.02

*;; if an agent holds something open, then that thing has the property open
(describe
(add forall ($thing $agent $object)
     ant (build agent *agent
                act (build action (build lex "hold open")
                           object *object))
     cq (build object *object
               property (build lex "open"))))

(m49! (forall v51 v50 v49)
 (ant
  (p68 (act (p67 (action (m47 (lex hold open))) (object v51))) (agent v50)))
 (cq (p69 (object v51) (property (m48 (lex open))))))

(m49!)
 CPU time : 0.07

*;; if something1 is a part of something2
;; and somehting2 is a part of something3
;; then something1 is a part of something3
(describe
(add forall ($part1 $part2 $whole)
     &ant  ((build part *part1
                   whole *part2)
            (build part *part2
                   whole whole))
    cq (build part *part1
              whole *whole)))

(m50! (forall v54 v53 v52)
 (&ant (p83 (part v53) (whole whole)) (p82 (part v52) (whole v53)))
 (cq (p84 (part v52) (whole v54))))
(m20! (part (m18 (lex sand))) (whole (m19 (lex desert))))

(m50! m20!)
 CPU time : 0.08

* ;; COMMENTED OUT BECAUSE IT SENDS CASSIE INTO AN INFINITE LOOP
```

```
;; if an agent mounts some animal1 using instrument1, and instrument1
;; performs some action on part of animal2, then animal1 and animal2
;; are equivalent (they are the same entity)
;(describe
;(add forall ($person $action1 $animal1 $animal2 $part-animal2 $instrument1)
;     &ant ((build agent *person
;                 act (build action (build lex "mount")
;                           object *animal1
;                           instrument *instrument1))
;           (build part *part-animal2
;                 whole *animal2)
;           (build agent *instrument1
;                 act (build action *action1
;                           object *part-animal2)))
;    cq ( build equiv *animal1
;               equiv *animal2))
;)

;; if a worm has a ring segment, the ring segment is part of the worm
(describe
(add forall ($rs $w)
     &ant ((build possessor *w
                 object *rs
                 rel (build lex "ring segment"))
           (build member *w
                 class (build lex "worm")))
     cq (build part *rs
               whole *w)))

(m52! (forall v56 v55)
 (&ant (p89 (class (m23 (lex worm))) (member v56))
  (p88 (object v55) (possessor v56) (rel (m51 (lex ring segment)))))
 (cq (p90 (part v55) (whole v56))))

(m52!)
 CPU time : 0.05

*;; if something is the forward edge of something else,
;; then the forward edge is part of the other thing
(describe
(add forall ($edge $thing)
        ant (build possessor *thing
                 object *edge
                 rel (build lex "forward edge"))
        cq (build part *edge
                  whole *thing)))
```

```
(m54! (forall v58 v57)
 (ant (p93 (object v57) (possessor v58) (rel (m53 (lex forward edge)))))
 (cq (p94 (part v57) (whole v58))))

(m54!)
 CPU time : 0.03

*;; if an agent has some ability then the agent
;; possibly acts on that ability
(describe
(add forall ($agent1 $act1 )
        ant (build agent *agent1
                   ability *act1)
        cq (build mode (build lex "possibly")
                  agent *agent1
                  act *act1)))

(m55! (forall v60 v59) (ant (p95 (ability v60) (agent v59)))
 (cq (p96 (act v60) (agent v59) (mode (m10 (lex possibly))))))

(m55!)
 CPU time : 0.03

* ; CASSIE READS THE PASSAGE:
; ========================
; "With the whip like hook-staffs Paul knew he could mount the maker's
;  high curving back. For as long as a forward edge of the worm's
;  ring segment was held open by a hook open to admit abrasive sand into
;  the more sensitive interior the creature would not retreat beneath the desert."

;;-----------------------------------------------------------------------
; "With the whip like hook-staffs Paul knew he could mount the maker's
;  high curving back."
;;-----------------------------------------------------------------------
;; something in the story is named "Paul"
(describe (add object #paul proper-name "paul"))

(m58! (class (m21 (lex human))) (member b1))
(m57! (class (m42 (lex male))) (member b1))
(m56! (object b1) (proper-name paul))

(m58! m57! m56!)
 CPU time : 0.05

* ;; something in the story ("the maker") is  maker
(describe (add member #themaker class (build lex "maker")))
```

```
(m60! (class (m59 (lex maker))) (member b2))
(m60!)
 CPU time : 0.02

* ;; something in the story is a hookstaff
(describe (add member #thehook
                  class (build lex "hook staff")))

(m62! (class (m61 (lex hook staff))) (member b3))
(m62!)
 CPU time : 0.02

* ;; something is the back of the maker
(describe (add object #theback
                  rel (build lex "back")
                  possessor *themaker))

(m63! (object b4) (possessor b2) (rel (m4 (lex back))))
(m63!)
 CPU time : 0.04

*;; the hook staff(s) is/are "whip like"
(describe (add object *thehook
                  property (build lex "whiplike")))

(m65! (object b3) (property (m64 (lex whiplike))))
(m65!)
 CPU time : 0.02

*;; the maker's back is high
(describe (add object *theback
                  property (build lex "high")))

(m67! (object b4) (property (m66 (lex high))))
(m67!)
 CPU time : 0.01

*;; the maker's back is curved
(describe (add object *theback
                  property (build lex "curved")))

(m69! (object b4) (property (m68 (lex curved))))
(m69!)
 CPU time : 0.02

*;; paul knows that he has the ability to mount the back of the maker,
;; using the hook staff as an instrument
```

```
(describe  (add agent *paul
                act (build action (build lex "know")
                           object (build
                                       agent *paul
                                       ability (build action (build lex "mount")
                                                     object *theback
                                                     instrument *thehook)))))

(m93! (class (m11 (lex animal))) (member b2))
(m92! (object1 b1) (object2 b2) (rel (m8 (lex on))))
(m91! (class (m11)) (member b2) (mode (m10 (lex possibly))))
(m90! (object1 b1) (object2 b2) (rel (m15 (lex smaller than))))
(m89! (class (m11)) (member b4))
(m88! (object1 b1) (object2 b4) (rel (m15)))
(m87! (object1 b1) (object2 b4) (rel (m8)))
(m86! (class (m11)) (member b4) (mode (m10)))
(m85! (act (m84 (action (m6 (lex ride))) (object b2))) (agent b1))
(m83! (object1 b2) (object2 b1) (rel (m13 (lex larger than))))
(m82! (object1 b4) (object2 b1) (rel (m13)))
(m81! (act (m80 (action (m6)) (object b4))) (agent b1))
(m79! (act (m78 (action (m3 (lex mount))) (object b2))) (agent b1))
(m77! (act (m70 (action (m3)) (instrument b3) (object b4))) (agent b1))
(m76! (act (m75 (action (m3)) (object b4))) (agent b1))
(m74! (act (m70)) (agent b1) (mode (m10)))
(m73!
 (act (m72 (action (m1 (lex know)))
       (object (m71! (ability (m70)) (agent b1)))))
 (agent b1))

(m93! m92! m91! m90! m89! m88! m87! m86! m85! m83! m82! m81! m79! m77! m76!
 m74! m73! m71!)
 CPU time : 0.28

* ^(
--> defineNoun "maker")
 Definition of maker:
 Possible Class Inclusions: animal,
 Actions performed on a maker: human male mount, human male ride,
 Possible Properties: larger than human male,
nil

 CPU time : 4.20
* ;;-----------------------------------------------------
;; The worm's ring segment and its forward edge

;; "the worm" is a worm
(describe (add member #worm class (build lex "worm")))
```

```
(m106! (act (m32 (action (m30 (lex tunnel))) (object (m31 (lex ground)))))
 (agent b5))
(m105! (act (m32)) (agent b5) (mode (m10 (lex possibly))))
(m104! (ability (m32)) (agent b5))
(m103! (object b5) (property (m25 (lex flexible))))
(m102! (object b5) (property (m26 (lex elongated))))
(m101! (class (m23 (lex worm))) (member b5))

(m106! m105! m104! m103! m102! m101!)
 CPU time : 0.19

*;; there's something that is the worm's ring segment
(describe (add possessor *worm
                  object #ring-segment
                  rel (build lex "ring segment")))

(m108! (part b6) (whole b5))
(m107! (object b6) (possessor b5) (rel (m51 (lex ring segment))))
(m108! m107!)
 CPU time : 0.08

*;; there's some forward edge of the ring segment
(describe (add possessor *ring-segment
                  object #forward-edge
                  rel (build lex "forward edge")))

(m110! (part b7) (whole b6))
(m109! (object b7) (possessor b6) (rel (m53 (lex forward edge))))
(m110! m109!)
 CPU time : 0.10

*;; the interior of the worm is the worm's interior
(describe (add possessor *worm
                        rel (build lex "interior")
                        object #sens-interior))

(m113! (part b8) (whole b5))
(m112! (object1 b8) (object2 b5) (rel (m45 (lex inside))))
(m111! (object b8) (possessor b5) (rel (m44 (lex interior))))
(m113! m112! m111!)
 CPU time : 0.08

*;; the interior of the worm is sensitive (more so than the rest of the worm...)
(describe (add object *sens-interior
                  property (build lex "sensitive")))
```

```
(m115! (object b8) (property (m114 (lex sensitive))))
(m115!)
 CPU time : 0.02

*;; some (a piece of) sand exists
(describe (add member #sand class (build lex "sand")))

(m116! (class (m18 (lex sand))) (member b9))
(m116!)
 CPU time : 0.03

*;; the piece (portion?) of sand is abrasive
(describe (add object *sand
                  property (build lex "abrasive")))

(m118! (object b9) (property (m117 (lex abrasive))))
(m118!)
 CPU time : 0.02

*;; admitting abrasive sand into the worm's sensitive interior
(describe (build action (build lex "admit")
                 object *sand
                 location *sens-interior) = admit-sand)

 CPU time : 0.00
* ;; "the desert" is a desert
(describe (add member #desert
                  class (build lex "desert")))

(m121! (class (m19 (lex desert))) (member b10))
(m121!)
 CPU time : 0.04

*;; "the worm" is a creature
(describe (add member *worm class (build lex "creature")))

(m122! (class (m28 (lex creature))) (member b5))
(m122!)
 CPU time : 0.02

*;; if the hook holds open the forward edge
;; then the forward edge admits the abrasive sand into the sensitive interior
;; then the creature(worm) will not retreat below the desert
(describe
(add forall ()
     ant (build act (build action (build lex "hold open")
                           object *forward-edge)
```

```
                agent *thehook)
     cq (build agent *forward-edge
               act (build action (build lex "admit")
                         object (build object1 *sand
                                       rel (build lex "inside")
                                       object2 *sens-interior)))
     cq (build min 0
               max 0
               agent *worm
               act (build action (build lex "retreat beneath")
                         object *desert))))

(m131!
 (ant
  (m124 (act (m123 (action (m47 (lex hold open))) (object b7))) (agent b3)))
 (cq
  (m130 (min 0) (max 0)
   (act (m129 (action (m128 (lex retreat beneath))) (object b10))) (agent b5))
  (m127
   (act (m126 (action (m119 (lex admit)))
         (object (m125 (object1 b9) (object2 b8) (rel (m45 (lex inside)))))))
   (agent b7))))

(m131!)
 CPU time : 5.70

*;; the hookstaff holds the forward edge of the ring segment open
(add agent *thehook
     act (build action (build lex "hold open")
                object *forward-edge))

(m134! m130! m127! m124!)
 CPU time : 0.32

* ;;  the maker and the worm are equivalent :see section 6.3
(describe (add equiv *themaker equiv *worm))

(m153! (object1 b1) (object2 b5) (rel (m15 (lex smaller than))))
(m152! (class (m11 (lex animal))) (member b5) (mode (m10 (lex possibly))))
(m151! (object1 b1) (object2 b5) (rel (m8 (lex on))))
(m150! (object1 b5) (object2 b1) (rel (m13 (lex larger than))))
(m149! (act (m32 (action (m30 (lex tunnel))) (object (m31 (lex ground)))))
 (agent b2))
(m148! (act (m32)) (agent b2) (mode (m10)))
(m147! (act (m146 (action (m3 (lex mount))) (object b5))) (agent b1))
(m145! (act (m144 (action (m6 (lex ride))) (object b5))) (agent b1))
(m143! (object b2) (property (m26 (lex elongated))))
```

```
(m142! (object b2) (property (m25 (lex flexible))))
(m141! (class (m59 (lex maker))) (member b5))
(m140! (class (m11)) (member b5))
(m139! (class (m28 (lex creature))) (member b2))
(m138! (subclass b2) (superclass b5))
(m137! (subclass b5) (superclass b2))
(m136! (subclass b5 b2))
(m135! (equiv b5 b2))
(m122! (class (m28)) (member b5))
(m104! (ability (m32)) (agent b5))
(m103! (object b5) (property (m25)))
(m102! (object b5) (property (m26)))
(m101! (class (m23 (lex worm))) (member b5))
(m98! (class (m23)) (member b2))
(m97! (ability (m32)) (agent b2))
(m93! (class (m11)) (member b2))
(m91! (class (m11)) (member b2) (mode (m10)))
(m85! (act (m84 (action (m6)) (object b2))) (agent b1))
(m79! (act (m78 (action (m3)) (object b2))) (agent b1))
(m60! (class (m59)) (member b2))

(m153! m152! m151! m150! m149! m148! m147! m145! m143! m142! m141! m140! m139!
 m138! m137! m136! m135! m122! m104! m103! m102! m101! m98! m97! m93! m91!
 m85! m79! m60!)
 CPU time : 4.57
* ; Ask Cassie what "maker" means:
^(
--> defineNoun "maker")
 Definition of maker:
 Possible Class Inclusions: animal, worm, creature,
 Possible Structure: b6, b8,
 Possible Actions: retreat beneath desert, tunnel ground,
 Possible Properties: elongated, flexible, larger than human male,
nil

 CPU time : 8.54
*

End of /home/csgrad/jpbona/class/663/projects/one/code/maker.demo demonstration.
```