DEDUCING VERB MEANINGS FROM CONTEXT

Ananthakrishnan Ugrasenan Santha

10th December 2003

CSE663: Advanced Knowledge Representation

Abstract

This paper is based on a study where a cognitive agent was made to deduce the meaning of the verb *proliferate* from its context. The study was part of the Contextual Vocabulary Acquisition (CVA) project at the University at Buffalo. The paper introduces the CVA project and then describes the study. The cognitive agent is "told" the passage containing the verb and then asked to come up with the definition. The various considerations in choosing particular representations and appropriate background knowledge is discussed. Following this, is a discussion on the result obtained and what enhancements are necessary to the verb definition algorithm to improve this result.

1. Introduction

This paper is based on a study where a computational cognitive agent deduced the meaning of a verb from its context. This study was conducted as part of the Contextual Vocabulary Acquisition (CVA) project at the University at Buffalo (Rapaport & Ehrlich 2000) (Rapaport & Kibby 2002).

CASSIE[1], Cognitive Agent of the SNePS System, is the computational cognitive agent of the CVA project. SNePS, or Semantic Network Processing System, is a propositional semantic network language which can be used as a full-fledged knowledge representation system (Shapiro & Rapaport 1987). The mind of the agent Cassie is SNePS (Shapiro & Rapaport 1995). The most important difference between SNePS and such propositional semantic networks, and those that are not, is that using SNePS, we can represent propositions as nodes in the network, which allows propositions about

1 From now on, will be called Cassie, and some human attributes implied.

propositions. This is crucial to the CVA project.

In the CVA project, a word, along with its context is “told” to Cassie in SNePSUL, or the SNePS User Language. Cassie then attempts to figure out the meaning of the target word from what she knows. The context of the word, as it is used here, stands for the whole passage where the word occurs, as well as appropriate background knowledge and rules that one may reasonably expect a person to have. This “common knowledge” is almost entirely culture and background specific. Therefore, the plan here will be to make do with the minimum possible, which will give all the necessary information for the process. The final goal of the project is to develop a curriculum with the techniques that the cognitive agent made use of to perform this, so that these techniques can be “taught” in a classroom.

2. The Verb and Passage

The verb that was used in this study was “proliferate”. The passage that it appeared in is an excerpt from a scientific discussion dealing with pathogens in food processing.

> Post-farm food processing, storage, and improper handling and cooking are major contributors to the chain of events that allows the pathogen to contaminate the product, proliferate on or in the food, and attain the large numbers that cause disease.
>
> Board on Agriculture (1999). The use of drugs in food animals: Benefits and risks. National Academy Press. [Underlining added.]

This passage was rephrased as follows with slight changes to simplify the task of representing it:

> Post-farm food processing, storage, and improper handling and cooking cause the chain of events that allows the pathogen to contaminate the product, proliferate in the food, and attain the large numbers that cause disease.

The assumption made is that this rephrased passage is identical to the original passage in terms of its supporting the contextual vocabulary acquisition process on the target word, proliferate. By substituting “are major contributors to” with “cause”, we get rid of complexity that contributes nothing to the deduction of the verb meaning. Similarly, the substitution of “on or in” with “in”.

3. Human Protocols

These protocols were obtained from human subjects who were given passages including the one on which this study is based, where the verb “proliferate” was replaced by a made-up word “taratrate”, and asked to deduce its meaning. The stress was on finding out the background information they made use of, as well as the inferences they made. The goal of using this strategy was to arrive at a set of assumptions that we may use as an extended context when representing the passage.

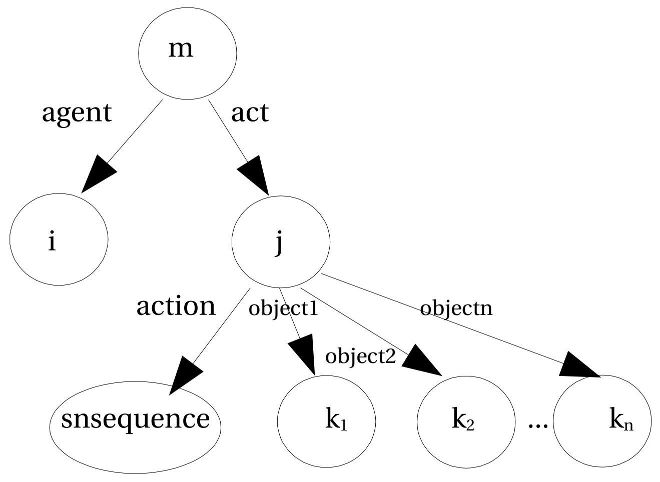
From the protocols obtained, these conclusions were drawn:

- All people noted that the verb caused the pathogen to attain large numbers.
- Pathogens are micro-organisms that grow in numbers by reproduction. (Represented here as “multiply”).

4. Case-frames: Syntax and Semantics

The only case-frame that does not appear in Scott Napieralski' scase-frame dictionary (Napieralski) is the following case-frame, which will be referred to as the snsequence case-frame, which is used in the SNeRE package. The SNeRE package or the SNePS Rational Engine is used to make Cassie perform real-world actions. “snsequence” is used in SNeRE to represent a series of actions that an agent performs. This is not a separate case-frame as such, but an extension of the normal *agent-act-action-object* case-frame. When a sequence of acts is represented, the action is always “snsequence” and instead of one object, there could be many, such as *object1*, *object2* etcetera, where each of these are *act* nodes that have an action node as well as possibly an object node. This structure denotes that the agent performs, in order, the acts that appear as objects to the action snsequence.

Syntax:



Semantics: [[m] is the proposition that agent [[i]] performs act [[j]] which consists of a sequence of actions [[k_1]] ... [[k_n]].

5. Background Knowledge

Background knowledge "told" to Cassie will be knowledge that a normal person can be expected to have. As was mentioned earlier, the strategy with regard to background knowledge is to make do with the minimum.

The first five are assumptions that most people will make, probably without even being conscious of it, when reading the passage. Assumptions such as, "there is a certain chain of events that occurs", or "there is a particular (food) product which is the same as the *food* that is mentioned later on in the passage". With the last representation, "pathogens are micro-organisms", we assume that most people know this.

```
;; There is a chain of events called #thecoe that is a member
;; of the class chain of events
(describe (assert member #thecoe
        class (build lex "chain of event") = coes))

;; There is a product called the product that is a member of the
;; class '  produts'
(describe (assert member #theproduct
      class (build lex "product") = products))

;; There is a food called #thefood that is a member of the
;; class 'foods'
(describe (assert member #thefood
      class (build lex "food") = foods))

;; "The food" and "the product" refer to the same thing here.
(describe (assert equiv #thefood equiv #theproduct))

;; There is a pathogen called #thepathogen that is a member of the
;; class "pathogens"
(describe (assert member #thepathogen
      class (build lex "pathogen") = pathogens))

;; The class "pathogens" is a sub-class of  "micro-organisms"
(describe (assert subclass *pathogens
        superclass (build lex "micro-organism") = micro-organisms))
```

6. Background Rules

Similar to the case of background knowledge, these are rules of inference that we assume any person reading the passage to have. The first rule says that micro-organisms attain large numbers by "multiplying". In other words, if a micro-organism attains large numbers by performing a certain act, then the action of that act is equivalent to the concept represented by "multiply"

```
;; "Micro-organisms attain large numbers by multiplying."
;; ->
;; If agent agentX is a member of the class "micro-organisms"
;; and an action of an act agentX performs causes agentX to
;; "attain large numbers", then, that action is equivalent to “multiply”
(describe (assert forall ($agentX $actionA)
     &ant(build member *agentX class *micro-organisms)
     &ant (build cause (build agent *agentX
                  act (build action *actionA))
          effect (build agent *agentX
                    act  (build action (build lex "attain") = attain
                    object (build lex "large numbers")) = attainlargenums))
     cq (build equiv *actionA equiv (build lex "multiply") = multiply)))
```

The second rule deals with an “snsequence” - the structure that deals with a sequence of acts. The rule essentially says that if an agent performs an act where the action is snsequence, the agent performs all the constituent acts of the snsequence. Furthermore, there is a cause-effect relation between the consecutive acts in the sequence.

```
;; If an agent agentA has performed a sequence of actions,
;; it has performed all of the actions in that sequence.
;; That is, if agentA performs an act where the action is a
;; “snsequence”, then, agentA performs all the actions that
;; make up that sequence and consecutive acts have a
;; cause-effect relation.
(describe (assert forall ($agentA $act1 $act2 $act3)
     ant (build agent *agentA act (build action snsequence
                         object1 (build act *act1)
                         object2 (build act *act2)
                         object3 (build act *act3)))
     cq  (build agent *agentA act *act1) = firstAct
     cq  (build agent *agentA act *act2) = secondAct
     cq  (build agent *agentA act *act3) = thirdAct

     cq  (build cause *firstAct effect *secondAct)

     cq  (build cause *secondAct effect *thirdAct)))
```

7. Representation of the Passage

The representation of the passage is reasonably straight-forward.

```
;; Post-farm food processing, storage, and improper handling and cooking
;; cause "the chain of events".
(describe (add  cause ((build lex "post-farm food processing") = fproc
                (build lex "storage") = storage
                (build lex "improper handling") = handling
                (build lex "cooking") = cooking)
                        effect *thecoe))

;; Assert the lex's for contaminate & proliferate
(describe (add lex "contaminate") = contaminate)
(describe (add lex "proliferate") = proliferate)
```

The last assertions say that the chain of events causes a sequence of actions by the pathogen.

```
;; "The chain of events" allows the pathogen to contaminate
;; the product, proliferate in the food, and attain the
;; large numbers that cause disease.
(describe (add agent *thepathogen
        act (build action snsequence
            object1 (build act (build action *contaminate
                        object *theproduct))
            object2 (build act (build action *proliferate))
            object3 (build act *attainlargenums))) = thesequence)

(describe (add cause *thecoe effect *thesequence))
```

8. The Result

After representing the passage, Scott Napieralski's verb definition algorithm was called. Excerpts from the result obtained is as below:

The most common type of sentences I know of that use 'proliferate'

are of the form: 'A something can proliferate.'

No superclasses were found for this verb.

A micro-organism can proliferate.

A pathogen can proliferate.

The cause-effect relation between “proliferate” and “attain large numbers” was ignored by the verb algorithm simply because that portion of the algorithm has not yet been implemented. Similarly, although Cassie comes up with the equivalence relation between “proliferate” and “multiply”, the verb algorithm, at present, ignores that also.

9. Future Work

The next step that logically follows this is to alter or enhance the verb algorithm to make use of all the inferences Cassie made about the verb. This task, for a specific verb should not be hard. On the other hand, ensuring that this works as a general rule will be challenging.

Consider the cause-effect relation that the verb-algorithm ignores. It seems straight-forward to have the algorithm find this for this particular case. In fact, this might be generalized to take care of all similar cases, where the verb in question is an action, and it causes another action, without much trouble. But, to have it take care of all conceivable cases which includes cases where the effect may not be another action and in general different from the present case looks comparatively complicated. Nevertheless, in the short and medium term, relevant work on the algorithm is probably a good idea.

Another interesting area is dealing with sequence of acts. In this case, the rule on snsequence was coded in as a background rule. But, when dealing with verbs, one may reasonably expect to deal with such sequences fairly frequently. So, it might be a logical to generalize these rules and have them as part of the verb algorithm.

10. Conclusion

The verb algorithm, in spite of its limitations, was able to come up with a reasonably usable definition for the verb. Cassie was able to make all appropriate inferences that humans appear to make with the same set of facts and rules. With further enhancements, the verb algorithm should be able to deliver a fuller definition.

11. References

1. Rapaport, William J., & Ehrlich, Karen (2000), "A Computational Theory of Vocabulary Acquisition", in Lucja M. Iwanska & Stuart C. Shapiro (eds.), Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language (Menlo Park, CA/Cambridge, MA: AAAI Press/MIT Press): 347-375.

2. Rapaport, William J., & Kibby, Michael W. (2002), "ROLE: Contextual Vocabulary Acquisition: From Algorithm to Curriculum".

3. Shapiro, Stuart C., & Rapaport, William J. (1987), "SNePS Considered as a Fully Intensional Propositional Semantic Network", in Nick Cercone & Gordon McCalla (eds.), The Knowledge Frontier: Essays in the Representation of Knowledge (New York: Springer-Verlag): 262-315.

4. Shapiro, Stuart C. and Rapaport, William J. (1995), "An Introduction to a Computational Reader of Narrative", in Judith Felson Duchan, Gail A. Bruder, & Lynne E. Hewitt (eds.), Deixis in Narrative: A Cognitive Science Perspective (Hillsdale, NJ: Lawrence Erlbaum Associates): 79-105.

5. Napieralski, Scott (2002), Dictionary of SNePS case-frames
   http://www.cse.buffalo.edu/~stn2/cva/case-frames/

6. Napieralski, Scott (200_b), Verb-definition algorithm
   file:///projects/rapaport/CVA/STN2/defun_verb.cl

APPENDIX A: The code

```
;; CSE 663: Advanced Knowledge Representation
;; Term Project
;; Contextual Vocabulary Acquisition
;; Verb: 'proliferate'
;;
;; Ananthakrishnan Ugrasenan Santha
;;

;;; ****************************
;;; PREPARE THE SNePS ENVIRONMENT
;;; ****************************

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

; Load the verb definition algorithm:
^(load "/projects/rapaport/CVA/STN2/defun_verb.cl")

; Clear the SNePS network:
(resetnet t)

; 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")

; load all pre-defined path definitions:
(intext "/projects/rapaport/CVA/mkb3.CVA/paths/paths")
;; Define path: class
(define-path class (compose class (kstar (compose subclass-
superclass))))

;;;
;;; **************************************************
;;; PASSAGE REPRESENTATION & BACKGROUND KNOWLEDGE/RULES
;;; ****************************************************
;;;

;; Passage for representation:

;; Post-farm food processing, storage, and improper handling and cooking
are
;; major contributors to the chain of events that allows the pathogen to
;; contaminate the product, proliferate on or in the food, and attain the
;; large numbers that cause disease.

;; Passage re-written:

;; Post-farm food processing, storage, and improper handling and cooking
;; cause the chain of events that allows the pathogen to
;; contaminate the product, proliferate in the food, and attain the
;; large numbers that cause disease.

;; Define relations
;; object3 is not defined in the default set.
(define     object3)

;; ******************** ;;
;; BACKGROUND KNOWLEDGE ;;
;; ******************** ;;

;; There is a chain of events called #thecoe that is a member
;; of the class chain of events
(describe (assert member #thecoe
            class (build lex "chain of event") = coes))

;; There is a product called the product that is a member of the
;; class 'products'
(describe (assert member #theproduct
          class (build lex "product") = products))

;; There is a food called #thefood that is a member of the
;; class 'foods'
(describe (assert member #thefood
          class (build lex "food") = foods))

;; "The food" and "the product" refer to the same thing here.
(describe (assert equiv #thefood equiv #theproduct))

;; There is a pathogen called #thepathogen that is a member of the
;; class 'pathogens'
(describe (assert member #thepathogen
          class (build lex "pathogen") = pathogens))

;; The class 'pathogens' is a sub-class of 'micro-organisms'
(describe (assert subclass *pathogens
            superclass (build lex "micro-organism") = micro-organisms))

;; ***************** ;;
;; BACKGROUND RULES  ;;
;; ***************** ;;
;; "Micro-organisms attain large numbers by multiplying."
;; ->
;; If agent agentX is a member of the class "micro-organisms"
;; and an action of an act agentX performs causes agentX to
;; "attain large numbers", then,
;; that action is equivalent to 'multiply'

(describe (assert forall ($agentX $actionA)
        &ant(build member *agentX class *micro-organisms)
        &ant (build cause (build agent *agentX
                        act (build action *actionA))
                effect (build agent *agentX
                           act  (build action (build lex "attain") =
attain
                           object (build lex "large numbers")) =
attainlargenums))
        cq (build equiv *actionA equiv (build lex "multiply") =
multiply)))


;; If an agent agentA has performed a sequence of actions,
;; it has performed all of the actions in that sequence.
;; That is, if agentA performs an act where the action is a
;; 'snsequence', then, agentA performs all the actions that
;; make up that sequence and consecutive acts have a
;; cause-effect relation.
(describe (assert forall ($agentA $act1 $act2 $act3)
      ant (build agent *agentA act (build action snsequence
                              object1 (build act *act1)
                              object2 (build act *act2)
                              object3 (build act *act3)))
      cq  (build agent *agentA act *act1) = firstAct
      cq  (build agent *agentA act *act2) = secondAct
      cq  (build agent *agentA act *act3) = thirdAct
      cq  (build cause *firstAct effect *secondAct)
      cq  (build cause *secondAct effect *thirdAct)))


;; *********************** ;;
;; THE PASSAGE REPRESENTED ;;
;; *********************** ;;
;; Post-farm food processing, storage, and improper handling and cooking
;; cause "the chain of events".
(describe (add    cause ((build lex "post-farm food processing") = fproc
                  (build lex "storage") = storage
                  (build lex "improper handling") = handling
                  (build lex "cooking") = cooking)
                              effect *thecoe))

;; Assert the lex's for contaminate & proliferate
(describe (add lex "contaminate") = contaminate)
(describe (add lex "proliferate") = proliferate)

;; "The chain of events" allows the pathogen to contaminate
;; the product, proliferate in the food, and attain the
;; large numbers that cause disease.
(describe (add agent *thepathogen
            act (build action snsequence
                  object1 (build act (build action *contaminate
                                     object *theproduct))

                  object2 (build act (build action *proliferate))
                  object3 (build act *attainlargenums))) = thesequence)
(describe (add cause *thecoe effect *thesequence))

;;                                                                   ;;
;; ***************************************************************** ;;
;;                                                                   ;;

;; Call the verb definition algorithm
^ (defineVerb "proliferate")
```

APPENDIX B: Annotated Run

```
Script started on Thu Dec 11 19:30:05 2003
pollux {~/AKR_CSE663/Project} > acl
International Allegro CL Enterprise Edition
6.2 [Solaris] (Oct 28, 2003 9:00)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights
Reserved.

This development copy of Allegro CL is licensed to:
   [4549] SUNY/Buffalo, N. Campus

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
optimization settings evaluate
;; (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1): :ld /projects/snwiz/bin/sneps
; Loading /projects/snwiz/bin/sneps.lisp
Loading system SNePS...10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
SNePS-2.6 [PL:0a 2002/09/30 22:37:46] loaded.
Type `(sneps)' or `(snepslog)' to get started.
cl-user(2): (sneps)


   Welcome to SNePS-2.6 [PL:0a 2002/09/30 22:37:46]

Copyright (C) 1984--2002 by Research Foundation of
State University of New York. SNePS comes with ABSOLUTELY NO WARRANTY!
Type `(copyright)' for detailed copyright information.
Type `(demo)' for a list of example applications.

   12/11/2003 19:30:43

* (demo "proliferate.demo")
```

```
File /home/csgrad/aku2/AKR_CSE663/Project/proliferate.demo is now the
source of input.


 CPU time : 0.01

* ;; CSE 663: Advanced Knowledge Representation
;; Term Project
;; Contextual Vocabulary Acquisition
;; Verb: 'proliferate'
;;
;; Ananthakrishnan Ugrasenan Santha
;;

;;; ******************************
;;; PREPARE THE SNePS ENVIRONMENT
;;; ******************************

; 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 verb definition algorithm:
^(
--> load "/projects/rapaport/CVA/STN2/defun_verb.cl")
; Loading /projects/rapaport/CVA/STN2/defun_verb.cl
t



 CPU time : 0.08

*
; Clear the SNePS network:
(resetnet t)

Net reset


 CPU time : 0.01

*
; 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

*
; 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

* ;; Define path: class
(define-path class (compose class (kstar (compose subclass-
superclass))))
class implied by the path (compose class (kstar (compose subclass-
superclass)))
class- implied by the path (compose (kstar (compose superclass-
subclass)) class-)


 CPU time : 0.00

*
;;;
;;; ***************************************************
;;; PASSAGE REPRESENTATION & BACKGROUND KNOWLEDGE/RULES
;;; ***************************************************
```

```
;;;

;; Passage for representation:

;; Post-farm food processing, storage, and improper handling and cooking
are
;; major contributors to the chain of events that allows the pathogen to
;; contaminate the product, proliferate on or in the food, and attain the
;; large numbers that cause disease.

;; Passage re-written:

;; Post-farm food processing, storage, and improper handling and cooking
;; cause the chain of events that allows the pathogen to
;; contaminate the product, proliferate in the food, and attain the
;; large numbers that cause disease.

;; Define relations
;; object3 is not defined in the default set.
(define object3)

(object3)

 CPU time : 0.00

*
;; ******************** ;;
;; BACKGROUND KNOWLEDGE ;;
;; ******************** ;;

;; There is a chain of events called #thecoe that is a member
;; of the class chain of events
(describe (assert member #thecoe
                class (build lex "chain of event") = coes))

(m2! (class (m1 (lex chain of event))) (member b1))

(m2!)

 CPU time : 0.00

*
;; There is a product called the product that is a member of the
;; class 'products'
(describe (assert member #theproduct
            class (build lex "product") = products))

(m4! (class (m3 (lex product))) (member b2))

(m4!)

 CPU time : 0.01

*
;; There is a food called #thefood that is a member of the
;; class 'foods'
```

```
(describe (assert member #thefood
            class (build lex "food") = foods))

(m6! (class (m5 (lex food))) (member b3))

(m6!)

 CPU time : 0.00

*
;; "The food" and "the product" refer to the same thing here.
(describe (assert equiv #thefood equiv #theproduct))

(m7! (equiv b5 b4))

(m7!)

 CPU time : 0.00

*
;; There is a pathogen called #thepathogen that is a member of the
;; class 'pathogens'
(describe (assert member #thepathogen
            class (build lex "pathogen") = pathogens))

(m9! (class (m8 (lex pathogen))) (member b6))

(m9!)

 CPU time : 0.03

*
;; The class 'pathogens' is a sub-class of 'micro-organisms'
(describe (assert subclass *pathogens
                superclass (build lex "micro-organism") = micro-
organisms))

(m11! (subclass (m8 (lex pathogen))) (superclass (m10 (lex micro-
organism))))

(m11!)

 CPU time : 0.00

*
;; ***************** ;;
;; BACKGROUND RULES  ;;
;; ***************** ;;

;; RULE 1
;; "Micro-organisms attain large numbers by multiplying."
;; ->
;; If agent agentX is a member of the class "micro-organisms"
;; and an action of an act agentX performs causes agentX to
;; "attain large numbers", then,
;; that action is equivalent to 'multiply'
```

```
(describe (assert forall ($agentX $actionA)
        &ant(build member *agentX class *micro-organisms)
        &ant (build cause (build agent *agentX
                        act (build action *actionA))
                effect (build agent *agentX
                           act  (build action (build lex "attain") =
attain
                           object (build lex "large numbers")) =
attainlargenums))
        cq (build equiv *actionA equiv (build lex "multiply") =
multiply)))

(m16! (forall v2 v1)
 (&ant
  (p5 (cause (p3 (act (p2 (action v2))) (agent v1)))
   (effect (p4 (act (m14 (action (m12 (lex attain))) (object (m13 (lex
large numbers))))) (agent v1))))
  (p1 (class (m10 (lex micro-organism))) (member v1)))
 (cq (p6 (equiv (m15 (lex multiply)) v2))))

(m16!)

 CPU time : 0.01

*
;; RULE 2
;; If an agent agentA has performed a sequence of actions,
;; it has performed all of the actions in that sequence.
;; That is, if agentA performs an act where the action is a
;; 'snsequence', then, agentA performs all the actions that
;; make up that sequence and consecutive acts have a
;; cause-effect relation.
(describe (assert forall ($agentA $act1 $act2 $act3)
        ant (build agent *agentA act (build action snsequence
                                        object1 (build act *act1)
                                        object2 (build act *act2)
                                        object3 (build act *act3)))
        cq  (build agent *agentA act *act1) = firstAct
        cq  (build agent *agentA act *act2) = secondAct
        cq  (build agent *agentA act *act3) = thirdAct
        cq  (build cause *firstAct effect *secondAct)
        cq  (build cause *secondAct effect *thirdAct)))

(m17! (forall v6 v5 v4 v3)
 (ant
  (p11
   (act (p10 (action snsequence) (object1 (p7 (act v4))) (object2 (p8
(act v5))) (object3 (p9 (act v6)))))
   (agent v3)))
 (cq (p16 (cause (p13 (act v5) (agent v3))) (effect (p14 (act v6) (agent
v3))))
  (p15 (cause (p12 (act v4) (agent v3))) (effect (p13))) (p14) (p13)
(p12)))

(m17!)
```

```
 CPU time : 0.01

*

;; ********************** ;;
;; THE PASSAGE REPRESENTED ;;
;; ********************** ;;
;; Post-farm food processing, storage, and improper handling and cooking
;; cause "the chain of events".
(describe (add  cause ((build lex "post-farm food processing") = fproc
                       (build lex "storage") = storage
                       (build lex "improper handling") = handling
                       (build lex "cooking") = cooking)
                                      effect *thecoe))

(m22!
 (cause (m21 (lex cooking)) cooking handling (m20 (lex improper
handling)) (m19 (lex storage)) storage fproc
  = (m18 (lex post-farm food processing)))
 (effect b1))

(m22!)

 CPU time : 0.00

*
;; Assert the lex's for contaminate & proliferate
(describe (add lex "contaminate") = contaminate)

(m23! (lex contaminate))

(m23!)

 CPU time : 0.01

* (describe (add lex "proliferate") = proliferate)

(m24! (lex proliferate))

(m24!)

 CPU time : 0.00

*
;; "The chain of events" allows the pathogen to contaminate
;; the product, proliferate in the food, and attain the
;; large numbers that cause disease.
(describe (add agent *thepathogen
               act (build action snsequence
                       object1 (build act (build action *contaminate
                                           object *theproduct))
                       object2 (build act (build action *proliferate))
                       object3 (build act *attainlargenums))) =
thesequence)
```

```
;; ####
;; Cassie finds the equivalence relation between proliferate and multiply
;; as well as the cause effect relation between ptoliferation and
;; attaining large numbers. That is, both RULE1 and RULE2 fired.
;; ####
(m45! (equiv (m24! (lex proliferate)) (m15 (lex multiply))))
(m44! (class (m10 (lex micro-organism))) (member b6))
(m43! (act (m42 (action (m23! (lex contaminate))))) (agent b6))
(m41! (act (m40 (action (m12 (lex attain))))) (agent b6))
(m39! (cause (m38! (act (m25 (action (m23!)) (object b5))) (agent b6)))
 (effect (m35! (act (m27 (action (m24!)))) (agent b6))))
(m37! (cause (m35!))
 (effect (m36! (act (m14 (action (m12)) (object (m13 (lex large
numbers))))) (agent b6))))
(m34!
 (act (m30 (action snsequence) (object1 (m26! (act (m25)))) (object2
(m28! (act (m27))))
       (object3 (m29! (act (m14)))))))
(m33! (act (m32 (action snsequence))) (agent b6))
(m31! (act (m30)) (agent b6))

(m45! m44! m43! m41! m39! m38! m37! m36! m35! m34! m33! m31! m29! m28!
m26!)

 CPU time : 0.03

*
(describe (add cause *thecoe effect *thesequence))

(m46! (cause b1)
 (effect (m45! (equiv (m24! (lex proliferate)) (m15 (lex multiply))))
  (m44! (class (m10 (lex micro-organism))) (member b6))
  (m43! (act (m42 (action (m23! (lex contaminate))))) (agent b6))
  (m41! (act (m40 (action (m12 (lex attain))))) (agent b6))
  (m39! (cause (m38! (act (m25 (action (m23!)) (object b5))) (agent b6)))
   (effect (m35! (act (m27 (action (m24!)))) (agent b6))))
  (m38!)
  (m37! (cause (m35!))
   (effect (m36! (act (m14 (action (m12)) (object (m13 (lex large
numbers))))) (agent b6))))
  (m36!) (m35!)
  (m34!
   (act (m30 (action snsequence) (object1 (m26! (act (m25)))) (object2
(m28! (act (m27))))
         (object3 (m29! (act (m14)))))))
  (m33! (act (m32 (action snsequence))) (agent b6)) (m31! (act (m30))
(agent b6)) (m29!) (m28!) (m26!)))

(m46!)

 CPU time : 0.01

*
;;
;;
;;
```

```
*************************************************************************
************** ;;
;;
;;

;; Call the verb definition algorithm
^(
--> defineVerb "proliferate")
"You want me to define the verb 'proliferate'.

I'll start by looking at the predicate stucture of the sentences I know
that use 'proliferate'.  Here is what I know:

The most common type of sentences I know of that use 'proliferate' are of
the form:
     'A something can proliferate.'



No superclasses were found for this verb.

 Sorting from the most common predicate case to the least common here is
what I know.  I will first attempt to unify the components of the
sentences that use the verb giving a generalizaiton based on my
background knowledge:

A micro-organism can proliferate.


Now, looking from the bottom up I want to get a sense of the categories
that most of the agents, objects and indirect objects belong to.  This is
different from looking for the most unified case.  Instead I am looking
for the classes that contain approximately half of the agents, objects
and indirect objects.  This is an attempt at generalization but from
another approach.

A pathogen can proliferate.


"
 CPU time : 0.05

*

End of /home/csgrad/aku2/AKR_CSE663/Project/proliferate.demo
demonstration.
 CPU time : 0.28

* (lisp)
"End of SNePS"
cl-user(3): :exit
; Exiting Lisp
pollux {~/AKR_CSE663/Project} > exit
exit

script done on Thu Dec 11 19:31:17 2003
```