Computationally Defining ‘Ceilidh’ from Contextual Cues

Jun Xu

April 30, 2004

CSE 740: Seminar on Contextual Vocabulary Acquisition

Abstract

CVA project is a SNePS based computational program for computers to understand the meaning of a word from the context. A SNePS based agent Cassie acts as a reader. After being given the background knowledge and read a passage with an unknown word, Cassie can give a definition of the unknown word. In this paper, Cassie is told the background knowledge and read a passage with word ‘ceilidh’, and is asked to define the word ‘ceilidh’. The representation of the background knowledge and the passage is described in the paper. The revision of the noun algorithm is discussed. New functions were added to the original noun algorithm to give a more complete definition of ‘ceilidh’. Finally, the result is given, and the problems and future work are discussed.

1. Introduction

Contextual vocabulary acquisition (CVA) is active, deliberate acquisition of word meanings from text by reasoning from contextual clues, prior knowledge, language knowledge, and hypotheses developed from prior encounters with the word, but without external sources of help such as dictionaries or people (Rapaport and Kibby, 2002). When people are reading some text containing unknown words, most of people are unwilling to look up a dictionary. From the contextual clues and their background knowledge, they could guess the meaning of the word. The meaning may be incorrect. When they see the word again, they may have the chance to correct their definition about this word.

CVA project is a computational program for computers to understand the meaning of a word from the context. The CVA project is based on the SNePS, a semantic network knowledge representation and reasoning system. A SNePS based agent Cassie acts as a reader in the project. After being given the background knowledge and read a passage with an unknown word, Cassie can give the definition of the unknown word by certain computational algorithm. The background knowledge and the passage are represented as semantic network.

The purpose of this paper is computationally defining the word 'ceilidh' from a passage. First, human subjects were interviewed to get the speak-loud protocols. Then tell Cassie the necessary background knowledge and let she read the passage. All the background knowledge and the passage are told by SNePSUL (SNePS User Language) representation. Finally ask Cassie to give the definition of 'ceilidh' by noun algorithm.

2. SNePS and CVA Project

2.1 Introduction to SNePS

SNePS, the Semantic Network Processing System, is an intensional, propositional, semantic-network knowledge representation system that is used for research in artificial intelligence and in cognitive science (Shapiro and Rapaport, 1995). It uses an intensional propositional semantic network to represent knowledge, and supports multiple representation of what could be one physical object. So it should be able to represent anything and everything expressible in natural-language, it should be able to represent generic, as well as specific information; it should be able to use generic and specific information to reason and infer information implied by what it has been told (Shapiro and Rapaport, 1992).

SNePS is a programming language whose primary data structure is a semantic network. A Semantic network is a labeled, directed graph whose nodes represent entities and whose arcs represent binary relations between entities (Shapiro and Rapaport, 1995). It is a propositional semantic network, which means that all information, including propositions, "facts", etc, is represented by nodes. Base nodes are distinguished by having no arcs coming from them. A base node is assumed to represent some entity—individual, object, class, property, etc. No two nodes represent the same, identical entity. Molecular nodes may represent propositions. Arcs represent the relations between nodes. SNePS is an intentional knowledge-representation system, which means it supports multiple representation of what could be one physical object (Shapiro and Rapaport, 1995). Thus two intentional entities might be equivalent for some propose without being identical (Shapiro and Rapaport, 1987).

Users can interact with SNePS with a variety of interface language. In this paper, SNePSUL, a LISP-like SNePS user language, is used to represent all the knowledge. Cassie is the computational cognitive agent used to interact with SNePS. Cassie reads the passage, and infers the unknown information from the background knowledge and what she is told. SNePS nodes represent Cassie's thoughts.

2.2 CVA project

CVA project is a computational program, which can deduce the meaning of the unknown word from the context. It is "a natural-language-understanding systems that can automatically acquire new vocabulary by determining from context the meaning of words that are unknown,

misunderstood, or used in a new sense, where ‘context’ includes surrounding text, grammatical information, and background knowledge, but no external sources such as dictionary or human” (Rapaport and Ehrlich, 2000).

CVA project is built on SNePS. Cassie is the read agent. Cassie’s background knowledge is represented in a knowledge base. Cassie’s input consists of information from the text being read and questions that trigger a deductive search of the knowledge base. Output consists of a report of Cassie’s current definition of the word, or answers to other queries (Rapaport and Ehrlich, 2000).

CVA project uses different algorithms to define different kinds of words. Currently there are noun algorithm and verb algorithm. The algorithms hypothesize and revise meanings for nouns and verbs that are unknown, misunderstood, or being used in a new way. The algorithms deductively search the network for information appropriate to dictionary-like definition (Rapaport and Ehrlich, 2000). In this paper, noun algorithm is used to define the word ‘ceilid’

3. The Passage and Verbal Protocols

The resource passage was taken from the paper of Sternberg and Powell (1983):

> “Two ill-dressed people ... sat around a fire where the common meal was almost ready. The mother ... peered at her son through the oam of the bubbling stew. It had been a long time since his last *ceilidh* and Tobar had changed greatly. ... As they ate, Tobar told of his past year .... Then all too soon, their brief *ceilidh* over, Tobar walked over to touch his mother's arm and quickly left.”

The passage was given as a example of how to learn from external context. The unknown word is 'ceilidh'. In the dictionary (http://www.dictionary.com), the definition is given as:

"**cei·lidh** *n.* An Irish or Scottish social gathering with traditional music, dancing, and storytelling."

What we need to do is to give a dictionary-like definition of the unknown word 'ceilidh' by using the contextual cues in this passage. First, we have to know how humans get the meaning of ceilidh from the context. Several human subjects were interviewed. They were asked to read the passage and give a definition for the word 'ceilidh'. Here are some results from human subjects:

HS1: "ceilidh is a ceremony during the meal. It doesn't happen often, probably once a year."
HS2: "ceilidh is a kind of celebration happens once a year, and it is probably outdoor."
HS3: "ceilidh is a gathering during which people eat and talk. It is short, and there is long time between two ceilidhs, probably one year."

We can notice that every one of them gave the definition that ceilidh is a kind of event, and the frequency of ceilidh is not often, probably once a year. In the passage, it does say: "It had been a long time since his last ceilidh." The interesting thing is that there is nothing in the passage talks about the last ceilidh was one year before, but everyone who read this passage thought it is once a year. When asked, they told that they got this from "Tobar told of his past year". Tobar told mother of his past year, which implicates that Tobar and mother didn't see each other during the past year. From "Tobar had changed greatly since last ceilidh", we can infer

that mother hadn't seen Tobar since last ceilidh. So we can conclude that last ceilidh was one year before. One of the human subjects also said that ceilidh is outdoor, because people sit around the fire, and usually the fire is outdoor.

Sternberg and Powell (1983) described eight external contextual cues that can be used to acquire the meaning of unknown words. In this passage, the most prominent contextual cue is the temporal cue, such as "long time", "one year", "all too soon". From the verbal protocols, we can see that every human subject used these temporal cues. There are also spatial cues and stative description cues in this passage. Our goal is letting Cassie read this passage, and use these contextual cues to give a definition of the word 'ceilidh'.

4. SNePS Representation

4.1 SNePS Case Frame: Syntax and Semantics

The standard SNePS case frames, which are recognized by CVA noun algorithm, are used in this project. They are:

- member/class
- subclass/supclass
- object/property
- object/proper-name
- object/location
- lex
- object1/rel/object2
- agent/act/action

- agent/act/action/object
- object/rel/possessor

The syntax and semantics of these case frames can be found at http://www.cse.buffalo.edu/~rapaport/CVA/CaseFrames/case-frames.

Since there is lots of temporal information in this passage that Cassie needs to know, the temporal case frames introduced by Almeida (1995) are used to represent temporal information. Every event has an *attachment time*, and some nodes that represent time intervals can be the subintervals of this attachment time. Almeida used a variable node called *now* to represent the current time interval. During the reading on the narrative-line, the variable *now* will move to the next current position of the story present (Almeida, 1995). The case frames are:

- before/after
- before/after/duration
- subinterval/superinterval
- initial-subinterval/superinterval
- final-subinterval/superinterval

If x1 … x5, y1 … y5, and z2 are individual nodes, and m1 … m5 are not previously used, then each of the structure in Figure1 is a network, and m1 … m5 are preposition nodes.

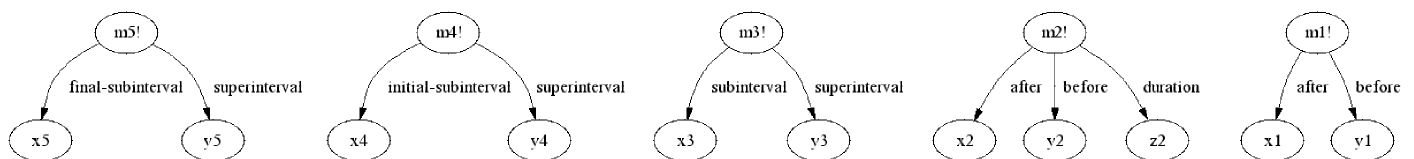

Figure1 Syntax of Temporal Case Frames

Semantics:

- [[m1]] is the proposition that [[x1]] is after [[y1]].
- [[m2]] is the proposition that [[x2]] is after [[y2]] for [[z2]] time duration.
- [[m3]] is the proposition that [[x3]] is a subinterval of [[y3]].
- [[m4]] is the proposition that [[x4]] is the initial subinterval of [[y4]].
- [[m5]] is the proposition that [[x5]] is the final subinterval of [[y5]].

In addition, some other "non-standard" case frames are used to represent the sentences in the passage. The syntax and semantics of each case frame will be introduced individually as follow, in which i, j, k, l and t are individual nodes, and m6 … m16 are structured proposition nodes.

- object/location/time

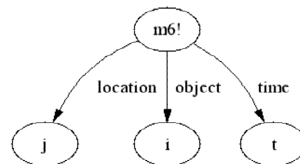

Figure 2 Syntax of object/location/time

Semantics:

[[m6]] is the proposition that [[i]] is located at [[j]] at time [[t]].

- object/property/time

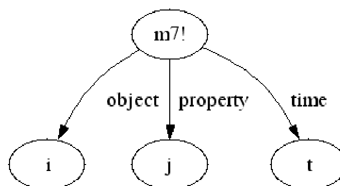

Figure 3 Syntax of object/property/time

Semantics:

[[m7]] is the proposition that at time [[t]], [[I]] has the property [[j]].

- object/time

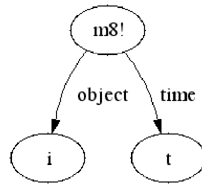

Figure 4 Syntax of object/time

Semantics:

[[m8]] is the proposition that [[i]] exists at time [[t]].

- agent/act/action/time

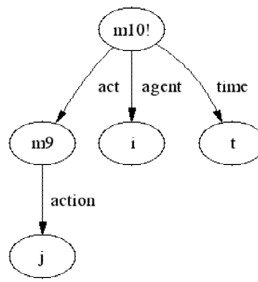

Figure 5 Syntax of agent/act/action/time

Semantics:

[[m10]] is the proposition that agent [[i]] performs action [[j]] at time [[t]].

- agent/act/action/location/time

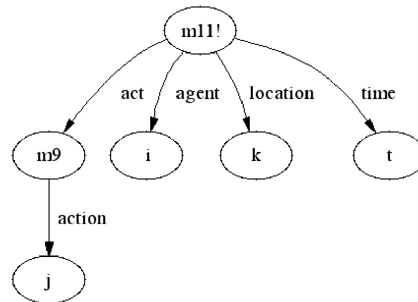

Figure 6 Syntax of agent/act/action/location/time

Semantics:

[[m11]] is the proposition that agent [[i]] performs action [[j]] at location [[k]] at time [[t]].

- agent/act/action/object/time

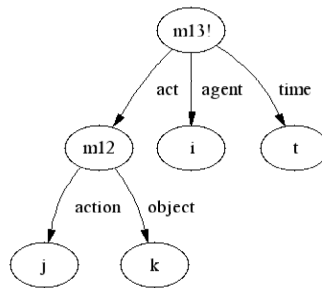

Figure 7 Syntax of agent/act/action/object/time

Semantics:

[[m13]] is the proposition that agent [[i]] performs action [[j]] with respect to object [[k]] at time [[t]].

- agent/act/action/object/to/time

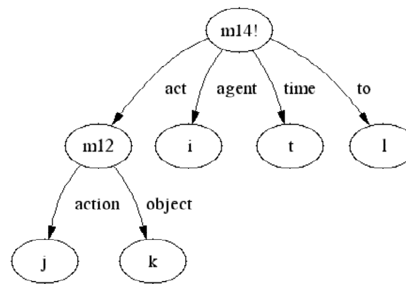

Figure 8 Syntax of agent/act/action/object/to/time

Semantics:

[[m14]] is the proposition that agent [[i]] performs action [[j]] with respect to object [[k]] to [[l]] at time [[t]].

- last/next

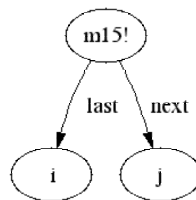

Figure 9 Syntax of last/next

Semantics:

[[m15]] is the proposition that [[i]] is the last one of [[j]].

- equiv/equiv

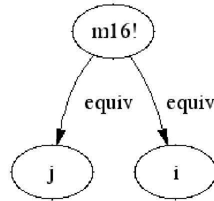

Figure 10 Syntax of equiv/equiv

Semantics:

[[m16]] is the proposition that [[i]] and [[j]] are equivalent.

4.2 Background Knowledge Representation

Before Cassie reads the passage and gives the definition of 'ceilidh', she should know some background knowledge. A knowledge base is built for Cassie. In the knowledge base, there are basic knowledge and some rules that Cassie will use to infer the meaning of the word. Basic knowledge is the facts about the real world. In this project, Cassie only uses very small amount of basic knowledge. They are:

```
;Gathering is subclass of events.
(describe (assert subclass (build lex gathering) superclass (build lex
event)))

;meal is subclass of food.
(describe (assert subclass (build lex meal) superclass (build lex food)))
```

In the second part of the knowledge base, there are a rules that Cassie needs to use when she infers the meaning of 'ceilidh'. These rules are based on the verbal protocols. A complete SNePSUL representation of these rules is given in Appendix A. The following is the description and of the rules. Figures 11 to Figure 38 show the diagrams of these rules.

The first two background rules tell Cassie some actual information (Figure 11, Figure 12).

RULE1: Ill-deressed people are people.

RULE2: Common meals are meals.

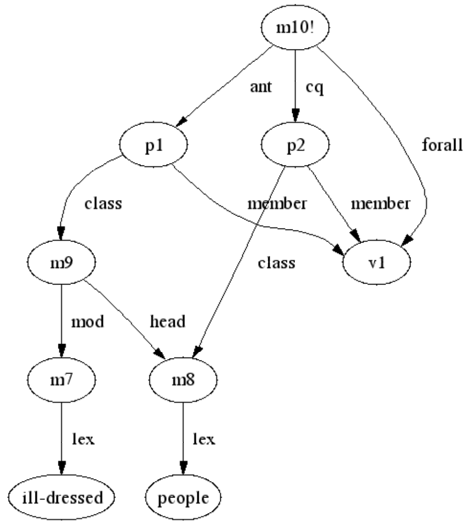

Figure 11 "Ill-deressed people are people."

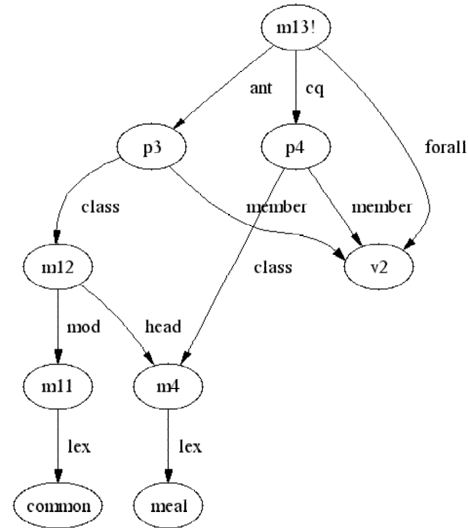

Figure 12 "Common meals are meals."

The next few background rules are about the location (Figure 13 - 15).

RULE3: If something is a member of class fire, then it is probably outdoor.

RULE4: If something is around another thing that is probably outdoor, then this thing is probably outdoor.

RULE5: If someone sits at some place that is probably outdoor at a time, then this person is probably outdoor at this time.

From these rules, Cassie can infer that people sit around fire are probably outdoor. If we want Cassie to infer that ceilidh happens outdoor, we have to tell Cassie more rules, such as "if people are outdoor when they have an event, then this event happens outdoor." But I didn't tell Cessie about this. I will explain the reason in the result and discussion section.

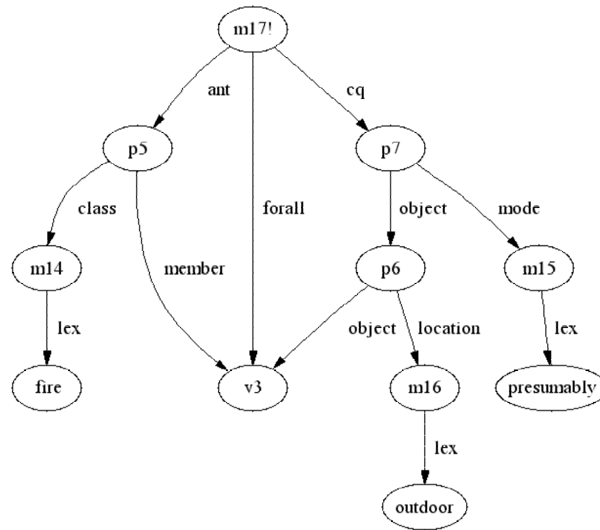

Figure 13 "If something is a member of class fire, then it is probably outdoor."

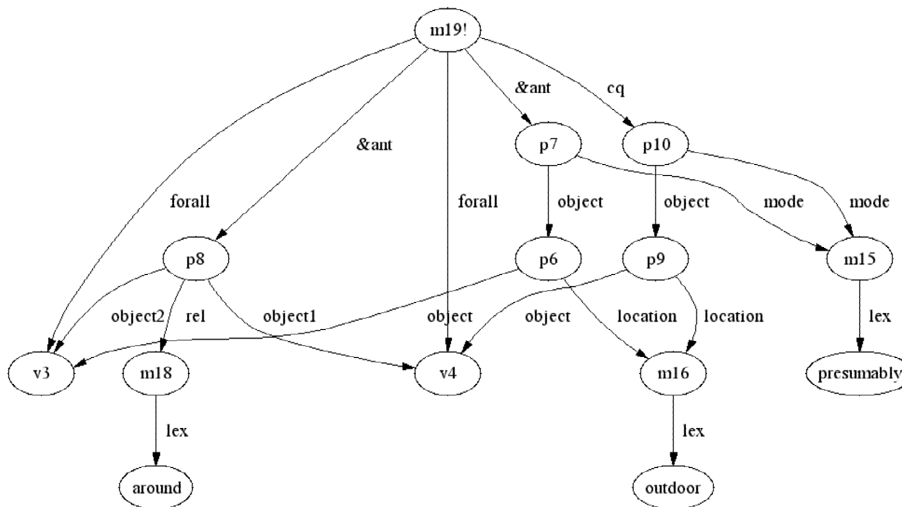

Figure 14 "If something is around another thing that is probably outdoor, then this thing is probably outdoor."

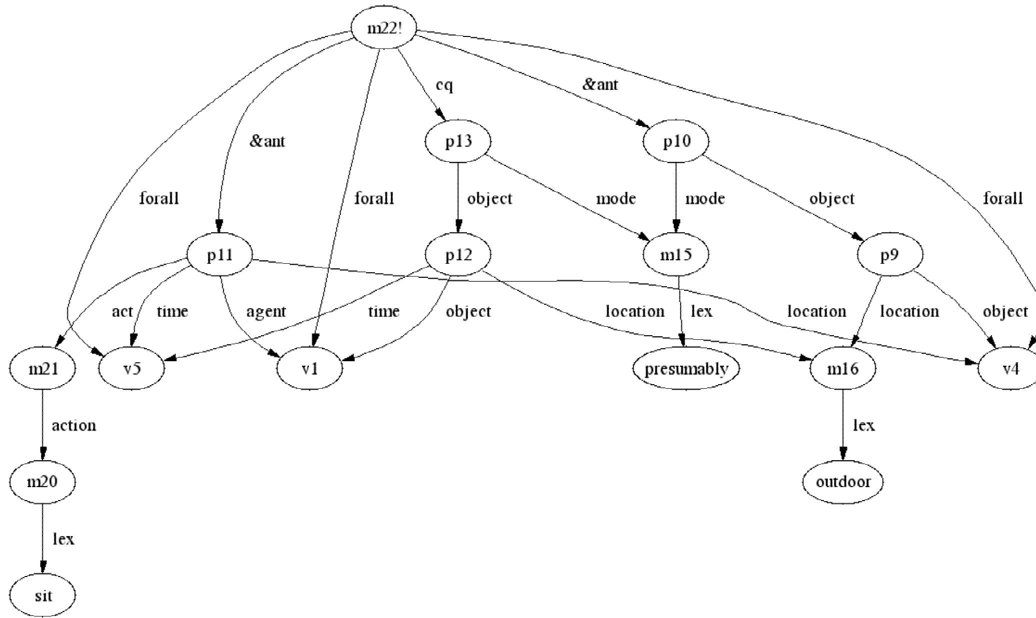

Figure 15 "If someone sits at some place that is probably outdoor at a time, then this person is probably outdoor at this time."

The following rules are for reasoning the temporal information (Figure 16 - 20). With these rules, Cassie can infer the time sequence related to one thing.

RULE6: If something is a member of meal, someone is a member of person eats it, then the person eats it shortly after it is almost ready.

RULE7: If something is over at a time, then this time is its end time.

RULE8: If something has an end time, then it has an initial time that is before the end time.

RULE9: If a time period has a subinterval, an initial-subinterval, then the initial subinterval is before the subinterval.

RULE10: If a time period has a subinterval, an final-subinterval, then the final subinterval is after the subinterval.

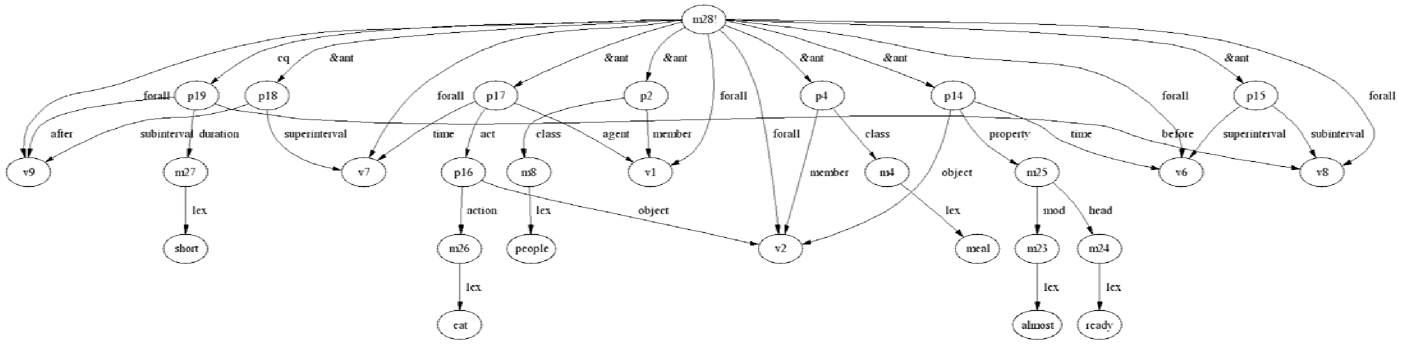

Figure 16 “If something is a member of meal, someone is a member of person eats it, then the person eats it shortly after it is almost ready.”

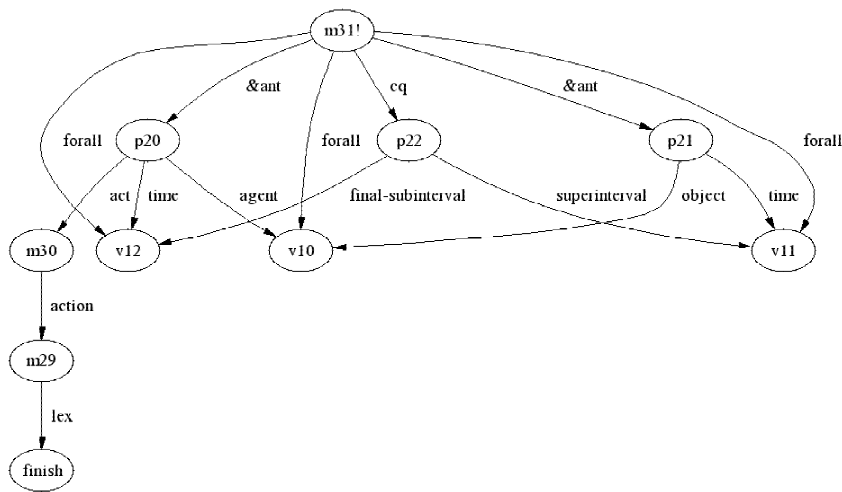

Figure 17 “If something is over at a time, then this time is its end time.”

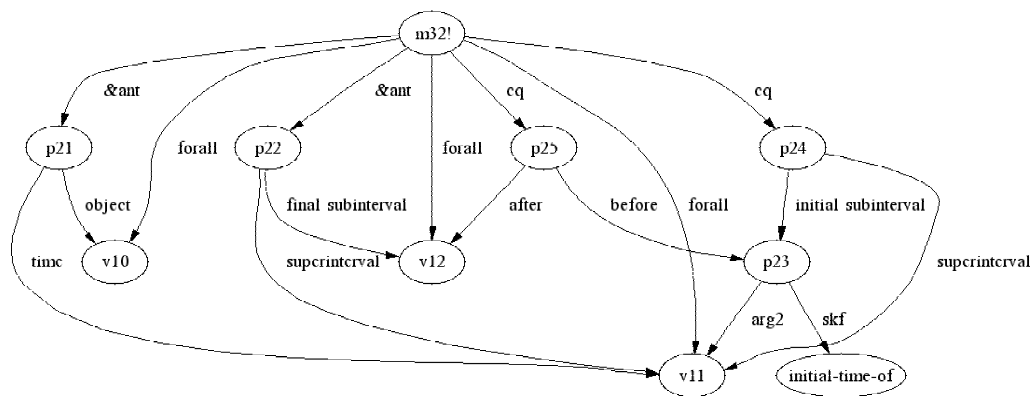

Figure 18 “If something has an end time, then it has an initial time that is before the end time.”

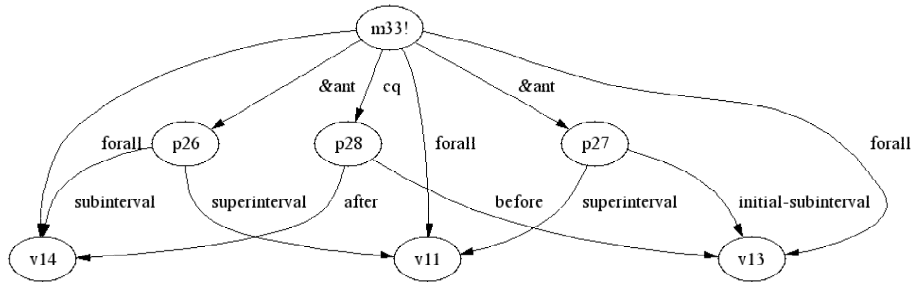

Figure 19 "If a time period has a subinterval, an initial-subinterval, then the initial subinterval is before the subinterval."

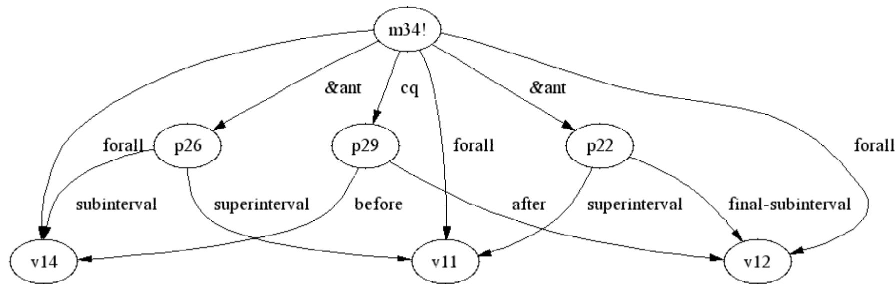

Figure 20 "If a time period has a subinterval, an final-subinterval, then the final subinterval is after the subinterval."

The next background rules tell the class inclusion information of 'event' and 'gathering', and the characteristics of events (Figure 21 – 24). With these rules, Cassie can infer that ceilidh is probably the subclass of event and gathering.

RULE11: If something has an initial time and an end time, and the initial time is before the end time, then this thing is a member of event.

RULE12: If something is an event and part of another unknown class, then probably the class is a subclass of event.

RULE13: If something is probably a gathering and part of another unknown class, then probably the class is a subclass of gathering.

RULE14: If something is an event, and it is brief, then the duration between its initial time and final time is short.

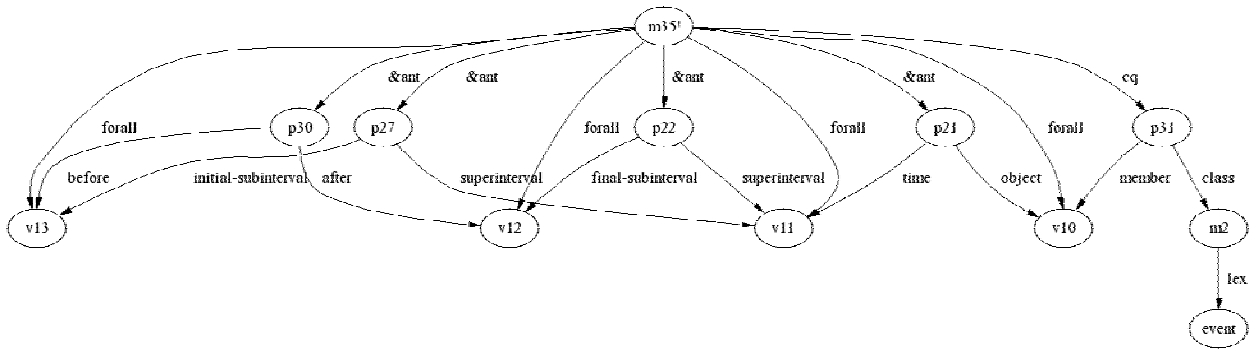

Figure 21 “If something has an initial time and an end time, and the initial time is before the end time, then this thing is a member of event.”

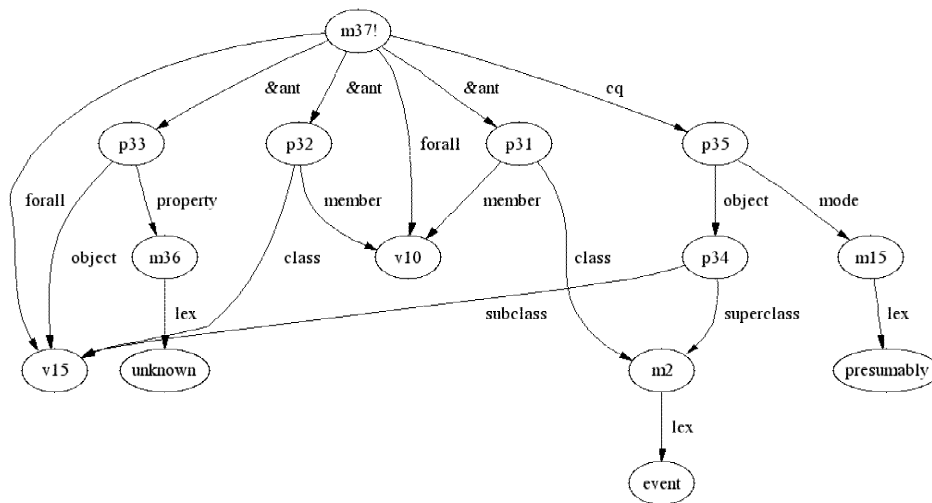

Figure 22 “If something is an event and part of another unknown class, then probably the class is a subclass of event.”

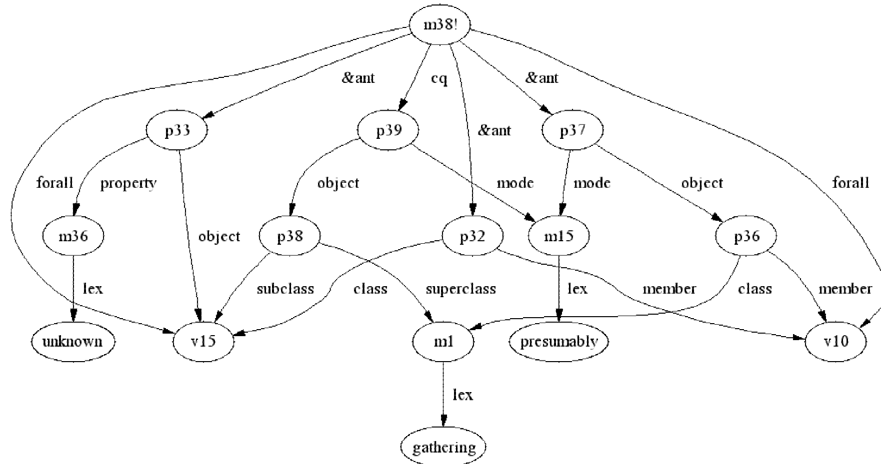

Figure 23 “If something is probably a gathering and part of another unknown class, then probably the class is a subclass of gathering.”

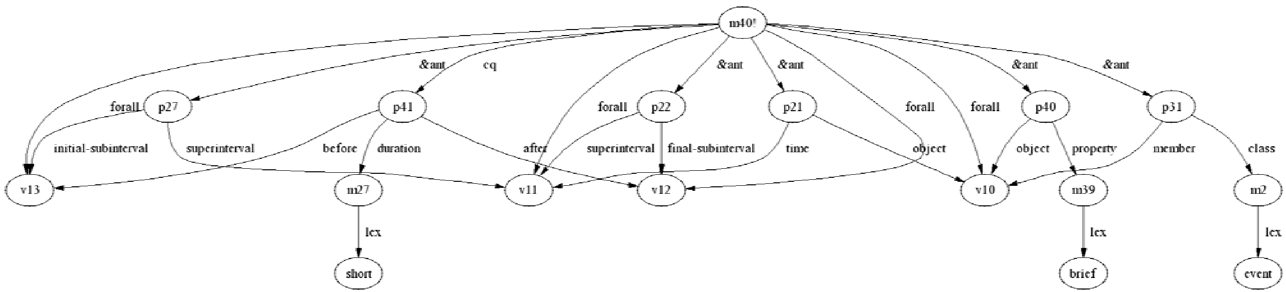

Figure 24 “If something is an event, and it is brief, then the duration between its initial time and final time is short.”

There are many time intervals need to be represented in this passage. The sequences and durations between some of the intervals are explicit in this passage, but that between other intervals are not. The next rules are used to infer the implicit time sequences and durations (Figure 25 – 30).

RULE15: If time1 is before time2 and time2 is before time3, then time1 is before time3.

RULE16: If time1 is long time before time2, and time2 is before time3, then time1 is long time before time3.

RULE17: If time1 is long time before time3, and time2 is short before time3, then time1 is long time before time2.

RULE18: If time1 is before time2, time2 is before time3, and time1 is short before time3, then time1 is short before time2 and time2 is short before time3.

RULE19: If time1 is one year before time3, and time2 is short before time3, then time1 is one year before time2.

RULE20: If there is any time interval between the initial subinterval and the final subinterval of something, then this time interval is the subinterval of this thing.

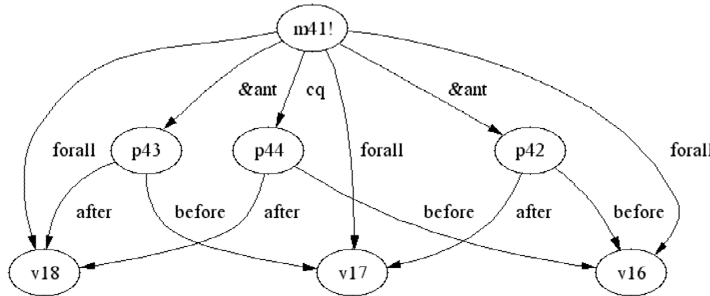

Figure 25 “If time1 is before time2 and time2 is before time3, then time1 is before time3.”

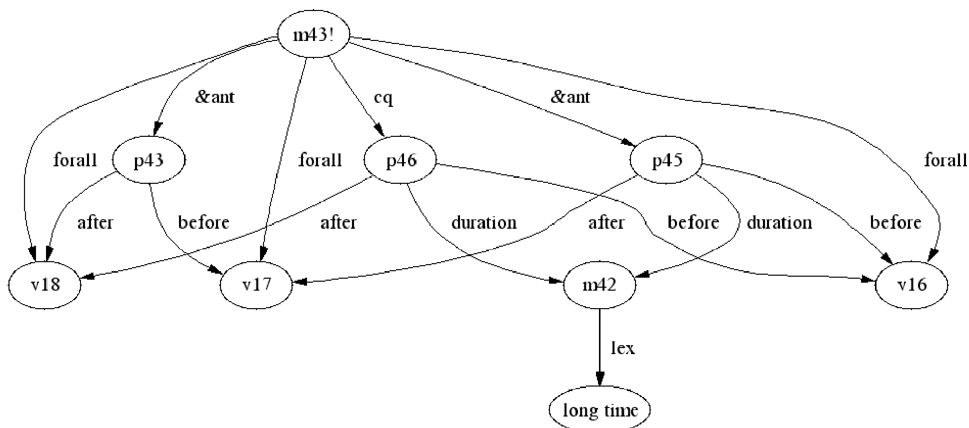

Figure 26 “If time1 is long time before time2, and time2 is before time3, then time1 is long time before time3.”

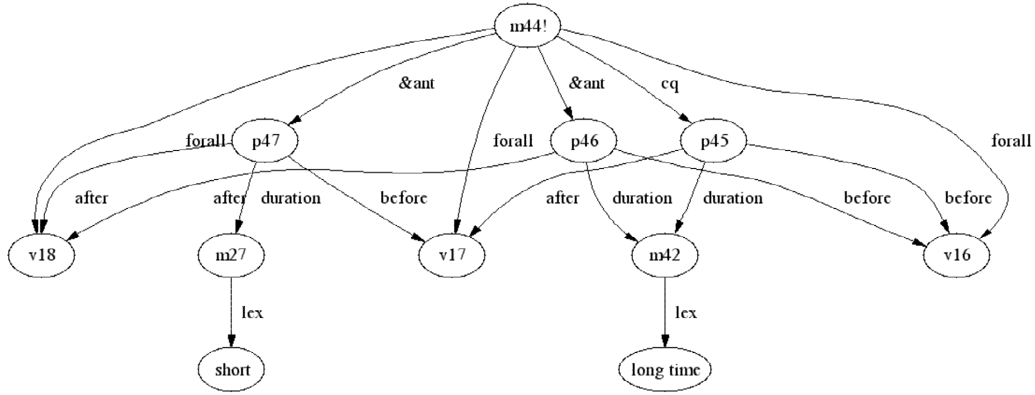

Figure 27 “If time1 is long time before time3, and time2 is short before time3, then time1 is long time before time2.”

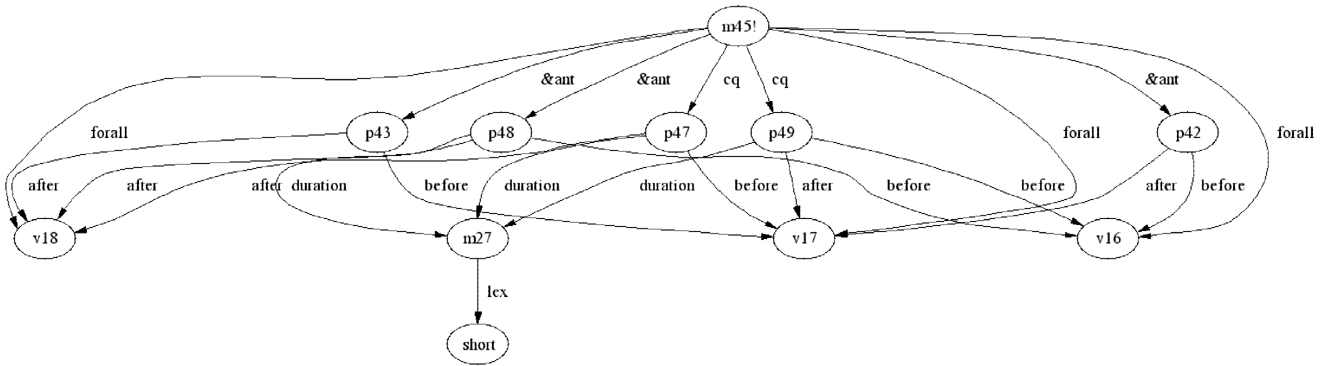

Figure 28 “If time1 is before time2, time2 is before time3, and time1 is short before time3, then time1 is short before time2 and time2 is short before time3.”

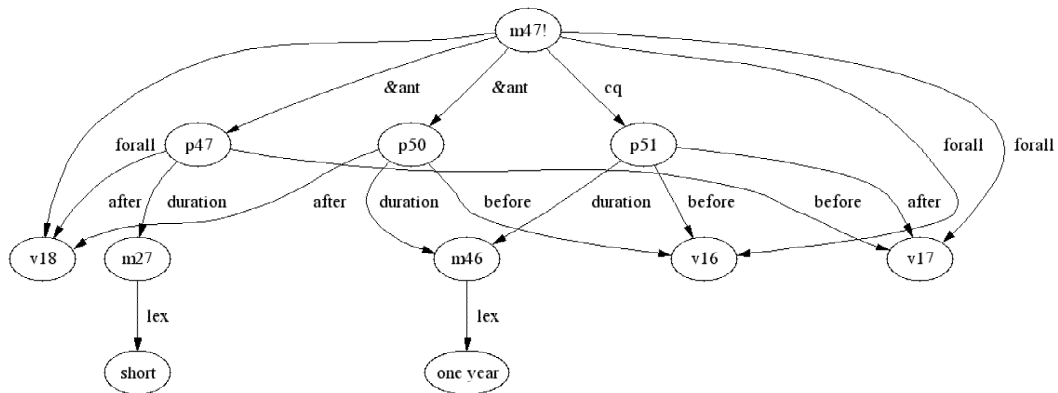

Figure 29 “If time1 is one year before time3, and time2 is short before time3, then time1 is one year before time2.”

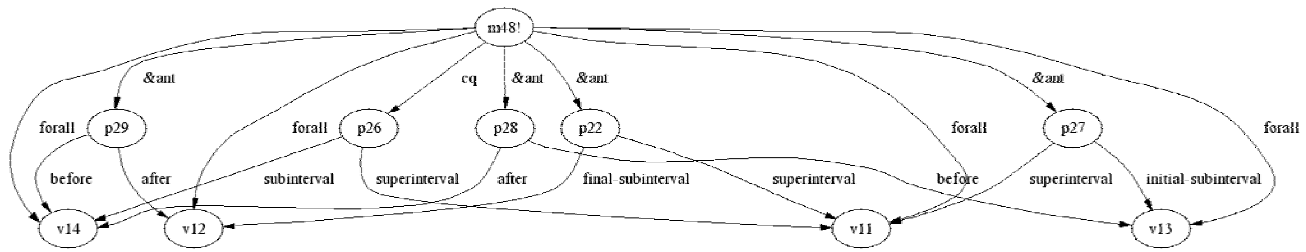

Figure 30 "If there is any time interval between the initial subinterval and the final subinterval of something, then this time interval is the subinterval of this thing."

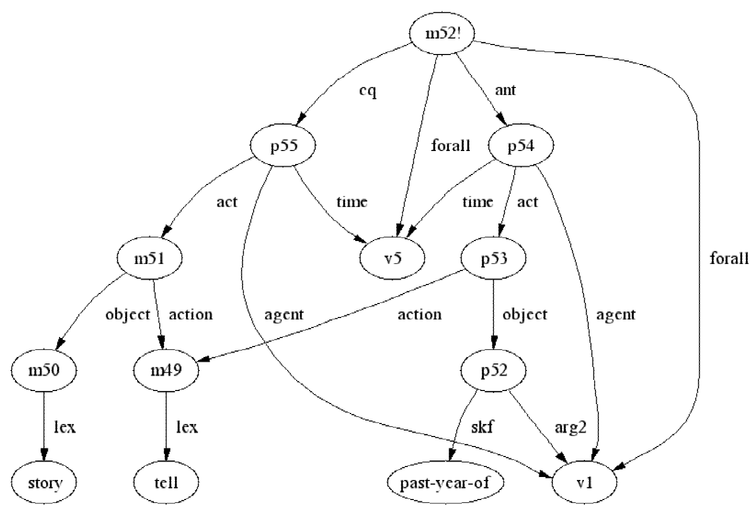

Figure 31 "If some person tells his past year at a time, he tells story at this time."

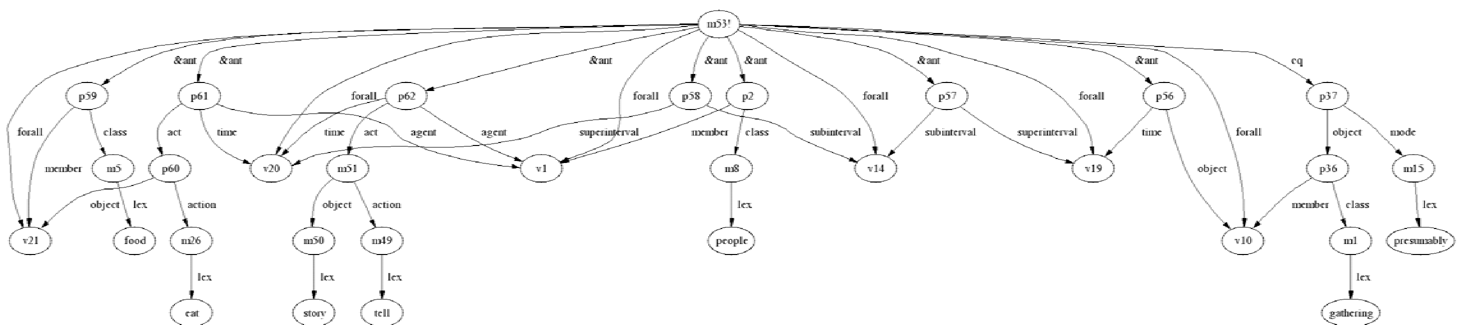

Figure 32 "If there is subinterval of something, and during this interval people eat and tell story, then probably this thing is a gathering."

These two rules tell Cassie some information about gathering (Figure 31, Figure 32).

RULE21: If some person tells his past year at a time, he tells story at this time.

RULE22: If there is subinterval of something, and during this interval people eat and tell story, then probably this thing is a gathering.

From all the background rules described above, Cassie can infer most information about ceilidh that people can acquire from the ceilidh passage, except that it is probably once a year. It is so implicit in this passage. We have already explained how people deduce this in the verbal protocols section. Here we have to tell Cassie the complicated rules that people used.

First, we tell Cassie that from some information she can infer that people don't see each other during a certain time period, so they have one meeting before this period of time, and they have their next meeting after this period of time (Figure 33, Figure 34). We use "last/next" case frame to represent that there are two things of same kind, one is before another and there is no other one in between.

RULE23: If some person tells another person of his past year at a time, then they meet at this time, and they meet at another time whose final interval is one year before this time.

RULE24: If at one time some person found another person changed greatly, and the change happened during a period of time, then they had last meeting which finished at the time when the change began, and they had this meeting which happened at this time.

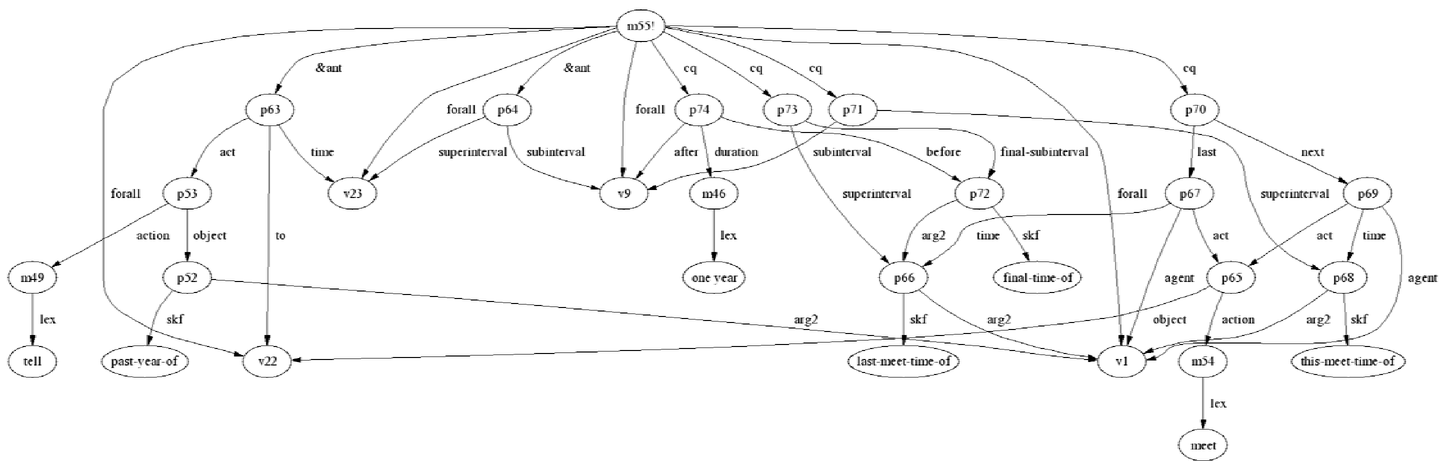

Figure 33 "If some person tells another person of his past year at a time, then they meet at this time, and they meet at another time whose final interval is one year before this time."

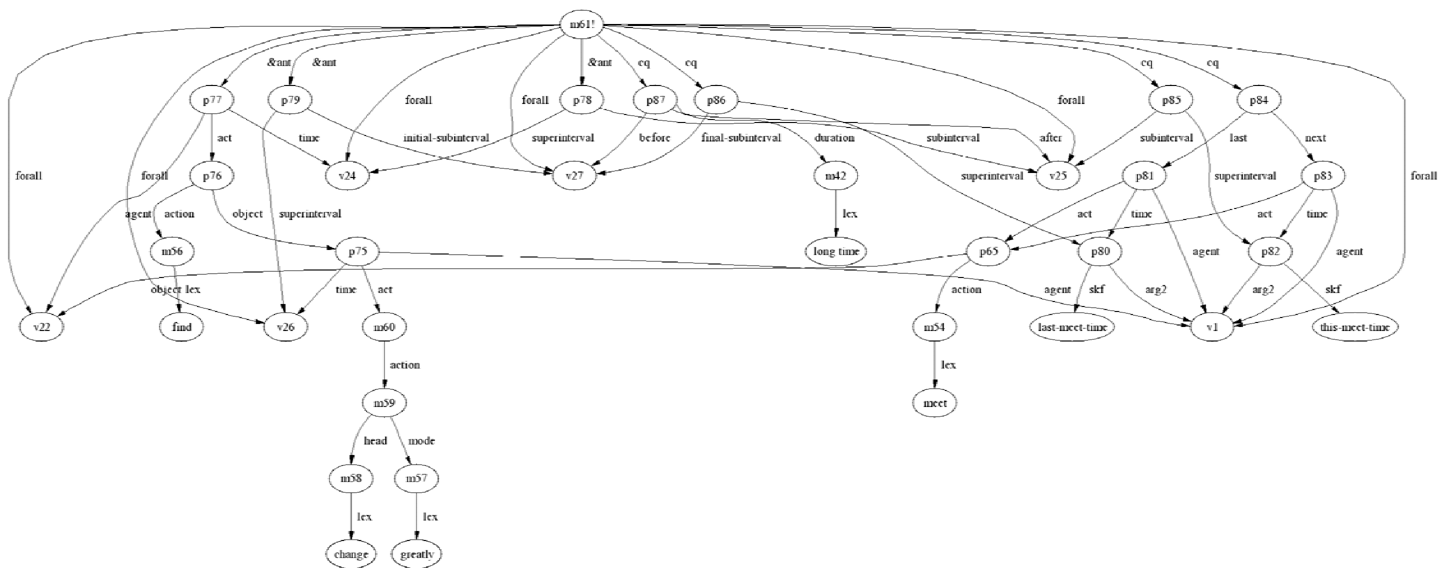

Figure 34 "If at one time some person found another person changed greatly, and the change happened during a period of time, then they had last meeting which finished at the time when the change began, and they had this meeting which happened at this time."

The last few background rules are used for deducing the relations between two "last-next meeting" groups (figure 35 – 38). Then Cassie can conclude that in the passage, "long time" is actually "one year".

RULE25: If some person had last meeting with another person, then had the next meet with the same person one year later, and if he had a third meeting with the same person short before the next meeting, then the third meeting and the next meeting are actually the same meeting.

RULE26: If proposition1 is the last one of proposition2, proposition3 is the last one of proposition4, and if proposition2 and proposition4 are same, then proposition1 and proposition3 are same.

RULE27: If some person meets another person at one time, and he meets the same person at another time, and the two meetings are actually the same meeting, then these two time periods have the same final time interval.

RULE28: If sub1 and sub3 are the same time interval, and sub1 is before sub2 for a period of time, then sub3 is before sub2 for the same period of time.

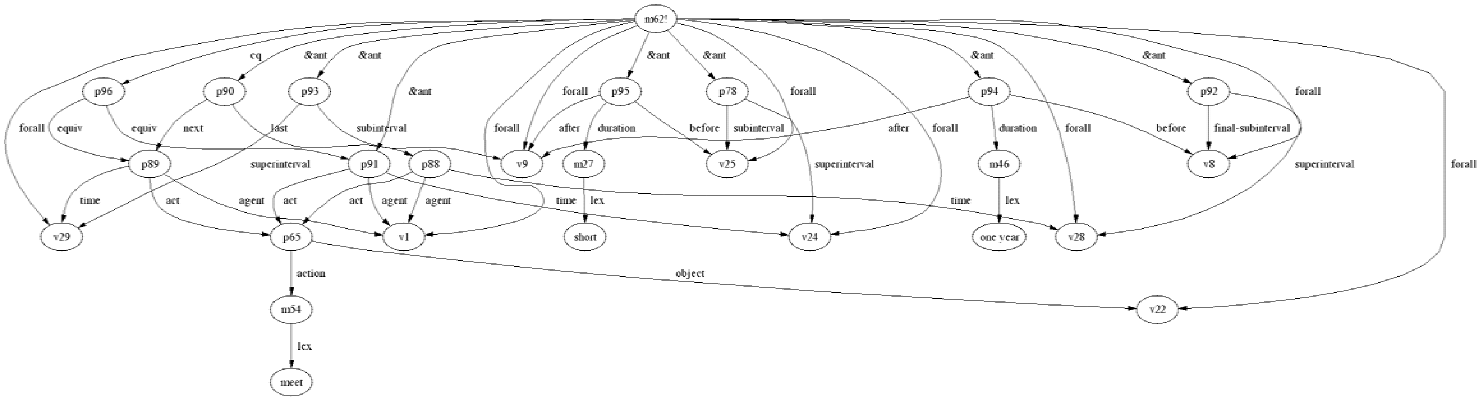

Figure 35 "If some person had last meeting with another person, then had the next meet with the same person one year later, and if he had a third meeting with the same person short before the next meeting, then the third meeting and the next meeting are actually the same meeting."

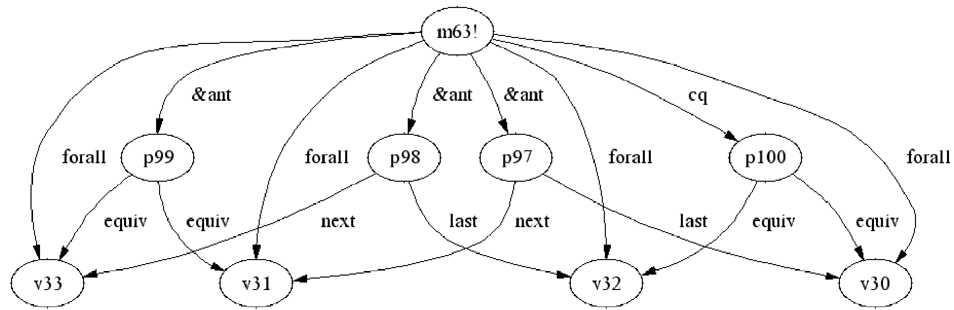

Figure 36 “If proposition1 is the last one of proposition2, proposition3 is the last one of proposition4, and if proposition2 and proposition4 are same, then proposition1 and proposition3 are same.”

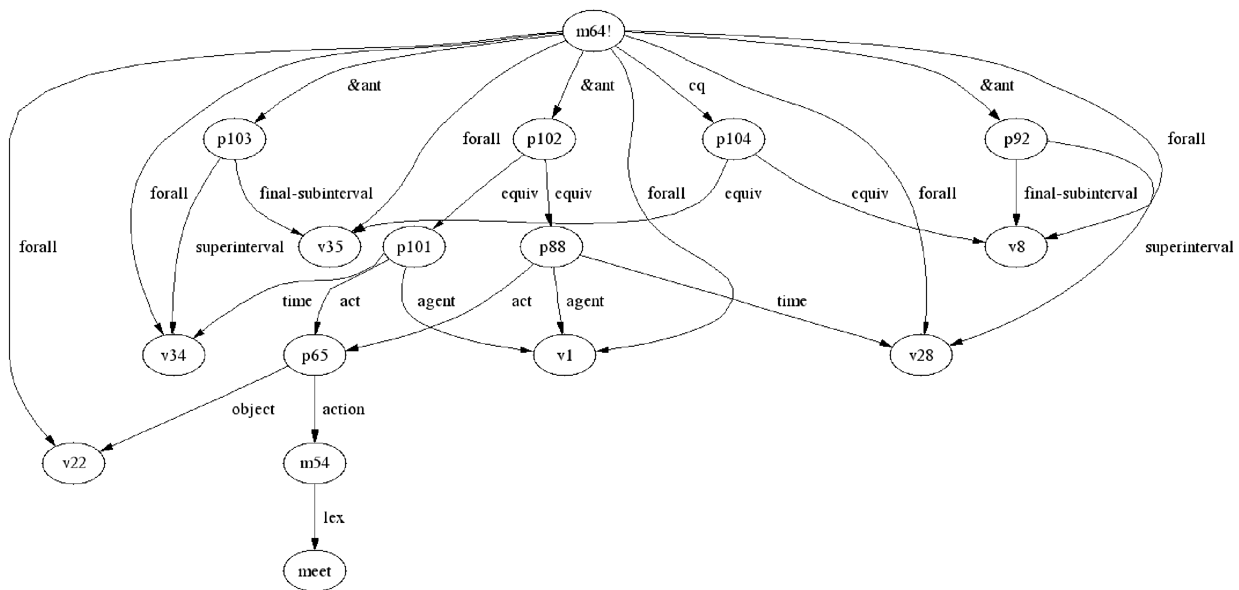

Figure 37 “If some person meets another person at one time, and he meets the same person at another time, and the two meetings are actually the same meeting, then these two time periods have the same final time interval.”

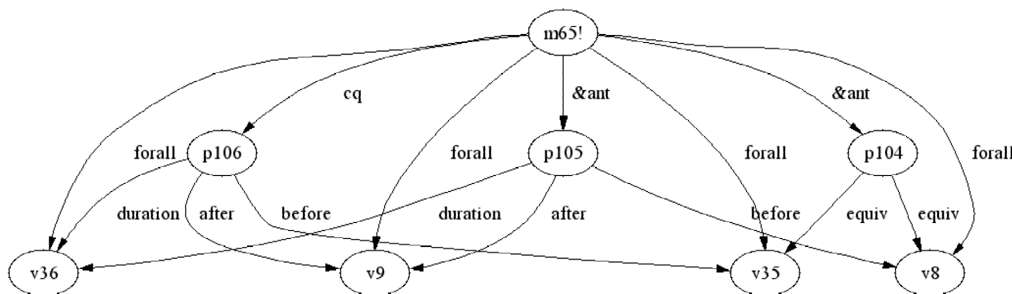

Figure 38 “If sub1 and sub3 are the same time interval, and sub1 is before sub2 for a period of time, then sub3 is before sub2 for the same period of time.”

4.3 Passage Representation

Deleted the sentences irrelevant with the definition of ‘ceilidh’, the original passage is reduced to:

> “Two ill-dressed people ... sat around a fire where the common meal was almost ready. … It had been a long time since his last *ceilidh* and Tobar had changed greatly. ... As they ate, Tobar told of his past year .... Then all too soon, their brief *ceilidh* over.”

There are four sentences in the passage. We are going to represent them sentence by sentence. For easier to represent, each sentence is changed to several short sentences and represented separately. Here I will show how each sentence is divided into short sentences. The SNePSUL representation of this passage is given in Appendix A.

SENTENCE I: “*Two ill-dressed people ... sat around a fire where the common meal was almost ready*.”

Only in this one sentence, there is a lot of information. There are two ill-dressed people, a fire, and a common meal, and people sat around the fire, and the common meal was almost ready. And all these things happened at the same time. So the first sentence is represented by the sentences below:

- There is an ill-dressed person.
- His name is Tobar.

- There is another ill-dresses person.
- The other person is Tobar's mother.
- There is a fire.
- There is a place around the fire.
- There is a time when Tobar sat at the place around the fire.
- At the same time mother sat at the place around the fire.
- There is a current time (now1) that is the subinterval of the time when they sat.
- There is a common meal.
- There is a time when the common meal was almost ready.
- The current time is also the subinterval of the time when the meal is almost ready.

SENTENCE II: “It had been a long time since his last *ceilidh* and Tobar had changed greatly.”

There is a ceilidh, which is Tobar’s last ceilidh. And it is a long time between the current time and the time when last ceilidh finished.

- There is last ceilidh.
- Ceilidh is an unknown word.
- It was Tobar's last ceilidh.
- There is a time for the last ceilidh.
- There is a finish time of last ceilidh, and the finish time is the final subinterval of the last ceilidh.
- It has been a long time since last time.

As mentioned before, to infer that last ceilidh is one year before this one, we need to know that Tobar and mother meet at current time, and their last meeting is long time ago. Because Tobar had changed greatly, Tobar meet someone long time ago, and he meet this person again at this time, and this person found Tobar had changed greatly. So here we use "mother found Tobar had changed greatly" instead of "Tobar had changed greatly".

- At a time, mother found that Tobar had changed greatly in a period of time.
- The time when mother found Tobar had changed is a superinterval of current time.
- The time period that Tobar changed began at the final subinterval of last ceilidh.
- The time period that Tobar changed ended at current time.

Even it is not mentioned in this passage, there is another ceilidh, which is Tobar and mother's this ceilidh.

- There is this ceilidh.
- This ceilidh is the next one of last ceilidh.
- There is a time for this ceilidh.
- Current time is a subinterval of the time of this ceilidh.

SENTENCE III: "*As they ate, Tobar told of his past year*."

In this sentence, the current time is after the time in the previous two sentences. Because the same reason we used in the second sentence, we represent “Tobar told of his past year to his mother” instead of “Tobar told of his past year”.

- There is a new current time (now2) after the previous time (now1).
- At a time, they eat meal.
- At the same time, Tobar told of his past year to his mother.
- The current time is a subinterval of their eating time.

SENTENCE IV: “Then all too soon, their brief *ceilidh* over.”

Agine, the current time changed to another later time. This last sentence is represented by the following sentences:

- There is a new current time (now3) shortly after previous time (now2).
- This ceilidh is Tobar's this ceilidh.
- This ceilidh is mother's this ceilidh.
- This ceilidh is over at current time.
- This ceilidh is brief.

After Cassie reads the whole passage, the final semantic network can be simply represented as in Figure 39. Some unimportant relations are neglected. The circles in the diagram represent molecular nodes and variable nodes, and the triangles represent propositions.

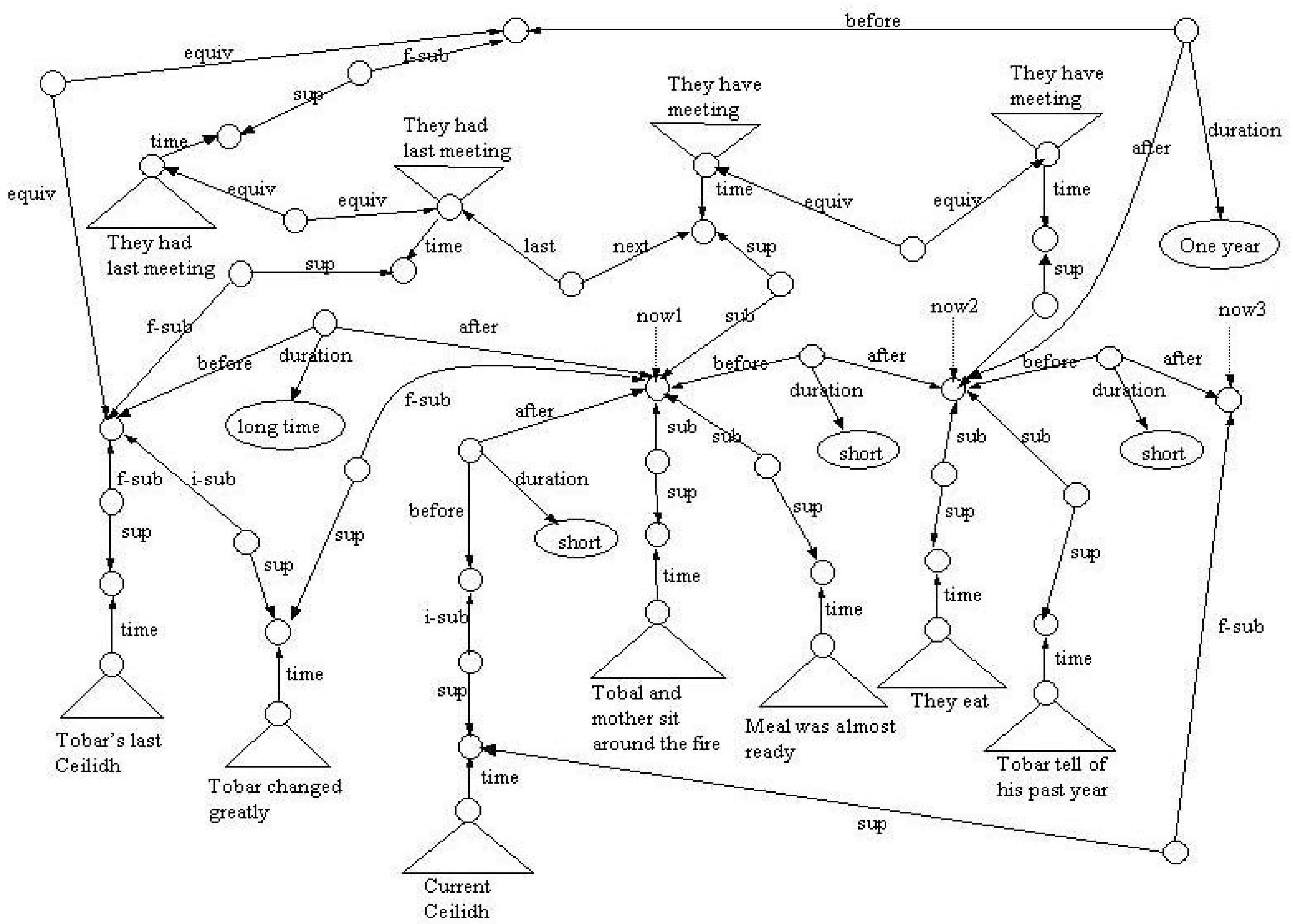

Figure 39 The Simplified Semantic Network Representation of the Passage

5. Revision of Noun Algorithm

5.1 New Functions in Noun Algorithm

All the human subjects interviewed gave the definitions include temporal information. Some of them also included spatial information in the definition. We hope that Cassie can give a complete definition of the word ‘ceilidh’ through reading the passage. In the original noun algorithm, the definition of noun is given in a structure with many slots, each of which explains one aspect of the noun, such as its class inclusion, structure element, properties, etc. These slots are not enough to define the noun ‘ceilidh’. We need more slots in the structure, which can tell information such as the frequency, duration, and location of ceilidh. Since ceilidh is the subclass of event, it will be more helpful if the definition can also tell what people do during the ceilidh. So we need to add new functions to the noun algorithm to find these information.

Four new sections are added to the noun algorithm. They are location section, event frequency section, event duration section, and event occurrence section. In the location section, there are functions used to find the location information of noun. The main functions are:

- FindLocation
- FindProbableLocation
- FindPossibleLocation

The function *FindLocation* finds the location where you can find all things which are members of the class 'noun'. *FindProbaleLocation* finds the location that presumely you can find all things which are members of the class 'noun'. FindPossibleLocation finds the location you can find at least one thing which is a member of the class 'noun'.

In the event frequency section, currently there is only one major function *FindPossibleFrequency*, which is used to find time duration that you can find at least once between two consecutive things which are members of class 'noun'. The event duration section only has one function currently. It is *FindPossibleDuration*, which finds the duration that at least one thing which is member of class 'noun' lasts. In the event concurrence section, there is mainly one function at current time. It is *FindPossibleConcurrence*, which can find a list of agent(s) and act(s) happened at the same time with 'noun' if 'noun' is an event.

The codes of the functions are in Appendix B.

5.2 Bugs Report

When I tried to use noun algorithm to define different words, I found some bugs in the original noun algorithm. Here I will give the sections and functions where I found the bugs, what the problem is, and how to correct them.

- Action section (act-object-rule, obj-act-rule, act-object-&-rule, obj-act-&-rule)

  In these functions, there are paths like this:

```
(find (compose action- act- ! cq ! ant ! class lex ) ~noun)
(find (compose action- act- ! cq ! &ant ! class lex ) ~noun)
```

These paths are supposed to find the actions that all the members of class 'noun' can perform. In the universal quantifier representation, the nodes pointed by relations of 'ant' and 'cq' are pattern nodes. Pattern nodes cannot be asserted in the network. So the correct representation should be

```
(find (compose action- act-  cq ! ant  class lex ) ~noun)
(find (compose action- act-  cq ! &ant  class lex ) ~noun)
```

- Properties section (prop-rule, prop-&)

  In the functions of *prop-rule* and *prop-&*, there are paths

```
(find (compose property- ! cq ! ant ! class lex ) ~noun)

(find (compose property- ! cq ! &ant ! class lex ) ~noun)
```

They are used to find the properties belong to all the members of the class 'noun'. For the same reason, they should be represented as

```
(find (compose property-  cq ! ant  class lex ) ~noun)

(find (compose property-  cq ! &ant  class lex ) ~noun)
```

There is more such kind of bugs in other functions in this section.

- Lexicalize

  The function *lexicalize* is used to find and return the human language representation of the SNePS nodes. In this function, there are some codes for finding the natural language representation of nodes with *mod/head* arcs coming from them. The codes are:

```
;; look for mod/head arcs coming from the node
  ((setf humanRep (append #3! ((find (compose mod- lex-) ~nodes))
                          #3! ((find (compose head- lex-) ~nodes))))
```

In the path representation, the order of *mod/lex* and *head/lex* are reversed. The correct representation is:

```
  ((setf humanRep (append #3! ((find (compose lex- mod-) ~nodes))
                          #3! ((find (compose lex- head-) ~nodes))))
```

5.3 Other Change

The original *lexicalize* function can process the node list consists of at most two nodes. Since we add function *findPossibleConcorrence*, which will return a node list with three nodes

(for example, "people tell story", which has the case frame of "agent/act/action/object"), we need *lexicalize* function to return the natural language representation of the node list consists of three nodes. The following codes were added to *lexicalize* function:

```
; if we have a list consistind of three nodes, process them and
; concatenate the result
((and (listp nodes) (eql (length nodes) 3)
    (not (listp (first nodes))) (not (listp (second nodes)))
    (not (listp (third nodes))))
 (list (concatenate 'string (first (lexicalize (first nodes))) " "
              (first (lexicalize (second nodes))) " "
              (first (lexicalize (third nodes))))))
```

6. Result and Discussion

After we tell Cassie all the background knowledge and the passage, and with the noun algorithm updated, we can ask Cassie to give the definition of 'ceilidh'.

```
;; Ask Cassie what "ceilidh" means:
^
--> (defineNoun "ceilidh")
 Definition of ceilidh:
 Probable Class Inclusions: event, gathering,
 Possible Actions: finish,
 Possible Properties: brief,
 Possessive: people this ceilidh, people last ceilidh, ill-dressed people
this ceilidh, ill-dressed people last ceilidh,
 Possible Locations: outdoor,
 Possible Frequency: long time,
 Possible Durations: short,
 Possible Concurrences: people find m133, people sit, people eat meal,
people tell story, people tell something,
nil
```

Cassie didn't find the possible frequency of "one year". Check the script of running demo, we can find that RULE25 is not fired, although all the antecedent conditions are satisfied. We can describe some nodes here:

```
* (describe m145)

(m145!
 (last (m143 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
```

```
        (time (m140 (arg2 b1) (skf last-meet-time)))))
 (next
  (m144 (act (m142)) (agent b1) (time (m138 (arg2 b1) (skf this-meet-time))))))

(m145!)

 CPU time : 0.00

* (describe m174)

(m174 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
 (time (m167 (arg2 b1) (skf this-meet-time-of))))

(m174)

 CPU time : 0.00
```

These nodes have already existed in the network. We have to add them again:

```
* (describe (add agent b1 act m142 time m138))

(m144! (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
 (time (m138 (arg2 b1) (skf this-meet-time))))

(m144!)

 CPU time : 0.05

* (describe (add last m173 next m174))

(m221!
 (equiv
  (m173 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
   (time (m169 (arg2 b1) (skf last-meet-time-of))))
  (m143 (act (m142)) (agent b1) (time (m140 (arg2 b1) (skf last-meet-time))))))
(m220!
 (equiv
  (m174 (act (m142)) (agent b1)
   (time (m167 (arg2 b1) (skf this-meet-time-of))))
  (m144! (act (m142)) (agent b1)
   (time (m138 (arg2 b1) (skf this-meet-time))))))
(m175! (last (m173)) (next (m174)))

(m221! m220! m175!)

 CPU time : 0.15
```

Now RULE25 is fired, and Cassie inferred that the two recent meetings are actually the same meeting. RULE26 is also fired, and Cassie inferred that the meeting a year ago and the

meeting long time ago are actually the same meeting. But RULE27 is still not fired. Let's add the newly inferred node again.

```
* (describe (add equiv m143 equiv m173))

(m229! (after b6)
 (before
  (m170 (arg2 (m169 (arg2 b1) (skf last-meet-time-of))) (skf final-time-
of)))
 (duration (m42 (lex long time))))
(m228! (after b16) (before (m170)) (duration (m42)))
(m227! (after b18) (before (m170)) (duration (m42)))
(m226! (after (m191 (arg2 b15) (skf initial-time-of))) (before (m170))
 (duration (m42)))
(m225! (after b16) (before b11) (duration (m46 (lex one year))))
(m224! (after b6) (before b11) (duration (m46)))
(m223! (after (m191)) (before b11) (duration (m46)))
(m222! (equiv (m170) b11))

……

(m229! m228! m227! m226! m225! m224! m223! m222! m221! m209! m208! m205! m204!
 m185! m182! m181! m178! m177! m176! m172! m157! m156! m131! m130!)

 CPU time : 0.77
```

RULE27 and RULE28 are fired this time. Cassie begins to know that the final time of the last two meetings are same time, and "one year" and "long time" represent the same time period. Now we ask Cassie what ceilidh means again.

```
;; Ask Cassie what "ceilidh" means:
^
--> (defineNoun "ceilidh")
 Definition of ceilidh:
 Probable Class Inclusions: event, gathering,
 Possible Actions: finish,
 Possible Properties: brief,
 Possessive: people this ceilidh, people last ceilidh, ill-dressed people
this ceilidh, ill-dressed people last ceilidh,
 Possible Locations: outdoor,
 Possible Frequency: one year, long time,
 Possible Durations: short,
 Possible Concurrences: people find m133, people sit, people eat meal,
people tell story, people tell something, people meet people,
nil
```

The full forward inference in CVA project allows Cassie uses the previously inferred information as her knowledge to infer new information when she is reading. But here the full forward inference system doesn't work well. So we have to tell Cassie what she has already known to make her make inference.

In the background knowledge we told Cassie, there are rules to deduce spatial information. From these rules Cassie can deduce that fires and people who sit around fires are probably outdoor. We didn't tell Cassie any rule to infer that ceilidh could be outdoor activity. But in Cassie's definition, the possible location is outdoor. Describing all the nodes in the semantic network, we can not find there is such information in the network. Where did Cassie find this? If we try to find this information as below:

```
* (find object *thisceilidh location (build lex outdoor))
(p173)
 CPU time : 0.00
```

A pattern node is returned. This pattern node is not in the 'ceilidh' network. When SNePS processes node-based inference, it generates a pattern node to match the pattern node in the background rules which is pointed by a "cq" arc, and then match the antecedent pattern against the network (Shapiro, 1978). The pattern node p173 was generated during the node-base inference, and was discarded because there is no match found in the network. But for some reason, the noun algorithm found it.

In summary, Cassie tells us that ceilidh is probably a gathering that owned by ill-dressed people. It probably happens every long time, or once a year. It is probably hold outdoor, and it could be short. During ceilidh, people meet each other, eat meal, and tell stories.

7. Conclusion and Future Work

In this paper, Cassie was asked to define the word ‘ceilidh’ by reading a passage. The verbal protocols were analyzed, and the background knowledge human subjects used to infer the meaning of ‘ceilidh’ was represented as Cassie’s knowledge base in SNePSUL. To make it easier to understand, the passage was divided into many short sentences. Cassie read these sentences, and represented them in semantic network. During the reading, Cassie also inferred other information which is not told through her background knowledge. When asked to define word ‘ceilidh’, Cassie searched the network and found all the information related with ‘ceilidh’. Because non-standard SNePS case frames were used, and we need to find some information that can not be found by original noun algorithm, some new functions were added to the noun algorithm to find the location, frequency, duration and concurrence information of ceilidh.

Even though Cassie gave a very reasonable and comprehensive dictionary-like definition of ‘ceilidh’, there are still problem in this project. The first problem is related with the forward inference. Some background rules can not be fired when Cassie reads the passage, so we have to tell Cassie again what she have already known to make her infer new information. The second is that Cassie gives some information in the definition which is not in the network. In the next step, we should to solve these problems.

In the passage, it is told that “it has been a long time since Tobar’s last ceilidh”. And the passage doesn’t mention any other ceilidh until the end of it that “their brief ceilidh over”. Everyone reads this passage will know that there is “this ceilidh”, and “their brief ceilidh” is their “this ceilidh”. In this project, we didn’t tell Cassie the background knowledge that if some people had last ceilidh, then he also had this ceilidh. Because by this way, Cassie can automatically infer that there is this ceilidh when she reads “Tobar’s last ceilidh”, but we can not refer it when later Cassie reads something about it. So in this project, we just told Cassie that there is this ceilidh when she read the passage. The future work should concentrate on how to automatically generate a node represents “this ceilidh” if something about “last ceilidh” is told, and this node should be able to be referred later.

SNePS is a natural-language-understanding system. Eventually Cassie will read the passage in natural language. Since we represent a lot of temporal information with non-standard SNePS case frames, other grammar parser can not generate the representation with these case frames. So a grammar file which can parse the passage and generate the semantic network representation with these non-standard case frames is needed.

Appendix A: Script of Running Demo

```
pollux {~/cse740/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.

   4/25/2004 19:06:27

* (demo "ceilidh.demo")

File /home/geograd/junxu/cse740/project/ceilidh.demo is now the source of
input.


 CPU time : 0.01

*
;;;==========================================================================
;;; FILENAME:     ceilidh.demo
;;;
;;; DATE:   March, 2004
;;; PROGRAMMER:   Jun Xu
;;;==========================================================================


;;; Two ill-dressed people... sat around a fire where the common meal was
;;; almost ready. The mother... peered at her son through the oam of the
;;; bubbling stew. It had been a long time since his last ceilidh and Tobar
;;; had changed greatly. ... AS they ate, Tobar told his past year... Then
;;; all too soon, their breif ceilidh over, Tobar walked over to touch his
;;; mother's arm and quickly left.
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


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



 CPU time : 0.01

*
;; Load the appropriate definition algorithm:
^
--> (load "defun_noun.cl")
; Loading /home/geograd/junxu/cse740/project/defun_noun.cl
t



 CPU time : 0.26

*
;; Clear the SNePS network:
(resetnet)

Net reset - Relations and paths are still defined


 CPU time : 0.00

*
;; enter the "snip" package:
^
--> (in-package snip)
#<The snip package>



 CPU time : 0.01

*
;; 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)))))
    t)
```

```
broadcast-one-report



 CPU time : 0.01

*
;;re-enter the "sneps" package:
^
--> (in-package sneps)
#<The sneps package>



 CPU time : 0.00

*
;;load all pre-defined relations:
(intext "rels")
File rels is now the source of input.


 CPU time : 0.00

*
(agent object antonym lex member class rel location proper-name property
 possessor part whole superclass subclass synonym before after duration
 subinterval superinterval final-subinterval initial-subinterval time mode
 equiv skf mod head arg2 last next to)

 CPU time : 0.01

*

End of file rels


 CPU time : 0.01

*
;;load all pre-defined path definitions:
(intext "paths")
File paths is now the source of input.


 CPU time : 0.00

*
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

*
```

```
subclass implied by the path (compose subclass
                              (kstar (compose superclass- ! subclass)))
subclass- implied by the path (compose
                               (kstar (compose subclass- ! superclass))
                               subclass-)


 CPU time : 0.00

*
superclass implied by the path (compose superclass
                                (kstar (compose subclass- ! superclass)))
superclass- implied by the path (compose
                                 (kstar (compose superclass- ! subclass))
                                 superclass-)


 CPU time : 0.00

*
member implied by the path (compose member (kstar (compose equiv- ! equiv)))
member- implied by the path (compose (kstar (compose equiv- ! equiv))
member-)


 CPU time : 0.00

*

End of file paths


 CPU time : 0.00

*
;;load background knowledge
(demo "ceilidh.base")

File /home/geograd/junxu/cse740/project/ceilidh.base is now the source of
input.


 CPU time : 0.01

*
;;; This is the set of assertions which build the base network which
;;; corresponds to the "celidh" passage.



;Gathering is subclass of events.
(describe (assert subclass (build lex gathering) superclass (build lex
event)))

(m3! (subclass (m1 (lex gathering))) (superclass (m2 (lex event))))

(m3!)
```

```
 CPU time : 0.00

*
;meal is subclass of food.
(describe (assert subclass (build lex meal) superclass (build lex food)))

(m6! (subclass (m4 (lex meal))) (superclass (m5 (lex food))))

(m6!)

 CPU time : 0.00

*

;;;
;;; rules
;;;

;Ill-deressed people are people.
(describe
 (assert forall $p
      ant (build member *p class (build mod (build lex ill-dressed)
                                 head (build lex people)))
      cq (build member *p class (build lex people))))

(m10! (forall v1)
 (ant
  (p1 (class (m9 (head (m8 (lex people))) (mod (m7 (lex ill-dressed)))))
   (member v1)))
 (cq (p2 (class (m8)) (member v1))))

(m10!)

 CPU time : 0.04

*
;Common meals are meals.
(describe
 (assert forall $x
      ant (build member *x class (build mod (build lex common)
                                 head (build lex meal)))
      cq (build member *x class (build lex meal))))

(m13! (forall v2)
 (ant
  (p3 (class (m12 (head (m4 (lex meal))) (mod (m11 (lex common)))))
   (member v2)))
 (cq (p4 (class (m4)) (member v2))))

(m13!)

 CPU time : 0.00

*
;If something is a member of class fire, then this thing is probably
outdoor.
```

```
(describe
 (assert forall $y
      ant (build member *y class (build lex fire))
      cq (build mode (build lex presumably)
              object (build object *y location (build lex outdoor)))))

(m17! (forall v3) (ant (p5 (class (m14 (lex fire))) (member v3)))
 (cq
  (p7 (mode (m15 (lex presumably)))
   (object (p6 (location (m16 (lex outdoor))) (object v3))))))

(m17!)

 CPU time : 0.01

*
;If something is around another thing that is probably outdoor, then this
;thing is probably outdoor.
(describe
 (assert forall ($z *y)
      &ant ((build object1 *z rel (build lex around) object2 *y)
            (build mode (build lex presumably)
                     object (build object *y location (build lex outdoor))))
      cq (build mode (build lex presumably)
                object (build object *z location (build lex outdoor)))))

(m19! (forall v4 v3)
 (&ant (p8 (object1 v4) (object2 v3) (rel (m18 (lex around))))
  (p7 (mode (m15 (lex presumably)))
   (object (p6 (location (m16 (lex outdoor))) (object v3)))))
 (cq (p10 (mode (m15)) (object (p9 (location (m16)) (object v4))))))

(m19!)

 CPU time : 0.01

*
;If someone sit at some place which is probably outdoor at a time, then this
;someone is probably outdoor at this time.
(describe
 (assert forall (*p *z $t)
      &ant ((build agent *p act (build action (build lex sit))
                   location *z time *t)
              (build mode (build lex presumably)
                     object (build object *z location (build lex outdoor))))
      cq (build mode (build lex presumably)
              object (build object *p location (build lex outdoor) time *t))))

(m22! (forall v5 v4 v1)
 (&ant
  (p11 (act (m21 (action (m20 (lex sit))))) (agent v1) (location v4)
   (time v5))
  (p10 (mode (m15 (lex presumably)))
   (object (p9 (location (m16 (lex outdoor))) (object v4)))))
 (cq (p13 (mode (m15)) (object (p12 (location (m16)) (object v1) (time v5))))))

(m22!)
```

```
 CPU time : 0.01

*
;If something is a member of meal, someone is a member of person eats it,
;then the person eats it shortly after it is almost ready.
(describe
 (assert forall (*p *x $t1 $t2 $sub1 $sub2)
      &ant ((build member *x class (build lex meal))
             (build member *p class (build lex people))
             (build object *x property
                    (build mod (build lex almost) head (build lex ready))
                   time *t1)
             (build subinterval *sub1 superinterval *t1)
             (build agent *p act (build action (build lex eat) object *x)
                          time *t2)
             (build subinterval *sub2 superinterval *t2))
      cq (build before *sub1 after *sub2 duration (build lex short))))

(m28! (forall v9 v8 v7 v6 v2 v1)
 (&ant (p18 (subinterval v9) (superinterval v7))
  (p17 (act (p16 (action (m26 (lex eat))) (object v2))) (agent v1) (time v7))
  (p15 (subinterval v8) (superinterval v6))
  (p14 (object v2)
   (property (m25 (head (m24 (lex ready))) (mod (m23 (lex almost)))))
   (time v6))
  (p4 (class (m4 (lex meal))) (member v2))
  (p2 (class (m8 (lex people))) (member v1)))
 (cq (p19 (after v9) (before v8) (duration (m27 (lex short))))))

(m28!)

 CPU time : 0.00

*
;If something is over at a time, then this time is its end time.
(describe
 (assert forall ($sth $super $fsub)
      &ant ((build agent *sth act (build action (build lex finish))
                          time *fsub)
             (build object *sth time *super))
      cq (build final-subinterval *fsub superinterval *super)))

(m31! (forall v12 v11 v10)
 (&ant (p21 (object v10) (time v11))
  (p20 (act (m30 (action (m29 (lex finish))))) (agent v10) (time v12)))
 (cq (p22 (final-subinterval v12) (superinterval v11))))

(m31!)

 CPU time : 0.00

*
;If something has an end time, then it has an initial time which is before
;the end time.
(describe
 (assert forall (*sth *super *fsub)
```

```
      &ant ((build object *sth time *super)
             (build final-subinterval *fsub superinterval *super))
      cq ((build initial-subinterval (build skf "initial-time-of" arg2 *super)
                                  = itime
                 superinterval *super)
           (build before *itime after *fsub))))

(m32! (forall v12 v11 v10)
 (&ant (p22 (final-subinterval v12) (superinterval v11))
  (p21 (object v10) (time v11)))
 (cq (p25 (after v12) (before (p23 (arg2 v11) (skf initial-time-of))))
  (p24 (initial-subinterval (p23)) (superinterval v11))))

(m32!)

 CPU time : 0.01

*
;If a time period has a subinterval, an initial-subinterval, then the
;initial subinterval is before the subinterval.
(describe
 (assert forall ($isub $sub *super)
      &ant ((build subinterval *sub superinterval *super)
             (build initial-subinterval *isub superinterval *super))
      cq (build before *isub after *sub)))

(m33! (forall v14 v13 v11)
 (&ant (p27 (initial-subinterval v13) (superinterval v11))
  (p26 (subinterval v14) (superinterval v11)))
 (cq (p28 (after v14) (before v13))))

(m33!)

 CPU time : 0.00

*
;If a time period has a subinterval, an final-subinterval, then the final
;subinterval is after the subinterval.
(describe
 (assert forall (*fsub *sub *super)
      &ant ((build subinterval *sub superinterval *super)
             (build final-subinterval *fsub superinterval *super))
      cq (build before *sub after *fsub)))

(m34! (forall v14 v12 v11)
 (&ant (p26 (subinterval v14) (superinterval v11))
  (p22 (final-subinterval v12) (superinterval v11)))
 (cq (p29 (after v12) (before v14))))

(m34!)

 CPU time : 0.00

*
;If something has an initial time and an end time, and the initial time is
;before the end time, then this thing is a member of event.
(describe
```

```
 (assert forall (*sth *super *isub *fsub)
      &ant ((build object *sth time *super)
            (build initial-subinterval *isub superinterval *super)
            (build final-subinterval *fsub superinterval *super)
            (build before *isub after *fsub))
      cq (build member *sth class (build lex event))))

(m35! (forall v13 v12 v11 v10)
 (&ant (p30 (after v12) (before v13))
  (p27 (initial-subinterval v13) (superinterval v11))
  (p22 (final-subinterval v12) (superinterval v11))
  (p21 (object v10) (time v11)))
 (cq (p31 (class (m2 (lex event))) (member v10))))

(m35!)

 CPU time : 0.01

*
;If something is an event and part of another unknown class, then probably
;the class is a subclass of event.
(describe
 (assert forall (*sth $cls)
      &ant ((build member *sth class *cls)
            (build object *cls property (build lex unknown))
            (build member *sth class (build lex event)))
      cq (build mode (build lex presumably)
              object (build subclass *cls superclass (build lex event)))))

(m37! (forall v15 v10)
 (&ant (p33 (object v15) (property (m36 (lex unknown))))
  (p32 (class v15) (member v10)) (p31 (class (m2 (lex event))) (member v10)))
 (cq
  (p35 (mode (m15 (lex presumably)))
   (object (p34 (subclass v15) (superclass (m2)))))))

(m37!)

 CPU time : 0.00

*

;If something is probably a gathering and part of another unknown class,
;then probably the class is a subclass of gathering.
(describe
 (assert forall (*sth *cls)
      &ant ((build member *sth class *cls)
            (build object *cls property (build lex unknown))
            (build mode (build lex presumably)
                 object (build member *sth class (build lex gathering))))
      cq (build mode (build lex presumably)
              object (build subclass *cls superclass (build lex gathering)))))

(m38! (forall v15 v10)
 (&ant
  (p37 (mode (m15 (lex presumably)))
   (object (p36 (class (m1 (lex gathering))) (member v10))))
```

```
  (p33 (object v15) (property (m36 (lex unknown))))
  (p32 (class v15) (member v10)))
 (cq (p39 (mode (m15)) (object (p38 (subclass v15) (superclass (m1)))))))

(m38!)

 CPU time : 0.00

*
;If something is an event, and it is brief, then the duration between its
;initial time and final time is short.
(describe
 (assert forall (*sth *super *isub *fsub)
      &ant ((build member *sth class (build lex event))
            (build object *sth property (build lex brief))
            (build object *sth time *super)
            (build initial-subinterval *isub superinterval *super)
            (build final-subinterval *fsub superinterval *super))
      cq (build before *isub after *fsub duration (build lex short))))

(m40! (forall v13 v12 v11 v10)
 (&ant (p40 (object v10) (property (m39 (lex brief))))
  (p31 (class (m2 (lex event))) (member v10))
  (p27 (initial-subinterval v13) (superinterval v11))
  (p22 (final-subinterval v12) (superinterval v11))
  (p21 (object v10) (time v11)))
 (cq (p41 (after v12) (before v13) (duration (m27 (lex short))))))

(m40!)

 CPU time : 0.00

*
;If time1 is before time2 and time2 is before time3, then time1 is before
;time3.
(describe
 (assert forall ($time1 $time2 $time3)
      &ant ((build before *time1 after *time2)
            (build before *time2 after *time3))
      cq (build before *time1 after *time3)))

(m41! (forall v18 v17 v16)
 (&ant (p43 (after v18) (before v17)) (p42 (after v17) (before v16)))
 (cq (p44 (after v18) (before v16))))

(m41!)

 CPU time : 0.01

*
;If time1 is long time before time2, and time2 is before time3, then time1
;is long time before time3.
(describe
 (assert forall (*time1 *time2 *time3)
      &ant ((build before *time1 after *time2 duration (build lex long\ time))
            (build before *time2 after *time3))
      cq (build before *time1 after *time3 duration (build lex long\ time))))
```

```
(m43! (forall v18 v17 v16)
 (&ant (p45 (after v17) (before v16) (duration (m42 (lex long time))))
  (p43 (after v18) (before v17)))
 (cq (p46 (after v18) (before v16) (duration (m42)))))

(m43!)

 CPU time : 0.00

*
;If time1 is long time before time3, and time2 is short before time3, then
;time1 is long time before time2.
(describe
 (assert forall (*time1 *time2 *time3)
      &ant ((build before *time1 after *time3 duration (build lex long\ time))
            (build before *time2 after *time3 duration (build lex short)))
      cq (build before *time1 after *time2 duration (build lex long\ time))))

(m44! (forall v18 v17 v16)
 (&ant (p47 (after v18) (before v17) (duration (m27 (lex short))))
  (p46 (after v18) (before v16) (duration (m42 (lex long time)))))
 (cq (p45 (after v17) (before v16) (duration (m42)))))

(m44!)

 CPU time : 0.04

*
;If time1 is before time2, time2 is before time3, and time1 is short before
;time3, then time1 is short before time2 and time2 is short before time3.
(describe
 (assert forall (*time1 *time2 *time3)
      &ant ((build before *time1 after *time3 duration (build lex short))
            (build before *time1 after *time2)
            (build before *time2 after *time3))
      cq ((build before *time1 after *time2 duration (build lex short))
          (build before *time2 after *time3 duration (build lex short)))))

(m45! (forall v18 v17 v16)
 (&ant (p48 (after v18) (before v16) (duration (m27 (lex short))))
  (p43 (after v18) (before v17)) (p42 (after v17) (before v16)))
 (cq (p49 (after v17) (before v16) (duration (m27)))
  (p47 (after v18) (before v17) (duration (m27)))))

(m45!)

 CPU time : 0.00

*
;If time1 is one year before time3, and time2 is short before time3, then
;time1 is one year before time2.
(describe
 (assert forall (*time1 *time2 *time3)
      &ant ((build before *time1 after *time3 duration (build lex "one year"))
            (build before *time2 after *time3 duration (build lex short)))
      cq (build before *time1 after *time2 duration (build lex "one year"))))
```

```
(m47! (forall v18 v17 v16)
 (&ant (p50 (after v18) (before v16) (duration (m46 (lex one year))))
  (p47 (after v18) (before v17) (duration (m27 (lex short)))))
 (cq (p51 (after v17) (before v16) (duration (m46)))))

(m47!)

 CPU time : 0.00

*
;If there is any time interval between the initial subinterval and the final
;subinterval of something, then this time interval is the subinterval of
;this thing.
(describe
 (assert forall (*super *isub *fsub *sub)
      &ant ((build before *isub after *sub)
            (build before *sub after *fsub)
            (build initial-subinterval *isub superinterval *super)
            (build final-subinterval *fsub superinterval *super))
      cq (build subinterval *sub superinterval *super)))

(m48! (forall v14 v13 v12 v11)
 (&ant (p29 (after v12) (before v14)) (p28 (after v14) (before v13))
  (p27 (initial-subinterval v13) (superinterval v11))
  (p22 (final-subinterval v12) (superinterval v11)))
 (cq (p26 (subinterval v14) (superinterval v11))))

(m48!)

 CPU time : 0.00

*
;If some person tells his past year at a time, he tells story at this time.
(describe
 (assert forall (*p *t)
      ant (build agent *p act (build action (build lex tell)
                                object (build skf "past-year-of" arg2 *p))
                 time *t)
      cq (build agent *p act (build action (build lex tell)
                              object (build lex story)) time *t)))

(m52! (forall v5 v1)
 (ant
  (p54
   (act (p53 (action (m49 (lex tell)))
         (object (p52 (arg2 v1) (skf past-year-of)))))
   (agent v1) (time v5)))
 (cq
  (p55 (act (m51 (action (m49)) (object (m50 (lex story))))) (agent v1)
   (time v5))))

(m52!)

 CPU time : 0.00

*
```

```
;If there is subinterval of something, and during this interval people eat
;and tell story, then probably this thing is a gathering.
(describe
 (assert forall (*sth $sup1 *sub $sup2 *p $food)
      &ant ((build object *sth time *sup1)
             (build subinterval *sub superinterval *sup1)
             (build subinterval *sub superinterval *sup2)
             (build member *p class (build lex people))
             (build member *food class (build lex food))
             (build time *sup2 agent *p act (build action (build lex eat)
                                              object *food))
             (build time *sup2 agent *p
                  act (build action (build lex tell) object (build lex story))))
      cq (build mode (build lex presumably)
              object (build member *sth class (build lex gathering)))))

(m53! (forall v21 v20 v19 v14 v10 v1)
 (&ant
  (p62 (act (m51 (action (m49 (lex tell))) (object (m50 (lex story)))))
   (agent v1) (time v20))
  (p61 (act (p60 (action (m26 (lex eat))) (object v21))) (agent v1)
   (time v20))
  (p59 (class (m5 (lex food))) (member v21))
  (p58 (subinterval v14) (superinterval v20))
  (p57 (subinterval v14) (superinterval v19)) (p56 (object v10) (time v19))
  (p2 (class (m8 (lex people))) (member v1)))
 (cq
  (p37 (mode (m15 (lex presumably)))
   (object (p36 (class (m1 (lex gathering))) (member v10))))))

(m53!)

 CPU time : 0.00

*
;If some person tells another person of his past year at a time, then they
;meet at this time, and they meet at another time whose final interval is
;one year before this time.
(describe
 (assert forall (*p $p2 $super2 *sub2)
      &ant ((build agent *P
                  act (build action (build lex tell)
                         object (build skf "past-year-of" arg2 *p))
                  to *p2
                  time *super2)
             (build subinterval *sub2 superinterval *super2))
      cq ((build last (build agent *p
                         act (build action (build lex meet) object *p2)
                         time (build skf "last-meet-time-of" arg2 *p)
                                          = lastmeettime)
                next (build agent *p
                         act (build action (build lex meet) object *p2)
                         time (build skf "this-meet-time-of" arg2 *p)
                                    = thismeettime))
           (build subinterval *sub2 superinterval *thismeettime)
           (build final-subinterval (build skf "final-time-of" arg2 *lastmeettime)
                             = finalmeettime
```

```
               superinterval *lastmeettime)
          (build before *finalmeettime
               after *sub2
               duration (build lex "one year")))))

(m55! (forall v23 v22 v9 v1)
 (&ant (p64 (subinterval v9) (superinterval v23))
  (p63
   (act (p53 (action (m49 (lex tell)))
         (object (p52 (arg2 v1) (skf past-year-of)))))
   (time v23) (to v22)))
 (cq
  (p74 (after v9)
   (before
    (p72 (arg2 (p66 (arg2 v1) (skf last-meet-time-of))) (skf final-time-of)))
   (duration (m46 (lex one year))))
  (p73 (final-subinterval (p72)) (superinterval (p66)))
  (p71 (subinterval v9)
   (superinterval (p68 (arg2 v1) (skf this-meet-time-of))))
  (p70
   (last (p67 (act (p65 (action (m54 (lex meet))) (object v22))) (agent v1)
          (time (p66))))
   (next (p69 (act (p65)) (agent v1) (time (p68)))))))

(m55!)

 CPU time : 0.01

*
;If at one time some person found another person changed greatly, and the
;change happened during a period of time, then they had last meeting which
;finished at the time when the change began, and they had this meeting which
;happened at this time.
(describe
 (assert forall (*p *p2 $super4 $sub4 $ctime $itime)
      &ant ((build agent *p2
                  act (build action (build lex find)
                           object (build agent *p
                                     act (build action
                                           (build mode (build lex greatly)
                                                       head (build lex change)))
                                     time *ctime))
                  time *super4)
             (build subinterval *sub4 superinterval *super4)
             (build initial-subinterval *itime superinterval *ctime))
      cq ((build last (build agent *p
                          act (build action (build lex meet) object *p2)
                          time (build skf "last-meet-time" arg2 *p)
                               = lastmeettime)
               next (build agent *p
                         act (build action (build lex meet) object *p2)
                         time (build skf "this-meet-time" arg2 *p)
                              = thismeettime))
          (build subinterval *sub4 superinterval *thismeettime)
          (build final-subinterval *itime superinterval *lastmeettime)
          (build before *itime after *sub4 duration (build lex "long time")))))
```

```
(m61! (forall v27 v26 v25 v24 v22 v1)
 (&ant (p79 (initial-subinterval v27) (superinterval v26))
  (p78 (subinterval v25) (superinterval v24))
  (p77
   (act (p76 (action (m56 (lex find)))
         (object
          (p75
           (act (m60
                 (action
                  (m59 (head (m58 (lex change)))
                   (mode (m57 (lex greatly)))))))
           (agent v1) (time v26)))))
   (agent v22) (time v24)))
 (cq (p87 (after v25) (before v27) (duration (m42 (lex long time))))
  (p86 (final-subinterval v27)
   (superinterval (p80 (arg2 v1) (skf last-meet-time))))
  (p85 (subinterval v25) (superinterval (p82 (arg2 v1) (skf this-meet-time))))
  (p84
   (last (p81 (act (p65 (action (m54 (lex meet))) (object v22))) (agent v1)
          (time (p80))))
   (next (p83 (act (p65)) (agent v1) (time (p82)))))))

(m61!)

 CPU time : 0.00

*
;If some person had last meeting with another person, then had the next meet
;with the same person one year later, and if he had a third meeting with the
;same person short before the next meeting, then the third meeting and the
;next meeting are actually the same meeting.
(describe
 (assert forall (*p *p2 $super1 $super2 *super4 *sub1 *sub2 *sub4)
      &ant ((build last (build agent *p
                          act (build action (build lex meet) object *p2)
                          time *super1)
                      next (build agent *p
                          act (build action (build lex meet) object *p2)
                              time *super2))
            (build agent *p
                 act (build action (build lex meet) object *p2)
                 time *super4)
            (build final-subinterval *sub1 superinterval *super1)
            (build subinterval *sub2 superinterval *super2)
            (build subinterval *sub4 superinterval *super4)
            (build before *sub1 after *sub2 duration (build lex "one year"))
            (build before *sub4 after *sub2 duration (build lex short)))
      cq (build equiv (build agent *p
                          act (build action (build lex meet) object *p2)
                          time *super2)
              equiv (build agent *p
                          act (build action (build lex meet) object *p2)
                          time *super4))))

(m62! (forall v29 v28 v25 v24 v22 v9 v8 v1)
 (&ant (p95 (after v9) (before v25) (duration (m27 (lex short))))
  (p94 (after v9) (before v8) (duration (m46 (lex one year))))
```

```
  (p93 (subinterval v9) (superinterval v29))
  (p92 (final-subinterval v8) (superinterval v28))
  (p91 (act (p65 (action (m54 (lex meet))) (object v22))) (agent v1)
   (time v24))
  (p90 (last (p88 (act (p65)) (agent v1) (time v28)))
   (next (p89 (act (p65)) (agent v1) (time v29))))
  (p78 (subinterval v25) (superinterval v24)))
 (cq (p96 (equiv (p91) (p89)))))

(m62!)

 CPU time : 0.00

*
;If proposition1 is the last one of proposition2, proposition3 is the last
;one of proposition4, and proposition2 and proposition4 are same, then
;proposition1 and proposition3 are same.
(describe
 (assert forall ($pp1 $pp2 $pp3 $pp4)
            &ant ((build last *pp1 next *pp2)
            (build last *pp3 next *pp4)
            (build equiv *pp2 equiv *pp4))
      cq (build equiv *pp1 equiv *pp3)))

(m63! (forall v33 v32 v31 v30)
 (&ant (p99 (equiv v33 v31)) (p98 (last v32) (next v33))
  (p97 (last v30) (next v31)))
 (cq (p100 (equiv v32 v30))))

(m63!)

 CPU time : 0.01

*
;If some person meets another person at one time, and he meets the same
;person at another time, and the two meetings are actually the same meeting,
;then these two time periods have the same final time interval.
(describe
 (assert forall (*p *p2 *super1 $super3 *sub1 $sub3)
      &ant ((build equiv (build agent *p
                         act (build action (build lex meet) object *p2)
                         time *super1)
                  equiv (build agent *p
                         act (build action (build lex meet) object *p2)
                         time *super3))
            (build final-subinterval *sub1 superinterval *super1)
            (build final-subinterval *sub3 superinterval *super3))
      cq (build equiv *sub1 equiv *sub3)))

(m64! (forall v35 v34 v28 v22 v8 v1)
 (&ant (p103 (final-subinterval v35) (superinterval v34))
  (p102
   (equiv
    (p101 (act (p65 (action (m54 (lex meet))) (object v22))) (agent v1)
     (time v34))
    (p88 (act (p65)) (agent v1) (time v28))))
  (p92 (final-subinterval v8) (superinterval v28)))
```

```
 (cq (p104 (equiv v35 v8))))

(m64!)

 CPU time : 0.00

*
;If sub1 and sub3 are same time interval, and sub1 is before sub2 for a
period of time, then sub3 is before sub2 for the same period of time.
(describe
 (assert forall (*sub1 *sub2 *sub3 $d)
      &ant ((build equiv *sub1 equiv *sub3)
            (build before *sub1 after *sub2 duration *d))
      cq (build before *sub3 after *sub2 duration *d)))

(m65! (forall v36 v35 v9 v8)
 (&ant (p105 (after v9) (before v8) (duration v36)) (p104 (equiv v35 v8)))
 (cq (p106 (after v9) (before v35) (duration v36))))

(m65!)

 CPU time : 0.03

*


End of /home/geograd/junxu/cse740/project/ceilidh.base demonstration.


 CPU time : 0.29

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


;;;CASSIE READS THE PASSAGE:


;;Two ill-dressed people ... sat around a fire where the commom meal was
almost ready.

;There is an ill-dressed person.
(describe
 (add member #people1 class
      (build mod (build lex ill-dressed) head (build lex people))))

(m67! (class (m8 (lex people))) (member b1))
(m66! (class (m9 (head (m8)) (mod (m7 (lex ill-dressed))))) (member b1))

(m67! m66!)

 CPU time : 1.52

*
;; Newly inferred information:
;;
;; The ill-dressed person is a person.
```

```
;;

;His name is Tobar.
(describe (add object *people1 proper-name (build lex Tobar)))

(m76! (object b1) (proper-name (m75 (lex Tobar))))

(m76!)

 CPU time : 0.02

*
;There is another ill-dresses person.
(describe
 (add member #people2 class
      (build mod (build lex ill-dressed) head (build lex people))))

(m78! (class (m8 (lex people))) (member b2))
(m77! (class (m9 (head (m8)) (mod (m7 (lex ill-dressed))))) (member b2))

(m78! m77!)

 CPU time : 0.10

*
;; Newly inferred information:
;;
;; This ill-dressed person is a person.
;;

;The other person is Tobar's mother.
(describe (add possessor *people1 rel (build lex mother) object *people2))

(m83! (object b2) (possessor b1) (rel (m82 (lex mother))))

(m83!)

 CPU time : 0.01

*
;There is a fire.
(describe (add member #fire class (build lex fire)))

(m86! (mode (m15 (lex presumably)))
 (object (m85 (location (m16 (lex outdoor))) (object b3))))
(m84! (class (m14 (lex fire))) (member b3))

(m86! m84!)

 CPU time : 0.22

*
;; Newly inferred information:
;;
;; The fire is probably outdoor.
;;
```

```
; There is a place around the fire.
(describe (add object1 #aroundfire rel (build lex around) object2 *fire))

(m93! (mode (m15 (lex presumably)))
 (object (m92 (location (m16 (lex outdoor))) (object b4))))
(m91! (object1 b4) (object2 b3) (rel (m18 (lex around))))

(m93! m91!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; The place is probably outdoor.
;;

;There is a time when Tobar sat at the place around the fire.
(describe (add agent *people1 act (build action (build lex sit))
             location  *aroundfire
             time #sittime))

(m98! (mode (m15 (lex presumably)))
 (object (m97 (location (m16 (lex outdoor))) (object b1))))
(m96! (mode (m15)) (object (m95 (location (m16)) (object b1) (time b5))))
(m94! (act (m21 (action (m20 (lex sit))))) (agent b1) (location b4) (time b5))

(m98! m96! m94!)

 CPU time : 0.14

*
;; Newly inferred information:
;;
;; At this time Tobar is probably outdoor.
;;

;At the same time mother sat at the place around the fire.
(describe (add agent *people2 act (build action (build lex sit))
             location  *aroundfire
               time *sittime))

(m103! (mode (m15 (lex presumably)))
 (object (m102 (location (m16 (lex outdoor))) (object b2))))
(m101! (mode (m15)) (object (m100 (location (m16)) (object b2) (time b5))))
(m99! (act (m21 (action (m20 (lex sit))))) (agent b2) (location b4) (time b5))

(m103! m101! m99!)

 CPU time : 0.12

*
;; Newly inferred information:
;;
;; At this time mother is probably outdoor.
;;
```

```
;There is a current time (now1) that is the subinterval of the time when
;they sat.
(describe (add superinterval *sittime subinterval #now))

(m104! (subinterval b6) (superinterval b5))

(m104!)

 CPU time : 0.02

*
;There is a common meal
(describe
 (add member #commonmeal
      class (build mod (build lex common) head (build lex meal))))

(m107! (class (m5 (lex food))) (member b7))
(m106! (class (m4 (lex meal))) (member b7))
(m105! (class (m12 (head (m4)) (mod (m11 (lex common))))) (member b7))

(m107! m106! m105!)

 CPU time : 0.17

*
;;Newly inferred information:
;;
;; The common meal is a meal.
;; The common meal is food.
;;


;There is a time when the common meal was almost ready.
(describe (add object *commonmeal
             property (build mod (build lex almost) head (build lex ready))
             time #mealtime))

(m113! (object b7) (time b8))
(m112! (object b7)
 (property (m25 (head (m24 (lex ready))) (mod (m23 (lex almost))))) (time b8))

(m113! m112!)

 CPU time : 0.05

*
;current time is also the subinterval of the time when the meal was almost
;ready.
(describe (add superinterval *mealtime subinterval *now))

(m114! (subinterval b6) (superinterval b8))

(m114!)

 CPU time : 0.04

*
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;The mother ... peared at her son through the oam of the bubbling stew.
;;It had been a long time since his last ceilidh and Tobar had changed
;;greatly.


;There is last ceilidh
(describe (add member #lastceilidh class (build lex ceilidh)))

(m116! (class (m115 (lex ceilidh))) (member b9))

(m116!)

 CPU time : 0.18

*
;Ceilidh is an unknown word.
(describe (add object (build lex ceilidh)
             property (build lex unknown)))

(m117! (object (m115 (lex ceilidh))) (property (m36 (lex unknown))))

(m117!)

 CPU time : 0.01

*
;It was Tobar's last ceilidh.
(describe (add possessor *people1
             rel (build lex "last ceilidh")
             object *lastceilidh))

(m122! (object b9) (possessor b1) (rel (m121 (lex last ceilidh))))

(m122!)

 CPU time : 0.01

*
;There is a time for the last ceilidh.
(describe (add object *lastceilidh time #lasttime))

(m123! (object b9) (time b10))

(m123!)

 CPU time : 0.04

*
;There is a finish time of last ceilidh, and the finish time is the final
;subinterval of the last ceilidh.
(describe
   (add superinterval *lasttime final-subinterval #lastfinishtime))

(m129! (mode (m15 (lex presumably)))
 (object (m128 (subclass (m115 (lex ceilidh))) (superclass (m2 (lex event))))))
```

```
(m127! (initial-subinterval (m125 (arg2 b10) (skf initial-time-of)))
 (superinterval b10))
(m126! (after b11) (before (m125)))
(m124! (final-subinterval b11) (superinterval b10))
(m118! (class (m2)) (member b9))

(m129! m127! m126! m124! m118!)

 CPU time : 0.07

*
;; Newly inferred information:
;;
;; There is an initial time of last ceilidh.
;; The initial time of last ceilidh is before the finish time.
;; Last ceilidh is an event.
;; Probably ceilidh is subclass of event.
;;


;It has been a long time since last time.
(describe (add before *lastfinishtime
               after *now
               duration (build lex long\ time)))

(m132! (after b6) (before (m125 (arg2 b10) (skf initial-time-of))))
(m131! (after b6) (before b11))
(m130! (after b6) (before b11) (duration (m42 (lex long time))))

(m132! m131! m130!)

 CPU time : 0.03

*
;; Newly inferred information:
;;
;; The initial time of last ceilidh is before current time.
;;


;At a time, mother found that Tobar had changed greatly in a period of time.
(describe (add agent *people2
             act (build action (build lex find)
                    object (build agent *people1
                                 act (build action (build mode (build lex greatly)
                                                          head (build lex change)))
                                 time #changetime))
             time #findtime))

(m135!
 (act (m134 (action (m56 (lex find)))
       (object
        (m133
         (act (m60
               (action
                (m59 (head (m58 (lex change))) (mode (m57 (lex greatly)))))))
         (agent b1) (time b12)))))
```

```
 (agent b2) (time b13))

(m135!)

 CPU time : 0.03

*
;The time when mother found Tobar had changed is a superinterval of current
;time.
(describe (add subinterval *now superinterval *findtime))

(m136! (subinterval b6) (superinterval b13))

(m136!)

 CPU time : 0.05

*
;The time period When Tobar changed began at the final subinterval of las
;ceilidh.
(describe
   (add initial-subinterval *lastfinishtime superinterval *changetime))

(m145!
 (last (m143 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
        (time (m140 (arg2 b1) (skf last-meet-time)))))
 (next
  (m144 (act (m142)) (agent b1) (time (m138 (arg2 b1) (skf this-meet-time))))))
(m141! (final-subinterval b11) (superinterval (m140)))
(m139! (subinterval b6) (superinterval (m138)))
(m137! (initial-subinterval b11) (superinterval b12))
(m131! (after b6) (before b11))
(m130! (after b6) (before b11) (duration (m42 (lex long time))))

(m145! m141! m139! m137! m131! m130!)

 CPU time : 0.05

*

;; Newly inferred information:
;;
;; At the time when mother found Tobar changed, Tobar and mother had a meeting.
;; The current time is the subinterval of this meeting time.
;; There is another meeting which is Tobar and mother's last meeting.
;; The last meeting had a time, and this time ended at the inital time of
;;          Tobar's change.
;; The end time of last meeting is long time before current time.
;;


;The period of time when Tobar changed ended at current time.
(describe (add final-subinterval *now superinterval *changetime))

(m146! (final-subinterval b6) (superinterval b12))

(m146!)
```

```
 CPU time : 0.05

*
;There is this ceilidh.
(describe (add member #thisceilidh class (build lex ceilidh)))

(m147! (class (m115 (lex ceilidh))) (member b14))

(m147!)

 CPU time : 0.20

*
;This ceilidh is the next one of last ceilidh.
(describe (add last *lastceilidh next *thisceilidh))

(m151! (last b9) (next b14))

(m151!)

 CPU time : 0.03

*
;There is a time for time ceilidh.
(describe (add object *thisceilidh time #thistime))

(m152! (object b14) (time b15))

(m152!)

 CPU time : 0.03

*
;Current time is a subinterval of the time of this ceilidh.
(describe (add subinterval *now superinterval *thistime))

(m153! (subinterval b6) (superinterval b15))

(m153!)

 CPU time : 0.02

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

;;As they ate, Tobar told of his past year.

;There is a new current time (now2) after the previous time (now1).
(describe (add before *now after #now))

(m157! (after b16) (before b11) (duration (m42 (lex long time))))
(m156! (after b16) (before b11))
(m155! (after b16) (before (m125 (arg2 b10) (skf initial-time-of))))
(m154! (after b16) (before b6))

(m157! m156! m155! m154!)
```

```
 CPU time : 0.09

*
;; Newly inferred information:
;;
;; Current time (now2) is long time after the final time of last ceilidh.
;; Current time is after the initial time of last ceilidh.
;;


;At a time, they eat meal.
(describe (add agent *people1
             act (build action (build lex eat) object *commonmeal)
               time #eattime))

(m159! (act (m158 (action (m26 (lex eat))) (object b7))) (agent b1) (time b17))

(m159!)

 CPU time : 0.03

*
(describe (add agent *people2
             act (build action (build lex eat) object *commonmeal)
               time *eattime))

(m160! (act (m158 (action (m26 (lex eat))) (object b7))) (agent b2) (time b17))

(m160!)

 CPU time : 0.03

*
;At the same time, Tobar told of his past year to his mother.
(describe (add agent *people1
             act (build action (build lex tell)
                              object (build skf "past-year-of" arg2 *people1))
             to *people2
             time *eattime))
(m164! (act (m51 (action (m49 (lex tell))) (object (m50 (lex story)))))
 (agent b1) (time b17))
(m163! (act (m74 (action (m49)) (object (m73 (arg2 b1) (skf past-year-of)))))
 (time b17) (to b2))
(m162! (act (m74)) (agent b1) (time b17))
(m161! (act (m74)) (agent b1) (time b17) (to b2))

(m164! m163! m162! m161!)

 CPU time : 0.05

*
;; Newly inferred information:
;;
;; Tobar told story.
;;
```

```
;The current time is a subinterval of their eating time.
(describe (add subinterval *now superinterval *eattime))

(m178! (after b6)
 (before
  (m170 (arg2 (m169 (arg2 b1) (skf last-meet-time-of))) (skf final-time-of))))
(m177! (after b6) (before (m170)) (duration (m46 (lex one year))))
(m176! (after b16) (before (m170)))
(m175!
 (last (m173 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
        (time (m169))))
 (next
  (m174 (act (m142)) (agent b1)
   (time (m167 (arg2 b1) (skf this-meet-time-of))))))
(m172! (after b16) (before (m170)) (duration (m46)))
(m171! (final-subinterval (m170)) (superinterval (m169)))
(m168! (subinterval b16) (superinterval (m167)))
(m166! (after b16) (before b6) (duration (m27 (lex short))))
(m165! (subinterval b16) (superinterval b17))
(m154! (after b16) (before b6))
(m131! (after b6) (before b11))
(m130! (after b6) (before b11) (duration (m42 (lex long time))))

(m178! m177! m176! m175! m172! m171! m168! m166! m165! m154! m131! m130!)

 CPU time : 0.16

*
;; Newly inferred information:
;;
;; Current time (now2) is shortly after previous time (now1).
;; Tobar and mother had this meeting at a time.
;; Current time (now2) is the subinterval of this meeting time.
;; Tobar and mother had last meeting at a time.
;; The final time of last meeting is one year before current time.
;; THe final time of last meeting is one year before previous time (now1).
;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;Then all too soon, their breif ceilidh over.


;There is a new current time (now3) shortly after previous time (now2).
(describe (add before *now after #now duration (build lex short)))

(m185! (after b18) (before b11) (duration (m42 (lex long time))))
(m184! (after b18) (before b6))
(m183! (after b18) (before (m125 (arg2 b10) (skf initial-time-of))))
(m182! (after b18) (before b11))
(m181! (after b18)
 (before
  (m170 (arg2 (m169 (arg2 b1) (skf last-meet-time-of))) (skf final-time-of))))
(m180! (after b18) (before b16))
(m179! (after b18) (before b16) (duration (m27 (lex short))))
(m157! (after b16) (before b11) (duration (m42)))
```

```
(m156! (after b16) (before b11))

(m185! m184! m183! m182! m181! m180! m179! m157! m156!)

 CPU time : 0.18

*
;; Newly inferred information:
;;
;; This new current time (now3) is after the first current time (now1).
;; This new current time is long time after the final time of last ceilidh.
;; This new current time is after the initial time of last ceilidh.
;; This new current time is after the final time of last meeting.
;;


;This ceilidh is Tobar's this ceilidh.
(describe (add possessor *people1
             rel (build lex "this ceilidh")
             object *thisceilidh))

(m187! (object b14) (possessor b1) (rel (m186 (lex this ceilidh))))

(m187!)

 CPU time : 0.02

*
;This ceilidh is mother's this ceilidh.
(describe (add possessor *people2
             rel (build lex "this ceilidh")
             object *thisceilidh))

(m188! (object b14) (possessor b2) (rel (m186 (lex this ceilidh))))

(m188!)

 CPU time : 0.04

*
;This ceilidh is over at current time.
(describe (add agent *thisceilidh act (build action (build lex finish))
               time *now))

(m198! (mode (m15 (lex presumably)))
 (object
  (m197 (subclass (m115 (lex ceilidh))) (superclass (m1 (lex gathering))))))
(m196! (subinterval b16) (superinterval b15))
(m195! (after b16) (before (m191 (arg2 b15) (skf initial-time-of))))
(m194! (after b6) (before (m191)))
(m193! (initial-subinterval (m191)) (superinterval b15))
(m192! (after b18) (before (m191)))
(m190! (final-subinterval b18) (superinterval b15))
(m189! (act (m30 (action (m29 (lex finish))))) (agent b14) (time b18))
(m184! (after b18) (before b6))
(m180! (after b18) (before b16))
(m153! (subinterval b6) (superinterval b15))
```

```
(m150! (mode (m15)) (object (m149 (class (m1)) (member b14))))
(m148! (class (m2 (lex event))) (member b14))
(m129! (mode (m15)) (object (m128 (subclass (m115)) (superclass (m2)))))

(m198! m196! m195! m194! m193! m192! m190! m189! m184! m180! m153! m150! m148!
 m129!)

 CPU time : 0.43

*
;; Newly inferred information:
;;
;; Current time is the final time of this ceilidh.
;; There is an initial time of this ceilidh.
;; The initial time is before the current time.
;; The initial time is before the first previous time (now1).
;; The initial time is beofore the second previous time (now2).
;; The second previous time is a subinterval of this ceilidh.
;; This ceilidh is an event.
;; This ceilidh is a gathering.
;; Ceilidh is probably a subclass of gathering.
;;


;This ceilidh is brief.
(describe (add object *thisceilidh property (build lex brief)))

(m209! (after (m191 (arg2 b15) (skf initial-time-of)))
 (before
  (m170 (arg2 (m169 (arg2 b1) (skf last-meet-time-of))) (skf final-time-of))))
(m208! (after (m191)) (before (m170)) (duration (m46 (lex one year))))
(m207! (after (m191)) (before (m125 (arg2 b10) (skf initial-time-of))))
(m206! (subinterval (m191)) (superinterval b12))
(m205! (after (m191)) (before b11))
(m204! (after (m191)) (before b11) (duration (m42 (lex long time))))
(m203! (after b18) (before b6) (duration (m27 (lex short))))
(m202! (after b6) (before (m191)) (duration (m27)))
(m201! (after b16) (before (m191)) (duration (m27)))
(m200! (after b18) (before (m191)) (duration (m27)))
(m199! (object b14) (property (m39 (lex brief))))
(m195! (after b16) (before (m191)))
(m194! (after b6) (before (m191)))
(m192! (after b18) (before (m191)))
(m185! (after b18) (before b11) (duration (m42)))
(m184! (after b18) (before b6))
(m183! (after b18) (before (m125)))
(m182! (after b18) (before b11))
(m181! (after b18) (before (m170)))
(m180! (after b18) (before b16))
(m179! (after b18) (before b16) (duration (m27)))
(m178! (after b6) (before (m170)))
(m176! (after b16) (before (m170)))
(m166! (after b16) (before b6) (duration (m27)))
(m157! (after b16) (before b11) (duration (m42)))
(m156! (after b16) (before b11))
(m155! (after b16) (before (m125)))
(m154! (after b16) (before b6))
```

```
(m132! (after b6) (before (m125)))
(m131! (after b6) (before b11))
(m130! (after b6) (before b11) (duration (m42)))

(m209! m208! m207! m206! m205! m204! m203! m202! m201! m200! m199! m195! m194!
 m192! m185! m184! m183! m182! m181! m180! m179! m178! m176! m166! m157! m156!
 m155! m154! m132! m131! m130!)

 CPU time : 0.46

*

;; Newly inferred information:
;;
;; The time between the inital and the final time of this ceilidh is short.
;; The initial time of this ceilidh is short after the first previous time (now1).
;; The initial time of this ceilidh is short after the second previous time (now2).
;; Current time is short after the first previous time (now1).
;; The initial time of this ceilidh is long after the final time of last one.
;; The initial time of this ceilidh is after the initial time of last one.
;; The initial time of this ceilidh is one year after the final time of last meeting.
;; The initial time of this ceilidh is a subinterval of Tobar's change time.
;; The initial time of this ceilidh is after the inital time of last ceilidh.
;;


;; Ask Cassie what "ceilidh" means:
^
--> (defineNoun "ceilidh")
 Definition of ceilidh:
 Probable Class Inclusions: event, gathering,
 Possible Actions: finish,
 Possible Properties: brief,
 Possessive: people this ceilidh, people last ceilidh, ill-dressed people
this ceilidh, ill-dressed people last ceilidh,
 Possible Locations: outdoor,
 Possible Frequency: long time,
 Possible Durations: short,
 Possible Concurrences: people find m133, people sit, people eat meal,
people tell story, people tell something,
nil



 CPU time : 0.45

*
;Cassie didn't get the frequency of one year, because SNePS didn't infer
;that the two "next meeting" are the same meeting from Rule 25. But all the
;antecedant of Rule 25 are satisfied. For example:

(describe m145)

(m145!
 (last (m143 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
        (time (m140 (arg2 b1) (skf last-meet-time)))))
 (next
```

```
  (m144 (act (m142)) (agent b1) (time (m138 (arg2 b1) (skf this-meet-time))))))

(m145!)

 CPU time : 0.00

*
(describe m174)

(m174 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
 (time (m167 (arg2 b1) (skf this-meet-time-of))))

(m174)

 CPU time : 0.00

*
;Add these information again.

(describe (add agent b1 act m142 time m138))

(m144! (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
 (time (m138 (arg2 b1) (skf this-meet-time))))

(m144!)

 CPU time : 0.05

*
(describe (add last m173 next m174))

(m221!
 (equiv
  (m173 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
   (time (m169 (arg2 b1) (skf last-meet-time-of))))
  (m143 (act (m142)) (agent b1) (time (m140 (arg2 b1) (skf last-meet-time))))))
(m220!
 (equiv
  (m174 (act (m142)) (agent b1)
   (time (m167 (arg2 b1) (skf this-meet-time-of))))
  (m144! (act (m142)) (agent b1)
   (time (m138 (arg2 b1) (skf this-meet-time))))))
(m175! (last (m173)) (next (m174)))

(m221! m220! m175!)

 CPU time : 0.15


*
;; Newly inferred information:
;;
;; The two "last meeting" are same meeting.
;;


;Add the newly inferred information again.
```

```
(describe (add equiv m143 equiv m173))

(m229! (after b6)
 (before
  (m170 (arg2 (m169 (arg2 b1) (skf last-meet-time-of))) (skf final-time-of)))
 (duration (m42 (lex long time))))
(m228! (after b16) (before (m170)) (duration (m42)))
(m227! (after b18) (before (m170)) (duration (m42)))
(m226! (after (m191 (arg2 b15) (skf initial-time-of))) (before (m170))
 (duration (m42)))
(m225! (after b16) (before b11) (duration (m46 (lex one year))))
(m224! (after b6) (before b11) (duration (m46)))
(m223! (after (m191)) (before b11) (duration (m46)))
(m222! (equiv (m170) b11))
(m221!
 (equiv
  (m173 (act (m142 (action (m54 (lex meet))) (object b2))) (agent b1)
   (time (m169)))
  (m143 (act (m142)) (agent b1) (time (m140 (arg2 b1) (skf last-meet-time))))))
(m209! (after (m191)) (before (m170)))
(m208! (after (m191)) (before (m170)) (duration (m46)))
(m205! (after (m191)) (before b11))
(m204! (after (m191)) (before b11) (duration (m42)))
(m185! (after b18) (before b11) (duration (m42)))
(m182! (after b18) (before b11))
(m181! (after b18) (before (m170)))
(m178! (after b6) (before (m170)))
(m177! (after b6) (before (m170)) (duration (m46)))
(m176! (after b16) (before (m170)))
(m172! (after b16) (before (m170)) (duration (m46)))
(m157! (after b16) (before b11) (duration (m42)))
(m156! (after b16) (before b11))
(m131! (after b6) (before b11))
(m130! (after b6) (before b11) (duration (m42)))

(m229! m228! m227! m226! m225! m224! m223! m222! m221! m209! m208! m205! m204!
 m185! m182! m181! m178! m177! m176! m172! m157! m156! m131! m130!)

 CPU time : 0.77

*

;; Newly inferred information:
;;
;; The final time of the two "last meeting" are same.
;; The final time of last ceilidh is one year before the inital time of this one.
;; The final time of last ceilidh is one year before the first previous time (now1).
;; The final time of last ceilidh is one year before the second previous time (now2).
;; The final time of last meeting is long time before the inital time of
;;           this ceilidh.
;; The final time of last meeting is long time before the current time.
;; The final time of last meeting is long time before the second previous
;;           time (now2).
;; The final time of last meeting is long time before the first previous time (now1).
;;
```

```
;; Ask Cassie what "ceilidh" means:
^
--> (defineNoun "ceilidh")
 Definition of ceilidh:
 Probable Class Inclusions: event, gathering,
 Possible Actions: finish,
 Possible Properties: brief,
 Possessive: people this ceilidh, people last ceilidh, ill-dressed people
this ceilidh, ill-dressed people last ceilidh,
 Possible Locations: outdoor,
 Possible Frequency: one year, long time,
 Possible Durations: short,
 Possible Concurrences: people find m133, people sit, people eat meal,
people tell story, people tell something, people meet people,
nil



 CPU time : 0.32

*

End of /home/geograd/junxu/cse740/project/ceilidh.demo demonstration.


 CPU time : 7.16
```

Appendix B: New Functions in Noun Algorithm

```
;;;-----------------------------------------------------------------------------
;;;                              LOCATIONS SECTION
;;;-----------------------------------------------------------------------------

;;;-----------------------------------------------------------------------------
;;;
;;;   function: findLocations
;;;                                                              created: jx 2004
;;;-----------------------------------------------------------------------------
(defun findLocations (noun)
  "Find the location where you can find all things which are members of the
class 'noun'"
  (let (locations)
    (cond
     ((and (setf locations (append locations (location-rule noun))) *dmode*)
      locations)
     ((and (setf locations (append locations (location-& noun))) *dmode*)
      locations)
     (t locations))))


;;;-----------------------------------------------------------------------------
;;;
;;;   function: location-rule
;;;                                                              created: jx 2004
;;;-----------------------------------------------------------------------------
(defun location-rule (noun)
  "Find the locations of all members of the class 'noun', where 'noun' is a
category."
  #3! ((find (compose location- ! object member- ! class lex) ~noun
           (compose location- cq- ! ant class lex) ~noun)))

;;;-----------------------------------------------------------------------------
;;;
;;;   function: location-&
;;;                                                              created: jx 2004
;;;-----------------------------------------------------------------------------
(defun location-& (noun)
  "Find the locations of all members of the class 'noun', where 'noun' is a
category."
  #3! ((find (compose location- ! object member- ! class lex) ~noun
           (compose location- cq- ! &ant class lex) ~noun)))


;;;-----------------------------------------------------------------------------
;;;
;;;   function: findProbableLocations
;;;                                                              created: jx 2004
;;;-----------------------------------------------------------------------------
(defun findProbableLocations (noun)
  "Find the location that presumely you can find all things which are members of
the class 'noun'"
  (let (locations)
    (cond
```

```
     ((and (setf locations (append locations (location-presume noun))) *dmode*)
      locations)
     ((and (setf locations (append locations (location-&-presume noun))) *dmode*)
      locations)
     (t locations))))

;;;---------------------------------------------------------------------------
;;;
;;;   function: location-presume
;;;                                                            created: jx 2004
;;;---------------------------------------------------------------------------
(defun location-presume (noun)
  "Find the presumed locations of all members of the class 'noun', where 'noun'
is a category."
  #3! ((find (compose location- object member- ! class lex) ~noun
           (compose location- object- mode lex) "presumably"
           (compose location- object- cq- ! ant class lex) ~noun)))


;;;---------------------------------------------------------------------------
;;;
;;;   function: location-&-presume
;;;                                                            created: jx 2004
;;;---------------------------------------------------------------------------
(defun location-&-presume (noun)
  "Find the presumed locations of all members of the class 'noun', where 'noun'
is a category."
  #3! ((find (compose location- object member- ! class lex) ~noun
           (compose location- object- mode lex) presumably
           (compose location- object- cq- ! &ant class lex) ~noun)))


;;;---------------------------------------------------------------------------
;;;
;;;   function: findPossibleLocations
;;;                                                            created: jx 2004
;;;---------------------------------------------------------------------------
(defun findPossibleLocations (noun)
  "Find the location you can find at least one thing which is a member of the
class 'noun'."
  (let (locations)
    (cond
     ((and (setf locations (append locations (location-possible noun))) *dmode*)
      locations)
     ((and (setf locations
                 (append locations (location-possible-presume noun))) *dmode*)
      locations)
     (t locations))))


;;;---------------------------------------------------------------------------
;;;
;;;   function: location-possible
;;;                                                            created: jx 2004
;;;---------------------------------------------------------------------------
(defun location-possible (noun)
  #3! ((find (compose location- ! object member- ! class lex) ~noun)))
```

```
;;;--------------------------------------------------------------------------
;;;
;;;   function: location-possible-presume
;;;                                                         created: jx 2004
;;;--------------------------------------------------------------------------
(defun location-possible-presume (noun)
  #3! ((find (compose location- object member- ! class lex) ~noun
          (compose location- object- ! mode lex) "presumably")))


;;;--------------------------------------------------------------------------
;;;                           EVENT FREQUENCY SECTION
;;;--------------------------------------------------------------------------

;;;--------------------------------------------------------------------------
;;;
;;;   function: findPossibleFrequency
;;;                                                         created: jx 2004
;;;--------------------------------------------------------------------------
(defun findPossibleFrequency (noun)
 "Find the frequency of 'noun' if 'noun' is an event."
(set-difference (find-this noun) '(nil)))


;;;--------------------------------------------------------------------------
;;;
;;;   function: find-this
;;;                                                         created: jx 2004
;;;--------------------------------------------------------------------------
(defun find-this (noun)
 "find all the objects belongs to 'noun'."
 (let (thisnouns results)
   (setf thisnouns
       #3! ((find (compose member- ! class lex) ~noun)))
   (mapcan #'(lambda (thisone)
             (set-difference (find-next noun thisone) '(nil))) thisnouns)))


;;;--------------------------------------------------------------------------
;;;
;;;   function: find-next
;;;                                                         created: jx 2004
;;;--------------------------------------------------------------------------
(defun find-next (noun thisone)
  "find the other 'noun' which is the next of this 'noun'."
  (let (nextnouns results)
    (setf nextnouns
        #3! ((find (compose next- ! last) ~thisone
                  (compose member- ! class lex) ~noun)))
    (mapcan #'(lambda (nextone) (frequency-duration thisone nextone))
nextnouns)))


;;;--------------------------------------------------------------------------
;;;
```

```
;;;   function: frequency-duraion
;;;                                                          created: jx 2004
;;;-----------------------------------------------------------------------------
(defun frequency-duration (thisone nextone)
  #3! ((find (compose duration- ! before final-subinterval- ! superinterval
                      time- ! object) ~thisone
           (compose duration- ! after initial-subinterval- ! superinterval
                      time- ! object) ~nextone)))



;;;-----------------------------------------------------------------------------
;;;                              EVENT DURATION SECTION
;;;-----------------------------------------------------------------------------

;;;-----------------------------------------------------------------------------
;;;
;;;   function: findPossibleDurations
;;;                                                          created: jx 2004
;;;-----------------------------------------------------------------------------
(defun findPossibleDurations (noun)
 "Find the duration of a 'noun' if 'noun' is an event."
 #3! ((find (compose duration- ! before initial-subinterval- ! superinterval
                     time- ! object member- ! class lex) ~noun
            (compose duration- ! after final-subinterval- ! superinterval
                     time- ! object member- ! class lex) ~noun)))



;;;-----------------------------------------------------------------------------
;;;                              EVENT CONCURRENCE SECTION
;;;-----------------------------------------------------------------------------

;;;-----------------------------------------------------------------------------
;;;
;;;   function: findPossibleConcurrences
;;;                                                          created: jx 2004
;;;-----------------------------------------------------------------------------
(defun findPossibleConcurrences (noun)
 "Find a list of agent(s) and act(s) happed at the same time with 'noun' if
'noun' is an event."
 (let (concurrence)
   (cond
    ((and (setf concurrence (append concurrence (co-agent-object noun))) *dmode*)
     concurrence)
    (t concurrence)
    )))

;;;-----------------------------------------------------------------------------
;;;
;;;   function: co-agent-object
;;;                                                          created: jx 2004
;;;-----------------------------------------------------------------------------
(defun co-agent-object (noun)
 "Find agents who perform actions during the period 'noun' happens."
 (let (agents)
   (setf agents #3! ((find (compose agent- ! time superinterval- ! subinterval
```

```
                           subinterval- ! superinterval time- ! object
                           member- ! class lex) ~noun)))
   (mapcar #'(lambda (ag) (co-action-object noun ag)) agents)))


;;;-----------------------------------------------------------------------------
;;;
;;;   function: co-action-object
;;;                                                           created: jx 2004
;;;-----------------------------------------------------------------------------
(defun co-action-object (noun ag)
 "Find actions performed by 'ag'."
 (let (actions)
   (setf actions #3! ((find (compose action- act- ! time superinterval- !
                            subinterval subinterval- ! superinterval
                            time- ! object member- ! class lex) ~noun
                           (compose action- act- ! agent) ~ag)))
   (mapcar #'(lambda (act) (co-object noun ag act)) actions)))


;;;-----------------------------------------------------------------------------
;;;
;;;   function: co-object
;;;                                                           created: jx 2004
;;;-----------------------------------------------------------------------------
(defun co-object (noun ag act)
 "Find objects on which 'ag' performs 'act'."
 (let (objects)
   (setf objects #3! ((find (compose object- act- ! time superinterval- !
                            subinterval subinterval- ! superinterval
                            time- ! object member- ! class lex) ~noun
                           (compose object- act- ! agent) ~ag
                      (compose object- act- ! act action) ~act)))
   (if (not (null objects))
       (mapcar #'(lambda (obj) (list ag act obj)) objects)
     (list ag act))))
```

Reference

Almeida, M. J., 1995. Time in Narratives. In Judith Felson Duchan, Gail A. Bruder, & Lynne E. Hewitt (eds.), *Deixis in Narrative: A Cognitive Science Perspective* (Hillsdale, NJ: Lawrence Erlbaum Associates): 159-189.

Rapaport, W. J., & Ehrlich, K, 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.

Rapaport, W. J., & Kibby, M. W. 2002. ROLE: Contextual Vocabulary Acquisition: From Algorithm to Curriculum.

Shapiro, S. C., 1978. Path-based and node-based inference in semantic networks. In D. Waltz, editor, *Tinlap-2: Theoretical Issues in Natural Languages Processing*, pages 219-225, New York, 1978. ACM.

Shapiro, S. C., & Rapaport, W. 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.

Shapiro, S. C., and Rapaport, W. J., 1992. The SNePS family. *Computer for Mathematic application*, 23(2-5): 243-275.

Shapiro, S. C., and Rapaport, W. 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.

Sternberg, R. J., and Powell, J. S., 1983. Comprehending Verbal Comprehension. *American Psychologist*, 38: 878-893.