A Contextual Vocabulary Acquisition Algorithm for Verbs Implemented in SNePS

Justin Del Vecchio

August 19th, 2002

1 Introduction

The goal of my summer research was to provide the Cognitive Agent of the SNePS System-an Intelligent Entity (CASSIE) an algorithm that would allow her to infer the meaning of verbs based on context. Creating such an algorithm was a daunting task requiring much precursory work with SNePS, Ehrlich's verb algorithm and Lisp. In order to fully prepare myself for this undertaking I spent the Spring 2002 semester working on Ehrlich's verb algorithm in Dr. Rapaport's Advanced Knowledge Representation class (CSE 663). My work in this course consisted of two items: One, bi-weekly presentations made to Dr. Rapaport and the class detailing the functioning of Ehrlich's verb algorithm and two, a comprehensive final paper. During my class presentations many salient points were made by Dr. Rapaport and my classmates with respect to possible improvements to Ehrlich's verb algorithm. A list of the top features and improvements to be added to the verb algorithm can be found at http://www.cse.Buffalo.EDU/ jmdv/CSE663PresFour/0/1.html. The reader should consider this as the starting point for my summer research. A complete listing of all my presentations for the spring course can be found at http://www.cse.Buffalo.EDU/ jmdv.

In late May I met with Dr. Rapaport to decide on list of specific changes I would attempt to implement for the verb algorithm. From the long list of potential candidates a few were obvious to both of us as being necessary changes for future versions of the verb algorithm. The following is an enumeration of those topics which were the primary focus of my research. Some directly affect the presentation quality of CASSIE's output information about the unknown verb while others are technical details of interest only to future programmers who work on improving or refactoring the algorithm. Here are the points:

1. Avoid Ehrlich's scheme of directly asserting that the verb in the Agent/Act case frame has the property of being bitransitive, transitive, intransitive, etc. Instead, have the verb algorithm dynamically evaluate each Agent/Act case frame where the verb in question exists and return its type (bitransitive, transitive, etc.) based on the arc structure of the case frame.

2. Avoid the 'cross talk' issue that made Ehrlich's results confusing at times. It would report results that mixed the agents, subjects and idiect objects of sentences. For example if Ehrlich's verb algorithm had the two sentences:

   "Derek threw the baseball."
   "Godzilla threw the bus to King Kong."

   The verb algorithm would report something similar to:

"A (human, monster) can throw a (baseball, ball, bus, vehicle) to another (monster)."

The algorithm reported confusing results as it was possible to interpret that a human can throw a bus to a monster. This, of course, is not supported by the original two sentences.

3. Introduce a debugging facility that would allow: One, future programmers the ability to debug the program and two, users to view a human readable account of what the verb algorithm was doing.

4. Allow CASSIE to do eductions based on the passages with the verb in question. In a nutshell if CASSIE has the sentence:

   "Bill threw the medicine ball to Tom."

   and she has background knowledge that Bill and Tom are primates, and therefore mammals and therefore animals she can create new versions of the sentence by substituting in information she knows to be true. CASSIE could come up with the following sentences based on the original and her background knowledge:

   "A human can throw a medicine ball to a human."
   "A primate can throw a medicine ball to a primate."
   "A mammal can throw a medicine ball to a mammal."
   ...

Implementing the above points required rewriting the entire verb algorithm. I considered reworking Ehrlich's verb algorithm to incorporate these changes but its architecture was not suited to address issues like cross-talk.

The bulk of this paper discusses the way CASSIE outputs information to the user when asked to define a verb. It also contains non-technical explanations of how the algorithm allows CASSIE to pull or educe information from her knowledge base when reasoning about the meaning of an unknown verb. Future programmers reading this paper who are interested in the technical details of how the algorithm works with Lisp and SNePS are directed to Appendix A. Those seeking a better understanding of the basic principles behind SNePS are directed to http://www.cse.Buffalo.EDU/ rapaport/snepsrdgs.html

2 Predicate Structure

The current version of the verb algorithm is interested in the predicate structure of the sentences containing the unknown verb. There are four basic types of predicate structures: bitransitive, transitive, intransitive and reflexive. Throughout this paper three of the four types are mentioned and the reader should be familiar with the properties of each type. For conciseness, transitive, bitransitive and reflexive verbs require a receiver of the action to complete their meaning while intransitive verbs do not require a receiver of the action to complete their meaning. The following are examples of each type:

Bitransitive - If an agent acts on an object and there is an indirect object of the action.

Example: "John drove the car home."

Transitive - If an agent acts on an object.

Example: "John drove the car."

Reflexive(after reflexive pronouns like myself, yourself, himself, etc.) - If an agent acts on itself.

Example: "John hated himself."

Intransitive - If an agent acts.

Example: "John fished in the stream."

Currently the verb algorithm is equipped to handle sentences of the type bitransitive, transitive and intransitive. Reflexive has been left out as it covers multiple cases. That is a sentence can be reflexive-bitransitive, reflexive-transitive, etc. The algorithm provides hooks that will allow this feature to be easily added in the future.

2.1 Current and Future Uses of Predicate Information

The predicate structure currently serves two purposes for the verb algorithm: One, it serves as the template for how results are out-putted to the user and two, it serves as a platform from which CASSIE can make eductions. The latter purpose is the more technically complex of the two and will be discussed in greater detail in sections 4.2 and 4.3.

Future uses of the predicate structure for guessing the meaning of an unknown verb are wide ranging but at the same time technically complex and difficult to model. Consider the following two sentences provided by Winograd (1983) which demonstrates how sentences can be reordered and still make sense:

"We parked the orange truck easily."
"The orange truck parked easily."

Here the same event is described by each sentence but with a different choice of subject for each. It is conceivable that a future verb algorithm would use the predicate structure of sentences in combination with the type of the components (that is the class membership of the agent, subject and indirect object(s)) to reorder the structure of sentences. Clearly this type of manipulation for the purpose of guessing a verb's meaning is in the distant future but providing CASSIE the ability to recognize different predicate structures is one small step in allowing her to manipulate knowledge in the manner prescribed by Winograd.

3 CASSIE's Background Knowledge

Creating a new verb algorithm required choosing a verb to model test cases that served as testable input to the algorithm. I chose the verb 'throw'. Its choice was completely arbitrary and future researchers may want to chose a specific verb for specific purposes. I also created a set of class hierarchies that the verb, agents, subjects and indirect objects fit into. The set of sentences and class hierarchies serve as CASSIE's 'brain' as the verb algorithm allows CASSIE to educe information from her mind.

3.1 The Test Sentences

Here is a listing of the test sentences I created for the verb algorithm. CASSIE was given each of the following sentences and they were modeled in her memory space using the Agent/Act case frame. The actual file that contains the Agent/Act case frames is named 'transitivity.demo'. Appendix B contains a complete listing of the files used (demo files and the algorithm itself) and URLs for where to find the files.

Bitransitive Sentences

"Derek threw the ball to Tino."
"Godzilla threw the bus to King Kong."
"The elephant threw water on its back."

Transitive Sentences

"Derek threw the ball."
"Godzilla threw the bus."
"Godzilla threw the airplane."
"Derek threw a look at Alfonso."
"Homer threw a ball." (Homer refers to an instance of a lemur)
"Horatio threw a banana." (Horatio refers to an instance of a chimp)

Intransitive Sentences

"Derek threw."
"Godzilla threw."

3.2 Class Hierarchies

In addition to the test sentences I provided CASSIE with the following class hierarchies for the agents, subjects, indirect objects and the verb itself. Figure 1 shows a hierarchy for Animals, Figure 2 for Walking Animals and Figure 3 for the verb throw. In much the way a human would look at Figure 1 and be able to reason that Derek is a member of the class humans, and therefore a member of the class primates and therefore a mammal and therefore an animal, CASSIE is provided with a data model and algorithm so she can educe the same information.

One fact needs to be addressed: creating a hierarchy is completely subjective. It is left to the knowledge engineer to decide how a specific person might model a possible hierarchy and how she would structure the elements in the hierarchy. Here, assuming the role of all knowing knowledge engineer, I created two separate hierarchies, one with walking animals at the pinnacle and another with animals at the pinnacle. Another knowledge engineer might have modeled these two hierarchies as one reasoning as follows: "Walking animals are a type of animal and therefore a subclass of animal".

I have chosen to model the hierarchies as two separate instances because I wanted to model a person who approaches the problem with a mind set that two distinct hierarchies are at play in the sentences. The individual I am trying to model would review the sentences and think, "I am seeing two hierarchies here. One is a traditional taxonomical one that I learned in biology the other is based on the physical properties of the animal, namely whether or not they can walk." Again, this

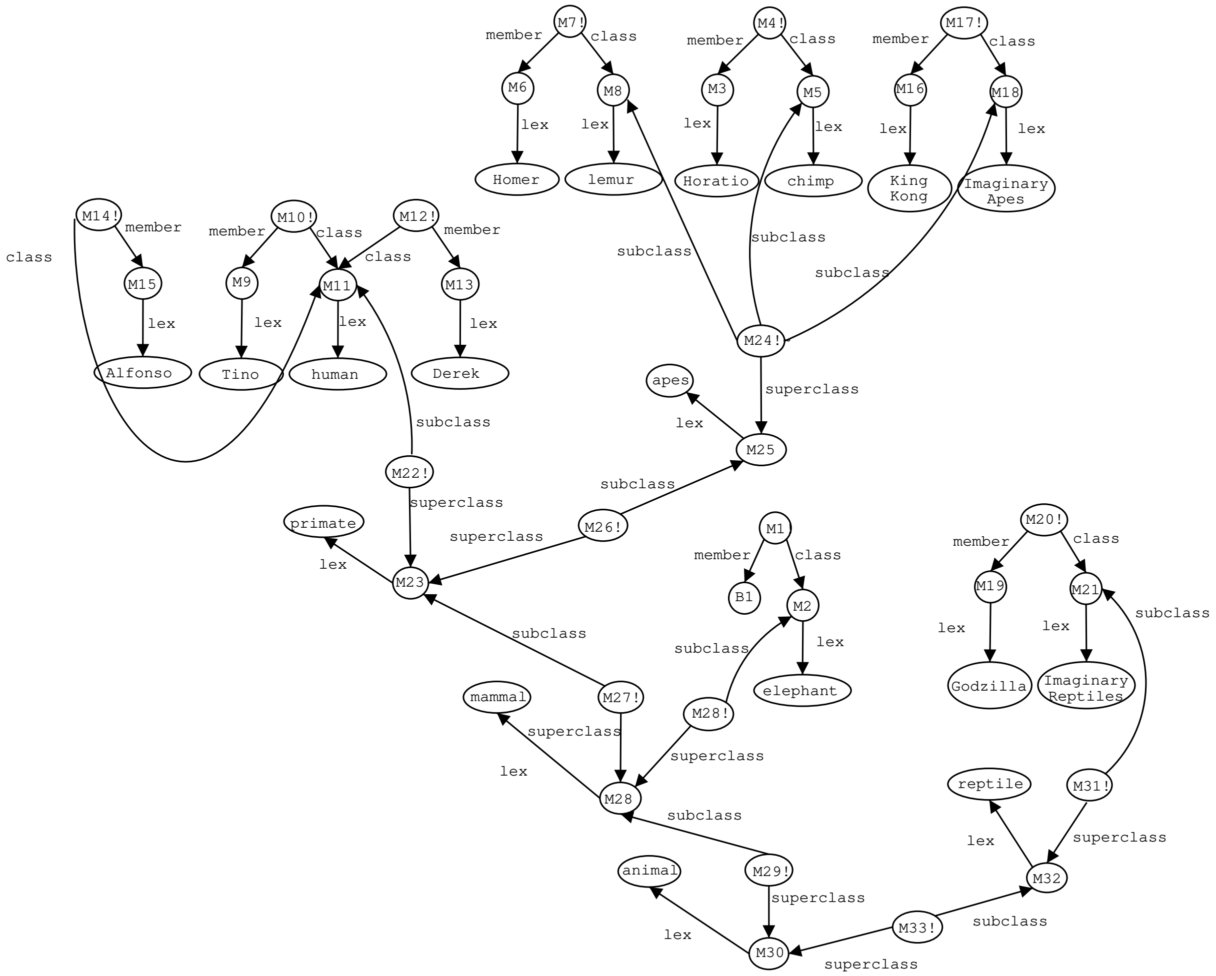

Figure 1: A Class Hierarchy for Animals.

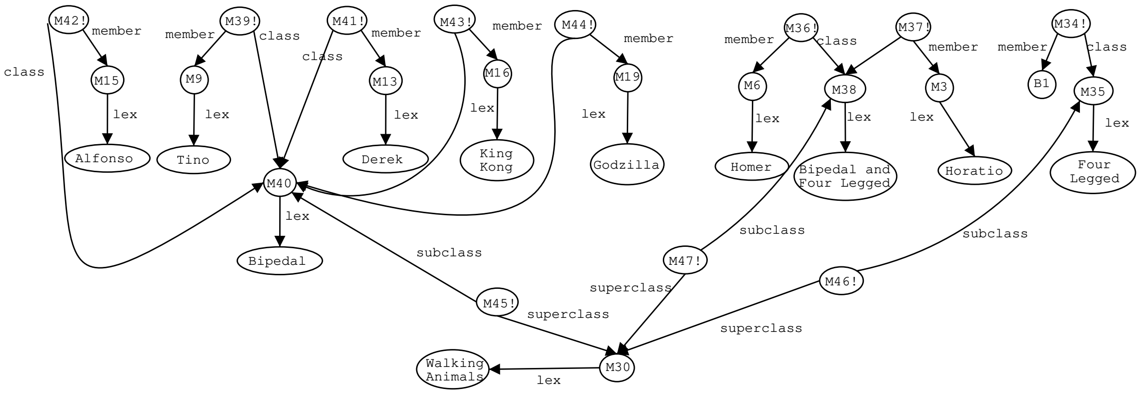

Figure 2: A Class Hierarchy for Walking Animals.

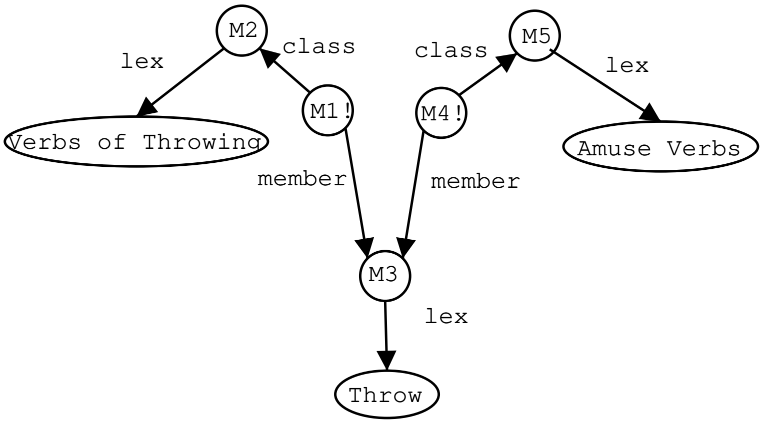

Figure 3: The Verb Hierarchy.

view is completely subjective and future researchers may frown upon this type of assumption. It was simply the first thing that struck me about all the sentences after creating them and therefore I coded the hierarchies from this perspective.

When CASSIE applies the verb algorithm she will inspect these hierarchies and attempt to do unification. Specifically she will review the agents, subjects and indirect objects for the different predicate types and attempt to do eductions for each type. This is described briefly in the Introduction and will be touched on in greater detail in section 4.

The verb hierarchy contains two categories of which throw belongs to. Both categories were prescribed by Levin (1993). Levin's research is one of the few resources where large numbers of verbs are categorized based not only on action but permissible predicate structures for sets of verbs. Future research should review these hierarchies and attempt to define a multilevel structure for verbs. I have ensured that the verb algorithm can recursively search multilevel structures for verb type but I did not research what possible multi-level hierarchies for verbs might be.

4 The Verb Algorithm's Output

In its current form the verb algorithm does not provide CASSIE with an enhanced ability to guess the meaning of unknown verbs and thus startle users with a dictionary-like definitions based solely on context. What it does provide CASSIE with is the ability to produce eductions from the knowledge base provided. These eductions will hopefully serve as the foundation for more complex eductions that will lead to a dictionary-like definition. With the current algorithm CASSIE now has an advanced reporting mechanism geared toward giving human readable results that shed light on the thought process she uses to arrive at her eductions. Here is CASSIE's output after applying the verb algorithm using the test sentences and hierarchies defined earlier (in 'transitivity.demo'):

```
"You want me to define the verb 'throw'.

######################## Introduction #####################################################

I'll start by looking at the predicate structure of the sentences I know that use 'throw'.
Here is what I know:
```

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

The next most common types of sentences that I know of that use 'throw' are of the form:
'A something can throw.'

'A something can throw something to something.'

####################### Verb Hierarchy ###

The verbs superclasses are: Verbs of Throwing- Levin 17.1 : Amuse Verbs - Levin 31.1 :

####################### Top Down Eduction ###

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

A animal can throw a physical object.
A Walking Animals can throw a physical object.

A Walking Animals can throw.
A animal can throw.

A Walking Animals can throw a physical object to a Walking Animals, animal.
A animal can throw a physical object to a Walking Animals, animal.

####################### Bottom Up Eduction ##

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

A human can throw a ball.
A imaginary reptiles can throw a bus, airplane.

A human can throw.
A imaginary reptiles can throw.

```
A human can throw a ball to a human, Bipedal.
A imaginary reptiles can throw a bus, airplane to a Bipedal, imaginary apes.
```

"

The output is broken into different sections, each delimited by `##### Section Name #####`. Each section is deserving of a brief overview of how CASSIE reaches her results.

4.1 Algorithm's Introduction

What CASSIE does for the introduction is quite straight forward. She reviews the predicate types for the verb in question(in this case 'throw') and based on the most common predicate type begins returning generic statements to the user. In this case she finds that most of the sentences are transitive, returns information about how transitive sentences work and then proceeds to do the same for both bitransitive and intransitive cases. This output is very generalized and is covered by other aspects of the verb algorithm. Future releases of the verb algorithm may excise this portion of the output though it is helpful to programmers who wish to verify the verb algorithm's accuracy.

4.2 Top Down Eduction

Creating an algorithm that would perform top down eduction was the research's grunt work and took a majority of the time. By 'top down eduction' I am referring to information CASSIE generates using the principles of unification in conjunction with the predicate structures and class hierarchies. She is essentially looking for substitutions for the agents, subjects and indirect objects of the sentences based on the class hierarchies in her knowledge base. CASSIE educes information by finding the most common predicate case and starting with that case, doing the following:

1. Find the most common agent for the given predicate type. CASSIE is looking up the class hierarchy for the pinnacle. She wishes to restate the sentence based on the hierarchy with the most instances. This deserves further explanation. Notice in the output that two agents ('animal' and 'Walking Animals') unify to the most general case and are reported back to the user. This is because CASSIE has six sentences and therefore six agents. In all cases the agent can unify to a hierarchy whose pinnacle is either 'animal' or 'Walking Animals'. The reader can verify this by looking at the transitive sentences and noting that the agent of each sentence is a leaf node for both the 'animal' and 'Walking Animals' hierarchies. Had five of the six cases unified to 'animal' and six of six unified to 'Walking Animals', CASSIE would have only reported back results for 'Walking Animals'.

2. The algorithm takes **only** those sentences that generalized to the majority agent(s) and analyzes the subject of this set of sentences (if necessary). This means if CASSIE had seven transitive sentences to begin with and five unified to a single hierarchy while two unified to another hierarchy, she would take those five sentences and unify their subjects ignoring the other two. It would not make sense to take all seven sentences and unify the subjects as the problem of cross talk might arise. The reader may also wonder why the other two sentences would not be unified as well. This is mostly a matter of preference. I simply wanted the

algorithm to report the most dominant cases and ignore the minority cases. It would be easy for future programmers to report this information is so desired.

3. The algorithm takes only those sentences returned from step two and generalize based on the indirect objects (if necessary).

4. The algorithm takes the next most common predicate case and runs steps one through three on this case. If no cases are left, the algorithm exits the analysis loop.

The four steps above generalize the sentences to the most common agent for a majority of the sentences. This type of eduction creates a unified view from the perspective of the agent but at the same time might exclude some of the sentences and does not do unification from the perspective of other predicate components, say the subject or indirect object. The goal here was to create a flexible mechanism to allow future developers to unify sentences in a variety of ways. Although it is not demonstrated in the example such a mechanism exists and can be the foundation for future work.

4.3 Bottom Up Eduction

This is a more tricky and is many ways a much more subjective analysis of the knowledge base possessed by CASSIE. Consider a hypothetical situation where I have ten transitive sentences using the verb 'throw' and for the agents of the sentences I know that six unify to the class primate. Referring to the hierarchy in Figure 1 it is seen that primate lies somewhere in the middle between being animal, a very general categorization, and being a chimp or a human, a very specific categorization. The algorithm attempts to allow CASSIE to look from the bottom up and educe something along the lines:

> "I know that the majority of my agents who throw are humans, apes and gorillas at the lowest level. I also know that if I unify these cases they are all animals at the highest level. Is there some intermediate class that these agents belong to which is not as generalized as animal yet not as specific as ape, gorilla or human?"

I wanted the algorithm to find this intermediary class that most of the agents belonged to but, ideally, not an overwhelming majority. I am hoping that this class can shed light on the type of thing that can do the verb in question. Of course this is extremely subjective and, interestingly, the amount of subjectiveness directly related to the amount of difficulty I had coding it into the algorithm. It is quite possible that the intermediary class found by CASSIE will show no useful information. I wanted this to be an exercise to see how difficult it would be to code for something that attempts to follow a crude statistical model.

Here is how bottom up eduction works. Starting with the most common predicate structure and working with each subsequent predicate case, CASSIE does:

1. Search from the bottom up. When she finds the first class that has 50% or more of the agents for the predicate case in question, she stops and report it as the median agent. An obvious problem is what happens if there is no class with a 50% majority. In this case CASSIE simply takes the first class with a majority and reports it back. If there are ties among majority classes, CASSIE reports back only those that are in separate hierarchies **and** are lowest in level in the classification hierarchy. The latter case means if CASSIE has eight agents that unify to both primate and mammal and mammal is at higher level in the hierarchy tree, CASSIE will only report back information for primate or the lower class.

2. Based on the set of sentences returned in step one CASSIE does bottom up eduction for the subjects of these sentences(if necessary).

3. Based on the set of sentences returned in step two, CASSIE does bottom up eduction for the indirect objects (if necessary).

4. The algorithm takes the next most common predicate case and runs steps one through three on this case. If no cases are left, the algorithm exists this analysis loop.

Bottom up may yield a class that is ripe for further analysis. Consider if CASSIE finds primates to be the most common class and a property of primates are that they use their hands to do numerous tasks, especially tasks involving objects smaller than themselves like balls. Potentially the verb algorithm could deduce that throwing is something primates use their hands to do with objects smaller than themselves. A staggering amount of background information would be required, not to mention a unification engine for rules, to make a guess as to what throw means in the manner described above. I simply wanted to provide the first in-roads to solving this problem.

5 Summary

The goal for my summer research was to create a verb algorithm that at the very least would allow CASSIE to perform unification for a set of sentences that used predicate components belonging to separate class hierarchies. I feel that I accomplished this goal and in the process created some useful and extendable code for future use. It is but a small step in creating a functioning verb algorithm and my opinion is that this work provides 5% of what a truly robust verb algorithm requires.

6 ToDo List

Here is a list of things that remain to be done or need to be re-visited in the future:

- Before continuing with another verb it would be a good idea to review Hasting's (1994) work. I read his work late in my summer research but the way he manipulates a sentences predicate structure in conjunction with the class membership of the agent, subject etc. in order to draw meanings for unknown verbs is very interesting. It could be the next foundation to build upon.

- Review the Agent/Act case frame and figure out how to better represent indirect objects. More than likely this will need to be assertional information, requiring a modification to the case frame and some modification to the search mechanisms. Upon implementing indirect objects via assertions, future researchers should review Levin's work to see how categories of verbs use prepositions in specific patterns.

- Review Levin's book and decide how best to represent verbs using a hierarchy. This is a very difficult task and researchers may want to find more information about the nature of verbs and how humans tend to categorize them.

References

[1] Winograd, Terry (1983), *"Language As A Cognitive Process"* (Menlo Park, California : Addison-Wesley Publishing Company) 311-325

[2] Hastings, Peter M., & Lytinen, Steven L. (1994b), "Objects, Actions, Nouns, and Verbs", *Proceedings of the 16th Annual Conference of the Cognitive Science Society* (Hillsdale, NJ: Lawrence Erlbaum Associates): 397-402.

[3] Levin, Beth (199e), *English Verb Classes and Alterations* (Chicago, IL: The University of Chicago Press): 111-296.

A A Technical Overview of the Verb Algorithm

Contained in the sections below are descriptions of each function in the verb algorithm. The ordering is (roughly) the order in which the methods are called. The ordering, of course, is not entirely sequential and certain methods are called more than once.

A.1 defineVerb - Entry Point of Algorithm

defineVerb is the coordinator of the entire verb algorithm. It kicks off a series of methods that lead the verb algorithm from beginning to end. Think of defineVerb as the driver of the entire algorithm. The arguments passed in are the verb to be defined and the trace (debugging) level. The following numbered items are either directly or indirectly called from defineVerb.

A.1.1 setTraceLevel - Setting Depth of Debugging

The method is located at the physical end of the file. It has two cases: One, untrace and two, trace. As named the untrace case traces no methods and trace traces all methods during execution of the algorithm. Note that these are the only two cases at this time. Future programmers are encouraged to add additional cases for customized debugging cases.

A.1.2 categorizeBasedOnPredicateStructure - Breaking Down the Predicate Structure

A driver for creating the lists of bitransitive, transitive and intransitive nodes in the SNePS network that reference the verb to be defined. The purpose of this method is to create lists of nodes representing each type of case.

A.1.3 isBitransitive, isTransitive, isInTransitive - Finding the Predicate Cases

These methods issue direct calls to SNePS that examine the case frames involving the verb and assign to lists of each the nodes that are bitransitive, transitive and intransitive. These lists will be globally accessible and will be used often by other methods.

A.1.4 createStructuresToRepresentEachInstanceOfVerb

As the name implies this method creates lists of Lisp structures to represent the nodes located in the bitransitive, intransitive and transitive lists. You can think of this method as creating a mirror image of the lists returned by *isBitransitive*, *isTransitive* and *isIntransitive* except instead or returning nodes it returns data bearing structures. Upon rereading the method it is a definite candidate for refactoring and could be combined with *categorizeBasedOnPredicateStructure* . The following information is stored in each structure:

- snepsNode
- predicateType
- agent
- agentHierarchyList
- act

- object
- indirect object
- prepositions
- superclass
- effects
- consequences

Each of the items represents information about that specific node instance (snepsNode). Refer to *Common Lisp* for an excellent description of structures. Think of structures as objects whose sole purpose is to serve as a data store.

A.1.5 instantiateInstances

Here a list is passed in corresponding to either the bitransitive, transitive, or intransitive case. The list is then populated with structures representing each node that is of the respective type. The three returned lists, bitransitiveList, transitiveList and bitransitiveList, are globally accessible and will be used throughout the algorithm at various points.

There are a host of utility methods called by *instantiateInstances*. Essentially each of the variables in the structure (agent, agentHierarchyList, act, etc.) has a corresponding utility method that finds the required information by typically making a direct call to SNePS. These methods are very short in length and straight forward. If the structures are not being instantiated with the correct information future programmers should refer to these utility methods and examine the SNePS code being invoked.

A.2 orderPredicateTypesByUsage

A difficult method to understand and rightly so, it is doing a complex operation. However, what it returns is quite simple and therefore the method does not require much explanation. Consider if the SNePS' network had 12 bitransitive case frames, 7 intransitive case frames and 2 transitive case frames for the verb in question. This method will simply return the following list:

((Bitransitive)(Intransitive)(Transitive))

Starting from the head and moving down the list the predicate cases are listed in decreasing order. If there were a tie between two, say bitransitive 7, transitive 7 and intransitive 2, the list returned would be:

((Bitransitive Transitive)(Intransitive))

Note that here the first element of the list is a list containing the two predicate types with equivalent numbers of nodes in the SNePS network. The list containing this information is called *cummulativeList*. This may inappropriately named.

A.3 introduceTheVerbAlgorithm

Responsible for the output:

```
"You want me to define the verb 'throw'.

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

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

The next most common types of sentences that I know of that use 'throw' are of the form:
     'A something can throw.'

     'A something can throw something to something.'
```

It essentially runs through the cumulative list of predicates and for each predicate type returns a generic sentence representing that type. Again, a complex method that might not be needed. It contains repetitive information and it is debatable that the introduction adds much information for the user. Future programmers can consider killing *introduceTheVerbAlgorithm* and its ancillary methods.

A.4 printPredicateUnificationTopDown

Extremely important method as it is responsible for handling the top down eduction and formatting the string returned to the user. Here is the output that *printPredicateUnificationTopDown* is responsible for:

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

A animal can throw a physical object.
A Walking Animals can throw a physical object.


A Walking Animals can throw.
A animal can throw.


A Walking Animals can throw a physical object to a Walking Animals, animal.
A animal can throw a physical object to a Walking Animals, animal.
```

For a summary of what the algorithm does in this instance refer to the Section 4.2. The most important thing to note is the mapping from the agents to their superclasses. Essentially the method creates a matrix that might look as follows:

Node Superclasses
M12 M71! M83! M19!
M17 M71! M22! M32!
M22 32!
M32

The matrix is mapping the node in the case frame to all its superclasses. The matrix is next manipulated so that the node which is the pinnacle of the hierarchy is found. This method and its ancillary methods are admittedly complex and should be analyzed carefully. However each function is logically named and the developer should refer to the function names and descriptions to see how the matrix is created and the pertinent information obtained. Once the developer understands the concept of the matrix and which node(s) she wants from the matrix she will understand how manipulations are made. Again, refer to Section 4.2 and then read the code. It will make more sense than looking at the code alone.

I would have, upon looking at this method again, broken it down considerably by creating many more logically named methods. It would improve the readability and make life easier on future programmers. Future programmers should chunk this portion of the algorithm into smaller methods and logically name them so readability is improved.

A.5 printPredicateUnificationBottomUp

Extremely important method as it is responsible for handling the bottom up eduction and formatting the string returned to the user. Here is the output that *printPredicateUnificationTopDown* is responsible for:

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

A human can throw a ball.
A imaginary reptiles can throw a bus, airplane.


A human can throw.
A imaginary reptiles can throw.


A human can throw a ball to a human, Bipedal.
A imaginary reptiles can throw a bus, airplane to a Bipedal, imaginary apes.
```

Works identically as *printPredicateUnificationTopDown* except it works from the bottom up. If you understand how that function works this one should basically be a mirror image except the manner in which it searches the matrix is slightly different. Refer to Section 4.3 for an English version of how this function operates.

B File Locations

I intend on placing the files required to run the verb algorithm and 'throw' demo in two separate locations:

1. A customized website I developed to be deployed by Dr. Rapaport. There will be a link to it from the CVA page maintained by Dr. Rapaport.
2. A directory named *verbAlgorithm2.0* placed somewhere in Dr. Rapaport's *projects* directory. The exact place is to be determined by him.

The contents will be as follows:

- *preliminary.lisp* - The Lisp implementation of the verb algorithm. Here you will find all the code referenced in Appendix A.
- *transitivity.demo* - The SNePS demo using the verb *throw* and a series of different predicate structures. Note how it invokes the verb algorithm and how it references external files.
- *rels* - The set of relations implemented for the demo file. Notice that all the names of relations are capitalized, a requirement of SNePS 2.6.
- *paths* - The set of path based relations. This may not be entirely needed.

To run the verb algorithm open a SNePS session and demo *transitivity.demo*. The verb algorithm is invoked at the end of this demo. Future developers should review this demo to view how the verb algorithm is invoked and how to set the trace level.