A Preferential Constraint Satisfaction Technique for Natural Language Analysis*

Katashi Nagao
Sony Computer Science Laboratory Inc.
3–14–13 Higashi-gotanda, Shinagawa–ku, Tokyo 141, Japan
E-mail: nagao@csl.sony.co.jp

Abstract. In this paper, we present a new technique for the semantic analysis of sentences, including an ambiguity-packing method that generates a packed representation of individual syntactic and semantic structures. This representation is based on a dependency structure with constraints that must be satisfied in the syntax-semantics mapping phase. Complete syntax-semantics mapping is not performed until all ambiguities have been resolved, thus avoiding the combinatorial explosions that sometimes occur when unpacking locally packed ambiguities. A *constraint satisfaction* technique makes it possible to resolve ambiguities efficiently without unpacking. Disambiguation is the process of applying syntactic and semantic constraints to the possible candidate solutions (such as modifiees, cases, and word-senses) and removing unsatisfactory candidates. Since several candidates often remain after applying constraints, another kind of knowledge to enable selection of the most plausible candidate solution is required. We call this new knowledge a *preference*. Both constraints and preferences must be applied to coordination for disambiguation. Either of them alone is insufficient for the purpose, and the interactions between them are important. We also present an algorithm for controlling the interaction between the constraints and the preferences in the disambiguation process. By allowing the preferences to control the application of the constraints, ambiguities can be efficiently resolved, thus avoiding combinatorial explosions.

1 Introduction

Ambiguities in sentences still present major problems in the development of natural language systems. In this paper, we present a new technique for the semantic analysis of sentences, including an ambiguity-packing method that generates a packed representation of individual syntactic and semantic structures. This packed representation is based on a dependency structure with constraints that must be satisfied in the syntax-semantics mapping phase. Since the ambiguities in sentences sometimes cause a combinatorial explosion, complete syntax-semantics mapping should not be performed until all ambiguities have been resolved without explicitly generating semantic structures. We also provide a preferential constraint satisfaction technique that is used for disambiguation and semantic analysis. Preferences are acquired by using both the taxonomies of a conceptual lexicon, which are called *natural language classes*, and real-world examples of dependency structures.

We previously proposed a method for structural disambiguation by using examples of word-to-word dependencies extracted from on-line dictionary definitions and text corpora [10, 11]. In this framework, each candidate solution of an ambiguity (word-to-word dependency) is assigned a numerical value indicating its preference and the ambiguity is resolved by selecting the most plausible candidate. Preferences for word-to-word dependencies are naturally extended to preferences for relationships between word-senses. However, the most plausible candidate solutions may violate semantic constraints on sentence constituents (such as slot-filler relationships). A *constraint satisfaction* technique makes it possible to efficiently resolve ambiguities.

A framework has been developed that regards syntactic analysis as being a problem of constraint satisfaction [8]. In this framework, grammar rules are represented by constraints on sentence constituents. Semantic constraints on word-senses are so-called *selectional restrictions* on consistency between case slots and their fillers. Disambiguation is the process of applying constraints to the possible candidate solutions (such as modifiees, cases, and word-senses) and removing unsatisfactory candidates. This process is called *constraint propagation* or *filtering*. Unfortunately, several candidates often remain after constraint propa-

*A shorter version of this paper was presented at *the 10th European Conference on Artificial Intelligence (ECAI-92)*, August 3–7, 1992.

gation. Constraints and preferences must be applied together for disambiguation, because either alone is insufficient for the purpose, and the interaction between them is important. We developed an algorithm that allows the preferences to control the application of the constraints.

Our approach is influenced by the *preference semantics* proposed by Wilks [17]. While Wilks' method uses only hand-coded knowledge for disambiguation and does not consider packing ambiguities as a means of avoiding combinatorial explosions, our method calculates the preferences by using knowledge (i.e., natural language classes) and experience (i.e., examples of dependency structures) and packs ambiguities with *delayed semantic composition*, a technique that is characterized by not doing a complete syntax-semantics mapping without first having resolved all ambiguities. Some syntactic/semantic constraints need not be satisfied in order to pack ambiguities. A constraint satisfaction technique is used for efficient disambiguation. Waltz and Pollack [16] proposed a microfeature-based approach. In a network, nodes are joined by microfeatures, which have to be prepared in advance. Instead of microfeatures, our system uses dependency structures as experience, that can be dynamically updated, and NL classes as knowledge. Our experience is acquired semi-automatically from text corpora, and is easy to maintain. Charniak [2] developed a mechanism called *marker passing*, and Hirst [6] developed disambiguation mechanisms called *poraloid words* and the *semantic enquiry desk*. Our methods, called DS (dependency structure) retrieval and class abstraction, are similar to marker passing. Only modifier-modifiee relationships included in DSs are found. In contrast, marker passing has a high computational cost and searches for too many relations.

2 Ambiguity-Packing Based on Delayed Semantic Composition

2.1 Dependency Structures

A *Dependency Structure* (DS, hereafter) is an ambiguity-packed syntactic representation of sentence structures. It is similar to a *syntactic graph* [14]. It explicitly represents modifier-modifiee relationships between words.

A DS is a tree-like structure that consists of *nodes* and *arcs*. A *node* includes some syntactic features (e.g., `cat`, `subcat`, and `person`). *Arcs* correspond to syntactic relationships between sentence constituents (e.g., `SUBJECT`, `DOBJECT`, and `PPADJUNCT`).
For example, Figure 1 shows the dependency structure for the sentence "Insert a diskette into the drive."

```
PRED ``insert'' ((cat v) (subcat trans))
     MOOD    imperative
     TENSE   present
     DOBJECT ``diskette'' ((cat n) (person 3))
             DET  ``a''
             NUM  sg
     PPADJUNCT ``drive'' ((cat n) (person 3) (prep into))
               DET  ``the''
               NUM  sg
```

Figure 1: Dependency Structure for Sentence "Insert a diskette into the drive."

A graphical representation of the above DS is shown in Figure 2 (the nodes corresponding to `MOOD`, `TENSE`, `DET`, and `NUM` are omitted for simplicity). We use such a representation of DSs in the following sections.

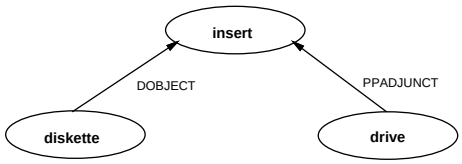

Figure 2: Graphical Representation of Dependency Structure

2.2 Natural Language Classes

We employ an object-oriented conceptual representation, called the *Natural Language* (NL, hereafter) *class system* [15] to express the meaning of a sentence. *NL objects*, which are particular instances of *NL classes*, define a set of conceptual word-senses. The set of NL classes consists of two special classes, `*TOP` and `*BOTTOM`, and two disjoint subsets of classes, *open* classes and *closed* classes. We have *is-a* relationships defined over NL classes. For any NL class `*x`, "`*x` is-a `*TOP`" and "`*BOTTOM` is-a `*x`" hold. Open classes correspond to entities which are expressed by nouns, verbs, adjectives, etc. Closed classes are used to represent attribute values of open classes. Typically, closed classes represent information conveyed by auxiliary verbs (aspects, tense, modals, etc.), determiners, inflections of verbs and nouns (number, gender, definiteness, etc.), and prepositions (relationships between two objects). The closed classes are stable, and are common to almost all domains and natural languages. The open classes are comprehensive, but are dependent on both the domain and language. In this paper, a class is defined in terms of a *frame* format, as shown below. By convention, an open class name is preceded by an asterisk, a closed class name is preceded by an asterisk and a hyphen, an instance has a hyphen and

a number following its class name, and user-defined slot (attribute) names are preceded by a colon. The system-provided slot names and facet names are not preceded by any special characters (comments, however, are preceded by semi-colons).

```
(defclass *insert
;; definition of a class *insert
   (is-a (value *action))
   ;; superclass is *action
   (:agent (sem *human *system))
   ;; :agent must be *human or *system
   (:theme (sem *physical-object))
   ;; :theme must be *physical-object
   (:goal (sem *location *physical-object))
   ;; :goal must be *location or *physical-object
)
```

In the above example, the *value* facet of the `is-a` slot shows a filler (actual attribute value) of the slot. The *sem* facet of other slots shows *selectional restrictions* on their potential fillers. Actual fillers for these slots, except for the `is-a` slot, are not given in the class definition. The `is-a` slot is the only system-defined slot for open classes. All other slots are user-defined (i.e., domain-dependent). The `is-a` slot defines generalization relationships among NL classes, which roughly correspond to the taxonomy of words. A class can inherit each slot definition from its superclasses, unless the slot is redefined. Figure 3 shows a sample hierarchy of NL classes. Our actual NL hierarchy consists of several thousand classes [15, 4] and the Longman Dictionary of Contemporary English (LDOCE, hereafter) [13] word-senses [5, 12, 3].

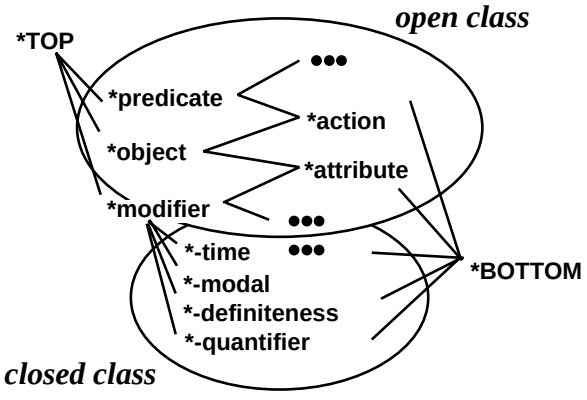

Figure 3: Sample NL Class Hierarchy

2.3 Syntax-Semantics Mapping for Natural Language Objects

Because NL classes constitute the implementation of lexical word-senses, we can define the composition of NL objects to represent the meanings of phrases and sentences. *Mapping rules* are thus introduced to specify the structural composition of NL objects from a given DS. A mapping rule is associated with each NL class, or an NL class and a surface word, as follows:

```
(map *insert <=l=> ``insert'' ((cat v) (subcat trans))
;; a class *insert is associated with a transitive
;; verb ``insert''
 (:agent = (SUBJECT))
 ;; :agent filler corresponds to a SUBJECT in the DS
 (:theme = (DOBJECT))
 ;; :theme filler corresponds to a DOBJECT in the DS
 (:goal = (PPADJUNCT ((prep into))))
 ;; :goal filler corresponds to a PPADJUNCT with a
 ;; preposition ``into'' in the DS
)
(map *physical-action <=s=>
;; *physical-action has no association with a specific
;; word
 (:mood = (MOOD))
 ;; :mood filler corresponds to a MOOD in the DS
 (:time = (TENSE))
 ;; :time filler corresponds to a TENSE in the DS
)
```

The first rule states *lexical mapping* between a syntactic word, or a node in DS, and an NL object. The bodies of the first and second rules define *structural mapping* (i.e., the mapping between a subtree in the DS and a semantic slot filler). The first rule defines that an instance of `*insert` is created to represent the meaning of the transitive verb "insert." Each of its semantic slot fillers is associated with a specific syntactic filler of the verb. The ⇐l⇒ operator indicates that the rule defines lexical mapping, including structural mapping, and the ⇐s⇒ operator indicates that the rule defines structural mapping that has no association with a specific word. Structural mapping in the first type of rule is applicable only to a specific NL object which is mapped from a syntactic word, while the structural mapping for an NL class defined in the second type of rule is inherited by any of its subclasses.

2.4 Delayed Composition of Natural Language Objects

During the sentence analysis process, the DS is the primary structure to be built. Mapping from DS to NL objects is not applied immediately. Rather, it is stored within DS nodes as semantic constraints on DS,

which makes it possible to pack lexical and structural ambiguities into one (or two) DS while keeping track of all possible semantic interpretations. For example, the sentence "VM/SP keeps the information on the virtual disk." can be analyzed as a single DS with two structural ambiguities, as well as many lexical ambiguities caused by "keeps," as shown in Figure 4.

Since the attachment of `PPADJUNCT` to either `DOBJECT` or `PRED` is ambiguous, its mapping is not shown in Figure 4 (the "?" mark in the figure represents an alternative modifiee of the phrase). If `PPADJUNCT` is attached to `DOBJECT`, `*information-1` will have a `(:location *virtual-disk-1)` slot. Otherwise, `PRED` (thus its five NL objects, `*execute-1` through `*own-1`) will get the slot. The mapping from `PPADJUNCT` to `:location` slot should be available from the structural mapping rules. All other mappings are shown in Figure 4. For example, `*execute-1` will have its `:mood` filler from the `MOOD` node of the DS (i.e., `*-declarative-1`). Note that we cannot avoid a combinatorial explosion if we try to materialize all NL-expressions along with the DS. We get ten distinct NL-expressions in the above example, and the DS can no longer be packed. The lexical ambiguity of `PRED` has yet to be resolved.

The disambiguation technique, described later in this paper, will determine the most probable interpretation. Once the DS becomes unambiguous, the structural mapping is evaluated to obtain a full NL-expression that represents the meaning of the sentence. This evaluation process is fairly straightforward.

3 Acquisition of Preferences Using Knowledge and Experience

We use two types of information in the disambiguation of sentence structures. One type is constraints, described later in this paper. The other is preferences, described in this section. The difference between these types can be characterized briefly as follows: Our constraints are symbolic or combinatorial, and restrict sentence structures, while our preferences are numerical, and compare the candidates of structures. We acquire the preferences by using NL classes and examples of DSs.

3.1 Examples of Dependency Structures as Experience

We constructed a set of *Dependency Structures* (DSs) from on-line dictionaries (LDOCE and the IBM Dictionary of Computing) and text corpora [10, 11]. We view the dependency structures as experience, whereas the NL classes are regarded as knowledge.

The DSs as experience are completely disambiguated. One node in each is mapped to one NL object. They are semi-automatically constructed by using large on-line dictionaries and corpora as source texts. The sentences in a source text, which can be ambiguous at first, are disambiguated by using a relatively small number of DSs and NL classes. The disambiguated DSs are also regarded as experience. Therefore, experience increases incrementally.

3.2 Acquisition of Preferences

By applying constraints, we can restrict the candidates of ambiguities. However, ambiguities usually remain in the sentences afterwards. Therefore, we try to assign a preference value to each candidate to select the most preferable semantic interpretation.

When there are candidate structures, DSs that are similar to each of the candidate structures are searched for. The NL classes are used to abstract those classes that are mapped from the nodes in the DSs, and the preferences are calculated using heuristics. We use the concept of *semantic distance* to calculate the value of each preference (defined in Section 3.2.3). The following example sentence illustrates these processes:

> (S1) VM/SP keeps the information on the virtual disk.

The DS of (S1) is shown in Figure 5.

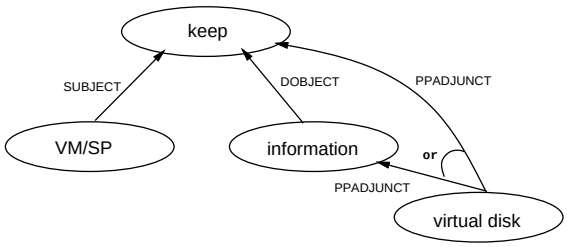

Figure 5: Dependency Structure of (S1)

Sentence (S1) has the following ambiguities:

- Structural ambiguity: the attachment of "on the virtual disk" is ambiguous.
- Lexical ambiguity: the meaning of "keep" is ambiguous (We assume that the other words contain no ambiguity).

```
PRED  ``keep'' ((cat v) (subcat trans)
      (sem (*execute-1 (:mood MOOD) (:time TENSE) (:agent SUBJECT) (:theme DOBJECT))
           (*guard-1   (:mood MOOD) (:time TENSE) (:agent SUBJECT) (:theme DOBJECT))
           (*hold-1    (:mood MOOD) (:time TENSE) (:agent SUBJECT) (:theme DOBJECT))
           (*employ-1  (:mood MOOD) (:time TENSE) (:agent SUBJECT) (:theme DOBJECT))
           (*own-1     (:mood MOOD) (:time TENSE) (:agent SUBJECT) (:theme DOBJECT))))
     MOOD    declarative ((sem *-declarative-1))
     TENSE   present ((sem *-present-1))
     SUBJECT ``VM/SP'' ((cat n) (person 3) (sem (*vm/sp-1 (:number NUM))))
             NUM  sg ((sem *-singular-1))
     DOBJECT ``information'' ((cat n) (person 3)
             (sem (*information-1 (:definiteness DET) (:number NUM))))
             DET  ``the'' ((sem *-definite-1))
             NUM  sg ((sem *-singular-2))
     ?       PPADJUNCT ``virtual disk'' ((cat n) (person 3) (prep on)
                       (sem (*virtual-disk-1 (:definiteness DET) (:number NUM))))
                       DET  ``the'' ((sem *-definite-2))
                       NUM  sg ((sem *-singular-3))
```

Figure 4: Ambiguous Dependency Structure with Partial Semantic Representation

```
;; Sense 1
(defclass *execute         (map *execute <=l=> ``keep'' ((cat v) (subcat trans)))
   (is-a (value *action))
   (:agent (sem *human *abstract))
   (:theme (sem *abstract)))
;; Sense 2
(defclass *guard           (map *guard <=l=> ``keep'' ((cat v) (subcat trans)))
   (is-a (value *action))
   (:agent (sem *human))
   (:theme (sem *human)))
;; Sense 3
(defclass *hold            (map *hold <=l=> ``keep'' ((cat v) (subcat trans)))
   (is-a (value *action))
   (:agent (sem *human *abstract))
   (:theme (sem *abstract *object)))
;; Sense 4
(defclass *employ          (map *employ <=l=> ``keep'' ((cat v) (subcat trans)))
   (is-a (value *action))
   (:agent (sem *human *abstract))
   (:theme (sem *movable-object)))
;; Sense 5
(defclass *own             (map *own <=l=> ``keep'' ((cat v) (subcat trans)))
   (is-a (value *action))
   (:agent (sem *human))
   (:theme (sem *abstract)))
```

Figure 6: Word Sense Definitions and Mapping Rules for "keep"

The word "keep" has five meanings, each of which can be mapped to one NL object. The mapping rules and class definitions are as shown in Figure 6[1]

The preference acquisition process consists of three parts: (1) DS retrieval, (2) class abstraction, and (3) preference calculation. Below, we explain these processes using the example sentence (S1).

3.2.1 DS Retrieval

In the DS retrieval process, DSs that include the words in each candidate dependency are retrieved. For (S1), there are two candidate interpretations of the structural ambiguity:

> (C1) "Virtual disk" modifies "keep."
> (Semantically, `*virtual-disk-1` may fill the `:location` slot of `*execute-1`, `*guard-1`, `*hold-1`, `*employ-1`, or `*own-1`.)
> (C2) "Virtual disk" modifies "information."
> (Semantically, `*virtual-disk-1` may fill the `:location` slot of `*information-1`.)

First, the system searches for DSs that include the word "keep," as well as words that can be mapped to one of the NL objects mapped from "keep" (i.e., synonyms). DSs that include "virtual disk" and its synonyms are also retrieved. Finally, DSs that are included in both sets of DSs are selected. The same procedure is followed for "information" and "virtual disk."

If there are DSs for only one candidate dependency, the candidate is preferable. If there are DSs for more than one candidate, we invoke the preference calculation process. If there are no DSs, then the class abstraction process is invoked. In this case, we cannot find any DSs that include these words, therefore we proceed to the class abstraction process. These processes continue until the abstracted class arrives at the `*TOP` or until the preferences are assigned to all candidates of each ambiguity.

3.2.2 Class Abstraction

Briefly, the class abstraction method is as follows: First, the class corresponding to the modifier is abstracted. (In our example the class of `*virtual-disk` is abstracted.) The immediate superclass is `*disk` and the retrieval process is re-executed. If no DSs can be found by class abstraction of the modifier, then the classes of modifiees `*keep` and `*information` are abstracted. In this case, we found two DSs for each candidate, as shown in Figure 7.

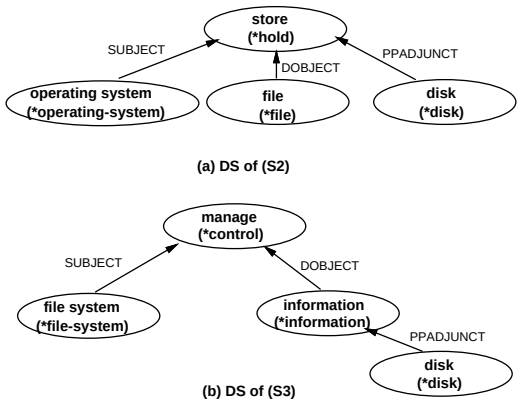

Figure 7: DSs of (S2) and (S3)

The sentences that correspond to the DSs are as follows:

> For (C1): (S2) The operating system stores the files in the disk.
> For (C2): (S3) The file system manages the information on the disk.

3.2.3 Preference Calculation

The preference calculation process calculates the semantic distance between the other words in the input sentence and the nodes in the DS whose rules are consistent with those words. We call each word pair a matching word pair and present it as a bracketed pair.

For example, ["VM/SP","operating system"] and ["file","information"] are matching word pairs for (C1). ["VM/SP","file system"] and ["keep","manage "] are matching word pairs for (C2). For each matching word pair, a distance value is assigned as follows:

- If the matching words are identical or synonymous, the value is 0.
- If the matching words have an is-a relationship, the value is the distance between the NL classes to which the words in the pair correspond.
- If there is no relationship, the value is the height of the NL class hierarchy.

The total preference value $Pref$ for a candidate C is calculated using the following formula:

[1] We extracted the classes mainly from the Longman Dictionary of Contemporary English (LDOCE) [13], restricting their meanings and definitions for simplicity.

(We assume that the candidate has n matching word pairs, $MWP_1 \ldots MWP_n$.)

$$Pref(C) = \frac{1}{\frac{\sum_{i=1}^{n} SemDist(MWP_i(C))}{n} + ClassAbs(C)}$$

$SemDist(MWP_i(C))$: semantic distance value of MWP_i for C
$ClassAbs(C)$: the number of the class abstraction

"VM/SP" is an immediate subclass of "operating system," so the semantic distance value is 1. For other matching words, if there is no relationship and the height of the NL class hierarchy is ten, the preference values for (C1) and (C2) are as follows:
For (C1): $\frac{1}{\frac{1+10}{2}+1} = \frac{2}{13}$
For (C2): $\frac{1}{\frac{10+10}{2}+1} = \frac{1}{11}$

The candidate having the highest preference value is the most preferable. In this case, the candidate (C1), where "virtual disk" modifies "keep," is preferable to (C2), "virtual disk" modifies "information."

The preferences for candidates of ambiguities are calculated in this way.

4 Disambiguation by a Preferential Constraint Satisfaction Technique

4.1 Disambiguation as Constraint Satisfaction

Constraints for disambiguation include syntactic and semantic constraints. Syntactic constraints can be considered as being declarative representations of syntactic rules. The *Constraint Dependency Grammar* (CDG, hereafter) [8] formalizes parsing as a constraint satisfaction problem. Our syntactic constraints are equivalent to CDG's syntactic rules. Semantic constraints are formed by the consistency between the semantic categories of case-slots and their fillers, which are called *selectional restrictions*.

From these constraints, a constraint network [9] is generated. Its nodes correspond to ambiguities that are represented as sets of candidates of slot-filler relationships. Its arcs correspond to constraint matrices that represent constraints on combinations of candidates. When the constraint network is generated, an algorithm for constraint propagation [7] is applied to resolve any ambiguities. We developed a modified algorithm for constraint propagation that is controlled by preference orderings.

4.2 Syntactic Constraints

We regard constraints between ambiguous modification relations (dependencies) as being syntactic constraints. They include the no-crossing constraint, which means that modification links cannot cross each other, and the no-duplicate-case constraint, which means that modification relations (i.e., case-slots) whose modifiees (i.e., frames with the slot) are the same cannot play a role of the same relation. Dependencies are considered as slot-filler relationships that are represented by instances of the form (`*frame (:slot *frame-of-filler)`).

Ambiguities are non-singleton sets of dependencies whose modifiers (i.e., fillers) are the same. If a candidate dependency of an ambiguity and a candidate of another ambiguity cross each other, then the combination of these candidates violates the no-crossing constraint. If an element of an ambiguity has the same modifiee (frame) and case relation (slot) as an element of another ambiguity, then their combination violates the no-duplicate-case constraint.

For example, in the sentence (S4) "Put the block on the table on the floor in the room.", the prepositional phrases "on the table" (PP1), "on the floor" (PP2), and "in the room" (PP3) have attachment ambiguities, and there are the following syntactic constraints between these ambiguities.

1. No-crossing
   If PP1 is attached to the verb "put" (V), then PP2 cannot be attached to the noun phrase "the block" (NP). If PP2 is attached to V, then PP3 cannot be attached to either NP or PP1.

2. No-duplicate-case (in this case, the case relations correspond to the `:goal` and `:location` slots)
   If PP1 is attached to V, then PP2 and PP3 cannot be attached to V. If PP1 is attached to NP, then PP2 and PP3 cannot be attached to NP.

In accordance with these constraints, the constraint matrices are constructed.[2]

Table 1 shows the constraint matrices constructed from (S4), and the sets A_1, A_2, and A_3 in the matrices are defined as follows.

A_1={a=(`*put-1 (:goal *table-1)`),
  b=(`*block-1 (:location *table-1)`)}
A_2={c=(`*put-1 (:goal *floor-1)`),
  d=(`*block-1 (:location *floor-1)`),
  e=(`*table-1 (:location *floor-1)`)}
A_3={f=(`*put-1 (:goal *room-1)`),
  g=(`*block-1 (:location *room-1)`),
  h=(`*table-1 (:location *room-1)`),
  i=(`*floor-1 (:location *room-1)`)}

[2] For the sake of simplicity, we don't consider word-sense ambiguities here.

Table 1: Constraint Matrices for (S4)

$A_1 \backslash A_2$	c	d	e
a	0	0	1
b	1	0	1

$A_1 \backslash A_3$	f	g	h	i
a	0	0	1	1
b	1	0	1	1

$A_2 \backslash A_3$	f	g	h	i
c	0	0	0	1
d	1	0	0	1
e	1	1	0	1

The values represent whether the combination satisfies the constraint. When the value is 0, the combination of elements in that row and column does not satisfy the constraint. If all the values in a column or row are 0, then the candidate in that column or row is removed from the ambiguity. Since all values in the column of d are 0 in matrix $A_1 \backslash A_2$, the candidate d of A_2 should be removed. Therefore, the values in row d in matrix $A_2 \backslash A_3$ all become 0. The values in column g of matrix $A_2 \backslash A_3$ and those of h in matrix $A_1 \backslash A_3$ also become 0 for the same reason.

Therefore, the matrices on A_3 are updated by constraint propagation. Table 2 shows the revised constraint matrices.

4.3 Semantic Constraints

We also consider semantic constraints between slot-filler relationships called *selectional restrictions*. They are taxonomic constraints on which structures can serve as part of another structure, and are used for word-sense disambiguation. Word-sense ambiguities are also represented as non-singleton sets of slot-filler relationships.

For example, in the sentence (S1) "VM/SP keeps the information on the virtual disk." in the previous section, there is a word-sense ambiguity for "keep" and a structural ambiguity on the modification of "on the virtual disk," as described in the previous section.

We represent these ambiguities as follows:

A_1={a=(*execute-1 (:agent *vm/sp-1)),
b=(*guard-1 (:agent *vm/sp-1)),
c=(*hold-1 (:agent *vm/sp-1)),
d=(*employ-1 (:agent *vm/sp-1)),
e=(*own-1 (:agent *vm/sp-1))}
A_2={f=(*execute-1 (:theme *information-1)),
g=(*guard-1 (:theme *information-1)),
h=(*hold-1 (:theme *information-1)),
i=(*employ-1 (:theme *information-1)),
j=(*own-1 (:theme *information-1))}
A_3={k=(*execute-1 (:location *virtual-disk-1)),
l=(*guard-1 (:location *virtual-disk-1)),
m=(*hold-1 (:location *virtual-disk-1)),
n=(*employ-1 (:location *virtual-disk-1)),
o=(*own-1 (:location *virtual-disk-1)),
p=(*information-1 (:location *virtual-disk-1))}

From the definitions of the senses of "keep" in the previous section, and from the class relations of *vm/sp (i.e., *vm/sp is a subclass of *abstract) and of *information (i.e., *information is also a subclass of *abstract), we find that *vm/sp-1 does not fill in the :agent slot of *guard-1 and *own-1, since their class definitions include the description of (:agent (sem *human)). Similarly, we can also find that *information-1 does not fill in the :theme slot of *guard-1 and *employ-1, since each of their class definitions includes the description of (:theme (sem *human)) or (:theme (sem *movable-object)). Furthermore, we also consider the trivial constraint whereby if the :agent, :theme, and :location slots are filled, then the frames including these slots will be identical, since the frames correspond to the meaning of the single word "keep" in the sentence (S1).

Considering the semantic constraints, the constraint matrices are constructed as shown in Table 3.

The matrices on A_3 are updated by constraint propagation as shown in Table 4.

4.4 Algorithm of Disambiguation

A brief overview of the disambiguation algorithm is as follows:

1. If all combinations of the most preferable candidates satisfy the constraints, then choose them and terminate. Otherwise, go to the next step.
2. Remove the least preferable candidate. Then perform constraint propagation.
3. If all candidate solutions of an ambiguity have been removed, undo the last propagation (restore the states that were modified in step 2).
4. If all ambiguities are singletons, then terminate.
5. Repeat steps 2 and 3 for the next least preferable candidate.

A detailed algorithm of constraint propagation under the control of preference orderings is as follows: Let A_i and A_j be ambiguities, a and b be each element

Table 2: Revised Constraint Matrices for (S4)

$A_1 \backslash A_3$	f	g	h	i
a	0	0	0	1
b	1	0	0	1

$A_2 \backslash A_3$	f	g	h	i
c	0	0	0	1
d	0	0	0	0
e	1	0	0	1

Table 3: Constraint Matrices for (S1)

$A_1 \backslash A_2$	f	g	h	i	j
a	1	0	0	0	0
b	0	0	0	0	0
c	0	0	1	0	0
d	0	0	0	0	0
e	0	0	0	0	0

$A_1 \backslash A_3$	k	l	m	n	o	p
a	1	0	0	0	0	1
b	0	0	0	0	0	0
c	0	0	1	0	0	1
d	0	0	0	1	0	1
e	0	0	0	0	0	0

$A_2 \backslash A_3$	k	l	m	n	o	p
f	1	0	0	0	0	1
g	0	0	0	0	0	0
h	0	0	1	0	0	1
i	0	0	0	0	0	0
j	0	0	0	0	1	1

respectively, $M(i,j)$ be a constraint matrix on A_i and A_j, and $M(i,a,j,b)$ be the value of the matrix when A_i takes a and A_j takes b. In addition, let $p(i,a)$ be a's preference value in ambiguity A_i. First, for all pairs of an ambiguity and its element (i,a), construct an ordering set PRF that is arranged (i,a) in ascending order of preference. Thus, for each element (i,a) and (j,b) of PRF, if (i,a) is ahead of (j,b), then $p(i,a)$ is equal to or less than $p(j,b)$.
The algorithm consists of the following steps:

1. Remove the front-most element (i,a) from PRF.

2. Using the following algorithm, for all ambiguities and their elements, construct their supported sets and the set of inactive elements IN.

   First, for each ambiguity A_i, its element a, and a constraint matrix $M(i,j)$, construct a supported set of (i,a), $S(i,j,a) = \{(j,b) | M(i,a,j,b) = 1$ (in this case, we say that (j,b) supports $(i,a))\}$. If $S(i,j,a) = \{\}$ then put (i,a) in IN, a set of inactive elements. Also, for each A_j and its element b, construct a supported set of (j,b), $S(j,i,b)$. If $S(j,i,b) = \{\}$, then put (j,b) in IN.

   Set the set CHG (the set containing the changed elements) to $\{\}$.

3. For each of the ambiguities A_j and A_k and their elements b and c, respectively, if $M(i,a,j,b) = 1$ or $M(k,c,i,a) = 1$, then set their values to 0 and put (i,a,j,b) or (k,c,i,a) in CHG.

4. Iterate the following sub-steps until IN becomes empty:

   (a) Remove element (j,b) from IN.

   (b) Set A_j to $A_j - \{b\}$.

   (c) If $A_j = \{\}$ then set A_j to $\{b\}$, for all elements (k,c,l,d) of CHG, set $M(k,c,l,d)$ to 1, then go to step 1.

   (d) For all elements (k,c) of $S(j,b)$, set $M(j,b,k,c)$ to 0 and $S(k,j,c)$ to $S(k,j,c) - \{(j,b)\}$, then put (j,b,k,c) in CHG.

   (e) If $S(k,j,c) = \{\}$ and (k,c) is not an element of IN, then put (k,c) in IN.

5. If all ambiguities are singletons, terminate. Otherwise, go to step 1.

The problem that this algorithm can solve is called the *consistent labeling problem (CLP)* [9]. CLP is a problem that determines the existence of an assignment that satisfies all the constraints, given a set of variables each of which can take any one of a set of values and constraints between these values. An example of CLP is a graph-coloring problem to assign colors to a graph with all adjacent vertexes in different colors. A CLP is *satisfiable* if there is an assignment that satisfies all the constraints simultaneously. Deciding the satisfiability of a CLP is NP complete in general. However, there are cases in which unsatisfactory values (i.e., values that are not included in any solution) can be identified simply by constraint propagation that checks local inconsistencies. Such an algorithm has been shown to have polynomial complexity [7].

It is possible to achieve more global consistency by looking at multiple constraint matrices simultaneously, but as Carter [1] argues regarding the experimental Propane parser, once local (pair-wise) consistencies have been achieved, performing a backtrack search is usually more efficient than using higher-level consistency algorithms. Our algorithm combines

Table 4: Revised Constraint Matrices for (S1)

$A_1 \backslash A_3$	k	l	m	n	o	p
a	1	0	0	0	0	1
b	0	0	0	0	0	0
c	0	0	1	0	0	1
d	0	0	0	0	0	0
e	0	0	0	0	0	0

$A_2 \backslash A_3$	k	l	m	n	o	p
f	1	0	0	0	0	1
g	0	0	0	0	0	0
h	0	0	1	0	0	1
i	0	0	0	0	0	0
j	0	0	0	0	0	0

constraint propagation with the preferential control rather than to perform a backtrack search. We extend a constraint propagation algorithm in two senses. First, we add a control mechanism based on preferential orderings over candidate solutions. Second, we add an inconsistency checking and undoing process when local inconsistencies occur after constraint propagation. These mechanisms are invoked when the given constraints are not tight enough to narrow down the number of candidate solutions to one. If a CLP can have a solution, then our algorithm can find a solution, since the algorithm maintains local consistency (locally consistent value sets must contain a solution) and narrows down the number of candidates. However, since our preference compares candidates of each value set (i.e., ambiguity) independently, our algorithm may not find the optimum solution that is prior to all other solutions.

5 Experimental Results

In this section, we describe some experiments in disambiguation and their results. First, we extracted 529 sample sentences including the verb "get" and it's synonyms from LDOCE. The word "get" has twenty senses, making it very difficult to select a particular sense simply by selectional restrictions.

We also prepared another 135 test sentences including "get" from LDOCE. We randomly divided the test sentences into three groups (which we call `group1`, `group2`, and `group3`). This division is done to see the effect of increasing experience. The following experiments were carried out on these groups.

Experiment1
Sentences in `group1` were disambiguated by experiments with a set of 529 disambiguated dependency structures.

Experiment2
After Experiment1, we checked and modified the output dependency structures of `group1` and added them to the original DS set. `group2` was then disambiguated with the modified DS set, to which DSs of `group1` were added.

Experiment3
After Experiment2, we checked and modified the output dependency structures of `group2` and added them to the DS set. `group3` was then disambiguated with the modified DS set to which DSs of `group1` and `group2` were added.

Table 5 shows the results of the experiments.

We evaluated the outputs of each of the groups and categorized them into three success levels:

A: There were exactly matched DSs and the system was able to select a correct answer.
B: More than one DS match had the same preference value, but the system was able to select a correct answer by using heuristics.
C: The system was not able to select a correct answer.

For the relatively small cases (about 500 DSs), we attained an average success ratio of 71.5%. This experience showed that most of time for disambiguation is taken by DS retrieval and preference calculation, not by applying constraints and constraint propagation.

6 Concluding Remarks

We have developed a sentence analysis technique containing an ambiguity-packing method, a preference acquisition method, and a constraint-based disambiguation method. Sentences are parsed and transformed into ambiguity-packed dependency structures. To avoid combinatorial explosions, complete syntax-semantics mapping must not be performed until all ambiguities have been resolved without generating individual semantic structures. A constraint satisfaction technique makes it possible to resolve ambiguities efficiently without unpacking. Constraints and preferences must be applied together for disambiguation, because either of them alone is insufficient, and the interaction between them is important. The constraints consist of syntactic and semantic constraints. The preferences are acquired from the taxonomies of a conceptual lexicon (NL classes) and examples of dependency structures. To resolve ambiguities efficiently, we

Table 5: Results of the Experiments

	Total sentences	Parsed sentences (S)	A	B	C	$\frac{A+B}{S}$ (%)
Experiment1	45	41	15	8	18	56.1
Experiment2	45	43	17	19	7	83.7
Experiment3	45	44	22	11	11	75.0

developed a new constraint satisfaction mechanism in which constraints are applied according to preference orderings.

Acknowledgments

This work was partially done while the author was a member of IBM Research, Tokyo Research Laboratory (TRL). The author would like to thank the members of the natural language processing group at TRL, especially Koichi Takeda, Naohiko Uramoto, and Hiroshi Maruyama, for their helpful discussions and comments on an earlier draft of this paper.

References

[1] D. Carter, "Efficient Disjunctive Unification for Bottom-Up Parsing," In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 70–75, 1990.

[2] E. Charniak, "A Neat Theory of Maker Passing," In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86)*, pages 584–588, 1986.

[3] M. S. Chodorow, R. J. Byrd, and G. E. Heidorn, "Extracting Semantic Hierarchies from a Large On-Line Dictionary," In *Proceedings of the 23th Annual Meeting of the ACL*, pages 299–304, 1985.

[4] K. Goodman and S. Nirenburg, editors, *The KBMT Project: A Case Study in Knowledge-Based Machine Translation*, Morgan Kaufmann Publishers, San Mateo, California, 1991.

[5] L. Guthrie, B. M. Slator, Y. Wilks, and R. Bruce, "Is There Content in Empty Heads?," In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 138–143, 1990.

[6] G. Hirst, *Semantic Interpretation and the Resolution of Ambiguity*, Cambridge University Press, 1987.

[7] A. K. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence*, volume 8, pages 99–118, 1977.

[8] H. Maruyama, "Structural Disambiguation with Constraint Propagation," In *Proceedings of the 28th Annual Meeting of the ACL*, pages 31–38, 1990.

[9] U. Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Sciences*, volume 7, pages 95–132, 1974.

[10] K. Nagao, "Dependency Analyzer: A Knowledge-Based Approach to Structural Disambiguation," In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 282–287, 1990.

[11] K. Nagao, "Constraints and Preferences: Integrating Grammatical and Semantic Knowledge for Structural Disambiguation," In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI'90)*, pages 484–489, 1990.

[12] J. Nakamura and M. Nagao, "Extraction of Semantic Information from an Ordinary English Dictionary and its Evaluation," In *Proceedings of the 12th International Conference on Computational Linguistics (COLING-88)*, pages 459–464, 1988.

[13] P. Procter, editor, *Longman Dictionary of Contemporary English*, Longman Group Limited, Harlow and London, England, 1978.

[14] J. Seo and R. F. Simmons, "Syntactic Graphs: A Representation for the Union of All Ambiguous Parse Trees," *Computational Linguistics*, volume 15, pages 19–32, 1989.

[15] K. Takeda, "Designing Natural Language Objects," In *Proceedings of the International Symposium on Database Systems for Advanced Applications*, pages 444–448, 1991.

[16] D. Waltz and J. B. Pollack, "Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation," *Cognitive Science*, volume 9, pages 51–74, 1985.

[17] Y. Wilks, "A Preferential, Pattern-Seeking, Semantics for Natural Language Inference," *Artificial Intelligence*, volume 6, pages 53–74, 1975.