Semantic Caseframe Parsing and Syntactic Generality

Philip J. Hayes, Peggy M. Andersen, and Scott Safier
Carnegie Group Incorporated
Commerce Court at Station Square
Pittsburgh, PA 15219 USA

Abstract

We have implemented a restricted domain parser called Plume.TM Building on previous work at Carnegie-Mellon University e.g. [4, 5, 8], Plume's approach to parsing is based on semantic caseframe instantiation. This has the advantages of efficiency on grammatical input, and robustness in the face of ungrammatical input. While Plume is well adapted to simple declarative and imperative utterances, it handles passives, relative clauses and interrogatives in an ad hoc manner leading to patchy syntactic coverage. This paper outlines Plume as it currently exists and describes our detailed design for extending Plume to handle passives, relative clauses, and interrogatives in a general manner.

1. The Plume Parser

Recent work at Carnegie-Mellon University e.g. [4, 5] has shown semantic caseframe instantiation to be a highly robust and efficient method of parsing restricted domain input. In this approach to parsing, a caseframe grammar contains the domain-specific semantic information, and the parsing program contains general syntactic knowledge. Input is mapped onto the grammar using this built-in syntactic knowledge. We have chosen this approach for PlumeTM, a commercial restricted domain parser[1] because of its advantages in efficiency and robustness.

Let us take a simple example from a natural language interface, called NLVMS, that we are developing under a contract with Digital Equipment Corporation. NLVMS is an interface to Digital's VMSR operating system for VAXR computers.[2] The Plume grammar for this interface contains the following semantic caseframe[3] corresponding to the copy command of VMS:

```
[*copy*
 :cf-type clausal
 :header copy
 :cases
   (file-to-copy
    :filler *file*
    :positional Direct-Object)
   (source
    :filler *directory*
    :marker from | out of)
   (destination
    :filler *file* | *directory*
    :marker to | into | in | onto)
]
```

This defines a caseframe called "copy" with three cases: file-to-copy, source, and destination. The file-to-copy case is filled by an object of type "file" and appears in the input as a direct object. Source is filled by a "directory" and should appear in the input as a prepositional phrase preceded or marked by the prepositions "from" or "out of". Destination is filled by a "file" or "directory" and is marked by "to", "into", or "onto". Finally the copy command itself is recognized by the header word indicated above (by header) as "copy".

Using this caseframe, Plume can parse inputs like:

Copy foo bar out of [x] into [y][4]
From [x] to [y] copy foo bar
foo bar copy from [x] to [y]

[1]More precisely, PlumeTM is the name of the run-time system associated with Language CraftTM, an integrated environment for the development of natural language interfaces. The Plume parser which translates English input into caseframe instances, is a major component of this run-time system. The other major component translates the caseframe instances into application specific languages. In addition to the Plume run-time system, Language Craft includes grammar development tools including a structured editor and tracing and performance measurement tools. Both Plume and Language Craft are products of Carnegie Group, and are currently in restricted release. Plume and Language Craft are trademarks of Carnegie Group Incorporated.

[2]VMS and VAX are trademarks of Digital Equipment Corporation

[3]This is a simplified version of the caseframe actually in the grammar.

In essence, Plume's parsing algorithm is to find a caseframe header, in this case "copy", and use the associated caseframe, *copy*, to guide the rest of the parse. Once the caseframe has been identified, Plume looks for case markers, and then parses the associated case filler directly following the marker. Plume also tries to parse positionally specified cases, like direct object, in the usual position in the sentence - immediately following the header for direct object. Any input not accounted for at the end of this procedure is matched against any unfilled cases, so that cases that are supposed to be marked can be recognized without their markers and positionally indicated cases can be recognized out of their usual positions. This flexible, interpretive style of matching caseframes against the input allows Plume to deal with the kind of variation in word order illustrated in the examples above.

The above examples implied there was some method to recognize files and directories. They showed only atomic file and directory descriptions, but Plume can also deal with more complex object descriptions. In fact, in Plume grammars, objects as well as actions can be described by caseframes. For instance, here is the caseframe[5] used to define a file for NLVMS.

```
[*file*
 :cf-type nominal
 :header file !
         !name ?(%period !extension)
 :cases
   (name
    :assignedp !name)
   (extension
    :assignedp !extension
    :marker written in
    :adjective <language>
    :filler <language>)
   (creator
    :filler *person*
    :marker created by)
   (directory
    :filler *directory*
    :marker in)
]
```

[4]n the syntax used with VMS, directories are indicated by square brackets.

[5]again simplified

[6]Plume automatically recognizes determiners and quantifiers associated with nominal caseframes

This caseframe allows Plume to recognize file descriptions like:[6]

foo
foo.bar
The file created by John
The fortran file in [x] created by John

The caseframe notation and parsing algorithm used here are very similar to those described above for clause level input. The significant differences are additions related to the :adjective and :assignedp attributes of some of the cases above. While Plume normally only looks for fillers *after* the header in nominal caseframes an adjective attribute of a slot tells Plume that the slot filler may appear before the header.

An :assignedp attribute allows cases to be filled through recognition of a header. This is generally useful for proper names, such as foo and foo.bar. In the example above, the second alternative header contains two variables 'name and 'extension, that can each match any single word. The question mark indicates optionality, so that the header can be either a single word or a word followed by a period and another word. The first word is assigned to the variable 'name, and the second (if it is there) to the variable 'extension. If 'name or 'extension are matched while recognizing a file header, their values are placed in the name and extension cases of *file*.

With the above modifications Plume can parse nominal caseframes using the same algorithm that it uses for clausal caseframes that account for complete sentences. However there are some interactions between the two levels of parsing. In particular, there can be ambiguity about where to attach marked cases. For instance, in:

Copy the fortran file in [x] to [y]

"in [x]" could either fill the directory case of the file described as "the fortran file" or could fill the destination case of the whole copy command. The second interpretation does not work at the global level because the only place to put "to [y]" is in that same destination case. However, at the time the file description is parsed, this information is not available, and so both possible attachments must be considered. In general, if Plume is able to fill a case of a nominal caseframe from a prepositional phrase, it also splits off an alternative parse in which that attachment is not made. When all input has been parsed, Plume retains only those parses that succeed at the global level, i.e., consume all of the input. Others are discarded.

The current implementation of Plume is based on the nominal and clausal level caseframe instantiation algorithms described above. Using these algorithms and a restricted domain grammar of caseframes like the ones shown above, Plume can parse a wide variety of imperative and declarative sentences relevant to that domain. However, there remain significant gaps in its coverage. Interrogatives are not handled at all;[7] passives are covered only if they are explicitly specified in the grammar and relative clauses can only be handled by pretending they are a form of prepositional phrase.

The regular and predictable relationship between simple statements, questions and relative clauses and between active and passive sentences is well known. A parser which purports to interpret a domain specific language specification using a built-in knowledge of syntax should account for this regularity in a general way. The current implementation of Plume has no mechanism for doing this. Each individual possibility for questions, relative clauses, and passives must be explicitly specified in the grammar. For instance, to handle reduced relative clauses as in "the file created by Jim", "created by" is listed as a case marker (compound preposition) in the creator slot of file, marking a description of the creator. To handle full relatives the case marker must be specified as something like "?(which <be>) created by".[3] While this allows Plume to recognize "the file which was created by Jim", "the file created by Jim", or even "the file created by Jim on Monday", it breaks down on something like "the file created on Monday by Jim" because the case marker "created by" is no longer a unit. Moreover, using the current techniques, Plume's ability to recognize the above inputs is completely unrelated to its ability to recognize inputs like:

the file Jim created on Monday
the person that the file was created by on Monday
the day on which Jim created the file

If an interface could recognize any of these examples it might seem unreasonable to a user that it could not recognize all of the others. Moreover, given any of the above examples, a user might reasonably expect recognition of related sentence level inputs like

Create the file on Monday!
Jim created the file on Monday
Did Jim create the file on Monday?
Was the file created by Jim on Monday?
Who created the file on Monday?
What day was the file created on?

The current implementation of Plume has no means of guaranteeing such regularity of coverage. Of course, this problem of patchy syntactic coverage is not new for restricted domain parsers. The lack of syntactic generality of the original semantic grammar [3] for the Sophie system [2] led to the concept of cascaded ATNs [10] and the RUS parser [1]. A progression with similar goals occurred from the LIFER system [9] to TEAM [6] and KLAUS [7].

The basic obstacle to achieving syntactic generality in these network-based approaches was the way syntactic and semantic information was mixed together in the grammar networks. The solutions, therefore, rested on separating the syntactic and semantic information. Plume already incorporates just the separation of syntax and semantics necessary for syntactic generality: general syntactic knowledge resides in the parser, while semantic information resides in the grammar. This suggests that syntactic generality in a system like Plume can be achieved by improving the parser's caseframe instantiation algorithms without any major changes to grammar content. In terms of the above examples involving "create", it suggests we can use a single "create" caseframe to handle all the examples. We simply need to provide suitable extensions to the existing caseframe instantiation algorithms. In the next section we present a detailed design for such extensions.

2. Providing Plume with Syntactic Generality

As described above, Plume can currently use clausal

[7]The current implementation of Plume contains a temporary method of handling interrogatives, based on pattern matching [illegible] caseframe instantiation

[3]where <be> is a nonterminal representing [illegible] be

caseframes only to recognize single clause imperative and declarative utterances in the active voice. This section describes our design for extending Plume so that relative and interrogative uses of clausal caseframes in passive as well as active voice can also be recognized from the same information.

We will present our general design by showing how it operates for the following 'create' caseframe in the context of NLVMS:

```
[*create*
 :cf-type clausal
 :header <create>
 :cases
   (creator
    :filler *person*
    :positional Subject)
   (createe
    :filler *file*
    :positional Direct-Object)
   (creation-date
    :filler *date*
    :marker on)
]
```

Note that symbols in angle brackets represent non-terminals in a context-free grammar (recognized by Plume using pattern matching techniques). In the caseframe definition above <create> matches all morphological variants of the verb "create" including "create", "creates", "created" and "creating" (though not compound tenses like "is creating" - see below). Using the existing Plume this would only allow us to recognize simple imperatives and active declaratives like:

Create foo bar on Monday
Jim created foo bar on Monday

2.1 Passives

Plume recognizes passive sentences through its processing of the *verb cluster*, i.e. the main verb plus the sequence of modal and auxiliary verb immediately preceding it. Once the main verb has been located, a special verb cluster processing mechanism reads the verb cluster and determines from it whether the sentence is active or passive[9]. The parser records this information in a special case called "%voice".

If a sentence is found to be active, the standard parsing algorithm described above is used. If it is found to be passive, the standard algorithm is used with the modification that the parser looks for the direct object or the indirect object[10] in the subject position, and for the subject as an optional marked case with the case marker "by". Thus, given the 'create' caseframe above, the following passive sentences could be handled as well as their active counterparts.

Foo bar was created by Jim
Foo bar could have been created by Jim
Foo bar is being created by Jim
Foo bar was created on Monday

2.2. Relative clauses

The detailed design presented below allows Plume to use the 'create' caseframe to parse nominals like:

the file Jim created on Monday
the person that the file was created by on Monday
the day on which Jim created the file

To do this, we introduce the concept of a *relative case*. A relative case is a link back from the caseframes for the objects that fill the cases of a clausal caseframe to that clausal caseframe. A grammar preprocessor generates a relative case automatically from each case of a clausal caseframe, associating it with the nominal caseframe that fills the case in the clausal caseframe. Relative cases do not need to be specified by the grammar writer. For instance, a relative case is generated from the createe case of 'create' and included in the 'file' caseframe. It looks like this:

```
[*file*
    .
    .
    .
 (:relative-cf *create*
  :relative-case-name createe
  :marker <create>
 ]
```

[9]It also determines the tense of the sentence and whether it is affirmative or negative.

[10]So if there is a case with a positional indirect-object slot, the indirect object is allowed to passivize. We can thus understand sentences like "Mary was given a book," from a 'give' caseframe with both a direct object and an indirect object case.

Note that :marker is the same as :header of 'create'. Similar relative cases are generated in the 'person' caseframe for the creator case, and in the 'date' caseframe for the creation-date case, differing only in :relative-case-name.

Relative cases are used similarly to the ordinary marked cases of nominal caseframes. In essence, if the parser is parsing a nominal caseframe and finds the marker of one of its relative cases, then it tries to instantiate the :relative-cf. It performs this instantiation in the same way as if the relative-cf were a top-level clausal caseframe and the word that matched the header were its main verb. An important difference is that it never tries to fill the case whose name is given by relative-case-name. That case is filled by the nominal caseframe which contains the relative case. For instance, suppose the parser is trying to process:

The file Jim created on Monday

And suppose that it has already located "file" and used that to determine it is instantiating a 'file' nominal caseframe. It is able to match (against "created") the :marker of the relative caseframe of 'file' shown above. It then tries to instantiate the relative-cf 'create' using its standard techniques except that it does not try to fill createe, the case of 'create' specified as the relative-case-name. This instantiation succeeds with "Jim" going into creator, and "on Monday" being used to fill creation-date. The parser then uses (a pointer to) the nominal caseframe currently being instantiated, 'file', to fill createe, the :relative-case-name case of 'create', and the newly created instance of 'create' is attached to this instance of 'file' as a modifier.

More completely, Plume's algorithm for relative clauses is:

1. When processing a nominal caseframe, Plume scans for the :markers of the relative cases of the nominal caseframe at the same time as it scans for the regular case markers of that nominal caseframe.

2. If it finds a :marker of a relative case, it tries to instantiate the relative-cf just as though it were the top-level clausal caseframe and the header were its main verb, except that:

   a. it never looks any further left in the input than the header of the nominal caseframe or, if it has already parsed any other post-nominal cases of the nominal caseframe, no further left than the right hand end of them;

   b. it consumes, but otherwise ignores any relative pronouns (*who, whom, which, that*) that immediately precede the segment used to instantiate the relative-cf. *This means that all words, including "that" will be accounted for in "the file that Jim created on Monday"*;

   c. it does not try to fill the case specified by the :relative-case-name in the relative-cf; instead, this case is filled by (a pointer to) the original nominal caseframe instance;

   d. if the relative-case-name specifies a marked case rather than a positional one in the relative-cf, then its case marker can be consumed, but otherwise ignored, during instantiation of the relative-cf. *This allows us to deal with "on" in "the date Jim created the file on" or "the date on which Jim created the file".*

3. Passive relative clauses (e.g. "the file that was created on Monday") can generally be handled using the same mechanisms used for passives at the main clause level. However, in relative clauses, passives may sometimes be *reduced* by omitting the usual auxiliary verb to be (and the relative pronoun) as in:

   the file created on Monday

   To account for such reduced relative clauses, the verb cluster processor will produce appropriate additional readings of the verb clusters in relative clauses for which the relative pronoun is missing. This may lead to multiple parses, including one for the above example similar to the correct one for:

   the file John created on Monday

   These ambiguities will be taken care of by Plume's standard ambiguity reduction methods.

2.3 Interrogatives

In addition to handling passives and relative clauses, we also wish the information in the 'create' caseframe to handle interrogatives involving "create", such as

Did Jim create the file on Monday?
Was the file created by Jim on Monday?
Who created the file on Monday?
What day was the file created on?

The primary difficulty for Plume with interrogatives is that, as these examples show, the number of variations in standard constituent order is much greater than for imperatives and *declaratives.* Interrogatives come in a wide variety of forms, depending on whether the question is yes/no or wh: on which auxiliary verb is used: on whether the voice is active or passive: and for wh questions, on which case is queried. On the other hand, apart from variations in the order and placement of marked cases, there is only one standard constituent order for imperatives and only two for declaratives (corresponding to active and passive voice). We have exploited this low variability by building knowledge of the imperative and declarative order into Plume's parsing algorithm. However this is impractical for the larger number of variations associated with interrogatives. Accordingly, we have designed a more data-driven approach.

This approach involves two passes through the input: the first categorizes the input into one on several primary input categories including yes-no questions, several kinds of wh-questions, statements, or imperatives. The second pass performs a detailed parse of the input based on the classification made in the first pass. The rules used contain basic syntactic information about English, and will remain constant for any of Plume's restricted domain grammars of semantic caseframes for English

The first level of processing involves an ordered set of *top-level patterns.* Each top-level pattern corresponds to one of the primary input categories mentioned above. This classificatory matching does not attempt to match every word in the input sentence, but only to do the minimum necessary to make the classification. Most of the relevant information is found at the beginning of the inputs. In particular, the top-level patterns make use of the fronted auxiliary verb and wh-words in questions.

As well as classifying the input, this top-level match is also used to determine the identity of the caseframe to be instantiated. This is important to do at this stage because the detailed recognition in the second phase is heavily dependent on the identity of this top-level caseframe. The special symbol, *$verb*, that appears exactly once in all top-level patterns, matches a header of any clausal caseframe. We call the caseframe whose header is matched by $verb the *primary caseframe* for that input.

The second more detailed parsing phase is organized relative to the primary caseframe. Associated with each top-level pattern, there is a corresponding *parse template.* A parse template specifies which parts of the primary caseframe will be found in unusual positions and which parts the default parsing process (the one for declaratives and imperatives) can be used for.

A simplified example of a top-level pattern for a yes-no question is:[11]

```
<aux> (~ ($verb !! <aux>)) (&s $verb) $rest
```

This top-level pattern will match inputs like the following:

Did Jim create foo?
Was foo created by Jim?

The first element of the above top-level pattern is an auxiliary verb, represented by the non-terminal <aux>. This auxiliary is remembered and used by the verb cluster processor (as though it were the first auxiliary in the cluster) to determine tense and voice. According to the next part of the pattern, some word that is not a verb or an auxiliary must appear after the fronted auxiliary and before the main verb (~ is the negation operator, and !! marks a disjunction). Next, the scanning operator *&s* tells the matcher to scan until it finds *$verb*, which matches the header of any clausal caseframe. Finally, *$rest* matches the remaining input.

If the top-level pattern successfully matches, Plume uses the associated parse template to direct its more detailed processing of the input. The goal of this second pass through the input is to instantiate the caseframe corresponding to the header matched by $verb in the top-level pattern. The concept of a *kernel-caseframe* is important to this stage of processing. A kernel-caseframe corresponds to that part of an input that can be processed according to the algorithm already built into Plume for declarative and imperative sentences.

[11]In this pattern, only inputs where the fronted auxiliary is the first word in the sentence are allowed. The more complex pattern that is actually used by Plume allows prepositionally marked cases to appear initially as well

The parse template associated with the above top-level pattern for yes/no questions is:

:aux :kernel-caseframe
+ (:query)

This template tells the parser that the input consists of the auxiliary verb matched in the first pass followed by a :kernel-caseframe. For example. in:

Did Jim create foo?

the auxiliary verb. "did". appears first followed by a kernel-caseframe. "Jim create foo" Note how the kernel-caseframe looks exactly like a declarative sentence. and so can be parsed according to the usual declarative/imperative parsing algorithm

In addition to specification of where to find components of the primary caseframe. a parse template includes annotations (indicated by a plus sign) In the above template for yes/no questions. there is just one annotation - *query*. Some annotations. like this one indicate what type of input has been found. while others direct the processing of the parse template. Annotations of the first type record which case is being queried in wh questions. that is. which case is associated with the wh word. Wh questions thus include one of the following annotations: *subject-query*. *object-query*. and *marked-case-query* Marked case queries correspond to examples like:

On what day did Jim create foo?
What day did Jim create foo on?

in which a case marked by a preposition is being asked about. As illustrated here the case-marker in such queries can either precede the wh word or appear somewhere after the verb. To deal with this. the parse template for marked case queries has the annotation *floating-case-marker*. This annotation is of the second type that is it affects the way Plume processes the associated parse template.

Some top-level patterns result in two possibilities for parse templates. For example. the following top-level pattern

```
<wh-word>  <aux>  (  ($verb !!  <aux>))  $verb  $rest
```

could match an object query or a marked case query. including the following:

What did Jim create?
By whom was foo created?[12]
Who was foo created by?

These inputs cannot be satisfactorily discriminated by a top-level pattern. so the above top-level pattern has two different parse templates associated with it:

wh-object :aux :kernel-caseframe
+ (:object-query)

wh-marked-case-filler :aux :kernel-caseframe
+ (:marked-case-query :floating-case-marker) .

When the above top-level pattern matches. Plume tries to parse the input using both of these parse templates. In general. only one will succeed in accounting for all the input. so the ambiguity will be eliminated by the methods already built into Plume.

The method of parsing interrogatives presented above allows Plume to handle a wide variety of interrogatives in a very general way using domain specific semantic caseframes. The writer of the caseframes does not have to worry about whether they will be used for imperative. declarative. or interrogative sentences. (or in relative clauses). He is free to concentrate on the domain-specific grammar. In addition. the concept of the :kernel-caseframe allows Plume to use the same efficient caseframe-based parsing algorithm that it used for declarative and imperative sentences to parse major subparts of questions.

3. Conclusion

Previous work (e.g. [4. 5. 8]) and experience with our current implementation of Plume. Carnegie Group's semantic caseframe parser. has shown semantic caseframe instantiation to be an efficient and highly robust method of parsing restricted domain input However like other methods of parsing heavily dependent on restricted domain semantics. these initial attempts at parsers based on semantic caseframe instantiation suffer from patchy syntactic coverage.

After first describing the current implementation of Plume, this paper presented a detailed design for endowing Plume with much broader syntactic coverage including passives, interrogatives, and relative clauses. Relative clauses are accommodated through some grammar preprocessing and a minor change in the processing of nominal caseframes. Handling of interrogatives relies on a set of rules for classifying inputs into one of a limited number of types. Each of these types has one or more associated parse templates which guide the subsequent detailed parse of the sentence. As the final version of this paper is prepared (late April, 1985), the handling of passives and interrogatives has already been implemented in an internal development version of Plume, and relative clauses are expected to follow soon.

Though the above methods of incorporating syntactic generality into Plume do not cover all of English syntax, they show that a significant degree of syntactic generality can be provided straightforwardly by a domain specific parser driven from a semantic caseframe grammar

References

1. Bobrow, R. J. The RUS System. BBN Report 3878, Bolt, Beranek, and Newman, 1978

2. Brown, J. S. and Burton, R. R. Multiple Representations of Knowledge for Tutorial Reasoning. In *Representation and Understanding*, Bobrow, D. G. and Collins, A., Ed., Academic Press, New York, 1975, pp. 311-349.

3. Burton, R. R. Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems. BBN Report 3453, Bolt, Beranek, and Newman, Inc., Cambridge, Mass., December, 1976.

4. Carbonell, J. G., Boggs, W. M., Mauldin, M. L., and Anick, P. G. The XCALIBUR Project: A Natural Language Interface to Expert Systems. Proc. Eighth Int. Jt. Conf. on Artificial Intelligence, Karlsruhe, August, 1983.

5. Carbonell, J. G. and Hayes, P. J. "Recovery Strategies for Parsing Extragrammatical Language". *Computational Linguistics 10* (1984).

6. Grosz, B. J. TEAM: A Transportable Natural Language Interface System. Proc. Conf on Applied Natural Language Processing, Santa Monica, February, 1983.

7. Haas, N. and Hendrix, G. G. An Approach to Acquiring and Applying Knowledge. Proc. National Conference of the American Association for Artificial Intelligence, Stanford University, August, 1980, pp. 235-239

8. Hayes, P. J. and Carbonell, J. G. Multi-Strategy Parsing and its Role in Robust Man-Machine Communication. Carnegie-Mellon University Computer Science Department, May, 1981.

9. Hendrix, G. G. Human Engineering for Applied Natural Language Processing. Proc Fifth Int. Jt. Conf on Artificial Intelligence, MIT, 1977, pp. 183-191

10. Woods, W. A. "Cascaded ATN Grammars". *American Journal of Computational Linguistics 6*, 1 (August 1980), 1-12