Semantic Rule Based Text Generation

Michael L. Mauldin
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213 USA

ABSTRACT

This paper presents a semantically oriented, rule based method for single sentence text generation and discusses its implementation in the Kafka generator. This generator is part of the XCALIBUR natural language interface developed at CMU to provide natural language facilities for a wide range of expert systems and data bases. Kafka takes as input the knowledge representation used in XCALIBUR system and incrementally transforms it first into conceptual dependency graphs and then into English.[1]

1. Introduction

Transformational text generators have traditionally emphasized syntactic processing. One example is Bates *ILIAD* system which is based on Chomsky's theory of transformational generative grammar [1]. Another is Mann's *Nigel* program, based on the systemic grammar of Halliday [4]. In contrast, other text generators have emphasized semantic processing of text, most notably those systems based on case grammar such as Goldman's *BABEL* generator [7] and Swartout's *GIST* [9]. Kafka combines elements of both paradigms in the generation of English text.

Kafka is a rule based English sentence generator used in the XCALIBUR natural language interface. Kafka uses a transformational rule interpreter written in *Franz Lisp*. These transformations are used to convert the XCALIBUR knowledge representation to conceptual dependency graphs and then into English text. Kafka includes confirmational information in the generated text, providing sufficient redundancy for the user to ascertain whether his query/command was correctly understood.

The goal of this work has been to simplify the text generation process by providing a single computational formalism of sufficient power to implement both semantic and syntactic processing. A prototype system has been written which demonstrates the feasibility of this approach to single sentence text generation.

[1]This research is part of the XCALIBUR project at CMU. Digital Equipment Corporation has funded this project as part of its artificial intelligence program. XCALIBUR was initially based on software developed at CMU. Members of the XCALIBUR team include: Jaime Carbonell, Mark Boggs, Michael Mauldin, Peter Anick, Robert Frederking, Ira Monarch, Steve Morrisson and Scott Safier.

2. The XCALIBUR Natural Language Interface

XCALIBUR is a natural language interface to expert systems. It is primarily a front-end for the XCON/XSEL expert systems developed by John McDermott [5]. XCALIBUR supports mixed-initiative dialogs which allow the user to issue commands, request data, and answer system queries in any desired order. XCALIBUR correctly understands some forms of ellipsis, and incorporates expectation directed spelling correction as error recovery steps to allow the processing of non-grammatical user input.

Figure 2-1 shows the gross structure of the XCALIBUR interface. Figure 2-2 shows some typical user queries and the corresponding responses from the generator[2]. More details about XCALIBUR can be found in [2].

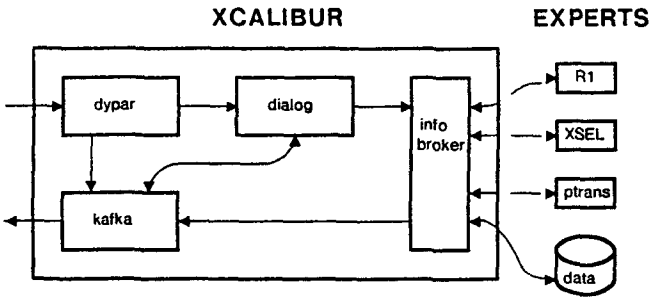

Figure 2-1: The XCALIBUR Natural Language Interface

3. Declarative Knowledge Representation

The XCALIBUR system uses a case-frame based *interlingua* for communication between the components. To provide as nearly canonical a representation as possible, the semantic information in each case-frame is used to determine the major structure of the tree, and any syntactic information is stored at the leaves of the semantic tree. The resulting case-frame can be converted into a canonical representation by merely deleting the leaves of the tree. The canonical representation is very useful for handling ellipsis, since phrases such as "dual port disk" and "disk with two ports" are represented identically. Figure 3-1 shows a sample query and its representation with purely syntactic information removed.

[2]These responses include confirmational text to assure the user that his query has been understood. Without this requirement, these sentences would have been rendered using anaphora, resulting in *It costs 38000 dollars*, or even just *38000 dollars*. See section 5.3.

+ *What is the largest 11780 fixed disk under $40,000?*
  The rp07-aa is a 516 MB fixed pack disk that costs 38000 dollars.

+ *Tell me about the lxy11.*
  The lxy11 is a 240 l/m line printer with plotting capabilities.

+ *Tell me all about the largest dual port disk with removable pack.*
  The rm05-ba is a 39140 dollar 256 MB dual port disk with removable pack, 1200 KB peak transfer rate and 38 ms access time.

+ *What is the price of the largest single port disk?*
  The 176 MB single port rp06-aa costs 34000 dollars.

Figure 2-2: Sample Queries and Responses

+ *What is the price of the two largest single port disks?*

```
(*clause
  (head (*factual-query))
  (destination (*default))
  (object
    (*nominal
      (head (price))
      (of
        (*nominal
          (head (disk))
          (ports (value (1)))
          (size (value (*descending))
              (range-high (1))
              (range-low (2))
              (range-origin
                  (*absolute)))
          (determiner (*def))))
      (determiner (*def))))
  (level (*main))
  (verb
    (*conjugation
      (root (be))
      (mode (*interrogative))
      (tense (*present))
      (number (*singular)))))
```

Figure 3-1: A Sample Case-frame

4. The Kafka Generator

Kafka is used to build replies to user queries, to paraphrase the user's input for clarificational dialogs, and to generate the system's queries for the user. Figure 4-1 shows the major components and data flow of the Kafka generator. Either one or two inputs are provided: (1) a case frame in the XCALIBUR format, and (2) a set of tuples from the information broker (such as might be returned from a relational database). Either of these may be omitted. Four of the seven major components of Kafka use the transformational formalism, and are shown in bold outlines.

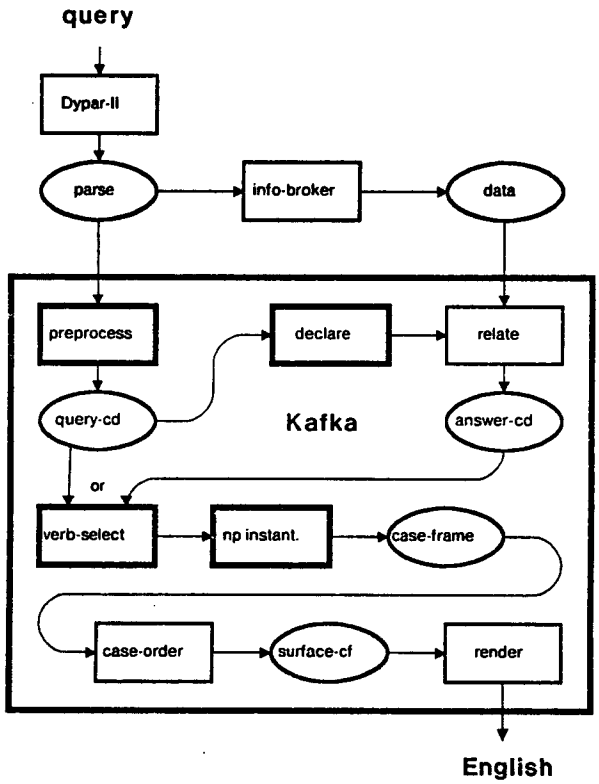

Figure 4-1: Data flow in the Kafka Generator

4.1. Relation to Other Systems

Kafka is a direct descendant of an earlier natural language generator described in [2], which in turn had many components either derived from or inspired by Goldman's BABEL generator [7]. The case frame knowledge representation used in XCALIBUR has much in common with Schank's Conceptual Dependency graphs [8]. The earlier XCALIBUR generator was very much *ad hoc*, and Kafka is an effort to formalize the processes used in that generator. The main similarity between Kafka and BABEL is in the verb selection process (described in section 5).

The embedded transformational language used by Kafka was inspired by the OPS5 programming language developed by Forgy at CMU [3]. OPS5 was in fact an early candidate for the implementation of Kafka, but OPS5 supports only flat data structures. Since the case frame knowledge representation used in XCALIBUR is highly recursive, an embedded language supporting case frame matches was developed. The kafka programming language can be viewed as a production system with only a single working memory element and a case frame match rather than the flat match used in OPS5.

4.2. Transformational Rules

Some of the transformation rules in Kafka were derived from the verb selection method of BABEL, and others were derived taken from TGG rules given in [10]. Although Kafka has been based mainly on the generative grammar theory of Chomsky, the rule syntax allows much more powerful rules

than those allowed in either the standard or extended standard theory. We have tried to provide a sufficiently powerful formalism to encode more than one grammatical tradition, and have not restricted our rules to any particular linguistic convention. Our goal has not been to validate any linguistic theory but rather to demonstrate the feasibility of using a single computational mechanism for text generation.

The basic unit of knowledge in XCALIBUR is the *case frame*. Kafka repeatedly transforms case frames into other case frames until either an error is found, no pattern matches, or a surface level case frame is generated. The surface case frame is converted into English by *render*, which traverses the case frame according to the sentence plan, printing out lexical items. A transformation is defined by an ordered set of rules.

Each rule has up to four parts:

- A **pattern**, which is matched against the current node. This match, if successful, usually binds several local variables to the sub-expressions matched.
- A **result**, which is another case frame with variables at some leaves. These variables are replaced with the values found during the match. This process is called *instantiation*.
- An optional **variable check**, the name of a lisp function which takes a binding list as input and returns either **nil** which causes the rule to fail, or a new binding list to be used in the instantiation phase. This feature allows representation of functional constraints.
- An optional **final** flag, indicating that the output from this rule should be returned as the value of the rule set's transformation.

A transformation is applied to a case frame by first recursively matching and instantiating the sub-cases of the expression and then transforming the parent node. Variables match either a single s-expression or a list of them. For example =HEAD would match either an atom or a list, =*REST would match zero or more s-expressions, and =+OTHER would match one or more s-expressions. If a variable occurs more than once in a pattern, the first binds a value to that variable, and the second and subsequent occurrences must match that binding exactly.

This organization is very similar to that of the ILIAD program developed by Bates at BBN [1]. The *pattern*, a *result*, and *variable check* correspond to the *structural description*, *structural change*, and *condition* of Bates' transformational rules, with only a minor variation in the semantics of each operation. The ILIAD system, though, is very highly syntax oriented (since the application is teaching English grammar to deaf children) and uses semantic information only in lexical insertion. The rules in Kafka perform both semantic and syntactic operations.

4.3. A Sample Rule

Figure 4-2 sample rule from the Kafka grammar for the XCALIBUR domain. The rule takes a structure of the form *The price of X is FOO* and converts it to *X costs FOO*. More sophisticated rules for verb selection check for semantic agreement between various slot fillers, but this rule merely encodes knowledge about the relationship between the PRICE attribute and the COST verb. Figure 4-3 shows an input structure that this rule would match; Figure 4-4 shows the structure which would be returned.

```
(R query-to-declare active-voice-cost
    (cd (primitive (be))
        (actor (*nominal
                   (head: (price))
                   (of =x)
                   =*other))
        (object =y)
        =*be-rest)
    =>
    (cd (primitive (cost))
        (actor =x)
        (object =y)
        =be-rest))
```

Figure 4-2: Rule 'active-voice-cost'

```
(cd (primitive (be))
    (actor
        (*nominal (head: (price))
            (of (*nominal (ports (1))
                    (size
                        (*descending
                            (range-low: (1))
                            (range-high: (2))
                            (range-origin:
                            (*absolute))))
                    (head: (disk))
                    (determiner: (*def))))
            (determiner: (*def))))
    (object (*unknown (head: (price))))
    (destination (*default))
    (level: (*main))
    (tense: (*present))
    (number: (*singular)))
```

Figure 4-3: Input Case Frame

```
(cd (primitive (cost))
    (actor
        (*nominal
            (ports (1))
            (size
                (*descending
                    (range-low: (1))
                    (range-high: (2))
                    (range-origin:
                        (*absolute))))
            (head: (disk))
            (determiner: (*def))))
    (object (*unknown (head: (price))))
    (destination (*default))
    (level: (*main))
    (tense: (*present))
    (number: (*singular)))
```

Figure 4-4: Output Case Frame

5. The Generation Process

The first step in the generation process is preprocessing, which removes a lot of unnecessary fields from each case frame. These are mostly syntactic information left by the parser which are not used during the semantic processing of the query. Some complex input forms are converted into simpler forms. This step provides a high degree of insulation between the XCALIBUR system and the text generator, since changes in the XCALIBUR representation

can be caught and converted here before any of Kafka's internal rules are affected.

In the second phase (not used when paraphrasing user input) the case frame is converted from a query into a declarative response by filling some slots with (*UNKNOWN) place-holders. Next the *relat* module replaces these place-holders with information from the back-end (usually data from the XCON static database). The result is a CD graph representing a reply for each item in the user's query, with all the available data filled in.

In the third phase of operation, the *verb* transform selects an English verb for each underlying CD primitive. Verb selection is very similar to that used by Goldman in his BABEL generator [7], except that BABEL uses hand-coded discrimination nets to select verbs, while Kafka keeps the rules separate. A rule compiler is being written which builds these discrimination nets automatically. The D-nets are used to weed out rules which cannot possibly apply to the current structure. Since the compiler in not yet installed, Kafka uses an interpreter which tries each rule in turn.

After verb selection, the *np-instantiation* transform provides lexical items for each of the adjectives and nouns present in each CD graph. Finally the *order* module linearizes the parse tree by choosing an order for the cases and deciding whether they need case markers. The final sentence is produced by the *render* module, which traverses the parse tree according to the *sentence plan* produced by *order*, printing the lexical items from each leaf node.

5.1. A Sample Run

The following is a transcript of an actual generation run which required 30 seconds of CPU time on a VAX 11/780 running under Franz Lisp. Most of the system's time is wasted by attempting futile matches during the first part of the match/instantiate loop. The variable **parse1** has been set by the parser to the case frame shown in Figure 3-1. The variable **data1** is the response from the information broker to the user's query. This transcript shows the Kafka system combining these two inputs to produce a reply for the user including (1) the answer to his direct query and (2) the information used by the information broker to determine that answer.

```
-> (print data1)

((name class number-of-megabytes ports price)
 ((rp07-aa disk 516 1 38000)))

-> (render-result parse1 data1)

Applying rules for preparse...

        Rule 'input-string-deletion' applied...
        Rule 'input-string-deletion' applied...
        Rule 'position-deletion' applied...
        Rule 'property-fronting1' applied...
        Rule 'input-string-deletion' applied...
        Rule 'position-deletion' applied...
        Rule 'property-fronting1' applied...
        Rule 'input-string-deletion' applied...
        Rule 'modifiers-breakout' applied...
        Rule 'modifiers-breakout' applied...
        Rule 'modifiers-deletion' applied...
        Rule 'input-string-deletion' applied...
        Rule 'project' applied...
        Rule 'input-string-deletion' applied...
        Rule 'cases-breakout' applied...
        Rule 'cases-breakout' applied...
        Rule 'cases-deletion' applied...

Applying rules for query-to-declare...

        Rule 'fact-to-cd' applied...
        Rule 'active-voice-cost' applied...

Applying rules for verb-select...

        Rule 'cd-cost' applied...

Applying rules for np-instantiate...

        Rule 'k-largest' applied...
        Rule 'size-deletion' applied...
        Rule 'prenominal-megabytes' applied...
        Rule 'prenominal-single-port' applied...
        Rule 'nominal-price' applied...


(cf (verb (cost))
    (agent
        (*nominal
            (head: (disk))
            (prenominal: (516 MB) (single port))
            (determiner: (*def))
            (name (rp07-aa))))
    (object
        (*nominal
            (head: (dollar))
            (determiner: (*generic))
            (count (38000))))
    (destination (*default))
    (level: (*main))
    (mode: (declarative))
    (voice: (active))
    (tense: (*present))
    (number: (singular))
    (person: (third))
    (subject: (agent))
    (plan: ((unmarked agent) *verb (unmarked object)))
    (verb-conj: (costs)))
```

And the resulting surface string is:

The 516 MB single port rp07-aa costs 38000 dollars.

5.2. Generating Anaphora

Kafka has minimal capability to generate anaphora. A discourse history is kept by the dialog manager which maps each nominal case frame to a surface noun phrase. Anaphoric references are generated by choosing the shortest noun phrase representing the nominal not already bound to another nominal. Thus the pronoun *it* could only refer to a single item. Each noun phrase is generated in the output order, so the discourse history can be used to make decisions about anaphoric reference based on what the user has read p to that point. This technique is similar to but less sophisticated than that used by McDonald [6]. Generation of anaphora is inhibited when new information must be displayed to the user, or when confirmational text is to be included.

5.3. Confirmational Information

Speakers in a conversation use redundancy to ensure that all parties understand one another. This redundancy can be incorporated into natural language interfaces by "echoing," or including additional information such as paraphrases in the generated text to confirm that the computer has chosen the correct meaning of the user's input. For example, of the user asks:

+ *What is the price of the largest single port disk?*

The following reply, while strictly correct, is likely to be unhelpful, and does not reassure the user that the meaning of the query has been properly understood:

34000 dollars.

The XCALIBUR system would answer with the following sentence which not only answers the user's question, but includes evidence that the system has correctly determined the user's request:

The 176 MB single port rp06-aa costs 34000 dollars.

XCALIBUR uses focus information to provide echoing. As each part of the user's query is processed, all the attributes of the object in question which are needed to answer the query are recorded. Then the generator assures that the value of each of these attributes is presented in the final output.

6. Summary and Future Work

The current prototype of the Kafka generator is running and generating text for both paraphrases and responses for the XCALIBUR system. This system demonstrates the feasibility of the semantic transformational method of text generation. The transformational rule formalism allows much simpler specification of diverse syntactic and semantic computations than the hard-coded lisp used in the previous version of the XCALIBUR generator.

Current work on Kafka is focused on three goals: first, more grammar rules are necessary to increase coverage. Now that the basic method has been validated, the grammar will be extended to cover the entire XCALIBUR domain. The second goal is making Kafka more efficient. Most of the system's time is spent trying matches that are doomed to fail. The discrimination network now being added to the system will avoid these pointless matches, providing the speed required in an interactive system like XCALIBUR. The third goal is formalizing the remaining *ad hoc* phases of the generator. Four of seven major components now use transformations; two of these seem amenable to the transformational method. The case ordering can easily be done by transformations. Converting the semantic processing done by the *relat* module will be more difficult, since the rule interpreter will have to be expanded to allow multiple inputs.

7. Acknowledgments

Many thanks to Karen Kukich for her encouragement and for access to her wealth of literature on text generation. I would also like to thank Jaime Carbonell for his criticism and suggestions which were most helpful in revising earlier drafts of this paper.

References

1. Bates, M. and Wilson, K., "Final Report, ILIAD, Interactive Language Instruction Assistance for the Deaf," Tech. report, Bolt Berank and Newman, 1981, No. 4771.

2. Carbonell, J.G., Boggs, W.M., Mauldin, M.L. and Anick, P.G., "The XCALIBUR Project, A Natural Language Interface to Expert Systems," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

3. Forgy, C.L., "OPS5 User's Manual," Tech. report, Dept. of Computer Science, Carnegie-Mellon University, 1981, CMU-CS-81-135.

4. Mann, W.C., "An Overview of the Nigel Text Generation Grammar," *Proceedings of the 21st Meeting of the Association for Computational Linguistics*, 1983, pp. 79-84.

5. McDermott, J., "R1: A Rule-Based Configurer of Computer Systems," Tech. report, Dept. of Computer Science, Carnegie-Mellon University, 1980.

6. McDonald, D.D., "Subsequent Reference: Syntactic and Rhetorical Constraints," *Theoretical Issues in Natural Language Processing-2*, 1978, pp. 64-72.

7. Schank, R.C., *Conceptual Information Processing*, Amsterdam: North-Holland, 1975.

8. Schank, R.C. and Riesbeck, C.K., *Inside Computer Understanding*, Hillside, NJ: Lawrence Erlbaum, 1981.

9. Swartout, B., "GIST English Generator," Tech. report, USC/Information Sciences Institute, 1982.

10. Wardhaugh, R., *Introduction to Linguistics*, McGraw Hill, 1977.