A SEMANTIC INTERPRETER FOR SYSTEMIC GRAMMARS

Tim F. O'Donoghue
timmy@uk.ac.leeds.ai

Division of Artificial Intelligence
School of Computer Studies
The University of Leeds
Leeds LS2 9JT. UK
+44 532 335430

ABSTRACT

This paper describes a method for obtaining the semantic representation for a syntax tree in Systemic Grammar (SG). A prototype implementation of this method — the REVELATION1 semantic interpreter — has been developed. It is derived from a SG generator for a large subset of English — GENESYS — and is thus, in contrast with most reversible grammars, an interpreter based on a generator. A task decomposition approach is adopted for this reversal process which operates within the framework of SG, thus demonstrating that Systemic Grammars can be reversed and hence that a SG is a truly bi-directional formalism.

Introduction

SG (see Butler [4] for a good introduction) is a useful model of language, having found many applications in the areas of stylistics, text analysis, educational linguistics and artificial intelligence. Some of these applications have been computational, the best known probably being Winograd's SHRDLU [22]. However, most computational applications have been designed from a text-generation viewpoint (such as Davey's PROTEUS [6], Mann and Matthiessen's NIGEL [16, 17] and Fawcett and Tucker's GENESYS [10]).

Because of this text-generation viewpoint of systemic grammarians, the mechanism for sentence analysis within SG (the reverse of the sentence generation process) has received much less attention. This paper describes one stage of the sentence analysis process: **semantic interpretation.**[1]

In Fawcett's SG,[2] a syntax tree (whose leaves define a sentence) is generated from a set of semantic choices. REVELATION1 reverses this process: it attempts to find the set of semantic choices needed to generate a given syntax tree. In sentence generation, a tree is generated, but only the leaves (the words) are 'output', the rest of the tree is simply deleted. In the reverse process, when a sentence is input, a syntax tree must first be found before it can be interpreted. REVELATION1 assumes that a separate SG parser (not discussed here) is available; for an example of such a parser see O'Donoghue [20]. Thus REVELATION1 directly mirrors the generator, while the parser mirrors the tree deletion process.

REVELATION1 has been developed within the POPLOG environment.[3] It is coded in a combination of POP11 [1] and Prolog [5] and utilizes GENESYS. GENESYS is a very large SG generator for English, written in Prolog (Fawcett and Tucker [10]) and it is version PG1.5 that has been used for the development and testing of REVELATION1. GENESYS and REVELATION are part of a much larger project, the COMMUNAL[4] Project, the aim of which is to build a natural language interface to a rich SG oriented IKBS in the domain of personnel manage-

[1] This assumes that sentence analysis can be decomposed into two processes: syntactic analysis (parsing) plus semantic interpretation. These processes are not necessarily sequential, although it greatly simplifies things if they are treated as such. Here I assume a sequential scheme in which the parsing process passes a syntax tree to the interpreter.

[2] Fawcett's Systemic Functional Grammar [8, 9], his development of a Hallidayan SG.

[3] POPLOG is a multi-language development environment containing incremental compilers for POP11 (the base language), Prolog, Common Lisp and Standard ML, an integrated editor and numerous support tools [12].

[4] The COMMUNAL Project at University of Wales College of Cardiff (UWCC) Computational Linguistics Unit and Leeds University School of Computer Studies was sponsored by RSRE Malvern, ICL and Longman.

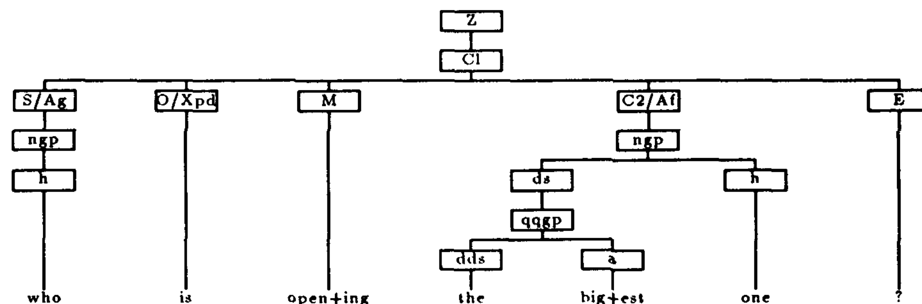

Figure 1: A Typical PG1.5 Syntax Tree

ment.

This is not the first time a semantic interpreter has been attempted for a large SG: Kasper [13] has developed a sentence analysis method (both parsing and interpretation together in my terminology) based on the NIGEL grammar. In his approach the SG is *compiled* into a Functional Unification Grammar (FUG) (see Kay [14], a representation language with some systemic links) and then existing (but extended) FUG parsing techniques are used to find a syntax tree plus an interpretation for a sentence.

The REVELATION1 approach differs from this compilation method since the interpretation is achieved *within* a systemic framework. No other SG-based model (to my knowledge) has been used for both generation *and* interpretation in this way.

Systemic Grammar

Fawcett's SG is a meaning-oriented model of language in which there is a large 'and/or' network of semantic features, defining the choices in meaning that are available. A syntax tree is generated by traversing multiple paths (see later) through this network, making choices in meaning, and so defining the meaning of the syntax tree to be generated. These choices fire realization rules which specify the structural implication of choosing certain features; they map the semantic choices onto syntactic structures and so transform the chosen meaning into a syntax tree (and hence a sentence) which expresses that meaning.

As an example of the syntax trees which are generated by PG1.5, consider Figure 1. Systemic syntax trees consist of a number of levels of structure called units; the tree in Figure 1 has four: one clause (**Cl**), two nominal groups (**ngp**) and a quantity-quality group (**qqgp**). The components (immediate constituents) of each unit are elements of structure labelled to represent the functions fulfilled by that component with respect to its unit. For example: subject (**S**), main verb (**M**), second complement (**C2**) and ender (**E**) in the clause, superlative determiner (**ds**) and head (**h**) in the nominal groups, superlative deictic determiner (**dds**) and apex (**a**) in the quantity-quality group. Some items may expound more than one function in a unit, e.g. "is" functions both as operator (**O**) and period-marking auxiliary (**Xpd**) in the clause and is labelled with the conflated functional label **O/Xpd**. Some elements may be conflated with participant roles, e.g. "who" is a subject playing the role of agent (**Ag**) and so is labelled **S/Ag**. Similarly "the big+est one" is a complement playing the role of affected (**Af**) and hence is labelled **C2/Af**. The immediate constituent of an element of structure is either a lexical item (either a word or punctuation, in which case we say that the item expounds the element, e.g. the lexical item "one" expounds **h**), or a further unit (when we say a unit fills the element, e.g. the unit **qqgp** fills **ds**).

Having introduced the type of syntax tree that is generated, let us now consider the actual process of generation. The key concept in SG is that of choice between small sets of meanings (systems of semantic features). For example, the **NUMBER** system contains the choices **singular** and **plural**. The choice systems in a systemic grammar are linked together by 'and' and 'or' relationships to form a complex system network, specifying the preconditions and consequences of choosing features. Consider the system network presented in Figure 2 (an excerpt from PG1.5 which contains ≈450 systems, some containing many more features than the binary systems illustrated in this example). In the systemic notation curly braces represent conjunctions and vertical bars represent exclusive disjunctions, i.e. choice. The upper-case labels are the names of systems and

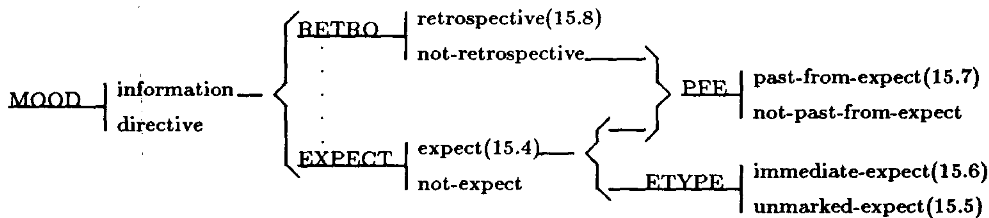

Figure 2: A fragment of the English 'tense' Network

```
directive
information, retrospective, not-expect ................................................... "touch"
information, retrospective, expect, immediate-expect ....................... "has been about to touch"
information, retrospective, expect, unmarked-expect ........................ "has been going to touch"
information, not-retrospective, expect, immediate-expect, past-from-expect ........ "is about to have touched"
information, not-retrospective, expect, immediate-expect, not-past-from-expect ............ "is about to touch"
information, not-retrospective, expect, unmarked-expect, past-from-expect .......... "is going to have touched"
information, not-retrospective, expect, unmarked-expect, not-past-from-expect ............. "is going to touch"
```

Figure 3: Examples of 'tense' Selection Expressions and their realizations

```
15.5:unmarked-expect:
    G @ 73, G < "going to".

15.6:immediate-expect:
    G @ 73, G < "about to".

15.7:past-from-expect:
    Xpf @ 85, Xpf < "have",
    if period-marked then Xpd <+ "en"
    else if unmarked-passive then Xp <+ "en".
```

Figure 4: Realisation Rules

the lower-case labels are the names of the features in those systems. Each system has an entry condition; a precondition which must be met in order to enter the system and make a choice. For example, to enter the **RETRO** (retrospectivity) and **EXPECT** (expectation) systems, **information** must have been chosen. To enter the **PFE** (past from expectation) system, both **not-retrospective** and **expect** must have been chosen. The sets of choices in meaning defined by this network fragment are listed in Figure 3. Each set of choices is a **selection expression**, i.e. a path (typically bifurcating) through the network.

Associated with certain features are realization rules. In Figure 2, the bracketted numbers are pointers to realization rules; thus realization rule 15.7 is triggered by the feature **past-from-expect**. A realization rule specifies the structural consequences of choosing a feature; they map the semantic choices onto syntactic structures. Often conditions are involved; consider the realization rules shown in Figure 4 (PG1.5 contains ≈500 realization rules, many of which are far more complex than these examples). The main types of rule are:

- Componence Rules: **El@N**, stating that the element **El** is at place **N** in the current unit (thus placing an ordering on the components of the unit currently being generated). These place numbers are relative rather than physical; an element at place **N** states that the element (if realized) will appear after elements whose places are less than **N** and before those elements whose places are greater than **N**.
- Filling Rules: **is_filled_by U**, defining the unit **U** to be generated, e.g. **U=ngp**.
- Conflation Rules: **F2 by F1**, stating that the two functions **F1** and **F2** are conflated with one another in the current unit (e.g. a Subject which also functions as an Agent).
- Exponence Rules: **El<Word**, stating that the element **El** is expounded by an item (e.g. **M<"open"**), i.e. exponence creates terminal constituents.
- Re-entry Rules: **for F re_enter_at f**, stating that the function **F** is filled by a unit which is generated by re-entering the network at the feature **f**.
- Preference Rules: **for F prefer [f...]**,

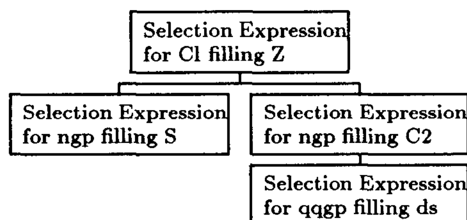

Figure 5: A Prototype Semantic Representation

stating that when re-entering to generate a unit to fill the function F, the features [f...] should be preferred, either absolutely (i.e. 'pre-selection') or tentatively (expressed as a percentage).

A sentence is generated by generating the syntax tree for the sentence, the leaves in this tree being the words of the sentence. The tree is generated by generating each of its units in a top-down fashion. Each unit is generated by a single pass through the system network. This pass is expressed as a selection expression which lists the features chosen on that pass. For example, suppose we wanted to generate the sentence "the prisoner was going to have been killed". The selection expression for the clause would contain the features (plus many others, of course):

```
information, not-retrospective, expect,
unmarked-expect, past-from-expect.
```

Any realization rules associated with features in the selection expression are then executed. Rules 15.5 and 15.7 in Figure 4 generate a structure whose leaves are "...going to ...have ...(be)en ...". For example, to generate the clause structure in Figure 1, the following realization statements were executed:

```
is_filled_by Cl
S@35, O@37, M@94, C2@106, E@200
Ag by S, Xpd by O, Af by C2
O<"is", M<"open", M<+"+ing", E<"?"
for Ag re_enter_at stereotypical_thing
for Af re_enter_at thing
```

The re-entry realization statements show which functions are to be filled by re-entering the system network to generate further units. Re-entry can be thought of as a recursive function call which generates a lower layer of structure in the tree — but typically with some of the choices 'preferred' via the preference realization statement. Thus the syntax tree in Figure 1 is generated with four passes:

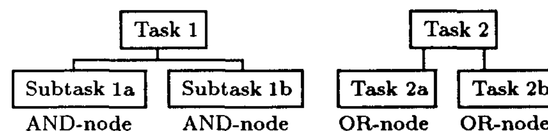

Figure 6: AO Tree Primitives

the clause is generated first, followed by the nominal group filling the subject agent, followed by the nominal group filling the affected complement, and finally the quantity-quality group filling the superlative determiner.

The information required to generate a syntax tree can be expressed as a tree of selection expressions; this is the **semantic representation** for the sentence. Each node in the semantic representation corresponds to a unit in the syntax tree and is labelled by the selection expression for that unit. For example, a semantic representation of the form shown in Figure 5 is needed to generate the syntax tree in Figure 1.

Interpreting a Syntax Tree

Given a syntax tree, the aim of interpretation is to find the semantic representation which would generate that tree. This semantic representation includes all the features that are needed to generate the tree and so defines the 'meaning content' of the syntax tree.

In the process of generation, a syntax tree is generated by generating its units; in the process of interpretation a syntax tree is understood by interpreting all of its units in a precisely analogous way. Thus a unit interpretation is the selection expression which generated that unit. In general there can be more than one selection expression for any given unit, since the same syntactic structure can have more than one meaning, just as a whole sentence can have more than one meaning. The potential unit interpretations are defined by constructing an AO tree[5] whose goal (root) is to prove unit realization, i.e. prove that the unit can be generated. Each potential solution of this AO tree defines a poten-

[5] AO (AND/OR) trees provide a means for describing task decomposition. These structures were first proposed by Slagle [21] and have since been used in a variety of applications including symbolic integration and theorem proving (Nilsson [18] lists a number of applications with references). The AO tree notation used throughout this paper is illustrated in Figure 6 with leaves — terminal tasks which cannot be decomposed any further — being represented as **bold nodes**.

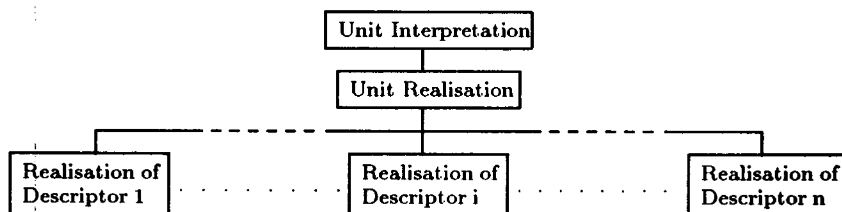

Figure 7: Decomposition of Unit Interpretation

tial selection expression for the unit. The semantic representation for the tree is given by some feasible combination of the potential selection expressions for each unit in the syntax tree. Not all combinations of selection expressions are feasible since the generation passes, and hence the selection expressions, are *interdependent*. Thus:

- A pass is dependent upon previous, higher passes through the use of the unary boolean operators **on_prev_pass** and **on_first_pass** in conditions inside realization rules. These operators are used to test the values of features in passes for higher units in the syntax tree. For example the condition **on_first_pass written** is only true if **written** is selected on the first pass.
- Subsequent passes are dependent upon the pre-selections that are made with the preference rules. For example, when generating a question (such as that in Figure 1) it is necessary to pre-select the feature **seeking-specification-of-thing** for the nominal group which fills the subject agent. This ensures that a Wh-subject is generated.

Decomposing Unit Interpretation

The first step in unit interpretation is the decomposition of the unit's structure into a set of descriptors;[6] each describing a different aspect of the unit's structure. For example, the descriptors required to describe the clause in Figure 1 are:

```
unit(Cl)
el(1,S), el(2,O), el(3,M), el(4,C2), el(5,E)
conf(S,Ag), conf(O,Xpd), conf(C2,Af)
stem(O,"is"), stem(M,"open"), stem(E,"?")
suffix(M,"+ing")
re_enter(S,Ag), re_enter(C2,Af)
```

[6] A descriptor is simply an abstract description of a realization; it attempts to capture the effect of the realization statements without using any of their syntax.

The **unit** descriptor specifies the name of the current unit, the **el** descriptor specifies the ordering of the component elements, the **conf** descriptor specifies the conflation relationships, the **stem** and **suffix** descriptors specify the items expounding lexical elements, and the **re_enter** descriptor specifies the non-terminal elements requiring a subsequent pass to generate a unit to fill them.

Decomposition 1 [Unit Interpretation] Unit interpretation is achieved by proving that the unit can be realized. To do this, the unit's structure is described, using a set of descriptors, and hence unit realization is decomposed into a number of separate realizations, with one realization for each descriptor. Thus unit interpretation is achieved by realizing *all* descriptors (hence realizing the whole unit). This decomposition is illustrated in Figure 7.

Each descriptor is realized by some realization statement (which will appear somewhere in the realization rules). There is a simple mapping between descriptors and suitable realization statements. For some descriptors there is only a single suitable realization statement: for example, **unit(U)** is only realized by the statement **is_filled_by U**. Other descriptors can be realized by a number of different realization statements: for example, **el(i,El$_i$)** is realized by statements of the form **El$_i$@N**, where the place **N** is greater than the places for **El$_{i-1}$** and less than the places for **El$_{i+1}$**. This ensures correct ordering of the constituent elements.

For each descriptor, a search of the realization rules is performed to find any statements which realize the descriptor. For example, for the descriptor **unit(Cl)** we would search for all occurrences of **is_filled_by Cl** in the realization rules. After the search is complete there will be a set of potentially active rules for each descriptor, with each potentially active rule containing a suitable statement (or possibly more than one suitable statement) that

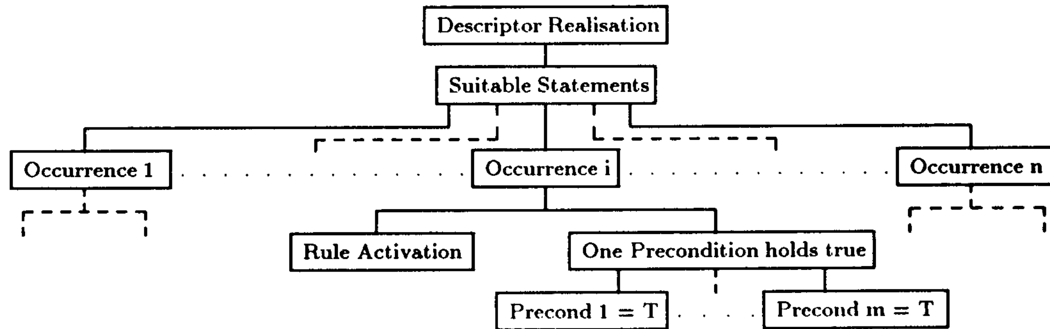

Figure 8: Decomposition of Descriptor Realisation

would realize the descriptor. Associated with each potentially active rule is a set of preconditions, one precondition for each suitable statement. (If the statement has no precondition, then the set of preconditions will be empty.)

Decomposition 2 [Descriptor Realization] Proving that a descriptor can be realized is decomposed into (i) activating any one of its potentially active rules, and (ii) proving that any one of the preconditions associated with that rule holds true (assuming there are any preconditions). This decomposition is illustrated in Figure 8.

What is required for a rule to be active? A rule can only be active if there is a feasible path (feasible in the sense that all features on that path can be selected) through the system network to a feature associated with that rule. There will be at least one potential path to each rule; 'at least' is required since a rule can have more than one potential path if (i) it is associated with more than one feature or (ii) it is dependent upon a disjunctive entry condition. Thus the potential paths to a rule can be represented as a boolean expression; i.e. as an exclusive disjunction of conjunctions:

```
path1 xor path2 xor ... xor pathN
```

where the **path** components are conjunctions composed from the features in each potential path and the exclusive disjunction represents choice between potential paths. This expression can be simplified into an expression of the form:

```
common and
(variant1 xor ... xor variantN)
```

where `common` is a conjunction of features that is common to all potential paths and `variant1` ...`variantN` are the conjunctions of features peculiar to each potential path. This expression can be considered as a precondition for rule activation and so must be true for the rule to be active.

Decomposition 3 [Rule Activation] Rule Activation is achieved by activating one of the potential paths to that rule. The potential paths can be decomposed into a common component and a number of variant components. One potential path is active if and only if the common component is active and its variant component is also active. This decomposition is illustrated in Figure 9.

At this stage in the decomposition all tip nodes are problems of the form 'boolean expression = T', i.e. satisfiability problems. In order to define how satisfiability problems are solved, the concept of a truth function must first be introduced. A truth function is a mapping between features and a three valued logic (with truth values false, true and undefined) which defines the value of each feature; true indicating selected and false indicating not selected. The problem of satisfiability for a boolean expression involves finding a truth function such that the boolean expression evaluates[7] to be true, such a truth function is called a satisfying truth function for the boolean expression. Unfortunately, this is an intractable problem since a disjunction with n disjuncts can have 3^n-1 potential satisfying truth functions, i.e. the task is exponential in problem size (in fact, the problem of satisfiability has the honor of being the first NP-Complete problem; see Garey and Johnson [11]). As is the case with inherently intractable problems; it is not worth searching for an efficient, exact algorithm to perform the task; it is

[7] Evaluation is performed with respect to Kleene's three valued logic [15].

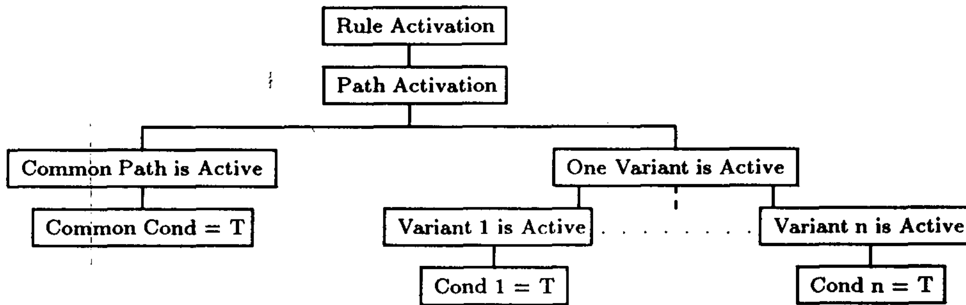

Figure 9: Decomposition of Rule Activation

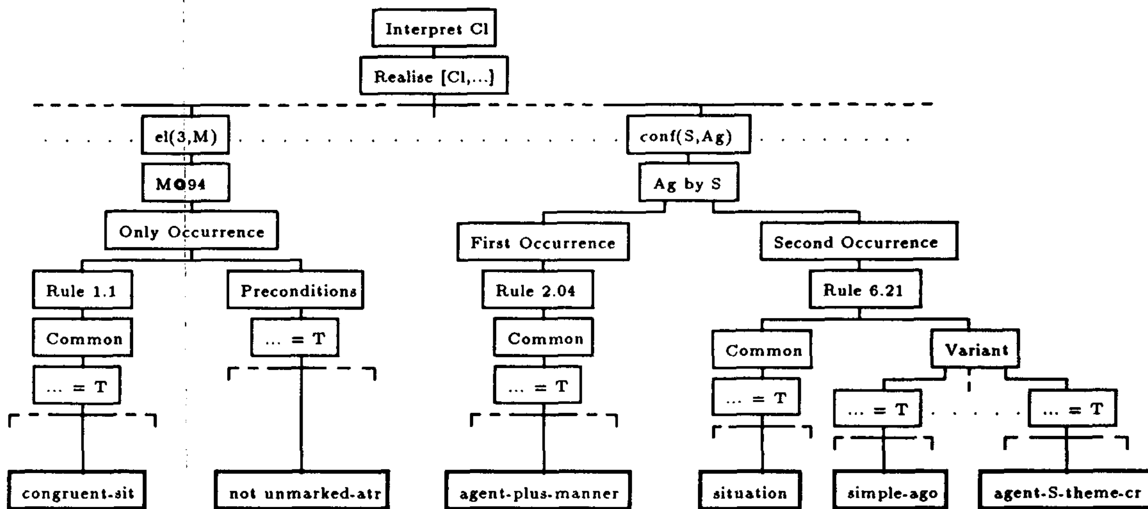

Figure 10: Part of the AO Tree for Clause Interpretation

more appropriate to consider a less ambitious problem. Consider the problem of partial satisfiability: it is essentially the same as satisfiability except that it does not attempt to satisfy disjunctive components (since it is these components which make the satisfiability problem 'hard'). It requires a redefinition of a truth function: in partial satisfiability a truth function is a mapping between boolean expressions (rather than simply features) and truth values. For example, the expression `(a or b) and c and not d` is partially satisfied by the function v:

$$v \quad : \quad v(\texttt{a or b}) = T, \;\; v(\texttt{c}) = T, \;\; v(\texttt{d}) = F$$

Since disjunctions are effectively ignored, there is a single unique partially satisfying truth function for any boolean expression. Thus tip nodes labelled with satisfiability problems are (partially) decomposed into a number of and-nodes; each and-node being labelled by a feature (possibly negated) or a disjunction which must be true. As an example of a fully decomposed AO tree for unit interpretation, consider the skeleton tree in Figure 10 which defines the potential interpretations of the clause in Figure 1.

A potential solution of an AO tree is specified by a subset of the leaves that are necessary for the root task to be achieved. A backtracking search is used to generate potential solutions: since the AO tree is finite, it is possible to inspect the structure of the tree and order the search in such a way so that the minimum amount of backtracking is performed.[8] A feasible interpretation for a unit is one of the potential solutions in which the statements labelling the leaves are all consistent. There are two ways in which a leaf statement may be inconsistent:

[8]The search scheme involves incrementally evaluating and pruning the AO tree. Full details can be found in O'Donoghue [19], an in-depth report on the interpreter.

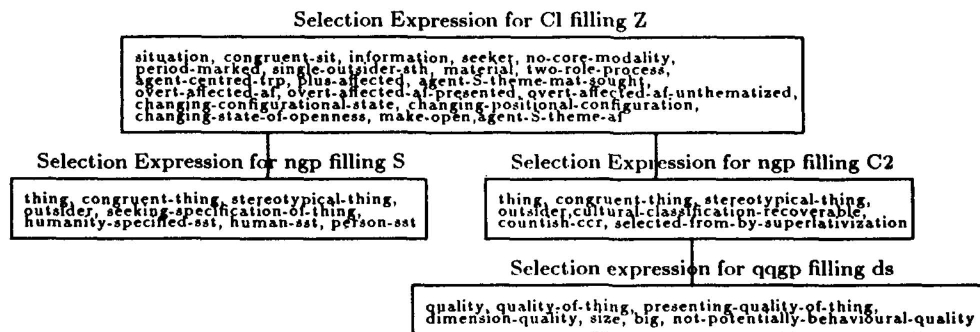

Figure 11: A Semantic Representation found by REVELATION1

- The statements logically contradict, for example: leaves labelled `a` = T and `a` = F. Or `b` or `c` = T and `b` = F, `c` = F.
- The statements systemically contradict, for example: `a` = T and `b` = T when `a` and `b` are members of the same choice system (from which at most one feature can be chosen).

The leaf statements in a feasible solution define which features must be true (i.e. selected) for the unit to be realized, i.e. they specify a selection expression for the unit.

Results and Discussion

The semantic representation found by REVELATION1 for the syntax tree in Figure 1 is presented in Figure 11. We can get a flavor of the semantic representation by identifying key features: the sentence is a question (**information**, **seeker**) about an ongoing event (**period-marked**) which involves opening something (**make-open**). It is a person that we are seeking (**person-sst**). The thing that is being opened is selected by superlativization (in fact it is the `biggest`) and it is recoverable from the previous discourse (the "one" referring to something that has previously been mentioned).

Unfortunately this semantic representation is incomplete. One of the factors contributing to this incompleteness is that of 'unrealized' selections: these are features which have no associated realization (e.g. **not-expect** in Figure 2). Consider the way in which interpretation works: it tries to prove that observed realizations have taken place and in so doing infer the features that were selected. However, if a realization does not take place as a (not necessarily direct) result of a selection, then there is no way to infer that the unrealized feature was selected. Consider the **POLARITY** system where there is a choice between **positive** and **negative**. The **positive** choice is unrealized where as **negative** is associated with the realization rule:

```
17:negative:
        do_support,
        O <+ "n't".
```

Suppose we have a positive clause, "Who is opening the biggest one?" (the example sentence from Figure 1). There is no way to tell from the structure of the clause that the sentence is positive. However by inspecting the realization rule associated with **negative** we find that the (unconditional) statement `O<+"n't"` can never be active as this would realize the sentence "Who isn't opening the biggest one?". Thus rule 17 can never be active and so **negative** can't be chosen, hence **positive** must be chosen. Thus by a process of eliminating features which cannot be true, it is possible to determine unrealized features. REVELATION2 (currently under development) will attempt to implement this process of elimination by moving forward through the system network (after a partial semantic representation has been obtained) systematically verifying any realization rules that it meets, eliminating features that cannot be chosen and so possibly inferring something about unrealized features which need to be chosen.

The other factor contributing to incompleteness is the definition of partial satisfiability. Some features in the network do not have realization rules attached to them: typically these appear as conditions on statements in realization rules. However, if any of these features appear in disjunctive conditions or disjunctive entry conditions to systems, then nothing can be inferred about their values since the definition of partial satisfiability ignores all dis-

junctions. This problem can only be overcome by searching for (exact) satisfying truth functions from which all feature values can be inferred. REVELATION2 attempts to solve this problem by deferring the search for exact satisfying truth functions until after a partial interpretation has been obtained: the partial interpretation is obtained as for REVELATION1, and then any disjunctions that have been ignored while obtaining this interpretation are exactly satisfied to try and fill the gaps in the partial interpretation.

In addition to the work on the next generation of interpreter, some work is being carried out on PG1.5 to make it more efficient; it is being tuned for interpretation by simplifying and normalizing conditions in the realization rules. This involves 'tightening up' the conditions by substituting **xor** for **or** wherever possible and reducing the scope of any valid disjunctions that remain (e.g. **a and (b or c)** rather than **a and b or a and c**).

Clearly efficiency is a problem, since the AO trees explode into or-nodes, through the use of (i) disjunctive entry conditions in the system network and (ii) disjunctive conditions in the realization rules. It has been proved (Brew [3]) that systemic classification is NP-Hard and is thus inherently intractable. This led to the choice of partial satisfiability rather than exact satisfiability (which itself is NP-Hard) in REVELATION1, and the development of an efficient technique for searching the AO trees which utilizes incremental evaluation and pruning to reduce the number of backtracking points (full details in O'Donoghue [19]). The delayed use of exact satisfiability being investigated in REVELATION2 is similar to Brew's 'partial' algorithm for checking systemic descriptions. Brew's checking algorithm has two stages, the first being a simplification stage in which all disjunctive entry conditions are eliminated from the system network by replacing them with a uniquely generated feature. The resultant simplified network can then be searched efficiently. The second stage involves checking all of the features generated in the first stage, each generated feature referring to a disjunctive entry condition. Here also we find a delaying tactic in which disjunctions are satisfied as late as possible in the search.

Although no theoretical calculations as to the complexity of the REVELATION1 method and PG1.5 have been undertaken, we can get a feel for the scale of the problem by considering the amount of CPU time that was needed to obtain the semantic representation of Figure 11. On a SPARCstation 1 with 16M this task required ≈25 CPU seconds, with this time being halved on a Sun 4/490 with 32M. When reading these timings bear in mind that REVELATION1 is coded in POPLOG languages which are incrementally compiled into an interpreted virtual machine code.

Conclusions

REVELATION1 has demonstrated that Fawcett's SG is a bi-directional formalism, although in the case of PG1.5, some reorganization was required to make it run in reverse. The main problem in reversing a SG seems to be that systemic grammars are written from a predominantly text-generation viewpoint. In developing their grammars systemic grammarians are concerned with how the grammar will generate rather than its suitability for interpretation. For instance, special care ought to be taken when expressing conditions in realization rules: writing **a xor b** rather than **a or b** can be a godsend to an interpreter. Similarly, simplifying and normalizing conditions so that they are as simple and specific as possible is a great aid to interpretation. It reduces the search space and hence speeds up interpretation — even though, from a generative point of view, it may make more sense to express a condition in a 'long winded' fashion that captures the generalization the linguist is attempting to make.

Fawcett states (private communication) that in developing his version of SG (which is different in a number of ways — especially in the realization component — from the NIGEL grammar of Mann and Matthiessen) he always had in mind its potential for reversibility. Perhaps the surprising thing is not that modifications in GENESYS are indicated by work on REVELATION, but how *few* modifications appear to be required. Clearly, the need now is for close collaboration between the builders of the successor versions of GENESYS and and the successor versions of REVELATION. Current research has precisely this goal; and we shall report the results in due course.

REVELATION1 combined with Fawcett's SG appears to be a step in the right direction towards a bi-directional systemic grammar. REVELATION2 may take a step closer. But a bi-directional systemic grammar will only be achieved when interpretation-minded people and generation-minded people get together and collaborate in developing such a grammar.

Acknowledgements

I would like to thank Eric Atwell and Clive Souter (Leeds University) for their valuable contributions to earlier drafts of this paper and Robin Fawcett (UWCC) for his comments and suggestions on the final version.

References

[1] Rosalind Barrett, Allan Ramsay, and Aaron Sloman. *POP11: a Practical Language for Artificial Intelligence.* Ellis Horwood, Chichester, 1985.

[2] James D. Benson and William S. Greaves, editors. *Systemic Perspectives on Discourse: selected papers from the 9th International Systemic Workshop.* Ablex, London, 1985.

[3] Chris Brew. "Partial Descriptions and Systemic Grammar". In *Proceedings of the 13th International Conference on Computational Linguistics (COLING '90)*, 1990.

[4] Christopher S. Butler. *Systemic Linguistics: Theory and Applications.* Batsford, London, 1985.

[5] William F. Clocksin and Chris S. Mellish. *Programming in Prolog.* Springer Verlag, 3rd edition, 1987.

[6] Anthony Davey. *Discourse Production: A Computer Model of Some Aspects of a Speaker.* Edinburgh University Press, 1978.

[7] David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors. *Natural Language Parsing: psychological, computational and theoretical perspectives.* Studies in Natural Language Processing. Cambridge University Press, 1985.

[8] Robin P. Fawcett. *Cognitive Linguistics and Social Interaction*, volume 5 of *Exeter Linguistic Studies.* Julius Groos Verlag, Heidlberg, 1980.

[9] Robin P. Fawcett. "Language Generation as Choice in Social Interaction". In Zock and Subah [24], chapter 2, pages 27–49.

[10] Robin P. Fawcett and Gordon H. Tucker. "Demonstration of GENESYS: a very large semantically based Systemic Functional Grammar". In *Proceedings of the 13th International Conference on Computational Linguistics (COLING '90)*, 1990.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[12] John Gibson. "AI Programming Environments and the POPLOG System". In Yazdani [23], chapter 2, pages 35–47.

[13] Robert T. Kasper. "An Experimental Parser for Systemic Grammars". In *Proceedings of the 12th International Conference on Computational Linguistics (COLING '88)*, August 1988. Also available as USC/Information Sciences Institute Reprint RS-88-212.

[14] Martin Kay. "Parsing in Functional Unification Grammar". In Dowty et al. [7], chapter 7, pages 251–278.

[15] Stephen C. Kleene. *Introduction to Metamathematics.* North-Holland, 1952.

[16] William C. Mann and Christian M. I. M. Matthiessen. "NIGEL: A Systemic Grammar for Text Generation". In Benson and Greaves [2]. Also available as USC/Information Sciences Institute Reprint RS-83-105.

[17] Christian M. I. M. Matthiessen and John A. Bateman. *Text Generation and Systemic-Functional Linguistics: experiences from English and Japanese.* Pinter, London, 1991. (in press).

[18] Nils Nilsson. *Principles of Artificial Intelligence.* Springer Verlag, 1983.

[19] Tim F. O'Donoghue. "The REVELATION1 Semantic Interpreter". COMMUNAL Report 22, School of Computer Studies, University of Leeds, 1991.

[20] Tim F. O'Donoghue. "The Vertical Strip Parser: a lazy approach to parsing". Report 91.15, School of Computer Studies, University of Leeds, 1991.

[21] J. R. Slagle. "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus". In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 191–203. McGraw-Hill, New York, 1963.

[22] Terry Winograd. *Language as a Cognitive Process, Volume 1: Syntax.* Addison Wesley, 1983.

[23] Masoud Yazdani, editor. *Artificial Intelligence: principles and applications.* Chapman Hall, 1986.

[24] Michael Zock and Gerard Subah, editors. *Advances in Natural Language Generation (Volume 2).* Pinter, London, 1988.