Reducing Complexity in A Systemic Parser

Michael O'Donnell

Department of Linguistics,
University of Sydney
email: mick@isi.edu

Abstract

Parsing with a large systemic grammar brings one face-to-face with the problem of unification with disjunctive descriptions. This paper outlines some techniques which we employed in a systemic parser to reduce the average-case complexity of such unification.

1 Introduction

Systemic grammar has been used in several text generation systems, such as PENMAN (Mann — Matthiessen 1985), PROTEUS (Davey, 1978), SLANG (Patten, 1986), GENESYS (Fawcett — Tucker, 1990) and HORACE (Cross, 1991). Systemics has proved useful in generation for several reasons: the orientation of Systemics towards representing language as a system of choices, the strongly semantic nature of the grammar, and the extensive body of systemic work linking discourse patterns and grammatical realisation (e.g., Halliday, 1985; Halliday — Hasan, 1976; Martin, 1992).

Parsing with systemic grammar has not, however, been as successful. To date, there have been six parsing systems using systemic grammar: Winograd (1972), McCord (1977), Cummings — Regina (1985), Kasper (1988a, 1988b, 1989), O'Donoghue (1991a, 1991b) and Bateman *et al.* (1992). However, each of these systems has been limited in some way, either resorting to a simplified formalism (Winograd — Cummings — McCord), or augmenting the systemic analysis by initial segmentation of the text using another grammar formalism (Kasper: Phrase Structure Grammar; Bateman *et al.*: Head-driven Phrase Structure Grammar; O'Donoghue: his 'Vertical Strip Grammar' (VSG)). There has not so far been a parser that parses using the full systemic formalism, without help from another formalism.

The reasons for this failure relate to those reasons which favour generation. Firstly, the orientation of systemic grammar towards choice means that the grammar is organised into a form full of disjunctions, which leads to complexity problems in parsing. Secondly, the strongly semantic content of systemic grammars (including roles such as Actor, Process and Circumstance in the grammar) leads to a structural richness which adds to the logical complexity of the task.

One result of the work in Systemic generation has been the availability of a large computational generation grammar using the systemic formalism – the Nigel grammar (Matthiessen — Mann, 1985, Matthiessen — Bateman, 1992). As this resource is available, it is desirable to use it for parsing. However, complexity problems have so far made this impossible, except by pre-parsing with another formalism.

In the last few years, we have developed a parser for Systemic grammar, particularly for use with the Nigel grammar. The parser handles the full Systemic formalism, and does not depend on another formalism for segmentation. The parser uses a bottom-up, breadth-first algorithm. A chart is used to handle some of the non-determinism.

This paper focuses on some methods we have used in the parser to reduce the complexity problems associated with using the Nigel grammar. In particular, we focus on the means used to make disjunctive unification more efficient.

Section 2 discusses the problem of disjunctive expansion, and some means of making it

more efficient at a general level. Before becoming more specific, the Systemic formalism is introduced (section 3). Section 4 explores one method of avoiding complexity – reducing the size of the disjunctive description by working with sub-descriptions rather than the whole description. Section 5 presents three ways of making expansion, when necessary, more efficient. We conclude the paper with a brief summarisation of our work.

2 Unification with Disjunctive Descriptions

Parsing with a systemic grammar involves much unification of disjunctive descriptions. The usual way to unify such is as follows:

1. Expand out the disjunctive descriptions to Disjunctive Normal Form (DNF) – a form with all disjunction at the top level of the description – a disjunction of non-disjunctive forms.
2. Unify each term of the first DNF form with each term of the other.

DNF expansion of a description is however an expensive task – the process takes exponential time in the worst case (Kasper — Rounds, 1986). Space is also a problem – DNF expansion is a transformation whereby a disjunctive description is replaced with a set of descriptions each of which contains no disjunction. For a description containing a high level of disjunction, the size of the DNF form can be excessive.

Space has not however been a problem in our processing, but time has. Systemic parsing is very slow. We thus focus on means for speeding up, or avoiding, the unification process.

2.1 Avoiding Expansion

There have been proposals for unification without DNF expansion. Karttunen, for instance, has proposed an algorithm which "uses constraints on disjuncts which must be checked whenever the disjunct is modified" (Kasper, 1987, p81). However, as noted by Kasper (1987, p61), Karttunen's unification algorithm works only for a limited type of disjunctive description, and not for general disjunction as is needed in the present work.

Kasper has proposed a method of re-representing disjunctive descriptions which in some cases avoids the need for expansion. His approach separates a disjunctive description into two parts – a *definite* component (which contains no disjunction), and an *indefinite* component (containing the disjunctive information of the description). A unification process can first check whether the definite components of two descriptions unify, and only proceeds to unify the indefinite components if the definite components unify successfully. The unification of the indefinites is avoided if the unification of the definites fails.

2.2 Delaying disjunctive expansion until necessary

The Kasper-Rounds form also allows us to delay expansion until a later time. When two descriptions are unified, only the definite components need to be checked for compatibility. The result of a Kasper-Rounds unification contains the indefinite descriptions from both descriptions without expansion. At some point in the processing it may be necessary to resolve the indefiniteness, and the disjunctive components are then expanded. However, in many cases, the definite component of the description may become inconsistent before this is necessary, expansion is thus avoided.

2.3 When expansion is necessary, expand efficiently

If DNF-expansion is required, then it should be performed as efficiently as possible. We here discuss some methods to achieve this goal:

1. **Reducing the disjunctiveness of the description**: By reducing the extent of the description, we reduce the amount of disjunction to be expanded, and thus speed up the expansion process. We use two methods to reduce the size of descriptions:
   (a) Extracting descriptions for special-purpose: we segment the grammar description into sub-descriptions for particular purposes. We found that different parsing processes drew upon

only subsets of the grammar. Rather than working with the full grammar, sub-descriptions tailored for particular purposes can be compiled-out. These sub-descriptions are less complex to expand than the full description

(b) Register Specific Pruning: parts of the grammar which are not expected to be used in a particular set of target texts are 'pruned-out' before processing begins.

2. **Expanding Disjunctions Efficiently**: a disjunctive description may contain a number of disjunctions. Ordering the expansion of these disjunctions in particular ways can result in improved expansion times:

   (a) Multiplying together disjunctions with high likelihood of inconsistency first, thus reducing the number of terms which we continue with.

   (b) Spotting inconsistent unifications with minimum of work e.g., checking for inconsistencies between single terms before checking for inconsistencies between combinations of terms.

   (c) Using some form of structure sharing in the expansion process: in the expansion process, the same terms may be multiplied together a number of times. A form of structure-sharing, such as a parse chart, can reduce the redundancy in the expansion process.

2.4 Caching and precompilation: avoiding repeating the same expansion.

The parser makes extensive use of caching – when any expansion is calculated which is likely to be used again, the result is stored away for later reuse.

Precompilation has also been a useful technique to improve parsing efficiency. Precompilation is basically a pre-caching of all the values which might be used in the parsing process. By performing most of the DNF expansion of the grammar as a precompilation step, we avoid doing that calculation during the parsing of a sentence.

3 A Systemic Grammar

3.1 Type and Role Logic

Systemic grammar, in distinction to value-attribute grammars, distinguishes *type logic* (the classes of units) and *role logic* (the constituency and dependency relations between units). The type logic is expressed in a network, called a *system network*. The role logic is expressed as a set of constraints on the types of the grammar.

3.2 System Networks

Systemic grammar (e.g., Halliday, 1985, Hudson, 1971, Matthiessen — Mann, 1985) uses an inheritance network to organise grammatical types (or 'feature' in Systemics[1]), and their structural consequences. A Systemic inheritance network is called a *system network*.

A system network is used to organise the co-occurrence potential of grammatical types, showing which types are mutually compatible, and which are incompatible. It consists of a set of *systems*, which are sets of mutually exclusive types. There is also a *covering* relation between the types of a system, meaning that if the entry condition of the system is satisfied, then one of the types in the cover must be selected.

Figure 1 shows a system network for a simple grammar of English. It includes 11 systems, representing various grammatical distinctions, for instance, between clause and word, between transitive and intransitive clauses, or between nominative and accusative pronouns.

Each type inherits the properties of types to its left in the network. Note that the system network may be logically complex, since entry conditions (the logical condition on a system) may consist of conjunctions and disjunctions of types.

[1]Note that the term 'feature' is used distinctly from its use in most unification paradigms. In Systemics, a feature is what Functional Unification Grammar would call a value, e.g., active, transitive and noun are features.

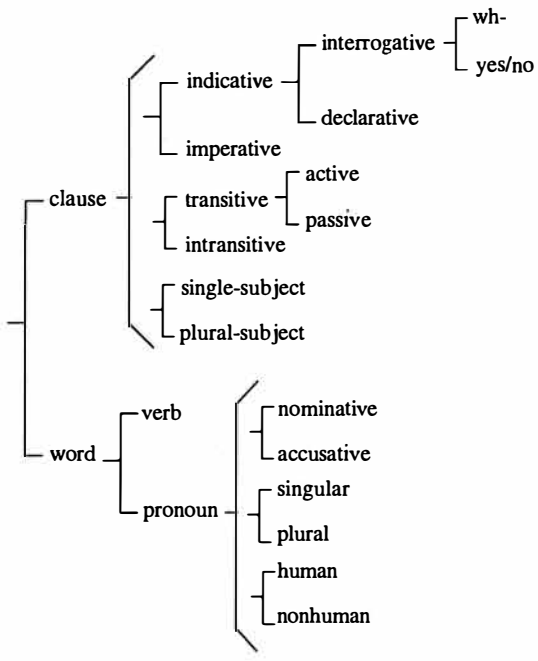

Figure 1: A partial Systemic network

3.3 Structural Templates

Types of the system network are associated with structural realisations – the structural consequence of the type. Figure 2 shows the realisations of the types in Figure 1.

```
clause:          Subject: nominative
                 Actor: human
                 Finite: finiteverb
                 Pred: lexical-verb
declarative:     Subject^Finite
yes-no:          Finite^Subject
transitive:      Object: accusative
                 Actee = [ ]
                 Pred...Object
active:          Subject/Actor
                 Object/Actee
                 Finite/Pred
passive:         Subject/Actee
                 Object/Actor
                 Pass: be-aux
                 AgentM = "by"
                 Finite/Pass
                 Pred: en-verb
                 Pass^Pred
                 AgentM^Object
intransitive:    Subject/Actor
                 Finite/Pred
single-subj:     Subject: singular
plural-subj:     Subject: plural
```

Figure 2: Realisation Rules

This grammar deals mainly with some systems involving the Subject and Object, what types of units fill these roles, and how these roles conflate with two other roles: Actor and Actee. The grammar assumes that both roles are filled by pronouns, which are either [nominative] or [accusative], [singular] or [plural], and [human] (e.g., "I", "you", "he") or [nonhuman] (e.g., "it", "that"). Only [human] pronouns can fill the Actor role of a clause.

The realisation operators used in the formalism are as follows:

Insert e.g., *Finite = [ ]*: indicates that the function *Finite* must be present in the structure.

Conflate e.g., *Modal/Finite*: indicates that the two functions *Modal* and *Finite* are filled by the same grammatical unit.

Order e.g., *Subject ^ Finite*: indicates the sequencing of functions in the surface structure. In this example, the *Subject* is sequenced directly before the *Finite*. Any number of elements can be sequenced in a single rule.

Partition e.g., *Thing... Event... End*: Another sequence operator, specifies that the appear in this order, but not necessarily immediately adjacent (linear precedence).

Preselect e.g., *Subject: nominal-group*: indicates that the *Subject* element must be filled by a unit of type *nominal-group*.

Lexify e.g., *Deict = "the"*: used to assign lexical items directly to elements of structure. Note that lexify overrides any preselect which may apply to the same element of structure.

3.4 Logical Expression of the Grammar

For the purposes of the expansion of this grammar, we re-express it in a logical formalism. Figure 3 shows Logical Form I of this grammar, including the structural constraints embedded in the form. Note that :xor indicates exclusive disjunction.

```
(:xor
 (:and clause
       Subject: nominative
```

```
      Actor: human
      Finite: finite-verb
      Pred: lexical-verb
      (:xor
        (:and declarative
              Subject^Finite)
        (:and yes-no
              Finite^Subject))
      (:xor
        (:and transitive
              Object: accusative
              Actee = [ ]
              Pred...Object
              (:xor
                (:and active
                      Subject/Actor
                      Object/Actee
                      Finite/Pred
                      (:and passive
                            Subject/Actee
                            Object/Actor
                            Pass: be-aux
                            AgentM= "by"
                            Pred: en-verb
                            Finite/Pass
                            Pass^Pred
                            AgentM^Object))
        (:and intransitive
              Subject/Actor
              Fin/Pred))
      (:xor (:and single-subject
                  Subject: singular))
            (:and plural-subject
                  Subject: plural))))
(:and word
      (:xor (:and pronoun
                  (:xor nominative
                        accusative)
                  (:xor singular
                        plural)
                  (:xor human
                        nonhuman))
            (:and verb ... ))))
```

Figure 3: Logical Form I of the Grammar

4 Extracting Sub-Grammars for particular Parsing Tasks

Rather than expanding out the whole grammar, it is more efficient to extract out subsets of the grammar, to be used for particular tasks in parsing. In our systemic parser, the description is used for three purposes:

1. Path Unification: checking that two type-paths can unify,

2. Predicting What Comes Next: seeing which function-bundle(s) can come next in the structure e.g., we have just analysed Subject/Actor ^ Fin/Mod, and want to predict what function-bundle can occur next in the structure.

3. Function-Bundle Assignment: seeing what function-bundle a given constituent can fill, e.g., we have just parsed a nominal group, and want to see what function-bundles it can be the filler of.

Each of these uses makes only partial use of the grammar description. Thus, rather than expanding out the entire grammar, we can simplify the process by extracting out sub-grammars, one for each of these applications. Since the size of each sub-grammar is smaller, the complexity problem is reduced. This section looks at these three sub-descriptions in more detail.

4.1 Separating Type Logic from Role Information

It has proved useful to separate the type logic component of the grammar from the role logic. The two logic components have different patterns of use – type logic is used to test whether two partial type-paths can unify. We never try to unify a partial type description with the type grammar as a whole. The type-logic component of the grammar thus does not need to be DNF-expanded.

The role logic, on the other hand, does need to be expanded. We expand the role-logic component to produce a set of non-disjunctive structure rules which can be applied during parsing (sometimes termed 'chunking').

```
(:and
 ;1. Type Logic Component
 (:xor (:and clause
             (:xor declarative yes-no)
             (:xor (:and transitive  (:xor active passive))
                   intransitive)
             (:xor single-subject plural-subject))
       (:and word
             (:xor (:and pronoun (:xor nominative accusative)
                                 (:xor singular plural)
                                 (:xor human nonhuman))
                   (:and verb ... ))))

 ;2. Role Logic Component
  (:and  (:implies clause (:and Subject: nominative
                                Actor: human
                                Finite: finite-verb
                                Pred: lexical-verb))
         (:implies declarative Subject^Finite)
         (:implies yes-no  Finite^Subject)
         (:implies transitive  (:and Object: accusative
                                     Actee: [ ]
                                     Pred...Object))
         (:implies active (:and Subject/Actor
                                Object/Actee
                                Finite/Pred))
         (:implies passive (:and Subject/Actee
                                 Object/Actor
                                 Pass: be-aux
                                 AgentM= "by"
                                 Pred: en-verb
                                 Finite/Pass
                                 Pass^Pred)
                                 AgentM^Object))
         (:implies intransitive  (:and Subject/Actor
                                       Fin/Pred))
         (:implies single-subject Subject: singular)
         (:implies plural-subject Subject: plural)))
```

Figure 4: Logical Form II of the Grammar

Thse two components of the description have different properties: type logic is acyclic, while role logic is potentially cyclic. Type logic is constrained such that types are always in disjoint coverings (which allows efficient negation), while role logic doesn't have this constraint.

Because of these differences in properties and uses, it has proved efficient to treat these two logics separately. Logical Form I of the systemic grammar provided in Figure 3 can be re-represented in the equivalent Logical form II shown in Figure 4, separating out the type and role logic.

4.1.1 Unification of Type Descriptions

The parser uses the type-logic component of this grammar without fully expanding it. Partial expansion, however, is performed, whereby the *type-path* (the logical-entailment of a system, i.e., the logical expression of types leading back to the

root of the network)[2] is pre-compiled for each system.[3] The negation of each type in the system is also pre-compiled, which speeds up unification involving negation of types.

Type-paths are represented in the form proposed by Kasper (1987), and his unification algorithm is used when two type-paths are unified. The main use of the type-logic component is checking the compatibility of two types or type-paths.

Type logic has thus been simplified using three strategies:

1. Separating from Role Logic
2. Using Kasper's 'delayed expansion' technique.
3. Precompiling each system's logical entailment, and the negation of types.

Because of these methods, unification of type-paths using even quite complex grammars operates quite quickly.

4.2 Function Assignment

Another use made of the grammatical description in parsing is to assign a set of structural roles to a unit. The set of roles a unit fills is called in Systemics the *function-bundle* of the unit. The systemic formalism allows each unit to be assigned multiple functions. For instance, using the NIGEL grammar, 'the cat' in "the cat scratched the woman" would be assigned the function-bundle Subject/Agent/Actor/Theme. The possibility of a unit serving multiple functions is a major source of complexity in systemic parsing.

Assigning function-bundles to a unit is one of the tasks in systemic parsing. For instance, assume we have just parsed a pronoun "he", assigning it a type-path:

```
(:and word:pronoun:nominative:human:singular)
```

Now, we wish to find what function-bundles the pronoun can serve at a higher level. One result could be:

```
                          [clause:transitive]
[pronoun]                   ______|___...
    |          =>                 |
  "he"                      Subject/Actor
                              [pronoun]
```

This process draws upon three parts of our grammar:

- Preselection and Lexify rules: used to discover what functions different units can fill.
- Conflation rules: used to discover which functions a unit can serve simultaneously, and thus, which of the preselection and lexify rules can combine.
- The Type Logic: to show which of these preselection, lexify and conflation rules are systemically compatible.

Since we have already set up the type-logic for path unification, we can draw upon that resource as needed. We do not need to include the type-logic in the sub-description for the function-assignment process.

4.2.1 Extracting the relevant description

For the function-assignment process, we do not need all of the role logic description. We can select out only those rules involving preselection, lexify, and conflation. See Logical Form III in Figure 5.

```
(:and (:implies clause
         (:and Subject: nominative
               Actor: human
               Finite: finite-verb
               Pred: lexical-verb))
      (:implies transitive
         Object: accusative)
      (:implies active
         (:and Subject/Actor
               Object/Actee
               Finite/Pred))
      (:implies passive
         (:and Subject/Actee
               Object/Actor
               Pass: be-aux
```

[2]Note that since entry conditions of systems can be logically complex, the path itself can contain disjunctions and conjunctions.

[3]Paths are stored with systems rather than types, since the path of all types in a system are identical.

```
            AgentM= "by"
            Pred: en-verb
            Finite/Pass))
    (:implies intransitive
       (:and Subject/Actor
             Fin/Pred))
    (:implies single-subject
          Subject: singular)
    (:implies plural-subject
          Subject: plural)))
```

Figure 5: Logical Form III: The Function Assignment Sub-Description

4.2.2 Implications Out

We next put this description into a form more suitable for DNF-expansion. Note that implication can be re-expressed using disjunction, conjunction and negation:

```
(:implies a b)  is-equivalent-to
       (:xor (:and a b) (:not a))
```

Using this rule, we can re-express the logical form III as Logical Form IV, as shown in Figure 6.

```
(:and (:xor (:and clause
                  Subject: nominative
                  Actor: human
                  Finite: finite-verb
                  Pred: lexical-verb)
            (:not clause))
      (:xor (:and transitive
                  Object: accusative)
            (:not transitive))
      (:xor (:and active
                  Subject/Actor
                  Object/Actee
                  Finite/Pred))
            (:not active))
      (:xor (:and passive
                  Subject/Actee
                  Object/Actor
                  Pass: be-aux
                  AgentM: "by"
                  Pred: en-verb
                  Finite/Pass)
            (:not passive))
      (:xor (:and intransitive
                  Subject/Actor
                  Fin/Pred)
            (:not intransitive))
      (:xor (:and single-subject
                  Subject: singular)
            (:not single-subject))
      (:xor (:and plural-subject
                  Subject: plural)))
            (:not plural-subject)))
```

Figure 6: Logical Form Form IV: The Function Assignment

4.2.3 Expansion to DNF

Simple algorithms exist to expand Logical Form IV into DNF (see section 5.1). A small part of the result appears in Logical Form V of the grammar, shown in Figure 7.

```
(:xor
  (:and clause transitive active
        single-subject
        Subject/Actor: (:and nominative
                             human singular)
        Object/Actee: accusative
        Finite/Pred: (:and verb finite-verb
                           lexical-verb))
  (:and clause transitive
        active plural-subject
        Subject/Actor: (:and nominative
                             human plural)
        Object'Actee: accusative
        Finite/Pred: (:and verb finite-verb
                           lexical-verb))
  etc...
```

Figure 7: Logical Form Form V: The Function Assignment Sub-Description in DNF

The order of worst-case complexity of the expansion to DNF is easily calculated – it is simply two to the power of the number of disjunctions, which is equal to the number of types which have realisation rules of type conflation, insertion, or preselection.

By opting to expand only subsets of the whole grammar, we have reduced the complexity of the description, since the size of n for this sub-description is smaller than for the whole description. However, for a real-sized grammar such as NIGEL, the size of n is still large.

4.2.4 Re-expression in terms of Function Bundles

From the DNF-form of this description, we can extract out partial-descriptions for each function bundle. We now re-express this logical form in

terms of the type constraints on each function-bundle, including both the constraint on the type of unit the function-bundle can be part of (the 'parent-constraint'), and the constraint on the filler of the function-bundle (the 'filler-constraint'). We show this as a set of triplets, of the form:

```
( <parent-types>
  <function-bundle>
  <child-types> )
```

```
1. ( (:and clause transitive
           active single-subject)
     Subject/Actor
     (:and nominative human singular)))

2. ( (:and clause transitive
           active single-subject)
     Object/Actee
     accusative)))

3. ( (:and clause transitive
           active plural-subject)
     Subject/Actor
     (:and nominative human plural)

4. ( (:and clause transitive
           active plural-subject)
     Object/Actee
     accusative)

5.  ( (:and clause transitive
            active singular-subject
      Finite/Pred
      (:and verb finite-verb lexical-verb))
etc....
```

This representation can now be used to assign function-bundles A unit can take on a function-bundle if it can unify with the filler-constraint on the function-bundle.

For the instance we started with, "he", with types: `(:and pronoun nominative human singular)`, only one triplet would unify. We could thus posit structure for our unit:

```
[clause:transitive:active:single-subject]
    ..----------|----------...
                |
         Subject/Actor
                |
 [ pronoun:nominative:human:singular ]
                |
              "he"
```

Note that we have also gained information about the types of the parent-unit of which the unit is a constituent.

4.2.5 Reducing the number of Rules

Note that there is another simplification we can make to the triplet list. We can take all triplets with identical function bundle and child-type specification, and join them. The parent-types slot is replaced with the disjunction of the two parent-type slots. Thus, elements 2 and 4 above become a single item. This process reduces the number of rules to apply:

```
2,4. ( (:and clause transitive active)
       Object/Actee
       accusative)))
```

4.3 Predicting What Comes Next

Another process we use in parsing involves the prediction of what function-bundles can come next in a partially completed structure. With a systemic grammar, this process requires:

- Ordering and Partition rules: to see which function can come next.
- Conflation rules: to see which functions can conflate with the function predicted to come next.
- The type logic: to show which of these ordering, partition and conflation rules are systemically compatible.

The processing of this sub-description, and any others, is exactly the same as for function-assignment.

1. Extract from the role logic description the relevant realisation rules;
2. Replace implications with disjunction and negation;
3. Expand out the grammar;
4. Index the rules in a form useful for the processing.

4.4 Register Restriction

Another means of reducing the overall complexity of the descriptions involves eliminating from the grammar parts which are unlikely to be utilised in the target texts. In systemic terms, we apply register restrictions to the grammar.

For example, in a domain of computer manuals, the description of interrogative structures is not likely to be drawn upon.[4] By eliminating this sub-description, we reduce the degree of disjunction in the whole description, and thus speed up the parsing of the forms which do appear in the text.

The method of deriving the register-restrictions was as follows:

1. We parsed by hand[5] a chapter of the computer manuals we were attempting to parse, building up a register-profile of our target texts.

2. An automatic procedure then extracted out all the grammatical types which occur in these sentences.

3. The process used this information to discover the types *not* occurring in the sample.

4. The process then eliminated these types and their realisations from the description.

We were thus left with a restricted grammar which was capable of parsing the sentences in the sample, and also parsing many which were not in the sample (under the assumption that the grammatical forms in the sample were representative of the forms found in the manual as a whole). We reduced the size of the grammar by approximately 60% using this method.

4.5 Summary

By extracting out sub-descriptions from the full description, we reduce the complexity of the description-to-be-expanded.

5 Improving the Efficiency of Expansion

Section 4 has proposed techniques which reduce the size of the description which needs to be expanded. However, for large-sized descriptions, the expansion is still complex. This section briefly explores two methods which increase the efficiency of the expansion process. If we can't avoid full expansion, then at least we can make the expansion process more efficient.

5.1 "Structure Sharing" in Expansion

This section assumes a disjunctive description of the following form:

```
(:and (:xor A B) (:xor C D) (:xor E F) )
```

Logical form V introduced above was of this form. Much of the pre-processing in the parser involves the DNF-expansion of disjunctions in this form.

5.1.1 Full Expansion

The brute force method for expanding this form involves:

1. Find all combinations of terms, taking one term from each disjunction.

2. Test compatibility of each combination, eliminating combinations which are internally inconsistent.

Step 1 of this process produces the following DNF form:

```
(:xor (:and A C E) (:and A C F)
      (:and A D E) (:and A D F)
      (:and B C E) (:and B C F)
      (:and B D E) (:and B D F) )
```

[4]Note that some of the forms we restrict through register restriction may actually appear in any one text, although quite rarely. We are trading off between speed for the majority of sentences, and ability to parse all sentences in a text.

[5]The hand-parsing is really computer-assisted, – a tool was developed to traverse the system network for each sentence (and each constituent of the sentence) asking the human which feature was appropriate for the target string. This process guaranteed that the human-analysis conformed to the computer grammar.

The problem with this approach is with the incompatibility checking – the same checks will be repeated over and over again. For instance, the incompatibility check between A and C is repeated twice: (:and A C E) and (:and A C F). This repetition occurs for every pair of terms in the conjuncts. The problem gets worse exponentially as we add more disjuncts.

To avoid this redundancy, we need something like a chart in parsing, a method to record the results of each unification and thus avoid repeating any unification.

Unfortunately, DNF expansion is not quite like parsing. We can test the consistency between any two pairs of terms (for instance A and C in the above), but we also need to know about the consistency of terms in combination e.g., the pairs: A&C, A&E and C&E may be consistent, but the combination A&C&E may not be.

The rest of this section describes two techniques which allow some redundancy reduction, sometimes known as structure-sharing.

5.1.2 Tree Organisation of Expansion

The disjunctive description above can easily be re-represented in the form below:

```
(:and (:and (:xor A B)
            (:xor C D))
      (:xor E F))
```

The process here involves expanding out the first two disjunctions, eliminating inconsistent results, and then expanding the result out with the next disjunction. The incremental expansion is illustrated in Figure 8.

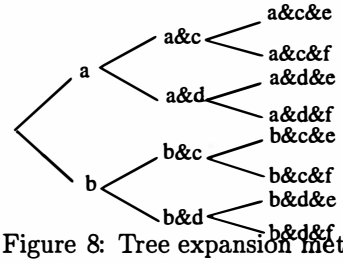

Figure 8: Tree expansion method

This method is more efficient than the full expansion method, since:

- Some terms, such as a&c, a&d etc. are unified only once. However note that terms e and f are still involved in multiple products.
- the failure of a combination of terms early in the unification process eliminates a large number of expansions by the end of the process.

5.1.3 Binary Organisation of Expansion

A third approach aims at maximising the degree of 'sharing' unifications in the expansion. The disjunctions in the description are split into pairs, and unified. The results of these unifications are then unified in the same pair-wise manner. This expansion for a conjunction of four disjunctions is shown in Figure 9.

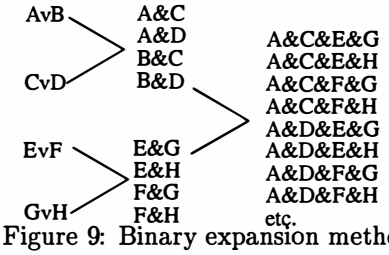

Figure 9: Binary expansion method

The advantage of this approach is that we are maximising the amount of structure-sharing in the unification.

5.1.4 Comparison of Expansion Approaches

We compared the number of unifications which take place using each of these methods for various numbers of disjunctions (all disjunctions having two disjuncts).

One can see from Table 1 that the worst-case score for the full expansion method is far worse than the other methods. It is not a practical method.

Comparing the worst-case for the 'tree' and the 'binary' expansion method, we see that the binary method clearly comes out better, by around 50%.

We also did a simulation to check an average case score, since the worst-case score doesn't take into account that many later unifications are avoided when early unification proves inconsistent. We found that while the binary method still seems superior, in some instances the tree method requires fewer unifications. More work is needed here.

N	Full	Tree	Binary
11	20480	4092	2364
12	45056	8188	4424
13	98304	16380	8448
14	212992	32764	16780
15	458752	65532	33236
16	983040	131068	66144
17	2097152	262140	133528
18	4456448	524284	266660
19	9437184	1048572	526956
20	19922944	2097148	1053304

Table 1: Worst-case comparison

5.2 Ordering Incompatible Disjunctions First

When using either the Tree or Binary expansion methods, fewer unifications will be required if we place the disjunctions with the greatest chance of inconsistency first. In a sense, we are pruning inconsistent branches of the expansion tree 'at the root'.

In the systemic parser, several heuristics have been used to group disjunctions which are most likely to produce the fewest cross-products, and perform these first.

One possible method for utilising this phenomenon is:

1. Separate the disjunctions into sub-sets which maximise likelihood of incompatibility between rules inside the sub-expressions.
2. Expand out the disjunctions inside each sub-set. The results of each sub-set are cached so they need only be expanded once.
3. Expand out the results of (2) against each other.

5.3 Avoiding Expansion of Incompatible Terms

Sometimes, it is possible to tell without full unification that a set of rules will not unify with another set. For instance, assume a larger grammar than the one we have been using, a grammar which includes clauses, nominal-groups[6], prepositional phrases, adverbial phrases and words. These categories are all types in the system network, just like any other types.

Since these types won't unify with each other, we can also know that types which inherit from one of these basic types will not unify with the the sub-types of another basic type. We thus do not need to try to unify descriptions which differ in their basic type. If we split any disjunctive description into sub-components for each basic type, we know a priori that there is no unification between these sub-components.

Before trying any of the expansion techniques outlined in this paper, the whole grammar is segmented into sub-descriptions, one for each of these basic types. The complexity of the expansion of each of these sub-grammars is less than for the grammar as a whole.

Other principles can be used to locate sets of rules which will not unify. These can be applied also.

6 Conclusion

While the techniques outlined here have been applied in ways particular to a systemic grammar, and for a particular implementation, there are principles behind the re-representations which are general to all implementations:

1. Avoid DNF-expansion where possible, as in Kasper's unification algorithm.
2. Delay expansion to a later time – information gained later may show the description to be inconsistent in the definite component.
3. When expansion is necessary,
   (a) Try to extract out sub-descriptions which can be used, rather than expanding the entire grammar.
   (b) Expand out first disjunctions which are most likely to conflict, since this will reduce the total number of terms which will need to be multiplied.
   (c) Avoid expanding terms that can be known to be incompatible.

[6]Systemics prefers the term 'nominal-group' over the equivalent term 'noun-phrase'.

As a result of the application of these techniques (and others not here mentioned), we have been able to implement a parsing system which parses using a large systemic grammar.

1. We start with the Nigel grammar, as used in the Penman Generation System, slightly modified for parsing purposes.
2. This grammar is then reduced by applying register-restrictions, leaving a less complex grammar, but a grammar which still handles the bulk of the phenomena in the target texts.
3. Sub-descriptions of the grammar tailored for particular processes are then extracted, and expanded out as a precompile step, producing a set of 'chunks' which can be used in parsing. This expansion takes approximately 2 minutes using Sun Common Lisp on a Sun Sparc II.
4. The 'chunked' grammar is then used to parse sentences. On the above-mentioned platform, parsing a sentence like "A user-password is a character string consisting of a maximum of eight alpha-numeric characters." took 35 seconds to parse[7]. This parser is slow, compared to most non-systemic parsers, but is far faster than the parser would be without the methods outlined here.

Future work will attempt to reduce this parsing time. Three directions are being followed:

- Streamlining the parsing process to further reduce the parsing time.
- Moving more processing to the pre-compilation stage.
- Reducing the complexity of the description without reducing its coverage.
- Incorporating heuristics to resolve ambiguities without full expansion.

Acknowledgements

The parser discussed in this paper was partially developed in the Electronic Discourse Analyser project, funded by Fujitsu (Japan). The development was aided by discussions with the members of that project: Christian Matthiessen, John Bateman, Zeng Licheng, Guenter Plum, Arlene Harvey and Chris Nesbitt.

Thanks to Cécile Paris for profuse commenting on this paper, and teaching me Latex, and to Vibhu Mittal, who solved the trickier Latex bugs.

[7]Note that when the parser is given a less complex systemic grammar, the parsing time is under two seconds for this sentence.

References

Bateman, John — Martin Emele — Stefan Momma (1992) "The nondirectional representation of Systemic Functional Grammars and Semantics as Typed Feature Structures" in *Proceedings of COLING-92*, Volume III, Nantes, France, 916-920.

Benson, J. — W. Greaves (eds.) (1985) *Systemic Perspectives on Discourse*, Volume 1. Norwood: Ablex.

Cross, Marilyn (1991) *Choice in Text: A Systemic-Functional Approach to Computer Modelling of Variant Text Production*, Ph.D. thesis submitted June 1991, Macquarie University.

Cummings, Michael — Al Regina (1985) "A PROLOG parser-generator for Systemic analysis of Old English Nominal Groups", in Benson and Greaves, 1985.

Davey, Anthony (1978) *Discourse Production: a computer model of some aspects of a speaker*, Edinburgh: Edinburgh University Press. Published version of Ph.D. dissertation, University of Edinburgh, 1974.

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

Halliday, M. A. K. (1985) *Introduction to Functional Grammar*, London: Edward Arnold.

Halliday, M. A. K. — R. Hasan (1985) *Cohesion in English*, London: Longman.

Hudson, R.A. (1971) *English Complex Sentences*, North-Holland.

Kasper, Robert (1986) "Systemic Grammar and Functional Unification Grammar" In Benson, J. and Greaves, W., *Selected Papers from the 12th International Systemics Workshop*, Norwood, N.J: Ablex.

Kasper, Robert (1987a) *Feature Structures: A logical Theory with Application to Language Analysis*, PH.D. dissertation, University of Michigan

Kasper, Robert (1987b) "A Unification Method for Disjunctive Feature Descriptions" in *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, held July 6-9, 1987 Stanford, California.

Kasper, Robert (1988a) "An Experimental Parser for Systemic Grammars", *Proceedings of the 12th Int. Conf. on Computational Linguistics*, Budapest: Association for Computational Linguistics.

Kasper, Robert (1988b) "Parsing with Systemic Grammar", Mimeo.

Kasper, Robert (1989) "Unification and Classification: An Experiment in Information-Based Parsing" In *Proceedings of the International Workshop on Parsing Technologies*, pages 1-7, CMU, Pittsburgh.

Kasper, Robert (1990) "Performing Integrated Syntactic and Semantic Parsing Using Classification" paper presented at Darpa Workshop on Speech and NL Processing, Pittsburgh, June 1990.

Kay, Martin (1979) "Functional Grammar" in *Proceedings of the Fourth Annual Meeting of the Berkeley Linguistics Society*.

Kay, Martin (1985) "Parsing In Functional Unification Grammar" in Dowty D., L. Karttunen , and A. Zwicky, (Eds): *Natural Language Parsing*, Cambridge University Press, Cambridge, England.

Mann, W. C. and C. I. M. Matthiessen (1985) "Demonstration of the Nigel Text Generation Computer Program". in Benson and Greaves, 1985

Martin, James (1992) *English Text: System and Structure*, Amsterdam: Benjamins.

Matthiessen, C. I. M. and W. C. Mann (1985) "NIGEL: a Systemic Grammar for Text Generation" in Benson and Greaves, 1985

Matthiessen, C. I. M. and J. Bateman (1992) *Text Generation and Systemic Functional Linguistics: Experiences from English and Japanese*. London: Pinter Publishers.

McCord, Michael (1977) Procedural Systemic Grammars in *Int. J. Man-Machine Studies*, 9, 255-286, London: Academic Press.

Mellish, Chris (1988) "Implementing Systemic Classification by Unification", *Computational Linguistics*, Vol. 14, Number 1, Winter 1988.

O'Donoghue, Tim F. (1991a) "The Vertical Strip Parser: A lazy approach to parsing" Research Report 91.15, School of Computer Studies, University of Leeds, Leeds, UK.

O'Donoghue, Tim F. (1991b) "A Semantic Interpreter for Systemic Grammars" in *Proceedings of the ACL Workshop on Reversible Grammars*, University of California at Berkeley, June 1991.

Patten, Terry and Graeme Ritchie (1986) "A formal model of Systemic Grammar", paper presented at 3rd International Workshop on Language Generation, Nijmegen, August 19-23, 1986.

Patten, Terry (1986) *Interpreting Systemic Grammar as A Computational Representation: A Problem Solving Approach to Text Generation*, Ph. D. dissertation, University of Edinburgh.

Winograd, Terry (1972) *Understanding Natural Language*. New York: Academic Press.