ACHIEVING FLEXIBILITY IN UNIFICATION FORMALISMS

Lena Strömbäck[1]

Department of Computer and Information Science,
Linköping University, S-58183 Linköping, Sweden

ABSTRACT

We argue that flexibility is an important property for unification-based formalisms. By flexibility we mean the ability for the user to modify and extend the formalism according to the needs of his problem. The paper discusses some properties necessary to achieve a flexible formalism and presents the FLUF formalism as a realization of these ideas.

1 MOTIVATION

Unification based-formalisms are in common use within natural language processing and many different formalisms have been defined. PATR-II (Shieber *et al.*, 1983) is the most basic and a kind of common denominator which other formalisms are extensions of. Other formalisms are STUF (Beierle *et al.*, 1988), TFS (Emele & Zajac, 1990), CUF (Dörre & Eisele, 1991) and, ALE (Carpenter, 1992). These formalisms include, for example, disjunction, various variants of negation and typing. When various grammatical theories, such as LFG (Kaplan & Bresnan, 1983) or HPSG (Pollard & Sag, 1987) are included, the range of extensions suggested to unification-based grammars becomes very wide. There are also many variant proposals on how the same extension should be used and interpreted.

When using these formalisms for a particular problem, it is often the case that the constructions provided do not correspond to the needs of your problem. It could either be the case that you want an additional construction or that you need a slight modification of an existing one. Since the extensions are numerous it seems hard to include everything in one single formalism.

In some formalisms, especially TFS and CUF, the user is allowed to define new constructions. This is an interesting property that I will develop further to achieve flexibility. In a flexible formalism, the user can define all the constructions he needs or modify definitions provided by the formalism. With this kind of formalism problems such as those mentioned above would not arise.

A flexible formalism would be a useful tool for defining various kinds of knowledge needed at different levels in a natural language system. It would be a great advantage to be able to use the same system for all levels, but adjusting it to suit the various structures that are needed at each level since the relations between the different levels would be clearer and it would be easier to share structures between the levels (cf. Seiffert (1992) for more motivation).

Another advantage with such a formalism is that it can be used to define and test extensions and various grammatical formalisms for the purpose of comparison.

Flexible formalisms allow the user to define an expensive extension and use it for the cases where he really needs it. Thus an extension that is considered too expensive to be provided by a general formalism, can be included as it can provide a more efficient representation of some particular phenomenon. This is particularly important since, in natural language processing, it is common that expensive constructions are required for few and limited cases.

However, if the user is allowed to define expensive constructions, an additional goal, predictability, must be considered. By this I mean that inefficient computations should be necessary only when the construction causing the inefficiency really participates in the computation. This idea was discussed in Strömbäck (1992) where I give a predictable algorithm for unification of feature structures containing disjunction. The goal of predictability is closely related to modularity, since if it is possible to make different constructions independent of each other, it is easier to find predictable algorithms for them. Since this paper will discuss properties of a flexible formalism rather than unification algorithms there, will be no further discussion of predictability.

In the following I first discuss the most important properties of a flexible formalism. I then present a flexible formalism, FLUF, by using it to define PATR-II. The size of this paper does not admit a thorough description of FLUF and its semantics. This is given in Strömbäck (1994a, 1994b).

1. Email address: lestr@ida.liu.se

2 ACHIEVING FLEXIBILITY

In this section I state some necessary properties of a flexible formalism.

It is essential that the formalism does not provide a single basic construction in which all other structures must be defined, as this often leads to clumsy representations. Instead the user defines everything he needs. Therefore a flexible formalism must provide some way of defining structures and objects. These can be divided into two groups; those that are used as general elements in the formalism, for example feature structures or the empty set; and those that are specific for a particular problem, for example the attribute *number*.

In addition to the elements defined in a formalism the user needs other ways of describing his objects, e.g logical operators, such as conjunction and disjunction, and functions, such as concatenation of lists. Important to note here is that these constructs do not add further elements to the defined language, they just specify additional syntax for describing the objects.

Another requirement for a flexible formalism is that the user must define the behaviour of his elements, that is, how they unify with each other. Similarly, when defining some additional syntax, he must specify the meaning of it. I have chosen to do this by stating constraint relations, which means that results from term rewriting systems (Middeltorp & Hamoen, 1992) and algebraic specification (Ehrig & Mahr, 1985) can be applied. Using constraint relations it can be specified that two defined objects should be interpreted as the same, or that one object should subsume some other object.

The last property I want to mention is the use of an inheritance hierarchy for the definitions. This is a good way of taking advantage of similarities between different extensions and also a suitable way of defining linguistic knowledge (see, for example, the articles in Computational Linguistics 18(2,3)).

By using an inheritance net for defining new constructions in the formalism several other benefits are obtained. One is that if the mathematical properties of a construction are known, this knowledge can be used instead of defining the construction with constraint relations. The inheritance net allows us to replace the construction defined with constraint relations with a predefined mathematical domain provided that both the properties of this domain and how objects unifies are known.

The inheritance net also provides ways to improve the efficiency in implementations of the system. Since a defined construction corresponds to a node in the net (together with inherited information), known unification algorithms for objects corresponding to this node can be used. This gives the design of a full-scale implementation as a library of the most common extensions, where the user can choose which he wants and define new ones or change existing definitions only when necessary.

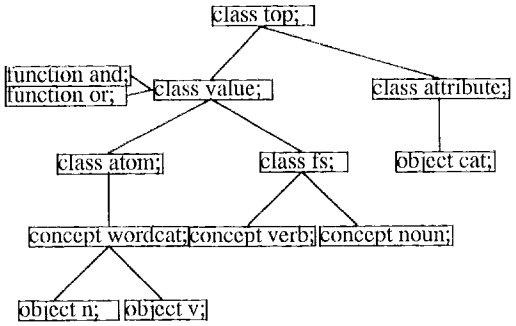

Fig. 1: A FLUF definition

3 THE FLUF FORMALISM

In this section I show how the FLUF formalism works by defining feature structures as they are used in PATR-II. I have defined conjunction and disjunction and some minimal linguistic knowledge. The inheritance hierarchy expressing the definition is shown in Fig. 1.

First the objects needed to build feature structures are defined. This is done using classes. The objects needed are attributes and values. These are the two subclasses of *top*. Values are divided into atomic and complex values, corresponding to the two classes *atom* and *fs*.

The two classes *top* and *value* are used to build up the hierarchy and contain no object definitions of their own, all objects are defined in *atom*, *attribute* and *fs*. I show the definitions of *fs* and *attribute* below. The definition of *atom* is very similar to the definition of *attribute*.

```
class fs;
isa value;
constructor empty;
constructor add_pair:attribute,value,fs;
constraint empty>add_pair(A,V,FS);
constraint add_pair(A,U,add_pair(A,V,FS))=
            add_pair(A,and(U,V),FS);
constraint add_pair(A,U,add_pair(B,V,FS))=
            add_pair(B,V,add_pair(A,U,FS)).

class attribute;
isa top;
constructor instances.
```

A class definition contains the name of the class and the parent in the hierarchy. Furthermore, classes can contain constructor and constraint definitions. The constructor definitions state what elements the class contains. In the definition of *fs* above, the class contains one element *empty*, and one element *add_pair(A,V,FS)* for each possible instantiation of *A* as an *attribute*, *V* as a *value* and *FS* as a *fs*. In the definition of *add_pair* the symbols after *:* refer to the type of the arguments to *add_pair*. Here it can be noted that FLUF makes use of typed terms. In the examples I omit the types when writing terms since they are clear from the context.

The definition of *attribute* makes use of a special constructor *instances*. This constructor means that the elements in the class are defined as objects below it.

In the definition of *fs* constraint relations are used. In FLUF '=' is used to specify that two terms should be interpreted as equal and '<'or '>' to specify the subsumption order between the defined elements. The reason for having both >- and <-relations is that the left hand side of a relation is seen as the expression being defined and the right hand side as what it is defined as.

In the example above the first constraint tells us that *empty* should subsume every constructor starting with *add_pair*.[2] The second relation states that *fs*'s containing the same attribute more than once should have the same interpretation as the structure containing the attribute only once with the conjunction of the two values as its value. The third equation says that the attribute order in a *fs* is irrelevant.

Next conjunction and disjunction are added. They do not add any new elements to our language and are defined as functions. For a function the name of the function, the type of the result and the number and type of arguments to the function are specified. To give the meaning of function expression constraints are specified as relations in the same way as for classes. The definitions of *and* and *or* are given below.

```
function and;
result value;
arguments value,value;
constraint and(X,Y)<X;
constraint and(X,Y)<Y.

function or;
result value;
arguments value,value;
constraint or(X,Y)>X;
constraint or(X,Y)>Y.
```

By these definitions both functions give a *value* as result and take two *values* as their arguments. The constraint definition of *and* tells us that *and* is subsumed by its arguments, while *or* subsumes its arguments.

Next, some linguistic knowledge is defined. First the attributes and atoms used by the application are given. This can be done using objects. An object is specified by just giving a name and a parent in the inheritance hierarchy. What is special here is that object definitions are only allowed if there is an ancestor in the hierarchy which has a constructor specified as *instances*. As an example I give the definition of *cat*.

```
object cat;
isa attribute.
```

When defining linguistic knowledge, concept definitions are used to group it into conceptual parts. In a concept definition the name of the concept and its parent in the inheritance hierarchy are specified. It is also possible to specify a requirement as a typed term. The meaning of such a requirement specification is that all objects that are of this concept must contain at least the information given by the requirement. Two concept definitions from the example are *wordcat* and *verb*. Their definitions are given below.

```
concept wordcat;
isa atom.

concept verb;
isa fs;
requires add_pair(cat,v,empty).
```

With this definition of PATR-II grammar rules can be represented as feature structures. The terms in FLUF allows assigning variables to subterms which gives a simple representation of coreferences in PATR-II.

A declarative and operational the semantics of FLUF is given in Strömbäck (1994b). The declarative semantics is an initial algebra semantics where the elements given by a definition are interpreted on a partial order. The operational semantics amounts to giving a unification algorithm which in many ways is similar to narrowing (see, for example, Middeltorp & Hamoen (1992)). The FLUF formalism is sound, but not fully complete.

4 FURTHER EXAMPLES

In this section I give two further examples that demonstrate the flexibility of FLUF. The first example

2. Here there is a slight difference from PATR-II since *empty* does not subsume atoms. The interpretation used in PATR-II can be obtained by defining *empty* as a *value*.

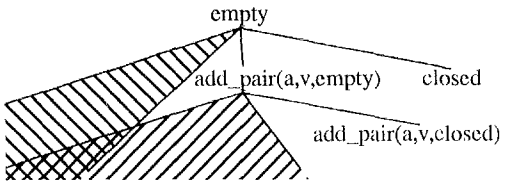

Fig. 2: The subsumption order for closed feature structures

shows how fixed arity feature structures (used in, for example, STUF (Beierle *et al.*, 1988)) can be defined.

```
class fs;
isa value;
constructor empty;
constructor closed;
constructor add_pair:attribute,value,fs;
constraint empty>closed;
constraint empty>add_pair(A,V,FS);
constraint add_pair(A,U,add_pair(A,V,FS))=
          add_pair(A,and(U,V),FS);
constraint add_pair(A,U,add_pair(B,V,FS))=
          add_pair(B,V,add_pair(A,U,FS)).
```

Compared to the definition of *fs* given previously, a new constructor *closed* is added. The idea here is that a feature structure ended with *closed* has a fixed arity and can not be extended with new attributes. The given constraint relations would give us the subsumption order shown in Fig. 2. The most general structure is at the top. The shadowed areas represent feature structures not explicitly written in the figure.

The next example shows how different interpretations of sets can be defined in FLUF. First I give a definition of sets corresponding to that used in HPSG (Pollard & Moshier, 1990).

```
class set;
isa ...;
constructor empty;
constructor add_elem(value,set);
constraint add_elem(V,add_elem(V,S))=
          add_elem(V,S);
constraint add_elem(V1,add_elem(V2,S))=
          add_elem(V2,add_elem(V1,S)).
```

Here the two constructors for sets *empty* and *add_elem* are defined. The two constraint relations in the definition say that each element only occurs once in a set and that the element order in a set is irrelevant. With this definition the unifications below hold. To increase readability I have used the common notation for sets.

$\{X,Y\}\cup\{a\}=\{a\}$
$\{X,Y\}\cup\{a,b\}=\{a,b\}$

In the first example the first constraint rule, identifying *X* and *Y*, is used.

For some linguistic phenomena it would be more useful to have an interpretation of sets where unification works like union. In FLUF this can be obtained by the definition below.

```
class set;
isa ...;
constructor empty;
constructor add_elem(value,set);
constraint empty>add_elem(V,S);
constraint add_elem(V1,add_elem(V2,S))=
          add_elem(V2,add_elem(V1,S)).
```

The difference between this definition and the previous one is that *empty* subsume all other sets. An element is also allowed to occur more than once in a set. With this second definition the first unification above has three possible results.

$$\{X,Y\}\cup\{a\}=\begin{cases}\{a,Y\}\\ \{X,a\}\\ \{X,Y,a\}\end{cases}$$

In the first result *a* is identified with *X*, in the second with *Y*, and in the third with neither of them. Presently FLUF gives all these three results as alternatives of the unification.

The reason why a set can be extended with new elements in the second definition but not in the first is that the semantics of FLUF assumes that if two expressions are not ordered by some constraint relation they are incomparable. Thus in the first definition sets are only related if all their elements are. FLUF assumes that all defined constructions are monotonic, so by the constraint relation given for *empty* in the second definition it can be concluded, for example, that $\{a\}>\{a,b\}$.

Other examples of what can be defined in FLUF are *lists*, *trees* and *strings*. It is also easy to define functions such as concatenation of lists in FLUF.

5 DISCUSSION

This paper discusses how a flexible unification formalism that can be used to obtain tailored unifiers for specific problems can be designed. I identify some important properties of a formalism that can be used to obtain flexibility. These are that the user must be allowed to define the elements he needs and functions on them. He must also have some way of defining the behaviour of his elements and functions. I observe that there are several advantages with using an inheritance hierarchy for defining the formalism and linguistic knowledge.

I present the FLUF formalism as a concretization of these ideas of a flexible formalism. As for the expressiveness of FLUF, it is still limited. There is a need for extending the hierarchy to allow for multiple inheritance and non-monotonicity. Strömbäck (1994a) provides more discussion on the expressiveness of FLUF.

There is very little discussion about unification algorithms in this paper. There is, however, a pilot implementation of the FLUF formalism. The implementation handles everything described above, but is very inefficient since it is based directly on operational semantics. There are, however, several improvements that can be made, for example applying existing results for more efficient narrowing (Hanus (1993) gives an overview) and integrating existing unification algorithms for some commonly used structures such as feature structures. The idea of integrating existing algorithms ensures us a more predictable behaviour for FLUF.

Another possibility is to use ideas from constraint logic programming (Jaffar & Lassez, 1987). This is particularly important in applications where this system is combined with some other process, for example, a parser.

ACKNOWLEDGEMENTS

This work has been supported by the Swedish Research Council for Engineering Sciences. I am also grateful to Lars Ahrenberg for guidance on this work.

REFERENCES

Beierle, C, U Pletat, and H Uszkoreit (1988). An Algebraic Characterization of STUF. LILOG Report IBM Deutschland, P.O. Box 800880, 7000 Stuttgart 80, West Germany.

Carpenter, B, (1992). *The Logic of Typed Feature Structures with Applications to Unification Grammars, Logic Programs and Constraint Resolution.* Cambridge Tracts in Theoretical Computer Science 32, Cambridge University Press.

Computational Linguistics 18(2-3). Special Issue on Inheritance in Natural Language. June and September 1992.

Dörre, J, and A Eisele (1991). A Comprehensive Unification-Based Grammar Formalism. DYANA Report. Deliverable R3.1B. January 1991.

Ehrig, H and B Mahr (1985). *Fundamentals of Algebraic Specifications 1. Equations and Initial Semantics.* Springer-Verlag, Berlin, Heidelberg.

Emele, M C, and R Zajac (1990). Typed Unification Grammars. *Proc. 13th International Conference on Computational Linguistics,* Helsinki, Finland, Vol 3, pp 293-298.

Hanus, M, (1993). The Integration of Functions into Logic Programming: From Theory to Practice. Manuscript, Max-Planck-Institut für Informatik, Saarbrücken.

Jaffar, J, and J L Lassez (1987). Constraint Logic Programming. *In Proceedings of the 14th ACM symposium of Principles of Programming Languages.* Munchen, Germany. pp 111-119.

Kaplan, R. and J. Bresnan (1983). A Formal System for Grammatical Representation. In: J Bresnan Ed., *The Mental Representation of Grammatical Relations,* MIT Press, Cambridge Massachusets.

Middeltorp, A, and E Hamoen (1992). Counterexamples to Completeness Results for Basic Narrowing. In: H. Kirchner and G. Levi Ed., *Proceedings of the 3rd international conference on Algebraic and Logic Programming,* Volterra, Italy. pp. 244-258, LNC 632, Springer-Verlag.

Pollard, C and Ivan A S (1987). *Information Based Syntax and Semantics. Vol 1.* CSLI Lecture notes, CSLI Stanford.

Pollard, C J, and M D Moshier(1990). Unifying Partial Descriptions of Sets. Manuscript.

Seiffert, R (1992). How could a good system for practical NLP look like? Paper presented at the workshop on *Coping with Linguistic Ambiguity in Typed Feature Formalism* at the *European Conference on Artificial Intelligence.* Vienna, Austria.

Shieber, S M, H Uszkoreit, F C N Pereira, J Robinson, and M Tyson (1983). The Formalisms and Implementation of PATR-II. In: Barbara Grosz and Mark Stickel, Ed., *Research on Interactive Acquisition and Use of Knowledge.* SRI Final Report 1984, SRI International, Menlo Park, California.

Strömbäck, L (1992). Unifying Disjunctive Feature Structures. Proc. 14th International Conference on Computational Linguistics, Nantes, France, Vol 4, pp 1167-1171.

Strömbäck, L (1994a). FLUF: A Flexible Unification Formalism - the Idea. Technical Report. LITH-IDA-R-94-12.

Strömbäck, L (1994b). FLUF: A Flexible Unification Formalism - Syntax and Semantics. Technical Report. LITH-IDA-R-94-13.