EFLUF - an Implementation of a FLexible Unification Formalism

Lena Strömbäck
Department of Computer and Information Science
Linköping University
S-58185 Linköping, Sweden
lestr@ida.liu.se

Abstract

In this paper we describe EFLUF - an implementation of FLUF. The idea with this environment is to achieve a base for experimenting with unification grammars. In this environment we want to allow the user to affect as many features as possible of the formalism, thus being able to test and compare various constructions proposed for unification-based formalisms.

The paper exemplifies the main features of EFLUF and shows how these can be used for defining a grammar. The most interesting features of EFLUF are the various possibilities to affect the behavior of the system. The user can define new constructions and how they unify, define new syntax for his constructions and use external unification modules.

The paper also gives a discussion on how a system like EFLUF would work for a larger application and suggests some additional features and restrictions that would be needed for this.

1 Background

During the last decade there has been a lot of work invested in creating unification-based formalisms and environments suitable for representing the knowledge needed in a natural language application. During the first years the work was concentrated on getting more expressive formalisms. This resulted in formalisms as, for example, (Johnson and Rosner, 1989), (Emele and Zajac, 1990), (Dörre and Dorna, 1993) and (Carpenter, 1992). Lately there has also been work concentrating on providing environments for large scale grammar development, such as (Alshawi et al., 1991) and (Krieger and Schäfer, 1994).

Even more recent is the work on GATE (Cunningham et al., 1996) which allows the user to combine different modules in a simple way. GATE differs from the systems mentioned above since it is an environment that enables to combine various kinds of modules into a system. This means that a particular submodule in a system built with GATE can be unification-based or of any other kind but GATE in itself does not make any prerequisites on the type of the module.

In this paper we will describe FLUF (FLexible Unification Formalism) (Strömbäck, 1994), (Strömbäck, 1996) and its implementation EFLUF. FLUF differs from other unification-based formalisms in that its aim is to provide a general environment for experimenting with unification-based formalisms. This means that the basic FLUF formalism does only cover very basic concepts used for unification, such as terms, inheritance and a possibility to define new constructions. The user can then tailor FLUF to his current needs by making definitions or importing external modules.

The work on FLUF covers both a theoretical description and an implementation, see (Strömbäck, 1996) for a thorough description on the theory of FLUF. The implementation started out as a testbench for experimenting with the theoretical ideas. Later on the implementation was extended with features necessary for handling larger examples. These extensions basicly covers the possibility of importing external procedures and a powerful way of defining syntax macros. However, also with these additional features EFLUF is a unification-based formalism and there has been no work on how to use it together with other types of frameworks.

The main part of this paper will describe the features of EFLUF by discussing a small example grammar. At the same time we will try to relate the various features of EFLUF to similar features in other unification-based formalisms. After that we will give some discussion on the experience from working with EFLUF which gives some directions for how a future system could be built.

2 EFLUF - an overview

In this section we will give a short description of how EFLUF works by giving an example that demonstrates how the different features of EFLUF can be used. For this example we have chosen to use a grammar similar to an extended DCG. In (Strömbäck, 1996) there are examples of how EFLUF also can be used for defining other grammars.

Classes and Inheritance An EFLUF grammar is similar to an object-oriented programming language where the user can define classes for the various objects he wants to handle. Each class contains a set of objects which are defined by `constructor` declarations. A class inherits its objects from the ancestors in the hierarchy. Multiple inheritance is allowed in a slightly restricted form.

To make it easier for the user to organize his definitions EFLUF code can be separated into different modules that can be combined in various ways. To use DCG grammars we use a basic module containing the four classes `word`,`constraint`, `category` and `rules` that represents the various DCG-objects we need.

```
#include "dcg.macro"
#unifierfile "dcgparse.pl"

class object.

class word;
isa object;
constructor instances.

class constraint;
isa object.

class lexconst;
isa constraint;
constructor lex:word.

class category;
isa object;

class list;
isa object;
constructor nil;
constructor add_elem:object,list.

class rules;
isa list;
constructor c:category;
constructor w:word;
unifier parse indef.
```

The class `word` is used for representing the word string. The definition `constructor instances` simply states that the class will contain the set of objects defined as words in the grammar.

The classes `category` and `constraint` represent grammatical categories and general constraints. In our DCG-module there is one subclass `lexconst` used to represent lexical constraints. This class contains objects consisting of terms with the functor `lex` and one argument.

The last class `rules` is used for representing grammatical and lexical rules. To build these rules there is a need for the constructors `w(_)` to mark a word and `c(_)` to mark a category. We also need lists to represent the right hand side of a grammar rule. Lists are here built from the two constructors `nil` and `add_elem(_,_)` and are defined from a separate class from which `rules` inherit.

Syntax macros In basic EFLUF syntax the given definition allows defining grammar rules and lexical entries in a rather complicated syntax.

```
constraint c(s(...))=
    add_elem(np(...),
             add_elem(vp(...),nil));
constraint w(john)=c(n(...));
```

To provide a more convenient syntax the user is allowed to define syntax macros. Syntax macros are defined in Emacs Lisp and are used as a preprocessor of EFLUF files. In the DCG-example above they are defined in a separate file and loaded by the include statement in the beginning of the example. The syntax macros allows the two example rules above to be written with the simplified syntax below.

```
gramrule s(...) -> np(...) vp(...);
lexrule john n(...);
```

In the examples syntax macros are also going to be used to allow a more convienient way for defining word strings. With syntax macros the user is allowed to write such definitions as:

```
defword john
```

This is a shorthand for defining a EFLUF object. The full EFLUF definition without using syntax macros would be:

```
object john;
is word.
```

In the examples used in this articles we are also going to use syntax macros to allow for a more convienient syntax for lists and feature structures.

External processes One last thing to note about the class `rules` defined above is the `unifier`-statement. This allows the user to specify an external process and in this case loads the Prolog chart-parser from (Gazdar and Mellish, 1989). The declaration `indef` at the end of this file means that the

parser can give more than one answer for a query. The actual code for the parser to be loaded by this definition is specified by the statement **unifierfile** at the top of the example.

In the current implementation an external process could be any Prolog program that takes two EFLUF objects as input and produces a new object as output. There are several ways in which external processes can be used. The parser above uses the grammar rules defined within EFLUF for parsing. Parsing could also have been done with the general unification operation provided by EFLUF but the chart parser provides more efficient parsing.

Another common use for external processes is as an alternative for writing unification rules within EFLUF. For some objects EFLUF's unification rules provides very inefficient unification or unification that seldom will terminate. In this case it is better to use an external process that provides more efficient unification for these kinds of objects. An example of this will be given when we introduce feature structures into our example later in this paper.

Inheritance versus Subsumption We also want to add some linguistic knowledge into the example. To demonstrate generality we show two different representations of number and person agreement in english. In the first representation the inheritance hierarchy is used. With this representation agreement information can be correctly unified by using inheritance between classes.

```
#include "dcg.fluf"

class agreement;
isa constraint.

class sg;
isa agreement.

class pl;
isa agreement.

class sgthird;
isa sg.

class sgnonthird;
isa sg.
```

A second way for the user to represent this information in EFLUF is to define a subsumption order of objects. The example shows the same relations as the inheritance hierarchy but now represented as a subsumption order of atomic objects.

```
#include "dcg.fluf"

class agreement;
isa constraint;
constructor sg;
constructor pl;
constructor sgthird;
constructor sgnonthird;
constraint sg > sgthird;
constraint sg > sgnonthird;
```

This way of defining a subsumption order by inequalities is in EFLUF called defining constraint relations. The defined constraint relations can be used with the EFLUF unifier, which uses a modification of lazy narrowing by inductive simplification, (Hanus, 1992), to unify the corresponding expressions according to the derived subsumption order.

Constraint relations Constraint relations can in EFLUF be used also together with expressions containing variables. This gives the possibility to define more general relations, and in particular functions can be defined in a way similar to, for example, (Johnson and Rosner, 1989) and (Emele and Zajac, 1990). Below we give an example of how appending of lists can be defined. Note that in this example we use = instead of >. This means that EFLUF will interpret the function call and its result as identical in the subsumption order.

```
#include "dcg.fluf"

function append;
result list;
arguments list,list;
constraint append(nil,L)=L;
constraint append({E|L1},L2)=
                {E|append(L1,L2)}.
```

When computing the unifications the unifier uses a mixture between lazy narrowing and inductive simplification. This means that the unifier uses the given constraint relations to simplify an expression as far as it can without binding any variables. When this is not possible anymore it tries to bind variables but only if necessary for finding a solution. When doing this it must keep track of alternative bindings. The user can affect this behavior by specifying for each constraint relation that it should be used only for narrowing or simplification. In the first case we obtain a more efficient behavior but all alternatives are not considered and the function cannot be run backwards. We might also sometimes lose alternative answers to a query. In the second case simplification is not used and we get a lazy behavior of the algorithm that always investigates alternative solutions.

To concretize this discussion we will give two example queries. To start with, the query

```
append({a,b},{c,d})=R
```

gives the expected answer `R={a,b,c,d}` using lazy narrowing combined with simplification. The same

answer would in this case be received by using only simplification since it can be found without binding any variables within the arguments of **append**. Using only lazy narrowing would however produce the answer {a|append({b},{c,d})} since this is the most *lazy* answer to this query.

If we instead consider the query

```
append(X,Y)={a,b,c,d}
```

both lazy narrowing and lazy narrowing together with inductive simplification will produce the expected five bindings of `X` and `Y` as results. Using simplification alone would however not find any answers since this is not possible without binding any of the variables `X` or `Y`.

Adding linguistic knowledge We will now continue by exemplifying how rules for nouns and nounphrases can be entered into the grammar. Doing this there is a need to both specify the actual words, the categories and constraints to be used and also the particular grammar rules and lexical entries.

```
#include "agreement.fluf"

defword john
defword apples
defword horses

class nhead;
isa constraint;
constructor nhead:lexconst,agreement.

class npcategories;
isa category;
constructor np:constraint;
constructor n:constraint.

class nprules;
isa rules;
gramrule np(HEAD) -> n(HEAD);
lexrule john
          n(nhead(lex(john),sgthird));
lexrule apples
          n(nhead(lex(apples),pl));
lexrule horses
          n(nhead(lex(horses),pl));
```

Here it can be noted that we make use of the basic classes for DCG when adding linguistic knowledge. To make it easier to separate the grammar into smaller modules we define the knowledge needed for nounphrases in new subclasses to the original classes defined for DCG.

Disjunctive information Next step is to extend this small grammar with information on phrases and verbs. Doing this we would like to add verbs that are either **pl** or **nonsgthird** in our specification of agreement. To avoid duplicate entries there is a need for disjunction. One way to define this in EFLUF is by defining disjunction as a function with constraint relation.

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

An alternative more specialized way to represent this would be to add one more constructor together with constraint relations into the given definition of agreement.

```
constructor plornonthird;
constraint plornonthird > pl;
constraint plornonthird > sgnonthird.
```

Combining different datatypes To demonstrate that it is possible to mix different structures in EFLUF we are going to use feature structures for representing the arguments of a verb. To do this we add a module containing the definitions needed for representing feature structures. Note that we use an external procedure to obtain efficient unification of feature structures. We also need some syntax macros to obtain a suitable syntax for writing feature structures.

```
#include "fs.macro"

class attribute;
isa object;
constructor instances.

class fs;
isa constraint;
unifier fsunify def;
constructor empty;
constructor add_pair:
              attribute,constraint,fs.
```

Inheritance of linguistic knowledge With the given definitions verbs and phrases can be defined. As mentioned above feature structures and terms are mixed for representing the constraints needed for phrases. Another thing that can be noted is that we now make use of the inheritance hierarchy for structuring linguistic knowledge. This is done when defining various types of verbs. For the class **verb** there is a so called **dimension** declaration. This declaration is used to specify whether classes are considered to be mutual disjunctive or not. This is very similar to multidimensional inheritance as used in (Erbach, 1994).

```
#include "nounphrases.fluf"
#include "fs.fluf"

defattributes agr, subj, obj.

defword eat
defword runs

class phead;
isa constraint;
constructor phead:lexconst,fs.

class verb;
isa fs;
dimensions sgthrdverb nonsgthrdverb /
           intransitive transitive.

class sgthrdverb;
isa verb;
requires [agr: sgthird].

class nonsgthrdverb;
isa verb;
requires [agr: plornonthird].

class intransitive;
isa verb;
requires [subj: _:nhead].

class transitive;
isa verb;
requires [subj: _:nhead, obj: _:nhead].

class phrasecategories;
isa category;
constructor s:constraint;
constructor vp:constraint;
constructor v:constraint.

class phraserules;
isa rules;
gramrule
  s(phead(LEX,V=[subj: SUBJ, agr: AGR])
  -> np(SUBJ=nhead(NLEX,AGR))
     vp(phead(LEX,V));
gramrule
  vp(phead(LEX,V:intransitive))
  -> v(phead(LEX,V));
gramrule
  vp(phead(LEX,
           V=[obj: OBJ]:transitive))
  -> v(phead(LEX,V)) np(OBJ);
lexrule runs
  v(phead(lex(runs),
          _:intransitive:sgthrdverb));
lexrule eat
  v(phead(lex(eat),
          _:transitive:nonsgthrdverb)).
```

Requirements on what information an object of a class must contain can be added by specifying a requirement definition. Requirement definitions are inherited from the parent classes in the hierarchy. In this way the user can create an inheritance hierarchy which is similar but not identical to how inheritance is used in other formalisms such as TFS (Emele and Zajac, 1990) or ALE (Carpenter, 1992). In general it can be said that the typing provided by requirements in EFLUF is a bit weaker than the typing provided in the two other formalisms. For the moment nonmonotonic inheritance is not allowed in EFLUF. There are however theoretical results on how to include this (Strömbäck, 1995).

Weighted unification Finally we want to exemplify one more possibility for the user to affect the behavior of the unification procedure. Suppose we want to use sets in our grammar, but we know that set unification will be very inefficient. Then we might want the unifier to postpone unification involving sets as far as possible. This can be done by specifying a high weight for sets which causes the unifier to postpone unifications involving sets if possible.

```
class set;
isa constraint;
weight 20;
....
```

3 Sample parses

To complete the sample grammar given in the previous section we will give some examples of the results given when parsing some sentences with the given definitions. These examples also show how parsing queries are made to the EFLUF unifier.

```
| ?- uf({john,runs}:rules,c(_),R).

R = c(s(phead(lex(runs),
              [subj:nhead(lex(john),
                          sgthird),
               agr:sgthird]))) ?;
no

| ?- uf({horses,eat,apples}:rules,
        c(_),R).

R = c(s(phead(lex(eat),
        [subj:nhead(lex(horses),pl),
         agr:pl,
         obj:nhead(lex(apples),pl)])))?;
no

| ?- uf({john,eat,apples}:rules,c(_),R).
no
```

As can be seen by these examples a unification query to EFLUF is made by calling the Prolog procedure `uf`. This procedure takes three arguments; the first two are the expressions to be unified while the third is the result of the unification. To parse a sentence the procedure is called with a sentence as first argument. To force the system to parse this as a sentence instead of unifying it as a list the sentence is typed as belonging to the class `rule`. The second argument is used to say that we want something that matches `c(_)` as result. The reason for this is to prevent the unifier from being too lazy and just return the given sentence as result.

As can be seen by the given examples the first two sentences give the expected structures as result of the parsing while the third does not give any result due to the difference in agreement within `john` and `eat`.

4 Experience from EFLUF

The current implementation of EFLUF has only been used for toy examples. Even so, working with this system gives ideas on how a better environment should be built and we will conclude this paper by discussing some of these ideas. First we will discuss some general problems and give suggestions for how these can be solved. We will then look more specificly into the problems of modularization and efficiency. In particular we will suggest how this kind of system can be used as a help when building a new application.

EFLUF has been designed to be flexible in the sense that the user should be able to decide as much as possible of the formalism. This also means that the basic constructions provided by EFLUF are rather simple and that it is the definitions made by the user that actually set out the constructions provided in a particular application. This has been a main goal when designing EFLUF but there is at least two major drawbacks with this idea when thinking about building an environment for larger applications. The first drawback is that a general formalism often becomes computationally less efficient than a more specialized one and the second is that it requires more knowledge of the user than using a more specialized one.

We believe that it is possible to avoid this by designing a future version of EFLUF as a large library of various standard definitions. Here we could achieve better efficiency by providing efficient external unifiers and other processes for the modules of this library. Since the user could start with these predefined modules the system would also be much more easy to use. This idea of providing a library of external procedures has previously been investigated in (Erbach et al., 1993).

This kind of library of definitions could be built using the possibility to structure definitions into separate files. However, the only thing in the EFLUF formalism that actually supports this division into modules is the inheritance hierarchy.

Even if EFLUF definitions are structured into a library there is still need to support the user in managing this hierarchy. One interesting point here is how the typing works. In EFLUF we have adopted an idea similar to (Carpenter, 1992) which in EFLUF means that the system should be able to judge the type of an expression by only knowing its functor and number of arguments. When considering building large applications it might be better to use the type hierarchy for distinguishing various definitions. This means that it should be possible to use the same name for different constructors in different modules and that the system uses the typing as a help to distinguish which of these the user means, similar to module constructions used in many programming languages.

As said above one major drawback with a general formalism is that it gets less efficient. In EFLUF we have tried to improve this by providing ways for the user to affect the behavior of the unification algorithm. This can be done in three ways. First the user can specify if equations should be used only for induction or for narrowing. Secondly he can get the unifier to avoid some classes by specifying weights. At last he can also provide his own more specialized and efficient algorithms. Other formalisms allow similar ways of affecting the unification algorithms, for instance RGR (Erbach et al., 1993) and TDL (Krieger and Schäfer, 1994).

An interesting use of a system like EFLUF is as a tool for supporting the development of a linguistic application with both grammar and specialized unification algorithms. This can be done in the following way. First, the EFLUF system can be used to compare how well different constructions are suited to describe some subparts of the linguistic input. When the user has decided that some construction is relevant to his application, the performance of the EFLUF system can be improved by defining specialized unifiers and syntax macros for this construction if they were not already provided by EFLUF. The EFLUF system can then be used for defining and testing grammars and lexicons. Further syntax macros can then be defined to provide a syntax that is the same as the syntax required for the final grammar. In parallel with the development of grammar and lexicon the work on developing a more efficient implementation can be started. While developing an implementation much of the code for the syntax macros and specialized unifiers can be reused.

5 Comparison with other systems

Finally we want to pinpoint the most important features within EFLUF and give some comments on how these relates to other formalisms.

The major idea when defining EFLUF was to let the user himself define all the constructions he needs. The work on EFLUF shows that it is possible to provide a formalism where the user is allowed to define almost everything. This is a difference to most other unification-based formalisms which sees the possibility to define the constructions as an extension to the formalism and not as a basic concept.

The design of EFLUF can be seen as having the possibility of defining own constructions as a kernel and then the typing system is built on top of these. This is also the case for CUF and TFS while, for instance ALE, is designed with typing as the most basic concept and the possibility to define constructions as an add-on. It seems that formalisms designed with the possibility to define own constructions as a basic concept instead of as an add-on achieve a higher level of flexibility since the new datatypes defined are better integrated into the formalism.

As for the typing system in EFLUF, variants of typing have been investigated and employed. EFLUF can handle both open- and closed-world reasoning, maximal and nonmaximal typing and provides two different kinds of typing through constructor and requirement definitions. Most other systems do not provide this rich variety of typing strategies.

One important way of achieving a better overall performance of EFLUF is to allow the user to affect the behaviour of the unification algorithm. In EFLUF only two such possibilities have been implemented. Other formalisms, especially CUF and TDL, offer other possibilities that can be incorporated in future versions of EFLUF.

The idea of allowing a general constraint solver to call more efficient specialized unifiers is the most promising way of achieving high efficiency within a general constraint solver. Other formalisms also have this feature, for instance, being able to use external constraint solvers in ALEP. However, EFLUF combine the external constraint solver with a general possibility for the user to define new datastructures within the system.

An interesting question is how EFLUF relates to the GATE system. In GATE it is possible to combine modules working on a text into a system by defining in which order they should process the text. EFLUF is orthogonal to this since it provides a way for putting together submodules into a larger module defining for instance the behaviour of a parser. An interesting line for future work would be to investigate if this could be done in a similar and as simple way as it is done in GATE and if it would be possible to integrate the two systems.

6 Conclusion

This paper exemplifies how EFLUF can be used for defining a small grammar. This formalism contains constructions for allowing the user to decide what constructions are needed for his application. The implementation also allows the possibility to import external procedures, to divide the definitions into modules and to define a suitable syntax for his constructions.

Experience from working with this system shows that it would be possible to use these ideas as a basis for a system for developing various grammars. In this case we would need to build a library of definitions as a base for the user to start working with. This kind of system would be an interesting tool for experimenting with unification grammars.

The experience also shows that even though EFLUF provides basic constructions for modularizations there is a need for better support for the user. This would, for instance, be to supply support for avoiding name clashes.

7 Acknowledgements

This work has been founded by the Swedish Research Council for Engineering Sciences. I would also like to thank Lars Ahrenberg and Göran Forslunds for helpful suggestions on this paper.

References

H. Alshawi, D. J. Arnold, R. Backofen, D. M. Carter, J. Lindop, K. Netter, S. G. Pulman, J. Tsulii, and H. Uzskoreit. 1991. Rule formalism and virtual machine design study. Eurotra ET6.1. Final report, SRI International, Cambridge Computer Science Research Centre, 23 Mille's Yard, Mill Lane, Cambridge CB2 1RQ.

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

Hamish Cunningham, Yorick Wilks, and Robert J. Gaizauskas. 1996. GATE - a general architecture for text engineering. In *Proceedings of the 16th International Conference on Computational Linguistics*, volume 2, pages 1057–1060, August. Copenhagen, Denmark.

Jochen Dörre and Michael Dorna. 1993. CUF - a formalism for linguistic knowledge representation. In Jochen Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description I.* August. Deliverable R3.1b.

Martin C. Emele and Rémi Zajac. 1990. Typed unification grammars. In *Proceedings of 13th International Conference on Computational Linguistics*, volume 3, pages 293–298. Helsinki, Finland.

Gregor Erbach, Mark van der Kraan, Suresh Manandhar, M. Andrew Moshier, Herbert Ruessink,

Craig Thiersh, and Henry Thompson. 1993. The reusability of grammatical resources. Deliverable D.A.: Selection of Datatypes LRE-061-61.

Gregor Erbach. 1994. Multi-dimensional inheritance. CLAUS-Report 40, Universität des Saarlandes, FR 8.7 Computerlinguistik, D-66041 Saarbrücken, Germany.

Gerald Gazdar and Chris Mellish. 1989. *Natural Language Processing in Prolog.* Adisson Wesley Publishing Company.

Michael Hanus. 1992. Lazy unification with inductive simplification. Technical report, Max-Planck-Institut fur Informatik, Saarbücken.

Rod Johnson and Michael Rosner. 1989. A rich environment for experimentation with unification grammars. In *Proceedings of 4th Conference of the European Chapter of the Association for Computational Linguistics*, pages 182–189. Manchester, England.

Hans-Ulrich Krieger and Ulrich Schäfer. 1994. TDL - a type description language for HPSG. Part 2: User guide. Technical report, DFKI Saarbrücken.

Lena Strömbäck. 1994. Achieving flexibility in unification formalisms. In *Proceedings of 15th Int. Conf. on Computational Linguistics (Coling'94)*, volume II, pages 842–846, August. Kyoto, Japan.

Lena Strömbäck. 1995. User-defined nonmonotonicity in unification-based formalisms. In *Proceedings of the 1995 Conference of the Association for Computational Linguistics*, June. Cambridge, Massachusetts.

Lena Strömbäck. 1996. *User-Defined Constructions in Unification-Based Formalisms.* Ph.D. thesis, Linköping University, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden.