Knowledge in Frame

KiF

Roux Claude
Parsing and Semantics
Xerox Research Centre Europe
Meylan, France
Claude.roux@xrce.xerox.com

***Abstract*—Xerox Incremental Parser (i.e. XIP) is the result of many years of work at Xerox to produce a syntactic parser to be used in industrial environments. However, even though we have implemented a fast and wide scale parser, we still have to deal with many aspects that have nothing to do with linguistics and can only be solved with traditional programming languages. After a first experiment with Python, we have decided to implement our own programming language KiF whose aim is to simplify the task of integrating algorithmic programming style within a Natural Language Processing platform.**

Natural language processing, embedded language, document processing, Python, KiF, industrial environment

I. Introduction

Natural Language Processing research usually focuses on the rule engine, either through machine learning techniques, which infer rules from already annotated sentences, or through symbolic approaches, where rules are directly designed by linguists. However, when dealing with actual applications, some of the problems that arise during grammar development are often out of the boundaries of the linguistic formalism. Rules are not designed, for instance, to record information or to count occurrences of words or categories. This part of the processing is usually implemented with external scripts, often in Perl or in any other script languages that come as handy for the task, with as a result, a system which lacks both in portability and in maintainability.

A. *Xerox Incremental Parser (i.e. XIP)*

For years, linguists have tried to implement syntactic parsers through complex and constrained theoretical models, such as LFG, GPSG or HPSG (see [1]), which despite their efforts ended up in almost intractable systems. Most of these earlier parsers would consider a grammar as a bag of rules, which the rule engine would combine on the fly to analyze a sentence. However, when the grammar comprises up to 10,000 rules, the complexity is such that the system either returns no output or thousands of solutions, usually after a pretty long processing time. To remedy this situation, computational linguists have implemented new approaches which use probabilities to sort out rules along weights computed out of annotated corpora (such as the Penn Tree bank [3]). However, if machine learning parsers have become prevalent today in the research community, they present some problems in an industrial environment. First, the learning stage induces a very specific bias due to the initial corpus on which the parser was trained. Usually, the overall performance decreases with new corpora. Second, if the parser commits the same recurrent errors, the only way to correct them is to present the system with as many examples as possible in order to bias the system out. Finally, the annotation process, whose role is often ignored or minored in the community, introduces many discrepancies and contradictions, which the machine learning model must take into account when extracting and weighting down the rules.

The Xerox Incremental Parser (see [4], [5]) on the other hand is still based on a symbolic implementation; an approach that is quite widespread in the industry, as symbolic parsers are usually less sensitive to new domains and easier to amend. XIP belongs to the same robust parsing tradition as Connexor (see [2], [14]) or IFSP (see [10]), a research current in which grammarians have access to a wide palette of tools to organize and write syntactic rules, which are applied in a deterministic way. In the case of XIP, grammars are implemented with the help of a Java IDE, which provides all that is necessary to debug a grammar. Rules in a XIP grammar are grouped in layers, which are applied in sequence, the output of a layer being the input to the next. Thanks to this strategy, the system does not drift into intractable analyses and more importantly always yields a unique solution. XIP also exposes different sorts of rules to solve different linguistics problems, which again is a difference with previous models, which would only provide linguists with one single formalism, that would have to be tweaked to implement rare or exotic structures. Also, compared to probabilistic models, the correction of a grammar is much simpler as it consists of modifying or adding new rules. Grammars for eight languages (French, English, German, Italian, Spanish, Portuguese, Dutch and Japanese) have been developed so far, usually by one linguist, with an average of six months to implement a new language from scratch. In comparison, annotating a large corpus by hand requires usually a quite large team of students (the task is quite tedious to say the least) over a period of one or two years.

1) XIP Processing

A syntactic analysis includes the following stages:

1. Tokenization, which splits a document into a collection of tokens along spaces or punctuations.
2. Morpho-analysis, which interpret each token as a word or an expression and returns their part of speech and a set of features.
3. Tagging, which reduces the part of speech ambiguity of a given word.
4. Chunking, which groups together words around a syntactic head
5. Dependency extraction, which extracts the relations between the heads in a sentence.

a) Example:

If we analyze the sentence: *The bird flies.*
Then each stage will perform the following operations:

1. Tokenization: [the,bird,flies]
2. Morpho-analysis: [the+determiner,bird+Noun, {flies+Pl+Noun,flies+Verb}] *flies is ambiguous*
3. Tagging: [the+determiner,bird+Noun, flies+Verb] *flies is no longer ambiguous*
4. Chunking: NP:[the bird] VP:[flies]
5. Dependencies: SUBJECT(flies,bird) and DETERMINER(bird,the).

For each of these stages, XIP provides specific implementations. For instance, tokenization and morpho-analysis are done with transducers, while tagging, chunking and dependency extraction are executed with hand written rules.

2) Formalism Enrichment

But rules are blind and deaf: they are executed in a "launch and forget" mode, which makes it quite difficult for linguists to write rules that would benefit from previous analyses, as *Declerck* [6] shows in his article. In order to solve this issue, we first introduced variables (strings, floats, vector and maps) into the formalism to track information throughout the whole linguistic process. We also added the possibility to implement small functions or procedures, which could be called from any rules. Named Entities recognition (see [11]) is a very nice illustration of how these variables can prove beneficial. A given Named Entity (a proper name or a company name for instance) usually occurs more than once in a text. However, the context in which this entity is clearly identifiable as such is often quite limited. Thanks to these variables, it is possible to keep a track of all entities that have been found so far in a document and then still give an interpretation to these entities whenever they find themselves in a poor context.

3) Python Integration

However, if the enrichment of this formalism would eventually turned out to be too complex within the parser code, the way functions were implemented proved to be a very interesting starting point. XIP is implemented in C++ and function calls were implemented as a specific class, which could be easily derived. Instead of trying to amend again and again the *XIP scripting language* code that was too closely knotted into the rule engine, we moved to another solution, which was to build an interface between XIP and Python.

Python is written in C and provides a straightforward API to create basic objects such as strings, integers or maps. We linked Python and XIP in two ways (see [9]). First, we transformed XIP into a Python library, which could be loaded from within a Python program. Second, we implemented a specific interface so that Python functions could be executed from within rules. We used the comment section of Python (a set of lines between two triple quotes) as a placeholder for XIP function declarations, which were bound with the actual Python functions implemented in the same file. Thus, every call to *a function* in XIP results into a call to the eponymous function in Python. Parameters are translated on the fly from XIP into Python, while the *return* data from Python functions are transformed back into XIP values. A Python function can then be used to return a numerical value that can be exploited from within the rules.

However, even if this solution proved quite reliable, we faced some issues. First, Python cannot have direct access to the internal linguistic structures in XIP, and can only apply to duplications of these structures through a set of dedicated functions. For instance, a syntactic node is known to Python only as an integer *id*, which is used to fetch its features through a specific function. Worse, memory leaks proved an absolute nightmare to track down.

Second, since XIP communicates with Python through a C programmatic interface, the XIP binary is heavily dependent on the version of Python with which it is linked. If XIP is linked with Python 2.6.2, then running grammars with Python functions embedded requires the exact same Python version on another machine or Python binary libraries such as *math* will not load at all.

II. Knowledge in Frame Language

Despite the issues mentioned above, Python had proven very useful in many cases, and since we wanted to keep a script language in our linguistic arsenal, we just could not simply put Python aside without replacing it with something else. We decided to implement a new language that would be close to Python in some aspects, but would be more suitable to our linguistic purposes. The *Knowledge in Frame* language (KiF) was implemented with the objective of having a full access to all XIP linguistic structures with a formalism that would be as light and simple as Python. Thus, KiF can modify XIP variables or linguistic structures and KiF functions can be transparently accessed from any rules without an interface, as with Python. Furthermore XIP can be used to run KiF scripts, which allows for a pre-processing of documents before passing the result to the rule engine itself. Actually, the integration is so transparent, that KiF functions can be implemented anywhere in a XIP grammar.

1) Red Thread

We will show in the article how we can build a very simple client/server application encapsulating the XIP parser, with *rules* exchanging data with KiF.

First of all, we assume that our grammar will contain the following rule:

```
if ($1(#1,#2) ) {store($1,kif_exchange_data);}
```

which reads as follow:

For any dependency (represented here with $1) with two arguments, we call store, *which is a KiF function, with two parameters.*

- *The first parameter is the dependency itself.*
- *The second parameter is an external parameter which was passed to XIP from KiF.*

This rule will be appended at the end of the grammar and will send back to KiF all the dependencies computed for a given sentence. *kif_exchange_data* will be crucial in our implementation to have both the calling program and the rules exchanging data.

B. KiF

KiF is a multithreaded language implemented in C++, and it is available on most platforms (Windows, Linux or Mac OS). It is a cross-over of C++, Python and Java. From Python, we have kept the notion of all-purpose dictionaries and vectors, while from C++ and Java we have borrowed the notion of type declaration. As Java or Python, it is a garbage-collector based language, where each object has a specific reference, which indicates whether the object has been fully released or not. We don't use the Python indentation style but rather the more traditional C++ or Java's utilization of curly brackets.

```
int i;    //we declare an integer
i+=10;  //we increment this integer by 10
for (i=0;i<10;i++) {//we loop
  int j=i*2;
  println(i,j);
}
```

1) Contextual Evaluation

From the above example, one would wonder what would differentiate KiF from Java or C++. The most important difference resides in the way expressions are analyzed. In KiF, expressions are evaluated from the left to the right, and the *type* of the expression is given by the leftmost element. For instance, strings, when used in a numerical environment are translated into the number they represent. Thus, the string "10" is automatically translated into the number 10. In the same way, any integer or float is also automatically translated into a string if the context demands it.

```
int i;
string s="10";
i=7+s;              //the result is: 17
s+=i;               //the result is:  "1017"
```

All types have different interpretations according to their context. A vector in a numerical expression returns its length. A file as a string, returns a pathname, while as an integer, it returns its size on the disk.

2) Parentheses

Another big difference is the way parentheses are interpreted. In most languages, parentheses are used to isolate or gather specific elements in a numerical expression: *(10+i)*5*. In KiF, they create a local context, where the new interpretation is imposed by the first element of the expression. *(10+i)*5* is still interpreted as in other languages, since 10 is a number. However, the interpretation of an expression such as: *"MY "+(10+2)* is quite different. The parenthetic part (10+2) is no longer interpreted as a string operation but as an integer operation, thanks to the *10*. Thus, *"MY "+(10+2)* is: "MY 12", while *"MY "+10+2* is: "MY 102".

Below is a list of expressions, whose interpretation is quite specific to KiF:

```
int i="2"+"3"+5;            // i= 2+3+5=10
i=("2"+"3")+5;              //i= ("23")+5=28
string s="MY "+(10+2);      //s="MY 12"
s="MY "+10+2;               //s="MY 102"
```

a) Reading a File

The small program below shows another example of this contextual evaluation. If we loop in a file with an integer, then KiF will read the file character by character. If we loop in the same file with a string, then each iteration will yield a full line.

```
file f('myfile.txt','r");
int i;
for (i in f)        //the file is read character by character
    println(i);
…
string s;
for (s in f)        //the file is read line by line
    println(s);
```

3) Marshalling, un-marshalling

The main reason for this choice is that in NLP we manipulate strings at length. Actually, NLP could be described as the science of string processing. For example, the vector *[1,2,3]* can be very easily transformed into the string "*[1,2,3]*", but if conversely a string is fed to a vector, then KiF will assume that this string is a vector description and will parse it to re-generate the corresponding object. This *marshalling* and *un-marshalling* applies to most objects in KiF. Complex structures can then be stored in databases as strings, leaving the task of parsing the objects back to KiF. It becomes much simpler to store a lexicon in a database. For instance, the feature list, which is rather tedious to describe in tables, can then be simply recorded as a string, which will be automatically re-analyzed as a map, when needed. The same idea presides over the choice of translating a string into a number when in the context of a calculus.

```
//s is initialized with a map description
string s={'gender':"masc",'pers':2};
map m=s;            //m is initialized with s
```

4) *Functions*

A function in KiF is composed of a name and a list of parameters. However, the philosophy of the language has an impact on the way parameters are converted. In fact, since KiF can take as input any objects and translates it as a string, there are very little constrains on the types of both the arguments and the parameters.

```
//A very simple declaration
function myfunc(string s) {
        println(s);
}

myfunc("test");     //of course no problem it is a string
myfunc(18);         //well "18" makes a nice string
myfunc(m);          //m is a map… but it does work
```

a) Strict

To overcome this behavior, KiF provides the keyword *strict* which prevents calls with a *map* as argument in the above example.

```
//A stricter declaration
strict function myfunc(string s) {
        println(s);
}

myfunc(m);          //m is a map… but here it fails…
```

In this case, the system will return a compile error.

In fact, if more than one function is declared with the same name, the *strict* behavior is automatically enforced.

```
//A very simple declaration
function func(string s) {…}
function func(map m) {…}
function func(int i) {…}

func("test");       //func with a string
func(18);           //func with an integer
func(m);            //func with a map
```

b) self

By default, variables in functions are passed as values, which forbid, for instance, a string variable to see its value modified in a function, as it is the case in Python.

KiF provides a specific type *self*, which sends a variable as a reference instead of a value. It is then possible in this case, to modify a variable within a function, if the parameter is of the *self* type.

```
//A very simple declaration
function modify(self s) {
        s+= "!";
}

string t= "toto";
modify (t);
println(t); //t is now toto!
```

c) Lambdas

This type *self* is especially useful in lambdas. Vectors and maps expose the method *apply* which is used to apply a function to each of their values. However, if the function in question is declared with a *self* parameter, then it becomes possible to also modify each of these values.

```
vector v=["a","b","c"]; //my vector

v.apply(modify); //we apply our function
println(v); //v=["a!","b!","c!"]
```

5) *Frames*

Frames are in KiF the equivalent of classes in other languages. Their declaration is very similar to a class definition in Java or in C++.

```
frame test {
        int i;
        string s;
        //the constructor is always _initial
        function _initial(int k) {i=k;   }
}

//To create instances of that class is pretty straightforward:
test toto(10);
//And access is also quite simple
println(toto.i);
```

Frames can be sub-framed and methods or operators can be overloaded. It is also possible to call the super-method of a mother class if necessary.

a) Dynamic Cast

Frame objects are also interpreted in context. We have described above how a simple string can be transformed into an integer, if this string finds itself in the context of a calculus. Frame objects can also be subjected to the same transformations. KiF provides a similar mechanism as the *toString* method in Java, requiring developers to implement specific functions to yield the right interpretation. In KiF, these functions bear the same name as the required cast. Thus, a developer can implement a *string()*, a *int()* or a *float()* function at this effect.

```
frame test {
        int i;

//A simple integer cast function
function int() {
        return(i);
}
}

test t;
t.i=10;
int k=20+t;  //k is 30…
```

This function will be automatically called by the interpreter in the context of a calculus. If no function is available, KiF will return *null*.

In the same way, it is also possible to implement a frame converter to translate a given object into another frame object. Consider for instance the following example, which implements the frame *othertest*.

```
frame othertest {
   int k;
  //A cast into a test object
   function test() {
        test t;       //we create our test object
        t.i=k;        //we copy the value
        return(t);   //we return this new object
    }…

othertest o;     //we create our object
o.k=100;         //we initialize o.k
test t=o;        //now t.i=100, the converter is automatically called to transform an othertest into a test.
```

b) Operator overload

It is also possible to overload any operators in a frame, through the re-declaration of that operator into a frame function.

```
frame add{
   int k;

   function _initial(int i) {
        k=i;
   }

  //We re-declare the "+" operator
   function +(add a, add b) {
        //This part matches a "+="
        if (a==this) {
           k+=b.k;
           return(this);
        }
        add c(0);
        c.k=a.k+b.k;
        return(c);
}

add a(10),b(2),c(0);
c=a+b;   //A simple addition
c+=b;    //the first element in the + will be this
```

All the available operators can be as easily overloaded, including the comparison operators.

6) Associate Functions (binding)

Associate functions are another specificity of KiF. Any variable can be bound with a specific function, which is called every time this variable is modified. This associate function must have two parameters (or three if it is a frame), which are basically the current value of the variable and its new value.

```
//before is the variable current value and after its new value
function modify(int before, int after)  {println(before,after);}
int  o with modif;
o=10;//will trigger a call to modify…
```

a) Graphics Functions

These functions are extensively used in the graphical library, which has been implemented on the top of the *FLTK* library. Thus, most of the graphical objects, such as *window*, *button, box etc…* can be associated with a function, which will be called whenever this object is modified.

In the case of *window*, the associate function is used to *redraw* the complete window with drawing primitives such as lines or circles. However, these functions have a very specific signature depending on the nature of the graphical object.

```
function gettext(button b,string s) {
        println("Button pressed=",s);
}

//we declare a window object
window w;
//which we position on screen with a title
w.create(300,200,100,100, "My window");
//we declare our button associated with gettext
button b("OBJECT") with gettext;
b.create(10,20,30,30,FL_Regular, "Ok");
w.end();
w.run();
```

Basically, the above code displays a window with a button in a corner with the Ok caption. Whenever the user clicks on this button, the *gettext* function is automatically called. "OBJECT", with which this button is associated is an object, which will be automatically passed to the associate function. In this example, it is a simple string, but it could be more complex.

7) Multithreading

Multi-threading is usually considered, with some good reasons, to be difficult. Since KiF is a language, which is targeted to linguists with little knowledge about programming, we wanted to keep multithreading as simple as possible. Actually, in KiF to declare a function as a thread consists of replacing the keyword *function* with *thread*. The whole language is implemented in such a way that the access to variables or the call to other functions is automatically protected within any *threads*.

```
//a simple thread description…
thread mythread(int k) {
      print(k, ",");
}
```

a) Synchronization

However, *threads* must synchronize with each other sometimes. KiF provides a full arsenal of tools at this effect: *mutex, join, semaphore, and synchronized variables*, yet implemented in such a way, that their utilization does not require any complex programming. For instance, the function *wait* puts a thread in hold, while the method *cast* releases it. The *wait* function accepts as parameters, a simple list of strings: *wait("one","two","three").* A *cast* on one of these strings will release all the *threads that wait on this string.* The *wait* function will then return as a result the very string that was cast in the first place.

```
//a thread waiting on strings…
thread mythread(int k) {
      string s=wait("test","off");
      println(s+","+k);  // print: "off,10"
}

mythread (10);  //We launch it
pause(0.001);   //We wait for 10ms
cast("off");    //We release it
```

b) Synchronized Variables

Synchronized variables are used in conjunction with the function: *waitonfalse*, which has been implemented as a semaphore. Any type of variables can be used as a synchronized variable, as long as it has a Boolean interpretation *(it can be a frame object)*. The variable must be associated with the pre-defined function: *synchronous.*

```
//a synchronized variable
int sync with synchronous=3;
…
waitonfalse(sync);
```

Thus, each time *sync* is modified, the system tests its Boolean value. When *sync* is 0, it then releases *waitonfalse*.

c) Join Threads

However, using *synchronized variables* is often too complex. A more simple way of synchronizing a set of threads with their master is to declare these threads as *join*, and then wait on their completion with *waitonjoin().*

```
join thread toto(int i) …

toto(10); //we launch our threads
toto(20);
waitonjoin(); //and wait for their completion…
```

Furthermore, the *waitonjoin* works at the thread level, which means that many *waitonjoin* can run in parallel waiting for different sets of threads in as many master threads.

d) Exclusive and protected

We have also borrowed from Java the notion of synchronized threads. In Java a *synchronized* function, for instance, is a function which is protected from concurrent access. However, this protection is only true from within the same object. For instance, if two different threads try to execute the same *synchronized* method S on an instance O of a given class, then only one of these threads can have access to it at a given time.

In KiF, the keywords *exclusive* offers the same protection for a given object instance. However, we have also added the notion of *protected*, which protects the access at the method level. In the case of a *protected* function, only one thread can have access to this method at a time, whatever the instance on which it depends.

In other words, in a *protected* thread, we use a lock that belongs to the method, while in an *exclusive* thread, we use a lock that belongs to the frame instance.

exclusive thread framethod(..) { lock(instanceid)…}
protected thread method(…) {lock(methodid)…}

```
//This frame exposes two thread methods
frame disp {

      //exclusive
      exclusive thread edisplay(string s) {
               println("Exclusive:",s);
      }

      //protected
      protected thread pdisplay(string s) {
               println("Protected:",s);
      }
}
```

III. XIP INTEGRATION

The bulk of the integration of XIP and KiF is a list of specific KiF types which are mapped over linguistic objects. For instance, syntactic nodes are mapped over *node* objects, which can then be used to access the part of speech, the lemma or any features from a given node. It is also possible to modify the content of that node, since we are dealing with an encapsulation of the node internal pointer in KiF.

A. XIP Rule

We have already introduced a rule at the beginning of the article, which calls a specific KiF function *store*. We can now flesh our current example

a) XIP side

The rule is the following:

```
if ($1(#1,#2)) {
        store($1,kif_exchange_data);
}
```

It loops among all dependencies that were computed and call the *store* function for each. We will come back to *kif_exchange_data* later.

b) KiF side

Here is our implementation of that function

```
function store(depedency d, map results) {
        //We store our data for this dependency
        //First we extract our parameters from d
        vector par=d.parameters();
        //We know we have two parameters
        string key=d.name()+"_";
        key+=par[0].lemma()+ "_"+par[1].lemma();
        results[key]=d.data();
}
```

For each sentence, the XIP rule will be executed and will call *store* for all dependencies with two arguments. This KiF function will then extract data from the dependency, and build a key out of the dependency name and the parameter lemmas. Thus, a dependency such as *subj(eat,dog)*, which might have been extracted from "the dog eats", will be stored in our map with the key: *subj_eat_dog.*

2) XIP types

Basically, XIP manipulates three sorts of data: *node, dependency and features.* In KiF, we have simply mapped these data into specific structures that encapsulate the pointer to these actual objects in memory. However, we have added a KiF flavor to these objects. For instance, a *node* as a string returns the part of the speech of the node and a node as an integer returns its numerical identifier. The same idea also applies to the *dependency* type.

In the case of the features, the transformation is less impressive, as features which are implemented in XIP as attribute/value pairs, are simply returned as a map.

```
function test(node n) {
        //we load the features from the node n as a map
        map m=n.data();
        println(n); //display the Part of Speech of the node
        int i=n; //we get the Id from the node
        println(n.lemma()); //return the lemma
}
```

The *node* type exposes many methods, which can be used to get the node position in the text stream or the node siblings in the syntactic tree. Furthermore, since the node type encapsulates the actual syntactic node, it is also possible to modify certain of its features.

```
function gender(node n) {
        //we modify a feature on the node
        n.setfeature("gender","neutral");
}
```

B. *XIP as a library:* kif_exchange_data

XIP is also implemented in such way that it is seen by KiF as an external library. In that specific case, it exposes a few classes and some methods.

1) *Type:* parser

"parser" is the main type through which a grammar can be loaded and applied to a string, a file or an XML document.

```
parser french;
map results;
//protected
french.load('c:\french.grm');
//we parse a string…
string res=french.parse("La dame mange une glace",results);
//res contains the result of the parse
println(res);
```

The method *parse* accepts two parameters:

- A string, the text chunk to analyze
- A second parameter which can be anything

2) *kif_exchange_data*

This second parameter (*results* in our example) is passed to XIP, and is then known from within the grammar as *kif_exchange_data.* The rules cannot actually manipulate this variable, but they can pass it back to KiF as shown in our *store* example. Thanks to this variable, it is possible to push a KiF object through the different layers of rules.

3) *Local callback*

Actually, our *store* function can be declared directly into the calling KiF program file.

```
function store(depedency d, map results) {
        vector par=d.parameters();
        …
}

parser french;
p.load('c:\french.grm');
map results;
//we parse a string…
french.parse("La dame mange une glace",results);
```

```
//dependencies has been enriched on the fly
println(results);
```

When the grammar is loaded, all the functions that have been declared in our KiF program are automatically available to XIP. This implementation is especially useful since it enables KiF to filter and clean the input before applying the grammar to a chunk of text.

C. *Client/Server*

A server traditionally launches a specific thread for each client, which is then used as a portal through which our server and our client exchange data.

KiF provides its own implementation of *socket*.

1) *Server side*

On the server side, we use the method *createserver(port,maxclients)*, which creates a listening socket. This socket can then *wait* on clients to connect. Each client is then identified with a specific numerical id.

```
socket s;
int clientid;
//We create our server on the port 2021, for 20 potential clients
s.createserver(2021,20);
while (true) {
//we wait for a client to connect
        clientid=s.wait();
        //we then launch our thread
        analyze(clientid,s);
}
```

The server can then use this *clientid* to communicate with the client through *read* and *write*.

- *read* receives strings, which can be automatically deserialized into any specific object.
- *write* automatically serializes any objects into a string.

2) *Client side*

On the client side, we use the method *connect(servername,port)* to connect to our server. We can then use *read* and *write* to communicate with the server.

```
socket s;
//we connect to our server
s.connect(s.gethostname(),2021);
//we then send a string to the server
s.write("La dame mange une glace.");
//which returns its value as a map
map res=s.read();
```

Again, *write* sends the object as a string, while *read* receives a string, which is automatically deserialized back into a map, in our example.

3) *Threads*

We have already introduced the threads in our presentation of KiF. Our thread implementation is a very simple function as the one below:

```
//we launch a thread with our client id
thread analyze(int clientid, socket s) {
        map deps;
        while (true) {
           try {
                  //we read our sentence
                  sx=s.read(clientid);
                  if (sx!="") {
                    deps.clear();
                   //we parse it
                   french.parse(sx,deps);
                   //the result is stored in our map, wich
                   //we send to our client
                    s.write(clientid,deps);
                  }
           }
         //if we have a problem, we stop
          catch(mes) {
                  s.close(clientid);
                  return;
        }
}
```

The thread will exchange data with the client, which could send as many text chunks as necessary. The result is stored in a *map* which is sent back to the client as a string. Thanks to the deserialization, the string is transformed back into a map.

IV. EVALUATION

We have compared KiF with Python in terms of speed and memory footprint. We did two evaluations: one as a standalone language, the other one from within XIP.

1) *As a standalone interpreter*

We have implemented many Python programs over the years, most of them doing intensive string manipulations in order to clean and extract proper data for our linguistic tools. We re-implemented about 20 of them into KiF. From our test, we could show that Python has a memory footprint which is about 15% to 20% smaller than KiF, with a speed which is between 10% and 15% faster.

2) *Within XIP*

We have also re-implemented many of our previous Python procedures into KiF, with results quite different from the previous tests. Python is still about 5% less greedy than KiF, in terms of memory footprint. As for the speed, the difference is quite difficult to compute, since most of these procedures account for only a tiny fraction of the whole processing time. Still, from our tests, it appears that Python and KiF are quite similar in terms of speed, thanks to the reduced data duplication, which consumes both time and resources.

3) KiF vs. Python

From our experiments, KiF appears slightly slower and greedier than Python. Actually, the difference is not that large that any KiF scripts would ruin the overall rule engine efficiency. KiF does not represent a terrible drift in terms of memory and speed, when evaluated from within XIP. However, we believe, that the benefits of improvements achieved in terms of functionality outweigh the differences in speed and memory footprint, since KiF represents a much better language in terms of integration into XIP than Python, as all linguistics objects and structures are seamlessly accessible from any scripts. Actually, since KiF and XIP are both written in C++, it is fairly simple to implement any specific KiF addendum in order to improve the access to all peculiar and strange data structures in XIP.

Also, KiF resolves some of the problems that plague script development with dynamic languages. First, KiF variables must be declared. Non-declared variables, means that errors such as misspelled names, can only be detected at run time, while in KiF they can be detected at compile time. Second, variable types are fixed, which means that variable types do not evolve during run time, reducing the risk of dark corners as in Python where problems can jump out of rarely used sections of code. We still think that Python is a remarkable language, with very powerful features, but the lack of declaritivity makes it difficult to debug and to maintain. Furthermore, Python programs cannot be debugged in development environments such as eclipse when they are run from within XIP. In the case of KiF, we have implemented an internal debugger, which can be called from within the rule engine at any times. Linguists can stop the parsing and examine which structures and which variables were sent to their KiF scripts, to see how these structures and variables are actually processed.

V. CONCLUSION

Natural language processing is the art of string manipulations. A large part of the effort of implementing grammars or NLP-based applications requires the programming of complex string transformations to deal with a large variety of inputs, either as documents or as linguistic data. The implementation of a language such as KiF, which provides a natural serialization/deserialization embedded into the fabric of the language helps simplify these tasks a lot. The language is of course richer than the simple description that has been given in this article. For instance, it is possible to build external libraries in which new types can be encoded to enrich the language. We have already encapsulated external libraries such as SQLite [15] or FLTK [13] into KiF modules, which can be loaded from any programs or grammars. Actually, KiF also exists as a standalone interpreter, which can be used as a script language on the command line. We have furthermore defined a C++ API, which can be used to execute KiF program from any C++ application in a very simple way.

The language still needs some polishing and some improvement both on speed and on memory footprint, however the language implementation is fairly recent and we have good hopes to progress on all these aspects.

Reference

[1] Gazdar G., Klein E., Pullum G., Sag A. I., 1985. Generalized Phrase Structure Grammar, Blackwell, Cambridge Mass., Harvard University Press.

[2] Tapanainen P., Järvinen T. 1994. *Syntactic analysis of natural language using linguistic rules and corpus-based patterns***,** Proceedings of the 15th conference on Computational linguistics, Kyoto, Japan, pages: 629-634.

[3] Grinberg D., Lafferty John, Sleator D., 1995. *A robust parsing algorithm for link grammars*, Carnegie Mellon University Computer Science technical report CMU-CS-95-125, also Proceedings of the Fourth International Workshop on Parsing Technologies, Prague, September, 1995.

[4] Roux C. 1999. *Phrase-Driven Parser*,Proceedings of VEXTALL 99, Venezia, San Servolo, V.I.U. - 22-24.

[5] Aït-Mokhtar S., Chanod J-P., Roux C., 2002. *Robustness beyond shallowness incremental dependency parsing*, NLE Journal, 2002.

[6] Declerck T. 2002, *A set of tools for integrating linguistic and non-linguistic information*, Proceedings of SAAKM.

[7] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan., 2002. *GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications*, Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02), Philadelphia, July 2002.

[8] Blache P., Guénot M-L. 2003. *Flexible Corpus Annotation with Property Grammars*, BulTreeBank Project

[9] Roux C. 2006, *Coupling a linguistic formalism and a script language*, Proceedings of the Third Workshop on Constraints and Language Processing, pages 33-40, 2006.

[10] Ait-Mokhtar, S. and Chanod, J.P. 1997, *Subject and object dependency extraction using finite-state transducers*, Proceedings of Workshop on Information Extraction and the Building of Lexical Semantic Resources for NLP Applications, ACL.

[11] Brun, C. and Hagege, C. - 2004, *Intertwining deep syntactic processing and named entity detection*, Advances in Natural Language Processing, 195--206, 2004, Springer

[12] [Python] http://www.python.org/

[13] [FLTK] http://www.fltk.org/

[14] [Connexor] http://www.connexor.com/

[15] [SQLite] http://www.sqlite.org/