

MorphTrans: a Language and Compiler to Specify and Generate Morphological Transfer Modules for Machine Translation

Alicia Garrido-Alenda

Mikel L. Forcada

interNOSTRUM

Departament de Llenguatges i Sistemes Informàtics

Universitat d'Alacant

E-03071 Alacant, Spain.

<http://www.internostrum.com>

Abstract

This paper presents a language to specify the rules of a morphological transfer module for a machine translation system as well as the compiler that may be used to turn the specification into an executable module. Morphological transfer works on the result of a morphological analyser followed by a part-of-speech disambiguator, that is, it works on sequences of lexical forms (lemma, part-of-speech, and morphological features). A set of rules specifies a module which detects the longest sequence of lexical forms among those patterns specified, calls the bilingual dictionary on all of the lexical forms, manipulates them, and then writes target-language lexical form sequences. The language is designed so that a linguist may easily write rules and the compiler generates a lex program that performs the task. The system has been used to build the transfer part of *interNOSTRUM*, a Spanish-Catalan machine translation system.

1 Introduction

Most machine translation (MT) systems are organized around the transfer architecture (Hutchins and Somers, 1992; Arnold et al., 1994; Arnold, 1993): source-language (SL) text is analysed into an abstract representation of some kind (analysis), then this representation is converted (transfer) into a similar representation but for the target language (TL), and finally, TL text is generated from it (generation). The Spanish-Catalan MT system *interNOSTRUM* (available at <http://internostrum.com/>) uses an advanced morphological transfer strategy which is very similar to the one used in commercial PC-based MT systems such as Transparent Technologies' Transcend RT, early versions of Globalink's Power Translator, and Softissimo's Reverso. *interNOSTRUM*'s operation has the following stages:

1. ANALYSIS : morphological analysis; part-of-speech disambiguation (statistical);
2. TRANSFER :bilingual dictionary lookup; word pattern processing (agreement, reordering, lexical changes) ;
3. GENERATION : morphological generation;postgeneration (rules for Catalan apostrophes and hyphens).

In particular, the design of the transfer module in *interNOSTRUM* is based on a careful black-box modeling of the transfer strategies of the MT systems mentioned above (for details, see Forcada (2000); Mira i Giménez and Forcada (1998)).

2 The transfer module

The transfer task is organized around *patterns* representing fixed-length sequences of source-language lexical forms (SLLFs); a sequence follows a certain pattern when it contains the corresponding sequence of lexical categories. The system contains a catalog of the patterns it knows how to process. Patterns are not "phrases" or constituents in the syntactic sense, because they are flat and unstructured, but pattern detection is an advancement with respect to mere morphological analysis and may be considered as a rudimentary form of syntax analysis, hence the name *advanced morphological transfer*.

The *pattern detection* phase occurs as follows: if the transfer module starts to process the i -th SLLF of the

text, l_i , it tries to match the sequence of SLLFs l_i, l_{i+1}, l_{i+2} ... with all of the patterns in its pattern catalog: the longest matching pattern is chosen, the matching sequence is processed (see below), and processing continues at SLLF l_{i+k} , where k is the length of the pattern just processed. If no pattern matches the sequence starting at SLLF l_i , it is translated as an isolated word and processing restarts at SLLF l_{i+1} (when no patterns are applicable, the systems resort to word-for-word translation). Note that each SLLF is processed only once: patterns do not overlap; hence, processing occurs left to right and in distinct "chunks".

Pattern processing takes the detected sequence of SLLFs and builds (using the bilingual dictionary lookup program) a sequence of TL lexical forms (TLLFs) which may be reordered, with LFs added to it or deleted from it. The inflection information in TLLFs is generated so that agreement is observed inside the sequence if necessary. For instance, the Spanish pattern article-adjective-noun (such as "una señal inequívoca" --an unmistakable signal--) is turned into an un reordered Catalan sequence ("un senyal inequívoc"), after propagating the Catalan masculine gender (was feminine in Spanish) of "senyal" to both the article and the adjective (in addition, the transfer module may maintain 'state' information to ensure interpattern relationships such as subject-verb number agreement. State information is updated after each pattern is processed).

Of course, a finite catalog of fixed-length "frozen" sequences cannot possibly cover all of the possible forms a certain constituent (i.e., a noun phrase) may take, but if patterns are chosen so that they cover the most common phenomena, one can obtain a reasonable quality, in particular when, as in the case of *interNOSTRUM*, source and target languages are syntactically similar. This design --favoring longest patterns-- allows the developers to build first a word-for-word system (which is the default mode of the transfer module unless a pattern is found) and add patterns incrementally with the confidence that, well chosen patterns can only lead to improvements in translation quality.

It has to be noted that one of the design choices in *interNOSTRUM* has been that all of the modules communicate with each other via text streams, for two reasons: first, for easier diagnosis during the development of the system, and second, because it is a natural choice when using a text-oriented operating system such as Linux (or, more generally, Unix); as a downside, this causes a small amount of reparsing at the input of each module. Even so, speeds obtained currently in *interNOSTRUM* exceed a thousand words per second in a regular 1998-model desktop PC. The transfer module, therefore, reads SLLFs from standard input and

writes TLLFs as standard output.

3 Specifying transfer modules

Instead of directly programming transfer modules in C or a similar language, we have designed a high-level language (with keywords in Catalan) which may be used directly by the linguist, after some training, to encode:

- the SLLF patterns that have to be detected for processing;
- the extraction of grammatical features such as the lemma, gender, or number of the SLLFs;
- the manipulation of SLLFs and the assembly of their translations (looked up in a bilingual dictionary), together with other internally-generated TLLFs (if necessary) to produce the target pattern.

3.1 Structure of the program

The program has two main sections: the declaration section and the rule section.

The declaration section: This section includes:

- The keyword *separa* followed by a regular expression defining the whitespace that may be found between lexical forms. In *interNOSTRUM* a module before actual analysis takes place encapsulates whitespace and RTF or HTML formatting material in double square brackets which are treated as a single space in all modules. This material will be used to reconstruct the format after generation.
- The keyword *lema* followed by a regular expression defining the part of the lexical form where the lemma is included (the lemma is assumed to be the first part in each lexical form). For example, “[A-Za-z] +” (in *interNOSTRUM* the form is too long to reproduce here because it contains all accented characters and punctuation marks, which are treated as words).
- The keyword *formlex* followed by a regular expression defines a generic lexical form. In *interNOSTRUM* it is the regular expression for a lemma followed by “[^/]+"/” (anything starting with a lemma and ending in “/” is a lexical form).
- The keyword *cua* followed by a regular expression defines the part of a lexical form coming after the lemma and the category. In *interNOSTRUM*, it is also “/”.
- A series of *lexical category declarations* consisting of the keyword *catlex*, an identifier, the sign := and a regular expression selecting those lexical forms that will be treated as a particular category. Note that the linguist may include any information from the lexical form to define a category; categories may be very coarse (e.g., all nouns) or very fine (e.g., only those determinants that are plural demonstratives).
- A series of *attribute declarations*, consisting of

the keyword *atribut*, an identifier, the sign := and a regular expression describing the possible values that may be found in a lexical form for a certain attribute (a morphological feature such as *gender* or *number*).

- A series of *state variable* declarations, consisting of the keyword *estat* and an identifier. State variables are used to transfer values of active attributes (see the *activa* keyword below) from one pattern to another.

The rule section: This section is a sequence of (free-form) *pattern-action* rules. Each rule has three parts:

- The definition of the pattern to be detected, consisting of the keyword *detecta* and the sequence of lexical categories previously declared with *catlex*. Note that, if input is matched by two different rules, first, the longest one will be used and, second, for rules of the same length, the rule defined first prevails.
- A declaration of those attributes which are relevant for that particular rule and therefore have to be activated, that is, extracted from LFs. Attribute names follow the keyword *activa*. In addition to attributes defined with *atribut*, categories defined with *catlex* are also extracted from lexical forms and may be manipulated.
- Finally, the action to be performed to the pattern, enclosed in curly brackets, and written using the rules in the following section.

3.2 The language of actions

Actions are sequences of statements specified in a high-level language, reminiscent of C, which may be specified by the following grammar:

```
Stats → Stat ;
Stats → Stats Stat
Stat → { Stats }
Stat → si ( Cond ) Stat
Stat → si ( Cond ) Stat altrament Stat
Stat → envia Expr
Stat → $ num .orig. id := Expr
Stat → $ num .meta. id := Expr
Stat → id := Expr
Stat →
Cond → Cond o Conj
Cond → Conj
Conj → Conj i SimCon
Conj → SimCon
SimCon → ( Cond )
SimCon → no SimCon
SimCon → Expr compop Expr
Expr → Expr SimplExp
```

```

Expr → SimplExp
SimplExp → net SimplExp
SimplExp → $ num .orig. id
SimplExp → $ num .meta. id
SimplExp → & num
SimplExp → string
SimplExp → id

```

where: `si (...) ... altrament ...` is the conditional statement; `$` followed by a positive number i (**num**) represents the i -th SLLF in the current pattern; the symbol `&` followed by a positive number i (**num**) represents the whitespace between the i -th and the $(i+1)$ -th lexical form; the operator `:=` assigns a value to a state variable **id** or to an attribute; `envia (send)` sends an expression to standard output; **compop** stands for any of the string comparison operators `==` (equal) or `!=` (not equal); the operators `no`, `i`, and `o` are *not*, *and* and *or* respectively; **string** is any double-quoted string, and `.orig.` and `.meta.` are used respectively to select the information in source language and target language lexical forms. For example, the construct `$2.orig.gen` would refer to the attribute `gen` of the second SLLF in the pattern whereas `$2.meta.gen` would refer to the same attribute of the TLLF obtained by looking up the bilingual dictionary (bilingual dictionary lookup is implicit in all actions). String concatenation is achieved simply by writing string expressions next to each other. An identifier in a string expression represents the contents of a state variable.

3.3 An example

The following is an example of a file containing a single rule for the agreement of nouns and determinants.

```

# DECLARATIONS
# spaces, returns, and anything in
# [[ ... ]] are whitespace.

separa :=
"([\n\t]|\\[[^\\]]*\\])+";

# Nouns contain the string "<n>"
# followed optionally by "<acr>"
catlex noun := "<n>(<acr>)?";

# Determiners may be subcategorized
# <det><def>, <det><ind>, etc.
catlex det :=
"<det>(<def>|<ind>|<dem>|<pos>)";

# We are interested
# in gender and number
atribut gen :=
{"<m>","<f>","<mf>","<GD>"};
atribut nbr :=

```

```

{"<sg>","<pl>","<sp>","<ND>"};
# GD stands for "target gender to be
# determined"; ND stands for "target
# number to be determined"

# RULES

# A single rule to agree number and
# gender in determinant-noun SNs
detecta det noun
{
# Extract gender and number
# and category strings
activa gen nbr det noun ;

# Do something only if both parts agree
# (that is, they form a source SN)
si ( ( ( ($1.orig.gen!="<mf>")
i ($2.orig.gen!="<mf>")
i ($1.orig.gen==$2.orig.gen) )
o
( ($1.orig.gen=="<mf>")
o ($2.orig.gen=="<mf>") ) )
i ( ( ($1.orig.nbr!="<sp>")
i ($2.orig.nbr!="<sp>")
i ($1.orig.nbr==$2.orig.nbr) )
o
( ($1.orig.nbr=="<sp>")
o ($2.orig.nbr=="<sp>") ) ) )

# Do something only if gender or
# number change from TL to SL

si ( ($1.orig.gen!=$1.meta.gen)
o ($2.orig.gen!=$2.meta.gen)
o ($1.orig.nbr!=$1.orig.nbr)
o ($2.orig.nbr!=$2.orig.nbr) )

# If noun gender has to be
# determined, take it from
# determinant, if determinant
# gender is defined. If not,
# make it masculine. If both are
# undefined, make both masculine.
si ($2.meta.gen=="<GD>")
{
si ( ($1.meta.gen!="<mf>")
i ($1.meta.gen!="<GD>") )
$2.meta.gen:=$1.meta.gen
altrament
{
$2.meta.gen:="<m>";
si ($1.meta.gen=="<GD>")
$1.meta.gen:="<m>"
}
}

# If not, do the agreement by
# transferring gender from the noun
# to the article if both are not
# ambiguous.

```

```

altrament
{
  si ($2.meta.gen!="<mf>")
  {
    si ($1.meta.gen!="<mf>")
      $1.meta.gen:=$2.meta.gen
    altrament
      si ($1.meta.gen=="<GD>")
        $1.meta.gen:="<m>"
  }
};

# Now, number. If target number is
# undefined for the noun but it is
# defined for the determinant, take
# it from there; otherwise
# make it singular.
si ($2.meta.nbr=="<ND>")
{
  si ($1.meta.nbr!="<sp>")
    $2.meta.nbr:=$1.meta.nbr
  altrament
    $2.meta.nbr:="<sg>"
}
# If both are defined, transfer
# number from noun to article.
altrament
{
  si ($2.meta.nbr!="<sp>")
    si ($1.meta.nbr!="<sp>")
      $1.meta.nbr:=$2.meta.nbr
};

# Write target pattern
# First the determiner, followed by the
# intervening mark
envia
$1.meta.lem $1.meta.det $1.meta.gen
$1.meta.nbr &l;
# Then the noun
envia
$2.meta.lem $2.meta.noun $2.meta.gen
$2.meta.nbr;

```

4 The compiler

The MorphTrans compiler has been developed under Linux using the classical compiler-construction tools yacc (bison) and lex (flex). It takes a source MorphTrans file (.trf) and writes a lex file which is subsequently compiled into a C file and linked it to the bilingual dictionary lookup program to produce an executable program. The lex program is constructed so that each pattern is converted into a regular expression which is matched and then, as part of the associated action, segmented in lexical forms and whitespace and searched for attributes using their regular definitions, and manipulated according to the code written by the linguist, conveniently converted to C. The bilingual dictionary lookup program has to be provided as a C function with prototype

```
char * Bilingue( char * ) ;
```

taking a SLLF and returning a single TLLF which is its designated equivalent for the target language.

5 Concluding remarks

We have presented a language and a compiler that may be used to specify and implement the transfer task in an advanced morphological transfer machine translation system, provided that lexical forms produced in the analysis phase (morphological analysis and part-of-speech tagging) are provided as strings. The resulting module looks up the lemmas in a bilingual dictionary, detects patterns of source-language lexical forms and manipulates them so that they are converted into the corresponding sequences of target-language lexical form. The language and the compiler are flexible enough to be used in any system and for any source and target languages, as long as lexical forms are given as text and have the form *lemma-category-inflection information*. We have found the resulting modules to be capable of processing lexical forms at a rate of about a thousand lexical forms per second for about two dozen noun-phrase and prepositional-phrase rules; time complexity has two components; pattern detection time increases very slowly with the number of rules, and the time actually used by this module depends strongly on the total coverage of the rules defined, that is, on how often the code associated to each pattern has to be actually executed because the pattern has been matched.

Acknowledgements: This work is part of project *interNOSTRUM*, which is funded by the Caja de Ahorros del Mediterraneo and the Vice-Rectorate for Information Technologies of the Universitat d'Alacant; partial funding from the Spanish Comisión Interministerial de Ciencia y Tecnología through project TIC2001-1599-C02-02. We also thank Anna Esteve, a linguist in our project, for her feedback during the design of this module.

References

- Arnold, D. (1993). Sur la conception du transfert. In Bouillon, P. and Clas, A., editors, *La traductique*, pages 64-76. Presses Univ. Montréal, Montréal.
- Arnold, D., Balkan, L., Meijer, S., Humphreys, R., and Sadler, L. (1994). *Machine Translation: An Introductory Guide*. NCC Blackwell, Oxford.
- Forcada, M.L. (2000). Learning machine translation strategies using commercial systems: discovering word-reordering rules. In *MT2000: Machine Translation and Multilingual Applications in the New Millennium (Exeter, UK, November 18-20, 2000)*, pages 7.1-7.8.

Hutchins, W. and Somers, H. (1992). *An Introduction to Machine Translation*. Academic Press.

Mira i Giménez, M. and Forcada, M.L. (1998).
Understanding PC-based machine translation systems
for evaluation, teaching and reverse engineering: the
treatment of noun phrases in Power Translator.
*Machine Translation Review (British Computer
Society)*, 7:20–27. (available at
<http://www.dlsi.ua.es/~mlf/mtr98.ps.Z>
).