Chapter IV

IMPLEMENTATION ASPECTS AND FURTHER RESEARCH

4.1 Implementation Aspects

4.1.1 Program Model

The program model that follows is to be implemented using the algorithm of C++ following the Object Oriented Approach. The point which must be made clear at this moment is that this Lexical Semantic Component in itself can not function as a Machine Translation System. It can not translate in isolation. For that purpose it has to be attached with a compatible parser and a generator.

The flow diagram given on the next page shows the proposed program model for the lexical semantic component containing the RLCSs, a bilingual dictionary for the share-market and a derivational mechanism to derive Coposed Lexical Conceptual Structures (CLCSs)
IN

WORD

LEXICAL SEMANTIC COMPONENT

DOMAIN R.L.C.S.s

DERIVATIONAL MECHANISM FOR DERIVING C.L.C.S.s

BILINGUAL DICTIONARY

OUT

C.L.C.S.s

THE PROGRAM MODEL
As has been clearly stated earlier, this LSC has to be combined with a parser and a generator for being able to translate in the given domain, or to give domain competence to any LCS based interlingual MTS. Keeping this fact in mind, the model above will receive parsed constituents from a parser. These terminal lexical items will be mapped in the bilingual dictionary to obtain the necessary information for the target language (Hindi). In addition, the verbs will be mapped to their LCS templates from the component contained in the component labeled ‘RLCSs’. This component consists of Root Lexical Conceptual Structures (RLCSs) which have two levels of information

-the basic conceptual structure which is language independent, and

-the notational marks for argument binding etc.

The notation a, b, and c are used for argument binding.

The RLCSs component will produce the conceptual structure which will be composed into CLCSs by the CLCS component. Finally the lexical selection process will yield the CLCS for the target language. This CLCS may have to be properly re-ordered with the help of a generator.
4.1.2 Location of LSC in the MTS

There can be two possible applications of the LSC, and hence its two different locations within a Machine Translation System:

1) If the LSC is intended to be used as a part of an MTS which will translate the language of share-market only (English to Hindi), then the present LSC can itself function as an interlingua after the RLCSs are composed into CLCSs by the derivational routine. In that case the technical language of share-market in a very limited domain will be translated. The diagram on next page will make it more clear
Using Domain LSC for translating from English to Hindi
In this case our requirement would be an English parser, and a simple Hindi generator, and not a UNITRAN type principle based parser and generator.

Procedure for translation:

The complete MTS based on this approach will also include X-bar compatible parser for English and generator for Hindi. The parser will feed the X-bar template with source language terminals into the LCS based domain Lexical Semantic Component (LSC) for lexical semantic processing. Within the LSC the following operations will take place in the given order:

- appropriate LCS selection
- composition of CLCS
- lexical selection

After the resultant CLCS from the LSC the target language generator takes over, and the syntactic realization for the target language is done. The use of the LCS as the guiding principle will ensure future extendibility and upgradability of the system even to include other specialized domains of language use.
2) If however, our share-market Lexical Semantic Component is intended to grant domain competence to any general purpose interlingual multilingual translation system, then the present LSC will only provide the domain knowledge thereby making the MTS function effectively in this domain also. In that case the model of the MTS will look like (diagram on next page)
Using share-market LSC to grant domain competence to any general purpose Interlingual MTS
In this case, the choice for a UNITRAN type parser and generator would be justified.

The goal of the present research at the moment is as shown in option (1), and the option (2) shows its potential application and also its augmentability and extendibility.

4.1.3 Choice of Algorithm

The algorithm proposed to be used for implementation of the formalization is C++ which incorporates the advantages of Object Oriented Programming Approach (OOPA). C++ implies a double incremental forward compatible extension of C language which is in widespread use in computer systems of all sizes and applications. However, at this point it may be just in order that C++ is one more addition to the list of languages using OOPA principles. The important ones among them are:

- Simula
- Smalltalk
- C++
- Turbo Pascal
Simula is considered to be the ancestor of object oriented programming languages. The concept of object as encapsulation of data and operations, as well as the class and inheritance concept were realized in Simula for the first time in a universal programming language.

As a result of the publication of the Smalltalk 80-system, object-oriented concepts became known to more and more people. Besides objects, classes and inheritance, Smalltalk provides further concepts like object-identity, message and polymorphism. An important aim for the development of the Smalltalk system was to provide a user friendly uniform and flexible interface for the end user. The Smalltalk system is supposed to support the end users to acquire a "personal assistant" for themselves for organizing and managing their daily work without a detailed knowledge of the system and to be able to develop further dynamically. This aim also became the starting point for the development of further object oriented (end-user) systems.

C++ is a forward-compatible extension of C. The extension contains elements, independent of OOP as well as elements which support OOP\(^1\). The extensions of the first kind include the marking of comment lines "//", the specification of default parameters for functions, the declaration of constants and the definition of inline functions ("macros").

The starting point for the development of C++ was to achieve a pro-gramming language with which big, efficient simulations programs could be realized. The available languages like Simula, Smalltalk proved to be too in-efficient for this purpose. C was already an effi-cient system programming language (procedural), and thus it was selected as a basis for the implementation of object oriented concepts.

Turbo Pascal is a version of Pascal which is mainly used on home and personal computers. Version 5.5 onwards, Turbo Pascal is equipped with object oriented features, so that object orientation has also found and entry in this area and is increasingly being used.

Among the chief concepts that form the theory of OOPA are included those of

-Object
-Class and instance
-Inheritance

From the object oriented viewpoint these are considered to be funda-mental.
Objects:

Objects are atomic units which communicate with one another with the help of messages. Contrary to the procedural point of view which considers the software to be processed and actions (procedures), which carry out the processing, the object oriented viewpoint recognizes only one unit i.e. the object which includes date and procedures. An object can be changed like data in programs of procedural languages. The changes can however be done by object’s own procedures, also called methods in object oriented languages.

Object orientation demands a strict maintenance of the module or the concept of the data encapsulation in its strictest form. An object does not permit other objects to access its data and operations directly. The services of an object can only be called upon through messages. The services are realized by the methods of an object. Objects thus encapsulate their state and their behaviour to make an atomic unit.

The structure of methods in the object oriented programming languages, and that of procedures in procedural languages is almost similar, except for a few minor syntactical difference depending upon the language used. An important difference is that methods cannot call themselves mutually like procedures do, and that the methods can change at the most local data i.e. the data of the object in which they
are present. Hence, methods can not be handled in isolation, but only through the objects.

Objects possess three essential features:

- An object is a copy (an 'instance') of a class, with that it possesses a time variant 'type'.

- Every object possesses a (time variant) 'state', which is determined by the current values of its state variable.

- Every object possesses a unique (time variant) 'identity'.

In most of the object oriented programming languages and application systems, the qualities of the objects generated deviate from the ones stated here.

Classes and instances:

An object oriented system comprises of a number of objects, which communicate with one another. Such objects can be elementary objects like numbers, boolean values, strings or compound data struc-
tures like lists, trees, geometric figures as well as complex tools like editors, compilers or database systems. The number of objects existing in an object oriented system is normally very high. It would be too troublesome to describe each and every object i.e. its status variable and its behaviour.

As many objects stand out because of their structural and functional similarity i.e. by having the same features and same behaviour e.g. whole numbers and rectangles, it makes sense to put together these objects in classes and to determine their common features and behaviour.

A class definition includes

-the name of the class

-the interface through which the services of the objects of this class can be demanded

-the definition of the status variables whose values represent respectively the state of objects of this class

-the implementation of the methods which realize the services of the objects of the objects of this class.
The interface can also be subdivided into three parts:

-the public part is the part which is visible to all the users who may require the services of the objects of this class

-the protected part is visible only to the objects of the subclasses of this class

-the private part of the interface is not visible from any other class. These services can only be demanded by the objects of this class.

Variables can be of two types:

- class variables, and

- instance variables

Class variables are variables whose values describe the quality of class. Instance variable describe the state of the instances of a class.

From the class definition given until now, it is clear that the variable definition is comparable to those in the conventional, procedural programming languages. From the available standard types (integer, real,
char, string, bool, ...), the structured types can be formed with the help of constructors (record, set, ...). In general, the values of objects can be complex values or objects (through base value ranges, attributes and object names formed with the help of corresponding tuple and series constructors.

A class definition thus builds up a scheme, which specifies structural pattern for the objects of this class, objects with the same status variables and the same behaviour, objects of class are also called instances of this class. The objects of class are generated by the generation message to the class:

new(class);

generates an instance of the class ‘class’. In general, the instances get a name or assigned to variables as values upon their generation:

object:= new(class);

The objects thus generated possess an identity, but still no definite status. In order to achieve a definite status, values must be assigned to the status variables. This can be done by the corresponding (initializations) messages.
In case an object does not remain a value of a variable any more, and also not an element of a series object or component of a tuple-object, it becomes useless (‘garbage’), and it can not be accessed any more. There are two ways of deleting such objects and releasing the memory occupied by them:

- by controlled release: the objects are deleted by a command to be stated explicitly in the program, or by a corresponding (system) message

\[ \text{dispose}(X) \]

before a new object is allocated to \( X \). This corresponds to the controlled release of dynamically generated memory by the procedure ‘dispose’ in the procedural language Pascal.

An advantage of controlled release is that it is done upon the request of the programmer and thus run time environment is troubled only a little with the release.

A disadvantage is that the programmer may possible to delete all the references to the object to be deleted. This can lead to the so-called "hanging" references i.e. references to objects already deleted.
-by automatic release: Objects to which no references are pointing are automatically deleted by run-time environment ('garbage-collector'). This type of memory release is done in the run-time environment of Lisp and Smalltalk.

The advantage is that no hanging references can appear. A disadvantage is that run-time can become extremely large as a result of this 'garbage removal', as must be checked for every object, whether it is value of a variable or component of a tuple-object, or an element of a series object.

Classes can also be considered as extensions of their class definition, i.e. a kind of container which contains the currently generated objects and instances not deleted as yet. Instances of a class are generated by sending the message 'new' to the class. Extensions permit the explicit insertion of instances of this class in the class or the explicit deletion of instances from the class.

Classes as extensions can themselves be objects, to which besides the generation message for producing new objects, other messages can be sent which lead to an execution of the class methods and a change of class variables. None of the known OOP languages explicitly support abstract data types or classes as extensions. However, Smalltalk has the possibility of defining so called collection objects, which in many
cases possess a functionality comparable to that of extensions.

**Example from C++ (classes & objects):**

A class definition has the following structure in C++

```
Class class
{
    private:
        private variables and functions
    protected:
        subclasses of common variables and functions
    public:
        public variables and functions

};
```

// implemented as shown in the following example:
Class Date
{
private:
    int Day, Month, Year;

public:
    Date();
    void Set (int, int, int);
    void print();
};

//implementation of the functions

Date :: Date()
{
    Day = 0;
    Month = 0;
    Year = 0;
};
The class ‘Date’ defines objects, which contain date specification as inner non-accessible state. The access to these objects i.e. to their state, is possible only through the public methods Set and Print.

Inheritance:

Inheritance permits the realization of new software modules (class) on
the basis of an already given hierarchy of modules without having to
design the complete system again. A new class can take up the be-
haviour as well as the structure of existing classes (inherit), without
having to re-design and re-implement this behaviour and structure for
the new class. Only the special behavioral and structural features of
the new class have to be supplemented. Software once developed can
thus be re-used easily on one hand, and on the other it can be easily
extended. Hence the concept of inheritance essentially supports the
software qualities reusability and extendibility.

The origin of inheritance can be found in considerations made within
the framework of Artificial Intelligence for knowledge representation
and structuring. The concept here is to describe knowledge by means
of terms with a taxonomical hierarchy.

The concept of inheritance thus helps to structure software systems,
enables mode sharing - status variables and methods can be used by
many classes - and supports the reusability of software designs and
implementations as also the extendibility of the existing software.
Together with other programming concepts e.g. data encapsulation or
type concepts, the concept of inheritance leads to problems which can
be solved in various ways.
Example from C++ (inheritance):

The definition of a subclass (derived class) for an already defined class (base class) is done in C++ as follows:

```cpp
class DerivedClass: [public | private] Baseclass
{
  variables and functions of DerivedClass
}
```

Only the variable and methods of the base class defined as public are inherited by the derived class. If public is specified these are then also defined as public in the derived class. If private (or nothing at all) is given these are defined as private in the derived class.

Let the class person be defined

```cpp
class person
{
  public:
```
char Name[3]
char Address[3]
int Age;

}

The derived class Student is defined as:

class Student : public Person
{
    public:
    char Subject[3];
}

Following commands (messages) generate the Student object Ram
and assign values to its variables.

Student X;

strcpy (X.Name, "xyz");
strcpy (X.Address, "abc");
X.Age = 25;
strcpy (X.Subject, "uvw");

Objects of derived classes can also be generated and initialized like the objects of the base classes. The following class definition shows an extension of the above given definition of the class ‘person’ and ‘student’ by the constructors:

class Person
{
    public:

    char Name[3];
    char Address[3];
    int Age;

    Person (char * N, char * Adr, int A);

}
Person (char * N, char * Adr, int A)
class student : public person
{

public:

char Subject[3]
student(int M, char *N, char *n, char *Adr, int A):(N,Adr,A)

}

student(char *N, char *n, char *Adr, int A): (N,Adr,A)

subject = s;

}

Upon the call of the constructor,
student xyz (Initialization Student):(Initialization Person)

for the generation of the student - object xyz, first the Person constructor is called in order to generate the instance variables inherited by person and to initialize them by the values defined in initialization person. Thereafter, the variables declared in Student are generated with the help of the student constructor and initialized with the values definition Initialization student.

4.2 Future Research

Future researches in this area should be encouraged on the following lines:

-to design similar LSCs for other technical and high application domains so that they can be combined into a single module for making the system translate in other domains also. Using the LCS formalism ensures future extendibility and upgradability of
the system.

-to include other target languages also on priority basis,

-since the LCS-interlingual mechanism has the benefit of bi-directional multi-lingual translations, the effort should be to include other source languages than only English. In that case, a real multi-lingual translation system for Indian languages could be achieved.