How Oberon's Type Extension Influences Other Programming Languages

Oberon Ilano
Computer Science and Information Systems
University of North Alabama
Florence Alabama USA
oilano@una.edu

ABSTRACTION

There are several purposes for this research paper, one of the reasons for this research paper is to learn and understand the Oberon programming language and its type extension. This paper will highlight some of its historical background, language overview, literature review, reasoning of why the Oberon programming language was created, and the future work that involves Oberon. There are a few programming languages that are influenced by Oberon. Therefore, another reason for this research paper is to find out how the Oberon's type extension influenced other languages or find out if Oberon's type extension has even made an impact on these other languages. Also, in the methodology section of this research paper, I will be showing and comparing some examples of Oberon's type-extension and other languages that have been influenced by this programming language for the validity of answering the question.

1 INTRODUCTION

Why I choose the programming language Oberon? I had never heard of the programming language Oberon before other than the king of the fairies in "A Midsummer Night's Dream" by William Shakespeare, one of Uranus' moons, and Carl Maria von Weber Overture "Oberon," but I was immediately attracted to it because we share the same name. The fact that the name of the programming language was Oberon instantly sparked my interest and made me want to learn more about it. Oberon is an object-centered operating system and a programming language. Oberon was known to be evolved from Modula-2 and is a general-purpose programming language. Some programming languages that are known to be influenced by Oberon are Oberon-2, Object Oberon, Active Oberon, Oberon-07, Component Pascal, Zonnon, and GO. Oberon was not widely known or largely programming language, therefore not much research has been done on this language by researchers. One of the main concepts and added features is the type extension. Type extension allows adding new data types and procedures based on the existing type without terminating the current code. So, how does Oberon's type extension influence these other programming languages or does the Oberon's type extension even had an impact on these programming languages?

2 BACKGROUND

2.1 Historical Background

Oberon programming language first appeared in 1986 at ETH (Federal Institution Technology) in Zurich, Switzerland and was designed by Niklaus Wirth and Jurg Gutknecht. The initial ideas and draft of Oberon were first initiated in 1985 and the programming language was completely defined by the following year, 1986. Niklaus Wirth also designed both Modula-2 and Pascal. Modula-2 is a direct inheritor of the Pascal and along with Oberon, all these programming languages are part of the Algol family. It started with the ALGOL 60 then followed by ALGOL-W, and the ALGOL-Like begun with Pascal, then followed by Modula to Oberon, which is the last member of the family. At the time of Wirth's new project, the space probe Voyager was passing by one of Uranus' twenty-seven moons it's named was Oberon, Wirth was fascinated by the accuracy of the probe that he gave his new system and language the name Oberon [1]. Wirth took inspiration designing Oberon from Albert Einstein's quote "Make it as simple as possible, but not simpler." The result of that is Oberon, an operating system for workstations or personal computers and a programming language that was built from scratch [2].

The sole purpose of the Oberon language is to serve as an implementation tool for the Oberon system. Project Oberon is one of the operating systems that Oberon programming language supported. The language can be used without the system, but it is more powerful when both are used together. Wirth's idea was to create Oberon and make it a more powerful language with better complexity than Modula-2. Oberon follows the same tradition that both Pascal and Modula-2 have. The high demand for reducing the complexity of the program was because of the growth of the other programming languages such as Ada, C, and C++. Though Oberon was a smaller language, less syntax volume, and was not widely popular, it was known more to be sufficient for engineering in the

industry. At one point, Oberon was also thought to be a possible choice of programming language and platform for Macintosh and Sun PC. Some say that Oberon is an extension of Modula-2 with some newly added features and some removed old features. However, there were still major problems in Oberon that was inherited from Modula-2. Some features that were not included in Oberon that Modula-2 had are enumeration, subrange, variant records, cardinal types, pointers to basic types, clauses, local modules, arbitrary lower bounds for arrays, with-statements, and for-statements. These features were removed because they are unnecessary, sensitive facilities, syntactic reasonings, semantic construct, and the random use of enumerations led to the excessiveness of the functionality rather than the clearness of the quality. One of the basic ideas of developing the first Oberon program was to create a task that would generate random numbers, a program that throws dice, an algorithm that is capable of producing a very long sequence of numbers where no pattern becomes noticeable. Some functionality included in the first Oberon was the declaration of variables, the procedures, module, import, export, the client module, and a body that has statements between *BEGIN* and *END* delimiters.

2.2 Language Overview

Some of the important features of Oberon language are strong type checking, package or module with separate compilations, and type extension. Oberon can be considered as structured, imperative, modular, and object-oriented paradigms. It has a static type checking that will guarantee the understanding of the messages that are sent out to the object and are possible to be checked during run time. Oberon's type-extension will allow the construction of new data types based on the existing one and can relate to each other, but precisely the static data typing. Wirth implemented inheritance by based type hierarchies, which will be touched on the methodology section.

The syntax of Oberon is called compilation units or formal language which is a limited sequence of symbols from a limited vocabulary. The symbols' representation is expressed by using ASCII. The vocabulary consists of identifiers, operators, strings, numbers, delimiters, and comments. An Extended Backus-Naur Formalism or EBNF is used to describe the syntax. The braces { } indicate its repetition or zero if it is none. The brackets [ ] indicate the optionality of the surrounded sentential form. If a construct *X* may be either *Y* or nothing, this is implied by *X =[Y].* If a construct *X* that contains the concatenation of any number of *Y*s, this is implied by *X = {Y}.* The syntactic factors and terms of Oberon are very similar to most programming languages. An example of a syntactic factor is *X=YZ* where *X* is the syntactic formula and both *Y* and *Z* are the syntactic factors. An example of syntactic terms is denoted by *X=Y|Z* which means that *X* is composed of a factor *Y* or a *Z*.

Tokens are taken from the formal language's vocabulary are substitute elements and an intermediate level of representation by symbol sequences. There are five classes that tokens of the vocabulary are divided into including identifiers, numbers, character constants, strings, operators, and delimiters. An identifier consists of a sequence of letters and digits [2]. The first character of the identifier must be a letter. Strings can be compared and assigned with arrays of characters and like most programming languages a string should be enclosed by quotation marks. The character constants can denote by enclosing quotation marks or it can be specifically specified by its standard ASCII code number in a hexadecimal notation followed by the *X* letter at the end. Some examples of character constants are *"a", "2", "!"*, and "*61X*." The "*61X* "and *"a"* denotes the same character. Any printable characters from the keyboard that are not letters, digits, or blank are considered special characters. It is case sensitive syntax with uppercase keywords. The numbers in Oberon are either integers or real numbers, and they are both unsigned. The hexadecimal integers start with a digit, followed by a sequence of hexadecimal digits that include the letters *A, B, C, D, E, or F,* and followed by the suffix *H*. The letter *E* or D can be attached as the prefix in the scale factor, but it is optional. Consider 500 as an example of an integer which is equivalent to *1F4* as a hexadecimal notation and expressed as *1F4H* in Oberon. The types of real numbers that are considered with the decimal point are *REAL* or *LONGREAL.*

There are eight basic types in Oberon, and they are predeclared identifiers including one of the newly added features in Oberon is the *type inclusion* that includes five numeric types such as *LONGREAL, REAL, LONGINT, INTEGER, and SHORTINT,* which three are integer types. The concepts of *type inclusion* bind all arithmetic types together. As Wirth explained on his Modula-2 and Oberon, it indicates that values of the included type can be allocated to variables of the including type

var *i: integer; k: longint; x: real*

assignments *k:=i* and *x:=i* are legal, whereas *i:=k* and *k:=x* is not [3]. This example shows that a smaller type is compatible with the larger type with automatic conversion of internal representation. Another reason for adding the new feature was for economical storage issues that the processor that was being implemented does feature the instruction sets for all arithmetic types that Oberon has. The other three basic types in Oberon are *SET, CHAR, and BOOLEAN.* Two different structured types that are considered in Oberon, which are records and arrays. These structures have different components including *ArrayType, RecordType, PointerType, ProcedureType,* and *qualident.* The array types have a property similar to other programming languages. The declaration of an array is slightly different as shown below.

```
ArrayType = ARRAY length {"," length} OF type.
        x: ARRAY 10 OF REAL
        y: ARRAY 8 OF
           RECORD ch: CHAR;
                  count: INTEGER
           END
```

In some programming languages, the length of the array is declared inside the brackets or braces. The record types are

extensible and can be expressed as an extension of another record type. Its properties have a set number of elements, which may have different types. The *field* and an identifier that represents the *field* are specified during the declaration for each element. The *field* identifiers can visible outside the declaring module if the record type is exported and must be marked with "*" indicates that they are *public fields*, which means that the *private fields* are the one that is unmarked fields. Examples of declaring record types are shown below.

```
RECORD
   age, height, weight: INTEGER
END
RECORD
  firstname, lastname: ARRAY 64 OF CHAR;
  id: INTEGER;
  salary: REAL
END
```

The pointer types in Oberon get the extension relativeness of their base types. The variable of type is assigned in free storage, and a pointer to it is assigned to the variable. Pointers are declared by

PointerType = POINTER TO TYPE

value of *NIL* will be the result if the allocation failed. The procedure types can be declared as

ProcedureType = PROCEDURE[FormalParameters]

and the procedure type's variable has a value of *NIL* or a procedure. The procedure can't be a predefined procedure, can't be declared locally to another procedure, and the same result type if it's a function procedure.

Every Object in Oberon such as variables, procedures, types, and constant need to be declared. In Oberon, an *export mark* denoted by * implies that the identifier is visible outside the module including the declaration which means that it may be used in other modules by declaring the module. The variables defined by variable declaration have two properties which are the type and the value. Declaring a variable is very similar to other programming languages, where the same type can be grouped in the same declaration, a sequence of variables can be separated by commas, and a sequence of variable declarations are separated by semicolons as an example:

```
VAR
i, j, k: INTEGER; index: LONGINT;
x, y, time*: REAL;
ch: CHAR;
```

The expressions in Oberon are evaluated in a similar passion as other languages from left to right and parenthesis is used to alter the order of its evaluation. Another newly added feature of Oberon is named *strong typing,* which checks the compatibility requirements, syntax, and semantic of expression and an assignment. *Strong typing* help evaluates large numbers of programming errors through the compiler. The expression consists of operators and operands. The operators are evaluated by their precedence. The semantics or the meaning of an expression is very easy and simple to understand as shown in table 2 below

Table 2 The Symbols, Relation, and operand types

Symbols	***Semantic/Relation***	***Valid operand types***
>	*Greater than*	*Numeric types, CHAR, ARRAY OF CHAR, String constant*
<	*Less than*	*Numeric types, CHAR, ARRAY OF CHAR, String constant*
>=	*Greater than or equal*	*Numeric types, CHAR, ARRAY OF CHAR, String constant*
<=	*Less than or equal*	*Numeric types, CHAR, ARRAY OF CHAR, string constant*
=	*Equal*	*Numeric types, CHAR, ARRAY OF CHAR, String constant*
#	*Not equal*	*Numeric types, CHAR, ARRAY OF CHAR, String constant*
IN	*A set membership*	*left: integer type* *right: set*
IS	*A type set*	*left: POINTER, RECORD* *right: type identifier*

with all the result types of *BOOLEAN.* This is probably the first time I have seen a *#* being used for relation because most other languages use either <>, or *!=*, which is interesting. In Oberon the reserved word *IN* is used as *s IN S* in another word, s must be an integer type and *S* is a *SET* type which means that *s* is an element of *S.* The reserved word *IS, is* a type test and is used as t *IS T*, which means *t is* of type *T.*

The arithmetic expression is very similar to other modern languages, a "*+*" for addition, a "*-*" for subtraction, a "***" for multiplication, a "*/*" for real division, "MOD" for modulus, with the exception for "*DIV*" which is used for integer division only. The Boolean expression is a little different from what other programming language uses except for "*OR*" which logically means of disjunction. In Oberon, uses a single "&" as a symbol or syntax with semantically and logically meaning of conjunction whereas other languages use double "&&" or "AND" with the same semantic meaning. Also, the symbol or syntax "~" which semantically means "not", which other programming languages use the symbol of "!" with similar meaning.

The assignment statement in Oberon is implemented with ":=" similarly with GO programming language and its syntax as [2]:

```
Assignment = designator ":=" expression.
        i := i + 1;
```

One of the actions of the assignment statements is the evaluation of the designator. Two things to be considered with the compatibility of the type of designator and the type of expression. Number one, the arithmetic types the designator must be larger type than the expression type and the larger type must able to include the expression type. A statement is a basic unit of action and a statement sequence can be expressed in Oberon. Semicolons can be used as the statement separator such S_1; S_2; ... S_i; ... S_n. *HALT(x)* is an expression that can be called where *x* is an argument with an integer value that identifies the end of the statement sequence. The procedure consists of parameters or arguments. The parameters are either a value parameter or variables. Some of the predeclared

procedures or special assignment statements can be used including the following in table 3 below.

Table 3 Predefined Procedures, argument types, and their meaning

Predefined Proper and Function Procedures	***Expression and Argument Type***	***Function/Function Return***
ABS	*Numeric type*	*Absolute value*
ASH	*ASH(b, n)* *b and n are integer types.*	*Arithmetic shift b *)*
ASSERT	*ASSERT(b)* *ASSERT(b, n)* *b is a Boolean expression and n is an integer constant.*	*It will terminate the program if not b.*
CAP	*CAP(b)* *b is a CHAR type.*	*Similar to the other languages' function ToUpper of which b is a letter corresponding to its capital letter.*
COPY	*COPY(b, a)* *b is a character array or string and a is the character array.*	*a is the designator of b, similar to a := b.*
DEC	*DEC(i)* *i is an integer type.*	*Decrement i by 1 similar to* *i := i -1*
ENTIER	*ENTIER(b)* *b is a real type*	*The largest integer but is not greater than b.*
EXCL	*EXCL(b, a)* *b is a SET and a is an integer type.*	*Exclude a from the SET b similar to b := b – {a}.*
INC	*INC(i)* *INC(i, j)* *i and j are integer types.*	*Increment i by 1 similar to* *i := i + 1 and* *i := i + j.*
INCL	*INCL(b, c)* *b is a SET and c is an integer type.*	*Include c to the SET b similar to b := b + {c}.*
LEN	*LEN (b)* *LEN(b, n)* *b is an array and c is an integer constant.*	*The length of array b in element n. The length of array b in element 0 similar to LEN(b, 0).*
LONG	*LONG(b)* *b is LONGREAL, INTEGER or LONGINT type.*	*Specify the identity of a numeric type.*
MAX	*MAX(S)* *S is a basic type or a SET.*	*Return the maximum value of type S or return the maximum element if a SET S.*
MIN	*MIN(S)* *S is a basic type or a SET.*	*Return the minimum value of type S or return the minimum element if a SET S.*
NEW	*NEW(b)* *NEW(v, …)* *b is a pointer to open array and e is an integer type.*	*Assign b with lengths .*
ODD	*ODD(b)* *b is an integer type.*	*Return true or false if b is an odd number.*
ORD	*ORD(c)* *c is a CHAR type.*	*Returns the ordinal number of c.*
SHORT	*SHORT(b)* *b is REAL, INTEGER or SHORTINT.*	*Specify the identity of a numeric type.*
SIZE	*SIZE(x)*	*Return the size of x.*

The comments are implemented by inserting it between the (* and *) and the semantic of the program does not get affected by it.

The Control Structures in Oberon are sequences of actions that are structured statements with a recursive definition that denotes selections, conditional executions, or repetition of statements. *The If-statements* and the *Case statements* are the *conditional statements* in Oberon. The *Repetitive statements* in Oberon consist of the *Repeat statements*, the *Loop statements,* and the *While statements*. The expression of the if-statement has a similarity with GO [4] with no parenthesis needed to surround the conditional statements and it is under a guard that only can be implemented with Boolean expressions.

```
IF x = 0 THEN Out.String("Class C")
ELSIF x = 1 THEN Out.String("Class A")
ELSIF x =2 THEN Out.String("Class B")
ELSE Out.String("Class D")
END;
```

The case statement in Oberon reminds me of the switch statement in modern languages they are very similar. The case labels are constant expressions or constants with the same type, which is either CHAR or integer type and no repetition value is allowed. An example below can be compared to the if-statement above paragraph.

```
CASE x OF
    0: Out.String("Class C")
  | 1: Out.String("Class A")
  | 2: Out.String("Class B")
ELSE Out.String("Class D")
END;
```

The while statements consist of a counting loop and a control variable. The meaning of while statements are similar to loop and repeat statements except repeat statement has termination condition is checked after execution of the statement sequence, does not use "END" with the statements, and is not guarded by an explicit condition. Therefore, the while-statement is a better use than the repeat statement. The loop statement uses the reserved word "EXIT" for terminating the loop. The following three examples below can be compared.

```
While statement example:
    a := 0;
    WHILE a < n DO
        a := a + 1
    END;
Repeat statement example:
    a := 0;
    REPEAT
        a := a + 1
    UNTIL a = n;
Loop statement example:
    LOOP
        IF a # n THEN
            a := a + 1
        ELSE EXIT END;
    END;
```

As we compared the three different repetitive statements, they all different syntactically but the essence of them is all the same.

The modules or some know it as packages are a sequence of statements with a collection of procedures, and declaration of constants, variables, or types. The scope of the module is a static that is considered not local and the variables are considered *global variables*. The *global variables or global objects* are visible outside the module, marked with "*" which means that they can be exported and imported to other modules. The modules are not allowed in the nested loop. Two types of procedures in Oberon are the function procedure and proper procedure. The function procedure has a *RETURN* clause and the proper procedure is identified through its heading and activated by a procedure call. The meaning of locality in both module and procedure conditions that the declarations are only local to their scope. Recursion can be implemented by using a function procedure and one great example is the factorial, Fibonacci sequence as explained in Reiser's and Wirth's "Programming in Oberon" [2].

```
PROCEDURE fact(n: INTEGER): LONGINT;
VAR fact: LONGINT
BEGIN fact := 1
   WHILE n > 0 DO
      fact:=fact * n;  DEC(n)
   END;
   RETURN fact
END fact;
```

In Oberon, the input and output streams were not properly included. One of the reasons is that they believed that the system that will run Oberon has a packaged that does the operations for the abstraction of input and output.

Oberon is object-oriented programming (OOP) paradigm, similar to Smalltalk, which is not purely a language but an entire system. The standard OOP terminology supports include Class, Object, Method, Message, Sub-class, Super-class, Inheritance, Overriding a method, Self, Super call, and Dynamic Binding. The object-orientation is utilized by using a type extension and module hierarchy. The meaning of OOP in Oberon including heterogeneous data structure, generic module, module structure, dynamic binding of procedures, the object and its representation. The heterogeneous structure in Oberon is an object. The objects are records with procedure fields or also known as functions or methods that can be accessed by a pointer. Objects considered instances of abstract data types with the dynamic binding of the procedure. The type Object can be an extension and type-specific procedures. The base type of the object is the type of the formal parameter. The modification of a new extension is textually localized and can be included in another module within its scope. The *handle* is a procedure field in the record. A *handler* processes the procedure and has two formal parameters that assign the object om which the handler operates and an identifier parameter which is *VAR*. An extensible record type is represented by its base type *Message*, which is used to send parameters to the *handler.* Oberon's object handler implementations are known to be much more reliable and better than Smalltalk.

2.3 Literature Review

J. Paakki's, A. Karhinen's, and T. Silander's (1990) article compare the orthogonally of type extensions, type hierarchies and type reductions between Oberon and the Alberich programming languages. The article touched on the improvement of type extension which is one of the inheritors of Oberon. L. Nigro's (1993) articles show the implantation and effectiveness by using the type extension and type-bound procedure of Oberon-2 that creates strong information hiding. B. Brown's (1994) article shows Oberon's type extension in a nonlinear passion. P. Reali's (2004) report expressed the changes in object-oriented extension including type extensions and type-bound procedures (methods) to better object-oriented types in Active Oberon.

L. Nigro's and B. Brown's articles touched on about inheritance and issue it has with the new Oberon-2. I agree with B. Brown's idea of considering Oberon-2 having more than one base-type that is equivalent to multiple inheritances, but H. Mossenbock's (1994) book mentioned that multiple inheritances are not supported in Oberon-2. J. Paakki's, A. Karhinen's, and T. Silander's and B. Brown's articles explained that Alberich's type systems are a better form in overall with the type extension than Oberon. The articles expressed the influence of Oberon's type extension to Oberon-2, Active Oberon, and Alberich that lead to the idea of inheritance, strong information hiding, and type-bound procedure.

3 METHODOLOGY

This section will explain what is type extension, show a few examples and compare the type-extension of Oberon and other languages that will support the method for answering the research question. The idea of type extension is to allow the program to be extensible and to define new record types based on or relating to the current record types. Record fields can be added to other modules and can provide a single inheritance that derived from a single base type. The concept of type extension originated from the Class of other programming languages such as Ada, C++, and Simula 67. The variables of extended types may be allocated to instances of their base types. The pointer type extension is bound to its base pointer type. The extended fields can be accessed through the type guard. The same assignment guidelines apply to the pointer, the base pointer can accept a dynamic type that is different from the declared type and the compatibility can be tested using the type-test. A procedure may be assigned to a variable if the formal parameter types and result type match those declared in the procedure type of the variable [2]. The technique of uniting the type extension and procedure enhances the extensibility that led to object-oriented.

The next few examples of type extension declarations from Oberon and other languages. An example of Oberon type extension and procedure in [2] are shown below.

```
TYPE
  Person = RECORD
    first, last: Name;
    id: INTEGER;
    birth: Date
   END;
 Pilot = RECORD(Person)
   hoursInFlight: INTEGER
END;
Clerk = RECORD(Person)
  jobCode: INTEGER
END;
PROCEDURE ProcessPerson(VAR p: Person);
   BEGIN … (*process common Person data name, id and birth *)
      IF p IS Pilot THEN
         WITH p: Pilot DO… (* process pilot data *) END
      ELSIF p IS Clerk THEN
         WITH p: CLERK DO … (* process clerk data*) END
   END
END ProcessPerson;
```

Example of pointer type in Oberon:

```
TYPE recA = RECORD o1: INTEGER END;
VAR rec1: recA;
TYPE ptrA = POINTER TO recA;
VAR ptr1: ptrA;
TYPE recB = RECORD(recA) o2: INTEGER END;
VAR rec2: recB;
TYPE ptrB = POINTER to recB;
VAR ptr2: ptrB;
```

In Alberich, the types are structured, set, pointer types and enumeration. In [5] express the type extension and reduction type by enumeration type.

```
type Scandinavian = (Danish, Norwegian, Finnish);
type S1 = extension of Scandinavian with (Swedish);
```

The enumeration type declaration implies that Norwegian is greater than Danish. Reduction type can be implanted by

```
type S2 = reduction of Scandinavian (Danish);
```

The type hierarchy of the enumeration types:

Scandinavian: {Danish, Norwegian, Finnish}
S1: {Danish, Norwegian, Finnish, Swedish}
S2: { Norwegian, Finnish}

In Oberon-2 [6], the type-extension is very similar with Oberon. The compatibility of a new type which consists of methods and new fields are maintained. Oberon-2 type extension can be declared by

```
TYPE
T0 = RECORD … END
T1 = RECORD(T0) … END
```

implies that *T0* is the base type or *superclass* of *T1* and *T1* is the direct extension or *subclass* of *T0* which inherits the methods of its superclass. The pointer type and procedure in Oberon-2 as expressed in [6] can be declared by

```
TYPE Figure = POINTER TO FigureDesc;
            FigureDesc = RECORD
            Selected: BOOLEAN;
            PROCEDURE (f: Figure) Draw; … END;
Rectangle = POINER TO RectangleDesc;
RectangleDesc = RECORD(FigureDesc)
            x, y, w, h: INTEGER;
            PROCEDURE(r: Rectangle)
 Fill(pat: Pattern)
END;
```

Similar to Oberon, the base type and the extension should be compatible with each other because a Rectangle object is also a Figure object.

In Active Oberon [7], the type-bound procedure or method is declared inside the object similar to Oberon-2, but declared differently in Oberon.

```
TYPE
 Coordinate = RECORD x, y: HUGEINT END;
 VisualObject = OBJECT
            VAR next: VisualObject;
            PROCEDURE Draw*;
            BEGIN HALT(99);
            END Draw;
END VisualObject;
```

Though some of the languages' declarations of type extension are syntactically different, they are still semantically the same to Oberon.

4 RESULTS

I have learned that not all of the descendants of Oberon carried its original type extension for a good reason. Some modified the implementation of the type extension to better their languages. Alberich programming language restored the idea of enumeration types. Since Oberon did not include the enumeration types from Modula-2, it had a lack of advantage in abstraction data type. This led to an idea and influence on Alberich by adding the enumeration type and became the central role of the language along with its type system that is based on supertype and subtype. This helped for both extension and reduction basis and abstraction clearness and uniformity of the Alberich. Record extensions of Oberon have in Alberich been simplified to cover data types and type reduction. Therefore, the type extension in Alberich is much more general than Oberon where they are applicable to record and type extension only [5].

One of the disadvantages of the type extension in Oberon was scoping rules particularly with the procedures in object scope. Oberon scope rules accept accesses without proper qualifications. This disadvantage led to the idea and influences of type-bound procedures in both Oberon-2 and Active Oberon. The type-bound procedure allows procedures or methods to be declared inside the object scope or record scope. This simplifies the method declaration and access to the fields and methods because they belong to the record scope and can only be accessed through record instances [7]. This also led to the idea

of consistency strong information hiding and better abstraction data types. The concept of type extension is implied through inheritance. The extension is called *subclass* and the base type is called a *superclass.* In sense, the *subclass inherits* the fields and methods of its *superclass*. Like Oberon, the type extension in Oberon-2 [6] works with pointer type as shown in the Methodology section.

5 CONCLUSION

Wirth's programming languages Oberon, Modula, and Pascal have had great influence on the programming world. Wirth's idea was to create Oberon to be better than Modula-2 with better complexity and more power. Wirth removed some features from Modula-2 such as enumeration types and gave Oberon a few new added features such as type extension that gave the ability to implement object-oriented ideas. Oberon was a small and simple programming language and was not as widely known as Ada, C, and C++. Though Oberon had influence on some of the modern programming languages, some do not show any influences from Oberon's type extension within these influenced languages. Oberon's type extension was one of the best features within its family and ALGOL-Like around that time, but it still had some weaknesses. Both Oberon-2 and Active Oberon carried most of Oberon's type extension style and was influenced to modify some of the type extension's aspect to improve the information hiding and object-centered access protection of the data fields by type-bound procedures. Oberon type extension had caused influence for implementing object-oriented programming features, inheritance and possibility of multiple inheritances with Oberon-2 [8]. Wirth's ideas eliminating enumerations types were a mistake and gave Oberon some difficulties and made it less flexible. This influenced Alberich to restore the enumeration types and became its centric idea. Some of the languages that are influenced by Oberon but were not influenced by its type extensions are Zonnon [9] and Go [4]. This shows Oberon's type extension influenced other programming languages mostly by its type extension's weaknesses and its type extension only influences its direct descendants such as Oberon-2 and Active Oberon. I am in no way bias towards Oberon because of its name, but it would have been interesting to learn the language, especially to someone who is interested in languages such as Modula-2 and Pascal. Though, I believed programming language such as C or C++ are a better "go-to" language because they offered more.

6 FUTURE WORK

Oberon-07 was created based on the original type of Oberon rather than Oberon/L also known as Oberon-2 with some modifications added. The Oberon-07 compilers have been developed to support different computer systems and target RISC processor to implement the Project Oberon's operating system. A further extension Oberon-2 but later named to Component Pascal was developed by Oberon microsystems for Windows and classic Mac OS. Component Pascal was also developed by the Queensland University of Technology for .NET. The .NET development concentrated on Zonnon, which contains features of Oberon and restores some features from Pascal. The .NET development environment includes Zonnon as a plug-in language for the Microsoft Visual Studio. Oberon system is still being maintained by Wirth and the latest Project Oberon compiler update is dated as Mar 6, 2020.

REFERENCES

[1] M. Reiser, The Oberon System: User Guide and Programmer's Manual, New York, NY: Association for Computing Machinery, 1991.

[2] M. Reiser and N. Wirth, Programming in Oberon: steps beyond Pascal and Modula, New York, NY: Association for Computing Machinery, 1992.

[3] N. Wirth, "Modula-2 and Oberon," *In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III). ,* pp. 3-1-3-10, 2007.
DOI: https://doi.org/10.1145/1238844.1238847

[4] N. Youngman and R. Peppé, Get Programming with Go, Manning Publications, 2018.

[5] P. Jukka, K. Anssi and S. Tomi, "Orthogonal Type Extensions and Reductions," *SIGPLAN Not.,* vol. 25, no. Jul, pp. 28-38, July 1990.
DOI: https://doi.org/10.1145/382076.382643

[6] H. Mossenbock, Object-Oriented Programming in Oberon-2, 2 ed., Heidelberg : Springer-Verlag , 1993, pp. 49-62.
DOI: https://doi.org/10.1007/978-3-642-79898-6

[7] P. Reali, "Active Oberon Language Report," ETH Zurich, 2004.
DOI: https://doi.org/10.1.1.12.7670

[8] L. Nigro, "On the Type Extensions of Oberon-2," *SIGPLAN Not.,* vol. 28, no. Feb, pp. 41-44, February 1993.
DOI: https://doi.org/10.1145/157352.157355

[9] J. Gutknecht and E. Zueff, "Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET," in *Companion of the 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, Washington, 2002.
DOI: https://doi.org/10.1145/985072.985108

[10] B. Brown, "Non-Linear Type Extensions," *SIGPLAN Not.,* vol. 29, no. Feb, pp. 39-43, February 1994.
DOI: https://doi.org/10.1145/181748.181757