A Proposed Software Description Language for Representing Program Logic in XML

لغة وصف برمجيات مفترضة لتمثيل البرامج بلغة التوصيف الموسعة

By

Khaled Zuhair Mahmoud

Supervised by

Prof. Azzam Sleit

Submitted in Partial Fulfillment of the Requirements for the Masters Degree in Computer Science

Department of Computer Science

Faculty of Information Technology

Middle East University

December, 2012

Authorization Statement

I, Khaled Zuhair Mahmoud, authorize Middle East University to supply hardcopies and electronic copies of my thesis to libraries, establishments, or bodies and institutions concerned with research and scientific studies upon request, according to the university regulations.

Name: Khaled Zuhair Mahmoud

Date: 23 /12/2012

Signature:

Examination Committee Decision

This is to certify that the thesis entitled **"A Proposed Software Description Language for Representing Program Logic in XML"** was successfully defended and approved on.

Examination Committee Member **Signature**

1- Prof. Azzam Sleit ....

2- Dr. Ghassan F. Issa ....
c. 1c/1c/cc

3- Ahmad Kayed ....
23-12-2012

Acknowledgements

I would like to express my sincere thanks to Prof. Azzam Sleit for his continuous support, efforts, and dedication.

I would also like to express my deepest gratitude, appreciation and love to Prof. Mohammad Al-Haj Hassan for encouraging me to continue in the idea while it was in the early beginnings. Special thanks are due to Dr.Hussien Owaied for helping me in publishing my first paper on the subject. I also thank my colleague Mohammed Salah Abu Saad for all his support, patience and invaluable advices.

I would like also to express my deep thanks, appreciation and admiration to Prof. Deya' Edeen Arafa for his unique and exceptional support and for guiding me in the most critical stage of the thesis, which is choosing the appropriate supervisor.

I would also like to thank the discussion committee for enriching my thesis with their comments and to MEU.

Dedication

I would like to express my thanks to my family for helping me and supporting me during my study.

Table of Contents

List of Tables

List of Figures

لغة وصف برمجيات مفترضة لتمثيل البرامج بلغة التوصيف الموسعة

الطالب

خالد زهير محمود

المشرف

الأستاذ الدكتور عزام سليط

الملخص

يقوم هذا البحث بطرح لغة برمجيات مفترضة لتمثيل البرامج المكتوبة بلغة ++C و الجافا و VB.net عن طريق لغة التوصيف الموسعة. إن التشابهات الدلالية بين هذه اللغات، تتيح تمثيل الشيفرات المصدرية بطريقة يمكن من خلالها مشاركة منطق، و شيفرة البرنامج، و إعادة استخدامها بسهولة بين هذه اللغات. و من خلال إجراء مقارنة بنائية، و دلالية بين لغات++C، و جافا و VB.net، تم تصميم اللغة المقترحة لتتضمن الصفات والتراكيب المتشابهة والمتطابقة، حيث تم تطوير تركيب متماثل في اللغة المقترحة لكل تركيب لغوي معتمد.

لقد تمت أيضا، دراسة صحة اللغة المقترحة، و إثباتها نظريا من خلال عقد مقارنة دلالية بين اللغات الثلاثة، و تجريبيا عن طريق تطوير تطبيق يحول الشيفرة المصدرية من لغة جافا إلى اللغة المقترحة، و من اللغة المقترحة إلى لغة VB.net.

و بالنسبة لحالات التحقق، فقد تم تصميمها لتشمل عدة برامج، منها الفرز والبحث، و لتشمل أيضا التراكيب البرمجية الأكثر استخداما في اللغات الثلاثة.

تم تحويل الشيفرة المصدرية لحالات التحقق من الجافا إلى اللغة المقترحة، و من اللغة المقترحة إلى VB.net. أما برامج الجافا و VB.net لحالات التحقق، فقد تم تنفيذها و مقارنة النتائج، و كانت النتائج متطابقة في جميع التجارب.

للغة المقترحة بعض الفوائد الأساسية في عمليات التحويل بين لغات البرمجة، كلغة وسيط، يمكن استخدامها في التكامل والدمج بين الأنظمة، كما أنها تسمح بمشاركة منطق البرامج خلال وقت التشغيل، في حين أن تقنيات الدمج الحالية تتيح فقط مشاركة البيانات بين الأنظمة المختلفة. لم تتطرق هذه الرسالة إلى تكيف مكتبات لغات البرمجة و وظائفها، إلا أن العمل في المستقبل قد يوسع اللغة المقترحة للتكيف مع مختلف المواصفات، كالمؤشرات، و التوارث المتعدد.

A Proposed Software Description Language for Representing Program Logic in XML

By

Khaled Zuhair Mahmoud

Supervised by

Prof. Azzam Sleit

Abstract

This thesis proposes a software description language to represent the source code of C++, Java, and VB.NET in the Extensible Markup Language. The similarity of semantics between these languages enables representing the source code in a form such that both, the source code and logic can be easily shared and reused between these languages.

By performing semantic and syntactic comparison between C++, Java and VB.NET, the proposed language has been designed to include the similar and identical features and language constructs. For every adopted language construct, a corresponding construct in the proposed language has been developed.

The validity of the proposed language has been investigated and proved theoretically by conducting a semantic comparison between the three languages and experimentally by developing applications to convert source code from Java into the proposed language and from the proposed language into VB.NET. Validation cases have been designed to include various programs such as sorting, searching and also to include the most used programming constructs in the three languages. Source code of the validation cases have been converted from Java into the proposed language, and from the proposed language into VB.NET. Java and VB.NET programs of the validation cases have been executed and results compared. The results were identical for all conducted experiments.

The proposed language has some major benefits in the conversion between programming languages as an intermediary language. It may also be used in the integration between systems as it enables sharing of programming logic at runtime. Existing integration technologies only enable the sharing of data between various systems.

This thesis is not concerned with adapting programming languages libraries and functions. Future work may extend the proposed language to adapt different features such as pointers and multiple-inheritance into the proposed language.

Chapter One

Introduction

1.1 Preface

The object oriented programming paradigm is widespread and many of the well-known and most used programming languages were designed to support this paradigm; Java and C++ are examples of such languages. Also, there are languages which were originally designed not to support the object oriented paradigm but have started to support it recently; PHP is a good example of those languages.

Many of the languages that support the object oriented paradigm have many semantics in common, for instance C++, VB.NET and Java support inheritance, overriding of functions, function overloading, and type casting. Differences also exist, for example while Java and VB.NET support interface declaration, C++ has no support for interfaces. In addition, nearly all programming languages provide the same basic set of features; control statements, declaration of variables, and calling of subroutines. Since there are similarities between programming languages, a description language can be developed to represent code written in similar languages so that code and logic can be easily shared and reused between similar languages, and this is the purpose and scope of this thesis where the features of different object oriented programming languages are compared to develop a proposed representation in XML (Extensible Markup Language) that includes the common features among C++, VB.NET and Java. This proposed representation has been named Software Description Language (SDL).

XML is widely used to store and exchange data, and its features that are borrowed from the relational, object oriented, and hierarchical models made it very powerful for data representation (Elmasri , et al. 2005). The structure of an XML document is specified by a document written in any of the schema description languages such as XSD(XML Schema Definition) and DTD (Document Type Definition) (Evjen, et al. 2007), and this enables automatic validation of XML documents. Since XSD is more powerful and expressive than DTD, XSD has been used for the specification of the developed model language.

1.2 Problem Definition

There are more than one hundred programming languages (Chen 2009), and many systems that use a wide variety of these languages. A subroutine written in VB.NET cannot be used directly by a subroutine written in Java and vice versa, unless that subroutine is converted from its language to the target subroutine's language. It is also the nature of software that there are functions that are similar in many different systems regardless of the implementing programming language. There even exist organizations whose main purpose is the development of reusable components in certain languages; Apache software foundation is a good example. If one of those companies wants to support another programming language, it has to rewrite its code for that language.

This thesis addresses the following issues:

1. There is no universally agreed upon representation for programming logic
2. If a program or module is to be converted from language A to language B, knowledge in both, the syntax and semantics of language B (the destination language) is required.

1.3 Contributions

The contributions of the thesis are as follows:

1. The development of the Software Description Language which is an XML representation of source code for C++, VB.NET, and Java that includes the common features between these languages.
2. The development of the algorithm that transforms an abstract syntax tree into the Software Description Language.

1.4 Significance

The adaptation of the proposed software description language facilitates converting a program from one language to another and it will initiate and ignite more advancement in different areas such as software integration between different platforms and systems. As an example of its significance as an intermediary language in the conversion between languages, programming language vendors such as Sun, Microsoft and Oracle may create tools to export and import source code from and to the proposed language. This enables a VB.NET developer to easily share and convert source code and libraries into C++ and Java.

Integration technologies such as XML web services and CORBA (Common Request Broker Architecture) enable heterogeneous systems to exchange data by agreeing on a well defined format of messages such as XML, JSON or other format, but they cannot exchange program logic. Adaptation of the proposed language enables systems to exchange algorithms and program logic at runtime by exchanging XML documents containing source code represented in the software description language. An application of is that a VB.NET application may send some part of its logic to be executed at a high performance application server running a C++ application The results of execution maybe exchanged via XML or any other appropriate formats.

1.5 Limitations

1. The proposed description language covers only C++, VB.NET, and Java.
2. The proposed description language does not adapt APIs specific to each of the three programming languages.
3. This research covers a subset of the semantics and features of the three languages that are common such that they can be adapted and migrated between the three languages.

1.6 Thesis Outline (Thesis Organization)

Chapter 2 presents information and theoretical background about the syntax and semantics of programming languages, types of programming languages including the object oriented programming languages and some of the technologies that are used by this thesis such as XML and XSD. It also lists the related researches about source code representation.

Chapter 3 presents the proposed model and the specifications of the proposed language as well as the algorithms that transform source code into the proposed description language. It also lists the features that are included in the proposed language.

Chapter 4 lists the experimental verification results that include source code in Java, its representation in SDL, and its representation after being transformed from SDL to VB.NET.

Finally Chapter 5 discusses the results and draw conclusions and future work.

Chapter Two

Literature Survey

This chapter presents knowledge and theoretical background about the syntax and semantics of programming languages, and types of programming languages including the object oriented programming languages and some of the technologies that are used by this thesis such as XML and XSD. It also presents the related researches about representation of source code in XML.

2.1 Theoretical Background

This section presents the necessary theoretical background and concepts necessary for understanding the topics related to thesis.

2.1.1 Programming Languages, Syntax and Semantics

A language is a set of symbols combined together according to a set of rules, known as the grammar or the syntax of the language, that are understood by both the sender and the receiver (Fischer & Grodzinsky, 1992). A language can either be natural or artificial. A natural language is a language that evolves naturally as means of communication between people (Vargas, J.V, 2011), while an artificial language is a language that is developed intentionally by the human for a specific purpose. Examples of natural languages include English, French, and Spanish and most human spoken languages. Examples of artificial languages include computer programming languages such as C++, Java, SmallTalk and Ada. Another example is Esperanto which is a planned human language intended for communication between people (Kadhim & Waite, 1996).

All languages have two types of rules which are the syntax rules and the semantic rules. The syntax rules define how to build correct sentences and structures and also include the set of words to be used in the language. Semantic rules define how to interpret those sentences and structures. A sentence that has a correct syntax is not necessarily meaningful. Figure 2-1 shows a sentence that has correct syntax but incorrect semantics because an integer variable cannot be assigned a string literal.

```
int i = "Software Description Language;"
```

Figure 2-1: Statement with correct syntax and incorrect semantics

Syntax specification of a programming language can be either concrete or abstract (Moses, 2006). Concrete syntax describes the phrase structure of the language (Kadhim & Waite, 1996) and determines which strings are accepted as programs (Moses, 2006), while abstract syntax deals with structure of programs without paying attention to the actual characters used to write the program. Abstract syntax specifies what the elements constitute the language and what the components of each element are. Abstract syntax for example may specify that an assignment statement is composed of one variable reference element on the left side and an expression element on the right side without specifying the actual textual representation. One concrete syntax specification may choose to use the '=' to denote equation and another may choose to use the word 'equals', and another one may prefer to surround the expression between square brackets '[]'. A sentence written in the abstract syntax form may be written in many concrete syntax forms. Table 2-1 shows how an arithmetic operation statement is written in various syntactic forms.

Table 2-1: Example of different syntactic representation

Syntactic Form	Representation
Infix	10 + 8
Prefix	(+ 8 10)
Postfix	(8 10 +)
JVM	bipush 10, bipush 8, add

Concrete syntax rules specify the keywords of the programming language and the naming rules of variables and also what the operators in the language are. In Java, syntax rules specify that a variable name is case sensitive and that it may only start with either '_', '$' or an alphabetical character.

Syntax of a programming language, either abstract or concrete, is specified through phrase-structure grammars. Language specification in phrase structure grammars consists of a set of symbols V, which is used to form sentences, words and literals and all members of the language, a set of terminal symbols T, a set of non-terminal symbols N, and a set of production rules P and a start element S. Elements of T and N are strings of finite elements of V. Terminal symbols cannot be broken into smaller parts and examples of them include the keywords of the programming language, and other symbols such as the semi colon, curly braces and square brackets. Non-terminal symbols can be broken into parts and they represent structural elements in the language. Productions are rules that specify what string from the set of all possible strings of V may replace another string from the same set. Productions are of the form 'a --> b'. An example is the use of a production to specify variable declaration. where the left hand side is the non-terminal symbol 'declaration' and the right hand side is the terminal symbol 'declare' followed by the non-terminal symbol 'variable name' followed by the non-terminal symbol 'type', assuming that the non terminal symbols 'type', 'variable name' are also declared and specified by other productions.

Phrase-structure grammars have four types; type 3 (regular grammar), type 2 (context free grammar), type 1 (context sensitive grammar) and type 0 (Rosen, 2011). Those types differ in the way productions are written. Table 2-2 shows the differences between these four types.

Table 2-2: Context Free Grammar Types

Grammar	Description
Type 3	Production rules in Type three grammars may contain 1) A non-terminal symbol on the right side and a terminal symbol on the left side or 2) A non-terminal on the left side and a non-terminal symbol on the right side along with any other symbols (Terminal or non terminal) or 3) A non-terminal symbol on the left side and an empty string on the right side.
Type 2	Productions have exactly one non-terminal symbol on the left hand-side and anything on the right hand side Examples : <address> => [<planet>, <country> , <city> , <building>]
Type 1	Productions are of the form s1 ==> s2 such that s1 = aBc and s2 = aWc and B is a non-terminal symbol and W a c are any string of terminal or non-terminal symbols
Type 0	Productions must at least one non-terminal on the left hand side Example : [<address>] ==> <B> <C> <B>

Formal syntax of a programming language is specified using a syntax meta-language, which is a notation for defining the syntax of a language by a number of rules (Scowen, 1993). BNF (Backus Naur Form) is a syntax meta-language that is widely used in language specifications and documentations such as MSDN (Microsoft Developer Network) and the official Java language specification (Gosling, Steele & Joy, , 2008). A variation of BNF is EBNF (Extended BNF) that it is used to specify syntax of context free languages. Production rules in EBNF consist of a non-terminal symbol on the left hand side and any number of terminals and non-terminals on the right hand side. EBNF has operators and symbols that enable it to define proper productions and of these symbols is the double quote, which is used to define terminal symbols. Figure 2-2 shows various examples of definitions in EBNF. The example shows the usage of various symbols such as ',' which denotes concatenation, '[ ]' which denotes that the symbols enclosed by the square brackets are optional, '{ }' which denotes that symbol enclosed by those curly braces may appear zero or more times, and the '|' which denotes an alternative.

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"| "8" | "9"
decimal fraction = ".", unsigned integer
unsigned integer = digit | unsigned integer, digit
```

Figure 2-2: Examples of definition in BNF.

SDF (Syntax Definition Formalism) is another language for describing syntax that is distinguished over BNF and EBNF by allowing syntax description to be divided into modules (Heering, Hendriks, Klint & Rekers, 1989). Each module declares its own syntax rules..For example, a dedicated module for numbers may contain definitions for floating point numbers, integers, hexadecimal representations of numbers, and a module may export his rules so they may be reused by other modules. Productions in SDF are written from right to left, which means that the defined entity is on the right side. Module declarations include the ‘sorts’ section, which declares the non-terminal symbols to be used in the productions, an imports section which imports grammars and other entities such as aliases from other modules. Figure 2-3 shows an example of a module declaration in SDF. In this declaration, the non terminal symbols declared are ‘Word’ and ‘Command. The non terminal ‘Word’ is declared as an alphanumeric string, and the non-terminal symbol ‘Command’, have five distinct forms. Valid syntactic statements according to this definition include ‘go to MEU’, ‘move to Amman’, ‘put Books on Shelf’, ‘fetch Pepsi from Refrigerator’.

```
module robots
exports
context-free start-symbols Command
sorts Word Command
lexical syntax
        [a-zA-Z]+ -> Word
context-free syntax
"go" "to" word   -> Command
"move" "to" Word   -> Command
"put" Word "in" Word -> Command
"fetch" Word "from" Word -> Command
```

Figure 2-3: Module declaration in SDF

Syntax is also specified using syntax diagrams (Reis, 2011) which graphically show how structural parts of the syntax are defined and connected to each other. Syntax diagrams are capable of specifying context free grammars and they were first used to specify the syntax of the Pascal language. The symbols of syntax diagrams consist of rectangles which denote non terminal symbols, and ovals which denote terminal symbols and arrows to specify the flow of definition. Parallel arrow paths within a declaration denotes options, equivalent to the symbol '|' in EBNF. Sequential symbols in a single arrow denote concatenation which is equivalent to the symbol ',' in EBNF. Figure 2-4 shows part of a syntax diagram that declares a Boolean literal.

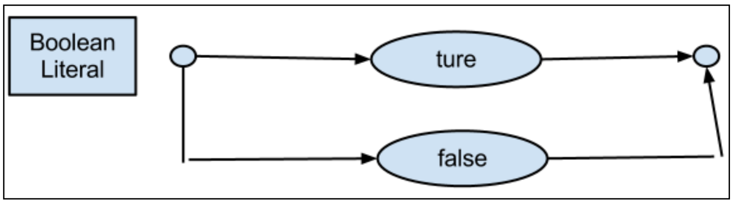


Figure 2-4: Definition of a Boolean literal in syntax diagrams

Syntax alone is not enough to specify a programming language, and without the specification of semantics a program is just a sequence of characters. Semantics show how a program should be interpreted and the meaning of each language element. Specification of semantics can be formal or in informal. In languages such as Java, C#, VB.NET, an informal semantic description is included in the official language documentation as textual paragraphs, clarifying examples and tutorials, while formal semantics is of narrow use in practice and it heavily makes use of mathematical notations, equations, and formal methods. There are three main types of formal semantics; denotational semantics, operational semantics, and axiomatic semantics (Zhan & Xu, 2004).

Denotational semantics, which are also called mathematical semantics, assign meanings to language elements by mapping them into mathematical objects such as functions and sets (Slonneger & Kurtz, 1995). A specification in denotational semantics consists of a description of the abstract syntax that lists all of the abstract syntax elements and the production rules which define the structure of those abstract elements, a semantic domain which consists of mathematical objects such as sets, structures, functions and equations that maps abstract syntactic elements into mathematical elements defined in the semantic domain. Figure 2-5 presents a part of the denotational semantics for arithmetic expressions. The abstract syntactic element 'ArithmeticExpression' is defined in the syntactic domain declaration, and a semantic domain denoted by the symbol 'Z' is defined to be the set of all natural numbers. In the semantic functions declarations, function 'A' is defined to map an ‘ArthimeticExpression’ element into a natural number. In the semantic equations declaration, four equations are defined to illustrate how function 'A' operates on plus, minus, multiplication, and division. At any point of program execution, a program is said to be in a certain state which is the value of all variables at that point, and this is denoted by the ‘σ’ symbol. A [[x]] σ means the denotational meaning of applying function ‘A’ on expression x under state ‘σ’. The first equation defines the arithmetic denotational meaning of (lexpression + rexpression) as the sum of arithmetic denotational meaning of lexpression and arithmetic denotational meaning of rexpression.

Syntactic Domain
ArithmeticExpression

Semantic Domain :
Z : The set of all natural numbers

Semantic Functions:
A[[-]] : ArithmeticExpression → (Σ→ Z)

Semantic Equations:
A[[lexpression+ rexpression]] σ = A[[lexpression]] σ + A[[rexpression]] σ

A[[lexpression - rexpression]] σ = A[[lexpression] σ - A[[rexpression]] σ

A[[lexpression * rexpression]] σ = A[[lexpression]] σ * A[[rexpression]]σ

A[[lexpression / rexpression]] σ = { undefined , if A[[rexpression]] = 0
A[[lexpression]] σ / A[[rexpression]] σ if A[[rexpression]] σ != 0 }

Figure 2-5: Denotational semantics for arithmetic expressions

Figure 2-6 shows part of the semantic denotation specification for Boolean expressions, and it uses the 'A' function declared in Figure 2-4. The first two equations, map the literals ‘true’ and ‘false’ into the mathematical values ‘true’, ‘false’ declared in the semantic domain. The third equation is an example of a Boolean expression resulting from comparison between two arithmetic expressions.

Syntactic Domain
BooleanExpression

Semantic Domain
Boolean : {true , false}

Semantic Functions
Boolean[[-]] : BooleanExpression → Σ → {true, false}

Semantic Equations
Boolean[[true]] σ = true
Boolean[[false]] σ = false
Boolean[[ar1 > ar2]] σ = {true, A[[ar1]] σ > A[[ar2]]
false, otherwise }

Boolean[[ar1 = ar2]] σ = {true , A[[ar1]] σ = A[[ar2]] σ
false, otherwise }

Boolean[[b1 or b2]] σ = {true , B[[b1]] σ is true or B[[b2]] σ is true]
false, otherwise }

Boolean[[not b]] σ = { true, Boolean[[b]] σ = false,
false, otherwise }

Figure 2-6: Part of denotational semantics for Boolean expressions.

During the execution of program statements, changes to the state of program occur. Operational semantics is concerned with the details of execution and how it transforms the program from a state to another. Specification in operational semantics is composed primarily of transition rules, also called execution rules, with each rule showing a transition in state (Turbak, Gifford, & Sheldon, 2008). An execution rule is composed of two parts, the premise and the conclusion. The premise is a set of preconditions that must be met in order for the program to be in the new state defined by the conclusion. Table 2-3 shows various executions rules along with their meanings.

Table 2-3: Examples of operational semantics

Statement	Operational Semantics	Meaning
Addition Expression x + y	$\sigma(x) \Rightarrow v1 \ \ \sigma(y) \Rightarrow v2$ --------------------------- $\sigma(x + y) \Rightarrow v1 + v2$	Preconditions : Value of x is v1 and value of y is v2 under state σ New State : The value of (x + y) is the value of x plus the value of y
Assignment Expression x = y	$\sigma(y) \Rightarrow v$ ------------- $\sigma(x=y;) \Rightarrow \sigma \oplus \{ (x, v) \}$ where the symbol ⊕ is defined as the overriding symbol operator	Preconditions : The value of y under state σ is v Post conditions: The new value of x is is v. The state σ is a set of variable name and variable value pairs. When using the overriding union with {(x,v)} this will result in the element (x,v) to be in the set regardless of the old value of x.
If Else Statement if c then t else e;	$\sigma(c) \Rightarrow True \ \ \sigma(t) \Rightarrow \sigma1$ ----------------------------- $\sigma(\text{if c then t else e; }) \Rightarrow \sigma1$ $\sigma(c) \Rightarrow False \ \ \sigma(e) \Rightarrow \sigma2$ ------------------------------- $\sigma(\text{if c then t else e; }) \Rightarrow \sigma2$	This is defined as two transition rules. The first one is the case in which c is true and the other is when c is false. When c is true, then the new state is the state that results by executing statement t. When c is false, then the new state is the state that results by executing statement e.

.

Axiomatic semantics describe the meaning of a program by providing assertions about the program (Regan, 2007), and they have wide applications in proving the correctness of algorithms and programs. Assertions are written as Hoare Triples which are based on predicate logic. Assertions are of the form C {S} Q where S is the program structure or statement, C are a set of assertions about the state of the program before executing and Q is a set of assertions about the state of the program after executing.

2.1.2 Paradigms of programming languages

Difference in semantics and features of programming languages is what makes them different. Some languages are similar to each other although they are different in syntax such as VB.NET and Java. According to their features and semantics, languages are classified to belong to one of the main paradigms; the object oriented paradigm, the functional paradigm, the logical paradigm and the imperative paradigm (Madsen, 2000). This section presents the main four programming language paradigms.

2.1.2.1 The Imperative Paradigm

Programs in imperative languages are composed of sequence of statements and commands that change the program state (Dowek, 2009), which is the set of all variables declared by the program. Almost all imperative programming languages include conditional statements, iteration structures, variable declaration statements, variable assignments, procedure and function declarations, and a mechanism for handling exceptions and errors.

A variable declaration statement allocates space in memory and associates it to a variable. The allocated space may contain an actual value or a pointer to another memory location. The contents of the reserved space is changed and controlled through the variable. A variable declaration statement must at least consist of the variable name, and according to the language type , statically or dynamically typed, may also consist of the type of the declared variable, and an optionally an initialization expression. C and Pascal are examples of statically typed imperative languages (Salus, 1999).

Declaration statements in C are composed of the variable type followed by the variable name and optionally the equal sign and an initialization expression, while in Pascal the ‘var’ keyword is used to begin the declaration statement, followed by a carriage return, and followed by multiple lines, each line declaring one or more variables. Figures 2-7 and 2-8 presents examples of variable declarations in C and Pascal.

```
int *p = &x;
char* message = "This is a simple string";
int[] numbers = {1,2,4,5,6,10};
```

Figure 2-7: Variable Declarations in C

```
Var
Number, Value: integer;
Text: string;
TrueOrFalse: Boolean
```

Figure 2-8: Variable Declarations in Pascal

In dynamically typed languages, the type of the variable is inferred at runtime based on the values assigned to it and according to the operations performed on the variable. An assignment statement that assigns a variable an integer value will set the variable type to integer. Also, the variable type in some dynamically typed languages such as JavaScript can be changed at run type by assigning the variable to another value of different type.

Assignment statements are composed of three main parts; referencing a variable, the assignment operator and the assignment expression. The assignment operator in C and Fortran is the equal sign while Pascal uses the '':=' symbol. The assignment expression can be a literal value such as 4, 'a', "A simple string", true or a reference to another variable, and in this case the value of the declared variable will be the same value of the referenced variable in the assignment expression. Also, any valid expression may be used in the assigned expression such as arithmetic expressions and Boolean expressions. Figure 2-9 presents various examples on assignment expressions in C.

```
x = 4;
y = x;
z = x+ y / 1000;
```

Figure 2-9: Assignment Statements in C

Conditional statements enable the conditional execution of a statement or group of statements based on the value of a boolean expression and the most common type is the if/else-if/else statement. Figure 2-10 shows a part of C program that prints whether an integer is even or odd.

```
int valueUnderTest = 4;

if (valueUnderText % 2 == 0) {
cout << ”Value is Even“;
} else {
cout << “Value is Odd”;
}
```

Figure 2-10: Conditional Statement in C

Iteration structures enable the execution of a sequence of statements repeatedly based on certain conditions. While-do/loop is one form of the iteration structures, which consists of a loop condition, and the body of the loop. In the While-do/loop, statements inside the body of the loop will be executed repeatedly until the loop condition evaluates to false. Another form of iteration structure is the do-while loop and it has the same structure as the while-do loop except that it will execute the body of the loop before checking the condition. Also there is the for-loop, which is in many languages such as Pascal and VB.NET consists of a lower bound and an upper bound and a loop step and a loop body. The loop body will be executed as long as the lower bound is not greater than the upper bound, and after each iteration the step statement will be executed to change the value of the lower bound. Figure 2-11 shows a complete program in Pascal that uses for-loop to calculate the sum of numbers from one to ten.

```
program sum;
total: real;
i : integer;
begin
total:=0.0;
for i := 1 to 10 do
begin
total := total + 10;
end;
end.
```

Figure 2-11: For loop in Pascal

During the execution of a program, various types of errors may occur such as unexpected errors due to division by zero, hardware failure, network communication, and other reasons. Also, the program code itself may decide that there is an error condition according to certain business rules, such as validating that the age is not less than zero. In both cases, if the exception is not handled, the program will terminate. Many imperative programming languages provide mechanisms for handling errors and taking actions such as logging the error in a log file, sending alert, or informing the user of incorrect input. The C++ language provides the try-catch structure for handling errors. The try-catch is composed of a try block containing the statements that are expected to generate errors, a catch block containing the statements to execute in case of error conditions, and an optional finally block that will always be executed. When an exception occurs, the flow of execution stops and moves up in the method stack until a ‘try’ block and its associated ‘catch’ block are encountered, where the statements inside the catch block get executed.

Imperative languages enable the declaration of subroutines which are groups of instructions declared inside the program that perform a specific task (Dhotre & Puntambekar, 2008) and can be called at any point by any part of the program. Using subroutine helps against writing the same group of statements whenever needed, thus reducing code size, increasing modularity and code clarity. Programming languages have rich libraries of built-in subroutines, in addition to the commercial and open source libraries. POSIX thread is an example of a library providing multi threading functionality to C++.

2.1.2.2 The Object Oriented Paradigm

The object oriented paradigm extends the imperative paradigm and introduces the idea of objects. In real life everything is an object, be it a car, a building, an elevator in a building, a computer, or even humans. The way people deal with these objects is through their characteristics and behavior. The behavior includes what an object does and what communications can be performed with that object. The main functionality of a car is transportation and people control the car by using the various tools available by the steering panel and the pedals to control speed. In this sense a car is an object and in addition to the behavior, it has properties such as price, size, speed and weight. The object oriented paradigm utilizes this view as the implementation and data structures of object oriented languages are based around classes and objects (Booch, et al.2007).

A class is a data structure that is a blueprint for an object that specifies what attributes and operations the object contains (Ambler, 1998) and an object is an instance and a realization of a certain class. Declaring a class does not allocate any memory until an object is instantiated in the program. Figure 2-12 shows the declaration of a class 'Car' and three instances from this class. The 'Car' class declares three attributes, weight, price, and model and one operation named 'move'.

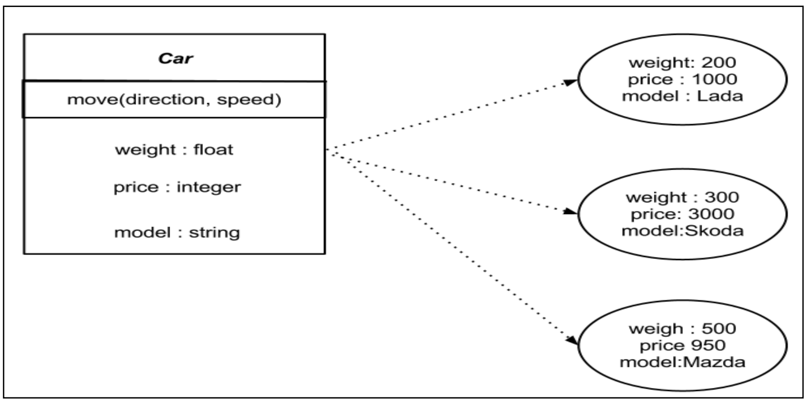


Figure 2-12: Classes and Objects

The main concepts of object oriented programming are data encapsulation, polymorphism, and inheritance (Weisfeld, 2008) and with the utilization of these concepts the advantages of object oriented programming such as modularity and re-usability can be achieved. Inheritance is a mechanism by which a class is created from an existing class and inherits all the properties and methods of the original class. In object oriented terminology, the inheriting class is called the subclass and the inherited class is called the super class. Inheritance promotes re-usability since the functionality of a certain class is made available to another class through inheritance. Figure 2-13 shows the class diagram of a class Employee inheriting from a Citizen class. The Citizen class contains attributes about the national number, social security number, date of birth and full name. The employee class needs all the information in the Citizen class in addition to another attributes such as salary and job title.

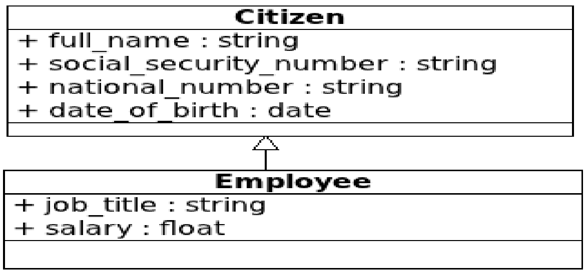


Figure 2-13: Inheritance

Encapsulation, which is also called data hiding, is a mechanism by which the related data and methods are placed in a class, and part of these data and methods are hidden and made visible only within the class (Swain, 2010). Encapsulation is useful when the internal implementation of a class changes, as clients of this class are not affected. Data encapsulation is enabled in many programming languages through access modifiers as they determine the visibility of the declared field, which is who can see and invoke what fields and methods.

Linguistically, polymorphism is the ability to appear in multiple forms, in object oriented programming, the same method definition may have multiple implementations, and upon calling the method, the type of object determines which implementation to call. One way to achieve polymorphism in statically typed languages is through inheritance and overriding of inherited methods. Since a super class reference can point to sub class object, the caller of a method on an object does not need to know the type of the object. Figure 2-14 presents an example of polymorphism in Java. One interface named ‘Greeting’ is declared with one abstract method named ‘greet’ and two classes that implement this interface are declared; EnglishGreeting and FrenchGreeting. Each of those classes prints a greeting message in a certain language.

```
interface Greeting {
   void greet();
}

class EnglishGreeting implements Greeting {
public void greet() {
   System.out.println(“Hello”);
}

}

class FrenchGreeting implements Greeting {
public void greet() {
   System.out.println("Bonjour");
}
}

public class Main {
 public static void main(String[] args) {
   Greeting fr = new FrenchGreeting();
   Greeting en = new EnglishGreeting();
   fr.greet();  //Prints Bonjour
   en.greet(); //Prints Hello
}
}
```

Figure 2-14: Polymorphism in Java

2.1.2.3 Lambda Calculus and the Functional Paradigm

Functional programming languages are based on the Lambda calculus, in which functions are first class objects (Lee, 2008), as they are passed as parameters, returned from function invocations. Lambda calculus consists of three main notations; variables, abstractions (function definitions) and function applications (Hindley & Seldin, 1986). Figure 2-15 shows the syntax specification of Lambda calculus expressions in BNF.

<expression> ::= <variable> ; identifiers

| (<expression> <expression>) ; function applications

| (λ <variable> . <expression>) ; abstractions.

Figure 2-15: Lambda Calculus Syntax

Abstractions in Lambda calculus denote anonymous functions in the form λ x.y, where x is called the bound variable and y is the expression that x is bound to. In function abstractions, x is the input to expression y.

A function application applies a function passing it an input value to produce a result. Function applications are in the form A B where A must evaluate to a function abstraction and B may evaluate to any expression.

In Lambda calculus all functions are anonymous (no names are given to functions) and all functions accept one argument only and this is enough to represent multi-argument functions through the use of currying. To illustrate currying, consider the function $f(x, y) = x + y$ such that $x \varepsilon Z$, $y \varepsilon Z$ and $f(x, y) \varepsilon Z$. With currying, the value of y is applied and a new function in terms of x is generated as $g(x) = x + y\_val$. Then the value of x is applied in the new generated function. Table 4 presents examples on Lambda calculus expressions and their meanings.

Table 2- 4: Lambda Calculus Expressions

Lambda Expression	Meaning	Expression Type
λ x.x	The identity function. If x is applied, x will be evaluated and returned as a result.	Function Abstraction
λ x λ y. x	A function that takes two arguments ignores the second argument and always returns the first argument.	Function Abstraction
λ f. λ x. f x	A function that takes two arguments, the first argument (f) is a function and the second argument (x) is variable. This function applies function f passing it the variable x.	Function Abstraction
λ f. λ x. f (f x)	A function that takes two arguments, the first argument (f) and the second argument (x) is a variable. This function applies the function f twice.	Function Abstraction
λ x.x (t)	Applies the identity function passing it , 't' as the input expression.	Function Application

While numbers, operators and predefined constants are not allowed in the original Lambda calculus, they are allowed in the applied Lambda calculus. In Applied Lambda calculus terms such as (x + 5) and (5) are permitted. Figure 2-16 shows various examples of applied Lambda calculus expression.

λx. λy x*x + y * y ==> Adds the square of two numbers
λxλ y λw (x + y + w) / 3 ==> Calculates the average of two numbers
λxx*x ==> Calculates the square of the input variable
λx1 λx2 λy1 λy2 (y2 - y1)/(x2 - x2) ==> The slope of a line formed by the points (x1, x2), (y1, y2)

Figure 2-16: Applied Lambda Calculus

2.1.2.4 First Order Predicate Logic and Logic Programming

A program in a logical programming language is composed of axioms and rules and queries to be answered and goals to be achieved based on these axioms. An axiom can be any fact like 'Khaled loves Programming' and 'Ali hates computers'. Queries may ask, 'What does Khaled hate?', 'Who loves programming?', and 'Who hates computers?'

Prolog is one of the most popular logic programming languages (Yasdi, 1997) that are based on first order predicate logic, which provides formal methods to describe facts. Figure 2-17 presents example in first order predicate logic containing two facts and one. The first two facts state that Khalid studies at MEU, and that Ai studies at MEU. The rule states that if two people study at the same place, then they know each other. In this example, "studies", "knows" are the predicates while "khaled", and "meu" are entities, X, Y are variables.

```
studeis (khaled , meu)
studies (ali, meu)
studies (X,Z) ^ studies(Y, Z) ⇒  knows(X,Y)
```

Figure 2-17: Facts in first order predicate logic

Prolog syntax is very similar to the syntax of first order predicate logic. In prolog, predicates have to be declared before being used in rules in a separate section and the rules and the facts are also declared in a separate section of the program. Figure 2-18 shows a Prolog program that declares the rule that two entities are considered to be brother if they have the same father and the same mother.

```
PREDICATES

father (string, string)
mother (string, string)
brother (string, string)

CLAUSES

father(ali, ahmad)
father(ali, issa)
mother(mona, ahmad)
mother mona, issa)

brother(X, Y) IF father(X, Z) ^ father (Y, Z) ^ mother(X, W) and mother (Y,W)
```

Figure 2-18: Facts and Rules in Prolog

2.2 Related Work

Cox, Clarke, & Sim (1999) developed a model for storing source code in software repositories that aids in performing queries and analysis of source code. In this model, original textual representation of source code and additional supplementary information in XML are stored. The supplementary information is very basic and primitive. Programming constructs such as data types, method invocations, operators are poorly modeled. Figure 2-19 shows an example of source code representation according in this model.

```
<comment>/*constant1 function2  returning3 zero4
*/</comment.

<varDef>
<type>int5</type>
<name>z6</name>=<const>07</const>;
</varDef>

<funDef>
<type>int8</type>
<funName>zero9</funName>
<params>()</params>.
<body>{
<stmt>
<keyW>return10</keyW>
<expr>(<varRef> <defined
loc=5,7>z11</varRef>)</expr>;</stmt>
}</body>
</funDef>
```

```
int z = 0

int zero () {
    return (z)
}
```

Figure 2-19: Source Code Representation in Model Independent Source Code Repository (Cox, et al. 1999, P.4)

Badros (2000) introduced JavaML as a method of representing Java Source code in XML. It emphasizes on the benefits of using such representations by Software Engineering tools as it aids in source code analysis and other tasks. This research developed a tool that transforms Java source code into XML. It also developed a tool for transforming JavaML representation in XML back to Java, and a specification for

JavaML in XSD. Figure 2-20 shows an example for source code representation in JavaML along with the original Java source code it represents. Programming constructs are well modeled in JavaML as all Java constructs are mapped to corresponding XML elements.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-source-program SYSTEM "java-
ml.dtd">

<java-source-program name="FirstApplet.java">
<import module="java.applet.*"/>
<import module="java.awt.*"/>
<class name="FirstApplet" visibility="public">
<superclass class="Applet"/>
<method name="paint" visibility="public" id="meth-15">
<type name="void" primitive="true"/>
<formal-arguments>
<formal-argument name="g" id="frmarg-13">
<type name="Graphics"/></formal-argument>
</formal-arguments>
<block>
<send message="drawString">
<target><var-ref name="g" idref="frmarg-13"/></target>
<arguments>.
<literal-string value="FirstApplet"/>
<literal-number kind="integer" value="25"/>
<literal-number kind="integer" value="50"/>
</arguments>
</send>
</block>
</method>
</class>
</java-source-program>
```

```java
 import java.app.*
 import java.awt.*;

public class FirstApplet extends Applet {


public void paint(Graphics g) {
   g.drawString(“FirstApplet”, 25, 50);
}

}
```

Figure 2-20: JavaML Representation of Source Code (Badros, 2000, P.4)

The code structure format project, Documentation for Code Structure Format (CSF2), represents the information about source code in XML and it was part of the (Software Development Foundation) open source project, with the aim of providing information about source code to analysis tools. It can represent code for C++ and Java, but it does not include method implementations.

Kontogiannis & Zou (2001) stressed on the importance of encoding abstract syntax trees in XML and the benefits gained in areas such as software re-engineering and software analysis. DTD is suggested as a specification language for the XML representation. Comparison between the size of the original source code and the transformed XML representation of source code is presented and these comparisons showed that XML representations are much larger in size. This research encourages future research in generating portable representations of source code for software engineering tools.

Mamas (2000), developed three meta-languages for representing source code in XML; JavaML for representing Java source code, CppMP for representing C++ source code, and OOML for representing general object oriented source code for the purpose of utilization by software analysis tools. DTD was used as a specification language for the three languages. OOML is missing many of the object oriented language features and the aim of the research did not aim at transformation between programming languages. Experimental results concentrated on developing simple software tools to illustrate the usefulness and effectiveness of these languages in software analysis.

GXL (Graph Exchange Language), (Winter , Kullbach, & Riediger ,2002) was developed and designed for the purpose of representing graph data structures in XML and to support interoperability between software engineering tools. Being an ordered directed graph, A GXL graph could represent class hierarchies, function calls, and passing of parameters, but no actual modeling of programming constructs was established, and the main elements of documents in GXL are ‘node’ and ‘edge’.

Simic, H (2003), discussed the benefits of representing source code in XML. It argues that XML representations leverages and benefits from the well developed standards and technologies already available for XML such as XSL. Benefits discussed included querying of source code, code refactoring and formatting, and also addition of extensions for other applications since addition of tags inside the source code will not disturb the normal processing.

Collard (2004), developed an XML representation for C/C++ for the purpose of supporting meta differencing and DTD was used as the specification language. Many programming constructs are not well modeled. Figure 2-21 shows how a variable initialization would look like in the proposed representation. In this example, the “=" operator is placed directly in the representation instead of mapping it to an XML element. Also, there is no clear distinction between primitive and non primitive types as they are both declared using the type element. This research suggests extending the proposed representation to support more languages such as Java and Python.

```
<type>
<name>void</name> </type>
<name>main</name>
<decl_stmt>
<decl><type><name>float</name></type>
<name>number</name>, <name>sum</name> =<init>
<expr>0</expr></init></decl>;
</decl_stmt>


<expr_stmt><expr><name>cout</name> &lt;&lt; "Entering 10
numbers will calculate their average." &lt;&lt;
<name>endl</name></expr>;</expr_stmt>
```

```
void main() {

   float number, sum = 0;
   cout << “Entering 10 nubmer will
calculate their average;” << endl;



}
```

Figure 2-21: C++ source code representation in XML

Raiser (2006) , developed an XML representation representing general source code. for the purpose supporting intentional programming tools. It could represent source code in C++, C#, Java as those languages are common and all are support the object oriented paradigm, but language portability was not an aim of this research. Many inconsistent features across the three languages are part of the specification, such as multiple-inheritance

Seato (2007), developed an interpreter than enables a program to be written in multiple programming languages. At run time, the active interpreter can be switched according to code in execution.

Prakash, Goebel, & Wang(2010) developed an intermediate language for representing executable code so that executable code can be ported to different platforms. An executable program source is converted to an intermediate language, and then processed to produce an executable code for the target platform. The intermediate representation is not in XML and contains symbol tables, object bindings … etc.

Jiří(2010), established a framework for transforming Java source code into XML, similar to Badros (2000), but focusing on latest technologies for transformation such as JAXB. It is intended to be a whole framework including tools rather than just a language representation.

2.3 What Distinguishes This Thesis?

1. Addressing source portability across different programming languages by performing semantic and syntactic comparison.
2. Experimental verification: Various programs are converted from Java to SDL, and then from SDL to VB.NET. Both, the original program in Java and the transformed program in VB.NET are executed and the results of execution are compared to prove the validity of the proposed language.
3. All supported programming language constructs have corresponding XML constructs in XML. Also SDL uses XSD for the specification of its structure.

Chapter Three
The Proposed Model

This chapter presents a model for a proposed description language in XML named as Software Description Language (SDL). It explains the schema and the main constructs of the language, and lists the features and semantics supported by the language as well as the main transformation algorithms and functions.

3.1 SDL and Its Role in the Conversion between Languages

SDL is an XML representation of source code in objects oriented languages. It includes the common semantics between C++, Java, and VB.NET. SDL can be used as intermediary to share source code across different languages and platforms. A program or a module written in Java can be converted to SDL, and then, VB.NET applications may utilize this representation by transforming the code from SDL to VB.NET. This process is illustrated in Figure 3-1

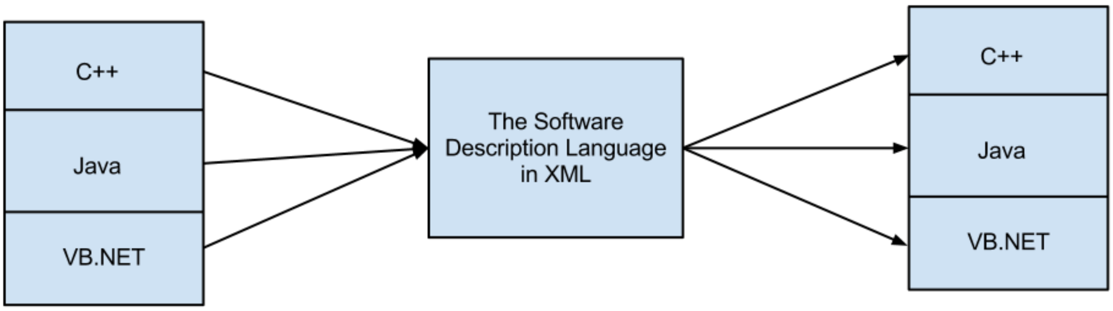


Figure 3-1: SDL and its rule in the conversion between languages

3.2 SDL's Features List

This section presents the features and semantics supported by SDL as well as the excluded ones.

3.2.1 The Object Oriented Features

Since SDL has been designed to represent source code in the object oriented paradigm, most of the features of this paradigm are supported.

1- Class: Since declaring classes is a central concept in the Object Oriented paradigm, C++, Java, and VB.NET support declaring of classes. Declaring of classes is supported in the SDL.

2- Abstract Class: Declaring abstract classes is supported in C++, Java, and VB.NET. This feature is supported by SDL.

3- Access modifier: Assigning access modifiers to Interface: Interface declaration is supported only in VB.NET and Java but not in C++. This feature is supported in SDL since a program declaring interfaces can be adapted when transformed into C++. Semantically, a class implementing multiple interfaces in Java or VB.NET is equivalent to C++ class inheriting from multiple abstract classes, and an interface is semantically equivalent to an abstract lasso having all the methods declared as abstract.

4- class fields and methods is supported in C++, Java, and VB.NET and since access modifiers are the not the same in those three languages, only the private and public access modifiers are supported in SDL. Any access modifier that is not private is suggested to be converted to public when transforming to SDL.

Public and Private access modifiers have the same meaning across the three languages. A public modifier means that any class can invoke and access the class member, while a private modifier means that only methods and statements from the same class can access and invoke the member.

5- Constructor Declaration: Declaring constructors is supported in the SDL since it is supported by C++, VB.NET and Java. Constructor chain invocation is the same in those three languages, as well as overloading of constructors and the rules of default constructor declaration.

6- Package Declaration: Declaring packages and assigning classes into packages is supported in the SDL since it is supported by the three languages (C++, VB.NET, and Java).

7- Static class member: Declaring static fields and methods is supported in SDL since it supported by the three languages. A static field or method can be accessed or invoked directly via the class name without the need for an instance of an object of the class.

8- Destructor Declaration: The declaration of destructor is not included in this research

9- Inheritance: Object oriented inheritance is supported in C++, Java, and VB.NET and it is supported in SDL. The following features in C++ inheritance are not supported in SDL since they are not common and not supported in VB.NET and Java.

- Multiple inheritance
- Private and protected inheritance types in C++.

10- Operator Overloading: Operator overloading that is supported in C++, is not supported in the SDL. However, operator overloading can be easily adapted upon transformation by

   - Transforming every overloaded operator into a function inside the class.
   - Transforming every overloaded operator into a function inside the class.

11- Overriding: Overriding of inherited methods is a central concept in the object oriented paradigm that enables to implement other central concepts such as polymorphism. This feature is included SDL. The below features are not supported in SDL that are related to overriding in VB.NET and C++.

   - Shadowing: VB.NET supports re-declaring an inherited method from a parent class. Re-declaring may include changing the return type of the shadowed method.
   - C++ supports early binding by declaring a function without the use of the virtual keyword. In this case, the type of object pointer, not the actual object type at runtime, determines which method to invoke. In the SDL, it is assumed that late binding is always used.

3.2.2 The Imperative Paradigm Features

Since the imperative paradigm extends the object oriented paradigm, most of the features of imperative programming languages are included. This section lists the included imperative features as well as the excluded ones.

1. Pointer: C++ supports pointers including pointers to functions, pointers to primitive data types and pointer to data structures. Pointers are not supported in SDL since it is not supported by Java.

2. Pass by Reference: When calling a method, passing the parameters can be either by value or by reference. Call by value passes a copy of the variables, while in pass by reference; the called function can change the original variables. Call by reference is supported by C++ and VB.NET but not by Java. This feature is not supported by SDL.

3. Conditional Statement: Conditional statements are supported by most of the imperative programming languages and are supported SDL. However, in C++ the logical expression of an 'if' statement may be a reference to an integer variable as Figure 3-2 shows. In this case, C++ evaluates the value of the expression and checks if it is not zero. If the value is zero, the expression evaluates to false, true otherwise. Transformers have to take this into consideration and produce an equivalent logical expression such as "x != 5".

```
int x = 5;

if (x) {

cout << "x is not zero";

}
```

Figure 3-2: Conditional Expression in C++

4. Switch Statement: C++, Java, and VB.NET support switch statements. However VB.NET does not support the flow feature that C++ and Java support. In this feature, if a case is matched and this case does not break the execution, all of the following cases will be executed. The flow feature can be into VB.NET by converting the switch statements into a sequence of if, else-if, and else statements. Chapter 4 includes validation case for a switch statement in Java that is transformed into the SDL and then adapted into VB.NET.

5. Do while and while loops: SDL supports the two types of loops as they are supported by C++, VB.NET, and Java.
6. Bitwise Operator: The following bitwise operators are supported by C++, Java, and VB.NET and by SDL: Or, Not, Exclusive Or, Right Shift and Left Shift and Complement.
7. Logical Operator: The following logical operators are supported by C++, Java, and VB.NET and by SDL. And ,Or ,Greater Than, Greater Than or Equals ,Less Than ,Less Than or Equals , Not.
8. Arithmetic Operator: The following arithmetic operators are supported by C++, VB.NET and Java and by SDL: Addition, Subtraction, Multiplication, Division, Division Reminder, and Sign (plus and minus).

   A difference that has to be taken into consideration is that the expression (1/2) evaluates to 1 in VB.NET and 0 in both C++ and Java. This is due to the fact that VB.NET calculates the ceiling of the expression while C++ and Java calculates the floor. So the use of direct mathematic functions has to be used to insure consistency
9. Exception Handling: Exception handling is not included in SDL and is not part of this study.
10. Data Types: When declaring a primitive variable in any of the three languages, its type has to be declared. Upon transforming code from SDL, special care has to be taken as not to use data types out of the ones supported by SDL as loss of data and therefore errors and logical mistakes in the resulting program may occur. The following are the list of data types supported by SDL.

- SIGNED_INT_ONE_BYTE: An integer data type with range from -128 to 127
- SIGNED_INT_TWO_BYTES: An integer data type with range from -32,768 and a maximum value of 32,767
- SIGNED_INT_FOUR_BYTES: An integer data type with range from -2,147,483,648 to 2,147,483,647
- SIGNED_INT_EIGHT_BYTES: An integer data type with range from -9,223,372,036,854,775 to 9,223,372,036,854,774
- BOOLEAN: True or false
- SIGNED_FLOAT_FOUR_BYTES: IEEE 754 floating point with size of 4 bytes

11. Variable Declaration

    Almost all imperative and object oriented languages support declaration of variables. However, each language has its own rules regarding the naming of variables. In VB.NET, variable and function names are not case sensitive, and upon transforming VB.NET code into SDL, variable names has to be unified to appear cases sensitive to SDL. SDL is a case sensitive.

12. Casting: This feature is supported in the three languages and is supported in SDL.
13. Array Declaration: Declaring array of primitive types and of abstract data types is supported in C++, VB.NET and Java and is supported by SDL.
14. Language APIs: Every programming language has a set of predefined libraries. It is the responsibility of transformers to build adapters that translates the calls into the appropriate functions. The use of the adapter and facade design patterns is recommended.

15. For Loop: The structure of a ‘for loop’ in C++ and Java described in Figure 3-3.

```
for (initialization statement ; logical expression ; expression statements) {

        loop body : statements

}
```

Figure 3-3: Loop structure in C++ and Java

The structure of a’ for loop’ in VB.NET is described in Figure 3-4

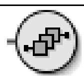
```
For Initital Value to Destination Value  [Step Increment ]

        Statements

Next
```

Figure 3-4: Loop structure in VB.NET

3.3 SDL Schema

XSD is used to specify the structure of SDL. This section presents the main elements of SDL and their attributes and relations with other elements. Every XML document in SDL is composed of one ‘source’ element which is also composed of zero or more package elements. The package element has a name attribute. Each package element may have zero or more class elements and zero or more interface elements. Table 3-1, shows a map showing the meaning of the graph symbols used by the figures in this section.

Table 3-1: Meanings of symbols used in the diagrams of the specification

Symbol	Meaning
	Sequence symbol: All elements or groups defined to the right of this symbol must appear in the order from up to down and according to the multiplicity constraints.
	Group of element or other groups: A group declares the membership of elements or other groups. Al elements or groups appearing in the box that is marked with this symbol are members of the group.
	Choice: It means one of the elements or groups defined on the right side of this symbol should be part of the defined element and according to the multiplicity constraints.

	A light gray line: Means that the element on the right side is optional (Either 0 or 1 times).
X.. Y	Multiplicity: Defines the minimum and maximum occurrence of a group within another element. If no multiplicity is present, it means that the element must appear one time only.

Figure 3-5 shows the ‘source’ element and Figure 3-6 shows the XSD specification for the ‘source’. It shows that a ‘source’ element may contain zero or more package elements.

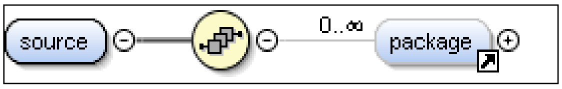


Figure 3-5: The ‘source’ element

```
<xs:element   name="source">
<xs:complexType>
<xs:sequence>
<xs:element ref="package" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 3-6: ‘source’ element XSD specification

Figure 3-7 shows the package element and Figure -8 shows its XSD specification. It shows that a ‘package’ element may contain any number of class‘ elements and any number of interface elements and has one attribute; ‘name’.

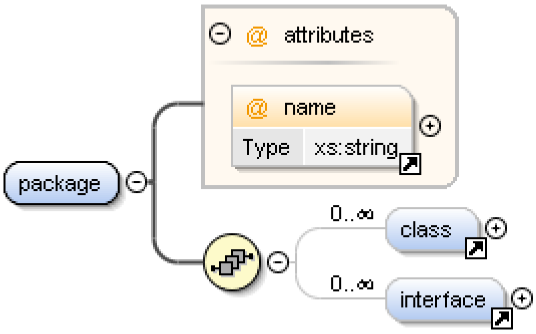


Figure 3-7: The ‘package’ element

```
<xs:element   name="package">
<xs:complexType>
<xs:sequence>
<xs:element ref="class" minOccurs="0" maxOccurs="unbounded" />
<xs:element ref="interface" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
<xs:attribute ref="name"/>
</xs:complexType>
</xs:element>
```

Figure 3-8: ‘package’ element XSD specification

Figure 3-9 presents the ‘class’ element and Figure 3-10 presents its XSD specification. The class has three attributes specifying the name of the class, and whether the class is abstract or not, and whether the class can be inherited or not. It optionally contains an ‘extends’ element to model inheritance and also optionally contains one or more 'implements' elements to model implementing interfaces. The other elements represent methods, constructors, fields, and abstract methods.

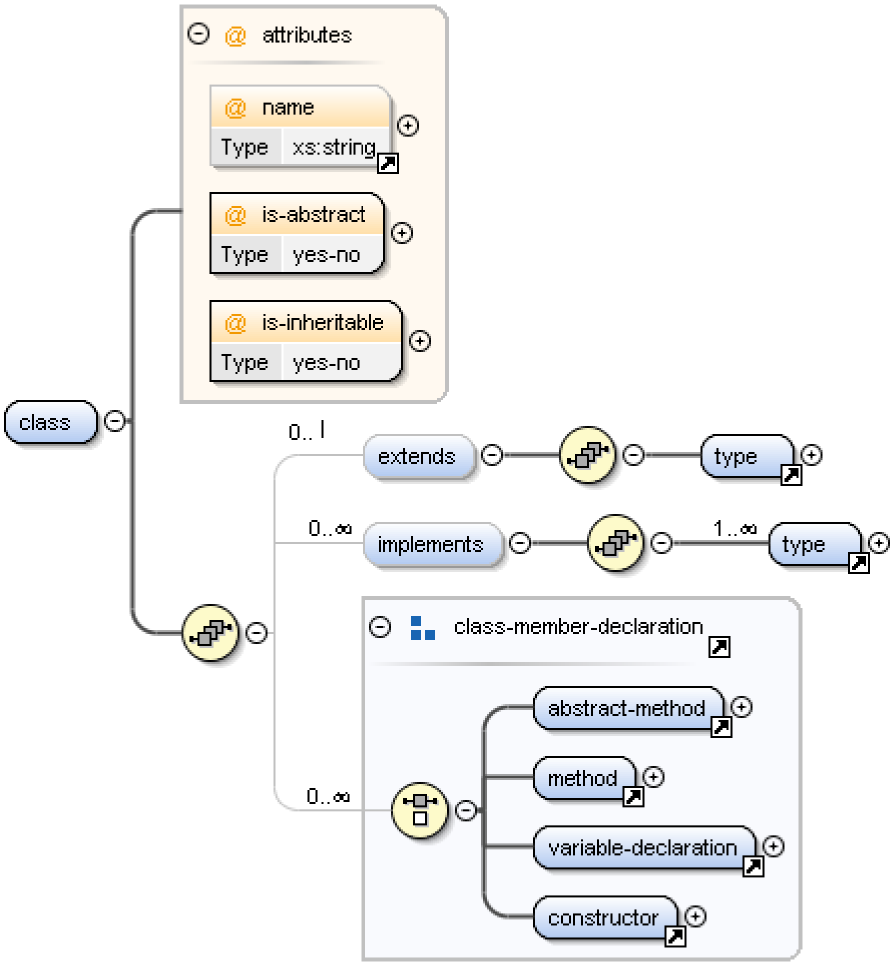


Figure 3-9: The 'class' element

```
<xs:element name="class">
<xs:complexType>
<xs:sequence>
<xs:element name="extends" minOccurs="0" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:element ref="type" minOccurs="1" maxOccurs="1" />
</xs:sequence>
</xs:complexType>
```

```
</xs:element>
<xs:element name=”implements” minOccurs=”0” maxOccurs=”unbounded”>
<xs:complexType>
<xs:sequence>
<xs:element ref=”type” minOccurs=”1” maxOccurs=”unbounded” />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:group minOccurs=”0” maxOccurs=”unbounded” ref=”class-member-declaration” />
</xs:sequence>
<xs:attribute ref=”name”/>
<xs:attribute name=”is-abstract” use=”required” type=”yes-no” />
<xs:attribute name=”is-inheritable” use=”required” type=”yes-no” />
</xs:complexType>
</xs:element>
```

Figure 3-10: XSD specification for the ‘class’ element

Figure 3-11 presents the ‘interface element and Figure 3-12 presents its XSD specification. The ‘interface’ element has one attribute to specify the name of the interface. The ‘interface’ contains elements to model method declarations and fields as well. It also contains an ‘extends’ element to model interface inheritance supported by Java and VB.NET.

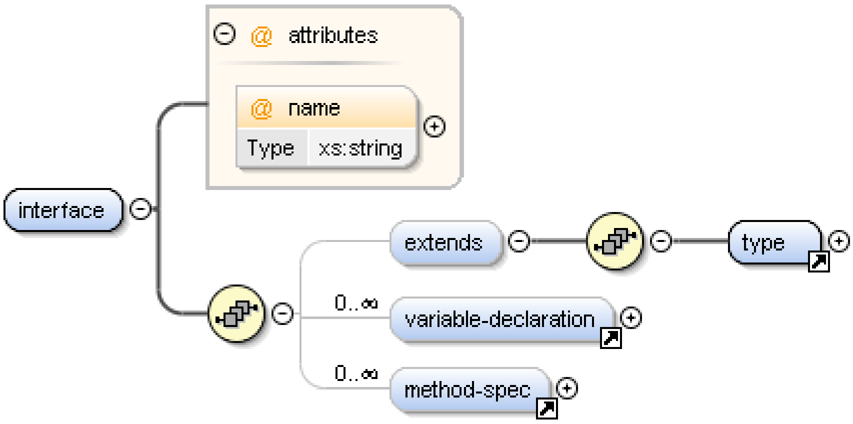


Figure 3-11: The ‘interface’ element.

```
<xs:element name="interface">
<xs:complexType>
<xs:sequence>
<xs:element name="extends" minOccurs="0" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:element ref="type" minOccurs="1" maxOccurs="1" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element ref="variable-declaration"  minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="method-spec" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
<xsl:attribute name="name" type="xs:string" />
</xs:complexType>
</xs:element>
```

Figure 3-12: XSD Specification for the ‘interface’ element

Figure 3-13 presents the ‘type’ element and Figure 3-14 presents its specification in XSD. The ‘type’ element models programming language types; both the primitive data types and the abstract data types. The ‘type’ element contains either a ‘primitive-type’ element for primitive data types, or an ‘object-type’ element for abstract data types, or an ‘array-type’ for array types.

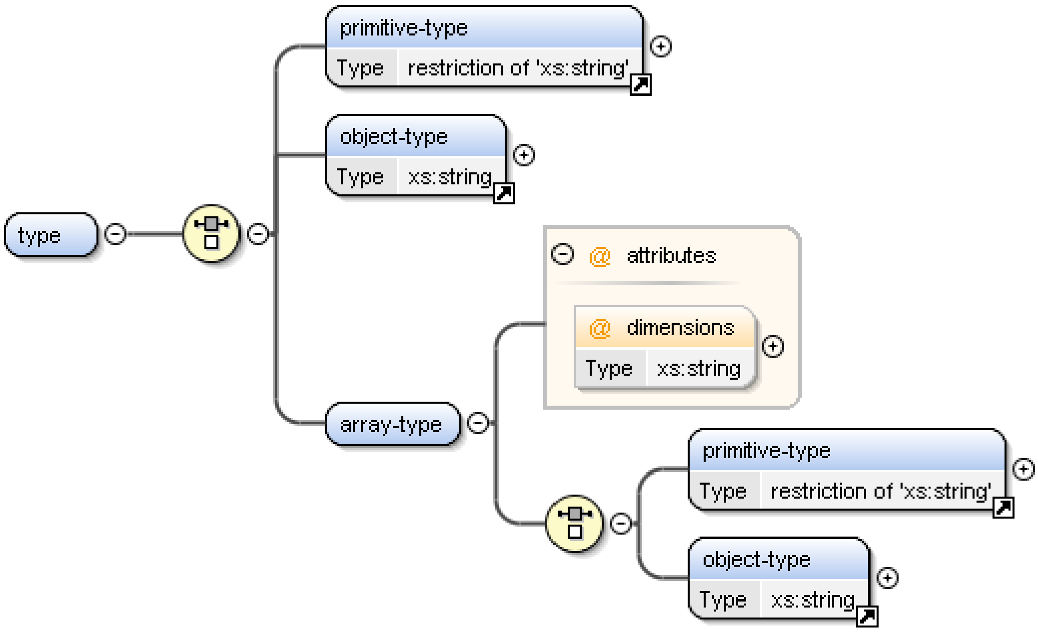


Figure 3-13: The ‘type’ element

```
<xs:element name="type">
<xs:complexType>
<xs:choice>
<xs:element ref="primitive-type"/>
<xs:element ref="object-type" />
<xs:element  name="array-type">
<xs:complexType>
<xs:choice>
<xs:element ref="primitive-type"/>
<xs:element ref="object-type" />
</xs:choice>
<xs:attribute name="dimensions" type="xs:string" />
</xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
</xs:element>
```

Figure 3-14: XSD specification for the ‘type’ element

Figure 3-15 presents the ‘method’ element Figure 3-16 presents its XSD specification. The ‘method’ element has three attributes to specify the access modifier of the method, the name, and whether the method is override-able. The ‘method-spec’ element inside the ‘method’ element has one attribute to specify the name and it contains a ‘type’ element to specify the return type of the method. The ‘method-spec’ element also contains zero or more ‘variable-data-declaration’ elements to model the parameters. The ‘method-body’ element represents the body of the method and contains zero or more the of elements that belong to the ‘statement’ group.

```
<xs:element name="method">
<xs:complexType>
<xs:sequence>
<xs:element ref="method-spec" />
<xs:element ref="method-body" />
</xs:sequence>
<xs:attributeGroup ref="method-modifiers" />
</xs:complexType>
</xs:element>
```

Figure 3-15: XSD specification of the 'method' element

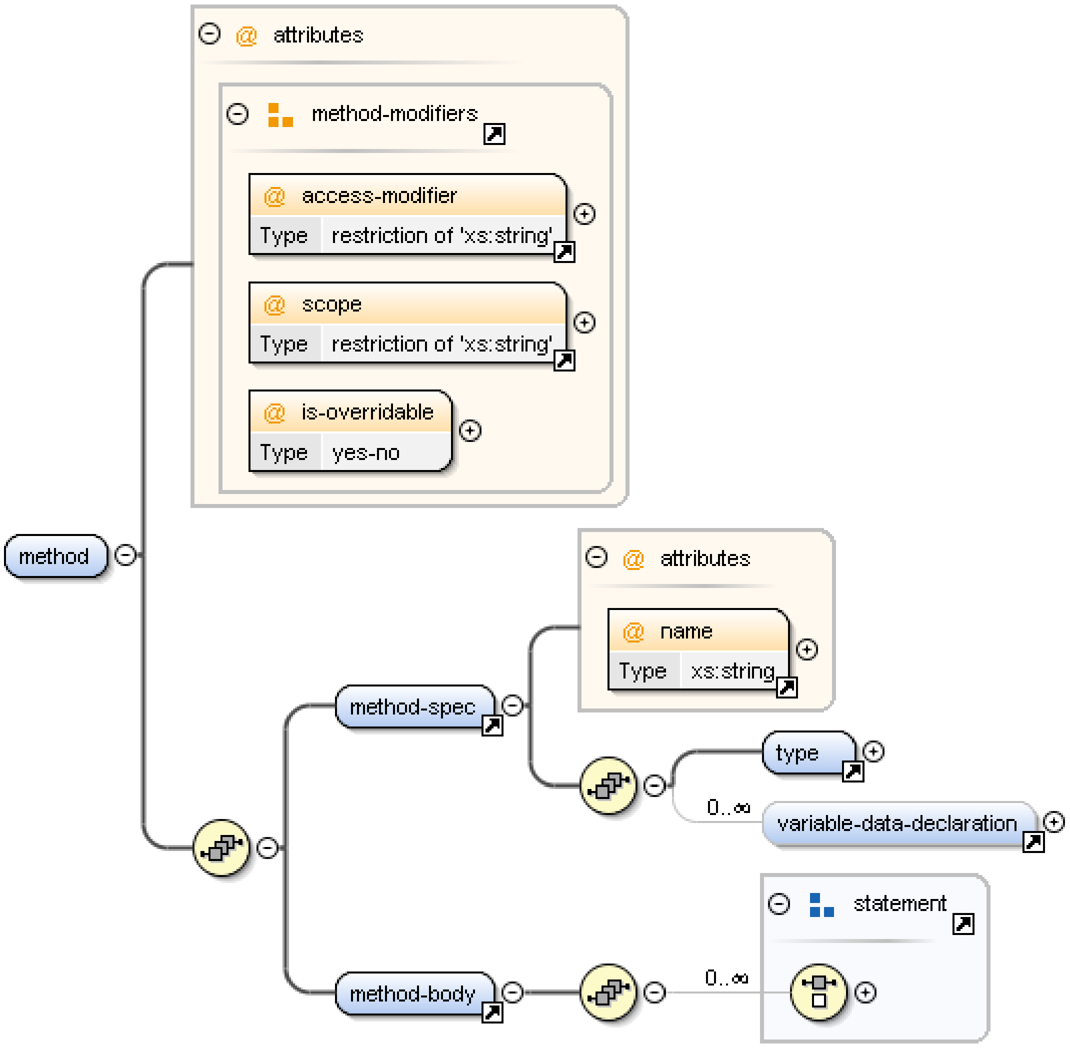


Figure 3-16: The 'method' element

Figure 3-17 presents the ‘variable-data-declaration ‘element ‘and Figure 3-18 presents its XSD specification. It has two attributes to specify whether name of the variable, and whether the declared variable is constant. The type is specified through the ‘type’ element contained inside this element. This element is used by ‘method-spec’ element to specify the parameters list.

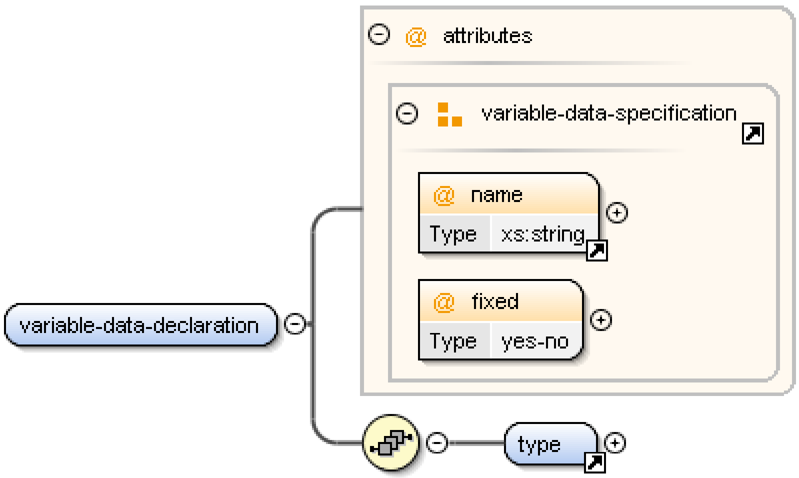


Figure 3-17: The’variable-data-declaration’ element.

```
<xs:element name="variable-data-declaration">
<xs:complexType>
<xs:sequence>
<xs:element ref="type" />
</xs:sequence>
<xs:attributeGroup ref="variable-data-specification" />
</xs:complexType>
</xs:element>
```

Figure 3-18: ‘variable-data-declaration’ XSD specification.

Figure 3-19 presents the ‘variable-declaration’ element and Figure 3-20 presents its XSD specification. The variable declaration models instance variable declaration statements inside classes and interfaces. It different from ‘variable-data-declaration’ is that ‘variable-data-declaration’ does not specify modifiers and does not model variable initialization.

```
<xs:element   name="variable-declaration">
<xs:complexType>
<xs:sequence>
<xs:element ref="type" minOccurs="1" maxOccurs="1"/>
<xs:element minOccurs="0" maxOccurs="1" ref="variable-initialization" />
</xs:sequence>
<xs:attributeGroup ref="variable-data-specification" />
<xs:attributeGroup ref="modifiers" />
</xs:complexType>
</xs:element>
```

Figure 3-19: ‘variable-declaration’ XSD specification

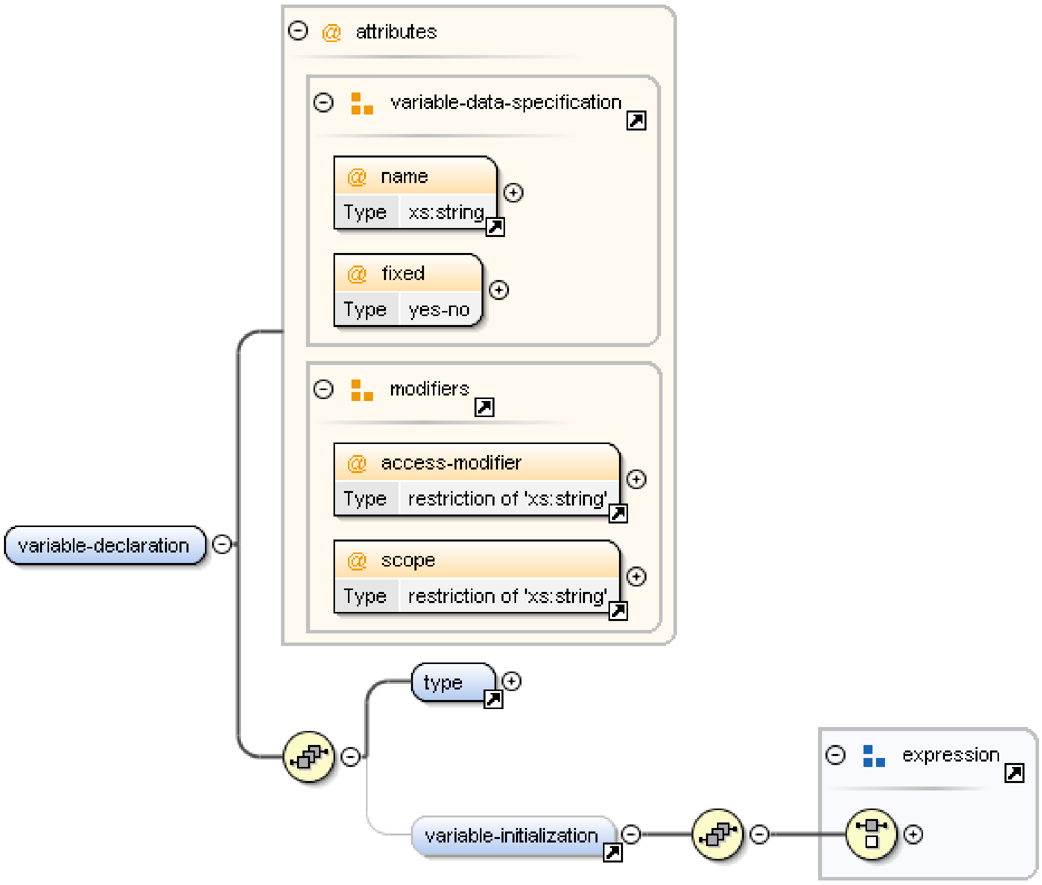


Figure 3-20: The ‘variable-declaration’ element

Figure 3-21 presents the ‘constructor’ element and Figure 3-22 presents its specification. The ‘constructor’ element is similar to the ‘method’ element but it does not have a name attribute.

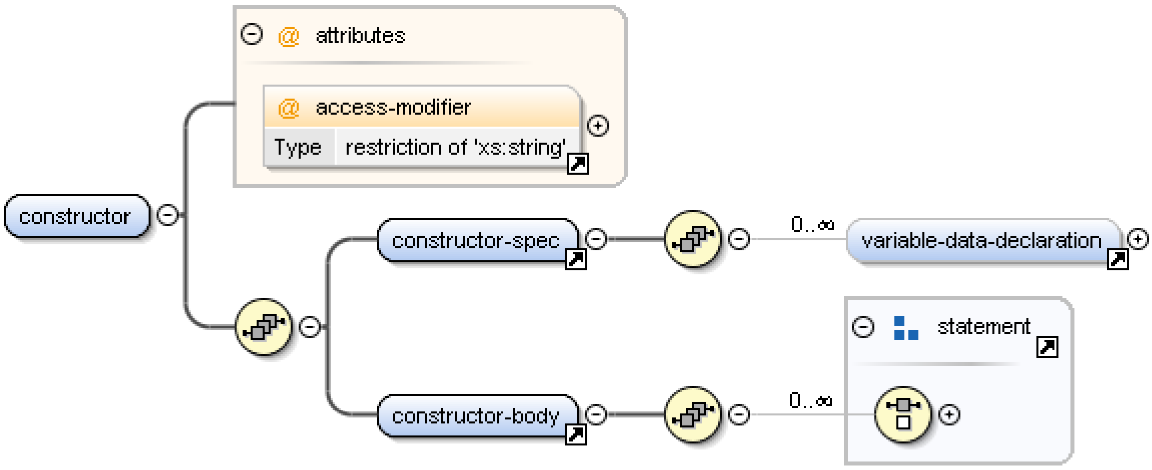


Figure 3-21: The ‘constructor’ element

```
<xs:element name="constructor">
<xs:complexType>
<xs:sequence>
<xs:element ref="constructor-spec" />
<xs:element ref="constructor-body" />
</xs:sequence>
<xs:attribute ref="access-modifier" />
</xs:complexType>
</xs:element>
```

Figure 3-22: XSD specification for the constructor element

Figure 3-23 presents the ‘statement’ group and shows the members of this group, and Figure 3-24 presents the ‘expression’ group and shows the members of this group. A statement in a programming language is an independent instruction that may appear directly inside a method body. Loops, conditional statements, compound statement, empty statements, variable declarations, and method invocations, variable assignment are all examples of methods. An expression is a construct that evaluates to value under a certain state. Variable reference, method invocation, literal values are examples of expressions.

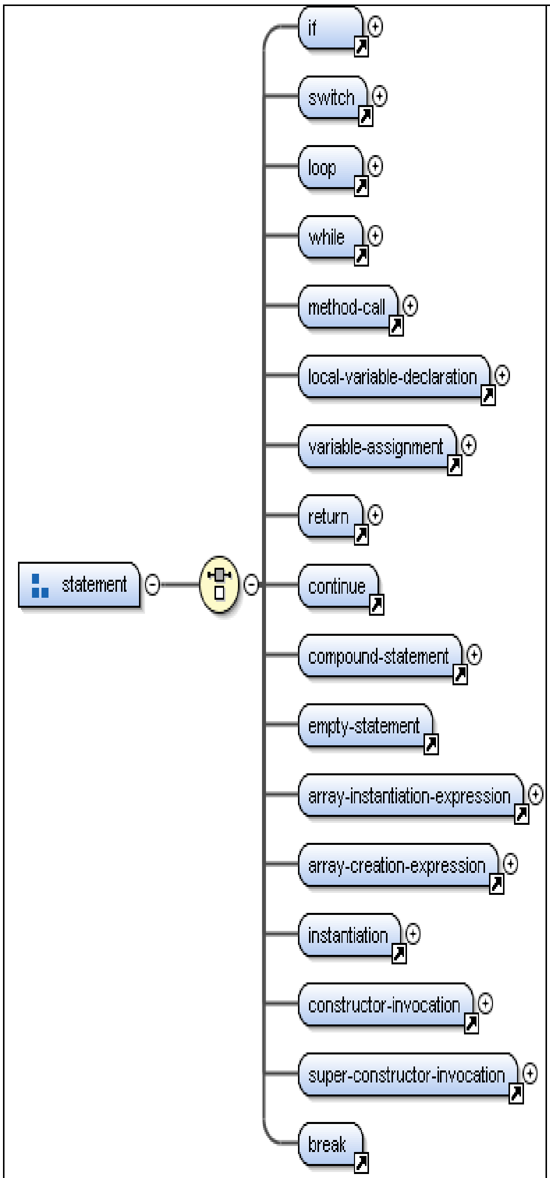


Figure 3-23: The ‘statement’ group

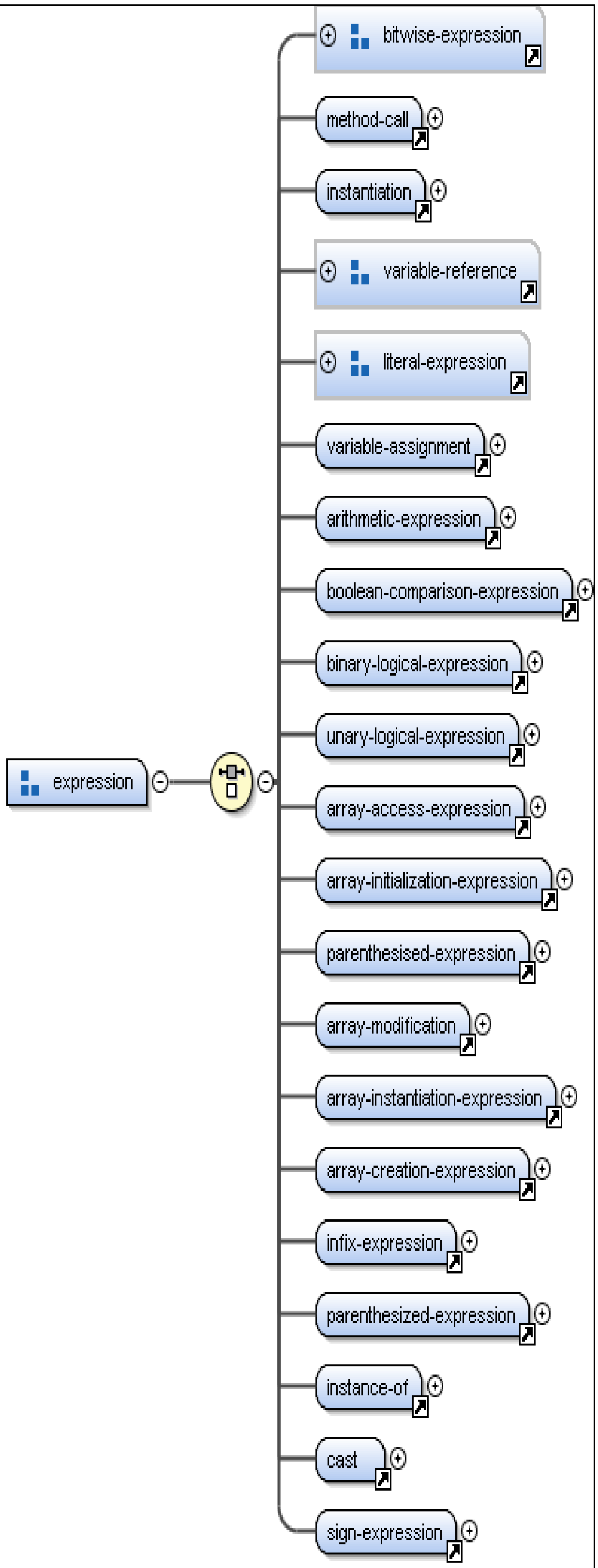


Figure 3-24: The ’expression’ group

Figure 3-25 presents the ‘literal-expression’ group and shows the members that belong to it. Figure 3-26 shows the XSD specification for the literal expression. A literal expression can be any number including integer and floating point numbers, characters such as ‘a’, ‘b’, ‘\n’, sting literals, and Boolean literals which can either true or false. It also includes the null literal that is denoted by the ‘null’ keyword in Java and the ‘Nothing’ keyword in VB.NET.

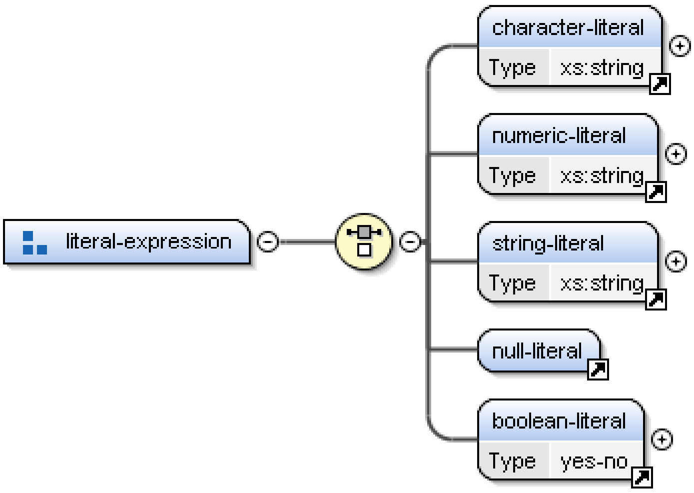


Figure 3-25: The ‘literal-expression’ group.

```
<xs:group name="literal-expression">
<xs:choice>
<xs:element ref="character-literal"/>
<xs:element ref="numeric-literal"/>
<xs:element ref="string-literal" />
<xs:element ref="null-literal" />
<xs:element ref="boolean-literal" />
</xs:choice>
</xs:group>
```

Figure 3-26: ‘literal-expression’ XSD specification.

Figure 3-27 presents the ‘loop’ element and Figure 3-28 presents its XSD specification.

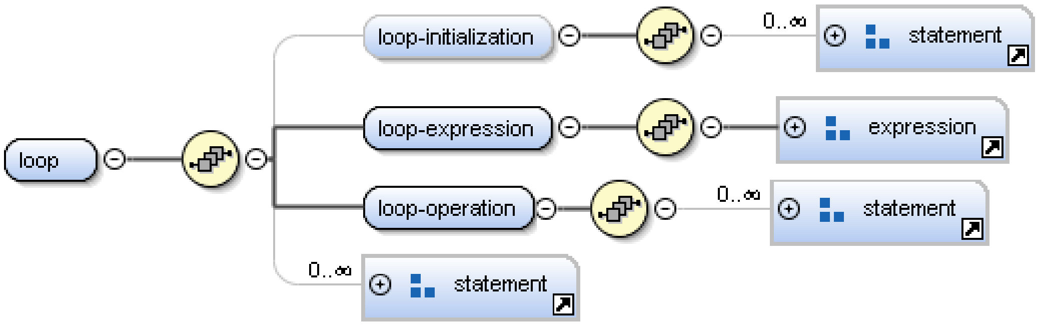


Figure 3-27: The “loop” element

```
<xs:element name="loop">
<xs:complexType>
<xs:sequence>
<xs:element name="loop-initialization" minOccurs="0" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:group ref="statement" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="loop-expression">
<xs:complexType>
<xs:sequence>
<xs:group ref="expression" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="loop-operation">
<xs:complexType>
<xs:sequence>
<xs:group ref="statement" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:group ref="statement" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 3-28: ‘Loop’ element XSD specification.

Figure 3-29 presents the ‘if’ element and Figure 3-30 presents its XSD specification. This element models the if statement in programming languages.

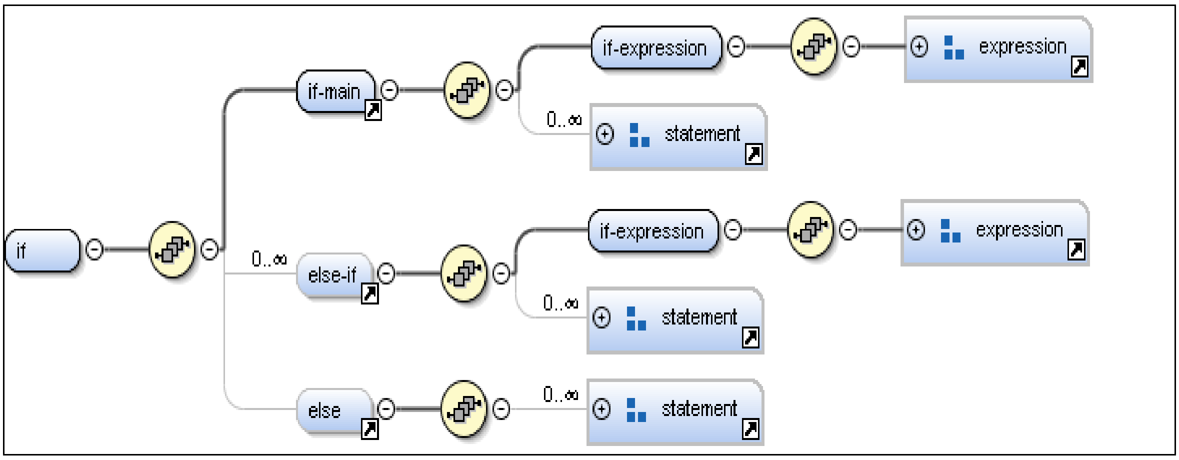


Figure 3-29: The ‘if’ element

```
<xs:element name="if">
<xs:complexType>
<xs:sequence>
<xs:element ref="if-main" minOccurs="1" maxOccurs="1"/>
<xs:element ref="else-if" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="else" minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:complexType> </xs:element>
```

Figure 3-30: XSD specification of the ‘if’ element

Figure 3-31 presents the switch ‘element’ and Figure 3-32 presents its XSD specification.

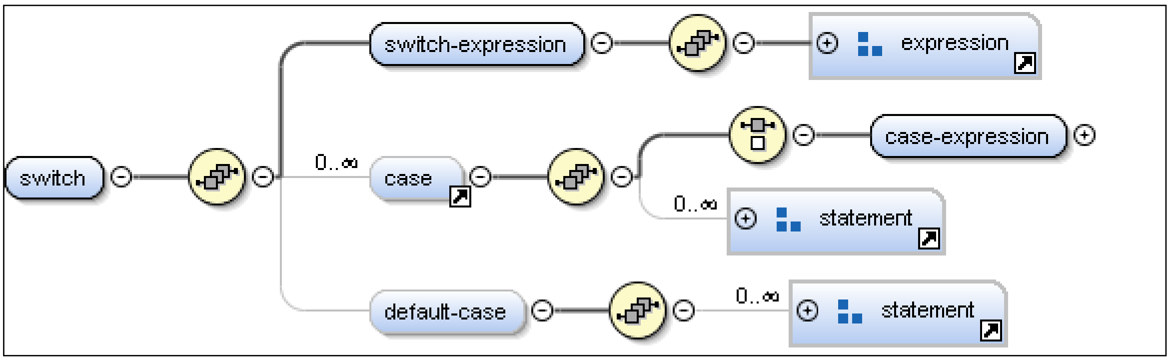


Figure 3-31: The ‘switch’ element

```
<xs:element name="switch">
<xs:complexType>
<xs:sequence>
<xs:element name="switch-expression">
<xs:complexType>
<xs:sequence>
<xs:group ref="expression" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element ref="case" minOccurs="0" maxOccurs="unbounded" />
<xs:element name="default-case" minOccurs="0" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:group ref="statement" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence> </xs:complexType> </xs:element>
```

Figure 3-32: XSD specification of the ‘switch’ element

Figure 3-33 presents the ‘while’ element and Figure 3-34 presents its XSD specification.

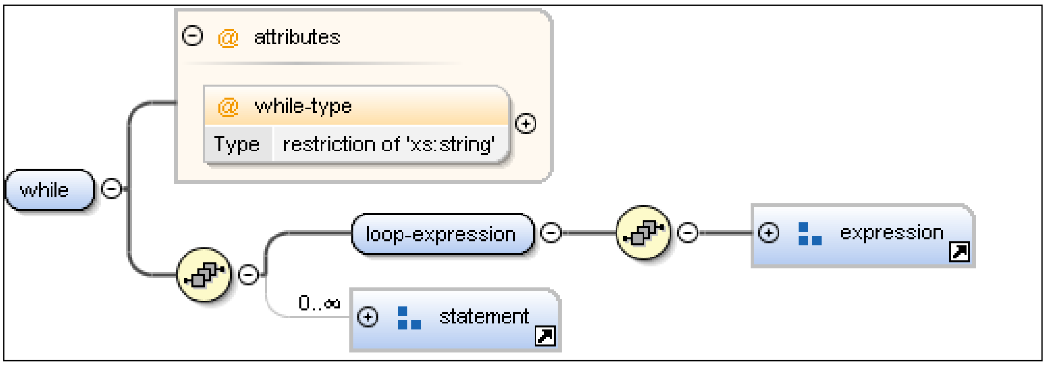


Figure 3-33: The ‘while’ element

```
xs:element name="while">
<xs:complexType>
<xs:sequence>
<xs:element name="loop-expression">
<xs:complexType>
<xs:sequence>
<xs:group ref="expression" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:group ref="statement" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
<xs:attribute name="while-type">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="while-do" />
<xs:enumeration value="do-while" />
</xs:restriction>
</xs:simpleType>
</xs:attribute> </xs:complexType> </xs:element>
```

Figure 3-34: XSD specification of the ‘while’ element

Figure 3-35 presents the 'arithmetic-expression' element and Figure 3-36 presents its XSD specification.

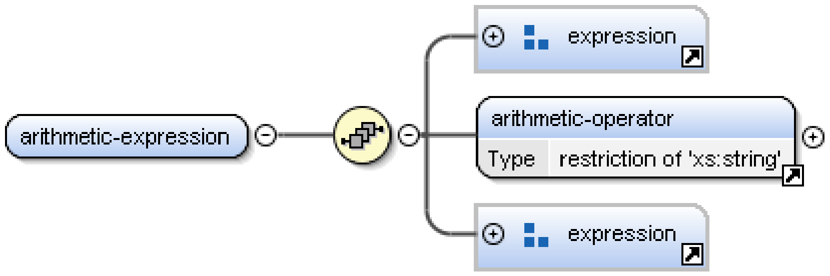


Figure 3-35: The 'arithmetic-expression' element

```
<xs:element name="arithmetic-expression">
<xs:complexType>
<xs:sequence>
<xs:group ref="expression" />
<xs:element ref="arithmetic-operator" />
<xs:group ref="expression" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 3-36: XSD specification for the 'arithmetic-expression' element

Figure 3-37 presents the 'cast' element and Figure 3-38 presents its XSD specification.

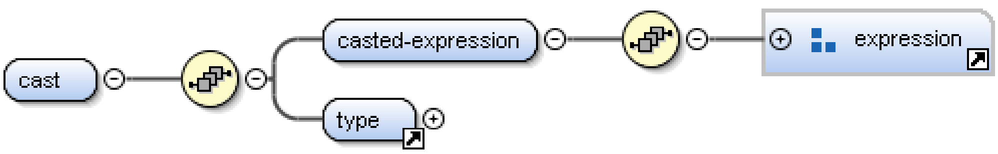


Figure 3-37: The 'cast' element

```
<xs:element name="cast">
<xs:complexType>
<xs:sequence>
<xs:element name="casted-expression">
<xs:complexType>
<xs:sequence>
<xs:group ref="expression" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element ref="type" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 3-38: XSD specification for the ‘cast’ element

Figure 3-39 presents the ‘instantiation element and Figure 3-40 presents its XSD specification.

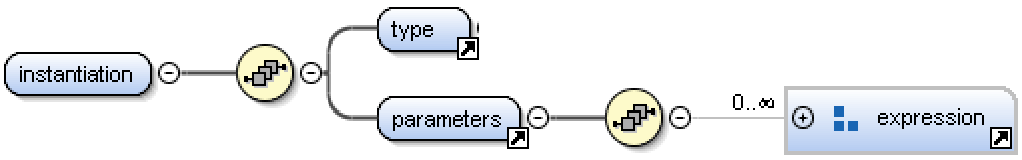


Figure 3-39: The ‘instantiation’ element.

```
<xs:element name="instantiation">
<xs:complexType>
<xs:sequence>
<xs:element ref="type" />
<xs:element ref="parameters" minOccurs="1" maxOccurs="1" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 3-40: XSD specification for the ‘instantiation’ element

Figure 3-41 presents the ‘variable-reference-group’ and Figure 3-42 presents its XSD specification. It models the various kinds of variable references, including the implicit variable references; the current object and the parent object. Also it models the various cases of variable references; “some_obj_reference.var”, “callAMethod().var”, “direct_var”, “TypeName.var”.

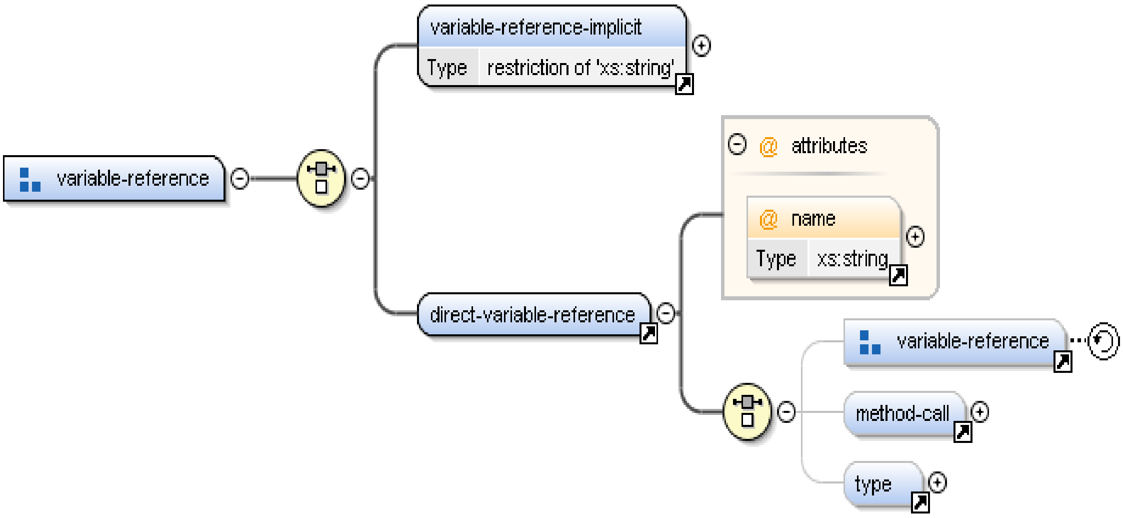


Figure 3-41: The ‘variable-reference’ group.

```
<xs:group name="variable-reference">
<xs:choice>
<xs:element ref="variable-reference-implicit"   />
<xs:element ref="direct-variable-reference" />
</xs:choice>
</xs:group>
```

Figure 3-42: The XSD specification for the ‘variable-reference’ group

Figure 3-43presents the ‘return’ element which models the return statement and Figure 3-44 presents its XSD specification.

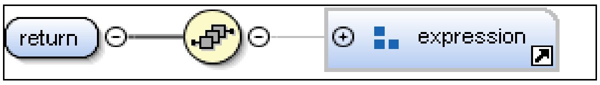


Figure 3-43: The ‘return’ element

```
<xs:element name="return">
<xs:complexType>
<xs:sequence>
<xs:group ref="expression" minOccurs="0" maxOccurs="1" />
</xs:sequence></xs:complexType></xs:element>
```

Figure 3-44: XSD specification for the ‘return’ element.

Figure 3-45 presents the ‘parenthesised-expression’ element and Figure 3-46 presents is XSD specification.

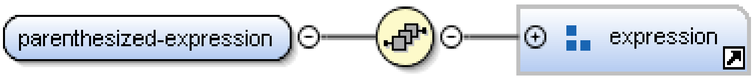


Figure 3-45: The ‘parenthesised-expression’ element

```
<xs:element name="parenthesized-expression">
<xs:complexType>
<xs:sequence>
<xs:group  ref="expression" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 3-46: XSD specification for the ‘parenthesized-expression’ expression

Figure 3-47 presents the ‘array-access-expression’ element and Figure 3-48 presents its XSD specification.

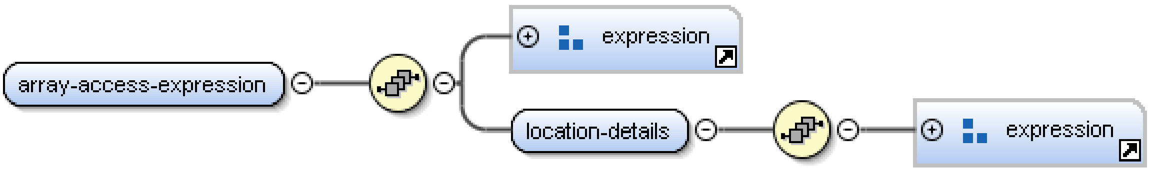


Figure 3-47: The ‘array-access-expression’ element

```
<xs:element name="array-access-expression">

<xs:complexType>

<xs:sequence>

<xs:group ref="expression" />

<xs:element name="location-details">

<xs:complexType>

<xs:sequence>

<xs:group ref="expression" />

</xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element>
```

Figure 3-48: The XSD specification for ‘array-access-expression’

Figure 3-49 presents the ‘array-creation-expression’ element and Figure 3-50 presents its XSD speciation.

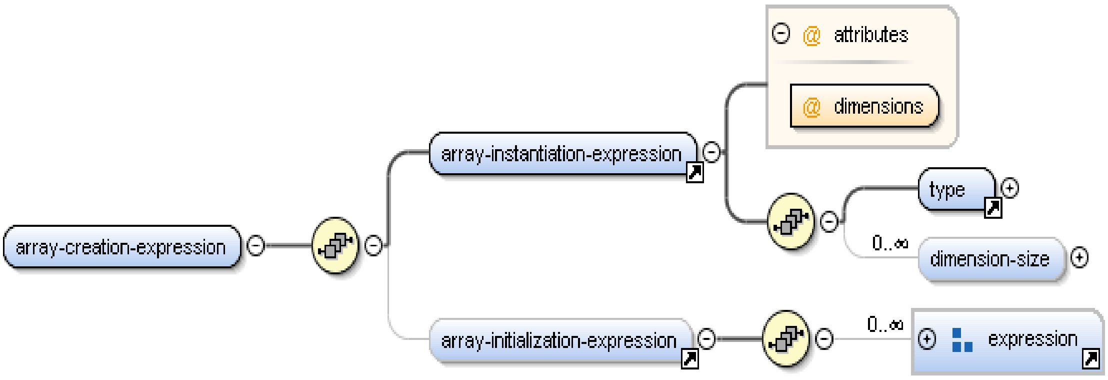


Figure 3-49: The ‘array-creation-expression

```
<xs:element name="array-creation-expression">
<xs:complexType>
<xs:sequence>
<xs:element ref="array-instantiation-expression" />
<xs:element ref="array-initialization-expression" minOccurs="0" maxOccurs="1" />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 3-50: The ‘array-creation-expression’

3.4 Transformation Algorithms and Functions

This section presents the main transformation functions and adaptation algorithms. Eclipse AST (Abstract Syntax Tree) API has been used to parse and compile Java source code and generate an abstract syntax tree. Upon invoking the eclipse AST API, a tree data structure is returned that represents the complete source code. In this data structure, every programming construct is represented by an object with unique attributes and methods, and contains a tree of the elements. For example the statement “int x = i” is represented inside the tree by an instance of the “SingleVariableDeclaration” class. This object contains information about the data type of the declared and if the variable has been initialized.

The time of the conversion process depends on the size of the program and the number of programming constructs it contains. The complexity of transformation is O(n) where n is the number of programming constructs contained in the program, or the number of nodes in the abstract syntax tree.

Figure 3-51 presents the entry point for the transformation process. The “transform” function accepts an abstract syntax tree which is an abstract representation of the source code in a form of a tree. In the same figure, the “output” function accepts an XML element and prints that element. The statement output(startElement) prints the string “<source>”.

```
function transform(Input:  abstractSyntaxTree)
Begin
   define startElement = create-start-element("source")
   out(startElement)
   for each package-declaration in abstractSyntaxTree.getPackageDeclarations()
   Element[] classNodes = package-declaratione.getAllClassNodes();
for every classNode in classNodes
         transformClass(classNode)
end for loop;
   Element[] interfaceNodes = abstractSyntaxTree.getInterfaceNodes();


for every interfaceNode in interfaceNodes
        transformInterface(classNode)
end   for  loop;
define endElement   = create-end-element("source")
 output(endElement)
End
```

Figure 3-51: The entry point function for the transformation

Figure 3-52 presents the algorithms which transforms a class element into its corresponding representation in SDL.

```
function transformClass(Input : classNode)
Begin
        define startTag = create_start_tag("class");
        if classNode.isAbstract then
                add_attribute_to_tag(startTag, "is-abstract", "true")
        else
                add_attribute_to_tag(startTag, "is-abstract", "false")
        end if
        if classNode.isFinal then
                add_attribute_to_tag(startTag, "is-inheritable", "true")
        else
           add_attribute_to_tag(startTag, "is-inheritable", "false"
        end if
        add_attribute_to_tag(startTag, "name", classTag.name)
        output(startTag)
     element[] methodDeclarations = classNode.getMethodElements()
     for each methodDeclaration in methodElements
        loop
                transformMethod(methodDeclaration)
        end loop

        element[] fieldDeclarations = classNode.getFields()
        for each fieldDeclaration in fieldDeclarations
        loop
                transformField(fieldDeclaration)
        end loop

      Element[] implementedInterfaceElements =   classNode.implementedInterfaces()
     For (each implementedInterfaceElement in implementedInterfaceElements)
      Loop
      define implementsStartTag = create_start_tag("interfece")
      output(implementsStartTag)
       out(implementedInterfaceElement.fullyQualifiedName)
      define implementsEndTag
      output(implementsEndTag)
      End Loop
      define inherited_class = classNode.inheritedClass
      define extendsStartTag= create_start_tag("extends")
      output(extendsStartTag)
      output(inherited_class.fullyQualifiedClassName)
      define extendsEndTag= create_end_tag("extends")
     output(extendsEndElement)
      define classEndTag = create_end_tag("class")
      output(classEndTag)
End
```

Figure 3-52: The transformation function for the class construct

For every statement and expression, there is a transformation function that transforms it into SDL. Figure 3-23 and Figure 3-24 presented the elements representing expressions and statements. Each element of these elements corresponds to a programming language construct. Statement constructs are contained within method bodies and constructors.

Transformation functions for those constructs iterate through all statements and calls the "transform_statement" function as illustrated in Figure 3-53. Figure 3-54 presents the "transform_statement" function.

```
for statement_node in statements
loop
transform_statement(statement_node)
end loop
```

Figure 3-53: Transformation of statements

```
function transform_statement(Input any_statement)
Begin
if (any_statement is if_statement) then transform_if_statement(any_statement)
if (any_statement is switch_statement) then transform_switch_statement(any_statement)
if (any_statement is while_statement) then transform_while_statement(any_statement)
if (any_statement is break_statement) then transform_break_statement)
if (any_statement is method_call then transform_expression(method_call_statement)
if (any_statement is local_variable_declaration_statement) then
transform_local_variable_declaration(any_statement)
if (any_statement is instantiation) then transform_instantiation(any_statement)
if (any_statement is constructor_call) then transform_constructor_call(any_statement)
if (any_statement is super_constructor_call) then transform_super_constructor_call(any_statement)
if (any_statement is instantiation) then transform_instantiation(any_statement)
End
```

Figure 3-54: The transform_statement function

Section 2 of this chapter showed that some included features in SDL are not exactly the same in the three languages. These features include, the "switch" statement, the for loop statement. This section also presents the algorithms that transform "loop" statements into "while" structures in VB.NET. Figure 3-55 describes the transformation process.

1. Start an if(true) statement
2. Execute all loop initialization statements
3. Open 'while do loop' with its condition be the stop condition for the original for loop.
4. For every statement inside the while loop execute steps 5 and 8, 9
5. If the statement is a 'continue' statement, execute the steps from 6 and 7
6. Execute the loop step statements
7. Execute the continue statement
8. If the statement is not a 'continue' statement , execute the statement
9. If the statement is the last statement in the body of the 'loop' statement, execute the step statements.
10. Close the 'while' Loop
11. Close the If (True) statement.

Figure 3-55: Transform "for" statement into "while" statement

Figure 3-56 shows the structure of a "switch" case in both C++ and Java.

```
swtich (switch-expression) {
    case constant_1: A sequence of Statements (May or may not contain a break statement) ;
    case constant_n: A sequence of Statements (May or may not contain a break statement);
    default : Sequence of statements
}
```

Figure 3-56: The structure of the "Switch" statement in C++ and Java

Figure 3-57 shows the steps to adapt the switch from SDL into VB.NET. This adaptation is necessary because VB.NET switch statement does not support the flow feature.

```
Open an if true statement as below
IF TRUE THEN
Explain:Declare a variable that delcares whether a case has been matched and initialize it to false
Dim match_var As Boolean = FALSE
Explain:Declared a variable that declare whether there is an execution flow or not (When a case is match
and there is no break statement)
Dim flow_var AS Boolean = FALSE
For Every case create an if statement as follows
IF (case_constant = expression OR flow_var) THEN
   For every statement inside the case statement , put the statement here as is.
   If there is no break statement then create the statement (flow_var = TRUE)
   Create the statement (match_var = TRUE)
END IF
if there is a default statement create the following if statement
IF (NOT match_var OR flow_var) THEN
Put the statments of the default case here as is.
END
END IF
```

Figure 3-57: Adapting the "switch" statement into VB.NET

Chapter Four
Experimental Results

In order to validate the proposed description language, we developed a program to transform Java source code into SDL, and another one for transforming SDL into VB.NET. Validation cases include transforming source code from Java into SDL, then from SDL into VB.NET and executing both programs to compare the results of executions. Validation cases have been designed to cover the main programming elements, as well as object oriented features and popular algorithms. This chapter shows fragments of the validation cases in the three languages. This chapter is organized into subsections each section covering a family of the validation cases. The chapter starts with the primitive validation cases that covers the basic programming elements and then moves to the more complicated cases such as algorithms. Each validation case shows the source code in Java and the transformed source code in VB.NET and fragments of the SDL representation of the validation case.

4.1 Switch Statements

As specified in 3.2.2, “switch” statements are converted into a series of conditional “if “statements in VB.NET. Every case including the default case is transformed into an “if” statement. A flag variable is used to determine if a “case” block used the “break” statement to break the flow or not. Also another flag is used to indicate whether a case has been matched or not. This validation case includes a program that prints a sequence of characters based on an input variable. If 0 is passed, it prints characters from A to D, and if 1 is passed it prints the characters from B to D and so on. Figures 4-1 and 4-2 show the source code in Java and VB.NET respectively of the switch block.

```
switch  (letterStart) {
        case 0:System.out.print(" A");          case 1:System.out.print(" B");
        case 2:System.out.print(" C             default:System.out.print(" D");
}
```

Figure 4-1: Java Source Code for the Switch flow program

The VB.NET code illustrates how the flag variable “case_applied7” is set to true inside the “if” statement corresponding to each case to indicate that a case has been matched. The “case_applied7” is used by the “if” statement corresponding to the “default” case as this statement will be executed if no case has been matches, or a case has been matches and the “flow_7” variable is set to true.

```
IF True
Dim  flow_7  As  Boolean  =  False
 Dim  case_applied7  As  Boolean  =  False
  If  0  =  letterStart  Or  flow_7  Then
 System.Console.Write(" A")
      case_applied7  =  True
      flow_7 =  True
 End If
If  1  =  letterStart  Or  flow_7  Then
   System.Console.Write(" B")
   case_applied7  =  True
   flow_7 =  True
End If
If  2  =  letterStart  Or  flow_7  Then
  System.Console.Write(" C")
  case_applied7  =  True
  flow_7 =  True
 End If
 If flow_7  OR  Not  case_applied7 Then
    System.Console.Write(" D")
 End If
End IF
```

Figure 4-2: VB.NET Source Code for the Switch flow program

Table 4-1 shows fragments of XML elements in SDL of the same program along with the explanation. Refer to figures 52-53 for the specification of the “switch” element.

Table 4-1: Switch Statement Components in XML

Fragment	Comments
<switch> … </switch>	All constituents of the switch elements are part of the main switch tag.
<switch-expression> <direct-variable-reference name=”letterStart” /> </switch-expression>	Directly under the switch element is the “switch-expression” which may contain any expression element. This fragment contains a variable reference to a variable named “letterStart”.
<case> <case-expression> <numeric-literal>0</numeric-literal> </case-expression> <method-call> <method-name>print</method-name> <parameters> <string-literal>A</string-literal> </parameters> <direct-variable-reference name="out"> <type> <object-type>java.lang.System</object-type> </type> </direct-variable-reference> </method-call> </case>	“case” blocks directly comes after the “switch-expression” block and represent “case” blocks in the “switch” statement. This fragment shows the first “case” block in the Java program, which also corresponds to the first “if” statement in the VB.NET code. The “case-expression” element may contain any expression element and in this example it contains a numeric literal element which has a value of zero. The “case-expression” must be present in every case element. Directly after the “case-expression” element, come the statements to be executed if the case is matched. This fragment uses the “method-call” element. The “method-call” element has no attributes and contains a “method-name” element to specify method name. The “method-call” element also contains the “parameters” element which directly comes after the “method-name” element. It specifies the parameters passed to the method and it may contain any number of expression elements. The last element in the “method-call” element specifies entity on which the method is called.

4.2 Conditional Statements

This validation case is a program that prints the grade of a student, such as "Excellent" and "Very good" based on his GPA. It does so by using an "if/else" statement. This validation case also shows the use of method declarations and referencing variables in SDL. Figures 4-3 and Figure 4-4 show the source code of a method that accepts the grade as a floating point variable and prints the grade name accordingly.

```
public  static void printUniversityGrade(double grade) {
    if(grade >= 85) {
        System.out.println("Excellent");
    } else if(grade >= 75) {
        System.out.println("Very Good");
    } else if(grade >= 65) {
        System.out.println("Good");
    } else if (grade >= 50) {
        System.out.println("Acceptable");
    } else {
        System.out.println("Untilmate Failure");
    }
}
```

Figure 4-3: Java Source Code for the Conditional Statements Program

Although this thesis does not develop a transformation model for language APIs, some simple language functions have been transformed. The "System.out.println" in Java has been transformed into "System.Console.WriteLine" in VB.NET.

```
Public Shared Sub printUniversityGrade(grade As Double)
         If grade >= 85 Then
        System.Console.WriteLine("Excellent")
         Else  If grade >= 75 Then
        grade >= 75
        System.Console.WriteLine("Very Good")
         Else If grade >= 65 Then
        grade >= 65
        System.Console.WriteLine("Good")
        End If
        Else If grade >= 50 Then
        System.Console.WriteLine("Acceptable")
         Else
        System.Console.WriteLine("Untilmate Failure")
         End If
        End Sub
```

Figure 4-4: VB.NET Source Code for the Conditional Statements Program

Table 4-2 shows the fragments of the conditional program in SDL along with the explanation of the elements. Refer to figures 3-29, 3-15 ,and 3-17 for the specification of “if”, “method”, “variable-data-declaration” elements respectively.

Table 4-2: Fragments of the SDL code for the conditional statements

Fragment	Comments
<method access-modifier=”public” is-overridable=”yes” scope=”class”> <method-spec name=” printUniversityGrade”> … </method-spect> <method-body> … </method-body> </method>	The “method” element declares a method with its specification and body. All other elements are contained with this element. The scope=”class” attribute specifies that this method is a static method. The access-modifier=”public” specifies that the method is accessible and available for any code outside the class.
<method-spec name="printUniversityGrade"> <type> <primitive-type>VOID</primitive-type> </type> <variable-data-declaration name="grade" fixed="no"> <type> <primitive-type>SIGNED_FLOAT_EIGHT_BYTES</primitive-type> </type> </variable-data-declaration> </method-spec>	The “method-spec” element specifies the name of the declared method through the “name” attribute and it is the first element inside the “method” element. It contains a “type” element to specify the return type of the method. In this case it is a primitive of type “void”. Directly after the type declaration, there comes a sequence of “variable-data-declaration” elements to declare the parameters of the methods. A “variable-data-declaration” element simply specifies the name of the variable through the “name” attribute, and whether the variable is fixed or not. It contains a type element to specify the type of the declared variable.
<method-body> … </method-body>	Directly after the “method-spec” element there is a “method-body” element that contains a sequence of statement elements.

<if> <if-main> … </if-main> </if>	Inside the "method-body" element there is one "if" element representing the main if statement in both programs. This element contains other "else-if" elements and the "else" element" and also the statements that print the grade.
<if-main> <if-expression> <boolean-comparison-expression> <direct-variable-reference name="grade" /> <comparison-operator>greater-equals</comparison-operator> <numeric-literal>85</numeric-literal> </boolean-comparison-expression> </if-expression> <method-call> … </method-call> </if-main>	The "if-main" element represents the code that checks whether the grade is greater or equal to 85 in this fragment. It is the first element inside the "if" element and contains a "boolean-comparison-expression" element that represents the logical comparison. It uses the "comparison-operator" element to specify the type of comparison. Directly after the "if-main" element there is a "method-call" element that prints the grade.

4.3 Bitwise Expression

This validation case shows a function that uses the bitwise operators to print the binary form of a number. For example if it passed 4, it would print "100". It does so by performing bitwise "and" with 1 to figure out the rightmost digit and then performing logical right shift with a magnitude of 1. It keeps repeating these steps until the number becomes zero. This validation case also shows the use of "loop" structures in SDL and declaration of local variables. Table 4-3 shows the code of the binary form in both VB.NET and Java.

Table 4-3: Java source code of the binary form program

Source Code in Java	Source Code in VB.NET

ArrayList digits = new ArrayList();
while(number != 0) {
int rightmost = number & 1;
digits.add(rightmost);
number = number >> 1;
}
int digitsSize = digits.size();
for(int i = digitsSize -1; i >= 0; i = i-1) {
System.out.print(digits.get(i));
}
	Public Shared Sub print(number As Integer)
	Dim digits AS System.Collections.ArrayList = new System.Collections.ArrayList()
	While number<> 0
	Dim rightmost AS Integer = number And 1
	digits.add(rightmost)
	number = number >> 1
	End While
	Dim digitsSize AS Integer = digits.Count
	If True
	Dim i AS Integer = digitsSize - 1
	While i >= 0
	System.Console.Write(digits.Item(i))
	i = i - 1
	End While
	End If
	System.Console.WriteLine()
	End Sub

Table 4-4 shows fragments of the same logic in SDL along with their explanations.

Table 4-4: Fragments of the binary form program in SDL

Fragment	Comments
<local-variable-declaration> …. </local-variable-declaration>	The list data structure is declared using the “local-variable-declaration” element. This element contains all other element related to variable declaration and initialization.
<variable-data-declaration name="digits" fixed="no"> <type> <object-type>java.util.ArrayList</object-type> </type> </variable-data-declaration>	A “variable-data-declaration” element is the first element inside the ”local-variable-declaration” element and it specifies the variable name and type. In this fragment the type of the declared variable is “java.util.ArrayList” and is not a primitive type.
<variable-initialization> <instantiation> <type> <object-type>java.util.ArrayList</object-type> </type> <parameters /> </instantiation> </variable-initialization>	The list variable is initialized and assigned a new instance of “java.util.ArrayList”. The “variable-initialization” element comes directly after the ”variable-data-declaration” in case the variable is initialized and it contains one expression element. The “instantiation” element represents calling the constructor of an object and passing it parameters. In this fragment, no parameters are passed to the constructor.
<while type=”while-do”> … </while>	A “while” element comes directly after the “local-variable-declaration”. The attribute “while-do” indicate this is a normal while loop. This element contains a “loop-expression” element and a sequence of statement elements after this element.
<loop-expression> <boolean-comparison-expression> <direct-variable-reference name="number" /> <comparison-operator> not-equals	The “loop-expression” element is the first element inside the “while” element. It contains a “boolean-comparison-expression” element which represents arithmetic comparison between two expressions. In this example the first expression the value of the

</comparison-operator> <numeric-literal>0</numeric-literal> </boolean-comparison-expression> </loop-expression>	variable “number” and the second expression is the literal “0” and the comparison operator is “not equals”. The statements after the “loop-expression” element constitute the body of the loop.
<binary-bitwise-expression> <direct-variable-reference name="number" /> <binary-bitwise-operator>and</binary-bitwise-operator> <numeric-literal>1</numeric-literal> </binary-bitwise-expression>	In this fragment, the “binary-bitwise-expression” element represents a bitwise “and” operation. This element must consist of one expression element followed by the”binary-bitwise-operator” element and followed another expression element.
<shift shift-direction="right" shift-type="arithmetic"> <shifted-expression> <direct-variable-reference name="number" /> </shifted-expression> <magnitude-expression> <numeric-literal>1</numeric-literal> </magnitude-expression> </shift>	The “shift element in this fragment represents an arithmetic right shift with a magnitude of two. The “shift-direction” attribute in this fragment specifies that the shift direction is right and the “type” attribute specifies that it is a logical shift. The first element inside the “shift” element is the “shifted-expression” element which must contain one expression element and this represents the expression to be shifted. The second element inside the “shift” element is the “magnitude-expression” element which must contain one expression element which evaluates to a numeric value.

4.4 Arrays

This validation case shows a function that declares a two dimensional array. The elements of the two dimensional array are initialized such that element(x)(y) = x if y has a value of 0 and element element(x)(y) = x * y otherwise. This function uses two nested loops to initialize the elements of the array. Figure 4-5 and Figure 4-6 show the code fragment responsible for the initialization in Java and VB.NET.

```
int[][] numbers = new int[5][5];
for(int I = 0; I <= 4; I = I + 1) {
        for (int j = 0; j <= 4; j = j +1) {
                if(j == 0 ) {
                        numbers[i][j] = I;
                } else {
                        numbers[i][j] = I * j;
                }
        }}
```

Figure 4-5: Java code fragment for the array initialization

```
Dim numbers AS Integer(,) =  New Integer(5,5){}
 If   True
       Dim i AS Integer = 0
        While i<=4
            If   True
               Dim j AS Integer = 0
                   While j<=4
                         If  j = 0 Then
                             numbers(j,i) = i
                          Else
                             numbers(j,i) = i * j
                          End If
                       j = j + 1
                    End While
        End If
       i = i + 1
       End While
  End If
```

Figure 4-6: VB.NET code fragment for the array initialization

Table 4-5 shows fragments of the SDL representation of the same code along with the explanations. Refer to Figure 3-24 for the specification of the “loop” element and to Figure 3-42 for the specification of the “array-access” and to figure 3-32 for the specification of the “arithmetic-expression” elements used in the SDL fragments and to figure 3-47 for the specification of “array-creation” element.

Table 4-5: SDL fragments for the arrays validation case

Fragment	Comments
<array-creation-expression> <array-instantiation-expression dimensions="2"> <type> <primitive-type> SIGNED_INT_FOUR_BYTES </primitive-type> </type> <dimension-size> <numeric-literal>5</numeric-literal> </dimension-size> <dimension-size> <numeric-literal>5</numeric-literal> </dimension-size> </array-instantiation-expression> </array-creation-expression>	This fragment represents the array creation part of the array declaration. It is equivalent to “new int[5][5]” in Java. The “array-creation-expression” element contains one “array-instantiation-expression” element. This element contains a “type” element that specifies the type of the instantiated array. The “dimensions” attribute of the “array-instantiation-expression” specifies the dimensions of the array, and in this fragment it is set to 2, which means a two dimensional array. After the “type” element, there are two “dimension-size” elements that species that the first dimension size 5, and the second dimension size is 5 as well.
<loop-expression> <boolean-comparison-expression> <direct-variable-reference name="i" /> <comparison-operator> less-equals	This fragment shows the loop expression for the first loop. This is equivalent to “I <= 4” in the out loop of the Java code.

</comparison-operator> <numeric-literal>4</numeric-literal> </boolean-comparison-expression> </loop-expression>	
<loop-operation> <variable-assignment> <direct-variable-reference name="i" /> <arithmetic-expression> <direct-variable-reference name="i" /> <arithmetic-operator>addition</arithmetic-operator> <numeric-literal>1</numeric-literal> </arithmetic-expression> </variable-assignment> </loop-operation>	The fragment shows the loop operation element. It is equivalent to “I = I + 1” in the outer loop of the Java code. The “variable-assignment” element contains one “direct-variable-reference” element which denotes the variable to be assigned and an one expression element that it the expression to be assigned to the variable. In this fragment, the assigned expression is an arithmetic expression. The arithmetic expression in this element is equivalent to “I + 1”.
<array-access-expression> <array-access-expression> <direct-variable-reference name="numbers" /> <location-details> <direct-variable-reference name="i" /> </location-details> </array-access-expression> <location-details> <direct-variable-reference name="j" /> </location-details> </array-access-expression>	This fragment is equivalent to number[i][j] in Java code.

4.5 Object Oriented Programming

This validation case shows the inheritance and polymorphism features of object oriented programming. It also shows how interfaces are declared and implemented in SDL. It shows the declaration of one interface “IGreeting” declaring one method “greet”. Two classes implement this interface “EnglishGreting” and “FrenchGreeting”. Each of these classes overrides the method to print the greeting in the appropriate language. The “greet” method in the “BritishEnglishGreeting” class does a call to the original “greet” method in the “EnglishGreeting” class.

```java
package oo.polymorphism;
public interface IGreeting {
   public abstract void greet();
}
public class EnglishGreeting implements IGreeting {
   public void greet() {
      System.out.println("Hello ....");
   }
}
public class FrenchGreeting implements IGreeting {
   public void greet() {
     System.out.println("French Greeting...");
   }
}
public class BritishEnglishGreeting extends EnglishGreeting {
   public void greet() {
      super.greet();
      System.out.println("UK");
   }}
```

Figure 4-7: Java code of the object oriented validation case

```vbnet
Namespace oo.polymorphism
 Public   Interface IGreeting
       Sub greet()
        End Interface
Public  Class EnglishGreeting
        Implements  oo.polymorphism.IGreeting
         Public Overridable Sub greet()
       System.Console.WriteLine("Hello ....")
      End Sub
 End Class
Public  Class FrenchGreeting
Implements  oo.polymorphism.IGreeting
   Public Overridable Sub greet()
   System.Console.WriteLine("French Greeting...")
   End Sub
End Class

End Namespace
```

```vbnet
Public  Class BritishEnglishGreeting
 Inherits
oo.polymorphism.EnglishGreeting
       Public Overridable Sub greet()
       MyBase.greet()
       System.Console.WriteLine("UK")
       End Sub
     End Class
```

Figure 4-8: VB.NET code of the object oriented validation case

Table 4-6 shows fragments of the SDL representation of this validation case. Refer to Figures 3-7, 3-9, 3-11 for the specification of the “package”, “class”, “interface” elements respectively.

Table 4-6: Fragments of the SDL representation of the object oriented validation case

Fragment	Comments
<package name="oo.polymorphism"> … </package>	All “class” and “interface” elements are inside the “package” element. This fragment also specifies the name of the package “oo.polymorphism”
<interface name="IGreeting"> <method-spec name="greet"> <type> <primitive-type>VOID</primitive-type> </type> </method-spec> </interface>	The fragment represent the declaration of the “IGreeting” interface. The “method-spec” element is used to specify the declaration of the “greet” method.
<class name="EnglishGreeting" is-abstract="no" is-inheritable="yes"> <implements> <type> <object-type>oo.polymorphism.IGreeting</object-type> </type> </implements> … </class>	This fragment shows the declaration of the EnglishGreeting class. The “is-abstract” indicates that the class is not abstract. The implemented interfaces are specified through the “implements” element. For every implemented interface, there is a “type” element inside the “implements” element to declare the implementation of that interface.
<class name="BritishEnglishGreeting" is-abstract="no" is-inheritable="yes"> <extends> <type> <object-type>oo.polymorphism.EnglishGreeting</object-type> </type> </extends>	This fragment shows the declaration of the “BritishEnglishGreeting” class. It clarifies the usage of the “extends” element to declare class inheritance.

4.6 Sorting Algorithms

This validation case shows two of the most popular sorting algorithms. Sorting algorithms accept a list L[l1, l2, l3, l4 .. l(n-1), l(n)] such that L(i+1) may be greater, less than or equal to L(i) and produces a list T such that T(i+1) is greater than or equal T(i) if the sorting algorithm is in ascending order. The insertion sort starts the sorting process by taking the first element in the array and considers as a sorted array of one element. It then iterates through all other elements from 1 to n, where n is the index of the last element and inserts each element at its proper location in the sorted array. Each iteration grows the size of the sorted array by one until all the elements become sorted.

Figures 4-9 and 4-10 show the code of insertion sort in Java and VB.NET respectively.

```
 package sorting
public class InsertionSort {
   public static void sort(int[] numbers) {
      int numberOfElements = numbers.length;
      for (int I = 1; I <= numberOfElements – 1; I = I + 1) {
         int element = numbers[i];
         int j = i- 1;
         while (j >= 0) {
            if(element > numbers[j]) {
               break;
            } else {
               int temp = numbers[j];
               numbers[j] = element;
               numbers[j+1] = temp;
               j = j -1;
            }         }       }     }  }
```

Figure 4-9: Java code of the insertion sort

```
Namespace sorting
Public  Class InsertionSort
Public Shared Sub sort(numbers As Integer())
Dim numberOfElements AS Integer = numbers.length
If   True
Dim i AS Integer = 1
While i<=numberOfElements - 1
Dim element AS Integer = numbers(i)
Dim j AS Integer = i - 1
While j >= 0
If element > numbers(j) Then
Exit  While
Else
Dim temp AS Integer = numbers(j)
numbers(j) = element
numbers(j + 1)= temp
j = j - 1
End If
End While
i = i + 1
End While
End If
End Sub
```

Figure 4-10: VB.NET code of the insertion sort

Figure 4-11 shows the complete representation of the insertion sort program in SDL.

```
<source>
<package name="sorting">
<class name="InsertionSort" is-abstract="no" is-inheritable="yes">
<method access-modifier="public" is-overridable="yes" scope="class">
<method-spec name="sort">
<type>
<primitive-type>VOID</primitive-type>
</type>
<variable-data-declaration name="numbers" fixed="no">
<type>
<array-type dimensions="1">
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</array-type>
</type>
</variable-data-declaration>
</method-spec>
<method-body>
<local-variable-declaration>
<variable-data-declaration name="numberOfElements" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<direct-variable-reference name="length">
<direct-variable-reference name="numbers" />
</direct-variable-reference>
</variable-initialization>
</local-variable-declaration>
<loop>
<loop-initialization>
<local-variable-declaration>
<variable-data-declaration name="i" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<numeric-literal>1</numeric-literal>
</variable-initialization>
</local-variable-declaration>
</loop-initialization>
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="i" />
<comparison-operator>less-equals</comparison-operator>
<arithmetic-expression>
<direct-variable-reference name="numberOfElements" />
<arithmetic-operator>subtraction</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</boolean-comparison-expression>
</loop-expression>
<loop-operation>
<variable-assignment>
<direct-variable-reference name="i" />
<arithmetic-expression>
```

```xml
<direct-variable-reference name="i" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</loop-operation>
<local-variable-declaration>
<variable-data-declaration name="element" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="i" />
</location-details>
</array-access-expression>
</variable-initialization>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="j" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<arithmetic-expression>
<direct-variable-reference name="i" />
<arithmetic-operator>subtraction</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-initialization>
</local-variable-declaration>
<while while-type="while-do">
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="j" />
<comparison-operator>greater-equals</comparison-operator>
<numeric-literal>0</numeric-literal>
</boolean-comparison-expression>
</loop-expression>
<if>
<if-main>
<if-expression>
<boolean-comparison-expression>
<direct-variable-reference name="element" />
<comparison-operator>greater</comparison-operator>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="j" />
</location-details>
</array-access-expression>
</boolean-comparison-expression>
</if-expression>
<break />
</if-main>
```

```xml
<else>
<local-variable-declaration>
<variable-data-declaration name="temp" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="j" />
</location-details>
</array-access-expression>
</variable-initialization>
</local-variable-declaration>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="j" />
</location-details>
</array-access-expression>
<direct-variable-reference name="element" />
</variable-assignment>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<arithmetic-expression>
<direct-variable-reference name="j" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</location-details>
</array-access-expression>
<direct-variable-reference name="temp" />
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="j" />
<arithmetic-expression>
<direct-variable-reference name="j" />
<arithmetic-operator>subtraction</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</else>
</if>
</while>
</loop>
</method-body>
</method>
</class>
</package>
</source>
```

Figure 4-11: Representation of insertion sort in SDL

The bubble sort algorithm performs (n – 1) iterations where n is number of element in the list. At each iteration, the minimum number is found and put at its proper location in the list. This sorts the list in ascending order. Figures 4-12 and 4-13 and 4-14 show the full program for the bubble sort in Java, VB.NET and SDL respectively.

```
package sorting;
public class BubbleSort {
   public static void sort(int[] numbers) {
      int numberOfElements = numbers.length;
      for(int i = 0; i <= numberOfElements - 1; i = i + 1) {
         int maxIndex = i;
         int j = i +1;
         while (j <=numberOfElements - 1) {
            if (numbers[maxIndex] > numbers[j]) {
               maxIndex = j;
            }
            j = j + 1;
         }
         int temp = numbers[i];
         numbers[i] = numbers[maxIndex];
         numbers[maxIndex] = temp;
      }
   } }
```

Figure 4-12: Representation of bubble sort in Java

```
Namespace sorting
        Public  Class BubbleSort
       Public Shared Sub sort(numbers As Integer())
        Dim numberOfElements AS Integer = numbers.length
        If   True
        Dim i AS Integer = 0
       While i<=numberOfElements - 1
        Dim maxIndex AS Integer = i
        Dim j AS Integer = i + 1
        While j<=numberOfElements - 1
        If numbers(maxIndex) > numbers(j) Then
       maxIndex = j
        End If
       j = j + 1
        End While
        Dim temp AS Integer = numbers(i)
       numbers(i) = numbers(maxIndex)
       numbers(maxIndex) = temp
       i = i + 1
        End While
        End If
       End Sub
```

Figure 4-13: Representation of bubble sort in VB.NET

```xml
<source>
<package name="sorting">
<class name="BubbleSort" is-abstract="no" is-inheritable="yes">
<method access-modifier="public" is-overridable="yes" scope="class">
<method-spec name="sort">
<type>
<primitive-type>VOID</primitive-type>
</type>
<variable-data-declaration name="numbers" fixed="no">
<type>
<array-type dimensions="1">
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</array-type>
</type>
</variable-data-declaration>
</method-spec>
<method-body>
<local-variable-declaration>
<variable-data-declaration name="numberOfElements" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<direct-variable-reference name="length">
<direct-variable-reference name="numbers" />
</direct-variable-reference>
</variable-initialization>
</local-variable-declaration>
<loop>
<loop-initialization>
<local-variable-declaration>
<variable-data-declaration name="i" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<numeric-literal>0</numeric-literal>
</variable-initialization>
</local-variable-declaration>
</loop-initialization>
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="i" />
<comparison-operator>less-equals</comparison-operator>
<arithmetic-expression>
<direct-variable-reference name="numberOfElements" />
<arithmetic-operator>subtraction</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</boolean-comparison-expression>
</loop-expression>
<loop-operation>
<variable-assignment>
<direct-variable-reference name="i" />
<arithmetic-expression>
<direct-variable-reference name="i" />
<arithmetic-operator>addition</arithmetic-operator>
```

```xml
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</loop-operation>
<local-variable-declaration>
<variable-data-declaration name="maxIndex" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<direct-variable-reference name="i" />
</variable-initialization>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="j" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<arithmetic-expression>
<direct-variable-reference name="i" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-initialization>
</local-variable-declaration>
<while while-type="while-do">
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="j" />
<comparison-operator>less-equals</comparison-operator>
<arithmetic-expression>
<direct-variable-reference name="numberOfElements" />
<arithmetic-operator>subtraction</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</boolean-comparison-expression>
</loop-expression>
<if>
<if-main>
<if-expression>
<boolean-comparison-expression>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="maxIndex" />
</location-details>
</array-access-expression>
<comparison-operator>greater</comparison-operator>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="j" />
</location-details>
</array-access-expression>
</boolean-comparison-expression>
</if-expression>
```

```xml
<variable-assignment>
<direct-variable-reference name="maxIndex" />
<direct-variable-reference name="j" />
</variable-assignment>
</if-main>
</if>
<variable-assignment>
<direct-variable-reference name="j" />
<arithmetic-expression>
<direct-variable-reference name="j" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</while>
<local-variable-declaration>
<variable-data-declaration name="temp" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="i" />
</location-details>
</array-access-expression>
</variable-initialization>
</local-variable-declaration>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="i" />
</location-details>
</array-access-expression>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="maxIndex" />
</location-details>
</array-access-expression>
</variable-assignment>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="numbers" />
<location-details>
<direct-variable-reference name="maxIndex" />
</location-details>
</array-access-expression>
<direct-variable-reference name="temp" />
</variable-assignment>
</loop>
</method-body>
</method>  </class>  </package>  </source>
```

Figure 4-14: Representation of bubble in SDL

Figures 4-15, 4-16, and 4-17 show the full program for the Merge sort in Java, VB.NET, and SDL respectively.

```
public class MergeSort {
   public static void mergesort(int[] data, int first, int n) {
      int n1;
      int n2;
      if (n > 1) {
         n1 = n / 2;
         n2 = n - n1
         mergesort(data, first, n1);
         mergesort(data, first + n1, n2);
         merge(data, first, n1, n2);
      }
   }
   public static void merge(int[] data, int first, int n1, int n2) {
      int[] temp = new int[n1 + n2];
      int copied = 0;
      int copied1 = 0;
      int copied2 = 0;
      int i;

      while ((copied1 < n1) && (copied2 < n2)) {
         if (data[first + copied1] < data[first + n1 + copied2]) {
            temp[copied] = data[first + (copied1)];
            copied = copied +1;
            copied1 = copied1 +1;
         }

         else  {
            temp[copied] = data[first + n1 + (copied2)];
            copied = copied +1;
            copied2 = copied2 +1;
         }
      }
      while (copied1 < n1) {
         temp[copied] = data[first + (copied1)];
         copied = copied +1;
         copied1 = copied1 +1;
      }

      while (copied2 < n2) {
         temp[copied] = data[first + n1 + (copied2)];
         copied = copied +1;
         copied2 = copied2 +1;

      }

      for (i = 0; i < n1 + n2; i = i +1)
         data[first + i] = temp[i];
   }
}
```

Figure 4-15: Representation of Merge Sort in Java

```
Public  Class MergeSort


        Public Shared Sub mergesort(data As Integer(),first As Integer,n As Integer)
            Dim n1 AS Integer
            Dim n2 AS Integer
            If n > 1 Then
                n1 = n / 2
               n2 = n - n1
               mergesort(data , first , n1)
               mergesort(data , first + n1 , n2)
               merge(data , first , n1 , n2)
            End If
        End Sub

        Public Shared Sub merge(data As Integer(),first As Integer,n1 As Integer,n2 As Integer)

         Dim temp AS Integer() =  New Integer(n1 + n2){}
         Dim copied AS Integer = 0
         Dim copied1 AS Integer = 0
         Dim copied2 AS Integer = 0
         Dim i AS Integer
         While (copied1<n1) And (copied2<n2)
         If data(first + copied1)<data(first + n1 + copied2) Then
         temp(copied) = data(first + (copied1))
         copied = copied + 1
         copied1 = copied1 + 1
         Else
         temp(copied) = data(first + n1 + (copied2))
         copied = copied + 1
         copied2 = copied2 + 1
         End If
         End While

         While copied1<n1
         temp(copied) = data(first + (copied1))
         copied = copied + 1
         copied1 = copied1 + 1
         End While

        While copied2<n2
        temp(copied) = data(first + n1 + (copied2))
        copied = copied + 1
        copied2 = copied2 + 1
        End While

        If   True
        i = 0
        While i<n1 + n2
        data(first + i) = temp(i)
        i = i + 1
        End While
        End If
        End Sub

End Class
```

Figure 4-16: Representation of Merge Sort in VB.NET

```xml
<class name="MergeSort" is-abstract="no" is-inheritable="yes">
<method access-modifier="public" is-overridable="yes" scope="class">
<method-spec name="sort">
<type>
<primitive-type>VOID</primitive-type>
</type>
<variable-data-declaration name="numbers" fixed="no">
<type>
<array-type dimensions="1">
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</array-type>
</type>
</variable-data-declaration>
</method-spec>
<method-body>
<local-variable-declaration>
<variable-data-declaration name="i" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="numberOfElements" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<direct-variable-reference name="length">
<direct-variable-reference name="numbers" />
</direct-variable-reference>
</variable-initialization>
</local-variable-declaration>
<loop>
<loop-initialization>
<variable-assignment>
<direct-variable-reference name="i" />
<numeric-literal>0</numeric-literal>
</variable-assignment>
</loop-initialization>
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="i" />
<comparison-operator>less</comparison-operator>
<direct-variable-reference name="numberOfElements" />
</boolean-comparison-expression>
</loop-expression>
<loop-operation>
<variable-assignment>
<direct-variable-reference name="i" />
<arithmetic-expression>
<direct-variable-reference name="i" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</loop-operation>
<method-call>
```

```
<method-name>mergesort</method-name>
<parameters>
<direct-variable-reference name="numbers" />
<numeric-literal>0</numeric-literal>
<direct-variable-reference name="numberOfElements" />
</parameters>
</method-call>
</loop>
</method-body>
</method>
<method access-modifier="public" is-overridable="yes" scope="class">
<method-spec name="mergesort">
<type>
<primitive-type>VOID</primitive-type>
</type>
<variable-data-declaration name="data" fixed="no">
<type>
<array-type dimensions="1">
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</array-type>
</type>
</variable-data-declaration>
<variable-data-declaration name="first" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-data-declaration name="n" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
</method-spec>
<method-body>
<local-variable-declaration>
<variable-data-declaration name="n1" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="n2" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
</local-variable-declaration>
<if>
<if-main>
<if-expression>
<boolean-comparison-expression>
<direct-variable-reference name="n" />
<comparison-operator>greater</comparison-operator>
<numeric-literal>1</numeric-literal>
</boolean-comparison-expression>
</if-expression>
<variable-assignment>
<direct-variable-reference name="n1" />
```

```xml
<arithmetic-expression>
<direct-variable-reference name="n" />
<arithmetic-operator>division</arithmetic-operator>
<numeric-literal>2</numeric-literal>
</arithmetic-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="n2" />
<arithmetic-expression>
<direct-variable-reference name="n" />
<arithmetic-operator>subtraction</arithmetic-operator>
<direct-variable-reference name="n1" />
</arithmetic-expression>
</variable-assignment>
<method-call>
<method-name>mergesort</method-name>
<parameters>
<direct-variable-reference name="data" />
<direct-variable-reference name="first" />
<direct-variable-reference name="n1" />
</parameters>
</method-call>
<method-call>
<method-name>mergesort</method-name>
<parameters>
<direct-variable-reference name="data" />
<arithmetic-expression>
<direct-variable-reference name="first" />
<arithmetic-operator>addition</arithmetic-operator>
<direct-variable-reference name="n1" />
</arithmetic-expression>
<direct-variable-reference name="n2" />
</parameters>
</method-call>
<method-call>
<method-name>merge</method-name>
<parameters>
<direct-variable-reference name="data" />
<direct-variable-reference name="first" />
<direct-variable-reference name="n1" />
<direct-variable-reference name="n2" />
</parameters>
</method-call>
</if-main>
</if>
</method-body>
</method>
<method access-modifier="public" is-overridable="yes" scope="class">
<method-spec name="merge">
<type>
<primitive-type>VOID</primitive-type>
</type>
<variable-data-declaration name="data" fixed="no">
<type>
<array-type dimensions="1">
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</array-type>
</type>
</variable-data-declaration>
```

```
<variable-data-declaration name="first" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-data-declaration name="n1" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-data-declaration name="n2" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
</method-spec>
<method-body>
<local-variable-declaration>
<variable-data-declaration name="temp" fixed="no">
<type>
<array-type dimensions="1">
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</array-type>
</type>
</variable-data-declaration>
<variable-initialization>
<array-creation-expression>
<array-instantiation-expression dimensions="1">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
<dimension-size>
<arithmetic-expression>
<direct-variable-reference name="n1" />
<arithmetic-operator>addition</arithmetic-operator>
<direct-variable-reference name="n2" />
</arithmetic-expression>
</dimension-size>
</array-instantiation-expression>
</array-creation-expression>
</variable-initialization>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="copied" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<numeric-literal>0</numeric-literal>
</variable-initialization>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="copied1" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
```

```xml
<numeric-literal>0</numeric-literal>
</variable-initialization>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="copied2" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
<variable-initialization>
<numeric-literal>0</numeric-literal>
</variable-initialization>
</local-variable-declaration>
<local-variable-declaration>
<variable-data-declaration name="i" fixed="no">
<type>
<primitive-type>SIGNED_INT_FOUR_BYTES</primitive-type>
</type>
</variable-data-declaration>
</local-variable-declaration>
<while while-type="while-do">
<loop-expression>
<binary-logical-expression>
<parenthesized-expression>
<boolean-comparison-expression>
<direct-variable-reference name="copied1" />
<comparison-operator>less</comparison-operator>
<direct-variable-reference name="n1" />
</boolean-comparison-expression>
</parenthesized-expression>
<binary-boolean-operator>and</binary-boolean-operator>
<parenthesized-expression>
<boolean-comparison-expression>
<direct-variable-reference name="copied2" />
<comparison-operator>less</comparison-operator>
<direct-variable-reference name="n2" />
</boolean-comparison-expression>
</parenthesized-expression>
</binary-logical-expression>
</loop-expression>
<if>
<if-main>
<if-expression>
<boolean-comparison-expression>
<array-access-expression>
<direct-variable-reference name="data" />
<location-details>
<arithmetic-expression>
<direct-variable-reference name="first" />
<arithmetic-operator>addition</arithmetic-operator>
<direct-variable-reference name="copied1" />
</arithmetic-expression>
</location-details>
</array-access-expression>
<comparison-operator>less</comparison-operator>
<array-access-expression>
<direct-variable-reference name="data" />
<location-details>
<infix-expression expression-type="arithmetic-expression">
```

```
<arithmetic-operator>addition</arithmetic-operator>
<direct-variable-reference name="first" />
<direct-variable-reference name="n1" />
<direct-variable-reference name="copied2" />
</infix-expression>
</location-details>
</array-access-expression>
</boolean-comparison-expression>
</if-expression>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="temp" />
<location-details>
<direct-variable-reference name="copied" />
</location-details>
</array-access-expression>
<array-access-expression>
<direct-variable-reference name="data" />
<location-details>
<arithmetic-expression>
<direct-variable-reference name="first" />
<arithmetic-operator>addition</arithmetic-operator>
<parenthesized-expression>
<direct-variable-reference name="copied1" />
</parenthesized-expression>
</arithmetic-expression>
</location-details>
</array-access-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied" />
<arithmetic-expression>
<direct-variable-reference name="copied" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied1" />
<arithmetic-expression>
<direct-variable-reference name="copied1" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</if-main>
<else>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="temp" />
<location-details>
<direct-variable-reference name="copied" />
</location-details>
</array-access-expression>
<array-access-expression>
<direct-variable-reference name="data" />
<location-details>
<infix-expression expression-type="arithmetic-expression">
<arithmetic-operator>addition</arithmetic-operator>
```

```xml
<direct-variable-reference name="first" />
<direct-variable-reference name="n1" />
<parenthesized-expression>
<direct-variable-reference name="copied2" />
</parenthesized-expression>
</infix-expression>
</location-details>
</array-access-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied" />
<arithmetic-expression>
<direct-variable-reference name="copied" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied2" />
<arithmetic-expression>
<direct-variable-reference name="copied2" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</else>
</if>
</while>
<while while-type="while-do">
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="copied1" />
<comparison-operator>less</comparison-operator>
<direct-variable-reference name="n1" />
</boolean-comparison-expression>
</loop-expression>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="temp" />
<location-details>
<direct-variable-reference name="copied" />
</location-details>
</array-access-expression>
<array-access-expression>
<direct-variable-reference name="data" />
<location-details>
<arithmetic-expression>
<direct-variable-reference name="first" />
<arithmetic-operator>addition</arithmetic-operator>
<parenthesized-expression>
<direct-variable-reference name="copied1" />
</parenthesized-expression>
</arithmetic-expression>
</location-details>
</array-access-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied" />
<arithmetic-expression>
```

```xml
<direct-variable-reference name="copied" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied1" />
<arithmetic-expression>
<direct-variable-reference name="copied1" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</while>
<while while-type="while-do">
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="copied2" />
<comparison-operator>less</comparison-operator>
<direct-variable-reference name="n2" />
</boolean-comparison-expression>
</loop-expression>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="temp" />
<location-details>
<direct-variable-reference name="copied" />
</location-details>
</array-access-expression>
<array-access-expression>
<direct-variable-reference name="data" />
<location-details>
<infix-expression expression-type="arithmetic-expression">
<arithmetic-operator>addition</arithmetic-operator>
<direct-variable-reference name="first" />
<direct-variable-reference name="n1" />
<parenthesized-expression>
<direct-variable-reference name="copied2" />
</parenthesized-expression>
</infix-expression>
</location-details>
</array-access-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied" />
<arithmetic-expression>
<direct-variable-reference name="copied" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
<variable-assignment>
<direct-variable-reference name="copied2" />
<arithmetic-expression>
<direct-variable-reference name="copied2" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
```

```
</while>
<loop>
<loop-initialization>
<variable-assignment>
<direct-variable-reference name="i" />
<numeric-literal>0</numeric-literal>
</variable-assignment>
</loop-initialization>
<loop-expression>
<boolean-comparison-expression>
<direct-variable-reference name="i" />
<comparison-operator>less</comparison-operator>
<arithmetic-expression>
<direct-variable-reference name="n1" />
<arithmetic-operator>addition</arithmetic-operator>
<direct-variable-reference name="n2" />
</arithmetic-expression>
</boolean-comparison-expression>
</loop-expression>
<loop-operation>
<variable-assignment>
<direct-variable-reference name="i" />
<arithmetic-expression>
<direct-variable-reference name="i" />
<arithmetic-operator>addition</arithmetic-operator>
<numeric-literal>1</numeric-literal>
</arithmetic-expression>
</variable-assignment>
</loop-operation>
<variable-assignment>
<array-access-expression>
<direct-variable-reference name="data" />
<location-details>
<arithmetic-expression>
<direct-variable-reference name="first" />
<arithmetic-operator>addition</arithmetic-operator>
<direct-variable-reference name="i" />
</arithmetic-expression>
</location-details>
</array-access-expression>
<array-access-expression>
<direct-variable-reference name="temp" />
<location-details>
<direct-variable-reference name="i" />
</location-details>
</array-access-expression>
</variable-assignment>
</loop>
</method-body>
</method>
</class>
```

Figure 4-17: Representation of Merge Sort in SDL.

4.7 Sample of Validation Cases Execution Results

This section shows the results of executing some of the validation cases presented in this chapter as well other validation cases such as the binary and linear search. Table 4-7 shows the inputs and outputs resulted from executing both. Java programs and VB.NET programs after transformation.

Table 4-7: Inputs and outputs for the sorting validation cases

Input Array	Sorted Array
2463 548 -750 -254 996 1975 4257 4247 3403	-750 -254 548 996 1975 2463 3403 4247 4257
4188 2960 4117 1536 1377 3114	1377 1536 2960 3114 4117 4188
2080 2694 -850 3060 3950 4096 -294 1008 -994 2573	-994 -850 -294 1008 2080 2573 2694 3060 3950 4096
3257 -253 1720 2053 1483	-253 1483 1720 2053 3257
-472 3551 3457 3285 366 4009 621 3531 4868 3901	-472 366 621 3285 3457 3531 3551 3901 4009 4868
1448 3709 2618 4375 1271 2745 4835 1775 4610	1271 1448 1775 2618 2745 3709 4375 4610 4835
3489 2629 4935 2671 -136 -145	-145 -136 2629 2671 3489 4935
3275 1589 1989 939 -518 1096 1028 4288 267 1055	-518 267 939 1028 1055 1096 1589 1989 3275 4288
-911 1405 4578 2343 2624 3485 4314 2888 -598 -459	-911 -598 -459 1405 2343 2624 2888 3485 4314 4578
1965 2586 -14 3065 3043	-14 1965 2586 3043 3065

Table 4-8 shows the results of executing the binary and linear search programs in the Java and on VB.NET after transformation from SDL.

Table 4-8: Inputs and output for the linear search and binary search programs

Input Array	Search Element	Result Index
1678 4967 -460 1525 613 3289 -530 3009 1490 4170	3289	5
-100 718 -154 4676 208 3456 2332 86	208	4
276 2369 -847 1718 -249	-847	2
-264 2346 3410 1009 1625 3033 2649 1142	1625	4
-694 1520 41 401 484 3154 674 -800 4972 -647	3154	5
304 1750 -72 1862 4240 3109 900 1979	4240	4
85 4857 2462 4152 2397 592 4014 2490 4770 4667	592	5
2608 1083 870 -52 1860 3969 2640 504 4282 247	3969	5
1999 829 2932 2986 -319 -806 2917	2986	3

Table 4-9 shows the results of executing the binary form validation cases by both, the Java program and the VB.NET program after transformation.

Table 4-9: Results for the binary form validation case

Number	Binary Representation
5	101
10	1010
8	1000
9	1001
3	11
32	100000
14	1110
21	10101
44	101100

Chapter Five
Conclusion and Future Work

The software description language could represent code in C++, Java and VB.NET due to the similarities between those languages and code, converters has to be built for every language to convert from and to the software description language. Differences in semantics were not included, so for a code to be convertible has to use the common semantics only.

Transforming language APIs is one of the areas that have a lot of work. It will certainly save huge effort and time in the transformation process. This thesis recommends establishing a unified API specification that includes the minimum set of functionality across language APIs such as printing to the console and the collection APIs that includes dynamic arrays, sets, maps and other collection data structures.

The validity of the proposed language has proved theoretically by conducting a semantic comparison between the three languages and experimentally by developing applications to convert source code from Java into the proposed language and from the proposed language into VB.NET. Validation cases have been designed to include various programs such as sorting, searching and also to include the most used programming constructs in the three languages. Source code of the validation cases have been converted from Java into the proposed language, and from the proposed language into VB.NET. Java and VB.NET programs of the validation cases have been executed and results compared. The results were identical for all conducted experiments.

Future work may include covering the area of multiple inheritance, which is a very powerful technique, and in fact, some problems are quite difficult to solve without this technique. Multiple inheritance can even solve some problems quite elegantly. However, multiple inheritance can significantly increase the complexity of a system, both for the programmer and the compiler writers, thus having a way to work it out may have a great effect in reducing time and effort.

Exception Handling is another improvement area that can be also further adapted, although the syntax varies between programming languages. Some languages do not call the relevant concept 'exception handling'; others may not have direct facilities for it, but can still provide means for implementing it.

The use of destructors, pass by value, and language APIs, are all examples on what more similar features can be adapted. The use of pointers is one of the challenges that still need to be addressed. It requires intensive verification and testing.

Reducing the gap between programming languages has still a lot to achieve. More languages can be included and also more work has to be done to the uncommon semantics and to adapt them in some way so they can be accessible to other languages that do not support them.

References

Ambler, S.W (1998), Building Object Applications that Work: Your Step-by-Step Handbook for Developing Robust Systems with Object Technology (1st ed), UK, Cambridge University Press

Badros, G.J (2000), JavaML: A Markup Language for Java Source Code, Computer Networks, 33(1-6), 159–177

Booch, G, Maksimchuk, R, Engel, M, Young, B, Conallen,J, Houston, K (2007), Object-Oriented Analysis and Design with Applications (3rd ed), USA, ddison-Wesley Professional

Chen H, Comparative Study of C, C++, C# and Java Programming Languages. (Master's Thesis), University of applied Sciences, Vasa : Finland

Collard, M.L (2004), Meta-Differenencing : An Intrastructure for Source Ccode Difference Analysis (Ph.D Disassertion), Kent, Kent State University

Cox, A, Clarke, C, Sim, S (1999), A Model Ondependent Source Code Repository, Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research(CASCON '99)

Dhotre, I.A., Puntambekar A.A (2008), Systems Programming (1st ed), USA, Technical Publications

Dos Reis, A.J (2011), Compiler Construction Using Java, JavaCC, and Yacc (1st ed), USA, Wiley-IEEE Computer Society

Dowek, G (2009), Principles of Programming Languages (1st ed), USA, Springer

Elmasri R , Li Q, Fu J,Wu Y,Hojabri B,Ande S Conceptual modeling for customized XMLschemas. DKE. 54: 57-76 (2005)

Evjen B, Ed, Sharkey K, Thangarathinam T, Kay M, Vernet A, and Ferguson S. Professional XML, Wrox USA, 2007,Part 3

Fishcer, A, Grodzinsk, F (1992), The Anatomy of Programming Languages (1st ed), USA, Prentice Hall

Gosling, J, Steele , G, Joy , B, Bracha ,G (2005), The Java Language Specification(4th ed), USA, Prentice Hall

Heering, J, Hendriks, P, Klint, P, Rekers, J (1989), The syntax definition formalism SDF - Reference Manual, ACM SIGPLAN Notices, 24(11), 43 – 75

Hindley, J. R, Seldin, J. P (1986) Introduction to Combinators and Lambda Calculus, UK, Cambridge University Press

Jiří, Š (2010), XML Based Framework for Transformation of Java Source Code (Masters Thesis), Croatia , Univerzita Karlova

Kadhim.B.M., Waite,W.M (1996), Maptool - Supporting Modular Syntax Development, Proceedings of the 6th International Conference on Compiler, 268 – 280

Kontogiannis, K, Zou Y (2001), Towards A Portable XML-based Source Code Representation, In International Conference on Software Engineering (ICSE) 2001 Workshops of XML Technologies and Software Engineering (XSE), Check content according to reference

Lee K.D (2008), Programming Languages: An Active Learning Approach (`1st ed), USA, Springer

Lee, D, Chue, W (2000), Comparative analysis of six XML schema languages, ACM SIGMOD Record, 29 (3), 76 - 87

Madsen, O.L (2000), Towards a Unified Programming Language, Proceedings of the 14th European Conference on Object-Oriented Programming, 1 – 26

Mamas, E (2000), Towards Portable Source Code Representations Using XML, Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), 172

Mosses, P.D (2006), Formal Semantics of Programming Languages: — An Overview, Electronic Notes in Theoretical Computer Science , 148(1), 41-73.

O'Regan, G (2007), A Brief History of Computing (1st ed), USA, Springer

Prakash, R , Goebel, K, Wang, F (2010), Portable Executable Source Code Representations (Patent 10/813,889), Online- available at : http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/7434213.

Raiser, F (2006), SrcML: A Language-neutral Source Code Representation as a Basis for Extending Languages in Intentional Programming (Masters Thesis), Germany, Fakult¨at f¨ur Informatik

Rosen, K (2011), Discrete Mathematics and Its Applications (7th ed), USA, McGraw-Hill.

Salus, P.H (1999), Handbook of Programming Languages, Vol. II (1st ed), USA, Macmillan Technical Pub

Scowen R. S (1993), Extended BNF – A Generic Base Standard, Proceedings of the 1993 Software Engineering Standards Symposium (SESS'93)

Seato C.G (2007), A Programming Language Where the Syntax and Semantics Are Mutable at Runtime (Masters Thesis), UK, University of Bristol

Simic (2003), H, Prospects of Encoding Java Source Code in XML (2003), Proceedings of the 7th International Conference on Telecommunications (ConTel 2003), 573- 578 vol.2

Slonneger, K, Slonneger, K, Kurtz, B (1995), Formal Syntax and Semantics of Programming Languages (1st ed), USA, Addison Wesley Longman

Swain, G (2010), Object-Oriented Analysis and Design through Unified Modeling Language (1st ed), USA, Laxmi Publications, Ltd

Turbak, F, Gifford, D, Sheldon ,M (2008), Design Concepts in Programming Languages (1st ed), USA, The MIT Press.

Vargas, J.V (2011), Text Modification Methods for Natural Language Generation (Ph.D Disassertion), Universitat Autonoma de Barcelona, Spain

Weisfeld, M (2008), The Object-Oriented Thought Process (1st ed), USA, Addison-Wesley Professional

Winter,A, Kullbach,B, Riediger,V (2002), An Overview of the GXL Graph Exchange Language, Revised Lectures on Software Visualization, International Seminar, 324-336

Yasdi, R (1997), Logic Programming in Prolog (1st ed), UK, Psychology Press

Zhang, Y, Xu, B (2004), A survey of semantic description frameworks for programming languages, ACM SIGPLAN Notices,39 (3), 14 – 30

The Code Structure Format (Online Website), accessed on 7/October/2012, http://sds.sourceforge.net/doc/csf2-doc.html