ILJc: Porting Microsoft.NET IL (Intermediate Language) to Java

S.Siddique[1], S.D Sheriff[2], H. Wijesuriya[3], C. Wickramaratne[4], J. Makalanda[5]
[1,2,3,4,5]IT Depratment, Sri Lanka Institute of Information Technology, Sri Lanka
mscyberguest@gmail.com[1], shifani2000@yahoo.com[2], hasitha_mhg@yahoo.com[3],
wickrama7@yahoo.com[4], jmakalanda@yahoo.com[5]

Abstract – ILJc (Intermediate Language to Java compiler) would map IL (Intermediate Language) of the Microsoft.NET framework to Java based applications. ILJc would be the middle-base in compiling and translating the IL code which ports out of Microsoft Visual Studio.NET into the equivalent Java byte code. Thus, ILJc would simply transform Visual Studio.NET into relative Java applications. ILJc does its best to map Microsoft.NET entities to Java equivalents, furthermore ILJc must deal with the exceptions too. For an instance, in the case of exceptions, Java provides one exception class, but Microsoft.NET provides three; ILJc is forced to map all three Microsoft.NET exceptions to the single Java exception. The task is no easy to fix, but simply have to write around this many-to-one mapping. Although all Microsoft.NET exceptions are derived from the System.Exception class, not all Java exceptions are derived from Java.lang.Exceptions; some generated by the JVM (Java Virtual Machine) are derived from the Java.lang.Error class. This incongruence shows up in Microsoft.NET code that tries to catch all exceptions, when translated into Java, exceptions derived from Java.lang.Error is taken for granted and slip off. As a proof of concept of how that system could be, we started sometime ago to implement ILJc, featuring a modular, flexible architecture. In this paper we show its structure, how it can be used, and how it can be further extended.

I INTRODUCTION

The ILJc is a compiler that compiles the Microsoft.NET Intermediate Language Code into the relative Java byte code. Microsoft Visual Studio.NET has a very rich, fast and powerful developing environment. Visual Studio.NET also provides an environment, where programming in multiple languages is possible. Thus the Visual Studio.NET provides language neutral environment where programs or modules can be written in many languages and integrated to one system. The former mentioned language neutrality is made possible by the Microsoft.NET intermediate language. In spite of all the above mentioned features there is a major hiccup in Microsoft.NET, it is not platform independent.

Java, it is platform independent, thus programs written in Java could be run on many other different platforms. But programs written in Java exclusively binds to the Java Language, where it is not language independent.

ILJc compiles the intermediate language code into Java byte code. In doing so, it provides a whole new approach in creating platform independency as well as language independency. And all in all rich and power full development could be done through Microsoft.NET and the porting on to the Java side could be done by ILJc.

II THE ILJC SYSTEM

ILJc in design and development will deliver the productivity advantages of the Visual Studio development system and provides native deployment of applications on J2EE environment. It is proposed to follow the functionality of basic compiler and review the necessary features to implement in meeting the goal of mapping of the IL to the Java byte code. ILJc would plug in between IL (Intermediate Language) and the Java byte code.

The core of the project can be viewed as in the Figures. The plug-in of ILJc is visualized in Figure 1. The system of ILJc compiles with sequence of functions integrating each other to produce the final out come of Java class file. The sequence is explained in Figure 2. The source version will be the out put of the compiled Microsoft.Net Program as Intermediate file (IL). It will be directed to the lexical analyzer where it will be broken into set of tokens. The tokens will be evaluated at parser where it is checked for errors. Then the error free tokens will be mapped one by one in high level implementation. The system eventually would allow the user to transform the Microsoft.Net program to run on the Java environment. This sequence of flow will be basically implemented using three sub modules as integrating at the end. This is visualized in Figure 3.

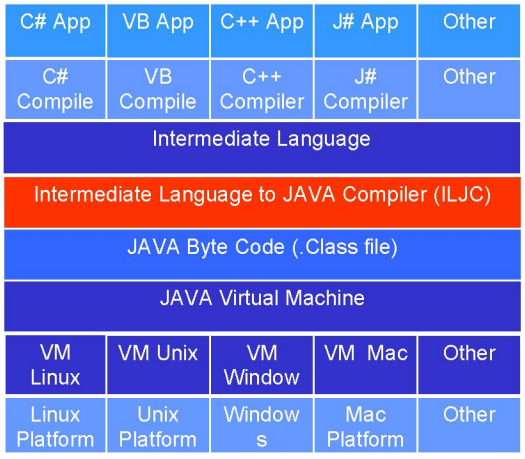

Figure 1 ILJc Plug In

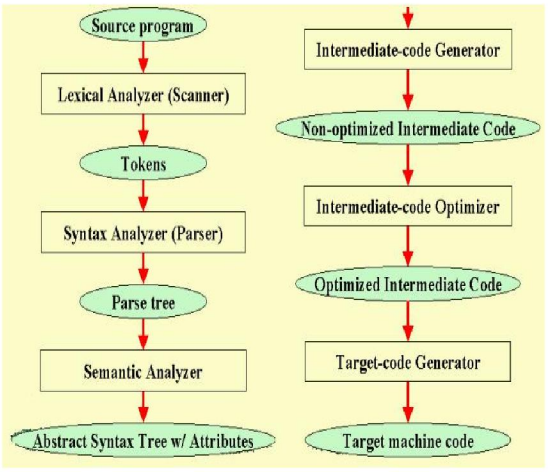

Figure 2 ILJc Sequence Flow

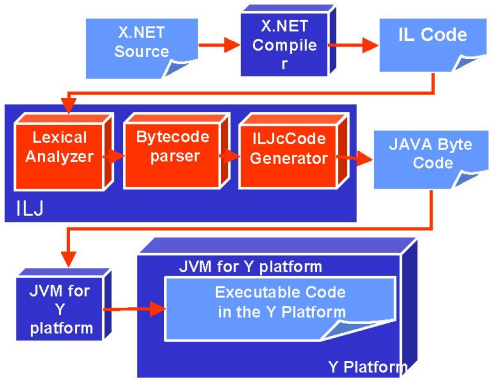

Figure 3 ILJc Sub Modules

A *Lexical Analyzer: Subsystem*

A Lexical Analyzer is a system that takes string file(s) as input, and outputs substrings which it recognizes. Each substring is referred to as a 'token' and has a numeric identifier assigned to it to uniquely identify its type (string, int, keyword etc). The purpose of Lexical Analyzer is to take a stream of input characters and decode them into higher level tokens that a parser can understand.

The Functionality;

- Take the source file as string.
- Break the strings into tokens.
- Check for the string pattern like for example: a23, DIT etc.
- Out put the values of the string tokens

B *Byte Code Parser: Subsystem*

The process of determining whether or not a series of tokens satisfies the expressed syntactic rules of a language is known as *syntactic analysis* or *parsing*. Parsers consume the output of the lexical analyzer and operate by analyzing the sequence of tokens returned. The parser matches these sequences to an end state, which may be one of possibly many end states. The end states define the *goals* of the parser. When an end state is reached, the program using the parser does some action, either setting up data structures or executing some action-specific code. Additionally, parsers can detect from the sequence of tokens that have been processed when no legal end state can be reached; at that point the parser identifies the current state as an error state. It is up to the application to decide what action to take when the parser identifies either an end state or an error state.

The Functionality;

- The parser requests tokens from the tokenizer (sometimes called the *lexer*) and attempts to determine if the tokens occur in sequences that are permitted by the general form of the language in question.

This general description of the syntactical form of a language is called a *grammar*.

C *Code Generator: Subsystem*

The code generator is responsible in taking the error free tokens passed by the parser and mapping the tokens of IL to relevant Java byte code.

The Functionality;

- Mapping the basic language fundamentals.
- Mapping the class libraries.
- Handling and maintaining Library links.

III CONCLUSIONS

Currently there is no system which takes Microsoft.NET IL file as input to produce the relevant Java byte code which runs on any environment. This is a major draw back of Microsoft.NET that it cannot run on multiple environments.

It was very necessary to scrutinize into the total background of the two programming languages, as well as to select an optimized solution which holds the objective of the project. Considering the fact it is yet not being examine, the project start on the researches background to develop an approach to the solution. The conversion of IL to Java was not available so far as to plug and convert the source of Microsoft.NET to the Java Class files.

ILJc brings Microsoft.NET programming languages to the Java platform that would enable many Microsoft® skilled developers to quickly and efficiently develop applications to work on multiple platforms thus it will not cost additional packages at all.

Windows with ILJc platform solution would enable Microsoft.NET developers to write custom J2EE applications, reducing their application development costs, time-to-benefit, and to be stable on their code to run on multiple platforms.

The console mode of converting the Microsft.Net to java has been successfully completed by the team of ILJc. Now the windows application conversion module is under testing phase. Since this a very complex and heavy research project it would not be possible to come up with the total mapping module from Microsoft.Net to Java. Hence it is well come to all the programmers and interest parties to undergo with the modules which have been

already created and to use the evolutionary prototype methodology to make it increase with the functional module.

It is very much complex when it comes to map modules from Microsoft.Net to Java ones, hence it would require to write java modules on own to run the functions on the Java platform.

IV Acknowledgment

We would like to thank all those helped us in striving to workout with this paper.

V References

[1] G. Eason, B. Noble, I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," *Phil. Trans. Roy. Soc. London*, Vol. A247, pp. 529-551, April 1955

[2] J. Clerk Maxwell, *A Treatise on Electricity and Magnetism*, 3rd ed., Vol. 2, Oxford: Clarendon, 1892, pp. 68-73

[3] I. S. Jacobs, C. P. Bean, "Fine particles, thin films and exchange anisotropy," in *Magnetism*, Vol. III, G. T. Rado, H. Suhl, Eds. New York: Academic, 1963, pp. 271-350

[4] K. Elissa, "Title of paper if known," unpublished

[5] R. Nicole, "Title of paper with only first word capitalized", *J. Name Stand. Abbrev.*, in press

[6] Y. Yorozu, M. Hirano, K. Oka, Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," *IEEE Transl. J. Magn. Japan*, Vol. 2, pp. 740-741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982]

[7] M. Young, *The Technical Writer's Handbook*. Mill Valley, CA: University Science, 1989