A CLR Back-end for a FLOSS Eiffel

Final Year Project Final Report

Daragh Hurley

A thesis submitted in part fulfilment of the degree of BSc. (Hons.) in Computer Science with the supervision of Dr. Joseph Kiniry and moderated by Dr. Michael Walsh.

School of Computer Science and Informatics

University College Dublin

06 April 2006

Abstract

Eiffel is a widely respected object-oriented programming language [1]. The Microsoft .NET Framework is a popular new programming environment available on the Windows and Linux platform. Visual Eiffel is an Open Source Eiffel compiler. The Visual Eiffel compiler compiles Eiffel source code to C code and produces executables from the C code. The compiler does not have a complete and working .NET backend and runtime support. The goal of this project is to provide a Visual Eiffel compiler with a .NET backend and runtime support [2].

Table of Contents

Acknowledgements

Firstly, thanks must be given to the project supervisor, Dr. Joseph Kiniry for the huge amount of help and advice he gave throughout the whole project. Thanks also to Dr. Michael Walsh, who was the project moderator. Thanks to Object Tools Ltd., especially Mr. Eugene Melekhov, Mr. Andrew Tischenko and Mr. Frieder Monninger for their help and advice on Visual Eiffel. Mr. Radu Grigore, a research student of Dr. Kiniry in University College Dublin and a member of the Systems Resource Group in University College Dublin, helped with the bootstrapping of the Visual Eiffel compiler. The Systems Research Group, in the School of Computer Science and Informatics in UCD, as a whole was very helpful when it came to the presentation. They organised mock presentations for the final project presentation.

1 Introduction

1.1 Project Specification

Supervisor	Dr. Joseph Kiniry
Subject Area	Programming Languages
Pre-requisite	Good understanding of object-oriented programming, basic knowledge of compilers
Co-requisite	Eiffel
Subject Coverage	Compilers
Project Type	Design & Implementation
Software	Visual Eiffel
Hardware	A machine running anything but Windows

Description

A commercial product from Eiffel Software (http://eiffel.com/) called EiffelEnvision (http://www.eiffel.com/products/envsn/) is a plug-in for Microsoft's VisualStudio .NET development environment. This tool supports the compilation of Eiffel code to the .NET framework by compiling to byte code for the Common Language Runtime (CLR).

No existing Free License Open Source Software Eiffel compiler can currently compile to CLR byte codes. The Visual Eiffel compiler, which is now available under a FLOSS license, has a partially complete CLR backend.

The purpose of this project is to implement a proposed language extension in the SmartEiffel or Visual Eiffel compilers so as to evaluate their design, effectiveness, and utility

Mandatory

1. Review the Visual Eiffel compiler source, documenting its overall structure, particularly with regards to extension, using a high-level modelling language.

2. Review the existing CLR backend in Visual Eiffel and document its requirements and capabilities.

3. Finish the implementation of the CLR backend for Visual Eiffel according to the proposed requirements from step 2.

1.2 Initial Background to Project

Eiffel is an object-oriented language originally developed by Interactive Software Engineering (ISE) Inc., now called Eiffel Software, in 1985 [1]. Visual Eiffel, a compiler implementation of Eiffel, is not yet fully compatible with the .NET Framework. The Visual Eiffel source code contains code that can convert Eiffel source code to Common Intermediate Language (CIL) byte code, but it does not create an executable. This project aims to integrate the Visual Eiffel compiler with the .NET Framework Common Intermediate Language. The CIL is the .NET-specific byte code, that is comparable to the Java byte code, which is executed by the .NET Framework's runtime environment, known as the Common Language Runtime (CLR). The CLR executes the CIL byte code much like the Java Virtual Machine executes Java byte code. Thus, the final result of this project is a new version of the Visual Eiffel compiler that fully supports compilation to CIL.

The first step in the project was to carry out background research on Eiffel and the .NET framework.

2 Background Research

2.1 Eiffel as a Programming Language

Bertrand Meyer and Interactive Software Engineering (ISE) Inc., now called Eiffel Software, developed Eiffel. It is an object-oriented programming language, whose design is based on the core principles of reusability, extendibility and reliability [3].

- The reusability of Eiffel is realised in the code and specification by the use of components, e.g. the business logic in the architecture that is reusable in a broad range of situations with minimal need to modify the code-base of the component.

- In terms of extendibility, new features and components can be added into a system in a variety of ways. For example, multiple class inheritance is available. Unlike single class inheritance in Java and C++, multiple class inheritance provides the capability of combining the methods and fields of many classes in non-trivial, but safe, ways.

- The reliability of Eiffel is reinforced by Design by Contract. Design by Contract is a principle whereby every program written in Eiffel must conform to a specification. Each program is tested to conform to the contract on which it was based. This reduces the occurrences of bugs in software and, when a bug occurs, it is easily identifiable and fixable.

An Eiffel compiler is designed to cooperate with other languages, such as C and assembly language. Also, many programs developed in Eiffel are portable. Three of the four Eiffel compilers compile to C source code, which is then passed to a C compiler [4]. This provides portability.

The Eiffel community is split into four compiler implementations:

1. SmartEiffel [5] is a new implementation of an Eiffel compiler with the goal of creating a fast, slim and multi-platform GNU Eiffel compiler.

2. Eiffel Envision [1] has been created to help provide .NET Framework support primarily within Windows. Envision is a commercial front-end expansion of Microsoft Visual Studio, a development environment for the .NET Framework.

3. Visual Eiffel [6] is the compiler being used in the development of this CLR back-end. Object Tools Ltd. develops it. Source code is portable across many platforms. The project was carried out in collaboration with Object Tools Ltd.

4. EiffelStudio [1] is an Integrated Development Environment (IDE) available on many platforms. It provides an Eiffel compiler and allows for the development of executable files from Eiffel source code. It cannot use other Eiffel compilers. It can generate class diagrams from Eiffel source code.

2.2 The Visual Eiffel Development Environment

Object Tool's Visual Eiffel IDE was used for development in Windows. In Linux, Eiffel Software's EiffelStudio was used. These were used as the main development environments for this project.

2.3 Overview of the .NET Framework

Microsoft developed the .NET Framework in the 1990s. The .NET Framework is a software development platform that provides support for development of applications in many computer languages at the same time. All program code is compiled down to one common byte code, known as the Common Intermediate Language (CIL). The Common Language Infrastructure (CLI) executes this byte code. Microsoft's implementation of the CLI is the Common Language Runtime (CLR). The .NET Framework is seen as an alternative and competitor to Sun Microsystems Java 2 Enterprise Edition.

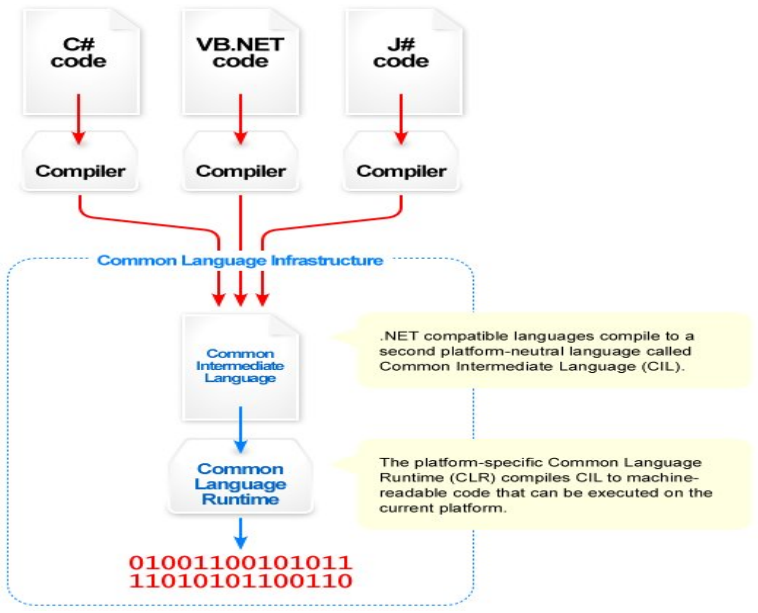
C# code
VB.NET code
J# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

Figure 1. How source code is compiled and run in the .NET Framework. Wikipedia, 2006: Wikipedia page on the Common Language Infrastructure <http://en.wikipedia.org/wiki/Image:Overview_of_the_Common_Language_Infrastructure.png>

The .NET Framework provides support for many programming language libraries, whether they are classic languages, such as Pascal and FORTRAN, business-oriented languages like COBOL, or modern object-oriented languages, such as C++ and C#, among others [7]. At runtime, CIL byte code is compiled down to platform-specific assembly code.

2.4 Overview of the Common Language Infrastructure

In order to learn about the CLI in detail, the European Computer Manufacturers Association (ECMA) standard 335 on the CLI [8] was obtained. The standard was first skimmed through briefly and then Partition III was read in detail. According to this standard, the Common Language Infrastructure is a means "in which applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments" [8].

The core of the CLI is the Common Type System (CTS). Compilers, tools and the CLI share the CTS. It defines the rules by which data types are defined and enables the integration of code written in different languages. Metadata that is independent of any programming language is used to define and access the types in the CTS. According to the ECMA standard, the metadata stores information to:

- manage the execution of the code, including memory management and monitoring of the execution state,
- administer the code such as install it, and
- reference the types in the code, so that, for example, the code is importable into other languages and tools.

Using the metadata, the CIL byte code is executable on many execution models.

A subset of the CTS called the Common Language Specification (CLS) describes how the rules defined in the CTS are used. CIL byte code is loaded and executed by the Virtual Execution System.

CIL byte code is type-safe. In this context, type-safe CIL byte code has the following properties:

- All references in the byte code must have a type and everything referenced must also be typed.
- Objects created can only be manipulated by the ways set out by the CTS.
- The only operations that can be carried out are those made allowable by the byte code. For example, a private method cannot be accessed outside the class it is defined in.

Like Java, the CLI supports garbage collection. This means that the allocation and de-allocation of memory is managed.

2.5 Overview of the Common Intermediate Language

When compiled, the source code of the .NET-compatible programming language is converted into CIL byte code. This byte code is independent of the CPU and platform. The independence enables the code to be run on any .NET framework-supporting platform. CIL is an object-oriented assembly language that uses a stack. The Microsoft implementation of the CIL is known as the Microsoft Intermediate Language.

During execution, the CIL byte code is often converted to code that is immediately executable by the CPU. This conversion occurs incrementally during the execution of the program. This process is known as Just-In-Time (JIT) compilation. Another option is for the code to be converted to a native binary image, known as Native Image Generator (NIG) compilation. This is executable on the current environment only. NIG compilation does not require JIT compilation. In the .NET framework, a binary image that the user is trying to run that is not native is identifiable. When this occurs, the runtime environment reverts to JIT compilation. Metadata is read, during program execution, to identify the interfaces that are used in the application and provide support for them [9].

After the background research on the .NET framework, tutorials were carried out to learn about development in the .NET framework.

3 How the Project was Approached

The Mono Project was used [10] as the development environment for the .NET framework in Linux. Eiffel is not supported by the Mono Project, so C# was used as an alternative to write .NET programs. Using C#, it is possible to generate CIL byte code to review to learn about it’s syntax and overall structure.

3.1 C# and the Common Language Infrastructure

C# was developed by Microsoft. It has full .NET support. It is possible to generate CIL byte code from C# source code that is executable in the .NET framework. C# is similar in syntax to Java but compiles down to Windows executables (files with .exe or .com extensions), or dynamic linked libraries (.dll files). Code that is used by many applications, often at the same time, should be put in a dynamic linked library. This saves space and disk memory, as only one copy of the code is needed to be stored.

Using the Mono Project compiler, `mcs` it is possible to generate and run CIL executables in Linux. Code is compiled using the `mcs` command and run using the `mono` command. It is also possible to disassemble these executables to CIL byte code using the `monodis` command. Using these commands, a ‘Hello World’ program was written in C# and then compiled to an executable. It was then disassembled. Sections 4.1.1 and 4.1.2 provide a walkthrough of the C# source code and the CIL byte code that `monodis` generated from the executable.

3.1.1 Walkthrough of HelloWorld with a Method Call in C#

```
/* HelloWithMethodCall.cs - C# implementation of
 * HelloWorld compiled using Mono C# compiler
 *
 * Author: Daragh Hurley
 */
using System;
namespace HelloCS {
  public class Hello {
    public static void Main() {
      String s = "Hello C# World :-)";
11.       printString(s);
12.     }
13.     //Method that prints a String
14.     static void printString(String str) {
15.       Console.WriteLine(str);
16.     }
17.   }
18. }
```

- Line 6: The use of the System library is needed. It provides classes required by the program for I/O.

- Line 7: A namespace is created to store identifiers. This is used in the creation of dynamic linked libraries (dlls). The code in the namespace is reusable by other applications.

- Line 10 and 11: A string is created and passed into the `printString` method.

- Line 14 to 16: The `printString` takes in a string as a parameter and prints it to the console.

C# code is similar to Java. A namespace is equivalent to a Java interface. It enables the reuse of code.

3.1.2 Walkthrough of Disassembled CIL Byte code

```
1.  /* DAHelloWorld – HelloWorldWithMethodCall.cs disassembled
2.   * to CIL byte code
3.   */
4.  .assembly extern mscorlib {
5.    .ver 1:0:5000:0
6.    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) // .z\V.4..
7.    }
8.  .assembly 'HelloWithMethodCall' {
9.    .hash algorithm 0x00008004
10.   .ver 0:0:0:0
11.   }
12. .module HelloWithMethodCall.exe
13.   /* GUID = {A5E0018E-3F60-48C0-971A- 149659885F01} */
14. .namespace HelloCS {
15.   .class public auto ansi beforefieldinit
16.    Hello extends [mscorlib]System.Object {
17.     // method line 1
18.     .method public hidebysig specialname rtspecialname
19.      instance default void .ctor () cil managed
20.     {
21.       // Method begins at RVA 0x20ec
22.       // Code size 7 (0x7)
23.       .maxstack 8
24.       IL_0000: ldarg.0
25.       IL_0001: call instance void object::.ctor()
26.       IL_0006: ret
27.     } /* end of method Hello::instance default void .ctor () */
28.     // method line 2
29.     .method public static hidebysig
30.      default void Main () cil managed {
31.       // Method begins at RVA 0x20f4
32.       .entrypoint
33.       // Code size 13 (0xd)
34.      .maxstack 2
35.      .locals init (
36.       string V_0)
37.      IL_0000: ldstr "Hello C# World :-)"
38.      IL_0005: stloc.0
39.      IL_0006: ldloc.0
40.      IL_0007: call void class HelloCS.Hello::printString(string)
41.      IL_000c: ret
42.    } // end of method Hello::default void Main ()
43.    // method line 3
44.    .method private static hidebysig
45.     default void printString (string str) cil managed {
46.      // Method begins at RVA 0x2110
47.      // Code size 7 (0x7)
48.      .maxstack 8
49.      IL_0000: ldarg.0
50.      IL_0001: call void class
51.       [mscorlib]System.Console::WriteLine(string)
52.      IL_0006: ret
53.    } // end of method Hello::default void printString (string str)
54.  } // end of class HelloCS.Hello
55.} // END DAHelloWorld
```

- Line 4: Uses the external library `mscorlib`. This is a code-base that is an implementation of the libraries of all the .NET programming languages. The code in `mscorlib` is similar to the Java Advanced Programming Interface.

- Line 4 to 11: Security information for this code.

- Line 14: Namespace created

- Line 15: Identify the coding standard used in compilation.

- Line 16: Extends `System.Object` class in `mscorlib`.

- Line 23: Maximum size of stack is set. The stack is the program code execution model the CIL byte code uses.

- Line 30: Start of main method. This code is managed. This means that the metadata describing the method can be located, the stack can be walked, exceptions can be handled and the code has security information [8].

- Line 34: Maximum size of stack is set.

- Line 37: Load a string onto the stack.

- Line 40: Call C# specific code that prints out the string in it's C# representation.

- Line 44/45: Start of `printString` method.

- Line 50/51: The method `WriteLine`, from the `mscorlib` library, is called with string passed in as a parameter.

3.2 The Visual Eiffel Compiler

The Visual Eiffel compiler produces executables using source code written in Visual Eiffel itself. The executable produced by compilation is a Dynamic Linked Library, .com, or .exe file. In Linux, it is a binary executable. During compilation, the source code is first compiled to C and then converted to an executable format. All executables are either produced from a library or a project. Projects and libraries can be grouped into clusters. When first compiling, the compiler will mount a cluster and the projects or libraries will be compiled. Before re-compilation, the cluster can be re-mounted.

The Visual Eiffel compiler is an efficient compiler due to the compilation to C. This C code is similar in length to the Java equivalent. The executable consumes a similar amount of resources as a Java executable also.

Like all compiler implementations of Eiffel, it is possible to call C functions directly in Visual Eiffel source code. However, an additional tool is needed to compile code containing these calls.

3.3 Visual Eiffel and the Common Language Infrastructure

Prior to this project Visual Eiffel's support for the .NET Framework was under development. Code was written to convert Eiffel source code to CIL byte code. For example, this code can convert an Eiffel keyword to the CIL equivalent. The requirement to make this code work is to combine it with the higher-level code that builds an executable from it. Once complete, there will be a platform emitter. This platform emitter will generate, or emit, CIL byte code given Eiffel source code. It will then be possible to use the compiler to generate an executable that will run on a platform compatible with the .NET Framework.

The first step taken was to download the Visual Eiffel source code and bootstrap it under Linux. Bootstrapping is the process of taking the Visual Eiffel libraries and building an executable from them. The resulting executable is the compiler itself. It is a complete and executable version of the compiler, based on all the Visual Eiffel source code. This required the installation of the GNU C compiler, version 2.9x. In collaboration with Mr. Radu Grigore, a Ph D. student of Dr. Kiniry, the GNU C compiler was installed on his machine and an account was set up to access it remotely. A set of tools and libraries needed to bootstrap Visual Eiffel, known as Gobo Eiffel source, was downloaded. The Gobo Eiffel sources needed to be bootstrapped prior to bootstrapping Visual Eiffel. The first attempt to bootstrap Gobo Eiffel failed. It was found that the latest version available from sources was not a stable release. Thus, version 3.4 sources were downloaded and bootstrapped instead. Once the bootstrapping of the Gobo Eiffel sources was done, Visual Eiffel was bootstrapped. The bootstrapping of Visual Eiffel source code produces an executable file that provided a full compiler implementation of the source code.

Once the bootstrapping was completed, the preparation work to design of the back-end was started. The preparation work involved reviewing the Visual Eiffel source code needed for the CLR back-end and comprehending the code.

4 Preparation Work for Designing the Back-end

At the start of the project, tutorials on EiffelStudio and Visual Eiffel were done. The tutorials provided preparation for writing the compiler's source code. EiffelStudio IDE was used as the development environment for Eiffel in Linux and Visual Eiffel IDE was used in Windows.

The first step in preparation for designing the back-end was to identify the necessary files that the platform emitter used to generate CIL byte code. These files are `platform_manager.ge`, `platform_cil.e` and `platform_emitter.e`.

- `platform_manager.ge` sets up the platform emitter and calls the methods for a specific platform that are defined in that platforms class. The class for the x86 platform specific code is `platform_x86.e`. The class for the CIL platform is `platform_cil.e`.

- `platform_emitter.e` is a deferred class that names the methods needed to emit code for a particular platform. The class calls the methods used by the particular platform when building an executable for it.

- `platform_cil.e` is the class that extends `platform_emitter.e`. This file partially defines what the methods named in `platform_emitter.e` do and contains additional methods needed for the CIL platform. It provides a partial implementation of the CLR back-end.

`platform_manager.ge` is a source file that has not been passed through the Gobo Eiffel pre-processor (`gepp`). `gepp` works similar to the C pre-processor. It generates source code according to the information provided in the Eiffel source. Passing `platform_manager.ge` through `gepp` will provide an implementation of the compiler code for a particular platform. The platform that the Visual Eiffel compiler was bootstrapped for was the x86 architecture. `platform_manager.ge` provides an option that `gepp` uses to build the compiler for the .NET framework. It is not possible to build the compiler for the .NET framework without finishing the implementation of the source code in `platform_cil.e`.

When `platform_cil.e` is complete, it will provide a full implementation of these methods. A command line switch will be passed into the Visual Eiffel compiler to build a .NET executable. The platform emitter will then build the CIL byte code from the Visual Eiffel source code. The next section describes the platform emitter in detail and explains how `platform_cil.e` works.

4.1 The CIL Platform Emitter Source Code

```
deferred class PLATFORM_EMITTER

feature -- Status report

  is_null: BOOLEAN is
      -- Is this emitter null?
      -- Null emitter does not do anything and
      -- calling any feature on it can be safely skipped.
    deferred
    end

feature -- Access

  start_immediate is
```

```
15.       -- Start emission of the data and code for immediate
16.       -- features of the class
17.     deferred
18.     end
19.
20.   stop_immediate is
21.       -- Stop emission for the immediate features
22.     deferred
23.     end
24.
25.   start_inherited (cls_dsc: CLASS_DESCRIPTOR) is
26.       -- Start emission of the features inherited
27.       -- from the class `cls_dsc'
28.     require
29.       cls_dsc /= Void
30.     deferred
31.     end
32.
33.   stop_inherited is
34.       -- Stop emission of the inherited features
35.     deferred
36.     end
37.
38.   emit_routine (
39.     descriptor: FEATURE_DESCRIPTOR
40.     implementation: FEATURE_IMPLEMENTATION
41.     as_old_expressions: ARRAY [OLD_EXPRESSION]
42.   ) is
43.       -- Emit routine
44.     require
45.       descriptor /= Void
46.       implementation /= Void
47.       as_old_expressions /= Void
48.     deferred
49.     end
50.
51.   emit_attribute (descriptor: FEATURE_DESCRIPTOR) is
52.       -- Emit attribute
53.     require
54.       descriptor /= Void
55.     deferred
56.     end
57.
58.   emit_class_invariant (assertions: ARRAY [ASSERTION_CLAUSE]) is
59.       -- Emit class invariant
60.     require
61.       assertions /= Void
62.     deferred
63.     end
64.
65.   close is
66.       -- Stop emission for the class and
67.       -- release all associated resources
68.     deferred
69.     end
70.
71. end
```

- Line 5-10: This method returns a boolean stated whether the current emitter is null. The null emitter is acceptable and any methods called on it are skipped.

- Line 14-18: `start_immediate` will initiate the generation of CIL byte code. It will do this by emitting the immediate features of the class such as its name.

- Line 20-23: `stop_immediate` terminates the action of `start_immediate` and generates stubs for the inherited routines. The stubs will act as placeholders for CIL byte code.

- Line 25-31: `start_inherited` will take in a class descriptor and emit code for the features that the class inherits.

- Line 33-36: Terminates the action of `stop_inherited`. The class code is then emitted and saved.

- Line 38-49: `emit_routine` takes a routine, reads in all it's features and emits the routine as CIL byte code.

- Line 51-56: `emit_attribute` processes converted functions.

- Line 58-63: `emit_class_invariant` converts the class invariants to Register Transfer Language (RTL). The GNU C compiler generates RTL during compilation. It is lower level than C and is processor-specific. It then prints the converted class invariant to an assembly file.

- Line 65-69: `close` terminates the emission, releases resources and completes the emission of CIL byte code from the features emitted.

Once the CIL platform emitter code was reviewed, the design of the back-end began.

5 Implementation of the Back-end

In order to start designing the CLR back-end, the Visual Eiffel compiler is built to support the compilation of CIL byte-code. The compiler is built using Ant. Ant is similar to a makefile. It defines exactly what is to be done when building a project. Unlike makefiles, it is written entirely in XML. A parameter is passed in specifying the compiler to be built by Ant with the .NET back-end support. The platform manager contains code used by the Gobo pre-processor to build the compiler for the CLR back-end. In this process, the Gobo Eiffel pre-processor generates a platform manager for the back-end.

During the build process, many errors were found. The process would terminate at the point the error occurred. These errors often occurred when a .ge file, which is not supported by the Visual Eiffel compiler, was found when the compiler required an .e file for compilation. The Gobo Eiffel pre-processor could be run on these files to generate the .e file. Once this was done, the build process was restarted and the error did not occur again. This pattern of trial and error continued until the following error occurred while the compiler was running:

```
File : /home/dmhurley/gobo/library/kernel/Unicode/uc_string.e
   Class : UC_STRING
Error VEVAL325: 906:3 (VUPR-3) Precursor in routine declaration which is
not a redefinition
```

This means that on line 906 of `uc_string.e`, a call was made to the parent class that was not redefined in a routine declaration.

The progress of the project as a whole is summarised in Chapter 7.

6 Progress Made

September:

4. A GForge account was set up called eiffelclrcomp [11]. This was used for the management of project tasks and listing of project sources.

5. European Computer Manufacturers Association standard on CLI was found, downloaded and reviewed.

6. The Mono Project, needed for .NET Framework development in Linux, was installed.

October:

7. Started reading Partition III of ECMA CLI standard in detail.

8. A C# Hello World program was compiled using the Mono C# compiler. The resulting executable was disassembled into byte code and studied.

9. Started using EiffelStudio IDE. A Hello World program was written and compiled in Eiffel. Examined source code Unified Modelling Language and Business Object Notation diagrams using the development environment.

10. The complete Visual Eiffel source code was downloaded. The examination of the CLI platform emitter files was started.

11. The first attempt to bootstrap Visual Eiffel source code was started using Visual Eiffel compiler and gcc-4.0.1.

November:

12. We collaborated with Radu Grigore on bootstrapping. The Gobo Eiffel source code was downloaded and bootstrapped using the Visual Eiffel compiler.

13. The Interim Report was started and submitted.

December:

14. The bootstrapping of the Visual Eiffel source code was completed.

January:

15. Figured out what classes and methods in the platform emitter code could / could not be run at runtime.

16. The Eiffel and C# tutorials were continued.

February:

17. Started building Visual Eiffel compiler that could produce .NET Framework source code.

18. The remaining code in the platform emitter classes to be completed was determined.

March:

19. The final report was started and submitted.

At the end of the project, a review of all progress was done and the conclusion was determined. This also involved determining what future work could be done related to the project.

7 Conclusion and Future Work

The aim of this project was to design an initial release of a tool. Progress has been made towards this goal, but it has not been fully achieved. The main reason while a working CLR back-end has not been produced is that we failed to build the Visual Eiffel compiler for the .NET framework, using Ant. The problem encountered is described in Chapter 5. Therefore, the compiler could not be built to produce a platform manager for the .NET framework. The immediate aim of future work is to resolve this problem. The problem occurred in a file that is required by Gobo Eiffel. The solution is to debug the error and rectify the problem. This involves identifying the parent class that is causing the problem to understand the error.

The first two mandatory steps have been completed. These were to document the Visual Eiffel compiler source and to review the existing CLR backend for Visual Eiffel. Step 3 of the project was to finish the implementation of the CLR backend for Visual Eiffel. This has been started, but as described above, was not completed. Therefore the project was not a complete success, as all the mandatory goals were not achieved.

In future, the documented changes to the code in `platform_cil.e` would have to be completed. Once this is done, the tool would have to be performance tested:

- This should first involve functional testing which will resolve any remaining bugs. This will involve rigorously running the compiler in different scenarios to identify any possible bugs.

- Load testing to analyse the execution of the compiler and to tune it will follow this.

- Following this, the compiler will be stress tested. This involves putting increasing the load of the compiler to find out the point at which it consumes all the resources of the system it is being tested on or breaks.

In order to complete this project, Eiffel was learnt from scratch. The .NET framework was also new to the student. These facts meant that background research was a key part of the project. Much background research was successfully carried out. However, more time should have been devoted to designing the back-end.

Glossary

- Ant: A tool that defines exactly what is to be done when building a project from source. It is similar to makefiles, but is written in XML.
- Business Object Notation: An alternative object modelling language and specification.
- Common Intermediate Language: The object/assembly code that is executed by the CLR. It is compiled from source code, often in many programming languages.
- Common Language Infrastructure: The CLI is the runtime environment that executes the CIL byte code compiled by the .NET framework. The CLR is Microsoft's implementation.
- Common Language Runtime: Microsoft's implementation of the .NET framework's runtime environment.
- Common Language Specification: This describes how the rules in the CTS are used.
- Common Type System: This defines the rules by which data types are defined and enables the integration of code written in different languages.
- Design by Contract: A principle whereby every program written in Eiffel must conform to a specification.
- Dynamic Linked Library: The reusable program source code written .NET framework.
- `gepp`: The Gobo Eiffel pre-processor. It generates Eiffel source code according to the pre-processing information provided in the code. It works similarly to the C pre-processor.
- Just-In-Time compilation: A process by which CIL byte code is converted to code that is immediately executable by the CPU. The conversion occurs incrementally during program execution.
- `mcs`: The Mono Project compiler. Compiles source code to CIL byte code and then to .NET executable code.
- `mono`: The Mono Project command used to run CIL executables.
- `monodis`: The Mono Project disassembler. This converts .NET executables to CIL byte code.
- `mscorlib`: A code-base that is an implementation of the libraries of all the .NET programming languages.
- Native Image Generator compilation: A process by which CIL byte code is converted to a native binary image.

- Register Transfer Language: The GNU C compiler generates RTL during compilation. It is lower level than C and is processor-specific. It defines individual processor actions.

- Unified Modelling Language: An object modelling language and specification designed for use in Software Engineering projects.

- Virtual Execution System: This loads and executes the CIL byte code.

References

[1] Eiffel Software, 1985-2006: Eiffel Software. <http://www.eiffel.com>.

[2] KindSoftware, 1993-2005: Project Specification. <http://secure.ucd.ie/documents/proposals/eiffel_clr_backend.html>.

[3] Meyer, Bertrand, 2001: *Eiffel The Language*. Prentice Hall Europe, Hertfordshire.

[4] Wikipedia, 2006: Wikipedia page on Eiffel. Internet: <http://en.wikipedia.org/wiki/Eiffel_programming_language>.

[5] The SmartEiffel Team: SmartEiffel. <http://smarteiffel.loria.fr>.

[6] Object Tools Ltd, 2006: Object Tools. <http://www.object-tools.com>.

[7] Microsoft Corporation, 2006: .NET framework community. <http://www.gotdotnet.com>

[8] ECMA, 2005: European Computer Manufacturers Association standard on the Common Language Infrastructure. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf>.

[9] Wikipedia, 2006: Wikipedia page on the Microsoft Intermediate Language. <http://en.wikipedia.org/wiki/Microsoft_Intermediate_Language>.

[10] Mono, 2006: Mono Project. <http://mono-project.com>.

[11] KindSoftware, 2006: GForge website. <http://sort.ucd.ie>.