RAIL: Code Instrumentation for .NET

Bruno Cabral
CISUC, University of Coimbra
Dep. Eng. Informática – Pólo II
3000-290 Coimbra, Portugal
+351 239790000
bcabral@dei.uc.pt

Paulo Marques
CISUC, University of Coimbra
Dep. Eng. Informática – Pólo II
3000-290 Coimbra, Portugal
+351 239790000
pmarques@dei.uc.pt

Luís Silva
CISUC, University of Coimbra
Dep. Eng. Informática – Pólo II
3000-290 Coimbra, Portugal
+351 239790000
luis@dei.uc.pt

ABSTRACT
Code instrumentation is a mechanism that allows modules of programs to be completely rewritten at runtime. With the advent of virtual machines, this type of functionality is becoming more and more interesting because it allows the introduction of new functionality after an application has been deployed, easy implementation of aspect-oriented programming, performing security verifications, dynamic software upgrading, among others.

The Runtime Assembly Instrumentation Library (RAIL) is one of the first frameworks to implement code instrumentation in the .NET platform. It specifically addresses the limitations that exist between the reflection capabilities of .NET and its code emission functionalities. RAIL gives the programmer an object-oriented vision of the code of an application, allowing assemblies, modules, classes, references and even intermediate code to be easily manipulated.

This paper addresses the design of an implementation of RAIL along with the difficulties and lessons learnt in building a framework for code instrumentation in .NET.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *Code generation, Run-time environments.*

General Terms
Languages.

Keywords
Code instrumentation, .NET platform.

1. INTRODUCTION
One key aspect of many modern programming languages is that they are compiled into a portable intermediate form and executed by a language runtime. Typically, the language runtime allows code to be loaded at runtime from a binary source (e.g. from a file or from the network) and executed.

One interesting side effect of having dynamic code loading is that before actually loading the code into a virtual machine, it is possible to instrument the code, introducing or removing specific instructions, changing the use of classes, variables and constants. The key idea is that it is possible to alter the code, performing some modifications, before the code is actually executed. These transformations are either performed after compile time, or at load time. With this approach it is possible to instrument the code so that proper resource control takes place, change the code so that it is possible to serialize and relocate executing threads in a cluster, perform program consistency checks according to security policies, redirect method calls to proxies, among others.

For making things clearer, let's consider an example. Suppose that an application has been loaded from the Internet but it can be considered trustworthy. It would be desirable to be able to know all the files that it is reading and writing from disk, so that one could be sure that it is not really stealing confidential information and sending it to the Internet. A possible approach is to try to use a disassembler and understand the code structure. But, as it is easy to comprehend, that is not really feasible except for trivial applications. Nevertheless, using code instrumentation, it is simple to replace all the references to classes that perform I/O with corresponding proxies that implement the same interface. Those proxies can log all the calls that are made before allowing the original calls to take place. This way, the user can examine the log and determine which files have been accessed.

RAIL – the Runtime Assembly Instrumentation Library – is a general purpose code instrumentation library for the .NET platform. It allows assemblies (the code load unit in .NET) to be easily manipulated in an object-oriented way. Its features include straightforwardly replacing references, types, variables, fields, properties and method calls; iterate and change Intermediate Language (IL) code; adding code epilogues and prologues to methods; copy types and methods between assemblies; among others. Also, RAIL provides several high-level patterns for assisting the programmer performing code instrumentation. Finally, RAIL allows assemblies to be instrumented at load-time, just before the types are actually defined in the Common Language Runtime (CLR) virtual machine.

2. RAIL APPROACH
Code instrumentation is not a new topic. For as long as binary code has been generated, people have been manipulating it, even in its binary form. The advent of generalized use of virtual machines with dynamic code loading, and particularly Java, fuelled an immense development in the field.

The RAIL approach consists in simplifying the programmers' task when performing code instrumentation. RAIL exposes a high level API that hides the low level details of code instrumentation and assembly's structure from the programmer. Some of the available high level functionalities are: change type references across the assembly; replace field access by other fields, methods and properties; replace method calls; replace IL instructions

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

responsible for object creation by calls to different static methods; insert prologues and epilogues in methods; copy types and methods between assemblies; manipulate Custom Attributes.

RAIL creates an Object-Oriented representation of the assembly being instrumented and allows the programmer to manipulate this structure through the use of the just mentioned high level API. Besides the OO structure, RAIL uses tables to hold the sequence of objects and object references that represent the application's IL code and all the exception handling related information. These tables can be directly manipulated and the objects that they hold modified by a secondary low level API built-in RAIL for this particular objective.

3. RELATED WORK

Abstract IL [1], developed by Microsoft Research, was the first library to allow the instrumentation of code in .NET assemblies. It does so by building an abstract syntax tree representing the assembly that can be manipulated using languages like OCaml, F# and C#. One of the shortcomings of this library is that it specifically targets the usage of functional languages (e.g. OCaml), and does not shield the programmer from the complexities of interacting and manipulating assemblies.

There are also Aspect Oriented Programming (AOP) tools for the .NET platform that use code instrumentation to apply crosscutting concerns across applications. Two of the most known examples are John Lam's Common Language Aspect Weaver (CLAW) [2] that was one of the first libraries to perform assembly manipulation by "weaving" aspects into class methods and Weave.Net [3].

In the Java community, several bytecode instrumentation libraries have been developed in recent years. BCEL [4] provides an API for manipulating Java byte code giving full control to the developer. Its major shortcoming is that it is quite low-level and hard to use. JOIE [5] provides a set of low-level primitives for manipulating Java classes and higher-level interfaces that directly support common transformer styles. Javassist [6] allows to perform both structural and behavioral reflection on *bytecode* [7][8]. Together, these libraries have been used on more than 40 projects [9] which instrument Java code to perform the most various tasks. Another good example of *bytecode* instrumentation is Binary Component Adaptation [10]. In this case, components can be adapted and evolve on-the-fly.

RAIL has been developed in parallel with AbstractIL, being one of the first code instrumentation libraries for .NET. The main difference of RAIL compared to AbstractIL is that it hides the complex details of code instrumentation from the programmer, adds an OO perspective to it and supports several common instrumentation patterns. It also distinguishes itself from the available .NET AOP tools for being more general purpose. RAIL uses many of the lessons learnt in the development of its Java libraries counterparts and adds new solutions for handling .NET specific problems and complexity.

4. CONCLUSION

The objective of RAIL is to simplify the task of code instrumentation, making possible to use it in areas such as AOP, runtime analysis tools, security verification, software fault injection and others.

Dynamic code instrumentation is currently an open issue. The possibility to modify assemblies and classes that are already loaded into the CLR virtual machine is yet to be addressed. Future work includes designing the algorithms and methods for achieving this.

Although RAIL is still very recent it already provides many features in terms of code instrumentation, such as code walkthrough, reference replacement, type coping, and addition of method prologues and epilogues. It also allows non IL experts to make complex assembly manipulations. For instance, it is possible to develop classes and methods in a separate assembly, using a high level programming language like C#, and later copy and manipulate them into the target assembly.

RAIL is publicly available at http://rail.dei.uc.pt.

5. ACKNOWLEDGMENTS

This project was partially supported by FCT, throught CISUC (R&D unit 326/97), scholarship SFRH/BD/12549/2003 and by a Microsoft Research ROTOR grant.

6. REFERENCES

[1] Syme, D. ILX: Extending the .NET Common IL for Functional Language Interoperability, MS Research, Cambridge, September 2001. Available at: http://research.microsoft.com/projects/ilx.

[2] Lam, J. Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime. *Demonstration at the AOSD2002*, Netherlands, 2002.

[3] Lafferty, D., and Cahill, V. Language-Independent Aspect-Oriented Programming. *In Proc. OOPSLA 2003*, Anaheim, USA, October 2003.

[4] Dahm, M. Byte Code Engineering. In JIT '99, Düsseldorf, Germany, September 1999.

[5] Cohen, G., Chase, J., and Kaminsky, D. Automatic Program Transformation with JOIE. In *USENIX 1998 Annual Technical Conference*, New Orleans, USA, 1998.

[6] Chiba, S., and Nishizawa M. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proc. GPCE '03*, LNCS 2830, Springer-Verlag, 2003.

[7] Ferber, J. Computational Reflection in Class Based Object-Oriented Languages. In *Proc. OOPSLA '89*, New Orleans, USA, October 1989.

[8] J. Malenfant, C. Dony, and P. Cointe. Behavioral Reflection in a Prototype-Based Language. *In Proc. IMSA '92*, Tokyo, 1992.

[9] BCEL Project, Apache Software Foundation, 2003. Available at: http://jakarta.apache.org/bcel/projects.html.

[10] Keller, R., and Hölzle, U., Binary Component Adaptation. *In Proc. ECOOP '98*, Brussels, Belgium, July 1998.