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 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 learned while 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, runtime.

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. Two well known examples are the Java platform, which supports dynamic class loading [1], and the .NET framework [2] which allows assemblies to be dynamically loaded and executed.

SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.

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. This approach can be quite powerful. For instance, it is possible to instrument the code so that proper resource control takes place [3], change the code so that it is possible to serialize and relocate executing threads in a cluster [4], perform program consistency checks according to security policies [5], redirect method calls to proxies, among others.

For making things clearer, let's consider an example. Suppose that you have downloaded an application from the Internet but you cannot really consider it 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 a foo. One 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 invocations 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 the first general purpose code instrumentation library for the .NET platform. It allows assemblies (the code load unit in .NET) to be easily manipulated into 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 a series of high level instrumentation patterns for assisting the programmer performing code instrumentation. Finally, RAIL allows assemblies to be instrumented at runtime, just before the types are actually defined in the Common Language Runtime (CLR) virtual machine.

The rest of this paper is organized as follows. Section 2 discusses related work. The implementation of RAIL is described in Section 3. Section 4 presents a simple application scenario. Section 5 draws the conclusions and discusses future work.

2. RELATED WORK

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. In terms of libraries for code instrumentation, probably one of the first worth noting is EEL [6]. EEL was a C++ library that allowed static instrumentation of executables.

The advent of generalized use of virtual machines with dynamic code loading, and particularly Java, fuelled an immense development in the field. Two of the most important libraries in this area are BCEL [7], which is now part of the Apache Project, and provides a high-level API for manipulating Java byte code, and JOIE [8] that also allows Java objects to be instrumented. Together, these libraries have been used on more than 37 projects [9] which instrument Java code to perform various tasks.

JOIE is a toolkit for constructing byte code transformers for Java. It provides a set of low-level primitives for manipulating Java classes, and higher-level interfaces that directly support common transformer styles. BCEL is a general purpose tool for "byte code engineering". It gives full control to the developer and is used as a base for a large number of projects, since the abstraction it provides of Java classes and byte code is very low level. An example of this is that the user must explicitly give the indexes of the constants in the symbol table, and must manually update these indexes if they become invalid. SERP [10] uses a similar approach as that of BCEL, based on an object representation of classes. Similarly, "Binary Component Adaptation" [11] allows components to be adapted and evolved on-the-fly. Han Lee's "Byte-code Instrumenting Tool" [12] allows the user to insert calls to analise methods anywhere in the byte code.

Javassist [13] uses a customizable class loader to instrument code at load-time. BCEL uses the same approach. The difference between these two libraries is that Javassist provides a higher level API to manipulate classes and bytecode. It hides the low level details, like indexes and references recalculation, that the programmer has to handle when using BCEL. JMangler [14] distinguishes from the rest by introducing a new technique for load-time code interception. JMangler modifies the system `ClassLoader` class implementation allowing applications that need to use there own class loaders to be instrumented. This is not possible to do in BCEL or Javassist.

One related area to code instrumentation is Aspect Oriented Programming (AOP) [15]. AOP allows the weaving of cross-cutting concerns into code (e.g. invariant validations on method calls, security checks, logging). Although AOP is quite limited for general code instrumentation; code instrumentation is an excellent tool for implementing AOP. For the .NET platform, John Lam's Common Language Aspect Weaver (CLAW) [16] was one of the first libraries to perform assembly manipulation by "weaving" aspects into class methods. Weave.Net [17] is another tool dedicated to AOP in the .NET platform. It uses the `CLIFileReader` [18] to access metadata and Intermediate Language (IL) code in assemblies.

As it was stated in the introduction, RAIL is the first general purpose instrumentation library for the .NET platform. RAIL's approach to application instrumentation is more wide-ranging than the solutions presented until now.

By means of a tree-like object-oriented structure representing the assembly and its components, RAIL supports structural [19] and behavioural [19,20] reflection. With RAIL it is possible to simultaneously perform low-level modifications in assemblies and high-level manipulations. RAIL is also a cross-language library, since it can be used from any language that compiles for the .NET platform.

One important aspect of RAIL is the abstraction of common instrumentation operations in usage patterns, easing the task of the programmer. This is especially relevant because, in general, performing code instrumentation is a hard low-level task which is quite error-prone.

The most related project to RAIL is AbstractIL [21] from Microsoft Research. AbstractIL transforms assemblies into a structured abstract syntax tree that can be manipulated using OCaml , F# and C# languages. 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 manipulating assemblies.

Finally, it is important to mention that although RAIL implements some of the abstractions found in other code instrumentation libraries for Java, internally, the way they are implemented is quite different. To illustrate this difference we can say, for instance, that the minimal load unit in Java is the Class and in .NET is the Assembly. We can also refer that the mechanisms used by these tools in Java to intercept class loading are not available in .NET. Customizable class loaders [1], for instance, are specific to the Java platform. This paper also details some of the problems, differences and lessons learned on implementing these abstractions in .NET.

3. IMPLEMENTING ASSEMBLY INSTRUMENTATION

RAIL is organized in four different modules. The first uses the `Mono.PEToolkit` [22] and it is responsible for reading the assembly. RAIL transforms the assembly into an object-oriented structure. `RAIL.Reflect` includes classes that represent the assembly and its modules, types, methods, properties, fields and events. `RAIL.MSIL` classes take care of the representation of IL instructions and method bodies. A set of high level functionalities is implemented under the `RAIL` namespace and allow to: Change type references across the assembly; Replace field access by other fields, methods and properties; Replace method calls; Replace IL instructions 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 allows the programmer to work at a high level, using classes that implement all these functionalities; or at the instruction level by directly manipulating the table of objects that represents the methods' code and the objects that represent the instructions. Instrumentation becomes a complex task when the insertion or deletion of program parts and instructions generate or destroy dependencies in the execution flow (e.g. removing a parameter from a method signature will invalidate all the references to that parameter).

The final step of all the instrumentations performed with RAIL is the conversion of the internal OO representation of the assembly into `System.Reflection.Emit` classes, dynamically generating an assembly [2].

3.1 Object-Oriented Representation of an Assembly

This section discusses the way RAIL represents assemblies. A Portable Executable (PE) file for the .NET platform [2] is considerably complex. Metadata describes the structure of an assembly and its components; it is stored in multiple tables inside the `.text` section of the PE along with the IL code.

A PE file is a module of an assembly. All the module files have the same structure. The structure of a Portable Executable file starts with an MS-DOS header, a COFF header, a PE header and several native PE sections like `.data`, `.rdata`, `.rsrc` and `.text` [2]. An assembly can be formed by several modules, and the difference between the main module and the rest is that the first one holds manifest information. Manifest is the name given to the metadata that describes the assembly and its modules.

Metadata is organized as a set of tables with information that describes the application and its components. Intermediate Language (IL) code, on the other hand is a stream of bytes representing instructions, values and references.

Metadata and IL code reference metadata using coded tokens. A coded token consists of a composed value, where a part of it refers to the target metadata table and the other part to the index of that table row.

The first task of RAIL is to read and interpret assemblies and then build a structured representation of them. RAIL's first layer transforms the assembly into an object-oriented tree structure down to the instruction level. RAIL creates the objects and builds the object structure that represents the assembly and its components based on data gathered from the metadata tables.

The hierarchy of assembly components follows the following structure from top down: Assembly, Module, Type and Type members. Type members are Properties, Fields, Constructors, Methods and Events. Enums and arrays are "special" types.

RAIL has a set of abstract classes that represent this hierarchy; they are `RAssembly`, `RModule`, `RType` and `RMember`. `RMember` class is specialized by `RProperty`, `RField`, `RMethodBase` and `REvent`; `RMethodBase` is a generalization for `RMethod` and `RConstructor` classes.

Each one of these abstract classes has two implementations: "Def" types and "Ref" types (e.g. `RTypeDef` and `RTypeRef`).

The difference between Def and Ref classes is that the first ones are part of the assembly being instrumented, it is possible to modify them and the others represent references to outside assembly members and they cannot be manipulated.

Altering the application by means of structural reflection consists in modifying the OO structure that represents the members of the assembly being instrumented.

3.2 IL Code Instrumentation

The classes of the `RAIL.MSIL` namespace represent the low level details of an assembly, such as the IL instructions and their operands.

RAIL represents the assembly structure using objects down to the instruction level. Since there are more than 200 different opcodes, an abstract class is used to represent a general instruction. Concrete classes such as `ILNone`, `ILString`, `ILMethod`, `ILType` and others represent instructions according to their operands. This way each class represents more than one kind of instruction, but with a common operator type.

IL instructions and metadata itself reference metadata using coded tokens which sometimes are stored in a compressed form in an assembly (signatures are also stored in a compressed form). RAIL has implemented methods that are able to decode these compressed values and to resolve encoded references to types. This way an object representing an instruction in RAIL does not have a token as an operand, but a reference to an object.

Resolving compressed values and tokens is essential when creating a representation of the IL code streams. Each instruction will be translated into an object entry into an array that contains the code of the method. The operands of each instruction are references to other objects such as:

- `RAIL.MSIL.Instruction`
- `RAIL.MSIL.LocalVariable`
- `RAIL.Reflect.RType`
- `RAIL.Reflect.RMethodBase`
- `RAIL.Reflect.RMember`

It is possible to manipulate the list of instructions directly. These manipulations should be done by experts because, the adding/removing/modifying instructions operations does not have any kind of security checks to avoid the generation of errors besides the ones implemented in the `ILGenerator` class of `System.Reflection.Emit`. This will be illustrated in section 3.4.

To create objects representing IL instructions there is a special factory class. This class provides a very exhaustive set of methods that allow the creation of any kind of Instruction object by its OpCode name (e.g. `ILFactory.Ldstr("Hello World")` returns an `ILString` object that represents the following code in IL [`ldstr "Hello World"`]).

Exception handling information is kept in a table. This table holds the objects that reference the instructions in the method at the start and end of the "try…catch" blocks. There are also objects pointing to the instructions that delimit the exception handler blocks according to their type.

3.3 Implementing High-Level Functionalities

The `RAIL.Reflect` and `RAIL.MSIL` namespaces define the classes that are used in creating an object-oriented representation of an assembly. The `RAIL.CodeTranformer` class provides the bases for high-level instrumentation of assemblies.

`CodeTransformer` holds a list of objects that specify the kind of transformations that should be applied to the assembly. `CodeTranformer` does not check or reorganizes the order that these transformers are applied to the assembly, and therefore does not verify if any kind of incompatibilities exist. This checking and verification should be done by the programmer.

To simplify the instrumentation process and the development of high-level functionalities, RAIL implements a Visitor pattern [23].

When designing a class using the Visitor pattern in RAIL, the programmer can implement methods that may "visit" the following components: Assembly; Modules; Types; Methods; Constructors; Fields; Properties; Events.

These classes can be used together with the `CodeTranformer` class as part of the list of transformations that it applies. RAIL already possesses a set of classes that implement the high-level functionalities described in the beginning of section 3.

3.4 Generating Assemblies

The process of assembly generation in RAIL is relatively simple. Basically, it consists of transforming the RAIL representation of an assembly into a .NET's `Reflection.Emit` representation.

To generate the IL code, RAIL provides a class that creates an `ILGenerator` object for each method. The generation of an IL instruction is done by executing the `Write()` method of the object representing the instruction. This method calls the `ILGenerator` with the appropriate parameters for each particular instruction and operand type.

3.5 Load-Time Instrumentation with RAIL

Up until now we have discussed how to represent and modify an assembly using RAIL. This support is enough to allow static manipulation of assemblies. Nevertheless a much more interesting functionality is code instrumentation at runtime.

In Java, most of the instrumentation libraries achieve this goal by writing their own customizable class loader (BCEL, JOIE, Javassist) [7,8,13]; by modifying the `defineClass()` method in the main `ClassLoader` of the platform using HotSwap technology [14]; or by modifying the platform bootstrap `ClassLoader` [11].

The .NET platform provides mechanisms to dynamically load assemblies, methods like `AppDomain.Load()` and `Assembly.Load()`. When RAIL generates the `AssemblyBuilder` object its classes and methods become available to the runtime environment. It is possible to save this assembly to disk so that it can be loaded latter on by an application.

The .NET platform also provides a redirect binding mechanism that allows providing to the CLR a different version of the assembly it is looking for.

In RAIL the technique used is to intercept the event that occurs when the CLR is not able to resolve an assembly reference. To do this RAIL registers a new method in the `ResolveEventHandler` delegate [2] associated with the event. This method should do all the modifications in the assembly being called and return a new version of it to the CLR.

4. EXAMPLE APPLICATION

This section illustrates a very simple example that was presented in the first section of this article, in which the disk accesses that an application makes are logged. The objective is to find out if there is any kind of untrusting operations going on without ones knowledge. This is a relevant scenario due to the fact that most of the times we use applications developed by third-parties. A lot of these applications are not very transparent about what kind of I/O operations they perform. We would like to know, for instance, in what files of our filesystem a certain program is writing to and reading from.

There are several ways of adding this functionality to an application. But, to illustrate how RAIL could be used in this situation, we will use the following approach:

Change all the references to system classes that implement the reading and writing onto the file system with references to other classes that proxy them, allowing the logging of the file accesses.

In C#, the `System.IO` namespace classes like `BinaryWriter`, `TextWriter`, `BinaryReader` and `TextReader` provide the methods needed to write and read into and from the filesystem. The first thing to do is to write proxy classes and implement them in a way that a log file is created of the operations being executed for posterior analysis. Then, compile this set of classes into an assembly (Assembly B).

Let's assume that we call the application being instrumented Assembly A and use RAIL to build an object-oriented representation of it and of the assembly where the proxy classes are defined (Assembly B).

From this point on two `RAssemblyDef` objects exist representing the assemblies.

The next code illustrates how to load an assembly using RAIL:

```
RAssemblyDef AssemblyA =
  AssemblyDef.LoadAssembly("AssemblyA.exe");
```

Next the proxy classes are copied into the assembly, this step is not essential, but it illustrates well the capabilities of RAIL to copy classes from one assembly to another.

It is necessary to obtain references to `RType` objects from assembly B implementing the proxy classes:

```
RType ProxyClassToBinaryWriter =
  AssemblyB.RModuleDef.GetType(
    "ProxyClassToBinaryWriter");
```

Then copy the RType obtained to Assembly A:

```
RTypeDef NewProxyClassToBinaryWriter =
  AssemblyA.RModuleDef.CopyType(
    ProxyClassToBinaryWriter,
    "ProxyClassToBinaryWriter");
```

Next it is necessary to get references to the original I/O classes:

```
RType OriginalBinaryWriter =
AssemblyA.GetType("System.IO.BinaryWriter");
```

To change the references to the original classes with references to the proxy classes through all the assembly, it is necessary to create a `ReferenceReplacer` object.

```
ReferenceReplacer rr =
  new ReferenceReplacer(
    OriginalBinaryWriter,
    NewProxyClassToBinaryWriter);
```

Finally apply the changes to the assembly and save it:

```
AssemblyA.Accept(rr);
AssemblyA.SaveAssembly("AssemblyA.exe");
```

ILDasm is an application that is part of the .NET platform and allows us to disassembly and see the IL code of an assembly. If

you use this tool to compare the original assembly and the instrumented one, you will immediately spot the differences in the code, because the calls to the original system classes are now replaced with the proxy classes. Plus you will find that these proxy classes are now part of the assembly.

5. CONCLUSIONS AND FUTURE WORK

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

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 manipulation. 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 them into the target assembly and give new implementations to the existing methods.

There is still work to be done in RAIL, like implementing the support of multiple module assemblies and resolving some important security questions when handling strong named assemblies, since the CLR does not allow modified strong named assemblies to be executed.

Dynamic code instrumentation is another open topic in this area. The possibility to modify assemblies and classes that are already loaded in the virtual machine has not yet been addressed.

RAIL could be used to transform applications in a way that they allow themselves to be modified after being loaded. Future work consists of designing the algorithms that would allow this to happen inside running programs.

The RAIL library is publicly available at [24].

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Lian, S., and Bracha, G. “Dynamic Class Loading in the Java Virtual Machine”. In *Proceedings of the Object-Oriented Programming Systems Languages and Applications (OOPSLA’98)*, ACM Press, Vancouver, Canada, 1998.

[2] ECMA International. Standard ECMA-335 Common Language Infrastructure (CLI), ECMA Standard, 2003.

[3] Binder, W., Hulaas, J., Villazón, A., and Vidal, R. “Portable Resource Control in Java: The J-SEAL2 Approach”. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001)*, ACM Press, Florida, USA, 2001.

[4] Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., and Verbaeten, P. “Portable Support for Transparent Thread Migration in Java”. In *Proceedings of the Joint Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA’2000)*, Springer-Verlag, LNCS 1882, Zürich, Switzerland, 2000.

[5] Chander, A., Mitchell, J., and Shin, I. “Mobile Code Security through Java Byte Code modification”, 1999. Available at: http://theory.stanford.edu/~vganesh/project.html.

[6] Larus, J. EEL – An Executable Editing Library, University of Wisconsin, Madison, U.S.A., 1996. Available at: http://www.cs.wisc.edu/~larus/eel.html.

[7] Dahm, M. “Byte Code Engineering”. I*n JIT '99 - Java-Informations-Tage, Informatik Aktuell,* Springer-Verlag, Dusseldorf, Germany, September 1999. ISBN 3-540-66464-5.

[8] Cohen, G.A., Chase, J.S., and Kaminsky, D.L.” Automatic Program Transformation with JOIE”. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, USA, 1998.

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

[10] White, A. A. SERP: Overview, 2002. Available at: http://serp.sourceforge.net.

[11] Keller, R., and Hölzle, U. “Binary Component Adaptation”. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP’98)*, Springer-Verlag, LNCS 1445, Brussels, Belgium, July 1998.

[12] Lee, H., and Zorn, B. G. “BIT: A Tool for Instrumenting Java Bytecodes”. *In Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, December 1997.

[13] Chiba, S. “Load-Time Structural Reflection in Java”. In *Proceedings of the European Conf. on Object-Oriented Programming (ECOOP’00),* Springer-Verlag, LNCS 1850, Sophia Antipolis and Cannes, France, June 2000.

[14] Kniesel, G., Costanza, P. and Austermann, M. “JMangler – A Framework for Load-Time Transformation of Java Class Files”. In *IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, November 2001.

[15] Elrad, T., Filman, R.E., and Bader, A. “Aspect-Oriented Programming”. In *Communications of the ACM*, ACM Press, New York, New York, USA, October 2001, Vol.44 (10), pp. 29-32. ISSN 0001-0782.

[16] Lam, J. “Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime”. D*emonstration at the 1st International Conference on Aspect-Oriented Software Development (AOSD2002)*, University of Twente Enschede, The Netherlands, 2002.

[17] Lafferty, D., and Cahill, V. “Language-Independent Aspect-Oriented Programming”. *In Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming (OOPSLA 2003)*, ACM Press, Anaheim, California, USA, October 2003.

[18] Cisternino, A. CLIFileReader Library, University of Pisa, Pisa, Italy, February 2004. Available at: http://dotnet.di.unipi.it/MultipleContentView.aspx?code=103

[19] Ferber, J. “Computational Reflection in Class Based Object-Oriented Languages”. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems,*

Languages and Applications (OOPSLA'89), ACM Press, New Orleans, Louisiana. October 1989.

[20] Malenfant, J., Dony, C. and Cointe, P. "Behavioral Reflection in a Prototype-Based Language". In *Workshop on Reflection and Meta-Level Architectures (IMSA'92)*, Tokyo, 1992.

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

[22] Chaban, S. Mono.PEToolkit. Mono Project. Novell, Inc. 2004. Available at: http://www.go-mono.com.

[23] Gamma, E., Vlissides, J., Johnson, R., and Helm, R., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Pub Co, October 1994. ISBN: 0-201-63361-2.

[24] Dependable Systems Group of the University of Coimbra, RAIL project, CISUC, 2004. Available at: http://rail.dei.uc.pt.