

Using Metadata Transformations as a Means to Integrate Class Extensions in an Existing Class Hierarchy

Markus Lumpe

Department of Computer Science
Iowa State University
113 Atanasoff Hall
Ames, IA, 50011-1040, USA
lumpe@cs.iastate.edu

TR #06-02
March 2006

Abstract. Class extensions provide a fine-grained mechanism to define incremental modifications to class-based systems when standard subclassing mechanisms are inappropriate. To control the impact of class extensions, the concept of *classboxes* has emerged that defines a new module system to restrict the visibility of class extensions to selected clients. However, the existing implementations of the classbox concept rely either on a “classbox-aware” virtual machine, an expensive runtime introspection of the method call stack to build the structure of a classbox, or both.

In this paper we present an approach for the .NET framework that allows the structure of a classbox to be constructed at compile-time by means of code instrumentation to rewire the inheritance graph of refined classes. The metadata transformations are language-neutral and more importantly preserve both the semantics of the classbox concept and the integrity of the underlying assemblies. As a result, metadata transformation provides a feasible approach to incorporate the classbox concept into programming environments that use a *virtual execution system*.

Using Metadata Transformations as a Means to Integrate Class Extensions in an Existing Class Hierarchy

Markus Lumpe
Department of Computer Science
Iowa State University
113 Atanasoff Hall
Ames, IA-50011, USA
lumpe@cs.iastate.edu

ABSTRACT

Class extensions provide a fine-grained mechanism to define incremental modifications to class-based systems when standard subclassing mechanisms are inappropriate. To control the impact of class extensions, the concept of *classboxes* has emerged that defines a new module system to restrict the visibility of class extensions to selected clients. However, the existing implementations of the classbox concept rely either on a “classbox-aware” virtual machine, an expensive runtime introspection of the method call stack to build the structure of a classbox, or both.

In this paper we present an approach for the .NET framework that allows the structure of a classbox to be constructed at compile-time by means of code instrumentation to rewire the inheritance graph of refined classes. The metadata transformations are language-neutral and more importantly preserve both the semantics of the classbox concept and the integrity of the underlying assemblies. As a result, metadata transformation provides a feasible approach to incorporate the classbox concept into programming environments that use a *virtual execution system*.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming; D.3.3 [Language Constructs and Features]: Classes and objects; D.3.3 [Language Constructs and Features]: Modules, packages

1. INTRODUCTION

It is generally accepted that the inheritance relationships supported by mainstream object-oriented and class-based languages are not powerful enough to express many useful forms of incremental modifications. To address this problem, several approaches have emerged (e.g., Smalltalk [12], CLOS [26], MultiJava [8], Scala [24], or AspectJ [14]) that

focus on a particular technique: *class extensions*. A class extension is a method that is defined in a packaging unit other than the class it is applied to. The most common kinds¹ of class extensions are the *addition* of a new method and the *replacement* of an existing method, respectively.

However, a major obstacle when specifying class extension is that their embodied changes have global impact [3]. Moreover, even if a system allows for a modular specification of class extensions (e.g., MultiJava [8] or AspectJ [14]), it may not support multiple versions of a given class to coexist at the same time. To remedy these shortcomings, Bergel et al. [2, 3] have recently proposed *classboxes*, a new module system that defines a packaging and scoping mechanism for controlling the visibility of isolated extensions to portions of class-based systems. Besides the “traditional” operation of *subclassing*, classboxes also support the *local refinement* of imported classes by adding or modifying their features without affecting the originating classbox. Consequently, the classbox concept provides an attractive and powerful framework to develop, maintain, and evolve large-scale software systems and can significantly reduce the risk for introducing design and implementation anomalies in those systems [3].

At present, there exist two implementations of classboxes in Smalltalk [3] and a restricted prototype in Java [2]. The first Smalltalk implementation relies on a modified, “classbox-aware” virtual machine in which a dedicated graph search algorithm implements local rebinding of methods. The second implementation uses a combination of bytecode manipulation and a reified method call stack to build the structure of a classbox. This technique is also applied in Classbox/J [2], an implementation of classboxes for the Java environment. In Classbox/J, a preprocessor translates each method redefinition into a `if` statement that uses a `ClassboxInfo` object to determine, which definition to call in the current context.

Common to all three implementations is that the integration of class extensions occurs at runtime by means of a specially-designed method lookup mechanism. This implementation scheme adds a significant execution overhead to redefined methods. For the Smalltalk implementations, for example, this overhead is generally in-between 25% to 60%, compared

¹Bracha and Lindstrom [5] have also presented a *hide* operator that renders a method of a class invisible to clients of that class.

to the “normal” method lookup [3]. Similarly, the method lookup of redefined methods in Classbox/J is on average 22 times slower than the normal method lookup [2].

In this paper we present an alternative implementation strategy that uses *metadata transformations* to integrate class extensions into a given class hierarchy. More precisely, we present a “classbox-aware” dialect of C# that defines a minimal extension to the C# language in order to provide support for the classbox concept, and Rewire.NET, a metadata adapter that implements a *compile-time* mechanism to incorporate the local refinements defined in a classbox into their corresponding classes. This approach allows us to treat standard .NET assemblies as classboxes, that is, we can import classes originating from standard .NET assemblies into a newly defined classbox, apply some local refinements to those classes, and generate a classbox assembly that is backward-compatible with the standard .NET framework. As a result, we obtain a mechanism that supports the coexistence of non-classbox-aware and classbox-aware software artifacts in one system and therefore allows for phased and fine-grained software evolution approach.

Our approach to incorporate the classbox concept into the .NET framework uses *code instrumentation* [6, 7, 13, 16, 17] to *rewire* the inheritance graph of a class hierarchy in order to build the structure of a classbox. This approach preserves the original semantics of the classbox concept while moving the process of constructing the structure of a classbox from runtime to compile-time. Furthermore, the application of metadata transformations allows us to use the standard method lookup mechanism for redefined methods. No dynamic introspection of the method call stack is required.

A key aspect of our approach is that a growing number of modern programming systems compile program code into a platform-independent representation that is executed in a *virtual execution system*. The virtual execution system provides an *abstract machine* to execute *managed* code. The two most known virtual execution systems are the Java platform [18] and the Common Language Infrastructure (CLI) [23]. Common to both systems is that the concrete layout of classes is not specified. This decision rests with the implementation of the virtual execution machine or a corresponding just-in-time (JIT) compiler. Both, Java and the CLI use a combination of *Intermediate Language* (IL) bytecode and *metadata*. Metadata provides the means for *self-describing* units of deployment in these systems. Besides application-specific resources like images or custom attributes, metadata contains information to locate and load classes, lay out instances in memory, resolve method invocations, and enforce security constraints. In other words, it is metadata and not the IL code that defines the structure of classes and their underlying class hierarchies. Rewire.NET exploits this special relationship between IL-bytecode and metadata in order to bind class extensions defined in a given classbox to their corresponding classes at compile-time.

The rest of this paper is organized as follows: in Section 2, we describe the classbox programming model for the .NET framework. In Section 3 we present the architectural elements to map the classbox concept to the CLI. We discuss the implementation of Rewire.NET in Section 4 and analyze

our approach in Section 5. In Section 6 we provide a brief overlook of related work. We conclude this paper in Section 7 with a summary of the presented work and outline future activities in this area.

2. CLASSBOXES AND .NET

The most important contribution of the classbox concept is that it provides a feasible solution to cope with *unanticipated changes* [3]. A classbox defines a well-delimited scope that confines the impact of class extensions to the defining classbox, avoiding conflicts that may break other parts of the system.

2.1 Classbox Characteristics

The main characteristics of classboxes can be summarized as follows [3]:

- A classbox is an explicitly named unit of scoping in which classes (and their associated members) are defined. A class belongs to the classbox it is first *defined*, but it can be made visible to other classboxes by either *importing* or *extending* it.
- Any extension applied to a class is only visible to the classbox in which it occurs first and any classboxes that either explicitly or implicitly import the extended class. Hence, redefining a particular method of a class in a given classbox will not have an effect on the originating classbox.
- Class extensions are only locally visible. However, their embodied refinements extend to all collaborating classes within a given classbox, in particular to any subclasses that are either explicitly imported, extended, or implicitly imported.

In addition, there are three critical aspects in the definition of the classbox semantics [2, 3] that need to be satisfied, when adding support for the classbox concept to a new programming environment:

Implicit import. The import mechanism provided by languages like Java or C# is non-transitive, that is, a declaration namespace `ns` cannot export a class `C`, if `C` was imported rather than defined in `ns`. In contrast, the module concept defined by classboxes uses transitive import. More precisely, if a classbox `cb` explicitly imports a class `C`, then all of `C`'s superclasses are *implicitly* imported into `cb` also. This not only allows for a local refinement of the explicitly imported class `C`, but also for a refinement of all other classes in the inheritance graph of `C` in `cb`.

Method extension. The decision, whether a method `m` is added or acts as replacement depends on its signature. That is, if a locally refined class `C` already defines a method with the same name and signature, then `m` replaces this method. Otherwise, `m` is added to `C`. Moreover, method replacement takes precedence in a *flattened* version of class `C` [3].

Identity of classes. A key element of the semantics of classboxes is that the identity of locally refined imported classes is preserved. By preserving the identity of a class `C`,

existing clients of C can benefit from the extensions applied to C also.

2.2 Issues and Limitations

Both, the original classbox concept [3] and Classbox/J [2] rest upon virtual methods and dynamic binding. There are no provisions for non-virtual methods. As a result, we treat each method extension as a virtual method.

The decision, whether a method m is added or replaced in a given class C that occurs locally refined in a classbox cb is based on the members defined by C and its superclasses. If a subclass of C , say class D , is also explicitly imported into cb , then D should benefit from the extensions applied to C . However, if D defines its own version of m , then this method may hide C 's method m , effectively rendering parts of the class extensions applied to C invisible to clients of D . A “classbox-aware” compiler can detect this situation, but the classbox concept is blind for this behavior.

The classbox semantics as given in [3] defines two flattening operations: one for classes, and one for classboxes. These operations are defined by means of a recursive expression of the form $\mu x.E = E\{\mu x.E/x\}$. Extending an imported class C with a class extension δ is specified by the term $\delta \triangleright (\mu \phi. \beta(\varepsilon \triangleright \phi))C$, where both ε and ϕ are class namespaces (i.e., method dictionaries), β is function from class namespaces to class namespaces, and \triangleright denotes method overriding from left-to-right. Now, since the class extension δ occurs outside the recursive expression $(\mu \phi. \beta(\varepsilon \triangleright \phi))$ it seems that this classbox operations should exhibit a behavior similar to the `new` specifier of C# [9], which can be used to hide any superclass method by declaring a `new` method with the same signature in a subclass. But this interpretation does not align with the semantics of the extension operator. The problem is that $\delta \triangleright (\mu \phi. \beta(\varepsilon \triangleright \phi))C$ correctly captures method addition, but not method replacement. To express method replacement, we need to move δ inside the recursive expression, that is, we have to write $(\mu \phi. \beta(\delta \triangleright (\varepsilon \triangleright \phi)))C$. Changing the specification this way captures the fact that method replacement is governed by dynamic binding and does not render this method invisible to the other members of the extended class.

Classbox/J extends the original classbox concept with an additional operator to access the *original* behavior of a re-defined method [2]. This operator was added to Classbox/J in order to facilitate the transformation of the Swing class hierarchy into a classbox. This operator adds a subtle dimension to the classbox concept, as it allows for a situation very similar to multiple inheritance: a *super*-call can be considered as accessing a method defined in one parent, whereas an *original*-call accesses a method defined in a second parent, with both parents derived from a common ancestor. Hence, having both, *super*- and *original*-calls, may lead to the well-known *diamond conflict* [4]. Moreover, in order to implement the `original` operator and to retain the code associated with the original method, we need to expand the `original`-call site² into the actual implementation of the original behavior (i.e., inline the original method into the

²Classbox/J uses source code refactoring for this purpose, that is, all method redefinitions are fused in one common method body that uses a cascade of `if` statements to select

replacement method). However, rather than using method inlining in this work, we use an approach, which is based on method overriding that will allow access to the original behavior through super-calls.

2.3 Classbox-aware C#

The .NET framework is a programming model for developing, deploying, and running component-based systems. The concepts that enable a component-oriented software development approach in the .NET framework are *namespaces* and *assemblies* [9]. Namespaces provide a logical construct to define and organize code and *globally-unique* types [9]. All namespaces have implicitly public access. Moreover, namespaces are used both as a tool to internally structure a program and as an external management system to present their functionality to other components or programs.

Assemblies are the units for physical packaging and deployment. They are structured in compliance with the *Common Language Infrastructure (CLI)* [23], a specification that provides a basis for a *virtual execution system* insulating programs from the underlying operating system. An assembly is a set of one or more files or modules deployed as a unit that contains types, executable code, and metadata. Metadata provides the means for *self-describing* assemblies.

To ally the classbox concept with the .NET framework, we introduce the notions of *logical* and *physical* structure of a classbox. These concepts do not change the underlying semantics of the classbox model, but provide us with the means to separate the program interface from the implementation of a classbox. The logical structure of a classbox defines a namespace to specify the *import* of classes, the introduction of *subclasses*, and the *extension* of classes. The physical structure of a classbox, on the other hand, identifies the assemblies that contain the executable code that is specified by the logical structure of a classbox.

To provide support for the classbox concept in the .NET framework, we define a “classbox-aware” dialect of C#³:

Class Import. To explicitly import a class, we use the *alias* form of the C# `using` directive [9, §16.4.1]. An alias for a type is a user-defined name that is only available within the namespace body that introduces it. However, in contrast to standard C#, the *using-alias-directive* in classbox-aware C# creates an “empty” subclass with the same name for each explicitly imported class in the importing classbox. This approach not only enables the local refinement of the explicitly imported class, but publishes the explicitly imported class to clients of the importing classbox as it had been defined in the importing classbox itself. The introduction of a new subclass does not preserve the identity of classes as required by the classbox model. To restore it, we apply `Rewire.NET` to the assemblies constituting the physical structure of the corresponding classbox.

Subclassing. Subclassing is represented by the standard the right definition and therefore allows direct access to the original behavior.

³We are currently experimenting with the open-source Mono compiler in order to define a frontend for classbox-aware C# [19].

```

using System;

namespace TraceAndColorCB
{
    using System.Drawing;

    using Point = OriginalCB.Point include
    {
        private Color color;

        public Color Color { get { return color; }
                             set { color = value; } }

        public void MoveBy( int dx, int dy )
        {
            Console.WriteLine( " MoveBy: {0}, {1}",
                               new object[] { dx, dy } );
            base.MoveBy( dx, dy );
        }
    }

    using LinearBPoint = OriginalCB.LinearBPoint;
}

```

Listing 1: Classbox `TraceAndColorCB` in classbox-aware `C#`.

class building mechanisms. The available `C#` language abstractions suffice to specify this operation. A subclass introduces a new type name in the defining classbox. This type name must be unique. However, the classbox concept allows for the coexistence of both the new subclass and implicitly imported classes with identical names in the same classbox.

Class Extension. We use the modified *alias* form of the `C#` `using` directive and add an `include`-clause to specify the local refinements to an imported class. The members of the local refinements are specified in a *class-body* [9, §17.1.3]. All methods and properties are implicitly marked `virtual`. If the extended class already defines a member with the same name and signature, then this member becomes overridden (i.e., replaced). Otherwise, the extension is added to the class. Extending an imported class results in a new subclass with the same name in the importing classbox. As in the case of class import, we have to use `Rewire.NET` to restore the class identity.

To illustrate the new language abstractions, consider the specification of the classbox `TraceAndColorCB`, as shown in Listing 1. The namespace `TraceAndColorCB` defines the logical structure of the classbox `TraceAndColorCB` in which we explicitly import two classes originating from classbox `OriginalCB`: `Point` and `LinearBPoint`. In `TraceAndColorCB`, we extend class `Point` with the property `Color` (utilizing a private instance variable `color`) and the method `MoveBy` that defines a tracing facility to monitor invocations of `MoveBy`. The method `MoveBy` overrides (i.e., replaces) an exiting method in class `Point`. It defines also an access to the original behavior through a `base`-call. The property `Color`, on the other hand, is new and therefore added to the refined class `Point` in classbox `TraceAndColorCB`. The class `LinearBPoint` is a subclass of class `Point` that defines a non-constant linear upper bound for point objects. Therefore, the local refinements defined for class `Point` impact class `LinearBPoint` also, that is, it possesses now a property `Color` and a method `MoveBy` with a tracing facility.

```

using System;

namespace TraceAndColorCB
{
    using System.Drawing;

    public class Point : OriginalCB.Point
    {
        public Point( int ix, int iy ) : base( ix, iy ) {}

        private Color color;

        public virtual Color Color { get { return color; }
                                     set { color = value; } }

        public override void MoveBy( int dx, int dy )
        {
            Console.WriteLine( " MoveBy: {0}, {1}",
                               new object[] { dx, dy } );
            base.MoveBy( dx, dy );
        }
    }

    public class LinearBPoint : OriginalCB.LinearBPoint
    {
        public LinearBPoint( int ix, int iy, int ibound ) :
            base( ix, iy, ibound ) {}
    }
}

```

Listing 2: Classbox `TraceAndColorCB` in standard `C#`.

The classbox-aware `C#`-compiler translates the specification of the classbox `TraceAndColorCB` into an internal representation that corresponds to the standard `C#`-code shown in Listing 2. Each explicitly imported class results in a new class definition in which the imported class becomes the direct supertype. Moreover, in order to preserve all constructors defined by class `Point` and `LinearBPoint`, we add corresponding “empty” constructors to the new class definitions. This approach prevents the automatic insertion of a *default*-constructor that would render the original constructors invisible.

The result of compiling the classbox `TraceAndColorCB` is the assembly `TraceAndColorCB.dll` that together with the assembly `OriginalCB.dll` (i.e., the assembly defining the classbox `OriginalCB`) constitute a *provisional* physical structure of the classbox `TraceAndColorCB`. In the provisional structure, the identity of imported classes has not yet been established. To restore the identity of imported classes, we have to rewire the inheritance graph of the classes `Point` and `LinearBPoint` by using `Rewire.NET`. The result is the final physical structure of the classbox `TraceAndColorCB`.

3. CLASSBOXES AND CLI

Each CLI-enabled language has to define a language-appropriate scheme to represent types and members in metadata. At the core of every CLI-enabled programming language are built-in data types compliant with the Common Type System (CTS), mechanisms to combine them to construct new types, and a facility to assign names to new types in order to seamlessly integrate them in the CLI [23].

The CLI uses an implementation-dependent declarative encoding mechanism to represent metadata information, called *metadata token*. A metadata token is a scoped typed identi-

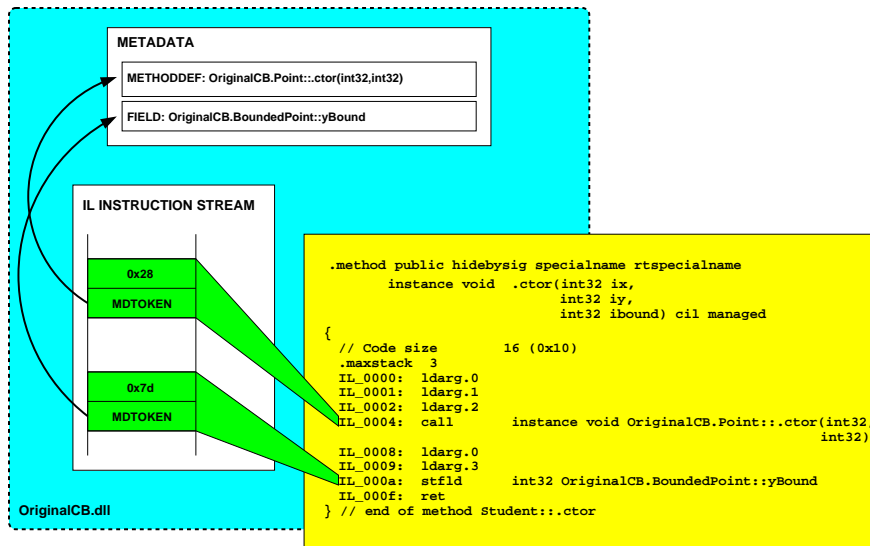


Figure 1: Layout of the constructor code of class `BoundedPoint` in assembly `OriginalCB.dll`.

fier of a metadata object and is represented as a *read-only* index into a corresponding metadata table. Consider, for example, the constructor for class `BoundedPoint` defined in classbox `OriginalCB` (cf., Listing 3) shown below:

```
public BoundedPoint( int ix , int iy , int ibound ) :
    base( ix , iy )
{
    yBound = ibound;
}
```

When compiled into the CLI (i.e., into the classbox assembly `OriginalCB.dll`), the constructor of the class `BoundedPoint` has a CLI-layout as shown in Figure 1. The constructor IL-bytecode contains two references to metadata tokens: the `METHODDEF` token `OriginalCB.Point::.ctor` and the `FIELD` token `OriginalCB.BoundedPoint::yBound`. In the CLI, a `METHODDEF` token encodes among other information the name of a method, its signature, and defines an index into the `PARAM` table that contains a contiguous run of `PARAM` metadata tokens describing the parameters owned by the method. A `FIELD` token, on the other hand, encodes the name, signature, and attributes of an instance variable.

3.1 Metadata Type Declarations

In CLI, new types are introduced via metadata type declarations [23]. `TYPDEF` tokens encode the name of a type, its declaration namespace, the super type (index into `TYPDEF` or `TYPREF` table), an index into the `FIELD` table that marks the first of a continuous run of field definitions owned by this type, and an index into the `METHODDEF` table that marks the first of a continuous run of method definitions owned by this type. In addition, a given assembly can refer to types defined in another module or assembly. These references are encoded by `TYPREF`, `MEMBERREF`, and `ASSEMBLYREF` tokens, respectively. A `TYPREF` token encodes the resolution scope (e.g., index into `ASSEMBLYREF` table), the name of the type, and its declaration namespace. `MEMBERREF` tokens are references used for both fields and methods of a class defined in another assembly. `MEMBERREF` tokens encode the type that owns the member, the member’s name, and its signature.

Finally, `ASSEMBLYREF` is a metadata token, which encodes the information that uniquely identifies another assembly on which the current assembly is depending. `ASSEMBLYREF` tokens not only encode the name to the referenced assembly, but also its version, which enables a deployment mechanism that allows for multiple versions of assemblies with the same name to coexist on the one system.

To illustrate a concrete metadata structure, consider the namespace `OriginalCB` as shown in Listing 3. `OriginalCB` does not contain any classbox-aware language elements. It is actually a standard `.NET` software unit, which defines classes that can be imported into a classbox (e.g., the classbox `TraceAndColorCB`). Compiling `OriginalCB` yields the assembly `OriginalCB.dll`, a *managed* dynamic link library that contains metadata, which has a structure⁴ as shown in Figure 2. When the assembly `OriginalCB.dll` is loaded at runtime, the CLI loader imports the metadata into its own in-memory data structures, which can be browsed via *Reflection* services. Both the metadata in an assembly and the corresponding in-memory runtime structures are immutable. However, they provide fast and direct access to required type information.

Metadata is organized in tables, whose rows start with index 1. Metadata may contain unreachable rows, but an index into a table must denote a valid row in that table. The indices into the metadata tables create a static dependency or *link* graph. For example, the `TYPDEF` token with the index 2 defines the members of class `Point`. That is, the class `Point` is derived from the class `Object`, which is defined in namespace `System` that is defined in assembly `mscorlib.dll` with the version number 2.0.0.0. Furthermore, the class `Point` defines two instance variables that are described in the `FIELD` table beginning at index 2, and seven methods that are described by in the `METHODDEF` table beginning at index 7. A run of `PARAM` or `METHODDEF` tokens continues to

⁴The reader should note that without loss of generality we only consider the metadata tokens, which are used by `Rewire.NET`.

```

using System;

namespace OriginalCB
{
    public class Point
    {
        private int x, y;

        public Point( int ix , int iy ) { x = ix; y = iy; }

        public int X { get{ return x; } set{ x = value; } }
        public int Y { get{ return y; } set{ y = value; } }

        public virtual void MoveBy( int dx, int dy )
        { X += dx; Y +=dy; }
        public virtual void MoveByXY()
        { this.MoveBy( X, Y ); }
    }

    public class BoundedPoint : Point
    {
        private int yBound;

        public BoundedPoint( int ix , int iy , int ibound ) :
            base( ix , iy ) { yBound = ibound; }

        public virtual int Bound { get{ return yBound; } }

        public override void MoveBy( int dx, int dy )
        { if ( Y + dy < Bound ) base.MoveBy( dx, dy ); }
    }

    public class LinearBPoint : BoundedPoint
    {
        public LinearBPoint( int ix , int iy , int ibound ) :
            base ( ix , iy , ibound ) { }

        public override int Bound { get{ return X; } }
    }
}

```

Listing 3: Classbox OriginalCB.

the smaller index. In addition, the reader should note that `.ctor` is the name of the constructor in CLI. Furthermore, properties are encoded as *getter* and *setter* methods. For example, both `get_X` and `set_X` stand for the property `X` defined in the class `Point`.

3.2 Dynamic Graph Search

Common to both the Smalltalk and the Java implementations of classboxes is a specially-designed method lookup mechanism that performs a dynamic search over a classbox graph in order to ensure that import takes precedence over inheritance [2,3]. More precisely, if a given method cannot be located in the current imported class, then rather than continuing with the superclass, the modified lookup tries to locate the required method in the provider classbox. Only if the requested method cannot be located in the provider classbox, then the search continues in the imported class' superclass. The effect of this method lookup mechanism is that local refinements to imported classes are *dynamically* linked into the corresponding class hierarchy. In other words, extending an imported class is an operation that is performed at *runtime*.

Consider, for example, Figure 3 in which we highlight the search for the property `Color` for the class `LinearBPoint` imported into the classbox `TraceAndColorCB`. The lookup starts in the class `LinearBPoint` (denoted by '1') and as

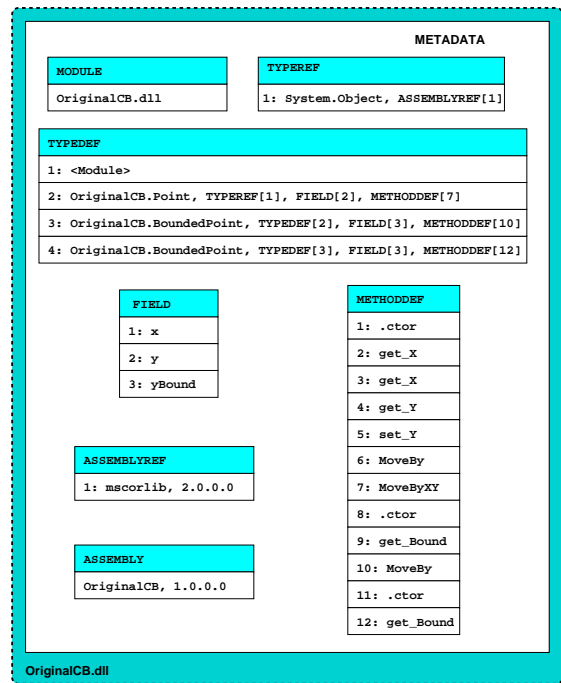


Figure 2: Metadata structure in assembly OriginalCB.dll.

this class neither implements nor has been extended with a corresponding property, the lookup continues in the implicitly imported class `BoundedPoint` denoted by '2'. Again, the class `BoundedPoint` does not implement the `Color` property. Therefore, the search has to continue by inspecting its superclass. However, since we have defined an extension to class `Point` (we use the rounded box as a graphical means to indicate that the class `Point` has been extended with a `Color` property and the method `MoveBy`), the search terminates in the extension that defines the property `Color` (denoted by '3') rather than in the class `Point` directly, as this is the first point along the search path that implements the property `Color`.

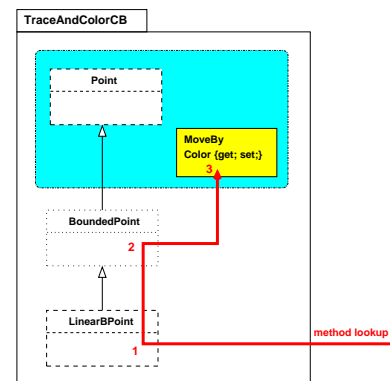


Figure 3: Method lookup as search over the classbox graph.

3.3 Changing the Metadata

To move the process of creating the structure of a classbox from runtime to compile-time, we take advantage of the separation of metadata and IL-bytecode. Both, the import of a class and extending an imported class trigger the creation of a new subclass with the same name as outlined in Sec-

tion 2.3. However, subclassing is an operation that breaks the connection to former clients [11]. To restore this connection and to enable a former clients of the extended class to benefit from the local refinements, we have to redirect the supertype edge of any direct explicitly or implicitly imported subclass of a refined class to the newly created class in the current classbox.

Consider again the classbox `TraceAndColorCB`. This classbox explicitly imports the classes `Point` and `LinearBPoint` from `OriginalCB`. As a result, we create two new subclasses with the same name in `TraceAndColorCB`. The resulting inheritance graph is shown in Figure 4 (explicitly imported classes are marked with a solid rounded box, whereas implicitly imported types are marked with a dotted rounded box).

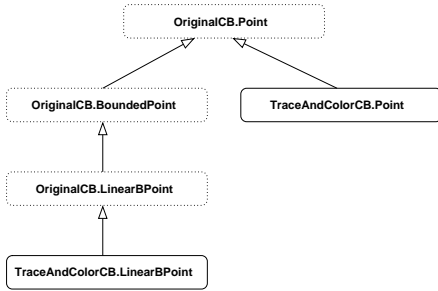


Figure 4: Inheritance graph before flattening.

A name of a type in CLI consists of two elements: a *type-name* and a *namespace*. Therefore, when we introduce the new subclasses for explicitly imported types, we create a new name in which the namespace component identifies the importing classbox. The scheme allows for the coexistence of different versions of a class in the same classbox, since it is always possible to distinguish them by using their namespace name.

In the provisional structure of classbox `TraceAndColorCB`, the class `TraceAndColorCB.Point` is not in the inheritance graph of class `TraceAndColorCB.LinearBPoint`. As a consequence, the class `TraceAndColorCB.LinearBPoint` does not yet benefit from the local refinements applied to the class `TraceAndColorCB.Point`, as required by the classbox model. To change this, we have to make `TraceAndColorCB.Point` a direct supertype of class `OriginalCB.BoundedPoint`. To accomplish this, we change the `TYPEDEF` metadata token defining the class `OriginalCB.BoundedPoint` in the metadata of the assembly `OriginalCB.dll`. More precisely, we need rewire the `Extends` column of `OriginalCB.BoundedPoint`'s `TYPEDEF` metadata token to point to the `TYPEDEF` metadata token defining class `TraceAndColorCB.Point` in assembly `TraceAndColorCB.dll`. We proceed by performing the following instructions:

1. Create a new version of `OriginalCB.dll` and name this assembly `OriginalCB(1).dll`.
2. Add an `ASSEMBLYREF` token for `TraceAndColorCB` to the metadata of `OriginalCB(1).dll`.
3. Add a `TYPDEF` token for `TraceAndColorCB.Point` to the metadata of `OriginalCB(1).dll`.

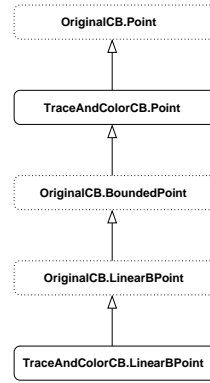


Figure 5: Flatted classbox `TraceAndColorCB`.

4. Set the `Extends` column of the `TYPEDEF` token for class `OriginalCB.BoundedPoint` to point to the newly added `TYPDEF` token in `OriginalCB(1).dll`.

The result of this transformation is a *flattened* classbox that publishes two classes: `Point` and `LinearBPoint`, whose inheritance graph is shown in Figure 5.

The metadata manipulations do not effect existing clients of `OriginalCB`, since we create a new version for this assembly, before applying the transformations. Moreover, in contrast to `Classbox/J`, we do not need access to the original source code to create to structure of a classbox. The logical structure of classbox `TraceAndColorCB` is defined by the static link graph in metadata of its corresponding physical representation, that is, the assemblies `TraceAndColorCB.dll` and `OriginalCB(1).dll`, as shown in Figure 6.

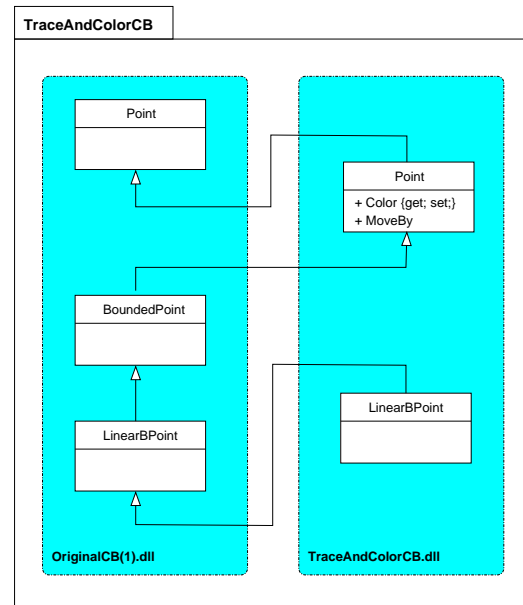


Figure 6: Structure of classbox `TraceAndColorCB`.

3.4 Restoring Constructor Integrity

The rewiring process outlined in the previous section manipulates metadata, but not the IL-bytecode. The process preserves the integrity of metadata, that is, all indices to tables in metadata denote a valid row.

```

.method public hidebysig specialname rtspecialname
        instance void .ctor(int32 ix,
                            int32 iy,
                            int32 ibound) cil managed
{
    // Code size          19 (0x13)
    .maxstack      8
    IL_0000:  ldarg.0
    IL_0001:  ldarg.1
    IL_0002:  ldarg.2
    IL_0003:  call         instance void OriginalCB.Point::.ctor(int32,
                                                    int32)

    IL_0008:  nop
    IL_0009:  nop
    IL_000a:  ldarg.0
    IL_000b:  ldarg.3
    IL_000c:  stfld         int32 OriginalCB.BoundedPoint::yBound
    IL_0011:  nop
    IL_0012:  ret
} // end of method BoundedPoint::.ctor

```

Figure 7: ILDASM view of broken IL-bytecode for `BoundedPoint` constructor in assembly `OriginalCB(1).dll`.

Unfortunately, changing the *Extends* column of the `TYPEDDEF` token describing class `BoundedPoint` does not preserve the integrity of the IL-bytecode in `OriginalCB(1).dll`. To illustrate the problem, consider the disassembled IL-bytecode for the constructor of class `BoundedPoint` shown in Figure 7.

In order to initialize a new object being created for a given class, the constructor for that class always calls its *statically known* superclass constructor first. In the original assembly `OriginalCB.dll`, this statically known superclass constructor is `OriginalCB.Point::.ctor` (cf., line IL.0003 in Figure 7). The situation in the assembly `OriginalCB(1).dll` is different, however, as we have changed the supertype of the class `BoundedPoint` to `TraceAndColorCB.Point`. It is, therefore, not correct to call `OriginalCB.Point::.ctor`. As a consequence, the IL-bytecode for the constructor of the class `BoundedPoint` loses its integrity, since object initialization cannot *skip* classes.

We can, however, easily restore the required integrity. The target of a static method call is indicated by a *method descriptor*. This method descriptor is a metadata token (either `METHODDEF` or `MEMBERREF`) that describes the method to call and the number, type, and order of the arguments that have been placed on the stack to be passed to that method. In other words, it is the method descriptor and not the IL-bytecode that determines the destination address of a method call. We exploit this fact, to restore the broken IL-bytecode integrity of constructor for the class `BoundedPoint` in assembly `OriginalCB(1).dll`, as follows:

1. Add a `MEMBERREF` token indicating the constructor for the class `TraceAndColorCB.Point` to the metadata of `OriginalCB(1).dll`.
2. Construct, using the new `MEMBERREF` token, a new method descriptor for `TraceAndColorCB.Point::.ctor`.
3. Use the *Relative Virtual Address* (i.e., the *RVA* column) of the `METHODDEF` token describing the constructor for the class `BoundedPoint` to locate the method descriptor for `OriginalCB.Point::.ctor` and replace it with the descriptor built in the previous step.

Using these instructions, the integrity of the constructor for the class `BoundedPoint` in assembly `OriginalCB(1).dll` is restored. As a result, we have obtained the final physical structure of the classbox `TraceAndColorCB`. The assemblies `OriginalCB(1).dll` and `TraceAndColorCB.dll` are standard .NET assemblies and pass verification. Thus, we can use them like any other non-classbox-aware assembly. The structure of the classbox `TraceAndColorCB` is imprinted in the metadata of the underlying assemblies.

4. IMPLEMENTATION

Rewire.NET is a .NET component, written in C#, that accepts as input a *rewiring specification* that lists the target classbox, the referenced assemblies, and all explicitly imported classes. Rewire.NET analyzes the provisional physical structure of the target classbox and performs the necessary transformations to produce a final physical structure of the target classbox.

The implementation of Rewire.NET has one subsystem for the representation of assemblies, called CLI. The CLI subsystem is a namespace that defines a collection of classes that provide an object-oriented interface to read, alter, and write .NET assemblies.

4.1 The CLI Subsystem

Several methods and tools have been proposed to perform assembly introspection. The .NET framework already provides the `System.Reflection` API, which can be used for this purpose. Using the services provided by this API, we are able to programmatically obtain the metadata contained in an assembly. Unfortunately, this API lacks the ability to access IL-bytecode. However, as outlined in Section 3.4, we need access to the IL-bytecode in order to restore the integrity of a constructor, whose class was subject to a supertype change. We face a similar problem with the *Metadata Unmanaged API* [22] that can be used by a compiler to query the metadata of a host assembly and emit the correspondingly updated information into a new version of the host assembly.

A framework that provides access to both metadata and IL-bytecode is the *Runtime Assembly Instrumentation Library*

(RAIL) [6]. RAIL closes the gap between the reflection capabilities in the .NET framework and its support for code emission. RAIL offers an object-oriented interface for an easy manipulation of assemblies, modules, classes, and even IL-bytecode. Nevertheless, RAIL cannot be used for the implementation of Rewire.NET, as this API does not allow for the manipulation of type references. RAIL treats type references (i.e., `TYPREF` metadata tokens) as read-only pointers to members defined outside the current assembly being instrumented.

The CLI API addresses these shortcomings. The primary purpose of this API is to provide an object-oriented view of an assembly with a symmetric support for reading and writing Portable Executable files. In addition, the CLI API defines mechanisms to manipulate the metadata of an assembly and to fetch the IL-bytecode. It does, however, not define any IL-bytecode manipulation capabilities, except for the update of method descriptors. We can use the `Reflection.Emit` API or RAIL for IL-bytecode instrumentation.

The file format for an assembly is a strict extension of the Portable Executable (PE) file format [23]. The structure of a Portable Executable file starts with a set of file headers (i.e., *MSDOS Header*, *PE Header*, and *PE Optional Header*). Following these headers is a set of section headers that describe the native PE sections contained in the assembly (e.g., the code section `.text`). The feature that distinguishes a managed from an unmanaged assembly is the presence of a *CLI Header* through which the metadata of the assembly is located. Metadata is organized as a set of tables, whose elements contain, for example, the physical representation of the logical string heap, the signatures of fields and methods, or metadata tokens describing type declarations, member references, etc.

At the center of the CLI API is the class `Assembly`, which is composed from the core elements of the extended Portable Executable file format, as shown in Figure 8.

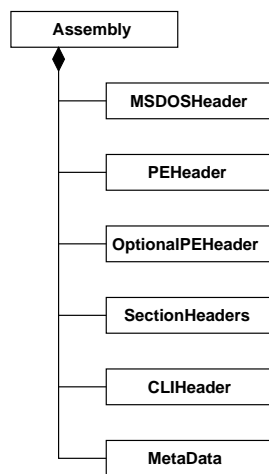


Figure 8: CLI.Assembly.

The class `Assembly` represents an in-memory image of a Portable Executable file. It provides access to the structure of the runtime file format of an assembly. The class

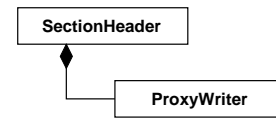


Figure 9: CLI.SectionHeader.

`Assembly` defines both a `Read` and a `Write` method to load an assembly into memory and to create a new PE image, respectively. However, rather than retaining the contents of all native PE sections in memory, the `Read` method constructs a `ProxyWriter` object and associates it with its corresponding section header (cf., Figure 9). The class `ProxyWriter` defines a method `FetchILMethod` to acquire the IL-bytecode associated with a given Relative Virtual Address (RVA), a method `Update` that takes a byte array and a RVA to change the byte sequence starting at RVA in the associated section data, and a method `Copy` that writes the associated section data to a new Portable Executable file.

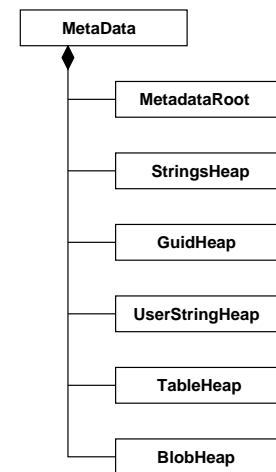


Figure 10: CLI.MetaData.

The class `MetaData`, as shown in Figure 10, represents the logical format of metadata. It provides access to all metadata stream heaps. These stream heaps are structured as tables and provide an index-based access to rows. Furthermore, each heap defines an `Add` method to append a new row to a table. Stream heaps do not allow for the removal of a row. Deleting a row may destroy the integrity of metadata. However, stream heaps may contain *garbage*, that is, rows that are not indexed by either metadata or IL-bytecode.

4.2 Rewire.NET

Rewire.NET is a *Console Application* that reads the rewiring specification that is generated by the classbox-aware `C#` compiler while compiling a classbox. The format of the rewiring specification is given in Figure 11.

Specification ::= { *Definition* }*

Definition ::= **R** # *ReferencedAssemblyFileName*
 | **T** # *ClassboxAssemblyFileName*
 | **I** # *ExplicitlyImportedClass*
 | **N** # *ClassboxName*

Figure 11: Syntax of Rewire.NET specifications.

```

.method public hidebysig specialname rtspecialname
    instance void .ctor(int32 ix,
                       int32 iy,
                       int32 ibound) cil managed
{
    // Code size          19 (0x13)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldarg.1
    IL_0002: ldarg.2
    IL_0003: call         instance void [TraceAndColorCB]TraceAndColorCB.Point::.ctor(int32,
                                                                                       int32)

    IL_0008: nop
    IL_0009: nop
    IL_000a: ldarg.0
    IL_000b: ldarg.3
    IL_000c: stfld      int32 OriginalCB.BoundedPoint::yBound
    IL_0011: nop
    IL_0012: ret
} // end of method BoundedPoint::.ctor

```

Figure 12: ILDASM view of verifiable IL-bytecode for `BoundedPoint` constructor in assembly `OriginalCB(1).dll`.

We have added support for the generation of a rewiring specification to the open source C#-compiler of the Mono project [27, version 1.1.8.3]. At compile-time, the modified C#-compiler generates a list regarding all explicitly referenced assemblies, all explicitly imported classes, and all extended imported classes. For example, consider again the classbox `TraceAndColorCB`. The specification for building the final physical structure of this classbox is given below:

```

N # TraceAndColorCB
T # TraceAndColorCB.dll
R # OriginalCB.dll
I # Point
I # LinearBPoint

```

After reading the rewiring specification, the rewiring process proceeds in two phases. In the first phase, we identify (i) all classes, whose super type is in the set of explicitly imported classes and register these classes for update and (ii) build a list of all assemblies for which we need to create a new version. For example, in the case of the classbox `TraceAndColorCB`, we need to update the class `OriginalCB.BoundedPoint` and have to create a new version of the assembly `OriginalCB.dll`, since this assembly hosts class `OriginalCB.BoundedPoint`.

In the second phase, we perform the actual metadata transformations. First, we create the required new assembly versions. Next, we add the required new `ASSEMBLYREF` metadata tokens to their respective assemblies. Adding the new `ASSEMBLYREF` tokens first simplifies the next step, as these new `ASSEMBLYREF` tokens are required for the update of the super type information. In the final step in this phase, we update the super type information and restore the integrity of the constructors of all classes marked for update.

Both phases take place in memory. To create the actual images of the updated assemblies, we have to call the their `Write` method. Metadata must be stored in the text section (i.e., the `.text` section). The `Write` method places the new metadata at the end of the text section. If necessary, the text section is enlarged to accommodate the new metadata. It is in general not possible to reclaim the space occupied by the old metadata, as there are no requirements to place metadata at the end of the text section. However, by placing

the new metadata at the end of the text section, we can recycle the space occupied by metadata in future updates.

5. ANALYSIS

We assess now the obtained results and examine the impacts on verifiability, performance, size, and deployment of classbox assemblies.

5.1 Verifiability

Rewire.NET imprints the structure of a classbox in the metadata of its underlying assemblies. The required metadata transformations are performed in a way that guarantees verifiability of the resulting assemblies.

Consider, for example, Figure 12 that shows the IL-bytecode for the constructor of the class `BoundedPoint` in assembly `OriginalCB(1).dll`. `OriginalCB(1).dll` is a part of the physical structure of the classbox `TraceAndColorCB`, which defines a refinement for the class `Point`. The resulting new subclass is exported under the *fully qualified* name `TraceAndColorCB.Point`, which indicates the defining classbox, but preserves the original class name. Since the classbox `TraceAndColorCB` also explicitly imports the class `LinearBPoint`, we have to rewire the supertype of the class `BoundedPoint` to point to `TraceAndColorCB.Point`. This also means that the statically known supertype of the class `BoundedPoint` is now `TraceAndColorCB.Point`, which is the defined in assembly `TraceAndColorCB.dll`. This information is encoded in the new method descriptor shown in line IL_0003 in Figure 12. Since the static the initialization process honors the inheritance graph of the class `BoundedPoint`, the constructor IL-bytecode passes verification.

5.2 Performance

The goal of our work is to reduce the execution overhead of redefined methods by moving the process of building the structure of a classbox from runtime to compile-time and to recover the standard method lookup mechanism for redefined methods.

We have recorded the execution time of constructors and methods during trials on a 2.4 GHz Pentium 4 based PC with 1GB of RAM running WindowsXP SP2 and the .NET

```

ldloc.0
ldc.i4.s 50
callvirt instance void ['OriginalCB(1)']OriginalCB.Point::set_X(int32)
ldloc.0
callvirt instance int32 ['OriginalCB(1)']OriginalCB.Point::get_X()
stloc.3
ldloc.0
call valuetype [System.Drawing]System.Drawing.Color [System.Drawing]System.Drawing.Color::get_Blue()
callvirt instance void [TraceAndColorCB]ColorCB.Point::set_Color(valuetype [System.Drawing]System.Drawing.Color)
ldloc.0
callvirt instance valuetype [System.Drawing]System.Drawing.Color [TraceAndColorCB]ColorCB.Point::get_Color()

```

Figure 13: ILDASM view of IL-bytecode to access the properties X and Color.

2.0 framework. The execution time is given as the mean average taken of 50 test trails for each method invocation.

While testing the constructor calls of explicitly imported classes, we have observed a difference between the first and the succeeding constructor calls. For example, the constructor for the class `OriginalCB.LinearBPoint` in the original assembly `OriginalCB.dll` showed a execution characteristics as illustrated in Table 1. That is, there is no difference between the first and any succeeding constructor calls.

	Execution time (ms)
1	< 0.0001
n > 1	< 0.0001

Table 1: `OriginalCB.LinearBPoint::ctor`

In contrast, when testing the constructor for the refined class `TraceAndColorCB.LinearBPoint`, we observed a slight delay for the first call as shown in Table 2. The initial overhead can be attributed to the more complex structure of the class `TraceAndColorCB.LinearBPoint`. In order to create new objects of class `TraceAndColorCB.LinearBPoint`, the virtual execution system has to construct a physical layout for that class in memory. Since the metadata describing class `TraceAndColorCB.LinearBPoint` spans several assemblies, the class loading process takes longer. Once the class layout has been constructed, the execution time of the constructors for class `TraceAndColorCB.LinearBPoint` takes normal values.

	Execution time (ms)
1	0.0022
n > 1	< 0.0001

Table 2: `TraceAndColorCB.LinearBPoint::ctor`

The IL-bytecode to access the properties X and Color of an object of type `TraceAndColorCB.LinearBPoint` is shown in Figure 13. Properties are encoded as *getter* and *setter* methods. Both properties use standard method lookup for virtual methods (i.e., late binding), even though property Color is part of the class extension applied to class Point in the classbox `TraceAndColorCB`. There is no measurable difference in execution time for the getter and setter methods of both properties.

5.3 Size

While the size of the IL-bytecode remains the same, the size of the metadata grows due to the rewiring process. The amount of change underlies several varying factors. First, the metadata is not located at the end of the text section. In this case, we cannot recycle the old metadata and

therefore create a new image of metadata at the end of the text section, which effectively renders the old metadata into garbage. The second factor influencing the growth of metadata is associated with the amount of “reusable” rows. The rewiring process takes a very conservative approach, as it only adds new rows to the metadata, if no appropriate row exists. All byte-indexed data (i.e., strings, blob data, and UTF-16 strings) cannot be reused, as this may break indices from IL-bytecode into the corresponding heaps. When a new row is needed, then this row is always added to the end of its corresponding table or heap.

The Size of both the metadata and the text section before and after applying the rewiring process to `OriginalCB.dll` and `TraceAndColorCB.dll` is shown in Table 3. Even though we only add 128 bytes to the metadata of `OriginalCB.dll`, the size of its text section doubles. The reason for this is that the size of new metadata exceeds the available free space at the end of the text section. Therefore, we are required to enlarge it by one unit of size *SectionAlignment*, which is 4K.

Assembly	old	new	Size of .text
<code>OriginalCB.dll</code>	1,864	1,992	4,096/8,192
<code>TraceAndColorCB.dll</code>	1,820	1,832	4,096/4,096

Table 3: Size of metadata in bytes.

Table 4 shows the size and location of metadata in selected assemblies (assemblies marked with * have been altered by `Rewire.NET`). The numbers for `System.Drawing.dll` and `System.Windows.Forms.dll` indicate that the overhead for placing the metadata at the end of the text section may reach a threshold at which it cannot be ignored anymore. Furthermore, the size of the metadata amounts to almost half of the total size of these assemblies. We plan, therefore, to explore alternative approaches in future work that will allow us to reorder the text section data, so that the space occupied by the old metadata can be reclaimed.

Assembly	MD size in bytes	located
<code>OriginalCB.dll</code>	1,864	middle
<code>OriginalCB(1).dll*</code>	1,992	end
<code>TraceAndColorCB.dll</code>	1,820	middle
<code>TraceAndColorCB.dll*</code>	1,832	end
<code>CLI.dll</code>	71,356	middle
<code>Rewire.exe</code>	5,752	middle
<code>mcs.exe</code>	302,128	middle
<code>System.Drawing.dll</code>	334,468	middle
<code>System.Windows.Forms.dll</code>	2,445,112	middle

Table 4: Size and location of metadata.

5.4 Deployment

The re-wiring process requires the originating assemblies to be copied. This appears to be a drawback of our implementation. However, the new versions of these assemblies play a major role in a compile-time-based approach to integrate extensions into an existing class hierarchy. The .NET framework uses a strong version control mechanism as each assembly is assigned a unique version number. In our implementation, we utilize this mechanism to distinguish between different classboxes. An extension to an imported class triggers the creation of new versions of referenced assemblies that contain types the imported class is depending upon. These new assemblies are bound to a particular classbox. The result is a physical and logical structure that captures precisely the defined classbox and does not affect previously defined classboxes. As a consequence, this structure can be deployed independently.

On the other hand, the Smalltalk and Java implementations of classboxes use an approach in which extensions to classes are linked into the class hierarchy at runtime. In this approach, different classboxes are merged into one deployment system (e.g., a Smalltalk image or extended Java package). At present, there is no direct support for independent deployment in neither the Smalltalk implementation of classboxes nor Classbox/J – an independent deployment of different classboxes would require additional user intervention.

6. RELATED WORK

Code instrumentation has been a subject of intense research in the last decade. Code instrumentation focuses on three primary purposes: introspection, optimization, and security. By using code instrumentation we can, for example, detect any places in compiled code, where this code accesses the local file system and insert an additional authentication layer.

To edit *fully-linked* executables, Larus and Schnarr [17] have proposed the *Executable Editing Library* (EEL). EEL is a framework for building tools to analyze and modify executable (i.e., compiled) code. EEL provides an object-oriented architecture- and system-independent set of abstractions (i.e., C++ class hierarchies) to read, analyze, and modify executable code. These abstractions are very similar to those found in a compiler, as the purpose of both EEL and a compiler is to manipulate programs.

Code instrumentation frameworks that target the Java platform are *Binary Component Adaptation* (BCA) [13] and *Javassist* [7], which allow for an *on-the-fly* code instrumentation of binary Java components. Both frameworks use a customizable class loader to rewrite and/or reflect on binary components before (or while) they are loaded. The rewriting process does not require source code access and guarantees release-to-release compatibility.

RAIL [6] is the first general purpose code instrumentation library for the .NET platform. RAIL supports structural [10] as well as behavioral reflection [20]. The abstractions provided by RAIL allow for both low- and high-level modifications of assemblies. RAIL enables the modification of assemblies at class level (e.g., substitution of classes, members, and member access). RAIL does not, however, allow

for the manipulation of references to external types.

Aspect-oriented programming (AOP) [15] has gained growing attention as being an approach that acknowledges the fact that certain system concerns *cross-cut* multiple components. The main idea of AOP is to encapsulate cross-cutting concerns (i.e., *aspects*) as first-class entities that are independently developed and then “weaved” with the domain components to produce complete systems [21]. As such, AOP can be seen as an approach that allows the modification of an existing class without directly editing the class itself.

Lafferty and Cahill [16] have presented Weave.NET, a load-time weaver for the .NET framework that allows aspects and components written in different languages to be freely intermixed. Weave.NET relies on the Common Language Infrastructure and XML to specify aspect bindings. By using CLI, Weave.NET provides a language-independent aspect-oriented programming model.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented an approach to seamlessly incorporate the classbox concept into the .NET framework. Classboxes provide a feasible solution to the problem of controlling the visibility of change in object-oriented systems without breaking existing applications, as they allow for strictly limiting both the scope and the impact of any modifications. Consequently, classboxes can significantly reduce the risk for introducing design and implementation anomalies due to the need to adapt a software system to changing requirements [3].

We replaced the dynamic integration of class extensions at runtime by a static, *compile-time*-based approach. Our approach not only eliminates the runtime overhead that is associated with the construction of the classbox structure, but allows us also to treat standard .NET assemblies as classboxes. The key method underlying the integration of the classbox concept in the .NET framework is *metadata manipulation*. Using this code instrumentation method we can restructure the inheritance graph of a class hierarchy in order to incorporate local refinements (i.e., class extensions) into the behavior of explicitly imported classes. Hence, by using the metadata concept of the underlying Common Language Infrastructure (CLI), classboxes can be seamlessly integrated into the .NET environment without the need to modify the underlying runtime infrastructure.

In order to explicitly separate the local refinement from the import of a class, Bergel and Ducasse have recently proposed to incorporate *traits* into the classbox model [1]. Traits are mixin-like abstractions that allow for the explicit extension of behavior to existing classes, but not new state [25]. Hence, in future work, we plan to investigate how to incorporate traits in our .NET implementation of classboxes also.

To enhance the separation of concerns in the classbox model, we also plan to decouple behavior and state by introducing *state-only* abstractions as first-class entities into the model. This would allow us to *independently* extend existing classes with either state or behavior, respectively. As a consequence, we expect that Scala-style class composition [24] could be seamlessly integrated into the classbox model.

In this work, we have used a rather conservative approach to manipulate metadata. However, metadata transformation allow for a variety of manipulations of the structure of classes. We plan, therefore, to explore more aggressive class restructuring techniques in the future in order to enrich the classbox concept.

Acknowledgements. We would like Alexandre Bergel, Andre Lokasari, Hua Ming, and Jean-Guy Schneider for their valuable comments and discussions.

8. REFERENCES

- [1] A. Bergel and S. Ducasse. Supporting Unanticipated Changes with Traits and Classboxes. In *Proceedings of Net.ObjectDays (NODE'05)*, pages 61–75, Erfurt, Germany, Sept. 2005.
- [2] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings OOPSLA '05*, volume 40 of *ACM SIGPLAN Notices*, pages 177–189, San Diego, USA, Oct. 2005.
- [3] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems & Structures*, 31(3–4):107–126, May 2005.
- [4] G. Bracha and W. Cook. Mixin-based Inheritance. In N. Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, Oct. 1990.
- [5] G. Bracha and G. Lindstrom. Modularity Meets Inheritance. In *Proceedings of the International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, Apr. 1992.
- [6] B. Cabral, P. Marques, and L. Silva. RAIL: Code Instrumentation for .NET. In L. M. Liebrock, editor, *Proceedings of Symposium On Applied Computing (SAC'05)*, pages 1282–1287. ACM Press, Mar. 2005.
- [7] S. Chiba. Load-Time Structural Reflection in Java. In E. Bertino, editor, *Proceedings ECOOP 2000*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer.
- [8] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings OOPSLA 2000*, volume 35 of *ACM SIGPLAN Notices*, pages 130–146, Oct. 2000.
- [9] European Computer Machinery Association. *Standard ECMA-334: C# Language Specification*, third edition, June 2005.
- [10] J. Ferber. Computational Reflection in Class based Object-Oriented Languages. In *Proceedings OOPSLA '89*, pages 317–326. ACM Press, Oct. 1989.
- [11] R. B. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34, pages 94–104, 1998.
- [12] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Sept. 1989.
- [13] R. Keller and U. Hölzle. Binary Component Adaptation. In E. Jul, editor, *Proceedings ECOOP'98*, LNCS 1445, pages 307–329, Brussels, Belgium, July 1998. Springer.
- [14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 327–355, Budapest, Hungary, June 2001. Springer.
- [15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, LNCS 1241, pages 220–242. Springer, June 1997.
- [16] D. Lafferty and V. Cahill. Language-Independent Aspect-Oriented Programming. In *Proceedings OOPSLA 2003*, pages 1–12. ACM Press, Oct. 2003.
- [17] J. R. L. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, California, June 1995.
- [18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Sept. 1996.
- [19] M. Lumpe and J.-G. Schneider. On the Integration of Classboxes into C#. In W. Löwe and M. Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, Vienna, Austria, Mar. 2006. Springer.
- [20] J. Malenfant, C. Dony, and P. Cointe. Behavioral Reflection in a Prototype-Based Language. In A. Yonezawa and B. Smith, editors, *Proceedings of International Workshop on Reflection and Meta-Level Architectures*, pages 143–153, Tokyo, Japan, Nov. 1992.
- [21] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings OOPSLA '98*, volume 33 of *ACM SIGPLAN Notices*, pages 97–116. ACM Press, Oct. 1998.
- [22] Microsoft Corporation. *Metadata Unmanaged API*, 2002.
- [23] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development Series. Addison-Wesley, 2003.
- [24] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Scinz, E. Stenmann, and M. Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences, 2004.

- [25] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 248–274. Springer, July 2003.
- [26] G. L. Steele. *Common Lisp the Language*. Digital Press, Thinking Machines, Inc., 2nd edition, 1990.
- [27] The Mono Project.
http://www.mono-project.com/Main_Page.