Using Metadata Transformations to Integrate Class Extensions in an Existing Class Hierarchy

Markus Lumpe

Department of Computer Science
Iowa State University
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 implementation technique that allows for the structure of a classbox to be constructed at compile-time by means of metadata transformations to rewire the inheritance graph of refined classes. These metadata transformations are language-neutral and more importantly preserve both the semantics of the classbox concept and the integrity of the underlying deployment units. As a result, metadata transformation provides a feasible approach to incorporate the classbox concept into programming environments that use a *virtual execution system*.

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 [10], CLOS [22], MultiJava [6], Scala [21], or AspectJ [13]) 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 kinds[1] 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 [2]. Moreover, even if a system allows for a modular specification of class extensions (e.g., MultiJava [6] or AspectJ [13]), it may not support multiple versions of a given class to coexist at the same time. To remedy these shortcomings, Bergel et al. [1, 2] have recently proposed *classboxes*, a new module system that defines a packaging and scoping mechanism for controlling the visibility of isolated

[1] Bracha and Lindstrom [3] have also presented a *hide* operator that renders a method of a class invisible to clients of that class.

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 [2].

At present, there exist two implementations of classboxes in Smalltalk [2] and a restricted prototype in Java [1]. 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 [1], 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 to the "normal" method lookup [2]. Similarly, the method lookup of redefined methods in Classbox/J is on average 22 times slower than the normal method lookup [1].

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* mechansism 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 form 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* [4, 5, 12, 14, 15] 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 [16] and the Common Language Infrastructure (CLI) [20]. 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 provide a brief overlook of related work in Section 5. We conclude this paper in Section 6 with a summary of the presented work and outline future activities in this area.

2 Integration of the Classbox Model in the .NET Framework

2.1 Classbox Characteristics

The main characteristics of classboxes can be summarized as follows [2]:

- 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.

There are four additional, yet critical aspects in the definition of the classbox semantics [1, 2] that need to be satisfied also, 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 [2].

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.

Virtual methods. The classbox concept rest upon virtual methods and dynamic binding [1, 2]. There are no provisions for non-virtual methods. In addition, 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.

2.2 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 [1, 2]. 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 1 in which we highlight the search for the method foo with respect to the class C. The lookup starts at point '1' and as class C neither implements nor has been extended with a corresponding method, the lookup continues in its superclass B (denoted by '2'), which occurs as an implicitly imported class in SampleClassbox. Again, the class B does not implement the foo method. Therefore, the search has to continue by inspecting its superclass. However, since we have defined an extension to class A (we use the rounded box as a graphical means to indicate that the class A has been extended with the method foo), the search terminates in the extension that defines the method foo (denoted by '3') rather than in the class A directly, as this is the first point along the search path that implements the method foo.

The reader should note that this special method lookup mechanism is required, because the structure of a classbox is not known until runtime in both the Smalltalk and

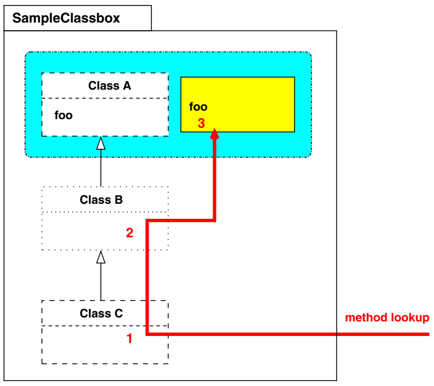

Fig. 1. Method lookup as search over the classbox graph

Java implementations. Moreover, even though extensions are bound dynamically into a class hierarchy, the classbox concept neither supports virtual classes [11] nor any form of "chameleon" objects that can change their structure based on the environment in which they are currently being used. Objects are instantiated with respect to a provider classbox that determines and finalizes the capabilities of that object. The dynamic graph search does not supersede the method layout, but amends it to build the structure of a given object's provider classbox at runtime.

2.3 Classbox-Aware C#

To ally the classbox concept with the .NET framework, we define a "classbox-aware" dialect of C#[2]. In previous work, we already explored a technique to amend the C# language with the classbox concept [17]. However, even though we were able to define a conceptual approach for the integration of the classbox concept in the .NET framework, the resulting language extensions could not be properly type-checked. Furthermore, the use of the *Metadata Unmanaged API* [19] turned out to be unsuitable for the purpose of manipulating .NET assemblies, as this API does not provide access to IL-bytecode, which is essential for a comprehensive solution. The language model proposed in this work not only follows closely the one proposed by Bergel et. al [2], but also allows for a proper type checking of the specified class extensions:

Class Import. To explicitly import a class, we use the *alias* form of the C# `using`-directive [7, §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

[2] We are currently experimenting with the open-source Mono compiler in order to define a frontend for classbox-aware C# [17].

```
using System;

namespace TraceAndColorCB
{
  using System.Drawing;

  using Point = PointHierarchyCB.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 = PointHierarchyCB.LinearBPoint;
}
```

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

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 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* [7, §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.

A classbox in the .NET framework has a *logical* and a *physical* structure. 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 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 the classes `Point` and `LinearBPoint`, both originating from classbox `PointHierarchyCB`. In `TraceAndColorCB`, we extend class `Point` with the

```
using System;

namespace TraceAndColorCB
{
  using System.Drawing;

  public class Point : PointHierarchyCB.Point
  {
    private Color color;
    public Point( int ix, int iy ) : base( ix, iy ) {}
    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 : PointHierarchyCB.LinearBPoint
  {
    public LinearBPoint( int ix, int iy, int ibound ) : base( ix, iy, ibound ) {}
  }
}
```

Listing 2. Classbox `TraceAndColorCB` in standard C#

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`, which defines a non-constant linear upper bound for point objects, is an indirect subclass of class `Point` (i.e., in `PointHierarchyCB` the class `LinearBPoint` is derived from `BoundedPoint` that is a direct subclass of `Point`). 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 in `TraceAndColorCB`.

The classbox-aware C#-compiler translates the specification of this classbox 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 `PointHierarchyCB.dll` (i.e., the assembly defining the classbox `PointHierarchyCB`) 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 Building the Structure of a Classbox at Compile-Time

3.1 Metadata Type Declarations

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 is a set of 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 to seamlessly integrate them in the CLI [20]. The CLI uses an implementation-dependent declarative encoding mechanism to represent metadata information, called *metadata token*. A metadata token is a scoped typed identifier of a metadata object and is represented as a ***read-only*** index into a corresponding metadata table.

New types are introduced via metadata type declarations [20]. `TYPEDEF` tokens encode the name of a type, its declaration namespace, the super type (index into `TYPEDEF` or `TYPEREF` 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 `TYPEREF`, `MEMBERREF`, and `ASSEMBLYREF` tokens, respectively. A `TYPEREF` 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.

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. 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.

3.2 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 Section 2.3. However, subclassing is an operation that breaks the connection to former clients [9]. 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.

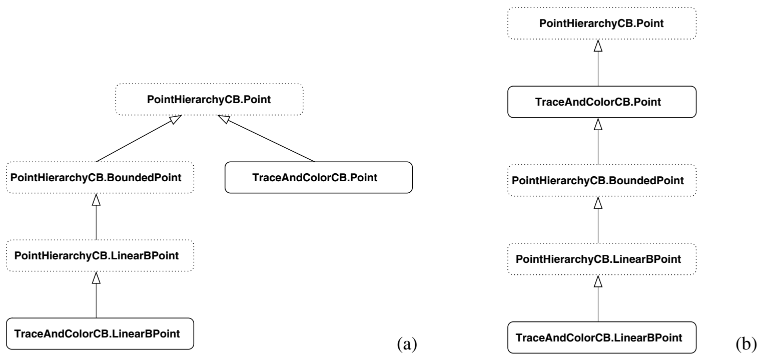
PointHierarchyCB.Point
PointHierarchyCB.BoundedPoint
TraceAndColorCB.Point
PointHierarchyCB.LinearBPoint
TraceAndColorCB.LinearBPoint
(a)
PointHierarchyCB.Point
TraceAndColorCB.Point
PointHierarchyCB.BoundedPoint
PointHierarchyCB.LinearBPoint
TraceAndColorCB.LinearBPoint
(b)

Fig. 2. Inheritance graph in classbox `TraceAndColorCB` before and after flattening

Consider again the classbox `TraceAndColorCB`. This classbox explicitly imports the classes `Point` and `LinearBPoint` from `PointHierarchyCB`. As a result, we create two new subclasses with the same name in `TraceAndColorCB`. The resulting inheritance graph is shown in Figure 2(a) (explicitly imported classes are marked with a solid rounded box, whereas implicitly imported types are marked with a dotted rounded box).

A name of a type in CLI consists of two elements: a *typename* 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 `PointHierarchyCB.BoundedPoint`. To accomplish this, we change the `TYPEDEF` metadata token defining the class `PointHierarchyCB.BoundedPoint` in the metadata of the assembly `PointHierarchyCB.dll`. More precisely, we need rewire the *Extends* column of `PointHierarchyCB.BoundedPoint`'s `TYPEDEF` metadata token to point to the `TYPEDEF` metadata token defining class `TraceAndColor CB.Point` in assembly `TraceAndColorCB.dll`. We proceed by performing the following instructions:

1. Create a new version of `PointHierarchyCB.dll` and name this assembly `PointHierarchyCB(TraceAndColorCB).dll`, where the name `TraceAndColorCB` firmly associates this new assembly with the classbox `TraceAndColorCB` to *disambiguate* multiple rewired versions of the `PointHierarchyCB` classbox.
2. Add an `ASSEMBLYREF` token for `TraceAndColorCB` to the metadata of `PointHierarchyCB(TraceAndColorCB).dll`.

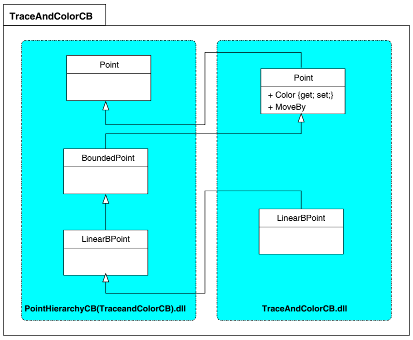

Fig. 3. Structure of classbox `TraceAndColorCB`

3. Add a `TYPEREF` token for `TraceAndColorCB.Point` to the metadata of `PointHierarchyCB(TraceAndColorCB).dll`.
4. Set the *Extends* column of the `TYPEDEF` token for class `PointHierarchyCB.BoundedPoint` to point to the newly added `TYPEREF` token in `PointHierarchyCB(TraceAndColorCB).dll`.

The result of this transformation is a *flattened* classbox that publishes two classes: `Point` and `LinearBPoint`, whose inheritance graph is shown in Figure 2(b). The metadata manipulations do not affect existing clients of `PointHierarchyCB`, 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 `PointHierarchyCB(TraceAndColorCB).dll`, as shown in Figure 3.

3.3 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. Unfortunately, changing the *Extends* column of the `TYPEDEF` token describing class `BoundedPoint` does not preserve the integrity of the IL-bytecode in `PointHierarchyCB(TraceAndColorCB).dll`.

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 `PointHierarchyCB.dll`, this statically known superclass constructor is `PointHierarchyCB.Point::.ctor`. The situation in the assembly `PointHierarchyCB(TraceAndColorCB).dll` is different, however, as we have changed the supertype of the class `BoundedPoint` to `TraceAndColorCB.Point`. It is, therefore, not correct to call `PointHierarchyCB.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 `MEMBEREF`) 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 `PointHierarchyCB(TraceAndColorCB).dll`, as follows:

1. Add a `MEMBERREF` token indicating the constructor for the class `TraceAndColorCB.Point` to the metadata of `PointHierarchyCB(TraceAndColorCB).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 `PointHierarchyCB.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 `PointHierarchyCB(TraceAndColorCB).dll` is restored. As a result, we have obtained the final physical structure of the classbox `TraceAndColorCB`. The assemblies `PointHierarchyCB(TraceAndColorCB).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. Moreover, by moving the process of building the structure of a classbox from runtime to compile-time we recover the standard method lookup mechanism for redefined methods and therefore, eliminate the execution overhead formerly associated with class extensions.

3.4 Evaluation of the Rewiring Technique

A major benefit of our solution is that we can use the standard method lookup mechanism for redefined methods. As a result, there is no measurable difference in the execution time of both plain and redefined methods.

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 the 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.

To illustrate the the change in size, consider, for example, the rewiring process of `PointHierarchyCB.dll`. The required transformations require 168 additional bytes of metadata. Unfortunately, the resulting size of the 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. However, the numbers for the two system assemblies `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. In these two assemblies, the size of the metadata amounts to almost half of their total size. 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.

One of the key features of the classbox concept is that multiple versions of a class can coexist in the same classbox or application. Our rewiring technique preserves this property of classboxes by adding a *target classbox tag* to the originating namespace names of all explicitly imported types[3]. For example, the namespace name `PointHie-rarchyCB` in classbox `TraceAndColorCB` is changed to `TraceAndColorCB: PointHierarchyCB`, an identifier that cannot be defined in C#. The effect of this tag is twofold. First, in C# the visibility of a superclass cannot be more restrictive than the one of any of its subclasses. As a consequence, even implicitly imported types possess public visibility in a provider classbox. The target classbox tag eliminates this problem completely, as it renders all implicit imported types invisible. Secondly, the target classbox tag *disambiguates* multiple versions of the same class. For example, a client can safely use both classboxes `PointHierarchyCB` and `TraceAndColorCB`, even though all provided classes occur multiple times either explicitly imported, implicitly imported or both in the client space. Therefore, different versions of a class can coexist and be unequivocally identified in the same declaration space.

4 Rewire.NET

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 (cf., Figure 4).

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

[3] We have omitted these tags in the above explanation of the rewiring technique to preserve readability.

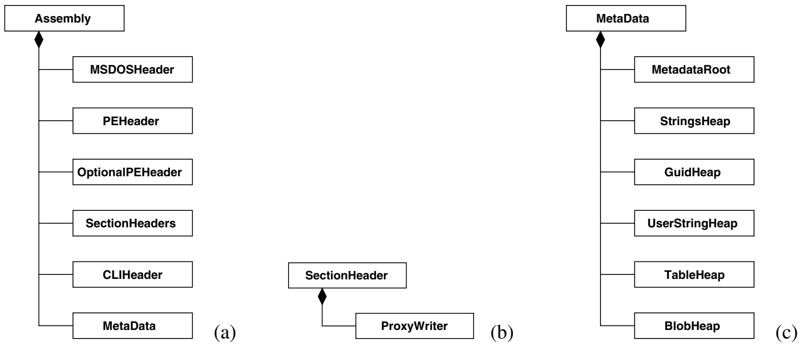

Fig. 4. `CLI.Assembly`, `CLI.SectionHeader`, and `CLI.MetaData`

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 3.3, 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* [19] 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) [4]. 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., `TYPEREF` 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.

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 4(a). 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 `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 data (cf., Figure 4(b)). 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.

The class MetaData, as shown in Figure 4(c), 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 below:

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

We have added support for the generation of a rewiring specification to the open source C#-compiler of the Mono project [23, 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 # PointHierarchyCB.dll
I # PointHierarchyCB.Point
I # PointHierarchyCB.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, (ii) build a list of all assemblies for which we need to create a new version, and (iii) add the required target classbox tags. For example, in the case of the classbox TraceAndColorCB, we need to update the class BoundedPoint originating from PointHierarchyCB, have to create a new version of the assembly PointHierarchyCB.dll, and add the classbox tag TraceAndColorCB: to the namespace name PointHierarchyCB. 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 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 [15] 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) [12] and *Javassist* [5], 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 [4] is the first general purpose code instrumentation library for the .NET platform. RAIL supports structural [8] as well as behavioral reflection [18]. 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.

Lafferty and Cahill [14] 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.

6 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 [2].

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.

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 a 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 the captures precisely the defined classbox and does not affect previously defined classboxes. As a consequence, this structure can be deployed independently.

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. In addition, we plan to apply the rewiring technique to Classbox/J. However, since Java platform uses a different deployment mechanism (usually based on JAR-files) that lacks a strong association between deployment unit, version, and package name, the physical structure of a classbox cannot span across multiple physical units as in the .NET framework. Future work on the classbox concept will include, therefore, the exploration of an alternative packaging mechanism to represent the physical structure of a classbox in which the classes of a classbox are grouped in one physical deployment unit.

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

References

1. Alexandre Bergel, Stéphane Ducasse, and Oscar 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, October 2005.
2. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems & Structures*, 31(3–4):107–126, May 2005.

3. Gilad Bracha and Gary Lindstrom. Modularity Meets Inheritance. In *Proceedings of the International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, April 1992.
4. Bruno Cabral, Paulo Marques, and Luís Silva. RAIL: Code Instrumentation for .NET. In Lorie M. Liebrock, editor, *Proccedings of Symposium On Applied Computing (SAC'05)*, pages 1282–1287. ACM Press, March 2005.
5. Shigeru Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *Proceedings ECOOP 2000*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer.
6. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings OOPSLA 2000*, volume 35 of *ACM SIGPLAN Notices*, pages 130–146, October 2000.
7. European Computer Machinery Association. *Standard ECMA-334: C# Language Specification*, third edition, June 2005.
8. Jacques Ferber. Computational Reflection in Class based Object-Oriented Languages. In *Proceedings OOPSLA '89*, pages 317–326. ACM Press, October 1989.
9. Robert Bruce Findler and Matthew 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.
10. Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, September 1989.
11. Atsushi Igarashi and Benjamin Pierce. Foundations for Virtual Types. In Rachid Guerraoui, editor, *Proceedings ECOOP '99*, LNCS 1628, pages 161–185. Springer, June 1999.
12. Ralph Keller and Urs Hölzle. Binary Component Adaptation. In Eric Jul, editor, *Proceedings ECOOP'98*, LNCS 1445, pages 307–329, Brussels, Belgium, July 1998. Springer.
13. Grégor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 327–355, Budapest, Hungary, June 2001. Springer.
14. Donal Lafferty and Vinny Cahill. Language-Independent Aspect-Oriented Programming. In *Proceedings OOPSLA 2003*, pages 1–12. ACM Press, October 2003.
15. James R. Larus Larus and Eric 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.
16. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, September 1996.
17. Markus Lumpe and Jean-Guy Schneider. On the Integration of Classboxes into C#. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, LNCS 4089, pages 307–322, Vienna, Austria, March 2006. Springer.
18. Jacques Malenfant, Christophe Dony, and Pierre 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, November 1992.
19. Microsoft Corporation. *Metadata Unmanaged API*, 2002.
20. James S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development Series. Addison-Wesley, 2003.
21. Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Scinz, Erik Stenmanm, and Matthias 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.
22. Guy L. Steele. *Common Lisp the Language*. Digital Press, Thinking Machines, Inc., 2nd edition, 1990.
23. The Mono Project. http://www.mono-project.com/Main_Page.