Flexible Dynamic Linking for .NET

Anders Aaltonen, Alex Buckley, Susan Eisenbach
a.buckley@imperial.ac.uk
Imperial College London

Abstract

A .NET application is a set of assemblies developed or reused by programmers, and tested together for correctness and performance. Each assembly's references to other assemblies are type-checked at compile-time and embedded into the executable image, from where they guide the dynamic linking process.

We propose that an application can potentially consist of multiple sets of assemblies, all known to the application's programmers. Each set implements the application's functionality in some special way, e.g. using only patent-free algorithms or being optimised for 64-bit processors. Depending on the assemblies available on a user's machine, the dynamic linking process will select a suitable set and load assemblies from it.

We describe how, in our scheme, an application is written to use a *default* set of assemblies but carries nominal and structural specifications about permissible sets of *alternative* assemblies. We implement the scheme on Rotor, a .NET virtual machine, by modifying its linking infrastructure to efficiently find assemblies on the user's machine that satisfy the application's specifications. Specifications can be applied to individual classes and methods, so that only code wishing to use alternative assemblies has to undergo the modified linking process.

1 Introduction

1.1 Dynamic linking in .NET

Modern virtual machines, like the Common Language Runtime in .NET, support dynamic linking of bytecode obtained from local and remote sites. The key concept in .NET linking is the *assembly*. An assembly is a file that contains classes' bytecode and serves as a versioned, tamper-proof unit of deployment. To guide linking, an assembly has metadata that describes its classes' dependencies on other assemblies and *their* classes. Source languages typically disallow a programmer from specifying which assembly is to provide which class; the choice is made by the compiler.

.NET Technologies 2006

So, for a method call in C♯:

```
System.Console.WriteLine(‘‘Hello’’)
```

the compiler will choose an assembly in the compile-time environment that contains the `System.Console` class, e.g. `mscorlib` 1.0.5. The compiler embeds the name and version of the chosen assembly into the metadata of the assembly being built, and emits bytecode that references (in `[...]`) the chosen assembly's name:

```
ldstr ‘‘Hello’’
call void [mscorlib]System.Console::WriteLine(string)
```

.NET will (try to) link exactly the version of the `mscorlib` assembly specified in the executing assembly's metadata. This helps to avoid "DLL hell" [6], because the user's machine can have multiple versions of an assembly installed, *e.g.* `mscorlib` 1.0.5 for application A and `mscorlib` 1.1.0 for application B, and both application's dependencies can be satisfied.

The problem is that while an assembly specified in the metadata of an executing assembly *was* available at compile-time on the programmer's machine, the user's

machine may *now* have alternative assemblies available at run-time. Reasons why assemblies at run-time may differ from those at compile-time include:

- .NET's standard libraries provide interfaces for well-understood features like XML processing, database access and networking, so it is straightforward for multiple vendors to provide different implementations of the interfaces. Each vendor's assembly is likely to have a different name.
- Within a company's IT department, developers often have different implementations of a business interface that version numbering alone cannot reasonably differentiate. For example, two assemblies that contain different implementation classes for a bond trading strategy may well be signed by different keypairs and have different versioning conventions; and thus different names.

Unfortunately, linking in .NET cannot cope when assemblies in the run-time environment have different names to those in the compile-time environment. The .NET assembly loader can only redirect a request for one *version* of an assembly to another version of the same assembly; it cannot redirect the *name* of an assembly. Thus, a programmer who wishes to make his application portable between differently-named assemblies (that are expected to contain implementation classes for popular interfaces) must code portability by hand. Typically, a Factory pattern or an inversion-of-control container [9] is used to abstract class names from the main application code, but some reflective, type-unsafe code is always needed to discover assemblies and extract implementing classes.

1.2 Flexible dynamic linking in .NET

We propose a more declarative approach to portability. An application programmer merely enumerates assemblies and classes that his application can use (*e.g.* that provide implementation classes of useful interfaces), and the dynamic linker finds and instantiates them as available. Thus, we reduce an assembly's dependence on a particular assembly (known to a compiler) by adding *potential* dependencies that increase the range of valid run-time environments.

In our scheme, *any* assembly or class name that appears in bytecode can be redirected. The programmer writes code as usual that references classes, but includes *nominal specifications* along the lines of "try assemblies `B` and `C` as well as `A`" and "try classes `P` and `Q` as well as `R`". After a compiler has generated an assembly from the programmer's code, we have a tool that, for the purpose of type-safety, takes the assembly and adds *structural specifications* based on its nominal specifications and the classes and members that it references. For example, given the nominal specifications above, the generated structural specifications would be along the lines that "any assembly used in place of assembly `A` must provide class `D`" and "any class used in place of `D` must provide a field `f` of type `E`".

Our modified dynamic linker inspects an assembly's nominal and structural specifications at run-time.[1] If an assembly name referenced in bytecode is not available, then the linker searches for substitute assemblies given in the nominal specifications; any found assembly must satisfy the structural specifications. Then, when an assembly's classloader searches for classes, it considers the nominal and structural specifications for classes.

Even with structural specifications providing type-safety, it is unlikely that an assembly exists at run-time with the "right" classes (with the "right" members) *unless the programmer knew about it in advance*. This is because only the programmer can ensure that the assemblies and classes named in nominal specifications are semantically compatible with (*i.e.* exhibit the same observable behaviour as) assemblies and classes known to the compiler. Thus, our policy is that only assemblies directly referenced in bytecode or enumerated in nominal specifications should be linked. To prevent third parties making or modifying specifications, specifications are embedded in an assembly's metadata rather than being expressed in standalone resource files that anyone could edit.[2]

1.3 Related work

The use of type variables for abstracting over data types is well-known in the functional [12] and object-oriented [10] worlds. For example, rather than having bytecode refer to specific classes, introducing type variables at bytecode at compile-time can provide true sep-

[1] Strictly speaking, the linker works at JIT-time, loading assemblies in support of member resolution. But we consider JIT-time as "run-time", because after bytecode is JIT-compiled, it is extremely difficult to inspect which assemblies and classes it uses.

[2] We assume that assemblies will routinely be strongly-named, thus making them tamper-proof. This is analogous to how a publisher policy is a strongly-named assembly that contains an XML file, rather than a standalone XML file.

arate compilation for object-oriented languages [1]. In [3], we also advocated inserting type variables into bytecode at compile-time and substituting them to available assembly and class names at run-time. We modified the .NET dynamic linker to recognise type variables, but the end-user could specify substitute assemblies and classes without any guarantee that their substitutions were type-safe, so clearly the system was not realistic.

Most work on assemblies in .NET concerns a coherent relationship between executing assemblies and installed assemblies. [7] describes a management tool that can, by respecting a model of binary compatibility, configure a program to safely use a different version of an assembly. *Type forwarders* [11] are a feature in .NET 2.0 that allow a class to be moved from one assembly to another without breaking programs that reference the class in its original assembly. Metadata is added to the class's old assembly specifying a new assembly, and Fusion silently redirects all requests for the old assembly to the new assembly. The feature is needed by framework maintainers because, as noted earlier, Fusion cannot redirect assembly names nor does it deal with classes. Type forwarders' redirections are one-to-one and unknown to the programmer, whereas our redirections are one-to-many and intended for programmers and deployers.

As the scope of link-time activity grows, describing the behaviour of dynamic linking gains importance. Dynamic linking for Java was formalised [8, 4] because of its perceived complexity. In fact, Java's linking for unversioned, unsigned classes is considerably simpler than .NET's linking for versioned, signed assemblies, and [2] describes the assembly resolution and loading process for various .NET implementations. [5] provides a simple framework for linking in both the Java Virtual Machine and .NET.

1.4 Structure of this paper

§2 describes the high-level features available to a programmer in our system for making code more flexible with respect to its execution environment. §3 describes the architecture of a dynamic linker capable of choosing assemblies and classes at run-time, and explains a key abstraction, the *LinkContext*. §4 describes our extensions to the dynamic linker of "Rotor", the shared-source version of Microsoft's .NET Framework.

2 Design

2.1 Specifying flexible linking

To let a programmer specify alternative assembly and class names, we define two classes of custom attribute. Custom attributes are a mechanism in .NET for specifying non-functional program properties in a language-independent way. They are attached to source language constructs, such as classes and methods in an object-oriented language, and have a canonical representation in bytecode.

Our custom attributes are `[LinkAssembly]` and `[LinkClass]`; we call them *linking attributes*. In fig. 1, we assume that class `C` uses GUI classes - specifically `System.Windows.Forms` - supplied with .NET on Windows. To help `C` run on a .NET implementation on MacOS or Linux, where `System.Windows.Forms` *may* exist but where alternative assemblies providing GUI classes *may* be available, we attach linking attributes to specify both the alternative assemblies and their classes.

```
[LinkAssembly(``System.Windows.Forms'',
              ``cocoa'', ``1.3.*'',
              ``macos'', LOCAL_INTERFACE)]
[LinkAssembly(``System.Windows.Forms'',
              ``qt'', ``*'',
              ``linux'', LOCAL_INTERFACE)]
[LinkClass(``System.Windows.Forms.Button'',
           ``GelButton'',
           ``macos'')]
[LinkClass(``System.Windows.Forms.Button'',
           ``qButton'',
           ``linux'')]
class C { ... }
```

Fig. 1: Preparing code for flexible linking

Attributes are attached to the `C` class to specify that any reference in `C`'s bytecode to the assembly `System.Windows.Forms` can be redirected by the dynamic linker to either 1) an assembly `cocoa` of version 1.3.x that the programmer expects to be available on MacOS, or 2) an assembly `qt` of any version ("*") that the programmer likes to use on Linux.

If either of these redirections happens, then class names used in `C` will be redirected by the dynamic linker. The `GelButton` class will be used in preference to `System.Windows.Forms.Button` if the linker chose assembly `cocoa`, while `qButton` will be used if the linker chose assembly `qt`.

We say that `C`'s bytecode is *subject to flexible linking* since it is in the scope of at least one `[LinkAssembly]` attribute. Our modified .NET dynamic linker will recog-

nise where code is subject to flexible linking, while an unmodified linker will ignore any linking attributes and simply link bytecode to the types embedded in metadata by the compiler.

2.2 Semantic interfaces

If the code above happens to run on a Linux machine, then the likelihood is that only the `qt` assembly and not MacOS' `cocoa` assembly would be found. It therefore makes sense to only look for Linux-specific assemblies after finding `qt`. To capture the fact that different `[LinkAssembly]` attributes are likely related by platform, vendor or maturity (*e.g.* alpha, beta, production), the penultimate parameter of `[LinkAssembly]` is a *semantic interface name* that characterises the relation. If a `[LinkAssembly]` attribute specifies an assembly name that is actually used by the dynamic linker, then we support multiple policies for which linking attributes to consider in future. The policy is determined by the final parameter of the successful `[LinkAssembly]`, which is called its *semantic interface qualifier* and can take one of the following values:

LOCAL_INTERFACE If an assembly has already been chosen based on a `[LinkAssembly]` with this semantic interface qualifier, then all further assembly and class resolutions in the same scope must use linking attributes with the same semantic interface name as that `[LinkAssembly]`.

LOCAL_INTERFACE_PREFERRED If an assembly has already been chosen based on a `[LinkAssembly]` with this semantic interface qualifier, then `[LinkAssembly]` attributes with the same semantic interface name are checked first when resolving other assemblies and classes in the same scope. If none of these attributes successfully specify a loadable assembly, then other `[LinkAssembly]` attributes are tried.

LOCAL_INTERFACE_EAGER A `[LinkAssembly]` attribute with this semantic interface qualifier is "eager" in the sense that all `[LinkAssembly]` attributes in the same scope with the same semantic interface name must be successfully resolved immediately.

ANY_INTERFACE No restriction on later resolutions.

2.3 Attribute Scoping

Custom attributes can be attached to assemblies, modules, classes and methods. This aids expressiveness because an attribute can be attached to the most refined scope necessary; only methods that require flexibility need to have attributes. We search for attributes "inside out" to aid performance, *i.e.* first at the method level, then the class and assembly levels.

As an example of how attribute scoping works, the following code calls `List::op1` and `List::op2` on an `ArrayList` implementation of a `List` interface. But if possible, the programmer would like to use the `SinglyLinkedList` implementation in the assembly that encloses class `C`, because `List::op1`'s traversal is suited to a linked list rather than an array-based list. `m2`, however, calls `List::op2`, that can reasonably expected to traverse the list backwards as well as forwards, so a `DoublyLinkedList` would be helpful:

```
[assembly: LinkClass(ArrayList, SinglyLinkedList)]

class C {
  void m1() {
    List l = new ArrayList();   l.op1();
  }

  [LinkClass(ArrayList, DoublyLinkedList)]
  void m2() {
    List l = new ArrayList();   l.op2();
  }
}
```

Linking attributes apply not only to member references, but to any type within an attribute's scope. Thus, the following code permits an object of either class `C` or `D` (or any of their subclasses) to be passed to `m1`, and an object of either class `C` or `E` (or any subclass) to `m2`.

```
[LinkClass(C,D)]
class C {
  void m1(C x) { ... }

  [LinkClass(C,E)]
  void m2(C x) { ... }
}
```

2.4 Type safety

Assemblies and classes specified in linking attributes must be binary-compatible with the assemblies and classes referenced by bytecode, or else resolution exceptions (*i.e.* "message not understood" errors) could arise at run-time. We therefore need a way to ensure that any assembly/class specified in a linking attribute chosen by the linker is type-compatible with

all references to the original assembly/class throughout an assembly. An assembly's metadata enumerates which other assemblies and classes it depends on, the members accessed in those classes are found only in individual bytecode instructions. Hence, they are only revealed at JIT-compilation when each instruction in the assembly is verified. To avoid an extra pass over bytecode during JIT-compilation, we gather constraints about member accesses with a compile-time tool, and store them as custom attributes attached to methods. These *constraint attributes* are similar in style to those in [13] and [1].

A `[LinkMemberConstraint]` attribute describes required fields and methods of classes, *e.g.*

```
[LinkMemberConstraint(``A1'', ``C1'', 100, ``M1'')]
```

states that whatever class is linked for `[A1]C1` is expected to contain a member (field or method signature) defined by token 100 in module `M1`. (A module is a unit inside an assembly that actually holds the assembly's class definitions. The metadata for the classes' dependencies - on other assemblies, class and members - is stored at the module level rather than class level, and indexed by integers known as tokens.)

A `[LinkSubtypeConstraint]` attribute encapsulates subtype constraints, *e.g.*

```
[LinkSubtypeConstraint(``A1'', ``C1'', 100,
                       ``A1'', ``C2'', 200, ``M1'')]
```

states that whatever type replaces `[A1]C1` is a supertype of whatever replaces `[A1]C2`. (100 and 200 are the tokens where `[A1]C1` and `[A2]C2` are defined in metadata.)

Fig. 2 shows how source code is annotated with linking attributes to support flexible dynamic linking. Ideally, a .NET compiler would emit member and subtype constraints after successful type-checking. But, to stay language-independent, we built a small program, `flex`, that inspects an assembly's bytecode, identifies member accesses and inserts `[LinkMemberConstraint]` attributes at the appropriate scope. Unfortunately, we cannot generate subtype constraints without performing complex data-flow analysis, as the verifier does during JIT-compilation. We currently require a programmer to specify `[LinkSubtypeConstraint]` attributes manually.

3 Architecture

We now describe how flexible dynamic linking is architected in Microsoft's shared-source version of .NET known as "Rotor". There are two candidates for which run-time subsystem should perform flexible linking for members and types: 1) the resolver called by the JIT-compiler, or 2) the loaders called by the resolver to physically find assemblies on disk and extract classes from them. The latter is an attractive place to check linking attributes, because .NET's assembly loader already consults user-defined policies for redirecting assembly versions. But, if the redirection to load a different assembly/class is done at too low a level and not exposed to the higher-level resolver, then the wrong types may be loaded later in the resolution process. For example, our constraint verifier needs to know exactly what assemblies and classes have been loaded in order to check member and class definitions. Therefore, we prefer to place our implementation closer to the JIT-compiler's resolver. (We do not consider performance, but do not believe either subsystem would have an advantage.)

Fig. 3 summarises where our implementation (boxes with bold text) lives in Rotor. It sits just below the the high-level resolution algorithm, intercepting requests to resolve members and types, and modifies requests those before assemblies and classes are actually loaded. By sitting just above the assembly loader, we apply linking attributes to a member or type resolution before user-defined policies are applied. This is appropriate, because versioning policies, *e.g.* to avoid a security flaw in an old assembly, should apply even to assemblies named in linking attributes.

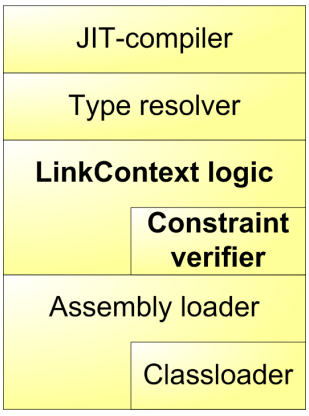

Fig. 3: Overview of flexible dynamic linking in Rotor

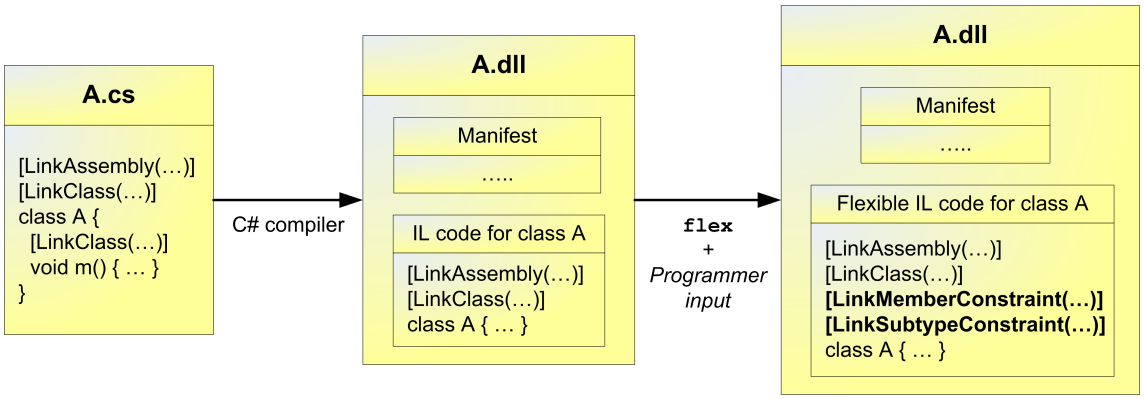

Fig. 2: Preparing code for flexible linking

3.1 Linking contexts

A member access (field access or method call) instruction in bytecode contains an integer "token" that is mapped, in metadata, to a *member descriptor* very similar to a field or method signature, *e.g.* `[A]C::m(void)`. Resolution is the process during JIT-compilation that turns a token, via a member descriptor, into a first-class object that directly represents the member's definition in a loaded class from a loaded assembly. Similarly, a class declaration contains one or more tokens that map to *type descriptors* for its superclasses, *e.g.* `[A]C`. Also, a type-cast instruction contains a token for the target type. Resolution turns these tokens into objects representing loaded class definitions. Note that a token always references at least an assembly name and a class name.

When resolving a particular token, we need to consider the linking attributes and constraint attributes applicable to, *i.e.* in scope at, the token being resolved. We call the set of in-scope linking attributes applicable to a token its *resolving context*. Each token has its own resolving context because different linking and constraint attributes apply to it.

We introduce a *LinkContext* to compute and encapsulate a resolving context. For a particular token, a LinkContext finds all the `[LinkAssembly]` attributes declared closest to it. If previous resolutions chose `[LinkAssembly]` attributes whose semantic interface qualifier was **LOCAL_INTERFACE_PREFERRED** or **LOCAL_INTERFACE_EAGER**, then a LinkContext will find only those `[LinkAssembly]` attributes with the appropriate semantic interface names. The resolving context consists of those `[LinkAssembly]` attributes, plus `[LinkClass]` attributes (with the appropriate semantic interface name, if necessary) in the same scope as the `[LinkAssembly]` attributes, plus constraint attributes in the same scope. A LinkContext can be queried for the linking and constraint attributes "relevant" to a particular token, *e.g.* if the token being resolved is for a type `[A]C`, then only `[LinkAssembly]` attributes for assembly `A` are relevant.

4 Implementation

4.1 Modifying the JIT-compiler

We add a stack of LinkContexts to each module loaded from an assembly. When a method is JIT-compiled, we push a "master" LinkContext on to the module's stack for efficiency reasons. This LinkContext immediately gathers all the linking and constraint attributes (at method, class, module and assembly levels) in scope for the method. These attributes are a superset of any individual token's resolving context.

Whenever the JIT-compiler reaches a token that it needs resolved, we push a further LinkContext on to the module's stack (and pop it after the token has been resolved). This LinkContext computes the token's resolving context by querying the master LinkContext for attributes in scope for the token, then selecting appropriate linking and constraint attributes as described in §3.1.

To actually push a LinkContext when the JIT-compiler encounters an unresolved token that refers to a member or type, we modify methods called by the JIT-compiler that resolve a token: `CEEInfo::findField`, `CEEInfo::findMethod` and

`CEEInfo::findClass`. These methods are made to create and destroy LinkContexts as follows:

```
// If linking attributes present...
if (pLink->HasLinkContext() && pLink->IsScopeFlexLinked())
  // Create a nested LinkContext
  pLink->NewNestedLinkContext(...);
else
  pLink = NULL;

// Pre-existing resolution code to
// find a field/method/class
...

if (pLink)
  // Remove the LinkContext from the stack
  pLink->GetParentLinkContext();
```

4.2 Modifying assembly loading

The pre-existing resolution code that we have elided above calls the assembly loader to visit the filesystem. As usual, the loader looks up the assembly name (of the token being resolved) in the currently executing assembly's metadata. This gives various details, such as the version number of the token's referenced assembly, which are stored in an `AssemblySpec` object. At this point, our code intercedes, passing the `AssemblySpec` to the top LinkContext on the current module's stack.

The LinkContext uses the "master" LinkContext to build the resolving context for the current token, then picks just the `[LinkAssembly]` attribute that specifies a redirection for the assembly mentioned by the token. (If there is more than one possible `[LinkAssembly]`, we pick the first.) For example, if the token mentions assembly `A`, then having `[LinkAssembly(‘‘A’’, ‘‘B’’, ‘‘1.0’’...]` in the resolving context will cause the LinkContext to choose assembly `B` v1.0. The LinkContext then loads this substitute assembly and performs some security checks that will be performed by the JIT-compiler later; we do not wish its checks to fail.

Having chosen and loaded a substitute assembly, we update the `AssemblySpec` object with the substitute assembly's name and pass the object back to the usual assembly loading logic. Since the assembly has already been loaded by LinkContext for constraint verification, it will be found immediately in the assembly loader's cache.

4.3 Modifying class loading

Ordinarily, once a valid assembly is loaded, the JIT-compiler's pre-existing resolution code uses the assembly's classloader to load the token's class. We intercede in the classloader to ask the top LinkContext on the current module's stack to choose a substitute class based on the resolving context.

The LinkContext again uses the "master" LinkContext, this time to retrieve a `[LinkClass]` attribute in the token's resolving context that has the appropriate semantic interface name and is for the token's class. The LinkContext tries to load the class specified in the `[LinkClass]`, and verify any applicable constraint attributes for it.

To respect `[LinkMemberConstraint]` attributes, a LinkContext first uses the ordinary method and field resolvers `EEClass::FindMethod` and `EEClass::FindField` to check the presence of members in the substitute class's definition (an `EEClass` object). Then, it verifies that the signatures of the members requested in constraint attributes match exactly the signature of the member found in the class. This entails resolving and loading each type (*i.e.* assembly+class) in the found members' signatures, such as method formal parameters. Since those members are in classes that *themselves* may have linking attributes, further LinkContexts are created and the whole flexible dynamic linking process recurses. A similar issue arises when verifying subtypes to respect `[LinkSubtypeConstraint]` attributes.

Having checked constraints on substituted classes, we pre-empt a later check by the JIT-compiler, which is that any loaded class is visible to the method being JIT-compiled. If the substitute class is visible and its definition satisfies member and subtype constraints, then the class's member definition or the class definition itself (depending on whether the token is a member descriptor or a type descriptor, respectively) is cached by the LinkContext for that token. The substitute class's name is then used by the classloader to retrieve an `EEClass` class definition from the assembly, as usual, and this succeeds immediately since we already loaded the class's definition to check constraints.

4.4 Source language issues

When compiling a method call, Rotor's C♯ compiler statically binds to the class that defines the method, relying on the runtime to dynamically dispatch the method in a subclass if necessary. Consider the following C♯ source code:

```
class A     { virtual  void m1() { ... } }
class B : A { override void m1() { ... } }

// Main program
[LinkClass(B,...)]
{ ... new B().m1(); ... }
```

The compiler produces bytecode that specifies `m1` *in class* `A`:

```
[LinkClass(B,...)]
newobj   instance void [...]B::.ctor()
callvirt instance void [...]A::m1()
```

At runtime, the body of `B::m1` will be executed as usual. But if the programmer wrote `[LinkClass]` attributes with alternatives to class `B`, they will never be used. LinkContext only sees a call (the `callvirt` instruction) to a method in class A, which no `[LinkClass]` attribute mentions. We could partially fix the problem by modifying the compiler to bind to the *virtual* declaration of `m1` rather than the overriding declaration; Sun made this change to `javac` between JDK1.3 and JDK1.4. Modifications to the virtual machine would also be necessary.

5 Conclusion

Dynamic linking in .NET is over-constrained because it must provide exactly the types known to a compiler on a programmer's machine. While software engineering techniques can find and link alternative code at run-time, they have to be coded into each application and often use type-unsafe reflection. We have designed a scheme that lets the programmer describe alternative choices for what types can be linked, which is the only way to ensure observational equivalence with types named in source code. If our dynamic linker picks a different type from that named in source code, then any check for type-safety, security or class visibility will succeed if it would have succeeeded for the original type. .NET's ability to attach attributes to code allows for precise specification of what and where choices should be available in a program, in a way that causes no overhead to unmodified .NET virtual machines. Also, our specifications let the programmer reflect the fact that families of assemblies are often grouped together logically, *e.g.* patent-free algorithms, so that linking one assembly should restrict later linking to the same family.

Further work is identifying when to perform flexible linking even if a compiler has "hidden" the opportunity with its static resolution, and finding real-world applications that can benefit from our scheme. Increasing portability between mobile and desktop frameworks may be a fruitful avenue, particularly as the number grows of .NET-enabled mobile devices with API support for differentiated capabilities (GPS, wi-fi, cameras, etc).

Acknowledgements This work was partially funded by a Microsoft Research grant under the 2004 Rotor RFP II. We thank Sophia Drossopoulou for comments.

References

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, Long Beach, CA, USA, January 2005.

[2] Alex Buckley. A Model of Dynamic Binding in .NET. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2005)*, Glasgow, Scotland, July 2005.

[3] Alex Buckley, Michelle Murray, Susan Eisenbach, and Sophia Drossopoulou. Flexible Bytecode for Linking in .NET. In *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005)*, ENTCS, Edinburgh, Scotland, March 2005. Elsevier BV.

[4] Sophia Drossopoulou. An Abstract Model of Java Dynamic Linking and Loading. In Robert Harper, editor, *Proceedings of the Third International Workshop on Types in Compilation (TIC 2000)*, volume 2071 of *LNCS*, pages 53–84. Springer-Verlag, 2000.

[5] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.

[6] Susan Eisenbach, Vladimir Jurisic, and Chris Sadler. Feeling the way through DLL Hell. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002)*, Malaga, Spain, June 2002. `http://joint.org/use2002/`.

[7] Susan Eisenbach, Dilek Kayhan, and Chris Sadler. Keeping Control of Reusable Components. In *Proceedings of Component Deployment (CD 2004)*, Edinburgh, Scotland, May 2004.

[8] T. Jensen, D. Le Metayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 4–15, Chicago, IL, USA, 1998.

[9] Rod Johnson. The Spring Framework. `http://www.springframework.org/`, 2005.

[10] Andrew Kennedy and Don Syme. The Design and Implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, Snowbird, UT, USA, June 2001.

[11] Richard Lander. The Wonders of Whidbey Factoring Features. `http://hoser.lander.ca/`, 2005.

[12] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'93)*, pages 439–450, Charleston, SC, USA, 1993.

[13] Frank Tip, Adam Kiezun, and Dirk Baumer. Refactoring for Generalization using Type Constraints. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2003)*, Anaheim, CA, USA, October 2003.