A Model of Dynamic Binding in .NET

Alex Buckley

Imperial College London
a.buckley@imperial.ac.uk

Abstract. Millions of programmers use ECMA CLI-compliant languages like VB.NET and C#. The resulting bytecode can be executed on several CLI implementations, such as those from Microsoft and the open-source Mono organisation. While assemblies are the standard unit of deployment, no standard exists for the process of finding and loading assemblies at run-time. The process is typically complex, and varies between CLI implementations. Unlike other linking stages, such as verification, it is visible to programmers and can be a source of confusion.
We offer a framework that describes how assemblies are resolved, loaded and used in CLI implementations. We strive for implementation-independence and note how implementations from different organisations vary in behaviour. We describe the reflection features available for dynamic loading, and give C# examples that exercise the features modelled in the framework.

1 Introduction & Motivation

Traditional language mechanisms for modular development - packages in Ada, modules in Modula-2, namespaces and classes in C++ - have no role at run-time. A compiler typically employs a static linker to emit a monolithic executable file, so the compilation environment automatically becomes the entire execution environment. Few (or no) dynamic checks are needed to resolve external dependencies. In contrast, the basic unit of development in Java and C# - the class - maintains its discrete identity throughout compilation and execution. A Java Virtual Machine or Microsoft's .NET Common Language Runtime (CLR) can start with the bytecode for just one class, then lazily load and link classes from the execution environment as necessary for continued execution.

Class loading tends to be highly configurable, unlike later linking stages such as verification. UNIX offered the `dlopen` C-language interface and most OO languages offer an API for dynamic class loading in their reflection libraries. Java has the familiar `CLASSPATH` mechanism for identifying class locations, and custom classloaders can be installed into the JVM.

Matters are complicated in a CLI implementation [ECM02] because classes are not deployed as standalone units. Instead, classes are encapsulated inside assemblies. An assembly enumerates classes it provides and also the names of other assemblies whose classes it uses. Assembly resolution consists of converting a bytecode's reference to an assembly name into a physical location where a

suitable assembly exists. Because an assembly's identity incorporates version and security information, resolving an assembly is more complex than (and indeed, a pre-requisite to) finding a class inside an assembly.[1]

Different CLI implementations have different rules for assembly resolution. Also, the process of loading an assembly from a given location is implementation-specific. We use the term *binding* for the combined resolution and loading process.

Microsoft's CLI implementation, the Common Language Runtime (CLR) [MG00], provides a user-configurable, network-aware system for binding called "Fusion". For resolution, it supports a hierarchy of policies that can modify the requested version of an assembly. This allows security and performance patches to be used without rebuilding an assembly's manifest (akin to recompiling source code). It can also resolve references to assemblies compiled for other platforms, such as the .NET Compact Framework. For loading, it supports downloading of code from remote machines and, as a last resort, on-demand installation where the user is asked to provide an assembly.

Fusion's behaviour is typically explained in verbose official technical documentation. Recently, "blogs" written by Microsoft employees [Coo05,Shi05] [Zha05,Far05,Pra05] have explained areas of poorly documented behaviour in the current CLR release (v1.1), and given detailed information about the next CLR release (v2.0). Programmer understanding is significantly enhanced by this new channel, but there is no single place where dynamic loading is explained in full detail from 'top to bottom'. One must piece together information from around the Internet in order to explain a program's exact assembly and class-loading behaviour.

An alternative CLI implementation is Mono [dI05]. Its functionality is a subset of the CLR's, including for assembly binding, so documentation is shorter and easier to understand. However, its binding process is subtly different from Fusion's. Other CLI implementations, such as Microsoft's .NET Compact Framework for mobile devices, also exhibit different behaviour from Fusion.

We wish to unify the rules that govern assembly binding in CLI implementations. We present a model that describes, at one level, how assemblies are bound (*i.e.* resolved and loaded), and at another level, how loaded assemblies are used when evaluating bytecode instructions. Assembly binding is interleaved with bytecode evaluation as in all current CLI implementations.[2] The model is parameterised by resolution and loading policies, so we specialise it for the Microsoft CLR v1.1 and Mono v1.1.

---

[1] We do not consider, in this paper, the resolution of classes in an assembly or of members in a class.

[2] The CLI specification allows resolution to take place when an application is installed, but we do not know of any implementation that takes such an eager approach.

2 The Assembly Model

2.1 Assembly structure

In the COM and Java environments, a file that contains code has only one identifier: its filename. "DLL hell"[EJS02] arises because multiple DLL files, each containing different code, share the same filename and are placed in a shared location on disk. An application's dependency is resolved to a filename, but there is no guarantee that the DLL file with that filename is what the application was tested against. Java applications face a similar problem, even without a standard location in the filesystem for classes.

In contrast, the CLI specification[ECM02] gives an assembly a logical identity quite different from its filename. We call this identity an *assembly name*, and reflection APIs in CLI-compliant languages typically make it a first-class value. It contains a display name, a version number (consisting of major, minor, revision and build numbers), a cultural identifier (for internationalisation) and a public key. It is convenient to just consider the presence of a security value in an assembly name, rather than the public key per se.

$AssemblyName\ \alpha : AN =$

$DisplayName : id, Version : int \times int \times int \times int, Culture : id, Security : id, Retargetable : bool$

Binding maps an assembly name to an assembly definition. All elements in an assembly name are potentially used during binding, *e.g.* if the culture is present, it can be used to choose a directory on disk where an assembly definition might be found. The security value plays the most important role because it determines whether an assembly name is a *strong name*. A non-null security value indicates that the assembly has been signed by a private key. A verification procedure can use the security value to detect unauthorised changes to the assembly, but we do not consider verification further since it happens after binding. However, whether or not an assembly name is strong significantly affects binding, so this definition will be useful:

$StrongName(\alpha) \equiv Key(\alpha) \neq \epsilon$

An *assembly definition* consists of an assembly name, assembly dependencies and class definitions. Bytecode refers to assemblies by their display name, so the dependencies map display names to full assembly names. We assume that bytecode is encapsulated in class definitions of some type $ClassDef$. An assembly definition knows the location of the file that it was loaded from; this is used in type-casting and reflection operations.

$AssemblyDefinition\ \delta : AD =$

$Name : AN, Refs : id \longrightarrow AN, Code : id \longrightarrow ClassDef, Loc : id$

The CLI specification defines an assembly as comprised of modules (that contain bytecode) and other resource files. An assembly's module and resource files may be placed in a single physical file or left as independent files. However,

modules and the physical layout of an assembly play no role in binding[3], so we ignore them in our model. This keeps the definition of the *Code* element simple.

2.2 Assembly environment

Most CLI implementations (though not the CLI specification itself) support a standard location on disk where assemblies can be placed, typically if they have a strong name. At load-time, this location is typically checked before others, and thus provides the default environment from which assemblies come. In the CLR and Mono, the environment is provided by the Global Assembly Cache (GAC).

Environment $\Delta : Env = AN \longrightarrow AD$

We introduce the *extended environment* to represent both the filesystem of the machine executing the code, and a URL-addressable space of machines that have assemblies available for download. Given a list of paths, the extended environment tries each in turn until an assembly definition is found; it returns ϵ if the list is exhausted without finding an assembly.

Extended Environment $EE : ExtEnv = id^* \longrightarrow AD$

3 Assembly-Oriented Execution

3.1 State

We wish to show how assembly identity, resolution and loading affect execution. We distinguish the state of the executing program from the state of the runtime system itself. Program state P is a pair whose elements are an instruction stack and an operand stack.

Program state $P : I^* \times V^*$

A CLI instruction I is parameterised by a display name and a member descriptor, M. The display name must have a corresponding entry in the $Refs$ element of the enclosing assembly. A member descriptor is simply a class and a field/method signature. Values come from a type V with which we are not concerned.

The runtime system's state is represented by three elements: an environment (defined in section 2), a heap and a stack.

The runtime system's heap stores assembly definitions loaded from the environment and extended environment. The CLR's heap is divided into two parts, called *contexts*.[4] Contexts stop a programmer circumventing the system's binding policies. The Mono system does not support contexts at present.

[3] Partition I, §9.6: "... rather than establishing relationships between individual modules and referenced assemblies, every reference is resolved through the current assembly. This allows each assembly to have absolute control over how references are resolved."

[4] In fact, there is a third heap context, but its role is not important in our current model.

An assembly loaded by the CLR itself is placed in the first context. This happens when a bytecode instruction is jitted and the instruction's display name is resolved. Assemblies loaded directly from a filename are placed in the second context. This happens when a programmer uses the reflection API provided by the core assemblies in the CLR and Mono. With a heap consisting of a pair of mappings from assembly name to definition, we write H_x for $H \downarrow_1$.

$Heap\ H : (AN \longrightarrow AD) \times (AN \longrightarrow AD)$

In a CLI implementation, the heap of loaded assemblies is part of an appdomain, which is a logical unit of isolation in a process. As we do not model the ability of a program to dynamically create and destroy appdomains, there is exactly one appdomain per executing application. Therefore, we do not need to qualify our heap of assemblies with an appdomain.

We need to track the call stack of assemblies at each dynamic program point. This is because the references of the currently executing assembly is consulted when resolving a reference to another assembly.[5] In addition, the context of the currently executing assembly is important when resolving an assembly reference. The stack starts with the assembly that the operating system considers is the entrypoint for an application.

$Stack\ S : (AN \times \{1, 2\})^*$

3.2 Evaluation

Evaluation is performed by a small-step operational semantics that evolves the state of the runtime system (δ, H, S) and the program state (P).

$$\Delta, H, S, P \longrightarrow \Delta', H', S', P'$$

The rules are shown in fig. 1. The bytecode instruction on the program's instruction stack can be evaluated if it depends on an assembly already loaded into the system heap. (Rules Exec-Instr, Exec-Instr-Call, Exec-Instr-Cast) Details of the evaluation are not important, so we abstract it into this judgement which evolves the program state given an assembly definition needed by the instruction:

$$\delta, P \longrightarrow P'$$

We are forced to differentiate the call instruction from other instructions because we need to add the called assembly's name to the system stack, and modify the program's instruction stream with the body of the called method. We assume a *lookup* function that can find a member M in an assembly.

A binding step can take place to resolve and load an assembly that an instruction is dependent on. (Rule Exec-Bind) It uses the binding rules that evolve an environment and heap with an assembly definition for assembly name α, returning the name of the actual assembly loaded:

$$\Delta, H, \alpha \longrightarrow \Delta', H', \alpha'$$

The execution is stuck if binding fails to find an assembly definition, *i.e.* α' is ϵ.

[5] CLI Specification Partition 1 §9.6

Heap contexts in evaluation To evaluate a bytecode instruction, a definition must be available for the assembly it refers to. As per the CLI specification, we take the display name N mentioned in an instruction and look it up in the references of the currently executing assembly T, obtaining a full assembly name α. In the CLR, which heap context to look up this assembly name α in depends on which context the currently executing assembly is loaded in. An assembly loaded in the first context can only "see" assemblies also loaded in the first context; an assembly loaded in the second context can see assemblies in both contexts, preferring the second. This policy is justified by the first context being where assemblies are "officially" loaded and the second context being where expert programmers place their own assemblies. (Mono only has one context, so the issue does not arise.)

$$context^{CLR}(\alpha, H, x) = \begin{cases} x \; if \; \alpha \in dom(H_x) \\ 1 \; if \; x = 2 \quad \wedge \quad \alpha \notin dom(H_2), \in dom(H_1) \end{cases}$$

$$context^{Mono}(\alpha, H, x) = x$$

Casting Casting is complicated because assemblies play the same role as classloaders in Java, *i.e.* scoping a class such that a type is an (assembly name,class name) pair. Ensuring that the same classes from different assemblies are not confused is an important defence against attacks. Therefore, in the CLR, the source and target classes must be defined in the same *assembly file on disk*.

In addition, the heap context in which an assembly is loaded provides another level of qualification for a class, *i.e.* a type in the CLR is a (context id,assembly name,class name) triple. The same assembly definition can be loaded into multiple contexts, but casting an object across contexts would give rise to the same problems as casting it across classloaders. Therefore, the assembly definitions containing the source and target classes must be in the same context.

The Exec-Instr-Cast rule first obtains the full assembly name α referred to by the `castclass` instruction. We assume that the object to be cast is accessible via the top value v on the program state's value stack, and that the auxiliary function *type* returns an (assembly name,class name) pair representing the object's type. The assemblies named by α and α' must be loaded, potentially in different contexts. We check that the two loaded assemblies were loaded from identical paths, as required by the CLR. If so, then the success of the cast is for the program to determine; we assume a notional *cast* operator that checks subclassing using the class definitions provided from an assembly definition:

$$Code(H_y(\alpha)), P[cast \; C' \; to \; C] \longrightarrow P'$$

4 Assembly Binding

The binding rules in fig. 2 take a logical assembly name and return an assembly definition plus a name. If the assembly is not already loaded in the heap, then

$$\boxed{P[-] \;=\; (- :: is \;\;,\;\; v :: vs)}$$

$$\boxed{I[-] \;=\; ldfld\ [-]M \;\mid\; stfld\ [-]M \;\mid\; new\ [-]M}$$

(Exec-Instr)
$$\frac{Refs(H_x(T))(N) = \alpha \qquad y = context(\alpha, H, x) \qquad Code(H_y(\alpha)), P[I[N]] \longrightarrow P'}{\Delta, H, (T, x) :: Ts, P[I[N]] \longrightarrow \Delta, H, (T, x) :: Ts, P'}$$

(Exec-Instr-Call)
$$\frac{Refs(H_x(T))(N) = \alpha \qquad y = context(\alpha, H, x) \qquad lookup(H_y(\alpha), M) = e}{\Delta, H, (T, x) :: Ts, P[call\ [N]M] \longrightarrow \Delta, H, (\alpha, y) :: (T, x) :: Ts, P[e]}$$

(Exec-Instr-Cast)
$$\frac{\begin{array}{l} Refs(H_x(T))(N) = \alpha \qquad y = context(\alpha, H, x) \\ type(v) = (\alpha', C') \qquad z = context(\alpha', H, x) \\ Loc(H_y(\alpha)) = Loc(H_z(\alpha')) \\ Code(H_y(\alpha)), P[cast\ C'\ to\ C] \longrightarrow P' \end{array}}{\Delta, H, (T, x) :: Ts, P[castclass\ [N]C] \longrightarrow \Delta, H, (T, x) :: Ts, P'}$$

$$\boxed{E[-] \;=\; I[-] \;\mid\; castclass\ [-]M \;\mid\; call\ [-]M}$$

(Exec-Bind)
$$\frac{Refs(H_x(T))(N) = \alpha \qquad \Delta, H_1, \alpha \longrightarrow \Delta', H_1', \alpha'}{\Delta, H, (T, x) :: Ts, P[E[N]] \longrightarrow \Delta', (H \cup_1 H_1'), (T, x) :: Ts, P[E[N]]}$$

(Exec-Run)
$$\frac{\Delta, H, S, P[E[N]] \longrightarrow \Delta', H', S, P[E[N]] \qquad \Delta', H', S, P[E[N]] \longrightarrow \Delta', H'', S, P'}{\Delta, H, S, P[E[N]] \longrightarrow \Delta', H'', S, P'}$$

$$H \cup_1 H' \equiv (H_1[y \mapsto H'(y) | y \in dom(H')], H_2)$$

Fig. 1. Execution and Loading

they use a name resolver η, a location resolver $\odot$, a assembly installer $\oplus$, and a name matcher $\sim$.

A name resolver performs a logical-to-logical mapping, applying versioning policy to an assembly name in order to obtain a more refined assembly name. A location resolver performs a logical-to-physical mapping, taking an assembly name and applying a "probing" policy that describes where to search for an assembly definition. If the location resolver fails to provide a location where a suitable assembly can be found, then an on-demand (*i.e.* "just-in-time") assembly installation operation is tried, via $\oplus$.

If the extended environment is able to find an assembly, or an assembly is installed on-demand, then the binding rules return the heap augmented with the assembly definition, plus the name of the assembly that was actually loaded. CLI

implementations require that the loaded name matches the name of the desired assembly (*i.e.* produced by the name resolver), according to $\sim$.

$Name\ Resolver\ \eta : AN \longrightarrow AN$

$Location\ Resolver\ \odot : AN \times E \longrightarrow id^*$

$Installer\ \oplus : AN \times E \longrightarrow E$

$Name\ Matcher\ \ \sim : AN \times AN \longrightarrow bool$

We introduce an *application context* that stores facts about the runtime environment for use by the name and location resolvers.

$$Application\ Context\ \Gamma : (RuntimeVersion : int \times int \times int \times int,\\ Mapping : AN \longrightarrow (AN \times id), AppPath : id)$$

We define a Binding Framework $BF = (\Gamma, \Delta, \eta, \odot, \oplus, \sim)$. A binding framework is instantiated for a specific combination of CLI implementation and user application. The CLI implementation supplies the environment Δ, which is a single directory for the CLR and one or more directories for Mono. The CLI implementation also supplies $\Gamma_{RuntimeVersion}$, η, $\odot$, $\oplus$ and $\sim$. The user application supplies its location on disk $\Gamma_{AppPath}$, which is independent of any CLI implementation. $\Gamma_{Mapping}$ is discussed in the next section.

Instantiating the binding framework several times allows modelling of "side-by-side execution", where several CLI implementations can be installed on the same machine, each with its own core assemblies. The operating system chooses which implementation is suitable for executing a given application, which provides further information necessary for its execution.

(Bind-Already-Loaded)

$$\frac{\alpha \in dom(H)}{\Delta, H, \alpha \longrightarrow \Delta, H, \alpha}$$

(Bind-Available)

$$\frac{\alpha \notin dom(H) \quad \eta(\alpha) = \alpha' \quad EE(\Delta \odot \alpha') = \delta \quad \alpha' \sim Name(\delta)}{\Delta, H, \alpha \longrightarrow \Delta, H[\alpha \mapsto \delta], Name(\delta)}$$

(Bind-Install-On-Demand)

$$\frac{\alpha \notin dom(H) \quad \eta(\alpha) = \alpha' \quad EE(\Delta \odot \alpha') = \epsilon \quad \Delta \oplus \alpha' = \Delta' \quad \alpha' \sim Name(\Delta'(\alpha))}{\Delta, H, \alpha \longrightarrow \Delta', H[\alpha \mapsto \Delta'(\alpha)], Name(\Delta'(\alpha'))}$$

(Bind-Unavailable)

$$\frac{\alpha \notin dom(H) \quad \eta(\alpha) = \alpha' \quad EE(\Delta \odot \alpha') = \epsilon \quad \Delta \oplus \alpha' = \Delta}{\Delta, H, \alpha \longrightarrow \Delta, H, \epsilon}$$

Fig. 2. Binding

4.1 Name Resolution

A name resolver η maps a logical assembly name to another logical assembly name, according to three policies: servicing, unification and retargeting. Fig. 3 shows name resolvers for the CLR and Mono.

Servicing policy To allow assemblies to be *serviced* (*i.e.* upgraded for security and performance reasons without modifying calling applications), the CLR supports policies for redirecting references to strongly-named assemblies. (A reference to a non-strongly-named assembly cannot be serviced.)

First, each application can supply a policy file for redirecting one version of a given assembly to another. Second, "publisher policies" can redirect requests for assemblies in the GAC. Third, a machine-wide redirection policy is applied after the application and publisher policies. We represent the union of these policies as a mapping from assembly name to assembly name in $\Gamma_{Mapping}$ (using the first element of the range). In contrast, Mono does not currently support redirection policies, so its $\Gamma_{Mapping}$ is empty.

Unification policy A CLI-compliant virtual machine, such as the CLR, is often developed by different individuals from those who program the core assemblies that accompany the VM.[6] It is often practical to test a VM only with the exact framework assemblies that will accompany it.

The CLR and Mono both impose a restriction that some core assemblies (the exact set differs) must be the same version as that of the runtime execution system itself.

Retargeting policy As well as the CLR, Microsoft produces a CLI implementation for mobile devices called the .NET Compact Framework. An application compiled *for the CLR* will not run on a mobile device equipped with just the .NET Compact Framework, even if the developer is careful to use only assemblies available in the Compact Framework. This is because the core assemblies that accompany the CLR have different strong names from the assemblies in the Compact Framework [Mot04].

However, an application compiled *for the .NET Compact Framework* will run on the CLR. This is possible because the generated assembly references the Compact Framework's assemblies by their strong names, as usual, but each reference features a *retargetable* flag. The .NET Compact Framework's runtime ignores this flag and resolves the core assemblies as usual. The CLR reacts to it by rewriting the retargetable assembly names to the relevant core assembly names; the version number is unified and the key token is set to a standard value that indicates a core assembly to Fusion. This is Microsoft-specific behaviour; the Mono runtime will halt on failing to resolve the strong names of the Compact Framework assemblies referenced by the application.

4.2 Location resolution

A location resolver $\odot$ supplies a list of physical filenames for the extended environment to try to obtain an assembly from. Fig. 4 shows location resolvers for the CLR and Mono.

[6] In Java, the java.lang.* class hierarchy.

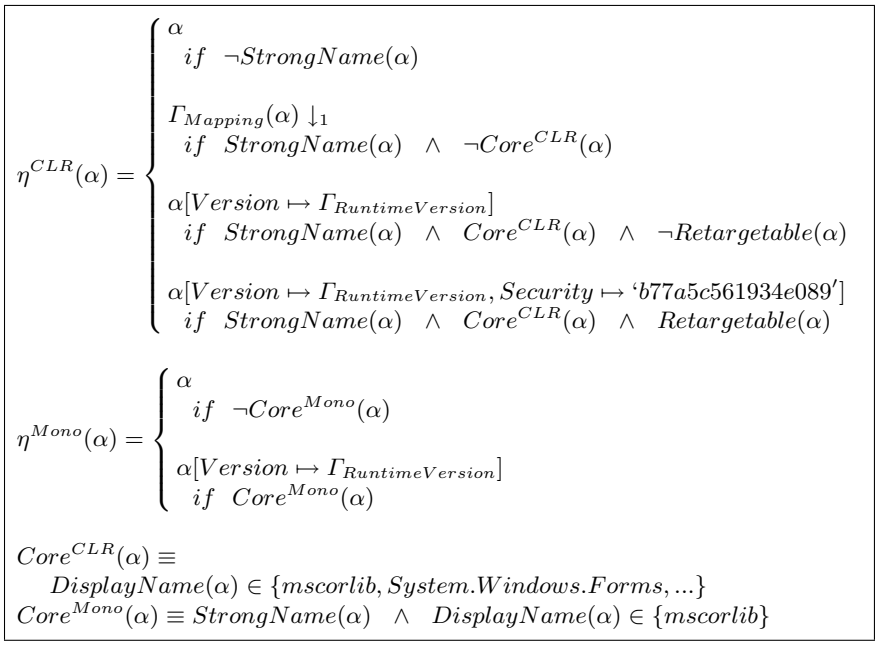

Fig. 3. Name Resolution

Given an assembly name, the CLR's location resolver prefers to search the environment first if the assembly's name is a strongname. The next possible location is a "codebase" from the application context, specifically the second element of the $\Gamma_{Mapping}$ entry for the target assembly name. The codebase's location is final in the sense that no alternative paths are tried if it is specified. If a codebase is not specified, then various locations in the filesystem are suggested, using the path of the currently executing application (which is not necessarily that of the currently executing assembly). The extended environment will "probe" each of these locations in turn.

When performing location resolution for an assembly name that is not a strongname, the environment is not used. If a codebase is available, it must come from the same location as the executing application. Otherwise, the filesystem is tried as before.

The location resolver for Mono is quite different. It tries the application's local directory first before the environment. (It also searches a `CLASSPATH`-style directory list before the environment, but we do not show this.)

4.3 Name matching

The CLR and Mono require an exact match between desired and loaded assembly versions:

$$\Delta \odot^{CLR} \alpha = \begin{cases} \Delta, L \\ \quad if \;\; StrongName(\alpha) \;\; \wedge \;\; \Gamma_{Mapping}(\alpha) \downarrow_2 = L \\ \\ \Delta, Locs(\alpha) \\ \quad if \;\; StrongName(\alpha) \;\; \wedge \;\; \Gamma_{Mapping}(\alpha) \downarrow_2 = \epsilon \\ \\ L \\ \quad if \;\; \neg StrongName(\alpha) \;\; \wedge \;\; \Gamma_{Mapping}(\alpha) \downarrow_2 = L \\ \qquad \wedge \;\; L = \Gamma_{AppPath} +'' /'' + x \;\; for \;\; some \;\; x \\ \\ Locs(\alpha) \\ \quad if \;\; \neg StrongName(\alpha) \;\; \wedge \;\; \Gamma_{Mapping}(\alpha) \downarrow_2 = \epsilon \end{cases}$$

$$\begin{aligned} Locs(\alpha) = {} & (\Gamma_{AppPath} + \text{“}/'' + DisplayName(\alpha) + \text{“}.dll''), \\ & (\Gamma_{AppPath} + \text{“}/'' + DisplayName(\alpha) + \text{“}/'' + DisplayName(\alpha) + \text{“}.dll''), \\ & (\Gamma_{AppPath} + \text{“}/'' + Culture(\alpha) + \text{“}/'' + DisplayName(\alpha) + \text{“}.dll''), \\ & (\Gamma_{AppPath} + \text{“}/'' + Culture(\alpha) + \text{“}/'' + DisplayName(\alpha) + \text{“}/''+ \\ & \quad DisplayName(\alpha) + \text{“}.dll'') \end{aligned}$$

$$\Delta \odot^{Mono} \alpha = (\Gamma_{AppPath} +'' /'' + DisplayName(\alpha) +'' .dll''), \Delta$$

Fig. 4. Location Resolution

$$\Delta \;\; \oplus^{CLR} \;\; \alpha = \Delta' \;\; for \;\; some \;\; \Delta' \supseteq \Delta \;\; where \;\; \Delta'(\alpha) \neq \epsilon \;\; \Longrightarrow \;\; \Delta(\alpha) = \epsilon$$

$$\Delta \;\; \oplus^{Mono} \;\; \alpha = \Delta$$

Fig. 5. Software Installation

$$\begin{aligned} \alpha \;\; \sim^{CLR,Mono} \;\; \alpha' \;\; \equiv \\ StrongName(\alpha) \Longleftrightarrow StrongName(\alpha') \;\; \wedge \\ Version(\alpha) = a.b.c.d \Longleftrightarrow Version(\alpha') = a.b.c.d \end{aligned}$$

4.4 Install-on-demand

If both the environment and the extended environment fail to supply an assembly, the $\oplus$ function tries to perform an “install-on-demand” operation. Unlike the extended environment, which is queried at a specific location (*e.g.* a URL), the installer is required to return an assembly given just its name.

In the CLR, we suppose that the end user is asked to supply an assembly, *e.g.* on a CD. Because the supplied assembly is totally free, we pass the old environment to $\oplus$ to see that if it does grow, then a truly new assembly is available in the new environment. This approach allows us to accept that the installation can fail, leaving the environment unchanged and propagating (through binding rule Bind-Unavailable) a loading failure.

Mono does not support on-demand installation, so returns an unchanged environment.

4.5 Dynamic loading through reflection

$LOAD \equiv call[mscorlib]System.Reflection.Assembly :: Load$
$LOADFROM \equiv call[mscorlib]System.Reflection.Assembly :: LoadFrom$

$$\boxed{P[-,-] \quad = \quad (- :: is \quad , - :: vs)}$$

(Exec-Instr-CallLoad)

$$\frac{\Delta, H_1, \alpha \longrightarrow \Delta', H_1', \alpha'}{\Delta, H, (T,x) :: Ts, P[LOAD, \alpha] \Longrightarrow \Delta', (H \cup_1 H_1'), (T,x) :: Ts, P[\epsilon, \alpha']}$$

(Exec-Instr-CallLoad2)

$$\frac{IsDisplayName(N) \qquad EE(\Delta \odot N) = \delta \qquad \Delta, H_1, Name(\delta) \longrightarrow \Delta', H_1', \alpha}{\Delta, H, (T,x) :: Ts, P[LOAD, N] \Longrightarrow \Delta', (H \cup_1 H_1'), (T,x) :: Ts, P[\epsilon, \alpha]}$$

(Exec-Instr-CallLoad3)

$$\frac{IsDisplayName(N) \qquad EE(\Delta \odot N) = \delta \qquad \Delta, H_1, Name(\delta) \longrightarrow \Delta, H_1, \epsilon}{\Delta, H, (T,x) :: Ts, P[LOAD, N] \Longrightarrow \Delta, (H \cup_1 [Name(\delta) \mapsto \delta]), (T,x) :: Ts, P[\epsilon, Name(\delta)]}$$

(Exec-Instr-CallLoadFrom)

$$\frac{\begin{array}{l} EE(L) = \delta \qquad \Delta, H_1, DisplayName(Name(\delta)) \Longrightarrow \Delta', H_1', \alpha \\ \alpha \neq \epsilon \quad \wedge \quad Loc(\delta) = Loc(H_1'(\alpha)) \end{array}}{\Delta, H, S, P[LOADFROM, L] \Longrightarrow \Delta', (H \cup_1 H_1'), S, P[\epsilon, \alpha]}$$

(Exec-Instr-CallLoadFrom2)

$$\frac{\begin{array}{l} EE(L) = \delta \qquad \Delta, H_1, DisplayName(Name(\delta)) \Longrightarrow \Delta', H_1', \alpha \\ \alpha = \epsilon \ \vee \ Loc(\delta) \neq Loc(H_1'(\alpha)) \end{array}}{\Delta, H, S, P[LOADFROM, L] \Longrightarrow \Delta, (H \cup_2 [Name(\delta) \mapsto \delta]), S, P[\epsilon, Name(\delta)]}$$

$H \cup_1 H'$ *as before*

$$H \cup_2 [\alpha \mapsto \delta] \equiv \begin{cases} H & if\ \alpha \in dom(H_2) \\ (H_1, H_2[\alpha \mapsto \delta]) & otherwise \end{cases}$$

Fig. 6. Dynamic loading through reflection

As stated in section 2, assemblies can be loaded using a reflection API. This is widely used by developers building applications that support plug-ins. Among the many reflection methods provided by the CLR's core assemblies, we consider `Load` and `LoadFrom`. Mono's core assemblies provide `Load` only. The full method signatures are shown in fig. 6.

The `Load` method takes a strong name α from the program state's value stack, and defers to the standard binding rules in fig. 2 to resolve and load it. It is as if a strongname α has been found in an assembly's metadata, *i.e.* `Load`'s behaviour is that of Exec-Bind in fig. 1.

`Load` can also take a display name N, *e.g.* "Calc". In this case, it probes the local directory first. If `Calc.dll` is found and does not have a strongname, then that file is bound to immediately. But if the file has a strongname, then that strongname is used to initiate the standard binding process. If this process succeeds, the assembly it finds is `Load`'s result, rather than the local `Calc.dll`. If the process fails, then `Load` return the local `Calc.dll` assembly. The interesting case is when `Calc.dll` is not present locally, because then there is no strongname available to attempt to bind with - *even if a suitable* `Calc` *assembly is in the GAC*. The formal system is stuck in this case, reflecting that no assembly would be returned by `Load`.

The `LoadFrom` method is complex too. It takes a location L from the program state's value stack, and loads the file at that location, *e.g.* `c:\app\Calc.dll`. It then initiates the standand binding process with the *display name* embedded in that file, *i.e.* `Calc`. If this process returns an identical assembly definition from the heap's first context - *i.e.* an assembly in that context was already loaded from `c:\app\Calc.dll` - then that assembly in the first context is `LoadFrom`'s result. The file just loaded from `c:\app\Calc.dll` is ignored and the second heap context is unchanged. However, if the standard process fails to find an exact match for `Calc` in the first context - perhaps one exists, but loaded from `d:\libs\Calc.dll` - then the assembly from `c:\app\Calc.dll` is bound in the second context.

5 Related & Further Work

Classloading in Java has received significant attention[Dea97,JLT98,LB98,QGC00], and [FZA04] presents it in an abstract setting. However, relatively little work focuses on the CLI platform. [DLE03] unifies dynamic linking in Java and the CLI, but abstracts the assembly binding process to a very high level. [EJS02] and [EJS03] offer a formal model of a well-formed GAC, where assembly addition and removal do not break existing dependencies. Our work is clearly complementary to this, as we show how the GAC is used in the wider assembly binding process. Our $\oplus$ operator would ideally maintain a stronger safety property concerning evolution of the GAC[EJS03].

We have described and formalised how assemblies are resolved and loaded by common CLI implementations. Most programmers assume that an assembly's strong name is its sole identity once loaded, but we show how the CLR, during execution, considers an assembly's identity to have more elements. Namely, it considers where an assembly was loaded from (*i.e.* a disk or URL-based location) and where it was loaded to (*i.e.* its heap context). These elements are necessary because the CLR exposes reflective assembly loading operations that can load arbitrary assemblies. While merely loading such assemblies is harmless, it is essential to avoid using their classes if the assembly's identity masquerades as one of the core assemblies. We plan to state formally that binding is "safe" in the current CLR in that it never leads to a heap where a non-core assembly is

mistaken for a core assembly. The Mono system avoids the problem at present by not offering reflective loading capabilities.

A weakness of the current model is that name resolution produces a very precise answer, *i.e.* a single assembly name. This does not accurately model the .NET Compact Framework or, indeed, more flexible future schemes for choosing an assembly to load[BD04,BMED05]. The .NET Compact Framework does not support servicing policies that redirect an assembly's desired version, so applications cannot be directed to use later, better code. However, the Compact Framework's binding rules permit the loader to provide version $a.b.c.x$ of an assembly when a reference is made to version $a.b.c.d$, *i.e.* the last element of the version number can "float". The binding rules also permit *any* version of an assembly to be loaded when the reference mentions version 0.0.0.0.

In our model, this equates to the name resolver producing $a.b.c.*$ for the desired version to locate. We could modify name resolution to produce a constraint on permitted names, rather than a specific name. Location resolution would then need to iterate through the files found in the extended environment to choose the "best" one matching the constraint. The name matcher would have the following definition:

$$\begin{aligned}&\alpha \quad \sim^{CompactFramework} \quad \alpha' \quad \equiv \\ &\quad StrongName(\alpha) \iff StrongName(\alpha') \quad \wedge \\ &\quad ((Version(\alpha) = a.b.c.\_ \iff Version(\alpha') = a.b.c.\_) \vee Version(\alpha) = 0.0.0.0)\end{aligned}$$

The CLR v2.0 will be released in late 2005 and makes some small changes to unification policy[Shi05], so we will need a new name resolver. More interesting are Microsoft's plans for binding in Longhorn[GR04], where assemblies are typed and servicing policy is affected by the types of referencing and referenced assemblies. A feature called "interim roll-back" is also planned, where assemblies installed in the environment are temporarily hidden due to flaws being found in them. Our model can handle the new servicing policy (at name resolution) and rollback policy (at location resolution). More challenging is to state whether syntactic or semantic compatibility is assured by these new features.

References

[BD04] Alex Buckley and Sophia Drossopoulou. Flexible Dynamic Linking. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2004)*, Oslo, Norway, June 2004.

[BMED05] 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.

[Coo05] Suzanne Cook. .NET CLR Loader Notes. `http://blogs.msdn.com/suzcook`, 2005.

[Dea97] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[dI05] Miguel de Icaza. Mono. http://www.mono-project.com/, 2005.

[DLE03] 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.

[ECM02] ECMA. *Standard ECMA-335: Common Language Infrastructure*. ECMA International, December 2002. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[EJS02] S. Eisenbach, V. Jurisic, and C. Sadler. Feeling the way through DLL Hell. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002)*, Malaga, Spain, June 2002.

[EJS03] S. Eisenbach, V. Jurisic, and C. Sadler. Managing the Evolution of .NET Programs. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pages 185–198, Paris, France, November 2003. Springer-Verlag.

[Far05] Shawn Farkas. .NET Security Blog. http://blogs.msdn.com/shawnfa, 2005.

[FZA04] Sonia Fagorzi, Elena Zucca, and Davide Ancona. Modeling Multiple Class Loaders by a Calculus for Dynamic Linking. In *Proceedings of the ACM Symposium on Applied Computing (SAC-2004)*, Nicosia, Cyprus, March 2004.

[GR04] Cathi Gero and Jeffrey Richter. The Future of Assembly Versioning. http://www.theserverside.net/articles/showarticle.tss?id=AssemblyVersio%ning, 2004.

[JLT98] 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.

[LB98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'98)*, Vancouver, BC, Canada, October 1998.

[MG00] Eric Meijer and John Gough. *Technical Overview of the Common Language Runtime*. Microsoft, 2000.

[Mot04] Daniel Moth. http://www.danielmoth.com/Blog, 2004.

[Pra05] Steven Pratschner. .NET CF WebLog. http://blogs.msdn.com/stevenpr, 2005.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of Java Class Loading. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2000)*, pages 325–336, Minneapolis, MN, USA, 2000.

[Shi05] Alan Shi. The Fusion Weblog. http://blogs.msdn.com/alanshi, 2005.

[Zha05] Junfeng Zhang. .NET Framework Notes. http://blogs.msdn.com/junfeng, 2005.

A Examples

C# examples that demonstrate heap contexts and assembly identity can be found at
http://slurp.doc.ic.ac.uk/pubs/dynamicbindingindotnet-examples.pdf.