What's In a Name?
.NET as a Component Framework (DRAFT)

Erik Meijer and Clemens Szyperski

{emeijer,cszypers}@microsoft.com

Abstract

In this paper, we analyze the challenges surrounding the problem of reuse and independent extensibility of software components beyond source sharing, relate them back to the development and deployment process, and show how many of the problems in this area can be traced back to name resolution problems. We show how the Common Language Infrastructure (CLI) addresses these problems in novel ways.

1 Introduction

Software components are usefully characterized as units of deployment [2]. In this paper, whenever we say component we mean software component in that sense. When targeting systems that accept new components over time, components also need to be units of versioning. As explained in more detail later, the reason is that dependencies between components are really dependencies on particular *versions* of components.

Componentized software evolves by means of adding and replacing components over time. Replacement of components can be categorized into the actual deletion of an old component (with possibly draconian consequences) and the addition of a new version of a previously installed component, without actually removing the latter. Both cases lead to interesting challenges. In this paper, we analyze these challenges, relate them back to the development and deployment process, and show how many of the problems in this area can be traced back to name resolution problems. We show how the Common Language Infrastructure (CLI) addresses these problems in novel ways. The CLI is the public specification underlying the Microsoft .NET Common Language Runtime (CLR).

Since in this context naming is ultimately a means of bridging between dependencies intended by a programmer, we conclude by stating requirements for

programming languages that wish to fully support a CLI-style name resolution strategy

1.1 Binding in programs

.

The process of name resolution is usually called *binding*. When refering to definitions contained in separate components, a name binding process needs to draw on *metadata* contained in or shipped with these components to determine how names *into* a component map to definitions inside that component.

For the purpose of this paper we consider the following *binding times* in the lifetime of a program:

Build time Program source is compiled into *components* or *deployment units* during *build time*. Components are not *a priori* committed to either specific programming language constructs, or granularity, moreover, a components might also contain other resources and constant data beyond code. To support reuse and independent extensibility beyond source sharing, the build process might refer to *meta data* attached to previously build components (details later).

Load time Components are loaded into an execution environment at *load time*, yielding an executable image. Metadata in a loaded unit indicates static dependencies on other units. The loader has to check whether these are already loaded and, if not, locate and load these as well. (A loader's strategy may vary depending on how lazily or how eagerly the resolution of such dependencies is performed.)

Run time Executables are run at *run time*. Like loading, execution might cause transitive building, loading and subsequent execution of other components demanded dynamically by executing code.

This is of course a simplification of real life where a program might go through design time, link-time, JIT-time, etc. but that would only clutter the presentation at this point. *The important issue is that in each of these phases, names have to be bound to values, and that these bindings might chance when the transitioning between binding time.*

The following $C^\sharp$ fragment defines a class `X`, and then creates instances of that class `X`, and of two other classes `Y` and `Z`:

```
class X { ... }

X x = new X ();
```

```
Y y = new Y ();

Object z =
  Activator.CreateInstance(
    Assembly.Load(
      @"A,Version=1.0.0.1,Culture=ge-CH,
      PublicKeyToken=a5d015c7d5a0b012").GetType("Z"));
```

The instance creating statements rest on a spectrum of binding times. The use of type `X` can be resolved at build time, the reference to type `Y` will be resolved at load time, while type `"Z"` is resolved at run time.

2 The need for versioning

There is a causality relation between the different binding times, viz., you have to build before you can load, and you have to load before you can run. However, it is perfectly possible that an externally referenced definition changes between build and load time, or that different versions of a program are around at different binding times.

Consider the following scenario where some vendor A builds a component R_0 which is subsequently used by vendor B to build a component S. In the mean time, vendor A releases a new version R_1 of its original product, which is then used by vendor C to build component T. Now a fourth vendor D wants to combine S and T to build component U. To make this possible, we have to:

- either guarantee that R_1 is backwards compatible with R_0 *and* have the forsight to load R_1 even if encountering a request to load R_0 first,
- or, allow R_0 and R_1 to be used side-by-side (details later),
- or, fail.

Technically, there is a fourth alternative: we can simply bind against whatever is found first and hope that things will work out ok. That is, unfortunately, the state of the art for the vast majority of systems in use today.

3 How to name components

The fact that we want both reuse beyond source sharing *and* independent extensibility [1] has several consequences on how to name components, and how names on the level of program source are mapped onto names of components.

Let's first consider the requirements on the names of components, which we will call *strong names*.

- Programmers like to refer to items by simple, readable names, so a strong name should possibly include the simple name of the component (a GUID is an example of a strong name that does not),
- There might be different versions of a component, so a strong name should possibly include version information,
- Different vendors might use the same simple name, so a strong name should possibly include owner or originator information,
- Components might be culture aware, so a strong name should possibly include culture information (as defined in IETF RFC1766; typically a pair of a language and a cultural region),
- There are many more aspects that we might want to use to distinguish between components, such as standard profiles (e.g., 'ECMA CLI Compact Profile' compliant), processor and platform constraints etc.

For example, by combining simple name, culture, unique token, and version tuple, we might get the following strong name:

```
(Name=A, Culture=ge-CH, PublicKeyToken=a5d015c7d5a0b012,
Version=1.0.0.0)
```

The choice of a token value derived from a public key is in line with the CLI definition, which calls this the *originator key*. The well-established methods to generate public/private key pair has several advantageous properties that are worth a brief mentioning here. For one, the keys are extremely likely to be unique – and therefore can replace other unique identifiers like GUIDs. In addition, when combined with the private key, the public key can serve to validate a signature. In the CLI, this signature is used to check that an assembly (the CLI's software components) has not been tampered with.

4 Names at build time

Now that we know the requirements on the strong names of components, we need to answer the question what to do with names that occur in a program source and that cannot be resolved at build time. Again, our assumptions are very much simplified, in reality the rules are much more complex and subtle taking things like overloading into account:

- One extreme solution is to leave *all* source level names unresolved until run time, this is roughly what Smalltalk does.

- The Java compiler preserves source level names at build time, and performs final name resolution at load time. In Java, a component (class, or jar file) does not include dependency information about which (version of which) external components it is compiled against.
- The $C^\sharp$ compiler uses a two-stage approach where source-level names are mapped to the strong name of a component, and a relative name within that unit. At load time, the strong name is resolved (details later), and then the relative name is resolved within the resulting component. Components do contain information about other components they are built against.
- In the .NET assembler language, MSIL, source level names are pairs of strong names and relative names to start with.
- In COM, source level names are mapped to GUIDs (the most rudimentary form of a strong name), which at run time are bound to values.

4.1 Load and Execution Time

The last questions we address is (i) how components actually contain strong names and (ii) what the execution environment infrastructure is to map strong names to components at load and exection time.

In the .NET framework, components are called *assemblies*. Each assembly contains a *manifest* that includes the following data about the assembly:

Identity An assembly's identity consists of four parts: an originator key (the public key token mentioned above), a name, a version number, and an optional culture.

File list The manifest includes a list of all files that make up the assembly. For each file, the manifest records its name and a cryptographic hash of its contents at the time the manifest was built. This hash is verified at load time to ensure that the component is consistent.

Referenced assemblies Dependencies between assemblies are stored in the using assembly's manifest. The dependency information includes a version number, which is used at load time to ensure that the correct version of the dependency is loaded.

Exported types and resources The manifest includes a list of types and resources that are made accessible to other assemblies.

Permission requests Security issues are outside the scope of this paper.

Assemblies are stored in the so called *Global Assembly Cache*. The owner of an assembly produces a digital signature by encrypting the secure hash of the assembly using its private key. A client checks the digital signature by

decrypting the signature using the public key and comparing that against the hash of the assembly. Signatures are verified when an assembly is installed into the global assembly cache. At load/execution time, when the references to imported assemblies are resolved, the public key that is stored as part of the reference to an external assembly is checked against the public key stored in the manifest of the referenced assembly.

The .NET execution environment uses the following steps to resolve an assembly reference at load or execution time:

- Determines the correct assembly version by examining applicable configuration files.
- Checks whether the assembly name has been bound-to before and, if so, uses the previously loaded assembly.
- Checks the global assembly cache. If the assembly is found there, the runtime uses this assembly.
- Tries to find the assembly using various heuristics (outside the scope of this paper).

5 Side-by-side

The execution environment can load multiple versions of an assembly side-by-side. In fact, since version information is part of strong names, this is the natural case: two versions of an assembly are really seen as two different assemblies and thus both get loaded if requests for the two versions come up. There are two important exceptions here. First, configuration rules (found in the configuration files mentioned above) can coerce a request for one version into the loading of another version. For example, a configuration file can state that version 2 is ok whenever version 1 is requested, or, more likely, that version 2 is ok for a particular client, although that client requests version 1.

The second exception are assemblies that are flagged as being unable to support side-by-side. The standard case is an assembly that represents an external unique resource - no two assemblies can represent the same resource at the same time without leading to inconsistencies. In the case of assemblies that cannot exist side-by-side in multiple versions, the only option when faced with requests for multiple versions (after considering configuration rules), is to fail all but one such request. (The first request will succeed.) For this reason it is advisable to factor out parts that cannot support side-by-side loading into special assemblies that are kept small and where it is understood that versioning has dramatic implications.

6 Conclusions and future work

We have shown how many of the issues related to reuse and independent extensibility of software components beyond source sharing can be explained as name binding problems. We believe that this provides a simple conceptual framework which will help language designers and programmers to think about this problem when they encounter it in its full gory details in the real world. With respect to future work we would like to indicate where current languages targeting the CLR tend to fall short of leveraging the versioning infrastructure of the CLR and give concrete advise on how to improve on this.

References

[1] Clemens Szyperski. Independently Extensible Systems: Software Engineering Potential and Challenge. In *19th Australasian Computer Science Conference, Adelaide, SA*, 191996.

[2] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998 (forth corrected reprinting, 1999).