Component Adaptation in Contemporary Execution Environments

Susan Eisenbach[1], Chris Sadler[2], and Dominic Wong[3]

[1] Department of Computing, Imperial College London
[2] School of Computing Science, Middlesex University
[3] Morgan Stanley, London

{S.Eisenbach@imperial.ac.uk}

Abstract. Because they are required to support component deployment and composition, modern execution environments embody a number of common features such as dynamic linking and support for multiple component versions. These features help to overcome some classical maintenance problems focused largely on component evolution, where successive generations of collaborating components need to be kept collaborating. What has been less studied has been component adaptation, whereby a component developed in an environment consisting of one set of service components is required to operate in one or several other environments containing qualitatively different components. In this paper we examine the needs developers and deployers have arising out of component adaptation and explore the concept of Flexible Dynamic Linking as a means of satisfying them. We describe a suite of tools developed to demonstrate this approach to component adaptation support within the .NET Common Language Runtime.

Keywords: component adaptation, component evolution, dynamic linking, execution environments, .NET, runtime systems.

1 Introduction

Applications based on software components offer computer users a variety of benefits including widespread utilization of robust ‘industrial-strength’ subcomponents; optimal exploitation of system resources through resource sharing and conditional loading; and potentially frequent and transparent updating. There are also benefits for the developers of the components who can continue improving and updating their products, even after their clients have taken delivery of and started to use their software.

Modern execution environments that have been built to run such applications need to embody a number of characteristic features in order to deliver these benefits. In the first place they need to be able to manage all the components. This has proved more difficult than might at first be thought and the history of recent operating systems development is sprinkled with cases where this rather obvious requirement has been inadequately accomplished. In an environment where any given component may be required by more than one application, it is essential that the component management system can deal with multiple versions of the component, since an upgrade which is beneficial to one application can easily prove disastrous to another. This phenomenon is known as DLL Hell in Microsoft[32] and is not unknown in other runtime environments[13].

The second feature that is needed for component-based support is dynamic linking, by means of which the components that an application depends on are located and loaded only at runtime and only on demand. This is how the use of system resources can be optimized. When code is compiled, information about the nature and location of external references needs to be recorded and retained with the object. In statically linked systems, the location tends to be recorded as a memory offset and all the code must be loaded together. In a dynamic linking system, the information will more likely be a symbolic reference (for example, a pathname) that can be passed to the operating system at runtime.

When these two features are combined in an execution environment, what emerges, in principle, is a powerful maintenance regime. *Component evolution* – implying that the improvements made to the next generation of one component will be automatically propagated to its existing clients – is generally well provided for in modern execution environments[15]. *Component adaptation* - porting an application from one environment to another - should not require significant intervention provided that compatible service components exist. So an application written to exploit, say, the ODBC of `SQLServer` should be able to execute with some generic ODBC without requiring an entire new build. In practice, applications are conventionally bound only to the actual components they were compiled against. The best the runtime system can do is use the symbolic references to re-locate those resources in the new (deployed) environment – so although linking is dynamic because it occurs at runtime, it is still essentially fixed. However, the redirections required to achieve both evolution and adaptation can be obtained by interfering with the symbolic reference data between compile-time and runtime. This intervention has been termed flexible dynamic linking[8] and different execution environments permit this to a greater or lesser extent.

In this paper we discuss the limitations of dynamic linking in section 2 and explore the interventions needed to achieve flexible dynamic linking in the .NET Common Language Runtime in section 3. Section 4 describes the various elements of the FLAME toolset that was developed to accomplish flexible dynamic linking to support specifically component adaptation. The paper concludes with related and future work.

2 Dynamic Linking

Dynamic Linking was first used in the MULTICS (Multiplexed Information and Computing Service) system[10]. It has found its way into many of today's programming environments including Java[17] and the .NET Framework[22] primarily as a means of satisfying the late binding requirements of modern object-oriented programming languages. The impact of dynamic linking on the efforts of software maintainers is therefore something of a side-effect. Nevertheless, component evolution has been rather well catered for by the approach taken which goes a long way to resolving DLL Hell[14]. Component adaptation has not received the same amount of attention partly because it has not been perceived of as such a big problem.

Since the dawn of Commercial Off-The Shelf (COTS) software, it has been the case that the computer system that a piece of software was developed on has not necessarily been the same as the sort of system that it eventually runs on. The developer needs to make some attempt to ensure that the software's clients' expectations of success will not be thwarted by missing or underspecified resources. The traditional method of tackling this problem consists of publishing a 'minimum specification' that the software will be guaranteed to run on.

In a component-based software environment, this approach can lead to situations where, at the majority of deployment sites, applications are bound to suboptimal resources. For example, an application might use software floating point processing on a system where floating point hardware exists. The developer's policy here is "The speed of the convoy is the speed of its slowest ship". This policy is not satisfactory for clients who have invested in higher-specification hardware or richer software resources. A generally more satisfactory approach is for developers to program to an Applications Programming Interface (API). Each client then has the obligation to provide an implementation of the API requirements as best as the system will allow. For existing component-based software environments, this involves creating or configuring components with the same signatures as those on the development system. The systematic approach to this process is termed *component adaption* (regrettably similar to component adaptation) where API mismatches between components are bridged by intermediate components, or *adaptors* [5].

However, this approach is still restrictive, as linking is constrained by compiler decisions. Compiling in a Microsoft environment will result in the expectation that `System.Console.Writeline` will come from `mscorlib`. Trying to execute the same code on a Linux system, where `System.Console.Writeline` comes from `monolib`, will result in a resolution error. Similar errors occur if the class names are not identical. The compiler has hardwired the symbolic reference with the classname and no further flexibility is possible.

In the context of this paper, another potentially confusing nomenclature is *compositional adaptation* [21] which describes a similar but essentially harder problem – the dynamic update, or hot-swapping of components *during runtime*. Considerations of these capabilities is largely focused on systems supporting ubiquitous computing [26] or autonomic computing [6].

3 Flexible Dynamic Linking

How often would the flexibility sought after in Section 2 make a difference to the applicability or portability of real-world components? This line of research was motivated by two cases where proprietary software that our components depended on could not be shipped to or otherwise accessed by some clients. In the first case a research package [20] utilized some routines derived from embargoed NASA algorithms. In order to make this tool available to a wider research community, it was necessary to embed some complex reflective code so as to effect the appropriate redirections.

In the second case an international merchant bank had developed a specialised DLL which was optimised for writing to their database. For confidentiality reasons they declined to distribute it to external software subcontractors. The subcontractors therefore had to develop using a generic database writer with no optimisation (see Table 1).

Table 1. Instead of **the** database library how can **a** database library be targeted?

Source code	Compile-time classes	Runtime classes
`New DBLib()`	`DBLib` OK	`SQLSvrLib` ??
`New DBLib()`	(None) ??	`DBLib,` OK `SQLSvrLib` OK

The idea behind Flexible Dynamic Linking is to allow the hardwiring performed by the compiler to be bypassed in some fashion. On the developer's side, this could allow for a range of alternative components to be suggested as binding partners at remote sites. On the deployer's side, it would permit the substitution of one component for another. This should make things more satisfactory in both of the real-life cases, without compromising type safety.

3.1 The Common Language Infrastructure

Like the Java Virtual Machine, the .NET Common Language Runtime (CLR) offers a managed environment for safe and secure program execution. Both systems take programs in the form of bytecode (called Microsoft Intermediate Language - MSIL - in the case of .NET). In .NET the MSIL is compiled into native code by the runtime just before it is executed whereas Java bytecode is normally interpreted. One of Java's strengths is its platform independence and at first glance it would seem the .NET Framework is missing this valuable attribute. However Microsoft has released its specification and it was standardised by the European Computer Manufacturers Association (ECMA). ECMA-335[16] defines the Common Language Infrastructure (CLI) where applications written in different languages can be run on differing systems without the need to take into account the characteristics of that environment.

The central store for shared libraries to be used by the CLR is called the General Assembly Cache (GAC). Microsoft's .NET Framework *assemblies* (Microsoft's term for components) are placed here for shared access. Only globally unique assemblies are allowed to be shared and installed into the GAC, all others are considered to be private, not trusted for sharing, and are usually kept within the application folder. Fusion is the assembly loader that handles the dynamic linking within the CLR and it is invoked whenever a reference to an external assembly is made.
Three important open source implementations of the ECMA-335 standard are Mono[31], DotGNU[11], and Rotor (Microsoft's own Shared Source Common Language Infrastructure (SSCLI)) [24,29].

3.2 Definition

Dynamic linking allows the linking at runtime to a class that was identified at compile-time. Flexible Dynamic Linking defers the decision of which class to link to *until* runtime when the linker will make the final decision. This serves to decouple the runtime environment from the compile time environment. Flexible Dynamic Linking, as set out in [8], achieves this by using *type variables* instead of class names in the bytecode generated during compilation. A type variable is a placeholder for a type. At runtime the decision on which type is used as a substitute is taken by the linker normally based on some predefined policy. For example, consider:

```
    public class Class1
    {
     static X list;
      public static void Main(string[] args)
      {
    list = new X(); list.Add(“foobar”);
      }
}
```

The type variable `X` is a placeholder for a real type. This will be compiled into the bytecode and when it comes to executing the code the linker will recognise this as a type variable and make a decision as to which type it should substitute in its place. In theory, as long as the chosen substitute has an empty constructor and has the method `Add(String s)` then it will execute without error. This conception of linking can assist component adaptation since creating platform independent code is simply a matter of using type variables and ensuring that there is a type on the target platform which provides the same interface as that being used by the type variable. The same applies to utilising DLLs which are known to be on the target system.

However, when we come to apply this strategy to .NET there is a slight modification which is needed due to the way in which external types are referenced in MSIL bytecode. Consider the following “Hello World” program in .NET:

```
.method private hidebysig static void Main(string[] args)
cil managed
   {
     .entrypoint
     .maxstack  8
     IL_0000: nop
     IL_0001: ldstr "Hello World"
     IL_0006:call void[mscorlib]
       System.Console::WriteLine(string)
     IL_000b: nop
     IL_000c: ret
   } // end of method Program::Main
```

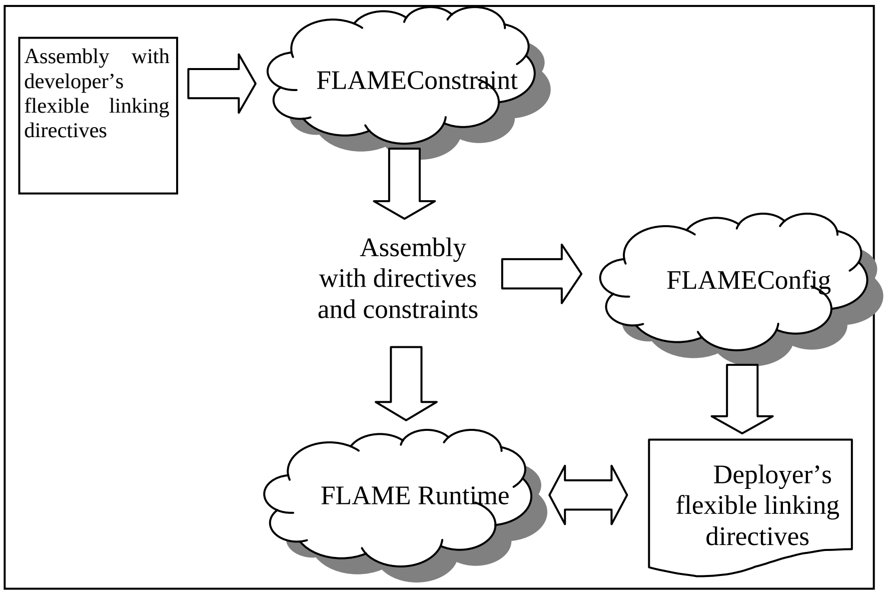
The reference to the type `System.Console` is tagged with the assembly in which it is found, `mscorlib`. As a consequence of this, every type variable which we generate for the bytecode must be represented in two parts; an assembly type variable and a class type variable.

4 FLAME

The tool described in this paper is named FLAME. It is based on CUPID[1] an implemention of Flexible Dynamic Linking that was designed so as to give developers the ability to indicate compatible substitutions at both the class and assembly levels. CUPID implements *logical* type variables – metadata inserted into the bytecode that tags specific classes and assemblies as potentially variable. CLIs that cannot interpret the metadata can execute the assembly as normal, linking to the original build references. The metadata is created via the use of *custom attributes*. These allow the developer to define the assemblies/classes to be replaced, what to replace them with, and some other linking options. A risky alternative would be to allow *any* assembly which provides the correct API (called a *binary compatible* assembly) to be a possible substitution candidate.

CUPID ensures type safety by analysing the bytecode of the application and automatically generating appropriate *member constraints* to be inserted. Member constraints specify all class/field accesses that the substitute member must satisfy during execution of the program. CUPID also allows the (manual) specification of *structural constraints* - ensuring that, if there is a supertype-subtype relationship between two classes, then whatever type replaces the supertype must be a supertype of the type that replaces the subtype.

The FLAME system was designed to automate the specification of the structural constraints for the developer and then to develop a deployer-centric solution. To achieve these two goals we have three distinct components; the FLAMEConstraint tool, the FLAME runtime and the FLAMEConfig tool. Fig. 1 shows how the three components are related.


Assembly with developer's flexible linking directives
FLAMEConstraint
Assembly with directives and constraints
FLAMEConfig
FLAME Runtime
Deployer's flexible linking directives

Fig; 1: Architectural overview of FLAME.

In the CUPID system member constraints are generated by a Perl script (dubbed `flxibl`). In order to improve efficiency `FlameConstraint` utilises the Phoenix compiler framework [23] to provide the basis for a new post-compilation tool which will generate both member and subtype constraints. Two attributes, `LinkAssembly` and `LinkClass`, are used to create linking directives attached at the appropriate scope: assembly, module, class or method. The constraints for substitute assemblies and classes are derived from these directives and are then inserted into the bytecode, again at the appropriate scope level. The constraints are defined using two custom attributes, `LinkMemberConstraint` and `LinkStructureConstraint`.

LinkAssembly Attribute. A `LinkAssembly` attribute redirects all class references, within a given scope, from its original assembly to a new one by essentially replacing the original assembly name with a new one. The `LinkAssembly` attribute has parameters that fully describe the original and new assembly.

LinkClass Attribute. The `LinkClass` attribute does for classes what `LinkAssembly` does for assemblies. However, since a class reference includes both the assembly and class names a `LinkClass` attribute must have a corresponding `LinkAssembly` attribute that contains the same `InterfaceName`.

LinkMemberConstraint Attribute. When we substitute one class for another, the new class must provide all of the method calls and field references that the program makes on the old class. These required methods and fields are called member constraints and are expressed through the `LinkMemberConstraint` attribute.

LinkStructureConstraint Attribute. The types referenced in a program have a complex set of subtype and supertype relationships. Among other things, subtypes are often used in place of supertypes as arguments to method calls and subtypes can be cast to one of their supertypes for further manipulation. Any new classes introduced as substitutes must satisfy the subtype and supertype relationships as the classes they replace. These relationships are expressed as `LinkStructureConstraint` attributes.

To clarify the usage of the attributes and what `FLAMEConstraint` does with them consider the following code:

```
[LinkAssembly(“System.Windows.Forms”, “SpecialForms”,
“1.1.*”, null, null, true, “special”,
InterfaceType.LOCAL_INTERFACE)]
[LinkClass(“System.Windows.Forms.Form”, “BlueForm”,
“special”)]
public static void Main {
Form f = new Form();
f.Show()
Form d = new MDIWindowDialog();
}
```

The use of the two attributes `LinkAssembly` and `LinkClass` describe a single flexible linking directive which redirects all references to the `System.Windows.Forms.Form` class (which has been defined in the `System.Windows.Forms` assembly) to the `BlueForm` class (defined in the

SpecialForms assembly). When this code is parsed by the FLAMEConstraint tool it generates member and subtype constraints based on the usage of all instances of the System.Windows.Forms.Form and results in the augmented code given below:

```
[LinkAssembly(“System.Windows.Forms”, “SpecialForms”,
“1.1.*”, null, null, true, “special”,
InterfaceType.LOCAL_INTERFACE)]
[LinkClass(“System.Windows.Forms.Form”, “BlueForm”,
“special”)]
[LinkMember(“System.Windows.Forms”,
“System.Windows.Forms.Form”, “Application1.exe”,
“100663300”, false)]
[LinkMember(“System.Windows.Forms”,
“System.Windows.Forms.Form”, “Application1.exe”,
“100663323”, false)]
[LinkStructure(“System.Windows.Forms”,
“System.Windows.Forms.Form”, “100782403”,
“System.Windows.Forms”,
“System.Windows.Forms.MDIWindowDialog”, “1008392532”,
“Application1.exe”)]
public static void Main {
  Form f = new Form();
  f.Show()
  Form d = new MDIWindowDialog();
}
```

The FLAMEConstraint tool has generated LinkMember constraints which specify that the replacement must provide the constructor and Show() methods, although this is hard to see since they are referred to only by metadata token numbers (for example “100663300”). A subtype constraint, in the form of a LinkStructure attribute, says its replacement must be a supertype of the MDIWindowDialog type.

4.2 FLAME Runtime

The *application configuration file* is an XML file which resides in the application’s directory and is named <applicationName>.exe.config. Under the normal .NET runtime when the application is run, execution will proceed as normal until an external type is referenced. Fusion will then find the referenced type’s enclosing assembly and load it into the runtime. .NET *strong-name* assemblies are identified by name, a public key ID, a ‘culture’ and a four-part version number. The first time that Fusion is invoked it searches the application directory for a corresponding application configuration file. If one is found, it will parse the XML and cache the information for future reference. Whenever Fusion receives an assembly load request it will first consult its cached application configuration file to see whether the assembly is subject to a version redirect and if so it will attempt to load the specified version else it will load the originally requested version. A typical binding redirection looks like this:

```
<assemblyIdentity name=”TestLibrary1”
                  publicKeyToken=”9D9229CF9B3C922D”
                  culture=”neutral”
  />
  <bindingRedirect oldVersion=”1.0.0.0”
                   newVersion=”2.0.0.0”
  />
```

To specify our flexible linking directives in FLAME we extended the existing `<bindingRedirect>` tag of the application configuration file so that we can describe a new assembly. This means accommodating the name, culture and public key token of the new assembly. Thus:

```
<bindingRedirect interfaceName=”macosx”
                 interfaceType=”ANY_INTERFACE”
                 oldVersion=”1.0.0.0”
                 newVersion=”2.0.0.0”
                 newAsmName=”TestLibrary2”
                 newPublicKeyToken=”9B9287CC6B3C809A”
                 newCulture=”neutral”
  />
```

This redirects all references from TestLibrary1 to TestLibrary2. This means that TestLibrary2 must define all of the types which TestLibrary1 offers and which are referenced in the application otherwise we will find a type load exception at runtime. We also need the capacity to redirect individual types within an assembly. This is achieved through `varClass` and `newClass` attributes of the `<bindingRedirect>` tag.

To carry out the deployer defined flexible linking directives in FLAME we could create and insert metadata into the assembly’s bytecode to describe the substitutions. This would involve invoking a tool before the code is executed to modify the original assembly with some new metadata. The underlying runtime would not have to be touched because in essence it is performing the same steps as the FLAMEConstraint tool with two major differences:

(i) The metadata would be generated from a given list of substitutions, not from custom attributes.

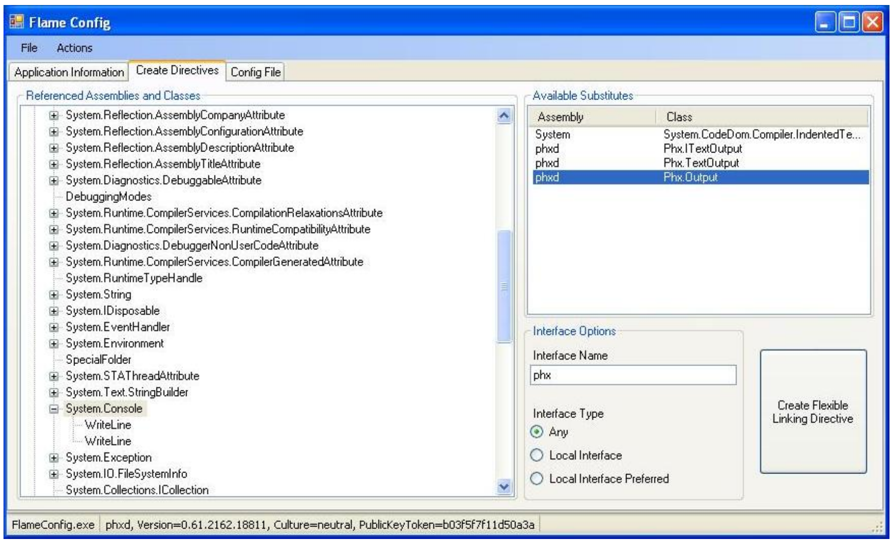
(ii) The bytecode changes would occur just before runtime at the deployer side, instead of occurring just after compilation at the developer side.

Unfortunately, to modify the metadata requires the assembly to be disassembled and then reassembled, and if the original assembly was signed with a private key by the developer it would need to be resigned when it was reassembled. The deployer would not be in possession of this key so would be unable to re-sign the assembly thus restricting usage to unsigned applications.

Therefore it is necessary to modify the runtime directly so that it can parse the additional binding redirection XML and then act upon it. The enhanced FLAME runtime does not check constraints on any types that it flexibly links. This means that after loading a substitute assembly/class it is possible that the runtime will not be able to load the required type or invoke the required method.

One possible solution is to use the application configuration file for storing the constraints, but this has two main drawbacks. First of all, XML is a very verbose representation format and representing a single member or subtype constraint takes several lines of XML. A reasonably sized application with a large number of constraints would end up with an extremely bloated application configuration file. Secondly, the application configuration is usually edited by hand which makes it very easy for someone to accidentally remove or alter a constraint.

A further reason for not incorporating runtime constraint checking is the potential performance decrease when verifying a large number of constraints. Member constraints are quite fast to check since it is only querying the existence of a method or field in the loaded class. However, subtype constraints can potentially take much longer. Consider a type T1, defined in assembly A1, with a subtype constraint which says that whatever replaces T1 must be a supertype of type T2. To check this constraint we must load type T2, which is defined in assembly A2, and then check the relationship between the two types. Unfortunately type T2 is also subject to flexible dynamic linking, it is to be replaced by type T3. So we must now also verify that T3 satisfies all of T2's constraints. Loading these types from the different assemblies, which may not be required during the run, causes delays in the execution and also increases the memory footprint of the running application.



Referenced Assemblies and Classes

```
⊞ System.Reflection.AssemblyCompanyAttribute
⊞ System.Reflection.AssemblyConfigurationAttribute
⊞ System.Reflection.AssemblyDescriptionAttribute
⊞ System.Reflection.AssemblyTitleAttribute
⊞ System.Diagnostics.DebuggableAttribute
  DebuggingModes
⊞ System.Runtime.CompilerServices.CompilationRelaxationsAttribute
⊞ System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
⊞ System.Diagnostics.DebuggerNonUserCodeAttribute
⊞ System.Runtime.CompilerServices.CompilerGeneratedAttribute
  System.RuntimeTypeHandle
⊞ System.String
⊞ System.IDisposable
⊞ System.EventHandler
⊞ System.Environment
  SpecialFolder
⊞ System.STAThreadAttribute
⊞ System.Text.StringBuilder
⊟ System.Console
    WriteLine
    WriteLine
⊞ System.Exception
⊞ System.IO.FileSystemInfo
  System.Collections.ICollection
```

Available Substitutes

Assembly	Class
System	System.CodeDom.Compiler.IndentedTe...
phxd	Phx.ITextOutput
phxd	Phx.TextOutput
phxd	Phx.Output

Interface Options

Interface Name: phx

Interface Type: [✓ Any] Local Interface / Local Interface Preferred

FlameConfig.exe | phxd, Version=0.61.2162.18811, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a

Fig. 2.Screenshot from the FLAMEConfig tool

4.3 FLAMEConfig

Without storing a great deal of semantic information, it is not feasible to perform constraint verification automatically at runtime, so it is essential to ensure that any substitute assembly identified in a flexible linking directive will be binary compatible with the application. `FLAMEConfig` is an interactive tool which is designed to achieve the required type-checking in an intermediate step taken at the deployment site. The operation of `FLAMEConfig` is as follows:

(i) The application for which flexible linking directives are to be created is loaded into the tool.

(ii) A list of all the assemblies and classes referenced within the loaded application is displayed to the user. (If the assembly is missing for some reason `FLAMEConfig` will inform the user.)

(iii) The user picks the assembly/class they wish to flexibly link and the list of possible substitute assemblies/classes is displayed to the user.

(iv) The user chooses the substitute from the list and defines what interface type and name they want for the directive. (see Fig. 2)

(v) Finally, the tool creates the appropriate XML to express the flexible linking directive and adds it to the application configuration file.

The list of possible substitutes is generated by examining the GAC and local application folder for every assembly. An assembly/class is then added to the list of eligible substitutes if it can satisfy the member and subtype constraints inferred from the selected referenced assembly/class. Provided that the application configuration file is not manually edited subsequent to this step, the flexible linking directives are guaranteed to substitute binary compatible assemblies/classes (as long as the execution environment does not change).

The three components of the FLAME system combined with CUPID make a complete system for flexible dynamic linking, enabling both developers and deployers to control the flexible linking process. Deployer-defined directives are located in the application configuration file whilst developer-defined ones are embedded in the assembly metadata. Thus there is no danger that they will conflict syntactically, so to speak. In circumstances where they conflict semantically, it is the deployer-defined directive that takes precedence.

4.4 Case Study: xmlValid

The FLAME system was tested on a real-world application called `xmlValid` - a simple command line XML validation tool[30] which checks whether an XML file is well formed and validates it against a given XSD file.

The `xmlValid` assembly references two external assemblies; `mscorlib` and `System.Xml`. The class `System.Xml.XmlTextReader` was chosen as the target for flexible dynamic linking. A new class, `MyXml.MyXmlTextReader` was developed as a binary compatible replacement. We ran timing tests to gauge the performance difference, the results of which are presented in Table 2.

Table 2. Execution times of with and without flexible linking

Run	Normal Time (s)	Flex Linked Time (s)	Difference (s)
1	9.51	10.12	0.61
2	9.17	10.08	0.91
3	9.78	10.00	0.22
4	9.14	9.98	0.84
5	9.10	9.93	0.83
6	9.24	10.23	0.99
7	9.07	10.01	0.94
8	9.12	10.29	1.17
9	9.16	10.60	1.44
10	9.20	9.92	0.72
Average	9.25	10.12	0.87

Flexible dynamic linking added an average 0.87 seconds or around a 9.4% increase in execution time using a test input file. Since (typically) larger XML files would take longer to validate, this overhead could be expected to fall. So the performance cost for having flexible dynamic linking does not seem unacceptable.

5 Related and Future Work

The idea of keeping types unspecific at compile-time by means of type variables has been examined in several programming communities [28,18,3]. In the meantime, linking-time behaviour, both for .NET and for the Java Virtual Machine has received some formal attention [2, 12, 7].

The current work is built on a number of earlier projects, focused initially on component evolution [14], which anticipated the .NET 2.0 introduction of type forwarders [19]; and then on component adaptation [9,1]. Execution environments that support the runtime interpretation of metadata, in conjunction with pertinent configuration files, are bound to receive increasing attention [25,27,4].

A number of future extensions to the FLAME toolset itself are possible. Instead of asking the developer or deployer to choose replacement assemblies or classes, an enhanced runtime could make the decision based on some heuristics. The heuristics used to decide which substitution is most appropriate would have to be based on the properties of the assembly.

The Phoenix framework offers a rich toolset for dataflow analysis and generation of member and subtype constraints could be based on dataflow information. Those referenced methods and fields and subtype relationships which applied during a typical run of the program could be used to constrain the possible replacement assembly.

Application configuration files are not the only files that the Fusion checks for binding information. The machine configuration file redirects the loading of particular assemblies for every executable run on that machine. The schema for the machine configuration file is identical to that for the application configuration file so modifying FLAME to extend flexible linking to this file should not be particularly difficult. Finally developers could distribute application configuration files directly with their programs, then these could be fed into the FLAMEConfig tool at the deployer end to verify that they obey the member and subtype constraints.

The main goal of this project was to provide a method for the deployer to specify any assembly or class which should be subject to flexible dynamic linking and to ensure that it will be carried out in accordance with all the directives and binary compatibly. Additionally the tools to help the developer were improved. The Flame toolset lets the developer suggest and the deployer choose different assemblies and classes than were available in the compilation environment. We have developed our toolset on .NET because it had metadata which made the implementation reasonably straightforward. However, we believe that the ability to do component adaptation should be more widely applicable.

Acknowledgments. The software described in this paper was inspired by Alex Buckley's PhD thesis work on Flexible Dynamic Linking. We would like to thank him for all his help in the development of Flame.

References

1. Aaltonen, A., Buckley A., Eisenbach S.: Flexible Dynamic Linking for .NET. Journal of .NET Technologies, Vol 4. June (2006).
2. Abadi M., Gonthier G., Werner B.: Choice in Dynamic Linking. Proceedings of the 7th International Conference FOSSACS 2004 (ETAPS 2004), Vol. 2987 of LNCS, Barcelona, Spain, March (2004).
3. Ancona D., Damiani F., Drossopoulou S., Zucca E.: Polymorphic Bytecode: Compositional Compilation for Java-like Languages. in ACM SIGPLAN-SIGACT Symposium on Principles of Progamming Languages. (2005). Long Beach, California.
4. Attardi G., Cisternino A., Colombo D.: CIL + Metadata > Executable Program. Journal of Object Technology, Special issue: .NET: The Programmers Perspective: ECOOP Workshop (2003).
5. Bracciali A., Brogi A., Canal C.: A formal approach to component adaption. In J. Syst. Softw., Vol. 74(1), (2005).
6. Bialek R., Jul E., Schneider, J-G., Jin y.: Partitioning of Java Applications to Support Dynamic Updates. In 11th Asia-Pacific Software Engineering Conference (APSEC'04), (2004).
7. Buckley, A. A Model of Dynamic Binding in .NET in ECOOP Workshop on Formal Techniques for Java-like Programs. (2005). Oslo, Norway.
8. Buckley, A., Drossopoulou S.: Flexible Dynamic Linking. in ECOOP Workshop on Formal Techniques for Java-like Programs. (2004). Oslo, Norway.
9. Buckley, A., Murray M., Eisenbach S., Drossopoulou S.: Flexible Bytecode for Linking in .NET in ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation. (2005). Edinburgh, Scotland.
10. Corbato, F.J., V.A. Vysssotsky.: Introduction and Overview of the MULTICS System. AFIPS Fall Joint Computer Conference. (1965).
11. DotGNU Project: Available from: http://dotgnu.org/.
12. Drossopoulou, S., Lagorio G., and Eisenbach S.: Flexible Models for Dynamic Linking. in European Symposium on Programming. (2003). Warsaw, Poland.

13. Eisenbach, S., Jurisic V., Sadler C.: Feeling the Way Through DLL Hell. in First Workshop on Unanticipated Software Evolution. (2002). Malaga, Spain.
14. Eisenbach, S., Kayhan D., Sadler C.: Keeping Control of Reusable Components in International Working Conference on Component Deployment. 2004. Edinburgh, Scotland.
15. Eisenbach, S., Sadler C.: Reuse and Abuse. Journal of Object Technology. Vol 6. 1 January, 2007 ETH Swiss Federal Institute of Technology.
16. ECMA International: Standard ECMA-335 Common Language Infrastructure (CLI). (2005) Available from: http://www.ecma-international.org/publications/standards/Ecma-335.htm.
17. Gosling, J., Joy, B, Steele, G, Bracha, G.: Java(TM) Language Specification, Second Edition, Addison Wesley, (2000).
18. Kennedy, A., Syme D.: Design and Implementation of Generics for the .NET Common Language Runtime. in ACM SIGPLAN Conference on Programming Language Design and Implementation. (2001). Snowbird, Utah, USA.
19. Lander R.: The Wonders of Whidbey Factoring Features. Part 1: Type Forwarders. Available from http://hoser.lander.ca/ 14 Sept. (2005).
20. Magee, J. and Kramer, J.: Concurrency : state models & Java programs Chichester, England. Wiley, (2006).
21. McKinley P., Sadjadi S.M., Kasten E.P., Cheng B.H.C.: A Taxonomy of Compositional Adaptation in Software Engnieering and Network Systems Laboratory Technical Report MSU-CSE-04-17, (2004).
22. Microsoft Corporation: Microsoft Developer Network. Available from: http://msdn.microsoft.com.
23. Microsoft Corporation. Phoenix Documentation. (2005) Available from: http://research.microsoft.com/phoenix/.
24. Microsoft Corporation. SSCLI Documentation. (2002) Available from: http://research.microsoft.com/sscli/.
25. Mikunov A.: Rewrite MSIL Code on the Fly with the .NET Framework Profiling API. MSDN Magazine, September (2003).
26. Paspallis n., Ppapadopoulos G.A.: An approach for Developing Adaptive, Mobile Applications with Separation of Concerns in Proc. COMPSAC'06), (2006).
27. Piessens F., Jacobs B., Truyen E., and Joosen W.: Support for Metadata-driven Selection of Run-time Services in .NET is Promising but Immature. Journal of Object Technology, Special issue: .NET: The Programmers Perspective: ECOOP Workshop (2003).
28. Shao Z., Appel A.W.: Smartest Recompilation. Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), Charleston, South Carolina, USA, (1993).
29. Stutz, D., Neward, T., Shilling, G.: Shared Source CLI Essentials. O'Reilly. (2003).
30. Sells, C.: .NET and Win 32 tools. available from :http://www.sellsbrothers.com/tools.
31. What is Mono? Available from: http://www.mono-project.com/Main_Page.
32. Wong, F.: DLL Hell, The Inside Story. (1998) available from: http://www.desaware.com/tech/dllhell.aspx