Keeping Control of Reusable Components

Susan Eisenbach[1], Dilek Kayhan[1], and Chris Sadler[2]

[1] Department of Computing
Imperial College
London, UK SW7 2BZ
[sue, dk02]@imperial.ac.uk
[2] School of Computing Science
Middlesex University
London, UK NW4 4BT
c.sadler@mdx.ac.uk

Abstract. Development and deployment via components offers the possibility of prolific software reuse. However, to achieve this potential in a component-rich environment, it is necessary to recognize that component deployment (and subsequent composition) is closer to a continual process than a one-off operation. This is due to the requirement that newly-evolved components need to replace their ancestors in a timely and efficient manner at the client deployment sites. Modern runtime systems which employ dynamic link-loading mechanisms can permit such *dynamic evolution*. We review the capabilities of several alternative runtime environments to establish some requirements for dynamic evolution. Then we describe a tool designed to support developers and administrators in the migration of component updates within the Microsoft .NET framework.

1 Introduction

In simple terms, the primary aim of software reuse is to allow for specialist developers to make their software available to more general (applications) developers in multiple and diverse contexts. To achieve this it is necessary to package each software artefact such that it is *composable* — it is capable of interoperating with other artefacts; and *deployable* — it can be installed independently in a runtime environment where it can be composed, on-the-fly, with other artefacts. These requirements bring the 'reusable software artefact' very close to the conventional definition of a software component [30] and in this paper we shall take reuse as the raison d'etre of the software component.

The component model of software envisions an application as a collection of collaborating components emanating from different developers, probably from different vendors. Over time it is to be expected that each component will undergo evolutionary adaptation (or maintenance). In a regime where applications are statically linked (the interoperability requirements of all components are fully resolved at build-time) evolutionary adaptation implies that the functionality and performance of the application may depend crucially on precisely when the build-time occurred. However, in a regime where applications are dynamically linked, they may be able to benefit from evolutionary adaptations which their components have undergone subsequent to their own build-time. This phenomenon can be characterised as *dynamic evolution*.

Nobody who has written software for use by other developers and who has had to maintain that software for any period or for any reasonably-sized client population would argue that the promise of dynamic evolution is not a better alternative than anything else on offer. Nevertheless, experience of those regimes which implement dynamic linking mechanisms shows that it has not been easy to live up to that promise.

In section 2 we review these (dynamic linking) mechanisms as implemented in a number of modern runtime systems before considering how dynamic evolution can be assured. Section 3 describes the development of SNAP, our prototype tool for system administrators and .NET application developers. In sections 4 and 5 related and future work have been outlined.

2 Dynamic Evolution

2.1 The Component Object Model

Dynamic evolution imposes a responsibility on the runtime system to locate required components in a timely fashion, whether they be memory-resident or in need of prior loading. To enable efficient loading and linking, systems (like COM [24]) have adopted a *registry* mechanism. The Registry is a data structure which stores essential information (for example, component locations) needed to facilitate rapid component interoperation. To improve efficiency, each component (or *dynamic link library* — DLL) has at least two entries in the Registry, one indexed by its *class identifier* (CLSID), and the other by its *programmatic identifier* (ProgID). There is an access method DllRegisterServer() to create entries in the Registry. Microsoft encourages third party developers to utilise this method to ensure that their installation will be successful, viz.

> "Most setup programs call DllRegisterServer as part of the installation process." [24]

Version control is implemented in COM by allowing ProgID entries for components to include a version number field (so that it is theoretically possible to register multiple versions of a component). However, there is some evidence that this is not taken very seriously, since the Registry also supports a version-independent *Current Version* (CurVer), and —

> "In most cases the client doesn't care which version of the component it connects to (so) a component usually has a ProgID that is version independent (that) maps to the latest version installed on the system." [24]

This way of managing components can give rise to a number of problems. In the first place it can be difficult to maintain the integrity of the registry: correctly formulating multiple entries for a single entity is never straightforward and any 'uninstallation' that is done relies on the an *explicit* activation of DllUnregisterServer(). Arbitrarily deleting DLLs will also compromise the integrity of the Registry.

Secondly, as we have seen, most client applications routinely register their own DLLs (that is, the versions of the components they were built against) upon installation, and routinely load the CurVer version of the component whenever they are launched.

Thus the only active version of the component will be overwritten practically every time a new client is installed. If it is truly the latest version (in absolute historical terms, rather than just the most recently registered) then there may be older applications which suffer an *upgrade problem* [12]. Most component developers can help users to avoid this by maintaining backwards compatibility, at least for a few generations. Equally, the *latest* version could be an historically earlier version that the application just happened to be built against. In this case, previously-built clients may experience the *downgrade* problem [12]. Component developers can only protect users here by maintaining *forward* compatibility, which is a bit like being able to foretell the future. In either case, the user risks entering the gates of DLL Hell [2,23,22,11].

Even though the COM architecture does not dictate that distinct versions of a component cannot be implemented, it does not make it very easy to impose a rigorous version control system which will support dynamic evolution. Each client can either be loaded alongside the version it was built against (and never evolve) or can be loaded alongside the CurVer version and take pot-luck on DLL Hell. So CurVer is one of the causes of DLL Hell. It is unlikely that any single version of a well-utilised component can satisfy all clients. Instead, for each individual client, there is likely to be a 'latest' version which satisfies its service requirements and which may or may not be the most recently produced (or installed) version. It is difficult to see how this kind of information can be compactly and reliably stored in a Registry.

There seem to be two things about COM which make dynamic evolution turn into DLL Hell. The first is the separation of the component from its *metadata* as recorded in the Registry. Since client applications build and link with reference to the Registry, the build *history* of each client — the record of the actual versions of the components it was built against — becomes lost as the Registry is updated. The second is the relative difficulty of maintaining multiple versions of a component in such a way that differing client requirements can be simultaneously satisfied.

2.2 The Java Virtual Machine

A different approach to dynamic linking has been used in the Java Virtual Machine (JVM). Here the compiler embeds 'fully qualified' path references to service components directly into the object code. Instead of using a registry therefore, each application and each component carries information about the compile-time location of its dependencies. At runtime the classloader recursively loads the full class hierarchy in order to enable *late binding*. If evolved versions occupy the same (or hierarchically lower) locations in the classpath, dynamic evolution will occur automatically. However, all the clients of the superclass will access the same (*current*) version of the class or method, and we are right back in DLL Hell, unless the user is somehow prepared to define distinct classpaths for each application.

This cumbersome solution is unlikely to be feasible, so we designed DeJaVue, a distributed tool [26] which allows a Java library developer to export a custom classloader to clients. Applications which invoke this classloader at runtime can immediately benefit from compatible server-side component updates at the cost of a little network traffic when the application is launched. The server maintains a repository of all versions of each component and, using information about the last version downloaded to that

client, it will automatically update the client with the most recent version that is *binary* compatible, even though more recent (but *incompatible*) generations may exist in the repository.

At this point it might be worth digressing to consider binary compatibility [17,8]. When one component is composed with another, some of the services it exports are imported by its clients. When the first component evolves, it will be composable with its client provided that none of the services required by the client have been removed or changed. In this case, the original and the evolved component are said to be binary compatible *relative* to that particular client. They will be *absolutely* binary compatible if they are binary compatible relative to all possible clients of the original service components. Most interesting evolution involves either expanding the export interface so that new clients can be accommodated, or modifying the behaviour of the services. The first kind of evolution should not make any difference to the effect of services on existing clients. However, the second kind does and leads to the distinction between *syntactic* binary compatibility (where components will link without error but the composed behaviour may be different) and *semantic* binary compatibility where the behaviour remains the same [30]. Throughout this paper we are only concerned with syntactic binary compatibility.

In the current implementation of DeJaVue [3], if the user site has two or more applications that utilise the same library, then each application needs to maintain its own copy of each component, so some of the benefits of reuse are lost. In addition, any application that utilised components from several libraries would require a separate classloader for each one, which makes application startup rather slow. Finally, any library developer who had a substantial client base would need to devote quite a lot of computing resources at the server-side to satisfying the update requirements of each client application every time it was launched. Although this might provide a solution for specialist Java library providers with reasonably limited clientele, a more general solution to the problem requires an approach focussed exclusively on the client side - but one which can bypass the problems of the COM registry.

2.3 Microsoft .NET Framework

.NET is a framework devised by Microsoft to promote the development of component-based applications and to enable their efficient and effective deployment [23,10]. The core component of .NET is the Common Language Runtime (CLR) which is actually a runtime environment. The CLR accommodates the interoperation of components rendered into the Common Intermediate Language (CIL). Rather than interpreting these statements for a virtual machine (as in the Java virtual machine setup) the CLR uses a Just-in-Time (JIT) compiler to generate momentary native code for the local platform. Having a *common* intermediate language means that applications can compose components written in different source languages (see Fig. 1) whilst the Platform Adaptation Layer (PAL), together with the Framework Class Library, makes .NET applications potentially highly portable across Windows platforms.

A number of the .NET design goals [29] are particularly relevant to dynamic evolution:

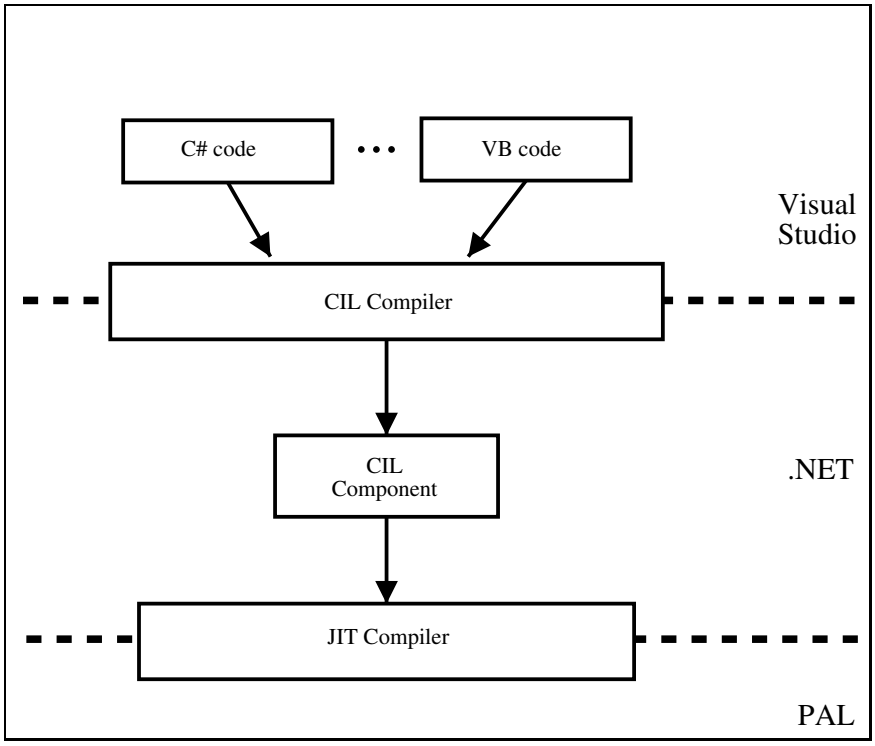

Fig. 1. .NET Framework Code Pathway

1. "Resolve intertype dependencies at runtime using a flexible binding mechanism." This is what makes it *dynamic*.
2. "Design runtime services to . . . gracefully accommodate new inventions and future changes." This is what is meant by *evolution*.
3. "Package types into portable, self-describing units." 'Portable' means that the packages are effectively components, as defined earlier, and in .NET are referred to as *assemblies*. The 'self-describing' means that the CLR need not depend on a registry to compose components at runtime. Instead, each assembly incorporates a *manifest* which lists the resources provided by the assembly and the nature and locations of any resources it depends on from external assemblies.
4. "Ensure isolation at runtime, yet share resources." This implies a DeJaVue-type repository that can hold multiple versions of a component and a runtime system that can correctly select and use the different versions appropriate for each application. In .NET this repository is known as the Global Assembly Cache (GAC) and the runtime capability is referred to as 'side-by-side' operation. The runtime address space is divided into Application Domains (*appdomains*). At runtime an Assembly Loader loads assemblies from secondary storage (or a URL) into the appropriate appdomain on demand. Assemblies loaded from the GAC are loaded into

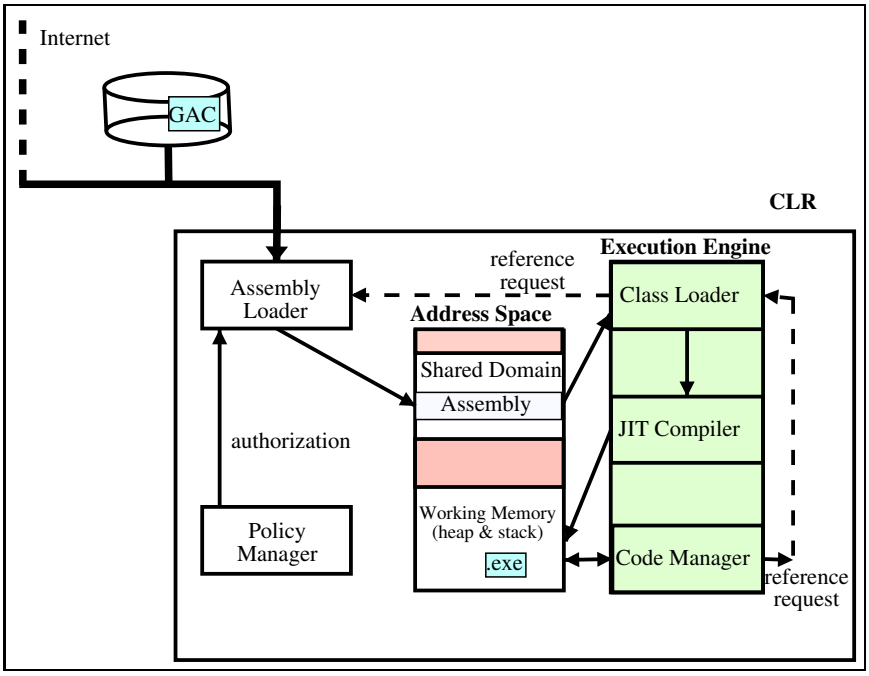

Fig. 2. CLR Loading

the Shared Domain where they are accessible to any running application. Other domains exist for system services and application-specific classes. Security checks are performed during the loading process.

5. "Execute code under the control of a privileged execution engine . . . ". The CLR execution engine in Fig. 2 performs 'managed execution' of code produced by the JIT compiler. When an executing object references an unloaded class, the Class-Loader loads the class from the appdomain for compilation. If the reference is to an unloaded assembly, the request is passed to the Assembly Loader. Code verification is performed during JIT compilation.

A .NET assembly that is capable of entering the GAC and participating in side-by-side operations must be identified by a *strong-name*. A strong-name incorporates a name; a four-part version number divided into <major>, <minor>, <build> and <revision> parts; a 'culture' and a public key ID originating from the assembly author. Any two assemblies are regarded as distinct if there is a variation in any one of these. Thus even if two providers simultaneously update the same assembly with the same version number increment, the public key information still allows the system to distinguish between them. Assemblies which are not in the GAC (and hence are not intended for side-by-side operation) need not be strong-named.

The CLR can handle the runtime code so flexibly and effectively (as illustrated in Fig. 2) because of the use it makes of the descriptive metadata embedded in each assem-

bly. Thus the Assembly Loader uses dependency metadata from one assembly to locate and authenticate a required assembly for loading. It also uses the target assembly's own metadata to bind its services to the appdomain. Likewise the ClassLoader uses assembly metadata to construct type descriptors and method tables for the runtime layout and for type-checking within the JIT compilation process.

Compared with other runtime systems we have studied, .NET appears to have a lot to offer for dynamic evolution. The reflective potential of the available assembly metadata allows us to escape the registry difficulties of COM; strong-names provide the necessary multiple versioning missing from the JVM model; the GAC gives a client-side DeJaVue-style repository; and JIT-compilation makes binding as late as possible[18]. So .NET looks like a good candidate for rational component management.

3 Rational Component Management

The strong-name assemblies in the GAC, which are so uniquely specified, are equally uniquely referenced in the metadata of dependent (client) assemblies. Thus, every time a client runs, it will only ever request the exact service assembly it was built against. Any improvements arising in future versions will by default be lost to the client until its next rebuild. This is deliberate:

> "Historically, platform vendors forced users to upgrade to the latest version shipped. Software developers ... were responsible for resolving any resulting incompatibilities" [21]

and the result was DLL Hell!

Instead, the default behaviour can be overridden because the Assembly Loader consults a sequence of XML 'policy' files which can be used to redirect the load operation. So

> "the .NET Framework team (puts) complete control in the hands of system administrators and developers who use the framework ..." [21].

Thus one policy file is the Application Configuration file and another is the Machine Configuration file, and these give some (manual) control to the client-side system administrator. On the developer side, there is a Publisher Policy file. Taken together, these files can no doubt provide a dynamic evolutionary pathway for any application provided that all parties with write-access to the policy files have full information about component dependencies and versions (that is, they know the information in the metadata and will act on it).

We wanted to provide this functionality in a way which could be more systematic and reliable than requiring a system administrator to update some files every time a component was updated. We planned to update the metadata embedded in the client assemblies, in situ, following component evolution. This proved to be very difficult for two reasons. Firstly, we wanted to use the Reflection API to extract and manipulate the strong-name assembly metadata and so loaded the assembly into an appdomain, where

it immediately became 'locked'. Unloading it required that the entire appdomain be unloaded. This implies that finding the most recent component for a particular application might require aborting and restarting the application numerous times and this seemed too much to ask the average user. Secondly, writing revised metadata back into the components, as we had planned, looks, from .NET's point of view, very much like the kind of virus attack that the CLR defences have been constructed to defeat, so we gave up.

Instead, for experimental purposes we built a GAC simulation and redirected the Assembly Loader to access that [12]. In addition, for ease of access and analysis, we extracted the assembly metadata and held it in separate XML files. This completely violates many of the .NET principles and it was never intended to be used in a 'production' environment. However it did allow us to develop and test an abstract model of suitable GAC operations and to establish the requirements for our current tool — SNAP (the Strong-Named Assembly Propagator).

The aim of SNAP was to exploit the evolutionary facilities offered by .NET to provide software support to application developers who wanted to migrate the benefits of service component updates to their own products. A major decision was to abandon the idea that updating should occur at application load-time (as with DeJaVue) when the appdomain locking occurs. Instead we envisaged a separate maintenance phase in which the user directs the tool to update the GAC and/or the policy files.

Before stating the requirements this imposes on the tool, it will be useful to list some terms and concepts derived from the theoretical model [12]:

Component A *Component* is a uniquely (strong-) named entity which requires a set of *import* services, provides a set of *export* services and maintains a set of *required* components (which collectively export those services which the component imports).

Cache A *Cache* is a set of components.

Coherent A cache is *coherent* if every service required by each component in the cache can be provided by one or more other components in the cache.

Add A component can be *added* to a coherent cache provided it is not already there and providing that adding it does not compromise the coherence property of the cache (*i.e.*, it cannot require any service which is not, a priori, provided).

Remove A component can be *removed* from a coherent cache provided that this does not compromise the coherence of the cache (*i.e.*, there must be other components that provide the services provided by the target component).

Where a sequence of 'physical' components (components actually stored in the cache) represents the temporal evolution of a single 'logical' component (a component designed to export a specific set of services), it is *assumed* that the ordering of the sequence can be deduced by examining component (strong-)names. [3]

3.1 Tool Requirements

A tool to manage GAC assemblies in the way envisaged should:

[3] This should be possible if the 4-part .NET version number is used consistently. For our work we assumed a simple linear sequence.

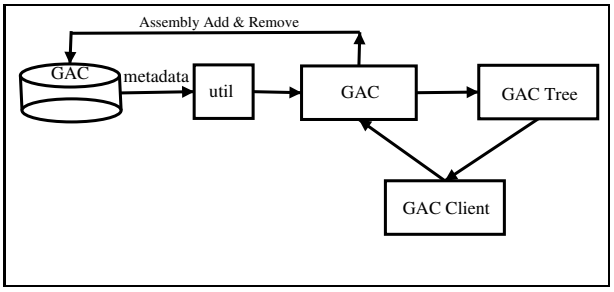

Fig. 3. GAC Administration via SNAP

1. encapsulate the contents of the GAC;
2. be able to discover the types, methods and fields *exported* by any assembly in the GAC;
3. be able to discover the types, methods and fields *imported* by any assembly in the GAC;
4. be able to provide the names of assemblies *required* by any assembly on the grounds that they export types and services that it imports;
5. establish dependencies between assemblies in the GAC utilising the capabilities in 2 — 4;
6. establish and maintain the *coherence* of the GAC utilising the capabilities in 2 — 5.
7. determine whether a later version of an assembly is (absolutely) binary compatible with an earlier version;
8. for a given version of an assembly, determine which of a sequence of subsequent versions is the most recent binary compatible version;
9. allow a system administrator to view the dependency tree of assemblies in the GAC;
10. allow a system administrator to *add* an assembly to the GAC provided this does not compromise coherence. The GAC is not a flat file-system, so this requirement involves creating an appropriately-named path to a folder containing the relevant .dll file together with an .ini file holding a copy of some of the manifest.
11. allow a system administrator to remove an assembly from the GAC. It is not possible to apply a simple 'coherence' test here since the target assembly may be explicitly referenced in the *required* metadata of another GAC assembly. If this is true, the *remove* function will not be applied. If the target is explicitly referenced in an external application, it would be necessary to rebuild the application before it could be coherently removed.
12. allow an application developer to configure an application so that it utilises the most recent (relative) binary compatible version of any available assembly. This requirement involves creating and/or maintaining an XML Application Configuration file. Note that this operation obviates the need to rebuild the application mentioned in 11.

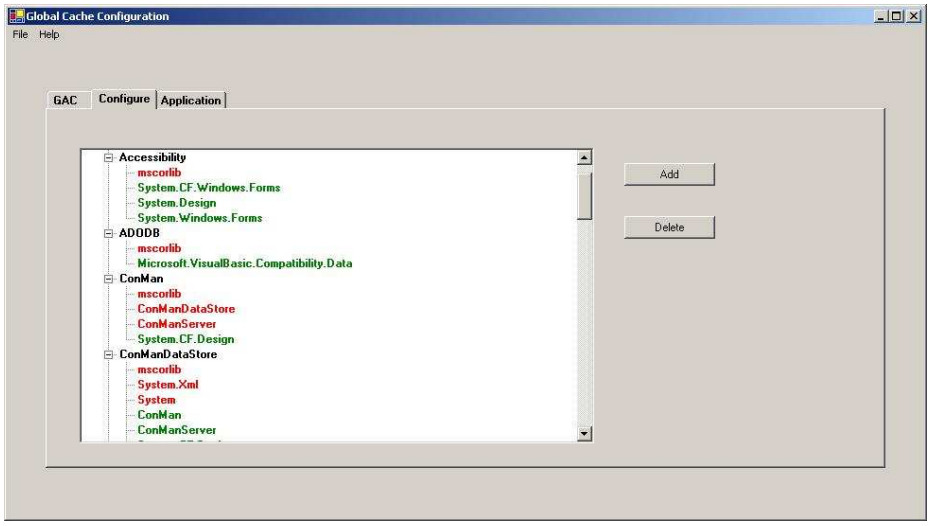

Fig. 4. GAC Configuration

3.2 System Design and Use

In the spirit of the .NET framework, it was decided to implement the tool as a set of cooperating components. In order to avoid the appdomain locking problem referred to in the previous section, it is necessary to extract the GAC assembly metadata without loading the assembly. This function is performed by an assembly called util which parses the byte-streams of each DLL in the GAC and constructs a table recording each assembly's strong-name, its required assemblies and a list of its strong-named clients.

This table is accessed by an assembly called GAC whose role is to fulfil the first requirement, namely to encapsulate the GAC. GAC exports methods which allow other components to gather assembly data and to Add and Remove assemblies from the GAC.

GACClient is the user-interface for system administrators, through which they can view the GAC and intra-assembly dependencies, via GACTree; and Add and Remove assemblies via GAC. Fig. 3 illustrates the component configuration for the fulfillment of requirements 9-11.

Fig. 4 shows the system administrator's tool (labelled Configure in SNAP). All the assemblies in the GAC are listed and the listing can be expanded to show each assembly's dependencies. The dependencies are colour-coded to distinguish between the required assemblies (red on the screen) and client assemblies (green on the screen). The Add button opens a browser so that the target assembly can be selected for insertion into the GAC. The requirements are that the assembly be strong-named— in particular that it has been 'signed'; and that any (strong-named) assemblies it depends on are already present in the GAC. Otherwise the Add operation will not succeed.

For the Delete operation, an assembly must be selected before the Delete button is pressed. Only the top-level (parent) components can be removed — it is not possible

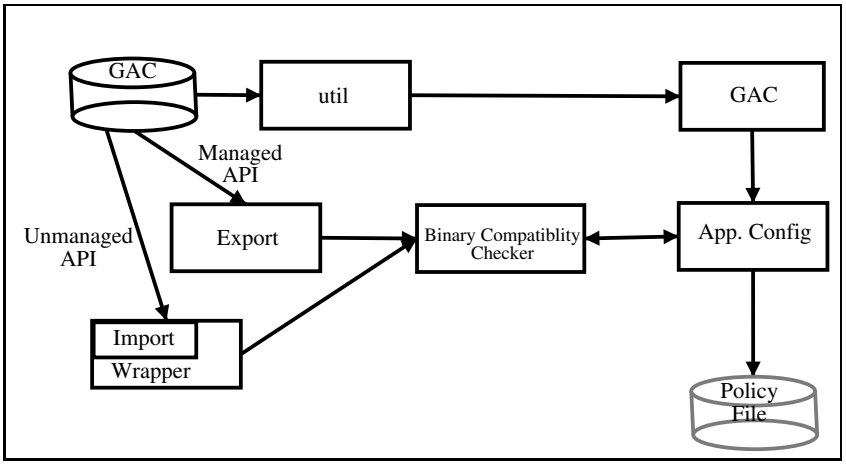

Fig. 5. Application Configuration Client

using the tool to remove entries from the dependency lists[4]. The operation will only succeed if the assembly has no dependants in the GAC.

The requirement for the Application Developer (requirement 12) involves the comparison of different versions of an assembly to discover whether the later one is binary compatible with the earlier one *relative* to the application in question. In order to do this it is necessary to establish whether the types, fields and methods imported by the application are the same as those exported for each version.

This information was extracted using reflection. The export metadata is easily available via the 'Managed' Reflection API; however when it comes to import metadata, the Managed Reflection API does not reveal the token values needed to compare with the corresponding exports. Therefore, the 'Unmanaged' Reflection API was used to examine the assembly header files, and a Wrapper assembly devised to bring this unmanaged component into the managed fold. The comparison is done in the BinaryCompatibility-Checker assembly (Fig 5), starting with the application imports and recursing through the entire dependency tree. Any redirections detected are recorded in an XML file created in the application source directory.

Fig. 6 depicts the Application Configuration tool in SNAP. The Browse button allows the user to select an application for analysis. The left window then displays a dependency tree showing all the GAC-resident assemblies. Selecting one these results in the display of its strong-name on the right-hand-side together with a window showing all available versions found in the GAC. Selecting any one of these will, via the Configure tab, result in the binary compatibility check being performed and, if this is successful, the selected assembly will be linked with the application via an entry in the application configuration file.

[4] This would horribly violate the coherence of the GAC.

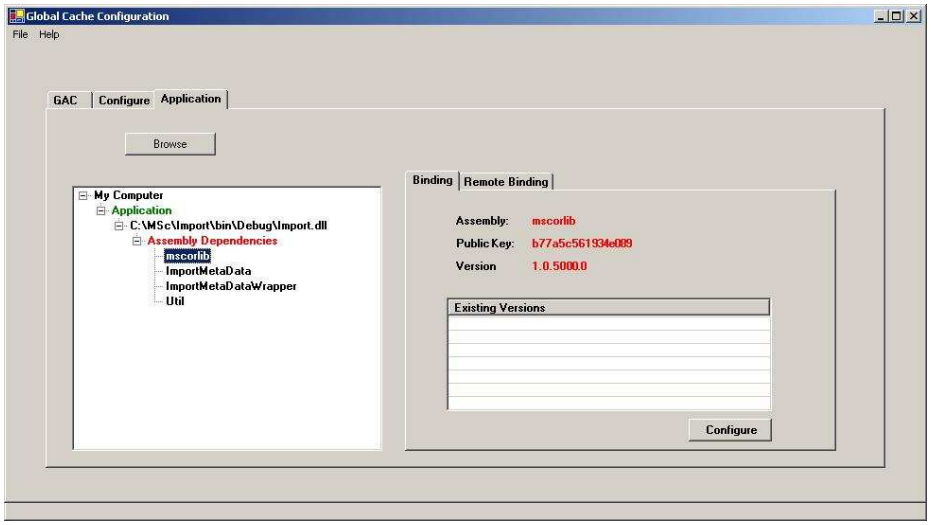

Fig. 6. Application Configuration Client

4 Related Work

Following the formal treatment of static linking [4], formal approaches to dynamic linking have been undertaken [14,1,25]. In addition, there have been proposals for modelling software evolution, utilising Requirements/Assurances Contacts [9], Reuse Contracts [28] and 'smart' composition [20]. In [12] the modelling of the GAC was accomplished using the Alloy specification language [6].

Recent work has turned to devising classification systems to assist in understanding and organising evolutionary actions and outcomes. Thus [5] identifies twelve types of maintenance activity, differentiated in terms of the purpose of the change — for example adaptive, preventative or corrective maintenance. By contrast [13] and [27] examine software source modifications from the point of view of when and how they propagate to the client module. Focussing exclusively on runtime changes, [16] have derived a very useful classification that distinguishes between the technical and motivational aspects of maintenance. The motivational facet of a change (identified as a bug fix, altered functionality or re-factoring) come into play when semantic compatibility is under consideration. The technical side of a change can be a code change, a state change and a restriction to the timing of the change. State changes and timing restrictions are in the domain of 'hot-swapping' [7,15] and have not been considered as part of the current study. However, the twenty nine different categories of code-change identified here cover the complete range of syntactic binary compatible (and incompatible) changes which SNAP has been designed to differentiate between. In an attempt to classify software change support tools [19] have proposed a taxonomy based on considerations of when (for example, compile-time), where (the scale and impact of the change), what (the nature of the change) and how (the mechanism). They have applied the taxonomy

to categorize a number a number of change support tools (for example a re-factoring browser).

5 Conclusions

Dynamic linking permits dynamic evolution — but the design and implementation of the runtime system can make a very big difference to how easy or practical it is to achieve. With extensive metadata and a repository capable of containing multiple versions of a component the .NET CLR is a good candidate. We have developed a tool, SNAP, to help to manage the GAC and the propagation of modifications between components.

A useful and practical development for SNAP would be to extend support to remote component providers and to try to make the assurance of the GAC's integrity a normal part of any deployment. Handing your GAC over to the tender mercies of any setup program downloaded from the Internet should be avoided. It would also be useful to consider ways to embrace semantic binary compatibility perhaps utilising the Publisher Policy file to convey the publisher's intentions. This is a difficult area but not one that has been left untouched by researchers.

Acknowledgements

We would like to thank Sophia Drossopoulou and the rest of the SLURP for useful insights in our numerous discussions about dynamic linking. SNAP was influenced by work done by Vladimir Jurisic and Vassu Joseph on the earlier Dejavue.NET tool and by was influenced by Shakil Shaikh and Miles Barr's Java program evolution tool DeJaVue.

References

1. Davide Ancona, Sonia Fagorzo, and Elena Zucca. A Calculus for Dynamic Linking. In *ICTCS 2003 Proceedings*, volume 2841 of *LNCS*, pages 284–301. Springer Verlag, 2003.
2. R. Anderson. The end of dll hell. In *MSDN Magazine*, http://msdn.microsoft.com/, January 2000.
3. M. Barr and S. Eisenbach. Safe Upgrading without Restarting. In *IEEE Conference on Software Maintenance ICSM'2003*. IEEE, Sept 2003.
4. Luca Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.
5. N. Chapin, J. Hale, K. Khan, J. Ramil, and W. Than. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance and Evolution*, 13(1):3–30, Jan. 2001.
6. D. Jackson, I. Schechter, and I. Shlyakhter. *Alcoa: the Alloy Constraint Analyzer*, pages 730–733. ACM Press, Limerick, Ireland, May 2000.
7. Misha Dmitriev. The Java HotSpot^{TM} Virtual Machine. http://java.sun.com/ products/hotspot/, August 2002.
8. S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java Binary Compatibility? In *Proc. of OOPSLA*, pages 341–358, 1998.

9. Dominic Duggan. Sharing in Typed Module Assembly Language. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*. Carnegie Mellon, CMU-CS-00-161, 2000.
10. F. Redmond (ed.). Microsoft .NET Framework. In *MSDN*, http://msdn.microsoft.com/ netframework/, January 2004.
11. S. Eisenbach, V. Jurisic, and C. Sadler. Feeling the way through DLL Hell. In *The First Workshop on Unanticipated Software Evolution USE'2002*. http://joint.org/use2002/proceedings.html, June 2002.
12. 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. Springer-Verlag, Nov. 2003.
13. S. Eisenbach and C. Sadler. Changing Java Programs. In *IEEE Conference in Software Maintenance*, November 2001.
14. Kathleen Fisher, John Reppy, and Jon Riecke. A Calculus for Compiling and Linking Classes. In *ESOP Proceedings*, March 2000.
15. Jens Gustavsson. Jdrums. www.ida.liu.se/ jengu/jdrums/, 2004.
16. Jens Gustavsson and Uwe Assmann. A Classification of Runtime Software Changes. In *The First Workshop on Unanticipated Software Evolution USE'2002*. http://joint.org/use2002/proceedings.html, June 2002.
17. G. Steele J. Gosling, B. Joy and G. Bracha. *The Java Language Specification*, pages 251–273. Addison Wesley, 2 edition, June 2000.
18. E. Meijer and C. Szyperski. What's In A Name: .NET as a Component Framework (Invited Paper). In *First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 22–28. http://www.ccs.neu.edu/home/lorenz/oopsla2001/, Oct. 2001.
19. Tom Mens, Jim Buckley, Awais Rashid, and Matthias Zenger. Towards a taxonomy of software evolution. In *Proc. of OOPSLA*, 2003.
20. M. Mezini and K. J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proc. of OOPSLA*, pages 97–116, 1998.
21. Microsoft. Versioning, Compatibility and Side-by-Side Execution in the .NET Framework. In *MSDN Flash Newsletter*, http://msdn.microsoft.com/netframework/technologyinfo, 2003.
22. M. Pietrek. Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, October 2000.
23. S. Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, November 2001.
24. D. Rogerson. *Inside COM*. Microsoft Press, 1997.
25. S. Drossopoulou, G. Lagorio and S.Eisenbach. Flexible Models for Dynamic Linking. In *Proc. of the European Symposium on Programming*. Springer-Verlag, March 2003.
26. S. Eisenbach, C. Sadler and S. Shaikh. Evolution of Distributed Java Programs. In *IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *LNCS*. Springer-Verlag, June 2002.
27. P. Sewell. Modules, Abstract Types, and Distributed Versioning. In *Proc. of Principles of Programming Languages*. ACM Press, January 2001.
28. P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proc. of OOPSLA*, 1996.
29. D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly Press, 2003.
30. C. Szyperski. *Component Software – Beyond Object Oriented Programming*. Addison-Wesley / ACM Press, 2 edition, 2002.