Managing the Evolution of .NET Programs

Susan Eisenbach[1], Vladimir Jurisic[1], and Chris Sadler[2]

[1] Department of Computing
Imperial College
London, UK SW7 2BZ
{sue,vj98}@doc.ic.ac.uk
[2] School of Computing Science
Middlesex University
London, UK N14 4YZ
c.sadler@mdx.ac.uk

Abstract. The component-based model of code execution imposes some requirements on the software components themselves, and at the same time lays some constraints on the modern run-time environment. Software components need to store descriptive metadata, and the run-time system must access this 'reflectively' in order to implement *dynamic linking*. Software components also undergo *dynamic evolution* whereby a client component experiences the effects of modifications, made to a service component even though these occurred after the client was built.
We wanted to see whether the dynamic linking mechanism implemented in Microsoft's .NET environment could be utilized to maintain multiple versions of components. A formal model was developed to assist in understanding the .NET mechanism and in describing our way of dealing with multiple versions. This showed that .NET incorporates all the features necessary to implement such a scheme and we constructed a tool to do so.

1 Introduction

The dynamic link library (DLL) was invented to allow applications running on a single system to share code. Because it is linked at run-time, when a DLL is corrected or enhanced, the effect on the client applications is experienced immediately.

This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility. Secondly, on any particular system, any given version of a DLL can only be replaced by a later version. Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Together these problems contribute to "DLL hell" which has been well described in [21, 20]. Many Microsoft support personnel report [2] DLL hell as the single most significant user problem that they are called upon to respond to.

There is another way to solve these problems, and that is by allowing the system to keep multiple versions of the same DLL such that each application is linked to a compatible version. Although it reduces the amount of code-sharing and loses the benefits of dynamic evolution, this idea is the one that is implemented in Microsoft's .NET environment. .NET aims to provide developers with a component-based execution model. Any

E. Najm, U. Nestmann, and P. Stevens (Eds.): FMOODS 2003, LNCS 2884, pp. 185–198, 2003.

application will consist of a suite of co-operating software *components* amongst whose members control is passed during execution. The CLR (Common Language Runtime) needs to be 'language-neutral' which means it should be possible to write a component in any one of a variety of languages and have it executed by the CLR; and also that it should be possible to pass data between components written in different languages (using the Common Type System – CTS).

The usual way to do this is with an intermediate language which the high-level languages compile to, and to constrain their data types to those recognized by the intermediate language. .NET achieves this with IL (Intermediary Language). Whereas most systems with an intermediate language have run-time systems that interpret the intermediate code (usually for portability reasons), the CLR uses a 'just-in-time' (JIT) compiler to translate IL fragments (primarily method bodies) into native code at run-time.

The native code must run within some previously loaded context, and when the JIT compiler encounters a reference external to the context, it must invoke some process to locate and establish the appropriate context before it can resolve the reference. A reference to a not yet loaded entity (type or type member) within the current, or another (previously loaded) component, causes a classloader to load the corresponding type definition. When the reference is to an entity external to all loaded components, it will be necessary to load the new component (or .NET *assembly*) via the Library Loader.

The design of the .NET assembly reveals some details about how this is accomplished, with each assembly carrying versioning information structured into four fields – Major, Minor, Revision and Build. This allows many versions of the same DLL to co-exist in the system. To make this system useful two things are required. Firstly, we have to agree on the semantics of what constitutes Major, Minor *etc.* Secondly, we have to be able to exercise some control over which version will be loaded by the Library Loader when a new component is needed at run-time. The Fusion utility in .NET is configured to find, according to some *policy*, the appropriate component and to pass its pathname to the Library Loader.

Each assembly normally contains at least one code module[1]. If we restrict it to exactly one module, then Microsoft's definition of an assembly coincides with nearly everybody else's definition of a component, so we shall adopt that restriction. It is the module that is the basic unit for loading in the .NET run-time. Each module contains a piece of code (*e.g.* a class definition) translated into an IL binary. In addition, the module incorporates metadata that records the name and location of every type and member defined within the module. This metadata is referenced by the CLR whenever it needs to instantiate an object, access an attribute, invoke a method or request a type from the class loader.

In addition to the above (module) features, the assembly contains its own metadata, consisting of *type* metadata and a *manifest*. These features distinguish .NET from other current run-time environments. The type metadata records external type and member references indexed against a manifest entry. The manifest contains details (name, version number, public key, *etc.*) of each external component needed by the assembly. This information is what permits dynamic linking between the executing components and, in .NET it must be *explicitly* provided by the programmer at compile-time. By contrast,

[1] An assembly can actually consist of resources only, but this is not common.

the Java Virtual Machine has a simpler dynamic linking mechanism that *implicitly* resolves references along various 'classpaths'. The significance of this difference is that the JVM can only accommodate one version of each component at a time – normally, the first one encountered in the classpath unless elaborate mechanisms are employed to invoke custom classloaders [27]. By forcing explicit references .NET allows the programmer to bind the client code to particular versions of the referenced services. At run-time, those exact services will be accessed, provided that they still exist and that the manifest binding is not superseded by an alternative policy.

The assembly metadata is the key to the management of both dynamic linking and the sensible evolution of components. Some of this information can be accessed at run-time by developers via a Managed Reflection API, but it cannot be manipulated by this means. Instead, a set of 'unmanaged' COM interfaces allows full read/write access via C++ hacking. By this means we can see a possible way out of DLL Hell: firstly, when a component evolves, we do not need to replace the old version with the new, because .NET lets us distinguish between them by means of version numbers. Secondly, by manipulating the manifests of client components we can ensure that suitable clients can benefit from the post-compile-time evolution of their service components, thus fulfilling the main benefit of evolving DLLs. By the same means we can leave other clients' bindings undisturbed and thus overcome DLL Hell. To do this, we need to –

1. establish the characteristics of a set of inter-related components – called, in .NET, the Global Assembly Cache (GAC);
2. investigate how these inter-relationships may be affected by component evolution, and identify any requirements this may impose;
3. reflect the requirements in new policies.

Section two tackles points (1) and (2) via a formal model. Section three describes a tool Dejavue.NET [17] based on the model which attempts to implement (3). In section four we report on other recent work which has addressed this problem, and in section five we give some indications of where this work might lead in the future.

2 Modelling the Component Cache

To investigate the characteristics of systems of components with dynamic evolution we modelled the system in Alloy [16]. Alloy is a notation that supports the declarative description of systems that have relational structures. Alloy is a first-order relational calculus with transitive closure. Alloy is supported by Alcoa, the Alloy analyser [8], which allows us to analyse our Alloy model to check for consistency and to generate example situations which we may not have considered.

The Alcoa tool provides a visualiser which will display example structures graphically. This representation is easy to interpret. We can see how the components have been joined together to form a system. The figures in this paper were generated by this visualisation tool. The visualisation tool is quite flexible, allowing us to omit parts of the model and to show fields either within an object or with an arrow from the object. In figure 1 we use both ways to show the components of a system.

To work out the characteristics needed to avoid dynamic linking problems caused by having multiple versions of components we modelled the system looking for unde-

sirable behaviour. Several properties not in the original system needed to be included to simulate the behaviour we desired. The complete Alloy model is available at [12].

The type $\mathcal{S}$ of services is atomic[2]. A service $s : \mathcal{S}$, is either a programming language type (such as a class) or a member (field or method) represented by the underlying .NET Common Type System. Components $c : \mathcal{C}$ consist of a name, a version, an optional `main` method, and three sets. These are the set of services that the component imports – import, the set of services that the component exports – export, and a set of components that provide the services that are listed in the imports – req. These sets model the metadata. In a programming context, the services that a component imports are the references that need to be resolved so that the component can be linked.

Definition 1 (Component)**.** *A component $c : \mathcal{C}$ is defined as:*

$$< name : id, \mathsf{version} : ord, \mathsf{main} : optional,$$
$$\mathsf{import} : \mathcal{P}(\mathcal{S}), \mathsf{export} : \mathcal{P}(\mathcal{S}), \mathsf{req} : \mathcal{P}(\mathcal{C}) >$$

We can extract an element of a component by its name, *e.g.* $c.\mathsf{main}$. We assume the ordering of version is temporal, that is, if two components have the same name but different version numbers, the one with the greater version number was produced more recently. The four number Microsoft version numbers are totally ordered so a single number is sufficient. The first of our version numbers is denoted by v_0 or `0`. Given a version number v, the one that follows is $next(v)$.

In our model services (class, method or field) don't exist in isolation, they only exist if they are exported from some component. The predicate in property 1 states this.

*Property 1 (*ServiceInComponent*).*

$$\forall s \in \mathcal{S}, \exists c \in \mathcal{C}\ (s \in c.\mathsf{export})$$

In our model two components with the same name and version are considered the same component, whatever their other elements are, whereas two components with the same name but different version are distinguishable. This is stated in property 2.

*Property 2 (*Unique*).*

$$\forall c_1, c_2 \in \mathcal{C}\ ((c_1.\mathsf{name} = c_2.\mathsf{name} \wedge c_1.\mathsf{version} = c_2.\mathsf{version}) \Rightarrow c_1 = c_2)$$

For a given component, there is a relationship between the set of services import that are imported by a component and the set of components req. Namely, the set req only contains components that export the required services. More than one component may export a given service so a given component's req set is not necessarily unique.

*Property 3 (*ExportsFoundInRequiredComponents*).*

$$\forall c \in \mathcal{C}, \forall s \in c.\mathsf{import}, \exists c_1 \in c.\mathsf{req}\ (s \in c_1.\mathsf{export})$$
$$\forall c_1, c_2 \in \mathcal{C}\ (c_2 \in c_1.\mathsf{req} \Rightarrow c_1.\mathsf{import} \cap c_2.\mathsf{export} \neq \varnothing)$$

Two different components with the same name (and hence different version numbers) may cause problems if one tries to link to both of them. So it is not possible for one component to import services from two components with the same name[3]. Nor is

[2] For a declared type $\mathcal{T}$, $t \in \mathcal{T}$ and $t : \mathcal{T}$ will be used interchangeably.

[3] This restriction does not exist in .NET although it does exist in many current .NET language implementations [18].

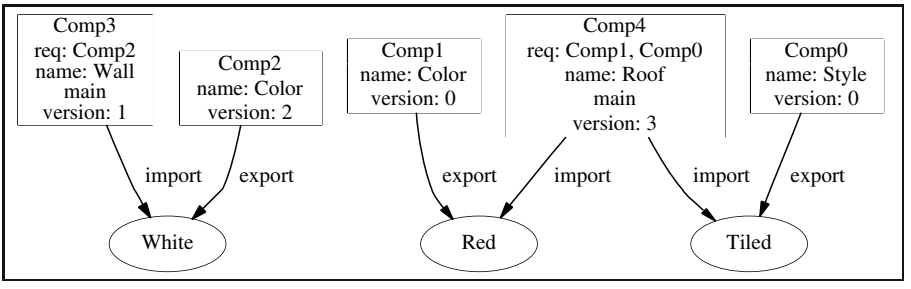

Fig. 1. A Closed set of five components containing two versions of one component.

it possible in the transitive closure of the req sets of a given component to require two components with the same name.

Property 4 (OnlyOneVersion).

$$\forall c, c_1, c_2 \in \mathcal{C}\ ((c_1 \neq c_2 \wedge c_1 \in c.\mathsf{req} \wedge c_2 \in c.\mathsf{req}) \Rightarrow c_1.\mathsf{name} \neq c_2.\mathsf{name})$$
$$\forall c, c_1 \in \mathcal{C}\ (c_1 \in c.^{*}\mathsf{req} \Rightarrow c.\mathsf{name} \neq c_1.\mathsf{name})$$

Figure 1, automatically generated from the Alloy model at [12][4], contains an example of a set of components that have already evolved over time. `Comp3` and `Comp4` have main methods and the other three components do not. There are two components named `Color`. The earlier version `Comp1` exports a service `Color.Red`. The later version of this component `Comp2` exports `Color.White`. `Comp4` requires services from `Comp0` and `Comp1` as can be seen from its req set whereas `Comp0` is not dependent on any other components.

It is possible that a service required by one of a set of components is not available. We define the predicate Closed to test for whether all services required by the components within a set are available in that set.

Definition 2 (Closed)**.**

$$\mathsf{Closed} \subseteq \mathcal{P}(\mathcal{C})$$
$$\mathsf{Closed}(C) \Leftrightarrow \forall c \in C, \forall s \in c.\mathsf{import}, \exists c_i \in C\ (s \in c_i.\mathsf{export})$$

We need to model the component cache G. By requiring G to be Closed it will have the property that services required from any component within the set are always available within the set itself.

Definition 3 (GlobalComponentCache)**.** *A global component cache $G : \mathcal{G}$ is a finite set of components s.t.* Closed(G).

Next we define a program $P : \mathcal{P}$. Programs consist of a set of components, such that one c contains a main method and there are no unresolved references, in any of the components. This component c is the starting point (main holds for this component)

[4] Alloy generated figures have had a small amount of hand editing of the labels labels to make the models easier to understand.

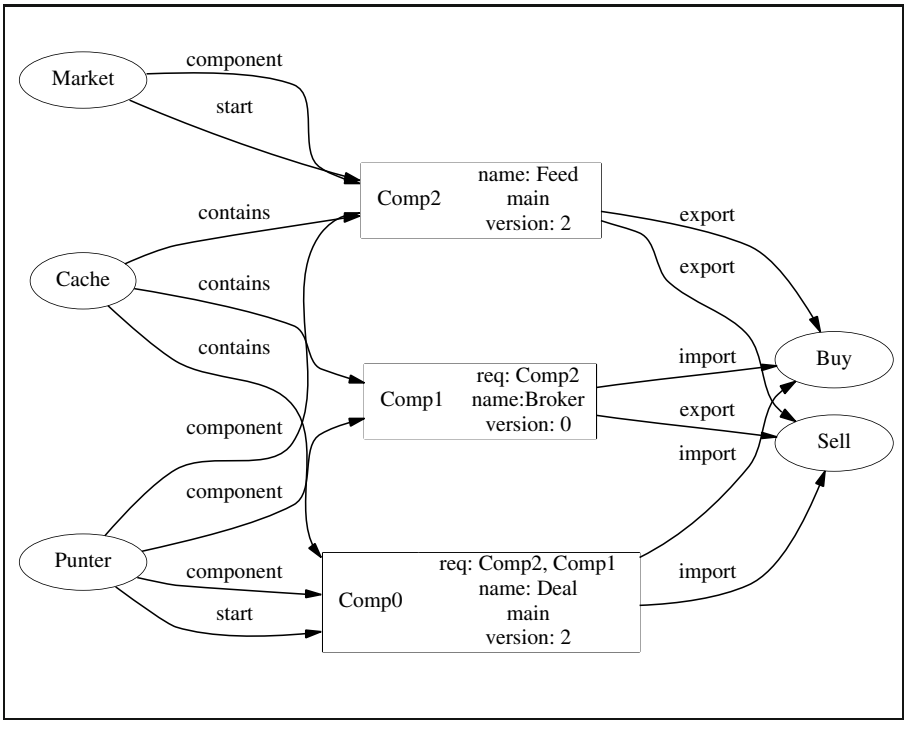

Fig. 2. A cache containing two programs.

and all other components are those that can be reached from this component by the transitive closure of c.req. Figure 2, automatically generated from the Alloy model, shows a cache G containing three components and two programs.

Definition 4 (program)**.** *A program $P : \mathcal{P}$ is a finite set of components including a component c with a* main *s.t.*

$$P = \{c\} \cup c.\mathsf{req}^*$$

Programs don't exist in isolation. They are held within caches.

*Property 5 (*ProgramInCache*).*

$$\forall P : \mathcal{P}, \exists G : \mathcal{G}\ (P \subseteq G)$$

Sometimes we need to know whether a set of components (*e.g.* a program or a cache) uses the latest versions of components that provide needed services available in that set[5]. This we call WellVersioned.

[5] This model assumes that there are no side-effects – everything provided by a component is in its exports.

Definition 5 (WellVersioned)**.**

$$\begin{aligned}&\mathsf{WellVersioned} \subseteq \mathcal{P}(\mathcal{C})\\ &\mathsf{WellVersioned}(C) = \forall c \in C, \forall c_1 \in C \setminus \{c\}\ ((c.\mathsf{name} = c_1.\mathsf{name}) \Rightarrow\\ &\qquad\qquad ((c.\mathsf{version} \geq c_1.\mathsf{version}) \vee \neg\mathsf{Closed}(c.\mathsf{req}^* \cup \{c\} \setminus \{c_1\})))\end{aligned}$$

A component can be added to a set of components (*e.g.* a program or a cache) only if adding it doesn't lead to unresolved references in the augmented set. Before adding a component a unique version number has to be given to the component. If the component to be added is the first possessing its name then its version number is 0. If another component already exists within the set with the same name, then the version number of the new component needs to be greater than all other version numbers of components with the same name.

Definition 6 (NewNum)**.**

$$\begin{aligned}&\mathsf{NewNum} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{C}\\ &\mathsf{NewNum}(C, c) \quad = \quad < n, 0, m, i, e, r >,\\ &\qquad\qquad\qquad\quad \textbf{if } \nexists c_1 \in C\ (c_1.\mathsf{name} = c.\mathsf{name})\\ &\qquad\qquad\quad = \quad < n, next(c_1.\mathsf{version}), m, i, e, r >,\\ &\qquad\qquad\qquad\quad \textbf{if } (\exists c_1 \in C\ (c_1.\mathsf{name} = c.\mathsf{name}) \wedge\\ &\qquad\qquad\qquad\quad \forall c_2 \in C\ (c_2.\mathsf{name} = c.\mathsf{name} \Rightarrow c_2.\mathsf{version} \leq c_1\mathsf{version}))\\ &\qquad\qquad where\ c =< n, v, m, i, e, r >\end{aligned}$$

Definition 7 (Add)**.**

$$\begin{aligned}&\mathsf{Add} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C})\\ &\mathsf{Add}(C, c) = C \cup \{\mathsf{NewNum}(C, c)\}, \ \textbf{if } \mathsf{Closed}(C \cup \{\mathsf{NewNum}(C, c)\})\\ &\qquad\quad = C, \qquad\qquad\qquad\qquad\quad otherwise\end{aligned}$$

As we add components to a cache (or a program) some housekeeping needs to be done, to ensure that the latest versions of components that don't break code will be dynamically linked. In particular before a given component starts executing we have to update the pointers that indicate which components will get chosen to resolve references. These are modelled as the req set for each component. None of our definitions so far, alter this set for a given component, even when newer versions of components have been added. Firstly we need to find out which is the latest version of a component that provides whatever was required. There will always be such a component since there is no operation that can remove the Closed property from a set.

Definition 8 (Latest)**.**

$$\begin{aligned}&\mathsf{Latest} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{C}\\ &\mathsf{Latest}(C, c) = c_1, \qquad \textbf{if} \qquad c \in C \wedge c_1 \in C \wedge\\ &\qquad\qquad\qquad\qquad\qquad\quad c.\mathsf{name} = c_1.\mathsf{name} \wedge c.\mathsf{export} \subseteq c_1.\mathsf{export} \wedge\\ &\qquad\qquad\qquad\qquad\qquad\quad \forall c_2 \in C\ (c.\mathsf{name} = c_2.\mathsf{name} \Rightarrow\\ &\qquad\qquad\qquad\qquad\qquad\quad (c_2.\mathsf{version} \leq c_1.\mathsf{version} \vee c.\mathsf{export} \subsetneq c_2.\mathsf{export}))\\ &\qquad\quad = undef, otherwise\end{aligned}$$

Once we know which version of a component should be linked we need to be able to create for each component a new req set. This we do with Provides which takes a component's req set within a cache G and returns a set with the newest components that contain the required exports.

Definition 9 (Provides)**.**

$$\begin{aligned}&\mathsf{Provides} : (\mathcal{P}(\mathcal{C}), \mathcal{P}(\mathcal{G})) \longrightarrow \mathcal{P}(\mathcal{C})\\ &\mathsf{Provides}(R, G) = \{c_i \mid \exists c \in R\ (c_i = \mathsf{Latest}(G, c) \land R \subseteq G \land c_i \in G)\}\end{aligned}$$

We now have the functionality needed to reconfigure all the pointers so that programs can evolve. We use the $\oplus$ symbol to override the values of a component's req set.

Definition 10 (Reconfigure)**.**

$$\begin{aligned}&\mathsf{Reconfigure} : \mathcal{P}(\mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C})\\ &\mathsf{Reconfigure}(G) = \{c_i \mid \exists c \in G\ (c_i = c \oplus \mathsf{Provides}(c.\mathsf{req}, G))\}\end{aligned}$$

Finally we need an algorithm for evolving the global cache. We only put into the cache (using Add) new components whose references can be completely satisfied by the components already in the cache. After we put a component in we need to do some housekeeping. Firstly, all components in the global cache (including the new one) have to have their req sets updated, so they now get their import services from the latest versions of components that provide them. Secondly, any components which are no longer needed should be removed.

Definition 11 (Evolve)**.**

$$\begin{aligned}&\mathsf{Evolve} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C})\\ &\mathsf{Evolve}(G, c) = \mathsf{Reconfigure}(\mathsf{Keep}(\mathsf{Add}(G, c), c))\\ &\ \mathit{where}\\ &\qquad \mathsf{Keep}(C, c) = \{c_i \mid c_i \in C \land\\ &\qquad\qquad (c.\mathsf{name} \neq c_i.\mathsf{name}) \lor\\ &\qquad\qquad ((c.\mathsf{name} = c_i.\mathsf{name}) \land ((c.\mathsf{version} < c_i.\mathsf{version}) \lor\\ &\qquad\qquad (c_i.\mathsf{version} \leq c.\mathsf{version} \land c_i.\mathsf{export} \subsetneq c.\mathsf{export})))\}\end{aligned}$$

Theorem 1. *If a set of components C is* Closed *and* WellVersioned *then*

$$\forall c : \mathcal{C}\ (\mathsf{Evolve}(C, c))\ \textit{is}\ \mathsf{Closed}\ \textit{and}\ \mathsf{WellVersioned}\ .$$

Proof. For each of the properties, by contradiction.

Unfortunately Alloy cannot be used to check the theorem (using its `assert`) since it is higher order. All one can see is that the theorem is not inconsistent with the model. This can be seen by adding the conditions Closed and WellVersioned of the input set and the output set to Evolve. The models generated are the same as those without these conditions.

The sets of components we are actually interested in are caches.

Corollary 1. *If the set of components in a* GlobalComponentCache G *is* WellVersioned *then*

$$\forall c : \mathcal{C}\ (\mathsf{Evolve}(G, c))\ \textit{is}\ \mathsf{Closed}\ \textit{and}\ \mathsf{WellVersioned}\ .$$

Using Alloy to build and test the model helped refine it considerably. Running early versions immediately threw up undesirable behaviour. Most of the properties (such as ExportsFoundInRequiredComponents or those in Component) were added to stop the converse from being possible. We convinced ourselves that the definition of Evolve behaved as we wanted by using Alloy to show that there were no models that failed to meet our theorem.

This model was built using features that are all available from .NET assemblies. New programs are WellVersioned and the changes during cache evolution are designed to maintain WellVersionedness. If the component cache is evolved according to our model, then the programs built from the cache components should contain the latest versions of components that do not cause problems.

Of course there are other ways of modelling a component cache. The formalization cannot cope with components that engage in cyclic import relationships because we only add one component at a time whilst maintaining Closed at all times. This restriction reflects current .NET development tools (but not .NET itself). Also in our model the set of services that are imported and the set of components that provide these services are not explicitly paired. Nor is there an explicit list of connections between components. The model might have been in some sense more natural, if these were included, but then it would have deviated from what actually occurs in .NET metadata.

We set out to build a versioning tool, Dejavue.NET, that could follow the precepts of our model to manage evolution in .NET. The prototype is described in the next section.

3 Dejavue.NET

Before designing a tool based on our model, a number of practical problems needed to be solved. Our model envisages a single version number with the convention that a *higher* number indicates a *later* version. Early Beta versions of .NET revealed a versioning policy that would resolve references against an assembly with higher Revision and Build numbers than the compile-time version, provided the Major and Minor numbers were the same. No checking was done to determine whether or not the two versions were binary compatible. Presumably, this approach did not solve the problems for the Beta testers because in later Beta releases and indeed in the .NET official release, this policy was dropped in favour of a default policy of an exact match between the compile-time and the link-time versions – thus ruling out dynamic evolution altogether[6].

Until there is some agreement amongst developers about what sort of evolution will cause change in which parts of the version number, it will never be possible to implement sufficiently sensitive policies based on version numbers alone. For the tool therefore, we will stick to the idea that a higher number indicates a later version, and

[6] So the approach is that rolling forward is only safe if explicit policy statements have been made.

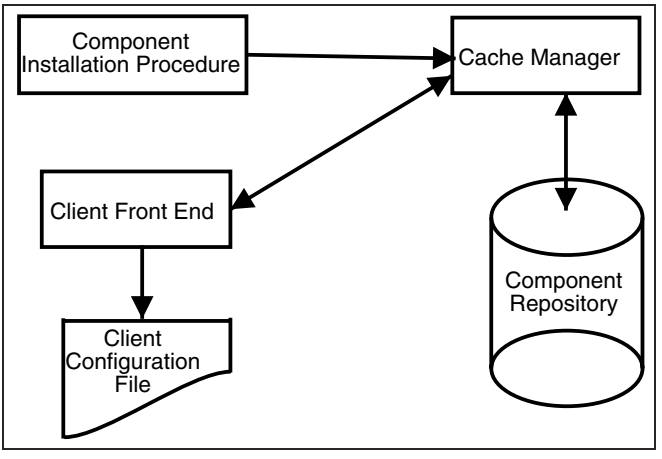

Fig. 3. High level architecture of the single-machine Dejavue.NET.

we will use Closed and WellVersioned properties to determine which later version, if any, maintains compatibility.

Initially we planned to manipulate assembly metadata in situ in order to keep client components abreast of service component evolution. This idea was frustrated in two ways – firstly, .NET encrypts assemblies in the GAC and this puts the metadata out of reach of unmanaged API manipulations. Secondly, in the .NET release the GAC assemblies were placed out of reach of even the managed API. This is another blow (like the inflexible versioning policy) to dynamic evolution.

However, before it looks in the GAC, the Fusion utility consults a sequence of configuration files. Whoever can control these files has the capability to override, with more up-to-date possibilities, the assembly metadata relationships established at compile-time. This provides a mechanism for implementing a more evolutionary dynamic linking policy, utilising the operations and predicates identified within our theoretical model.

Instead of using the GAC, our tool mimics the GAC with its own Component Repository (CR) which is maintained by a CacheManager module (see Figure 3). This faithfully implements the installation regime defined in our model via a Component Installation Routine. A second function in the CacheManager, the Redirection Information Retrieval Routine, traverses the CR dependency tree to construct, for a given component, a configuration file containing the current (link-time) dependency information.

The final part of the tool is the ClientFrontEnd. This allows users to execute the most up-to-date version of an executable component c, in the following steps:

1. requests the CacheManager to interrogate the CR to produce a list of executable components;
2. allows the user to select the required component c;
3. passes the component name back to the CacheManager to construct 'on the fly' the appropriate configuration file, which is stored in the ClientFrontEnd's working directory;
4. invokes c.

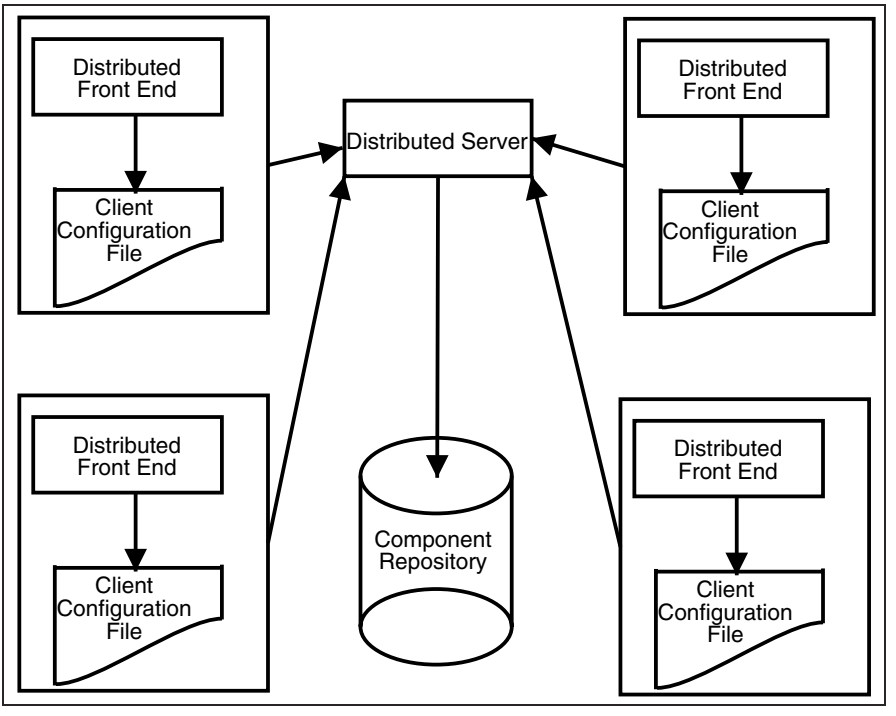

Fig. 4. Distributed Dejavue.NET architecture.

Once this has been done, any references external to c will be resolved by invoking the Fusion utility. Instead of looking in the GAC however, Fusion will use the configuration file to locate components in the CR.

A tool of this sort can help an end-user to keep up-to-date provided all updates are downloaded onto the local system and introduced to the Component Repository via the Cache Manager. This is not a very practical solution, especially from the point of view of component developers. Instead, a prototype distributed version of the tool has been developed (see Figure 4). In this scheme the Component Repository is hosted on a server which performs the functions of the Cache Manager. Each client then has a local copy of the Client Front End which maintains local copies of the application configuration files.

This scheme could slow down execution times if all external references have to be resolved across the network, so efficiency considerations may dictate some local caching mechanism. In addition, where support is required for distributed component developers and maintainers, each one needs to be able to install components in the Component Repository. This is a complex procedure which alters the state of the entire cache, therefore it is necessary to lock the cache during each execution of the Component Installation Routine. This may make the Component Repository noticeably inaccessible to other developers and to all users, so there is some risk of degrading the service. This gives further motivation to the idea of local caching.

4 Related Work

The model in the last section was built using Alloy [16], a language for building and analysing software systems. There are several different modelling approaches we could have chosen including [5, 15]. We chose to use Alloy because the models one can write are at the same level of abstraction as our models and there is tool support for analysing them. The models are declarative and describe the structure of systems.

The work described here is a natural extension of work on Java done in collaboration with Drossopoulou and Wragg [26, 25]. Drossopoulou then went on to model dynamic linking in [6] and we looked at the nature of dynamic linking in Java [9]. More recently with Drossopoulou and Lagorio [24] there has been work on providing an operational semantics model for dynamic linking flexible enough to model either Java or $C\#$. There has also been work looking at the software engineering aspects of dynamic linking. We have examined the problems associated with the evolution of Java distributed libraries [10, 11], have looked at the problems that arise with binary compatible code changes in [10] and have built tools, described in [11, 27]. Other formal work on distributed versioning has been done by Sewell in [23].

Rausch [3] models software evolution in terms of Requirements/Assurances contracts. Associated with every component there are explicit textual declarations of the services that the component requires and provides (assures) together with predicates to specify the desired behaviour. The Requirements/Assurances contracts are additional documents that map some or all of the Requirements of one component to the Assures of a second one. As the components evolve, the contract provides the means to check syntactic and behavioral consistency. Since the contracts must explicitly identify the components, it does not seem possible to automatically maintain multiple versions of a single component.

Other groups have studied the problem of protecting clients from troublesome library modifications. [22] identified four problems with 'parent class exchange'. One of these concerned the introduction of a new (abstract) method into an interface. The other issues all concern library methods which are overridden by client methods in circumstances where, under evolution, the application behaviour is adversely affected. To solve these problems, *reuse contracts* are proposed in order to document the library developer's design commitments. As the library evolves, the terms of the library's contract change and the same is true of the corresponding client's contract. Comparison of these contracts can serve to identify potential problems.

Mezini [19] investigated the same problem (here termed horizontal evolution) and considered that conventional composition mechanisms were not sophisticated enough to propagate design properties to the client. She proposed a *smart* composition model wherein, amongst other things, information about the library calling structure is made available at the client's site. Successive versions of this information can be compared using reflection to determine how the client can be protected. These ideas have been implemented as an extension to Smalltalk.

Formal treatments of static linking were suggested in [4]. Dynamic linking at a fundamental level has been studied in [13, 1, 28], allowing for modules as first class values, usually untyped, concentrating on confluence and optimization issues. [14], discuss dynamic linking of native code as an extension of Typed Assembly Language without

expanding the trusted computing base, while [7] takes a higher-level view and suggests extensions of Typed Assembly Language to support type safe dynamic linking of modules and sharing.

5 Conclusions

Perhaps DLL Hell is a special place reserved only for Microsoft people, but the Upgrade Problem has to be faced by everybody who writes or uses component-based software. Some part of the solution may lie in clever language design, and another may lie in employing strict software development and maintenance procedures – but a large part lies in the versioning strategy of the linking mechanism deployed by the run-time system.

In this paper we have looked at this mechanism as deployed by current .NET development tools. We have shown that if a discipline is kept over which components can be added to the Global Assembly Cache, DLL Hell can be avoided. All the features needed to avoid the problem exist in the .NET manifests but there currently does not seem to be the will to implement the discipline. For example, the multi-part version numbers give developers scope to classify the effects of modifications precisely, and the assembly metadata with the managed code API offers the prospect of reflective code compatibility analysis. However, some of the implementation trade-offs in the current release have frustratingly limited the scope of what is possible. Consequently, the GAC is not usable as a cache for evolving components. As a consequence we have developed our prototype tool, to work alongside and within .NET and to demonstrate what could be attainable.

Our tool is currently more restrictive then is necessary. In the future we will seek to integrate our tool more closely with the GAC and to relax the requirements that components can only be added singly.

Acknowledgements

We would like to thank Sophia Drossopoulou for her careful reading of earlier versions of the model and Michael Huth for suggesting Alloy. Matthew Smith and Rob Chatley made helpful suggestions about the paper. Dejavue.NET was influenced by DEJaVU a tool for Java evolution, implemented by Shakil Shaikh and Miles Barr. Andrew Tseng and Dave Porter of UBS Warburg provided motivation beyond idle academic curiosity. Finally, we would like to thank Sophia Drossopoulou and all the project students of our weekly meeting group for their helpful suggestions and insights while the ideas were being developed.

References

1. Davide Ancona and Elena Zucca. A Primitive calculus for module systems. In *PPDP Proceedings*, September 1999.
2. R. Anderson. The End of DLL Hell. http://msdn.microsoft.com/, January 2000.
3. A. Rausch. *Software Evolution in Componentware using Requirements/Assurances Contracts*, pages 147–156. ACM Press, Limerick, Ireland, May 2000.
4. Luca Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.

5. A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1994.
6. S. Drossopoulou. An Abstract Model of Java Dynamic Linking, Loading and Verification. In *Types in Compilation*, September 2001.
7. 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.
8. D. Jackson, I. Schechter, and I. Shlyakhter. *Alcoa: the Alloy Constraint Analyzer*, pages 730–733. ACM Press, Limerick, Ireland, May 2000.
9. S. Eisenbach and S. Drossopoulou. Manifestations of the Dynamic Linking Process in Java. http://www-dse.doc.ic.ac.uk/projects/ slurp/dynamic-link/linking.htm, June 2001.
10. S. Eisenbach and C. Sadler. Ephemeral Java Source Code. In *IEEE Workshop on Future Trends in Distributed Systems*, December 1999.
11. S. Eisenbach and C. Sadler. Changing Java Programs. In *IEEE Conference in Software Maintenance*, November 2001.
12. S. Eisenbach. Alloy Model of .NET Evolution. Technical report, http://www.doc.ic.ac.uk/˜sue/alloymodel, August 2003.
13. Kathleen Fisher, John Reppy, and Jon Riecke. A Calculus for Compiling and Linking Classes. In *ESOP Proceedings*, March 2000.
14. Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and Flexible Dynamic Linking of Native Code. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*. Carnegie Mellon, CMU-CS-00-161, 2000.
15. G. Holzmann. The Model Checker Spin. In *IEEE Transactions on Software Engineering*, volume 23, 5, May 1997.
16. D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, http://sdg.lcs.mit.edu/˜dng/, February 2002.
17. V. Jurisic. Deja-vu.NET: A Framework for Evolution of Component Based Systems. http://www.doc.ic.ac.uk/˜ajf/Teaching/Projects/DistProjects.html, June 2002.
18. E. Meijer and C. Szyperski. Overcoming independent extensibility challenges. *Communications of the ACM*, 45(10):41–44, October 2002.
19. M. Mezini and K. J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proc. of OOPSLA*, pages 97–116, 1998.
20. M. Pietrek. Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, October 2000.
21. S. Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, November 2001.
22. K. Mens P. Steyaert, C. Lucas and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proc. of OOPSLA*, 1996.
23. P. Sewell. Modules, Abstract Types, and Distributed Versioning. In *Proc. of Principles of Programming Languages*. ACM Press, January 2001.
24. S. Drossopouloum, G. Lagorio and S.Eisenbach. Flexible Models for Dynamic Linking. In *Proc. of the European Symposium on Programming*. Springer-Verlag, March 2003.
25. D. Wragg S. Drossopoulou and S. Eisenbach. What is Java binary compatibility? In *Proc. of OOPSLA*, volume 33, pages 341–358, 1998.
26. S. Eisenbach S. Drossopoulou and D. Wragg. A Fragment Calculus: Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *Logic in Computer Science*, pages 147–156, 1999.
27. S. Eisenbach, C. Sadler and S. Shaikh. Evolution of Distributed Java Programs. In *IFIP/ACM Working Conf on Component Deployment*, June 2002.
28. Joe Wells and Rene Vestergaard. Confluent Equational Reasoning for Linking with First-Class Primitive Modules. In *ESOP Proceedings*, March 2000.