DEJA-VU.NET

A FRAMEWORK FOR EVOLUTION OF COMPONENT-BASED SOFTWARE SYSTEMS

Submitted by Vladimir Jurisic

Supervised by Susan Eisenbach

Second marker – Phillipa Gardner

Project directory:
http://www.doc.ic.ac.uk/~vj98/project
Component Based Software Engineering (CBSE) focuses on the development of software systems composed from reusable binary components. In such component-based software systems, applications are evolved through the selective evolution of individual components. Despite its evident benefits, this ‘plug and play’ approach to software evolution is restricted by a number of practical problems.

The evolution of component-based software is hindered by the inability of existing component frameworks to guarantee that an evolved version of a server component preserves its ability to satisfy the syntactical, behavioral and concurrency guarantees imposed by its existing clients.

This project proposes an extension to the Common Language Infrastructure (CLI) component framework that applies the notion of binary compatibility in the context of link-time client re-configuration, thus automatically isolating client components from server versions known not to satisfy a set of syntactical requirements.

An investigation of the properties of the CLI and a formulation of a theoretical framework for reasoning about the effects of inter-component syntactical constraints on the evolution of a complete software system are followed by a proposal for an infrastructure for evolution of component-based software in distributed environments with large numbers of applications, users, component maintainers, and component developers.

A prototype tool - which automates the application reconfiguration, software deployment, and version management within complex component-based software systems - is developed to illustrate the theoretical concepts.

1 Complex component inter-dependencies are supported, but a set of assumptions in made on the underlying object model.
I saw the individual project as an opportunity to explore a set of concepts, and to relate my own ideas to the current state-of-the-art. Rather than focusing my efforts on a visually appealing piece of work, it was my desire to challenge myself and to make a contribution, no matter how small, to the general software engineering field, prior to my return to the job market.

First of all, my thanks go out to my supervisor Susan Eisenbach for her support, encouragement, guidance, understanding and patience, and for having an interesting anecdote to tell during each and every meeting. Working with her was a privilege.

I am grateful to Sophia Drossopoulou for the constructive criticism, and for showing me the importance of being confident about what you do not know.

I want to thank Andrew Tseng and Dave Porter who introduced me to the wondrous world of software components and their role in a cutting-edge IT environment.

I am ceaselessly grateful to my family and to my friends around the world who bared with me and supported me during this period, especially to my cousin Ivan who inspires me to be my best no matter what I do.
Physical Module Loading ................................................................. 41
Assembly Resolution ........................................................................... 41
  IL instructions containing MethodRef tokens ........................................... 46
  IL instructions containing TypeRef tokens ............................................. 47
Type Safety Verification ..................................................................... 48
  Constraints arising out of accessibility and visibility contracts ............... 48
  Constraints arising out of inheritance contracts ................................... 50
IL Opcode ......................................................................................... 51
Dynamic Linking as an Implementation of Dynamic Component Composition 52
Section Summary ............................................................................. 52

SECTION 3 – IMPORT interfaces, EXPORT interfaces, and Binary Compatibility ...... 53
  Introduction......................................................................................... 53
Representing Component Evolution ...................................................... 54
Re-considering Binary Compatibility .................................................... 55
Effects of CLI’s Delayed Loading and Linking .................................... 56
Benefits of the Component Signature Matching Approach .................... 57
Re-conceptualizing .NET Components ................................................ 58
The IMPORT interface ....................................................................... 60
Signature Matching and the SUBSET Function ...................................... 63
Limitations to Signature-matching Imposed by the .NET Framework Metadata APIs 65
Evaluating Binary Compatibility in Terms of Changes Made To a Component ..... 66
Using ☀ to Express Absolute Binary Compatibility ................................ 69
Defining ☀ ....................................................................................... 71
  Defining Modifications in (Є - L) ......................................................... 72
Binary Compatibility Detection Algorithms ......................................... 75
  Absolute Binary Compatibility Detection Algorithm ........................ 75
Section Summary ............................................................................. 80

SECTION 4 - Binary Compatibility and the Evolution of a Component-based Software
System ............................................................................................ 81
  Introduction ....................................................................................... 81
Limitations of a client-server model .................................................... 81
Modeling a Component-based system as a Set B ................................... 83
Dependency relation ........................................................................ 84
Indirect dependencies ..................................................................... 85
Interaction between Component Dependency and Interface Implementation and Subtyping ........................................................................................................................................ 153
APPENDIX B - Side-by-Side Execution and Component Merging .................................................. 157
APPENDIX C – BCEngine Design ........................................................................................................ 161
APPENDIX D – CacheManager OO Design ......................................................................................... 162
APPENDIX E - Information Sources Used to Model the CLR Dynamic Linking Process ................................................................. 164
APPENDIX F - Physical Representation of TypeRef and MemberRef Metadata Tokens ................................................................. 166
APPENDIX G – Dynamic Linking Tests .............................................................................................. 167
TEST 1 - A Single-Module Assembly .................................................................................................. 167
TEST 2 - A Simple Multi-Module Assembly ......................................................................................... 168
TEST 3 - A Simple Multi-Assembly System ......................................................................................... 169
TEST 4 - Interfering with Module Loading ......................................................................................... 170
TEST 5 - Changing the Name of a Referenced Type Declared Within a Different Module ......................... 170
Test 6 - Changing the Name of Referenced Methods and Fields......................................................... 171
TEST 7 - Cross-Assembly Type Migration .......................................................................................... 172
TEST 8 - Dependencies of Dynamic Linking Phases ......................................................................... 173
TEST 13 - CLR Verification is Lazier than JVM Verification ............................................................... 176
APPENDIX G ............................................................................................................................................... 190
The Problem

Component-based software engineering (CBSE) has the potential to reduce timescales, improve software quality, and increase the scope for specialization. Component-based applications promote reusability, extensibility and evolvability of software systems. A growing portion of all software - exemplified by Operating System layers and plug-in architectures - is delivered in the form of components.

Before the full benefits of component-based software development and deployment are unlocked, a number of practical problems must be confronted. Even on single machines, most users have experienced Dll Hell - a phrase coined to describe a situation where installation of a newer, evolved version of a library (usually to fix a problem in one application) breaks a number of existing applications.

The situation is aggravated in complex multi-user IT environments with large numbers of complex inter-dependent applications, all built from a pool of components that encapsulate the organization's business logic. The component pool is constantly growing as component developers produce new components, and component maintainers produce new versions of existing components that correct faults, meet new requirements, or improve the functionality of the software. The administrators of such systems face a fundamental problem - "how to ensure that a new, evolved, version of an existing component is used only by clients for which successful component composition process is guaranteed prior to application execution?" They are faced with two conflicting goals:

- Rapid propagation of all modifications embedded in evolved components to end-user applications
- Guarantees on successful outcome of dynamic component composition – even a single point of failure can break a complex applications built from a large number of components
The Framework

The Microsoft .NET framework is the first implementation of ECMA Standard 335 - a Common Language Infrastructure (CLI) for the development, execution, and inter-operation of language-independent components. CLI specification introduces a Common Type System (CTS) that enables components to inter-operate via a common object-model.

Unlike a number of competing component frameworks, CLI specifies components as single entities that combine the published interfaces and the binary implementations into single versioned entities. During dynamic component composition, the CLI runtime can determine the exact version of a component from which types are loaded.

CLI provides a powerful versioning infrastructure, but associates no intelligence with the versioning process - the task of configuring the versioning information within a CLI component-based system is still delegated to programmers or system administrators. With few means for determining whether a new server version will successfully link against existing clients (let alone a means to automate this process), this is a difficult task.

The Project

The first part (Section 1 through to Section 4) of this project is concerned with the development of conceptual connections between the component-model theories and the CLI implementation, the development of terminology and models necessary to tackle the issues within the problem domain, and the formulation of a set of algorithms. After a more detailed introduction to this large problem domain, the project investigates the CLI dynamic linking process (a physical implementation of the component composition concept); examines the constraints on the evolution of a server component imposed by its clients; analyzes the effects of changes made to a component's interface of exported features on the successful outcome of a dynamic linking process; develops a graph model of a component-based system and an algorithm for application of syntactical guarantees (derived from an analysis of changes made to individual components) to the automatic re-configuration of such a software system.

These theoretical concepts are then applied in the development of DejaVu.NET - a unified solution to the related issues of binary compatibility detection, evolution of component-based software, and distributed application deployment and version control within complex IT environments.
Under a set of assumptions\(^2\), DejaVu.NET guarantees that

“for all possible applications that can be executed from a common set of components, each client component will be dynamically composed with the latest version of the server component which does not cause this dynamic composition process to fail”

The **DejaVu.NET prototype** developed by this project extends the .NET framework with a centralised self-configuring infrastructure and application launchers that enable dynamic application composition in accordance with the above goal.

---

Pages 148 and 149 feature a glossary of commonly used .NET terms, and a glossary of terms specific to this project.

\(^2\) Discussed in detail in **Appendices A and B**
Benefits of Component-Based Software Systems

There is an increasing need for computer systems to be open and distributed. The need for computer systems to meet rapidly evolving and changing requirements has led to the interest in component-oriented software development approaches [1]. Component Based Software Development (CBSD) aims to reduce development and maintenance costs by encapsulating functionality into reusable components, thus shifting development focus from programming to component integration [2].

Software systems based on reusable components are making monolithic systems obsolete. Enabling the functionality of an open system to be changed and extended via component substitution and addition is just one of the advantages of component-based systems. They promote faster development, reduce development costs, promote higher levels of software quality assurance, reduce size of distributed binaries, provide instant functionality to encapsulated behavior, increase the scope for code reuse and outsourcing, promote system scalability, and enable dynamic application reconfiguration.

A ‘definition’ of a Component

There is no universally accepted definition of a component – the intuitive notion is reused across a wide range of contexts and granularities. Szyperski, a leading authority within the field of CBSD, defines software components as binary units of independent production, acquisition, and deployment that interact to form a functioning system [3]. A software component can be deployed independently and is subject to composition by third parties, such as client components. A component-based software application can thus be defined as a set of interconnected components, possibly distributed over a network.
A component package typically includes [4]:

- A list of provided interfaces
- A list of required interfaces, known as dependencies
- The external specification
- The executable code
- Validation code
- The design.

Despite a lack of a formal definition, certain properties must be satisfied by all components. First of all, clients interact with components solely via published interfaces. Hence, interfaces (usually sets of methods, procedures, or functions that can be invoked by clients) are the means by which components connect. Interfaces must be stored in a standard format that clients are aware of prior to dynamic linking. Clients must be aware of interface changes. Secondly, no look onto the sources of a component should be necessary to reuse it, i.e. compose it with other components. Dynamic composition requires components in a binary form. In general, components are accessed by consulting published interfaces in order to determine the linking positions within the binaries. In this context, both dynamically linked libraries and sets of classes distributed as modules or packages are treated as components.

It follows from the above properties that component composition is dependant upon the process of dynamic linking. Unlike static linking where references between client code and libraries are resolved during compilation, dynamic linking resolves such references at run-time [44]. Instead of placing the component code in the executable, the compiler makes a reference to the component and to the interface method accessed by the client.

---

3 For brevity, further discussion will refer to all services declared in published component interfaces as methods.

4 To avoid ambiguity, Szyperski defines the binary form as a non-source-code form of the component.
Evolution

All software must be continually modified after delivery to correct faults, to improve performance or other attributes, and to adapt the product to a changed environment [37]. The evolutionary pressures on computer applications and programs result in never-ending maintenance activity [43]. Unlike monolithic systems, component-based software systems can be selectively evolved by treating individual components as basic units of maintenance, upgrade, and deployment. A software engineering process, utilizing one of a number of existing methodologies, can be iteratively applied to evolve each component. Large software systems can be evolved through the evolution of individual components, each of which encapsulates specific functionality. Development of methodologies and frameworks for its evolution of component-based systems is an active research area [12, 13, 14, 15, 19, 34, 36, 40].

Informal Definition of Binary and Logic Compatibility

In order to evolve components successfully, two key requirements must be met:

**Binary Compatibility** or **Syntactic Compatibility** - client code accessing the component through its interface of exported features must be able to locate the required features after modifications performed during the component’s evolution.

**Logic Compatibility** or **Semantic Compatibility** – the exported features of an evolved component must still satisfy the specifications they satisfied prior to such evolution, enabling clients of pre-evolution component to access the expected functionality.

This project focuses solely on the syntactic evolutionary constraints imposed by the requirement on servers to maintain binary compatibility relative to existing clients. A change to a component is defined as binary compatible if it does not affect the dynamic linking processes between the component and its clients.

The maintenance of semantic compatibility between subsequent component versions is a problem within the domain of behavioural refinement. Behavioural refinement seeks to establish whether one component provides all the behaviour of another, such that the two components can be substituted without changing the behaviour of the whole software system. [34]. The majority of current research in this field focuses on matching the formal specifications of preconditions and postconditions of methods stored within a component. A complete approach to semantic compatibility detection
must also guarantee that changes to connections between component features\[28\] change only in accordance with a set of semantical constraints.

**Component Evolution in Commercial Component Frameworks**

Commercial Component Frameworks, such as the Component Object Model (COM), CORBA (Common Object Request Broker Architecture), and Enterprise JavaBeans (EJB), share the common idea of separately published interfaces (known as metadata) used by clients to access binary “black-box” components. These frameworks utilize a number of approaches to deal with the issue of evolving a server component while maintaining binary compatibility relative to existing clients. Generally, these approaches can be placed into one of these three categories:

- **Re-implementation of binary components with no modifications to published interfaces.** Clients will, assuming that logic compatibility is not broken, be unaware of the implementation changes.

- **Interface extension.** With this approach, a component’s interface is extended to publish new functionality while maintaining all of the previous functionality.

- **Interface re-implementation.** With this approach, a new interface is published alongside a re-implementation of the “black box” binaries, such that an older binary component is replaced with a new version which exposes multiple interfaces to different clients.

Interface extension technique is demonstrated by the CORBA standard - an open industry standard that facilitates remote object invocation via published Interface Definition Language (IDL) interfaces. CORBA does not yet specify a market-accepted component model for plug-and-play interoperability [10]. CORBA implementations support the notion of component evolution in terms of IDL inheritance – a new version of a component interface is a subclass of the previous version. It is also possible to evolve components without changes to published interfaces by changing the binary “black-box” implementations of components.

---

5 An example of such a connection is a method within a client component calling an **Initialize** method within a server component. If this method call is removed during the evolution of the client, the whole system can malfunction.

6 There is no universal formal definition of a component framework. In most contexts, a component framework is an infrastructure or an execution environment for components.
Interface re-implementation technique is demonstrated by Microsoft’s COM standard - a binary standard for platform and programming language independent component interoperability [11]. Binary components expose their functionality through interfaces that contain semantically related functions. COM component interfaces are immutable\(^7\) and are discovered dynamically through introspection. Developers place additional behavior in an updated version of the interface. Newer clients can query for the newer interface and use it if present, but still function with older functionality. Since interface and implementation are separate, component binaries can once again be re-implemented without changing any of the published interfaces.

The key shortcoming of these techniques is their inability to ensure that a correct version of the binary component is loaded for a given client. For example, the COM runtime offers no intrinsic support to enforce that the correct version of a binary server is loaded for the calling client [6]. The Microsoft .NET framework (discussed shortly) attempts to tackle this problem through support for version-aware component loading and for side-by-side execution.

**Object Code Modification**

This approach to component evolution is based on the idea of applying a specification of a modification to existing source code or existing binaries. Early examples of code modification include the superimposition [12] and class composition techniques. Since both of these techniques apply modifications to source code, they are inapplicable in the context of evolution of binary components.

Latest research in this field has focused around binary component adaptation techniques [13]. Unlike the earlier approaches, code modification is performed on the component binaries during component loading rather than compilation.

---

\(^7\) i.e. they provide no support for subtyping, subclassing or inheritance
Distinguishing Between Runtime and Link-time Evolution

There are numerous examples of real-time mission-critical component-based applications. Component evolution in such applications must take place without shutting down and restarting individual components. A majority of current techniques allow component-based systems to be evolved at linking time i.e. the complete software system must be restarted before effects of evolution are perceived. These techniques share a common inability to support unanticipated dynamic component evolution, which is still an active research area. Current approaches are based on the introduction of an additional layer of indirection between object references or function pointers, and the objects or functions.

Kniesel [14] introduces a runtime component adaptation technique known as delegation. Delegation (or object-based inheritance) is founded on the notion of the transmigration of object identity - runtime replacement of object references.

Siyama [36] applies a similar indirection approach to bind user programs to proxy functions which, in turn, selectively invoke the correct version of a function.

Architecture Definition Languages

The need to reason about properties of component-based software systems led to the development of methods for specification and analysis of such systems. Early methods applied object modelling techniques which are limited by a weak support for hierarchical descriptions, a reliance on method invocations as sole interconnection primitives, and a limited ability to reason about critical system design properties such as performance and reliability. These problems were overcome by the introduction of Architecture Definition Languages (ADLs) such as Darwin [18], Rapide [28], and ACME.

ADLs present architectures as high-level descriptions of a component-based software system through multiple views, including structural and behavioral views [5]. Evolution of software systems is treated in terms of the evolution of components, connectors, and architectural topologies [15]. A number of ADLs use connectors to enable components to have no knowledge of the ways in which they are connected\(^8\). Software system evolves as connectors are added, removed, replaced and reconnected.

While ADLs provide software engineers with powerful modeling tools they can use to analyze the impact of evolution of component-based systems, they provide no infrastructure for automating this evolution process.

\(^8\) Section 3 shows how the reverse is true for .NET components – they contain statically-embedded ‘awareness’ of the server components they need to be connected to
Component Repositories

Version control is a set of procedures and techniques that let multiple incarnations of the same application component exist simultaneously as the application evolves over time [16]. A standard method of providing version control in a software system is through the use of component repositories. A component repository stores and tracks the evolution of component interfaces and implementations, matches an interface to the latest correct implementation, and provides search and retrieval tools. With the provision of an inference engine that allows a runtime environment to infer the correct version of a component, component repositories can be used within the context of load-time component evolution.

DejaVu

A tool for evolution of distributed Java programs, DejaVU [40] introduces the concept of rule-checking as means to deduce whether the evolution of a single server component preserves binary compatibility relative to a single client component. DejaVU’s key weaknesses are its coarse definition of binary compatibility and its restricted application to single-client and single-server component-based software systems. This project will tackle these weaknesses while building on DejaVU’s strengths, such as its modular RuleEngine design.
Microsoft .NET Framework

The .NET framework is a development platform for multi-tier distributed computing environments. At its core, it provides a common runtime environment (known as the Common Language Runtime (CLR)) consisting of a common type system, a metadata system, and an execution system. Using a CLR-aware compiler, source code written in any programming language can be compiled into a standard IL (Intermediary Language) representation. Rich metadata is embedded into compiled code, describing all types and their functionality. Metadata enables a powerful reflection API and the language interoperability features of the framework.

Common Type System

All .NET components share a common underlying object model, known as the Common Type System (CTS). The use of CTS implies that, in contrast to traditional component models where components typically call library functions written in a different language, .NET components written in different languages can pass objects to one another, and extend each other’s capabilities.

Figure 1 shows the roots of the inheritance hierarchies of the CTS. All types are classified into classes, interfaces, delegates, enums, or structs. Details can be found in ECMA 335 [25, 26, 27]. All non-interface types extend System.Object.

An investigation of the interaction between the .NET object model (CTS) and the .NET component model has led to a number of interesting conclusions presented in Appendix A.
DejaVu.NET – A Framework for Evolution of Component-based Software Systems

Figure 1 - Roots of the CTS Inheritance Hierarchies

Managed and Unmanaged Code

- Managed code executes within the CLR, uses types declared by the CTS, and uses a number of managed services such as garbage collection.

- Unmanaged code executes outside the CLR, does not use CLR services, and introduces custom types which can exist outside the CTS inheritance hierarchies.

Assemblies and Modules

A .NET assembly is a versioned, self-describing, platform-neutral collection of type definitions and optional resources [6]. .NET requires all types to belong to an assembly [7]. Assemblies thus function as a basic unit of code versioning and deployment.

In order to decouple the physical and the logical notions of a reusable component and exploit code partitioning and on-demand loading, .NET framework introduces the concept of a module as the basic unit of code loading. A module is a single binary file stored in the Portable Executable (PE) format. Each logical assembly is composed of one or more physical modules, all of which share the assembly’s version number.
Figure 2 illustrates the layout of a typical single-module assembly. A multi-module assembly follows a similar layout, except that only one of these modules contains the assembly manifest.

The assembly manifest (also known as assembly metadata) documents each module within the assembly, establishes the assembly version, and documents dependencies by listing external assemblies referenced by the current assembly. The manifest provides a layer of indirection between assembly implementation details and assembly clients, and is instrumental in enabling the assemblies to be self-describing.

Each module consists of IL code and type metadata. The stored IL code is platform and CPU agnostic, and will be compiled at runtime into platform and CPU specific instructions. The type metadata completely describes each type within the module and all members of each type. This metadata is used by the .NET runtime to resolve the location of types and their members within the binary, to create object instances, and to facilitate method invocations. [6]. These processes, as well as the actual means of physical metadata and IL instruction storage, are covered in more depth in Section 2.

This project assumes single-module assemblies, and maps the concept of a component to a physical .NET module, as well as to a logical .NET assembly – the words **component**, **assembly**, and **module** are used interchangeably. This approach enables the project’s ontology to remain compatible with both the terms used throughout the .NET framework documentation and with the terms used in the theoretical computer science.

The manifest module\(^9\) is either an executable with an **.exe** extension, or a library with a **.dll** extension. All other modules have a **.netmodule** extension. To be treated as executables by the runtime and to have a **.exe** module, assemblies must contain one or more types with a method marked with a **.entrypoint** IL directive (a **static Main()** method in C# grammar), which serve as the application entry points. Viewing .NET assemblies as software components, it is evident that there is no distinction in the storage, versioning, and deployment of executable and library components.

Unlike Java, .NET requires the programmer to explicitly state dependencies between distributed binary files - every module explicitly references other modules whose types it accesses. While Java type referencing is performed relative to a classpath,

---

\(^9\) The module containing the physical metadata corresponding to the assembly manifest
.NET type referencing takes place in a specific **resolution** context – an external assembly, an external module, or an enclosing type definition.

**Metadata**

The Common Language infrastructure uses metadata to describe and to reference the types defined by the Common Type System [25]. It is stored in a language-independent format and serves as a common interchange mechanism between tools that manipulate programs, and the Common Language Runtime (CLR). CLR uses the metadata to manage code execution and to connect separately generated modules together at runtime.

Component-based systems typically store metadata in auxiliary files, such as interface definition language (IDL) files, type libraries, interface repositories, implementation repositories, and the system registry. .NET framework avoids this issue by storing metadata within modules – thus .NET components are **self-describing**. There is no separation between client-visible interfaces and “black box” binaries - the component’s interface becomes a part of the component binary.

.NET component metadata is extensible, so that additional information can be associated with components.

Metadata can be read (to access existing type declarations), extended, or written by direct access to a correct file format or through one of the two programmatic APIs.
Managed Reflection API

The Managed API is a subset of the Common Type System, and, as such, provides a set of classes that can be manipulated by any CLI-compliant programming language.

Any managed object or type has an associated `System.Type` object which, used as an entry point to the Managed Reflection API, enables runtime interrogation.

The Managed API provides a reduced-feature access to a component’s metadata.

Unmanaged Metadata API

The Unmanaged API consists of a set of unmanaged COM interfaces, accessible only via unmanaged C++ code. It provides direct access to the complete metadata contents of an assembly.

Metadata is physically stored in tables and heaps, both of which are indexed via metadata tokens. Tokens are 4-byte values where the most significant byte specifies the table or heap, and the remaining bytes specify the table row number or the heap offset [26]. Tokens are named according to the heap or table that they index. Metadata tokens are used as arguments to many IL instructions [27], as shown in Appendix F. Table 1 shows the mapping between a number of managed Reflection API classes and the corresponding unmanaged tokens that they wrap.

<table>
<thead>
<tr>
<th>Managed Type</th>
<th>Unmanaged Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>mdTypeDef</td>
</tr>
<tr>
<td>FieldInfo</td>
<td>mdFieldDef</td>
</tr>
<tr>
<td>MethodInfo</td>
<td>mdMethodDef</td>
</tr>
<tr>
<td>ParameterInfo</td>
<td>mdParamDef</td>
</tr>
<tr>
<td>EventInfo</td>
<td>mdEvent</td>
</tr>
<tr>
<td>PropertyInfo</td>
<td>mdProperty</td>
</tr>
<tr>
<td>Assembly</td>
<td>mdAssembly</td>
</tr>
<tr>
<td>AssemblyName</td>
<td>mdAssemblyRef</td>
</tr>
</tbody>
</table>

Table 1 – Mapping Managed Reflection API types to underlying unmanaged tokens

In addition to the ability to retrieve a number of additional token properties, the Unmanaged API can access two token types, of vital importance to this project, that are invisible to the Managed API – `mdTypeRef` and `mdMemberRef`. These tokens encapsulate references to the types and type members defined outside the current assembly, and are discussed in detail in Section 2.
Public Assemblies

The .NET framework makes a distinction between private and public assemblies. Private assemblies are distributed with and used by single applications, and hence provide no versioning support. Public assemblies are shared between applications, are versioned, and are protected from tampering through digital signing with a public/private key pair. The versioning infrastructure of each .NET compliant computer system includes a global public assembly repository known as a Global Assembly Cache (GAC).

Since private assemblies have no version numbers, only public assemblies are considered by this project.

Version Numbers

Association of each .NET public assembly with a version number allows multiple versions of a public assembly to co-exist on a single machine, thus isolating clients from incompatible versions of the same assembly [6].

The assembly’s version is determined by a version number attribute present in the assembly’s manifest. The version number consists of four discrete numerical parts: Major, Minor, Revision, and Build numbers. For example, all .NET assemblies are initially versioned with the “1.0.0.0” version number.

In addition to specifying the version of the current public assembly, the assembly manifest also stores version numbers of all external assemblies that the current assembly was built against. Access to this metadata allows the .NET class loader to enforce versioning policies during the dynamic linking process. Section 2 discusses the way in which assembly version numbers are used by the Common Language Runtime to locate the physical modules.

In general, the .NET framework provides a robust and feature-rich versioning infrastructure, while leaving the addition of intelligence to its versioning policies to the programmer or the administrator. This project seeks to exploit the existing infrastructure while providing a parallel solution to the related problems of inter-component binary compatibility detection and the provision of intelligence to the .NET assembly versioning. Generation of custom application-specific versioning policies based on the results of an extensible metadata-driven binary compatibility detection algorithm will form a heart of the tool.
SECTION 2 - A Model of the Common Language Runtime Dynamic Linking Mechanism

Introduction

It was stated in Section 1 that component composition is dependant upon the physical process of dynamic linking. The physical implementation of the CLR dynamic linking processes is now investigated.

A .NET assembly encodes all references to functionality defined in external assemblies in terms of references to external types and external type members. Such references are represented in the assembly metadata by the TypRef and the MemberRef metadata tokens [26], which, respectively, serve as indices into the TypeRef and MemberRef metadata tables. This report section shows how the process of resolution of these metadata tokens into runtime handles gives rise to the dynamic linking process.

Section 3 uses TypRef and MemberRef tokens as a basis for the declaration of an IMPORT\textsuperscript{10} interface associated with a component’s dependency. Similarly, the metadata tokens that export the type and type member definitions (includingTypeDef, Member, and Field tokens [26]), are used as a basis for declaring an EXPORT interface of a component.

Consider a client and a server component that are related by a dependency. Modifications to either of these components have a potential effect on the outcome of the dynamic linking process. More specifically, the dynamic linking process can fail after a server component’s EXPORT interface is modified such that it no longer satisfies the requirements imposed by the client’s IMPORT interface. To enable the definition of a precise set of requirements imposed by a client IMPORT interface and to enable an analysis of effects of metadata interface modifications on the dynamic linking process, it is crucial to gain an understanding of the CLR dynamic linking process itself. This process completes with a successful mapping of these metadata tokens to pointers to in-memory structures created using the definitions stored within the external server components. A number of sub-processes - including assembly resolution, physical file loading, type loading, verification, and JIT-compilation - will be defined and related in terms of their functions within the CLR dynamic linking process.

\textsuperscript{10} The IMPORT and EXPORT interfaces are not a part of the .NET terminology
Relating Notions of a Component and Computer Code

Different information sources\textsuperscript{11} employ heterogeneous views of the relationship between a conceptual component and its computer code expression. Such relationships are clarified in Figure 3. The component is physically stored as a set of physical modules. A disassembly process turns this collection of metadata tables and method bodies into IL expressions, expressible in ILASM grammar. Source text files conforming to either the IL grammar or the grammar of a high-level language can be translated into the physical representation using the IL assembler (ILASM) or a CLR-compliant compiler respectively.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Relationship between a .NET component and the code it embeds}
\end{figure}

\textsuperscript{11} See Appendix E
The dynamic linking model is developed in terms of the physical component representation. An assumption is made that physical components are generated solely through the use of the IL Assembler or a valid CLR-compliant compiler. The use of such tools places a number of constraints on the permissible modifications that can be performed as part of the evolution of a server component, thus guaranteeing that

- the structural integrity at the physical level is maintained – the physical component layout rules of the CLI[26] are respected

- lexical rules of the source language are obeyed – avoidance of static errors guarantees the satisfaction of a set of CLI requirements - correct application of visibility and accessibility constraints, and avoidance of class and interface circularities\(^\text{12}\) are two examples

\(^\text{12}\) Unlike Java, where circularities can be produced via two .class files, circularities in .NET (due to need for programmers to specify explicit dependencies) require at least three modules. A C# compiler responds to circularities by getting into an infinite loop.
A Model of the Virtual Execution System

The execution of every managed program begins with a call to the entry point\(^\text{13}\) of an executable assembly, and proceeds as shown in Figure 4 [35]. Figure 4 shows an adapted model of the VES (Virtual Execution System) — a subset of the CLR concerned with the execution of IL code. Dynamic linking is performed whenever the ClassLoader is requested to load a type declared outside the current component.

To enable the execution to commence and to complete on host computer architecture, the Common Language Runtime needs to interpret the metadata tables and method bodies stored in the physical components, assemble in-memory structures and, usually, generate code in a format required by the host architecture and verify the type safety of such code. All of these processes, performed by the Class Loader, the Verifier and the JIT compiler, rely on the ability to process and to resolve metadata tokens. Metadata tokens also appear as arguments to IL instructions within IL method bodies.

\(^{13}\) In C#, this is the static Main() method
ECMA 335 states that the resolution of such tokens can occur either at run-time or, more usually, during the JIT-Compilation process. Evidently, the entire CLR execution process is driven by metadata.

For a number of metadata tokens, the VES execution model of Figure 4 relates the temporal context of a token’s resolution to the spatial location of the token within the physical component representation. For example, aTypeDef token physically located in an IL instruction is resolved during the JIT compilation (or the execution, depending on the implementation of the CLI) of the method body containing this instruction. The extension of the given execution model, as a means of making it applicable to the goals of this project, calls for a deeper understanding of the temporal context of TypeRef and MemberRef token resolution and, consequently, the identification of the locations of these tokens within the physical representation of a component.
Locating External Reference Tokens

TypeRef tokens

TypeRef tokens appear in four contexts - both within the metadata and within the IL code of a physical module:

1. Columns in the metadata tables

<table>
<thead>
<tr>
<th>Metadata table</th>
<th>Column name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>EventType</td>
<td>Assigns an external type to an event</td>
</tr>
<tr>
<td>InterfaceImpl</td>
<td>Interface</td>
<td>Declares a local type as an implementation of an external interface</td>
</tr>
<tr>
<td>MemberRef</td>
<td>Class</td>
<td>Associates an external type member to the external type containing its definition</td>
</tr>
<tr>
<td>TypeDef</td>
<td>Extends</td>
<td>Declares that a local type inherits from an external type</td>
</tr>
<tr>
<td>TypeRef</td>
<td>ResolutionScope</td>
<td>References external inner types by using a TypeRef rather than AssemblyRef or ModuleRef as their ResolutionScope</td>
</tr>
</tbody>
</table>

Table 2 – Metadata tables containing TypeRef tokens

14 The format of IL instructions that embed TypeRef and MemberRef tokens is shown in Appendix F
2. Signature columns in metadata tables, that encode one or more type references along with a set of modifiers\textsuperscript{15}.

<table>
<thead>
<tr>
<th>Metadata table</th>
<th>Column name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Type</td>
<td>Assigns an external type to a property</td>
</tr>
<tr>
<td>Method</td>
<td>Signature</td>
<td>Assigns external type(s) to parameters and result value of a locally declared method</td>
</tr>
<tr>
<td>Field</td>
<td>Signature</td>
<td>Assigns an external type to a field</td>
</tr>
<tr>
<td>MemberRef</td>
<td>Signature</td>
<td>Assigns external type(s) to an external field, or to parameters and result value of an external method</td>
</tr>
<tr>
<td>StandAloneSig</td>
<td>Signature</td>
<td>Assigns external type(s) to local variables of a method, or to the arguments of an indirect method call</td>
</tr>
<tr>
<td>TypeSpec</td>
<td>Signature</td>
<td>Allows a Signature to be used as a metadata token, thus providing a type specification for array operations.</td>
</tr>
</tbody>
</table>

Table 3 – Metadata tables containing encoded TypeRef tokens

\textsuperscript{15} A Signature does not necessarily encode any TypeRef tokens. If all types in a signature are encoded as TypeDefs, they are all defined within the same module as the Signature.
3. Arguments to IL instructions shown in Table 4

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box</td>
<td>ValTypeTok</td>
</tr>
<tr>
<td>castclass</td>
<td>class</td>
</tr>
<tr>
<td>Cpobj</td>
<td>classTok</td>
</tr>
<tr>
<td>Initobj</td>
<td>classTok</td>
</tr>
<tr>
<td>Isinst</td>
<td>Class</td>
</tr>
<tr>
<td>ldlema</td>
<td>Class</td>
</tr>
<tr>
<td>ldobj</td>
<td>classTok</td>
</tr>
<tr>
<td>ldtoken</td>
<td>Token</td>
</tr>
<tr>
<td>mkrefany</td>
<td>Class</td>
</tr>
<tr>
<td>newarr</td>
<td>Etype</td>
</tr>
<tr>
<td>refanyval</td>
<td>Type</td>
</tr>
<tr>
<td>sizeof</td>
<td>valueType</td>
</tr>
<tr>
<td>stobj</td>
<td>classTok</td>
</tr>
<tr>
<td>unbox</td>
<td>Valuetype</td>
</tr>
</tbody>
</table>

Table 4 – IL instructions accepting TypeRef tokens as arguments

4. Externally defined exceptions

Externally defined exceptions are defined by associating TypeRefs with catch blocks within the ILASM view of a module. The catch block specifies the type of the exception object that the clause is designed to handle, and the handler code itself. It is expressed in the ILASM grammar as

catch <TypeReference> <handlerBlock>
**MemberRef tokens**

*MemberRef* metadata tokens appear solely as arguments to IL instructions.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ldtoken</td>
<td>Token</td>
</tr>
<tr>
<td>Call</td>
<td>Method</td>
</tr>
<tr>
<td>Jmp</td>
<td>Method</td>
</tr>
<tr>
<td>Ldfn</td>
<td>Method</td>
</tr>
<tr>
<td>Callvirt</td>
<td>Method</td>
</tr>
<tr>
<td>Ldvirtfn</td>
<td>Method</td>
</tr>
<tr>
<td>Newobj</td>
<td>Ctor</td>
</tr>
<tr>
<td>Ldfld</td>
<td>Field</td>
</tr>
<tr>
<td>Ldflda</td>
<td>Field</td>
</tr>
<tr>
<td>Ldsfld</td>
<td>Field</td>
</tr>
<tr>
<td>Ldsflda</td>
<td>Field</td>
</tr>
<tr>
<td>Stfld</td>
<td>Field</td>
</tr>
<tr>
<td>Stsfld</td>
<td>Field</td>
</tr>
</tbody>
</table>

Table 5 – IL instructions that accept MemberRef tokens as argument

**Basic process of token resolution**

An understanding of the basic process of token resolution is gained by observing the actions of the CLR through the unmanaged profiling interface.

A number of profiling tests are presented in Appendix G.
Simplifications

1. Event and Property tables associate a collection of Method tokens (references to sets of methods that provide the underlying implementations) to the higher-level notions of events and properties presented in the ILASM grammar. An evaluation of changes made to entries in the Method table is hence sufficient to determine whether binary compatibility is maintained, with no specific need to evaluate changes in Event and Property tables.

2. A conforming CLI implementation adds references to external exception types to the TypeRef table. Any impact that a control-flow analysis can have on the resolution of exception TypeRef tokens is disregarded and TypeRef associated with exceptions are not given any special treatment.

3. Stand-alone signatures (StandAloneSig metadata tokens) are used to:
   a. Encode types of a method’s local variables\(^{16}\).
   b. Enable the indirect method call mechanism. The calli IL instruction uses the method entry point (stored on the top of the stack) and a stand-alone signature to perform an indirect method call. The stand-alone signature is used to specify the calling convention, argument types and result type (if any) of the method.

   Any external types encoded into a stand-alone signature are also contained in the TypeRef table. No special consideration is given to stand-alone signatures as they have no effect on the dynamic linking process beyond the effects associated with the requirement to resolve any TypeRef tokens that they encode.

4. In a situation where a TypeRef token is used as a type specification token (TypeSpec), the TypeSpec table contains the encoded TypeRef token in the Signature column. The TypeSpec therefore references the TypeRef via two layers of indirection. Since they have no effect on the dynamic linking process beyond the effect of any TypeRef tokens that they reference, TypeSpec tokens can be disregarded in further discussion.

\(^{16}\) The ILASM grammar represents such a stand-alone signature by the .locals directive
Interaction between dynamic linking processes

Figure 5 models the interaction between the CLR dynamic linking processes.

The following conventions are adopted:

- Each phase that gives rise to, or is directly involved with, dynamic linking is conceptually modeled as a separate process. Processes such as Class Loading and JIT-Compilation contain a number of sub-processes.

- Interaction between phases is represented via requests, modeled by arrows that map to procedure call semantics. The procedure call passes control to another process whenever the condition expressed by the bold text is encountered.

- Only one phase is active at an instant of time. Concurrency issues are abstracted away.

- The execution phase is initially active. It passes control to the JIT-compilation phase whenever a method is called for which there is no compiled native code. Since the execution begins with the method marked with an .entrypoint ILASM directive [mapping to a static Main() method in C#], JIT-compilation of this method is requested before any code is executed.

- IL instruction compilation is tied to the sub-phases of verification and of resolution of any tokens provided as arguments. Additional type loading requests are sent to ClassLoader as side-effects of both the token resolution and the verification.

---

17 The term “procedure call semantics” is used here in the sense of control-passing. Only one process is executing at a time, and has the ability to pass the control to another process once it encounters a condition blocking its further execution.
**Figure 5 – A model of interaction between the CLR dynamic linking phases**

- Assembly Resolver
  - [Fusion]
  - Manifest module is not resolved to a loading path

- Library Loader
  - [Platform-specific]
  - Module that contains the metadata definition of the type is not loaded
  - The parent class or an interface implemented by the type is not

- Class Loader
  - Reference made to an unloaded type

- JIT Compiler
  - JIT Code Generator
    - [One block of IL code processed at a time]

- Key:
  - □ Phase
  - → Call
  - **Bold Text** Call condition

- Code block not verified
  - Verifier

- Code block contains unresolved tokens
  - Resolver

- A method without a native stub called

- Execution of native method code
Class Loading

Despite its name, the **ClassLoader** performs no physical disk-to-memory file loading. The disk-to-memory or network-to-memory file loading is referred to as module loading and is performed by the **LibraryLoader**. This anomaly must be dealt with directly in order to avoid ambiguity, especially when the CLR dynamic linking is compared with existing dynamic linking models of Java’s JVM [17].

The operation of the **ClassLoader** is better explained as a process of resolution of a *TypeDef* metadata token into a pointer to a new instance of *EEClass* – a class used internally to represent all ‘loaded’ types. Such a pointer is referred to as a runtime type handle. In addition to the creation of this new instance and its method table, this resolution process introduces a number of other side-effects and constraints. These side-effects and constraints, as well as the act of resolution itself, can be collectively understood as the process of type loading. Should the **ClassLoader** be requested to resolve an unresolved *TypeRef* token into a runtime handle, it will request the **LibraryLoader** to resolve the *TypeRef* into a *TypeDef*, before proceeding to resolve the *TypeDef* into a runtime handle. The act of physical module loading (performed by **LibraryLoader**) is a side-effect of this delegated resolution task.
The Shared Source CLI execution engine\textsuperscript{18} implements the \texttt{ClassLoader} in file \texttt{clsload.cpp}. Every type is loaded through a call to the \texttt{LoadTypeHandle} method that accepts a \{Module, metadata token\} pair to identify the desired class. Code inspection and analysis reveals that the \textbf{Class Loader} performs the following steps as part of the type loading process:

- Validates the metadata token - must be a \texttt{TypeDef, TypeRef or TypeSpec}
- Loads the parent type via a new \texttt{LoadTypeHandle} call
- Checks that the parent type is not sealed
- Checks that the parent type is not \texttt{System.Array}
- Gets the properties of the enclosing type without loading it
- Checks for existence of layout metadata and its consistency relative to the parent type
- Checks if the type is an enum or a delegate
- Creates an instance of \texttt{EEClass} to represent the new class through a number of sub-processes:
  - Ensuring that delegate types have no layout
  - Ensuring that layout metadata doesn’t specify both a sequential and an explicit instance layout
  - Ensuring that interfaces are abstract and that they have no base type\textsuperscript{19}
  - Allocating memory for an \texttt{EEClass} and its vtable and statics
  - Initializing class members
- Ensures that delegate types are sealed and that they extend \texttt{System.MultiCastDelegate}
- Ensuring that only nested types use accessibility constraints
- Retrieving properties of all interfaces of the type
- Loading each interface type via new \texttt{LoadTypeHandle} calls, and checking that token has been correctly resolved to an interface runtime handle
- Loading up and summarizing the field metadata for layout classes
- Building the method table for the type through a two sub-processes:
  - Creation of an interface map containing all interfaces implemented by the type
  - Placement of vtable methods (non-static and non-private methods) into the method table, taking the interface map into account
- Notifying the CLR Profiling interface that class loading has begun and has ended

The loading of parent types and interfaces implemented by the type is a side-effect of the type loading process. Side-effects can introduce constraints. For example, the inability to locate the parent class or the loading of a sealed parent class both lead to the throwing of a \texttt{TypeLoadException} during the dynamic linking. Constraints

\textsuperscript{19} It has no token in the Extends column of its metadata table
introduced by the type loading side effects form an important subset of constraints on successful evolution of a component.

**Verification and Validation**

The processes of validation and verification also impose significant constraints on the metadata consumed during the dynamic linking process.

**Validation** refers to the process of examination of metadata tokens in a module to ensure that they index the metadata tables correctly, and that no buffer overflow occurs when accessing string embedded in metadata. Validation is performed either during the physical loading of a Portable Executable module into memory (by a LibraryLoader), or during the process of installation of an assembly into the Global Assembly Cache\(^{20}\). The process of validation can be seen as a process enforcing a correct mapping between two different views of the same metadata – the view of metadata in terms of the ILASM grammar, and the view of the metadata in terms of the physical metadata tables. Since this project seeks to reason about maintenance of binary compatibility solely through an analysis of ILASM grammar (maintenance of structural metadata integrity during component evolution is assumed from the outset), validation is of no further significance.

**Verification** is performed in parallel to the JIT compilation. Given a sequence of IL instructions, the CLR verifier enforces type safety, checks for occurrence of stack underflow/overflow, and enforces the correct use of exception handling and object initialization facilities [31].

Verification is linked to the host computer’s security policy. The security model adopted by the commercial .NET framework is evidence-based in the sense that all code must present sufficient evidence to the runtime to prove its membership to one of a number of pre-defined code groups. There is a many-to-one mapping between a code group and a security policy. Policies are defined through permission sets. Most permission sets contain a SecurityPermission class with a SkipVerification flag. All code that is not type safe must have been granted a SecurityPermission with the enabled SkipVerification flag in order to run. If a sequence of unverifiable IL is found and the SkipVerification flag is disabled, the offending IL is replaced with a stub that throws an exception if that execution path is exercised.

Out of all the functions performed by the verifier, only the requirement to establish the type safety of a block of code has a direct impact on the process of resolution of *TypeRef* and *MemberRef* tokens.

\(^{20}\) For completeness, it should be mentioned that individual metadata tokens are also validated during JIT compilation.
Relationship between Assemblies, Modules, Class Loaders, and Application Domains

Microsoft defines an application domain as a unit of isolation for an application. The isolation guarantees the following [32]:

- An application can be independently stopped.
- An application cannot directly access code or resources in another application.
- A fault in an application cannot affect other applications.

All assemblies loaded during execution are associated with an application domain. There is a one-to-one mapping between assemblies and ClassLoaders. In addition to providing the top-level resolution scopes during type loading, assemblies are also responsible for managing the manifest metadata and for managing a codebase - the default path from which modules are loaded.

In Rotor source code, assemblies are implemented as Assembly objects. Each Assembly object contains a pointer (m_pClassLoader) to a ClassLoader which stores further pointers to all assembly modules loaded so far. Each Assembly object also contains a direct pointer (m_pManifest) to the manifest module of the assembly.

This arrangement implies that assemblies cannot be loaded directly. The task of loading the Portable Executable binaries from disk images, prior to the operation of the ClassLoader, is divided into two sub-tasks:

- A physical disk-to-memory module loading process implemented via a platform-specific LibraryLoader

- A process of converting the information contained in an assembly reference (columns of the AssemblyRef metadata table21) into a loading path, known as assembly resolution

---

21 Only static references are stored in the AssemblyRef table. Dynamic references are also possible. They are discussed in Section 4.
Physical Module Loading

Physical module loading occurs whenever a `ClassLoader` is requested to load a type whose resolution scope is a `ModuleRef` token of a module that has not yet been loaded, or an `AssemblyRef` token of an assembly whose manifest module is not yet loaded. In the former case, the runtime handle for the newly loaded physical module is added to the `ClassLoader` of an existing Assembly object. In the latter, a new Assembly object is created, and the module’s runtime handle is added to both its `ClassLoader` and its manifest pointer. This mechanism employs the delayed loading approach where physical files are loaded on an as-needed basis.

Physical module loading is performed by a set of functions implemented by the underlying platform, thus isolating the CLR from the low-level implementation issues. The commercial .NET framework provides both the file-system-based and the network-based module loading facilities. An HTTP-based module loader forms a basis for the framework’s powerful network-centric software deployment model.

Physical module loading can be seen as a side-effect of the resolution of a `TypeRef` token to `TypeDef` token. Successful module loading guarantees that the module containing the `TypeDef` exists in a memory location pointed to by the runtime module handle stored in an assembly’s `ClassLoader`.

Assembly Resolution

Assembly resolution process occurs whenever a `ClassLoader` is requested to load a type whose resolution scope is an assembly for which a new Assembly object has not yet been allocated. Prior to the creation of such an object, and prior to the physical loading of the assembly’s manifest module, the assembly reference (`AssemblyRef` metadata token) must be resolved to a loading path\(^\text{22}\).

In both the commercial .NET Framework and the free-source Rotor distribution, the assembly resolution task is performed by a software component known as `Fusion`, contained in `fusion.dll`. Given an `AssemblyRef` token, `Fusion` returns the path\(^4\) to the correct version of the server assembly’s manifest module. The assembly resolution process can be controlled via the use of external configuration files.

\(^{22}\) The loading path can be a platform-specific path on the local computer, a network path, or a URL. The ability to specify the loading path as a URL is a part of the HTTP deployment model.
The following steps are performed when resolving an assembly reference (AssemblyRef token) [22]:

1. The correct assembly version is established by examining applicable configuration files in the following order[23]:
   a. Application – These configuration files are deployed with an application, are stored in the same subdirectory as the main executable component, and apply to all components that might potentially be loaded by the application.
   b. Publisher – These configuration files are distributed with components by component publishers, and are stored in the GAC. A publisher policy affects all applications that use a shared component covered by this policy. Publisher policies can be globally disabled.
   c. Machine – This is a single configuration file that resides on the local computer in a known location, and is used by administrators to override all other versioning policies.

2. A check is made to determine if assembly name has been bound to before and, if so, the previously loaded assembly\textsuperscript{24} is used.

3. The Global Assembly Cache is consulted

4. Assembly probing is performed using the following steps:
   a. If a dynamic reference was made with an URL-specified loading path and the configuration files do not affect the original reference, the runtime checks for the location hints
   b. If a codebase exists within the configuration files, the runtime checks only this location
   c. Checks for the assembly in the local subdirectories and in the subdirectories specified in the probing section of the configuration files

   These steps do not apply to private assemblies. Private assemblies are not versioned and are always located within a local probing path.

In addition to assembly resolution, Fusion is responsible for the installation of public assemblies on a computer system, and other functions related to GAC management. Fusion fully encapsulates the complete component versioning semantics of the CLR.

\textsuperscript{23} DejaVu.NET builds custom Application Configuration Files, as means of enabling Fusion to resolve server references to newer component versions that those specified by the AssemblyRef metadata tables

\textsuperscript{24} A reuse of an existing Assembly object is implied
Just-in-Time Compilation

Upon loading, each type is represented by an EEClass object containing a method table. All entries in the method table initially contain a pointer to the IL routine in the hosting module, and a native CALL instruction which transfers control to a routine known as a Universal JIT Thunk [29]. This is a small assembly routine that, in turn, transfers control to the JIT Compiler. The JIT Compiler looks up the pointer to the IL routine and generates native code. The method table entry is then updated to a pointer to the native routine, and this routine is called. The native routine can be directly accessed in all future calls of the same method. This technique reduces the overhead of JIT-compilation and allows JIT-compilation to cascade in complex programs.
The sub-processes of verification of IL instructions and of resolution and validation of any reference tokens that they embed proceed in parallel with the JIT code generation. All processes proceed one IL block at a time. The Rotor implementation of the CLI performs token resolution, validation, verification and code generation one IL instruction at a time, but, since ECMA 335 sets no explicit requirements, this is implementation-specific and any number of IL instructions can be JIT-ed at a time.

Once the JIT-compilation process reaches an instruction which generates a verification or a resolution failure, the native code generated for the method so far is discarded and replaced with a native stub which throws an exception. The consequential execution of this stub halts the execution of the entire application - all modifications to a component that ‘break then binary compatibility’ thus result in dynamic linking errors caught by the JIT-compiler.
Reference Token Resolution

All unresolved TypeRef and MemberRef tokens, encountered during the JIT-compilation process, are resolved into runtime handles.

The concept of token resolution is encountered in many different contexts. To avoid any ambiguity, Figure 6 illustrates the relationships between different forms of token resolution encountered during the dynamic linking.

![Diagram](example.png)

**Figure 6 – Dynamic linking modeled as side-effects of multiple layers of resolution**
IL instructions containing MethodRef tokens

Consistently with the available information sources, a MemberRef token is resolved by the VES into a field or a method runtime handle prior to accessing the field or calling the method. The resolution is performed via an intermediate step where the MemberRef of a client module is resolved to a Method or a Field metadata token of a server module. If the entry in the server’s Method or Field metadata tables cannot be found, the VES shall throw a System.MissingMethodException or a System.MissingFieldException.

A MemberRef token represents an entry into the MemberRef metadata table which contains three columns: Class (relates a MemberRef to a TypeRef), Name, and Signature. A MemberRef token can be successfully resolved to a Method or a Field token only if metadata table entries of the two tokens have identical Name and Signature columns. Otherwise, an exception is thrown.

The resolution of a MemberRef token requires that:

1. The TypeRef token related to the MemberRef via the Class column of the MemberRef table is resolved into a runtime type handle

2. All MemberRef tokens encoded within the MemberRef’s Signature are resolved into runtime type handles

3. If the MemberRef token refers to a method (i.e. prior to being fully resolved into a runtime handle, it is resolved into a Method token), all TypeRef tokens encoded within the stand-alone signature associated with this Method token are resolved into runtime type handles

If any of these TypeRef tokens are unresolved, suitable requests are made to the ClassLoader to load the types. Conceptually, this means that the resolution of a reference to an external field requires the resolution of references to both the type declaring the field, and the type of the field itself. Similarly, the resolution of a

25 To avoid the confusion caused by terminology adopted by different authors, a Method table is the same as a MethodDef table. A Field table is the same as a FieldDef table.

26 The MethodRef tables of the client component and the Method and Field tables of the server component are indirectly related via the TypeRef and TypeDef tokens for the reference and the definition of the type being accessed. The need to resolve a TypeRef prior to resolving a MemberRef is due to such an indirect relationship.

28 For field references, the Signature encodes the type of the field. For methods, it encodes the types of the result and parameters.
reference to an external method requires the resolution of references to the type
declaring the method, the method’s result type, the method’s parameter types, and the
types of local variables declared within the method.

The Shared-Source CLI (Rotor) implements MemberRef resolution via the
GetDescFromMemberRef method of an EEClass object.

**IL instructions containing TypeRef tokens**

JIT-compilation of IL instructions shown in Table 5 results in resolution of
TypeRef tokens into runtime type handles via a direct request to the ClassLoader to load
the types.
Type Safety Verification

Type safety is a property of a block of IL code which guarantees that such code accesses only the memory locations that it is authorized to access by the Common Type System. Type safety guarantees that all types are accessed in accordance with their contracts persisted in the metadata. When code is type safe, CLR can completely isolate assemblies from each other and increase application reliability. The concepts of type safety and of application domains are inter-related.

The verifier is conservative and it is possible that code known to be type-safe will still fail a verification phase [26].

The entire set of constraints that is placed on the permissible changes to a server component by the verification phase of the dynamic linking process can be partitioned into three subcategories – constraints introduced by the requirement to enforce visibility and accessibility contracts, constraints introduced by the requirement to enforce inheritance contracts, and miscellaneous constraints.

Constraints arising out of accessibility and visibility contracts

ECMA 335 [25] associates a visibility level with each top-level type declared in an assembly. The type is either exported (visible outside the assembly it is defined in) or it is not exported (visible only within its assembly).

Inner types and type members have an accessibility level - compiler-controlled, private, family, assembly, family-and-assembly, family-or-assembly or public. Consult ECMA 335 for details.

To ensure type safety, verifier ensures that a type member can be accessed only if the type is visible and the member is accessible. Verifier thus guarantees that all references to external types, fields and methods have definitions with visibility and accessibility flags which permit them to be accessed via such external references.

The Rotor source code and the dynamic linking tests performed using the commercial .NET framework’s profiling interface, confirm that no visibility checks are performed on IL instructions that embed TypeRef tokens. All visibility and accessibility contracts are enforced solely through the verification of IL instructions which embed MemberRef tokens (Table 7).
### Table 2 – IL instructions that accept a MemberRef token as an argument, and that verify the visibility and accessibility of the type member represented by this token

<table>
<thead>
<tr>
<th>Accessibility of External Method References Verified</th>
<th>Accessibility of External Field References Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Ldfld</td>
</tr>
<tr>
<td>Callvirt</td>
<td>Ldsfld</td>
</tr>
<tr>
<td></td>
<td>Ldsfld</td>
</tr>
<tr>
<td></td>
<td>Ldsflda</td>
</tr>
<tr>
<td></td>
<td>Stfld</td>
</tr>
<tr>
<td></td>
<td>Stsfld</td>
</tr>
</tbody>
</table>

This design decision improves performance by reducing the verification overheads, but it can lead to fragile clients. Visibility contracts between the type reference and the type definition can be broken for any of the `TypeRef` tokens that are held by a component which holds no `MemberRef` tokens to any member of such a type. These changes would remain undetected during the dynamic linking process, but would cause static errors on re-compilations. Potential fragile clients include clients that contain types that inherit from external types without accessing their methods, clients which use an external type reference during casting operations, and clients which catch externally defined exceptions.

An interesting behavior of the CLR verifier is observed whenever one of the IL instructions in Table 7 embeds a `MemberRef` token whose matching `Method` or `Field` token is declared within a nested type. In this situation, the requirement to enforce the accessibility constraints causes the loading of all enclosing classes via requests to the `ClassLoader`. The type loading process iterates until all enclosing types, ending with the top-level type, have been loaded. The accessibility and visibility constraints of all loaded classes are consulted prior to granting the server component a permission to access the field or call the method.

If the verification process determines that the visibility and accessibility constraints associated with type and type member definitions within a server are such that a referenced field or method must not be accessible by a client component, the IL code block that is currently JIT-ed will be replaced by a native code block that throws a `System.FieldAccessException` or a `System.MethodAccessException`.

---

29 Fragile clients are defined as clients that, after modifications to server components, continue to correctly link to server components, but which cannot be recompiled – evolution of server components preserves binary compatibility, but not the source compatibility relative to fragile clients.
Constraints arising out of inheritance contracts

The CLR verifier also enforcing the inheritance contracts between types. An inheritance contract is successfully enforced whenever the type of an object reference (located on the stack) is found to be assignment-compatible to a type reference related to the metadata token supplied as the instruction argument.

The Open-Source CLI checks assignment-compatibility through a `CanAssign` method that accepts an object reference and a type reference (`TypeRef` token) as arguments. Invocation of this method results, via a dynamic callback over several layers of indirection, in a call to the `CanCastTo` method of the `ClassLoader`.

Consistently with the verification theory and the dynamic linking tests, `CanCastTo` requires the supplied type reference to be resolved to a runtime type handle prior to the assignment-compatibility check. Should the reference be unresolved, type loading is requested.

If the type reference is resolved to a handle of an interface type, `CanCastTo` determines whether the object reference implements this interface. Otherwise, `CanCastTo` determines whether the object reference is the same as or a subtype of the resolved type.

Table 6 lists the opcodes of the IL instructions that are subject to inheritance contract verification, relating them to the stack object references that are verified during the verification of each such instruction. Method result types are not verified unless assigned to fields or local variables.

<table>
<thead>
<tr>
<th>IL Instruction Opcode</th>
<th>Object References that are Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Method arguments</td>
</tr>
<tr>
<td>Callvirt</td>
<td>Method arguments</td>
</tr>
<tr>
<td></td>
<td>Method receiver (this)</td>
</tr>
<tr>
<td>Ldfld</td>
<td>Field access receiver</td>
</tr>
<tr>
<td>Ldflda</td>
<td>Field access receiver</td>
</tr>
<tr>
<td>Stfld</td>
<td>Field type</td>
</tr>
<tr>
<td></td>
<td>Field assignment receiver</td>
</tr>
<tr>
<td>Stsfld</td>
<td>Field type</td>
</tr>
<tr>
<td>Newobj</td>
<td>Constructor arguments</td>
</tr>
</tbody>
</table>

Table 6 – A listing of stack object references which are subject to inheritance contract verification and the IL instructions causing this verification

---

30 See file `fjitcompiler.cpp`