Miscellaneous constraints

Table 7 lists the instruction-specific constraints unrelated to either the accessibility or the inheritance contracts.

<table>
<thead>
<tr>
<th>IL Opcode</th>
<th>Token Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>MemberRef</td>
<td>Method must not be abstract</td>
</tr>
<tr>
<td>Box</td>
<td>TypeRef</td>
<td>Type must be a value type</td>
</tr>
<tr>
<td>Unbox</td>
<td>TypeRef</td>
<td>Type must be a value type</td>
</tr>
<tr>
<td>Ldvirtftn</td>
<td>MemberRef</td>
<td>Function (global method) must be non-static and non-abstract</td>
</tr>
<tr>
<td>Ldelema</td>
<td>TypeRef</td>
<td>Array element type(^{31}) must match TypeRef</td>
</tr>
<tr>
<td>Ldobj</td>
<td>TypeRef</td>
<td>Type must be a value type</td>
</tr>
<tr>
<td>Sizeof</td>
<td>TypeRef</td>
<td>Type must be a value type</td>
</tr>
<tr>
<td>Ldflda</td>
<td>MemberRef</td>
<td>An initonly field is accessed correctly</td>
</tr>
<tr>
<td>Ldsflda</td>
<td>MemberRef</td>
<td>An initonly field is accessed correctly</td>
</tr>
<tr>
<td>Stflda</td>
<td>MemberRef</td>
<td>An initonly field is accessed correctly</td>
</tr>
<tr>
<td>Stsflda</td>
<td>MemberRef</td>
<td>An initonly field is accessed correctly</td>
</tr>
</tbody>
</table>

Table 7 – Miscellaneous constraints introduced by the verification phase

---

\(^{31}\) Verifier enforces the requirement that the type of array elements is exactly the same as the type referenced by the TypeRef token. A subtype relationship is not accepted.
Dynamic Linking as an Implementation of Dynamic Component Composition

The .NET dynamic linking process is a physical implementation of the concept of dynamic component composition that is frequently encountered in theoretical work on component models [15, 19, 28, 39]. The act of composing a number of components into an application thus corresponds to a number of references token resolution processes, each of which operates within a context of an individual component pair. The in-depth understanding of the dynamic linking process can now be applied to reason about binary compatibility, component interfaces, application composition, and dependencies.

It should be noted that, unlike a number of theoretical models where components are completely unaware of each other prior to composition, .NET components have knowledge of the server components that they depend on and put constraints on the possible server components that can satisfy their requirements. As such, the .NET component model provides no implementation of the theoretical notion of component connectors – they are replaced by the TypeRef and MemberRef metadata tables, and by the dynamic linking process itself.

Section Summary

- The .NET dynamic linking process is implemented via a number of inter-related sub-processes – assembly resolution, library loading, class loading, JIT-compilation, and verification – which are driven by metadata token resolution.

- Versioning semantics are encapsulated by Fusion.

- TypeRef (references to imported types) and MemberRef (references to members of imported types) metadata tokens are subject to server-side existence checks

- Beyond simple existence checks, Class Loader and Verifier impose additional constraints on server-side definitions of imported types and type members

- Unlike JVM[17], CLR verifies IL instructions during the JIT-compilation of method bodies.
SECTION 3 – IMPORT interfaces, EXPORT interfaces, and Binary Compatibility

Introduction

In this report section, the findings of Section 2 are used to reason about the requirements placed by the client-side reference tokens on the server-side definitions, and the effects of modifications of such definitions on the outcome of the dynamic linking process. The section ends with a derivation of the Relative Binary Compatibility Detection Algorithm, and the Absolute Binary Compatibility Detection Algorithm.

Partition I of the CLI specification defines contracts as the shared assumptions on a set of signatures between all implementers and all users of the contract[25]. The signatures are the part of the contract that can be checked and mechanically enforced.

The idea of contracts and signatures, defined on types, is extended to components that embed, and serve as a resolution scope for, these types. The dynamic linking process between a client and a server component can thus be conceptualized as a process of run-time enforcement of a contract between the two components via an examination of the signatures embedded in the two components. However, the examination of these signatures – the IMPORT and the EXPORT interfaces – can only put syntactic guarantees on a contract. It is unable to place any behavioral and concurrency guarantees [41], which are needed to ensure full compatibility.

This project assumes that contracts between components embody no class inheritance or interface inheritance and implementation contracts between the types declared within such components. This restriction implies that types declared within components are related solely by associations.

32 The idea of Programming by Contract, first introduced in the Eiffel programming language [48], is used by researchers in the component-based software engineering field [47] in a range of contexts. Since this project is interested in purely syntactical guarantees, contracts between components are defined in terms of metadata-embedded signatures.

33 Due to this restriction, no investigation is made into the CLI manifestation of the Fragile Base Class Problem.

34 Expressed in terms of the physical module view of a component, this limitation states that no TypeRef metadata tokens are contained in TypeDef and InterfaceImpl metadata tables of a component.
Representing Component Evolution

Both the monolithic and the component-based software systems need to undergo continuous evolution. Component-based systems have the distinct advantage arising from the ability to evolve individual components in isolation.\(^{35}\)

Let \(\epsilon\) represent a single modification made to a single component.

Components are represented by bold lowercase characters.

Let Components represent a hypothetical set of all possible CLI components.

The ‘ symbol distinguishes different versions of a component. A modification is assumed to increment the component version number. e.g. \(a'\) represents \(a\) after a modification.

Let \(\mathcal{C}\) represents a hypothetical set of all single-step component modifications

\[ \forall \epsilon \in \mathcal{C} \]

e.g. \{addition of type \(T\), removal of type \(T\), modification of a public field \(F\) in type \(B\) nested in type \(A\) into private field \} \(\subseteq \mathcal{C}\)

Define \(\rightarrow\) operator:

\[ (b \rightarrow \epsilon \rightarrow b') \leftrightarrow (a \text{ single modification } \epsilon \text{ evolves a component } b \text{ into } b') \]

Version number of \(b'\) is implicitly assumed to be higher than version number of \(b\)

Define \(\rightarrow^*\) operator:

\[ (b \rightarrow \epsilon^* \rightarrow b') \leftrightarrow (a \text{ number of consecutive modifications } \epsilon^* \in \mathcal{C} \text{ evolve a component } b \text{ into } b') \]

\[ \Diamond (\epsilon^* = \epsilon \lor \epsilon^* = 0) \]

Further formalizations are outside the scope of this project. The informal definitions are sufficient.

\(^{35}\) The relationship between the evolution of a single component and the evolution of a complete component-based system is modeled in Section 4
Re-considering Binary Compatibility

Throughout current research, binary compatibility is considered in absolute terms as a consequence of the evolution of an individual component. It is assumed that client components make references to the entire contents of each component – a worst-case scenario. In fact, each client references only a subset of all types and type member definitions of a server component. Effectively, a change determined to ‘break binary compatibility’ by an absolute binary compatibility detection algorithm has no impact on a number of clients of the evolved component. This idea leads to the notion of relative binary compatibility, where binary compatibility between subsequent component updates is evaluated relative to each of its existing clients. A clear distinction between the two types of binary compatibility is necessary.

Let \( a \) represent a client component and let \( b \) represent a server component where \( b \rightarrow \mathcal{E} \rightarrow b' \).

Define

\[
\text{absoluteBC}(b, b') \iff \forall a: \text{possible_clients}(b). [\text{successful_dynamic_linking}(a,b) \rightarrow \text{successful_dynamic_linking}(a,b')]
\]

\[
\text{relativeBC}(b, b', a) \iff (\text{successful_dynamic_linking}(a,b) \rightarrow \text{successful_dynamic_linking}(a,b'))
\]

For example, consider three components \( a, b, \) and \( c \) where \( b \) and \( c \) are clients of \( a \), and \( c \) defines a type \( T \) which is referenced by \( b \) and not referenced by \( c \). Consider a situation where \( a \rightarrow \mathcal{E} \rightarrow a' \) and \( \mathcal{E} = \{\text{removal of type } T\} \). Now \( \text{relativeBC}(a, a', b) \) is true whereas \( \text{relativeBC}(a, a', c) \) is false. The absolute binary compatibility assumes a worst case scenario – both \( \text{relativeBC}(a, a', b) \) and \( \text{relativeBC}(a, a', c) \) are false.

\[36\] For now, \( \text{possible_clients}(1) \) is informally defined as a hypothetical set of all possible clients of a server component.

\[37\] \( \text{successful_dynamic_linking}(a,b) \) implies that no dynamic linking failures are observed whenever the component \( a \) is executed and the component \( b \) is loaded as a result of this execution.
Effects of CLI’s Delayed Loading and Linking

Despite the widespread use of Def 3.1, Def 3.1 and Def 3.2 carry an inherent weakness. Like most component frameworks, the .NET framework uses delayed loading. By enforcing the requirement that all components conform to the Common Type System, and by representing all external type and type member references by metadata tokens, the CLI increases the granularity of the delayed dynamic loading and linking - during execution, the reference tokens are resolved on an as-needed basis.

In many situations, a binary incompatible modification to a server component will not be detected by a client component until a particular execution path is exercised. When using such ‘timebomb servers’\(^\text{38}\) in application composition, a number of problems can remain undetected until the JIT-compilation of a particular method. The hidden problems can even persist over a number of evolutionary modifications of the server. Reliance on a component-based system built out of such components is clearly against the basic principles of software engineering.

Informally define a new predicate to handle the increased granularity - from complete libraries\(^\text{39}\) to individual types and type members - of the delayed loading and linking processes:

\[
\text{successful_reference_resolution}(a, b) \leftrightarrow \text{Def 3.3} \\
\quad \text{(all TypeRef and MemberRef tokens embedded in } A \text{ can be successfully resolved intoTypeDef, Member and Field tokens embedded in } B)
\]

\text{Def 3.3} does not force \(a\) to exercise all execution paths which would result in the resolution of all reference tokens. It only guarantees that, for any possible execution path in \(a\), all reference tokens encountered during a JIT-compilation process will be successfully resolved\(^\text{40}\).

\(^{38}\) A provisional term introduced by this report to represent situations where a breakage of binary compatibility is undetected by a client during its most frequent execution paths.

\(^{40}\) This definition avoids the requirement to formalize the effect of particular method executions on the metadata token resolution process.
Re-define binary compatibility\(^{41}\):

\[
\text{absoluteBC}(b, b') \leftrightarrow \quad \text{Def 3.4} \\
\forall a: \text{possible_clients}(b), [\text{successful_reference_resolution}(a, b) \rightarrow \text{successful_reference_resolution}(a, b')] \\

\text{relativeBC}(b, b', a) \leftrightarrow (\text{successful_reference_resolution}(a, b) \rightarrow \text{Def 3.5} \text{successful_reference_resolution}(a, b'))
\]

**Benefits of the Component Signature Matching Approach**

Definitions Def 3.4 and Def 3.5 are sound, but are of little practical use unless a mechanical method for evaluating the predicate \text{successful_reference_resolution}(a, b), when presented with a client component \(a\) and a server component \(b\), is devised. In practical terms, even when the versioning policy of a software system is set manually, a system administrator needs a means to evaluate whether the new version of a server component (just delivered by a component maintainer) will ‘successfully link’ against each of the existing clients.

Attempts to evaluate \text{successful_reference_resolution}(a, b) via an execution of the client component \(a\) are flawed both in theory and in practice, and hence cannot form a basis of a binary compatibility detection tool. Some of the problems of this approach are:

- There are no guarantees that all reference tokens are resolved during the execution
- Dependency\(^{42}\) interactions – execution cannot be constrained to a single client and a single server as either of these components can attempt to resolve references to types and type members defined in further components
- Theoretical program execution problems
  - There are no guarantees that every method that is called will terminate – Halting Problem
  - No performance bounds can be put on such an algorithm
  - User-driven programs require a simulation of user actions

The alternative means to evaluate \text{successful_reference_resolution}(a, b) is through offline matching of component signatures – the IMPORT and EXPORT interfaces. The interfaces isolate the concepts of type and type member references (IMPORT) and declarations (EXPORT) from the concept of program execution paths (as dictated by the method bodies embedded in a component).

\(^{41}\) The given definition of binary compatibility is based on an understanding of the physical token resolution process modeled in Section 2. A formal definition of CLI binary compatibility can be derived by treating binary modules as fragments and applying a fragment calculus[46].

\(^{42}\) The notion of dependency is formalized in Section 4
Re-conceptualizing .NET Components

The concepts of IMPORT and EXPORT interfaces contradict with the common conceptual view of a .NET component where a component is expressed in terms of a manifest (assembly metadata), the type metadata, and the method bodies.

Under this standard conceptual view, the execution of a component a can lead to the loading\(^{43}\) of a component b where \(\neg\)\text{successful_reference_resolution}(a,b). Potential problems occur due to the assembly resolution phase – path resolution is currently performed solely through the assembly’s manifest declaration. The assembly manifest limits the import interface of the component to a set of \text{assembly extern} directives, each of which is associated with a set of properties (including name, version, and public key) mapped to a row in the AssemblyRef metadata table. Similarly, the assembly manifest limits the export interface of the component to a single \text{assembly extern}, associated with a set of properties mapped to a single row in the Assembly metadata table. Since the path resolution process does not take into account any metadata tables other than Assembly and AssemblyRef, there are no guarantees that \text{successful_reference_resolution}(a,b).

To introduce a syntactical guarantee - \text{successful_reference_resolution}(a,b) – the path resolution process must also take into account the components’ type metadata. The first step towards an association of syntactical guarantees with a path resolution process is taken with the re-partitioning of a component’s metadata. The resulting conceptual view of a component is illustrated in Figure 7.

\(^{43}\) Physical module loading by the \text{LibraryLoader}
Figure 7 illustrates the process of splitting the entire metadata section of the component (i.e. the metadata tables containing both the manifest and the type metadata) into three sections:

- The component’s single EXPORT interface that includes metadata relating to definitions of types and type members exported from the component

- A number of IMPORT interfaces (one interface for each server component that the component depends on) that include all the metadata relating to references to external types and type members

- Other metadata – information about the component itself, and definitions of private and internal types and type members

Note:

- In the purely syntactical analysis, IL method bodies do not form a part of a component’s EXPORT interface

- IMPORT and EXPORT interfaces are vital and indivisible contents of a component. Since the component is the interface, the idea of conformance of a component to an interface[28] is not applicable
The following equivalence arises out of the new conceptual view:

\[
\text{successful_reference_resolution (a,b)} = \text{Def 3.6} \quad \text{successful_signature_match (IMPORT(a,b), EXPORT(b))}
\]

Now:

\[
\text{absoluteBC(b, b')} \iff \text{Def 3.7} \quad \forall a:\text{possible_clients}. [\text{successful_signature_match (IMPORT(a,b), EXPORT(b))} \implies \text{successful_signature_match (IMPORT(a,b), EXPORT(b))}]
\]

\[
\text{relativeBC(b, b', A)} \iff \text{Def 3.8} \quad (\text{successful_signature_match (IMPORT(a,b), EXPORT(b))} \implies \text{successful_signature_match (IMPORT(a,b), EXPORT(b'))})
\]

\[
\text{IMPORT(2), EXPORT(1) and successful_signature_match(2) must be defined.}
\]

An informal definition, sufficient to enable further discussion, is given.

### The IMPORT interface

The predicate IMPORT(a,b) refers to the IMPORT interface declared within a component a which specifies the signatures of types and type members that are imported from a component b. Each component can contain a number of IMPORT interfaces – one IMPORT interface per server component that it depends on.

IMPORT(a,b) consists of all the information extracted from the TypeRef and MemberRef metadata tables of a component a which is used by a to resolve references to types and type members exported from b. The TypeRef and MemberRef metadata tables and tokens are automatically embedded into the physical component modules by the IL Assembler or a CLR-compliant compiler. They have no corresponding representation in the ILASM grammar.

Each TypeRef token in the IMPORT interface conceptually represents either an external type which is associated to a local type, an external base type which a local type inherits, or an external interface which a local interface inherits or implements. This project assumes that type references in an IMPORT interface arise solely out of associations.

---

44 The notion of dependency is clarified in Section 4
45 Consult CLI Specifications [26] for the complete definition of contents of TypeRef and MemberRef tables
The IMPORT interface embodies the concept of clients’ awareness\textsuperscript{46} of which servers they require, which requirements\textsuperscript{47} (in terms of client-side type and type member signatures) they expect these servers to satisfy, and which versions of these servers are ‘known’ to satisfy these requirements.

The IMPORT interface captures the notion of client re-compilation after the evolution of a server. Given a client $a$, a server $b$, and knowledge that $b \rightarrow e^* \rightarrow b'$ and $\text{relativeBC}(b,b',a)$:

- Interface of a client which is ‘unaware’ of the new server version is expressed as $\text{IMPORT}(a,b)$

- The act of re-compilation embeds the ‘awareness’ of the new server version into the client. This ‘new awareness’ is captured by expressing the client’s interface as $\text{IMPORT}(a,b')$

If $\neg \text{relativeBC}(b,b',a)$, client re-compilation would fail and no ‘new awareness’ would be embedded into the client. Compilation is the ultimate check for maintenance of binary compatibility.

\textsuperscript{46} Appendix A illustrates the problem of ‘Loss of Awareness’ in which, informally, clients are not aware of the servers they indirectly depend upon.

\textsuperscript{47} The standard .NET conceptual component view does not capture these requirements. In this standard view, clients are aware only of the existence of servers and of a server version known to provide successful reference token resolution.
The EXPORT interface

The predicate \( \text{EXPORT}(b) \) refers to the single\(^{48}\) EXPORT interface of a component \( b \).

The EXPORT interface consists of signatures of types that the component \textit{exports}. This information is primarily collected from the \textit{TypeDef}, \textit{Field}, and \textit{Method} metadata tables of the component\(^ {49} \).

Unlike the IMPORT interfaces, the EXPORT interface of a component can be mapped to a higher-level ILASM representation. The ILASM constructs\(^ {50} \) expose the signatures, and hence the definitions, of types and type members (including global methods and fields for non-OO programming languages) that are accessible outside the component.

\(^{48}\) A component is ‘unaware’ of its clients

\(^{49}\) A number of other metadata tables, such as the InterfaceImpl and the NestedClass tables, are consulted when extracting an EXPORT signature

\(^{50}\) Consult ECMA Spec II for the complete definition of ILASM grammar relating to types and type members [26].
Signature Matching and the SUBSET Function

IMPORT(a,b) specifies a set of shared assumptions, or requirements, that must be met by EXPORT(b) in order to guarantee successful_reference_resolution(a,b).

In theory, the information contained within IMPORT(a,b) should be sufficient to define a single hypothetical component H whose EXPORT interface contains only those signatures that are sufficient to satisfy all of the requirements set out by IMPORT(a,b). The process of signature matching can be re-expressed as a process of ensuring that EXPORT(b) embeds, in an identical signature format, all of the definitions contained within EXPORT(H).

To clarify this concept, for a client component a and a server component b, SUBSET(EXPORT(b), IMPORT(a,b)) is defined as a subset of interface EXPORT(b) which contains server-side signatures for types and type members whose client-side signatures are contained in IMPORT(a,b). Conceptually, the SUBSET relation returns the portion of the server's EXPORT interface which is visible, and has an impact upon, a particular client of this server. A client a is ‘unaware’ of any definitions within (EXPORT(b) - SUBSET(EXPORT(b), IMPORT(a,b))).

The informal definition of SUBSET permits IMPORT(a,b) to reference types or type members not defined in b – their signatures are not contained in SUBSET(EXPORT(b), IMPORT(a,b)). In such cases, relativeBC is broken.

SUBSET function is illustrated in Figure 8.
Given the hypothetical component H containing definitions that are both necessary and sufficient to satisfy\(^{51}\) all of the requirements set out byIMPORT\((a,b)\),

\[
\text{EXPORT}(H) = \text{SUBSET}(\text{EXPORT}(b), \text{IMPORT}(a,b)) \rightarrow \text{successful\_reference\_resolution}\ (a,b)
\]

The SUBSET function is also used to, given a set \(e^*\) of all changes made to a server component, find a subset consisting of changes which affect a particular client component. Given \(b \rightarrow e^* \rightarrow b'\), all modifications \(e \cdot e^*\) which result in a change to \(\text{SUBSET}(\text{EXPORT}(b), \text{IMPORT}(a,b))\) are represented as a set \(e^*[a]\).

\[
(b \rightarrow e^* \rightarrow b') \rightarrow (e^*[a] = \{e \cdot e^* | (b \rightarrow e \rightarrow b'') \rightarrow \text{SUBSET}(\text{EXPORT}(b), \text{IMPORT}(a,b)) \neq \text{SUBSET}(\text{EXPORT}(b''), \text{IMPORT}(a,b)))\}
\]

Table 8 shows how, given \(b \rightarrow e^* \rightarrow b'\) and \(\text{relativeBC}(b,b',a)\), SUBSET function captures the server-side effect of client re-recompilation after an evolution of the server.

<table>
<thead>
<tr>
<th>Action performed</th>
<th>Representation of client-visible SUBSET of server EXPORT interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>(\text{SUBSET}(\text{EXPORT}(b), \text{IMPORT}(a,b)))</td>
</tr>
<tr>
<td>Server evolved and recompiled</td>
<td>(\text{SUBSET}(\text{EXPORT}(b'), \text{IMPORT}(a,b)))</td>
</tr>
<tr>
<td>Client recompiled</td>
<td>(\text{SUBSET}(\text{EXPORT}(b'), \text{IMPORT}(a,b')))</td>
</tr>
</tbody>
</table>

**Table 8 - SUBSET and client recompilation**

\(^{51}\) \text{successful\_reference\_resolution}\ (A,H) holds
Limitations to Signature-matching Imposed by the .NET Framework Metadata APIs

In comparison to a number of competing component frameworks, the Managed Reflection API and the Unmanaged Metadata API of the .NET framework provide an extremely powerful set of component introspection features. These features are still limited solely to the ability to retrieve the contents of the metadata tables stored in a component. It was shown in Section 2 that, within a physical module, the TypeRef and MemberRef tokens can also exist outside the metadata tables, as arguments to IL instructions. Unless a low-level IL method body inspection API is developed as part of this project, it is not possible to extract information regarding the full context of resolution of each reference to an external type or an external type member.

This limitation implies that it is impossible to constrain a hypothetical set of all possible .NET components into a single hypothetical component H where EXPORT(H) is sufficient to satisfy all of the requirements imposed by the interface IMPORT(a,b). With no information regarding the context of metadata token resolution, the logical formula

\[
\text{successful\_reference\_resolution (a,b)} \rightarrow \\
\text{successful\_signature\_match (IMPORT(a,b), EXPORT(b))}
\]

will always hold, but the logical formula

\[
\text{successful\_reference\_resolution (a,b)} \leftarrow \\
\text{successful\_signature\_match (IMPORT(a,b), EXPORT(b))}
\]

will not hold in a number of scenarios.

As an example of such a scenario, consider an IMPORT interface containing the signature of an imported field. Such a signature does not specify a resolution context which, in this case, is an IL opcode. So, with no IL opcodes to distinguish their resolution contexts, references to instance fields and references to static fields have an identical representation in an IMPORT interface. Yet, dynamic linking will fail if the server defines an abstract field and the client resolves the field reference as an instance method.

As a result of these limitations, it is impossible to use the Metadata APIs and a signature matching algorithm to place syntactical guarantees on the component composition process.
Evaluating Binary Compatibility in Terms of Changes Made To a Component

This project aims to place syntactical guarantees on component modifications - there is no explicit need to evaluate successful_reference_resolution \((a,b)\) directly. Instead, evaluation of relativeBC\((b,b',a)\) and absoluteBC\((b,b')\) given \(b \rightarrow \epsilon \rightarrow b'\) calls for the development of mechanical means to evaluate

\[
\begin{align*}
\text{successful_reference_resolution (a,b)} & \rightarrow \\
\text{successful_reference_resolution (a,b')} & 
\end{align*}
\]

Knowing that \text{successful_reference_resolution (a,b)} enables the evaluation of relativeBC\((b,b',a)\) without a requirement for IMPORT\((a)\) to explicitly specify the full context of metadata token resolution. For example, if a reference to a field defined in a type exported from \(b\) was resolved correctly, it can be inferred that making the field static (if it is currently defined within \(b\) as an instance field) will result in dynamic linking errors. The knowledge of the exact resolution context of the field is not necessary.

In general, knowing that \text{successful_reference_resolution(a,b)} is true implies that EXPORT\((b)\) satisfies the requirements imposed by IMPORT\((a,b)\). relativeBC\((b,b',a)\) can be evaluated by a process of identification of differences in EXPORT\((b)\) and EXPORT\((b')\) and reasoning about the impact of such differences on the ability of EXPORT\((b')\) to satisfy a client-side signature which EXPORT\((b)\) is known to satisfy.

Since relative binary compatibility detection is reduced to a process of evaluating the impact of changes made to a component’s EXPORT interface, the informational content of IMPORT interfaces can be safely minimized. IMPORT\((a,b)\) has to carry only the minimum information necessary to, given EXPORT\((b)\), fully specify SUBSET(EXPORT\((b)\), IMPORT\((A, B)\)). This information is shown in Table 9. Further discussion adopts this reduced-information definition of an IMPORT interface.
The method of reduction of the content of IMPORT interfaces can be justified by consulting ECMA 335 which puts the following restrictions on metadata table rows:

- There shall be no duplicate rows in the TypeDef table, based on Namespace+Name.
- There shall be no duplicate rows in the Field table, based upon owner+Name.
- If this is a nested type, there shall be no duplicate rows, based upon Namespace+Name+OwnerRowInNestedClassTable.
- There shall be no duplicate rows in the Method table, based upon owner+Name+Signature, where the Type defined in the signatures shall be different and the return type of the method is ignored.

The process of evaluating the impact of changes made to a component’s EXPORT interface is conceptualized into a binary operator $\otimes$, defined between two EXPORT interfaces.

Informally, for a component $b$, $\text{EXPORT}(b) \otimes \text{EXPORT}(b')$ holds if, and only if, all type and type member declarations in $\text{EXPORT}(b)$ also exist in $\text{EXPORT}(b')$, and all contracts that were honored by $\text{EXPORT}(b)$ are also honored by $\text{EXPORT}(b')$.

In terms of the evolution of a component $b$ where $b \to \theta^* \to b'$, $\text{EXPORT}(b) \otimes \text{EXPORT}(b')$ holds if $\theta^*$ is a subset of a set of legal delta functions.

---

52 CLI specifications permit compilers to override these requirements when adding rows to Field and Method tables, but this project will assume full compliance to the standard CLI specs.
Using the informal definition of $\exists$, given that $b \rightarrow \epsilon \rightarrow b'$,

\[
\text{relativeBC}(b,b',a) \iff (\text{successful_reference_resolution}(a,b) \land \text{cond3.1}) \land (\text{SUBSET(EXPORT}(b), \text{IMPORT}(a,b)) \land \text{cond3.2}) \land \text{SUBSET(EXPORT}(b'), \text{IMPORT}(a,b)) \}
\]

If $b \rightarrow \epsilon \rightarrow b'$, $b' \rightarrow \epsilon \rightarrow b''$ and $\text{relativeBC}(b,b',a)$, checking whether $\text{relativeBC}(b',b'',a)$ requires checking only $\text{cond3.2} - \text{cond3.1}$ is already satisfied by Def 3.5.

Since a always has to be compiled against a version of b and the static checks performed by a CLR-compliant compiler are a superset of dynamic checks made during the dynamic linking,

\[
\exists b. [ \text{successful_reference_resolution}(a,b) ]
\]

Def3.9 is referred to as a compiler-enforced base case.

The relative binary compatibility detection approach taken by this project would fail without this static compiler-enforced check. This limits its applicability to component frameworks where client components embed a degree of ‘awareness’ of server components that they depend on. In the .NET component framework, this ‘awareness’ is represented by the statically-generated IMPORT interfaces.

Figure 9 illustrates the role of $\exists$ in evaluating

\[
(b \rightarrow \epsilon \rightarrow b') \rightarrow \text{relativeBC}(b, b', a)
\]
DejaVu.NET – A Framework for Evolution of Component-based Software Systems

Figure 9 – The role of the $\Rightarrow$ operator in evaluating whether $b \rightarrow e^* \rightarrow b'$ implies $\text{relativeBC}(b, b', a)$

Def3.8 can be used to prove that absolute binary compatibility can be fully expressed via the $\Rightarrow$ operator.

Using $\Rightarrow$ to Express Absolute Binary Compatibility

Whenever $b \rightarrow e^* \rightarrow b'$ and $\text{absoluteBC}(b, b')$, $b'$ preserves the ability to satisfy a set of contracts between itself and a hypothetical set of all of its possible clients. The set of modifications $e^*$ is such that $\text{EXPORT}(b')$ satisfies all of the client IMPORT interfaces that $\text{EXPORT}(b)$ satisfies.

Define:

$$\text{possible_clients}(b) = \{ c: \text{Component} \mid \text{successful_signature_match (IMPORT}(c, b), \text{SUBSET(EXPORT}(c), \text{IMPORT}(c, b))) \}$$

Def3.10 is equivalent to:

$$\text{possible_clients}(b) = \{ c: \text{Component} \mid \text{successful_reference_resolution}(c, b) \}$$

Given $b \rightarrow e \rightarrow b'$,

$$\text{absoluteBC}(b, b') \iff |\text{possible_clients}(b)| = |\text{possible_clients}(b')|$$
Using Def3.4, Def3.5, and Def3.8:

\[
\text{absoluteBC}(b, b') \iff \\
\forall a: \text{possible_clients}(b), \ [\text{relativeBC}(b,b',a)] \iff \\
\forall a: \text{possible_clients}(b), [\text{successful_reference_resolution}(a,b) \land \\
\text{SUBSET}(\text{EXPORT}(b), \text{IMPORT}(a, b)) \supseteq \text{SUBSET}(\text{EXPORT}(b'), \text{IMPORT}(a, b))]
\]

The predicate \text{successful_reference_resolution}(a,b) is superfluous by the definition of \text{possible_clients}(b) - no compiler-enforced base case is required to deduce that absolute binary compatibility is maintained.

\[
\therefore \text{absoluteBC}(b, b') \iff \\
\forall a: \text{possible_clients}(b), [\text{SUBSET}(\text{EXPORT}(b), \text{IMPORT}(a, b)) \supseteq \\
\text{SUBSET}(\text{EXPORT}(b'), \text{IMPORT}(a, b))]
\]

But, a subset of \text{EXPORT}(b) seen by all possible hypothetical clients is equivalent to \text{EXPORT}(b) itself.

\[
\therefore \text{absoluteBC}(b, b') \iff \ \
\text{Def3.13} \\
\text{EXPORT}(b) \supseteq \text{EXPORT}(b')
\]

\[
\therefore \text{Multiple ways of viewing binary compatibility:}
\]

\[
\text{absoluteBC}(b, b') \iff \\
(|\text{possible_clients}(b)| = |\text{possible_clients}(b')|) \iff \\
\text{EXPROTS}(b) \supseteq \text{EXPROTS}(b') \iff \\
\forall a: \text{possible_clients}(b), [\text{successful_reference_resolution}(a,b) \to \\
\text{successful_reference_resolution}(a,b')]
\]

It has been demonstrated how an application of the \supseteq operator to \text{EXPORT} interfaces of subsequent component versions can guarantee absolute binary compatibility, and how it can (given a compiler-enforced base case – Def3.8) guarantee relative binary compatibility. It is now necessary to define \supseteq itself and refine this definition into an algorithm.
Defining \( \supset \)

Let \( \mathcal{L} \) represent a hypothetical set of legal modifications made to a component

\[
\mathcal{L} \subseteq \mathcal{E}
\]

Evaluating \( \text{cond}2 \) in Def3.8 reduces to checking \( \mathcal{E}^*[a] \subseteq \mathcal{L} \)

Similarly, evaluating Def3.13 reduces to checking \( \mathcal{E}^* \subseteq \mathcal{L} \)

In both cases, it is computationally more optimal (worst-case complexity is reduced from \( O(|\mathcal{L}|) \) to \( O(1) \)) to search for counterexamples.

\[\therefore \] Evaluating \( \text{cond}3.2 \) in Def3.8 reduces to checking \( \neg \exists \mathcal{E}: \mathcal{E}^* [\mathcal{E} \notin \mathcal{L}] \) \hspace{1cm} \text{Def3.14} \]

In words, given an EXPORT interface of a component before the evolution and the EXPORT interface of the component after the evolution, find a single change between the two EXPORT interfaces which is known to affect binary compatibility. If a single such change is found, absolute binary compatibility is broken.

\[\therefore \] Evaluating Def3.12 reduces to checking \( \neg \exists \mathcal{E}: \mathcal{E}^*[a] [\mathcal{E} \notin \mathcal{L}] \) \hspace{1cm} \text{Def3.15} \]

In words, given the knowledge that all reference tokens held by the client component can be successfully resolved against the pre-evolution server version, and given an EXPORT interface of a component before the evolution, the EXPORT interface of the component after the evolution and the IMPORT interface of a client of this component, find a single change between the client-visible subsets of the two EXPORT interfaces which is known to affect binary compatibility. If a single such change is found, relative binary compatibility is broken.

All that remains to be done before \( \supset \) can be fully defined is a complete specification of each modification which is a member of the set of all forbidden modifications \( \{ \mathcal{E} - \mathcal{L} \} \).

A modification is a \( \Delta \)-function operating on EXPORT interfaces of two different versions of a component. Given \( (b \rightarrow \mathcal{E}^* \rightarrow b') \), each \( \mathcal{E}: \mathcal{E}^* \) can be expressed in terms of a condition which holds on the EXPORT interface before the component evolution (denoted by prefixing \( \mathcal{E} \) with a \( \uparrow \) symbol) and a condition which holds on the EXPORT
interface after the component evolution (denoted by a post-fixing ∈ with a ↓ symbol), where e↓ -→ ∈e. Each member of the set (Є - L) is defined through a specification of the conditions ∈e and e↓.

Defining Modifications in (Є - L)

Modifications in (Є - L) are specified in terms of the grammar of a particular language in which the components are developed. For example, in Java, Chapter 13 of the Java Specification [43] defines all members of the set Є and specifies which of these members also exist within the set L. A similar approach can be taken by this project i.e. elements of Є can be specified and divided into two partitions using the grammar rules of a particular CLR-compliant programming language, such as C#. However, this approach would be incomplete - it would fail to take full advantage of the underlying Common Language Infrastructure (CLI). A different path is taken – each element of the set (Є - L) will be specified in terms of the ILASM grammar. Using this approach, DejaVu.NET will provide a framework for a cross-language component evolution.

Each modification e:\(Є - L\) is specified in terms of a pre-evolution condition (∈e) defined on the EXPORT interface of a past component version, and a post-evolution condition (e↓) defined on the EXPORT interface of the newer component version. It should be re-iterated that the definition of (Є - L) assumes that all type references contained in an IMPORT interface arise solely out of type associations.

No proofs are given for

\[
\begin{align*}
(b \rightarrow e^* \rightarrow b') \wedge (\exists e: e^*[\uparrow e \wedge e\downarrow]) \rightarrow & \text{ absoluteBC}(b, b') \\
(b \rightarrow e^* \rightarrow b') \wedge (\exists e: e^*[a].[\uparrow e \wedge e\downarrow]) \rightarrow & \text{ relativeBC}(b, b', a)
\end{align*}
\]

They can, however, be validated with respect to the ILASM grammar definitions in ECMA 335, and with respect to the dynamic linking model presented in Section 2. The model provides a deeper understanding of the way in which reference metadata tokens are resolved by clients which, alongside an understanding of the relationship between ILASM constructs and metadata tokens [27], provides a basis for reasoning about effects of server modifications on clients. All of the constraints imposed by the requirements of the ClassLoader, Verifier, JIT Code Generator and Resolver should be taken into account. Library loading and assembly resolution relate to actions required to locate and load a particular component version, and hence, do not relate to current context.

53 As shown in Section 2, an IL disassembler converts the physical representation into a set of ILASM grammar constructs. Since ECMA 335 features a full specification of the mapping between ILASM expressions and the physical metadata tokens and IL method bodies, no information is lost in the disassembly process.
Rather than partitioning the set \((C - L)\) into a subset which defines forbidden modification on whole components, a subset which defines forbidden modification on types, and a subset which defines forbidden modifications on type members, it is preferred to partition \((C - L)\) into two sets \(\Xi\) and \(\mathcal{M}\) where:

Set \(\Xi\) contains forbidden modifications related to the existence of type and type member declarations. Given that \(b \rightarrow \mathcal{E} \rightarrow b'\), define\(^{54}\)

\[
\Xi = \{ \mathcal{E}\!: (C - L) | \quad (\exists t: \text{TypeDef.} \uparrow \mathcal{E} = (t \in \text{EXPORT}(b)) \wedge \downarrow \mathcal{E} = (t \notin \text{EXPORT}(b'))) \lor \\
(\exists f: \text{Field.} \uparrow \mathcal{E} = (f \in \text{EXPORT}(b)) \wedge \downarrow \mathcal{E} = (f \notin \text{EXPORT}(b'))) \lor \\
(\exists m: \text{Method.} \uparrow \mathcal{E} = (m \in \text{EXPORT}(b)) \wedge \downarrow \mathcal{E} = (m \notin \text{EXPORT}(b'))) \}
\]

Existence-checking assumes that types are distinguished through a combination of names and namespace prefixes, fields are distinguished through names alone, and methods are distinguished through a combination of names and types of parameters (allows overloading). Existence checking will, indirectly, take the constraints introduced by the CLR verifier into account. Intra-component inheritance chains will be traversed when searching for a particular method or field within a particular type.

Set \(\mathcal{M}\) contains forbidden modifications related to signatures of types and type members. \(\mathcal{M}\) thus covers changes in visibility, type conversions, conversions of static members to non-static members, etc…

Define

\[
\mathcal{M} = C - L - \Xi
\]

\(^{54}\)The set membership condition \(t: \text{TypeRef} \in \text{EXPORT}(b)\) is used in an informal sense which does not take the nesting scopes into account – \(t\) can be either a top-level or a nested type.
Since $\Xi$ cover nesting hierarchies as well as global fields and global methods, the best way to proceed is to conceptualize the EXPORT interface of a component as a set of nested type declarations, as illustrated in Figure 10. Examples of modifications $\vartriangleleft: \Xi$ are shown as grey arrows. By Def3.14 and Def3.15, as soon as one such $\vartriangleleft$ is found, it can be inferred that binary compatibility has been broken. The conceptual connection between the shapes of Figure 10 and a micro-organism is more than a mere coincidence – there is a direct link between the concepts of evolution of software components and the biological concept of evolution of micro-organisms. Further discussion is beyond the scope of this project.

Figure 10 provides a visual illustration of $\text{SUBSET}(\text{EXPORT}(B), \text{IMPORT}(A, B))$.

The process of evaluating $(b \rightarrow \vartriangleleft * \rightarrow b') \rightarrow \exists: \vartriangleleft * [\exists: \Xi]$ can be conceptualized as a depth-first search through the nesting hierarchy of the EXPORT interfaces for a modification $\vartriangleleft: \Xi$. To optimize the algorithm and avoid the requirement to repeat this search for an $\vartriangleleft: M$, it is preferred to search for $\vartriangleleft: (\Xi \cup M)$. At each depth of the search tree -where a depth corresponds to a single enclosing scope55 (the actual component (Assembly) or an enclosing type (TypeDef) ) – the search algorithm must ensure that all $\text{TypeDef}$, $\text{Method}$, and $\text{Field}$ declarations in $\text{EXPORT}(b)$ are also declared in $\text{EXPORT}(b')$ (existence check), and that each such declaration has not undergone a change $\vartriangleleft: M$.

55 Server-side enclosing scopes are conceptually equivalent to client-side resolution scopes.
Binary Compatibility Detection Algorithms

Algorithms that prove or disprove the assertion that binary compatibility has been maintained via a search for a counterexample - a modification $\varepsilon: (\Xi \cup M)$ – are outlined.

Absolute Binary Compatibility Detection Algorithm

Inputs: $\text{EXPORT}(b)$
$\text{EXPORT}(b')$

Result: $\text{EXPORT}(b) \supset \text{EXPORT}(b')$

Pseudocode:

global $\textbf{Result} \leftarrow \text{true}$

Procedure SearchScope($\textbf{Scope}$)
{
    For each $\text{TypeDef, Method, or Field} \space T$ in $\text{EXPORT}(b)$ within $\textbf{Scope}$
    {
        if $T$ in $\text{EXPORT}(b')$ within $\textbf{Scope}$
        {
            if a modification $\varepsilon: M$ is performed on $T$
            {
                $\textbf{Result} \leftarrow \text{false}$
            }
            else
            {
                if $T$ is $\text{TypeDef}$
                {
                    SearchScope($T$)
                }
            }
        }
        else
        {
            $\textbf{Result} \leftarrow \text{false}$
        }
    }
}

SearchScope($\text{Assembly}$)
Return $\textbf{Result}$
Relative Binary Compatibility Detection Algorithm\textsuperscript{56}

Inputs: \(\text{SUBSET(EXPORT(b), IMPORT(a,b))},\) 
\(\text{SUBSET(EXPORT(b'), IMPORT(a,b))}\)

Result: \(\text{SUBSET(EXPORT(b), IMPORT(a,b))} \Rightarrow\)
\(\text{SUBSET(EXPORT(b'), IMPORT(a,b))}\)

Pseudocode:

```plaintext
global Result <- true

Procedure SearchScope(Scope)
{
   For each TypeDef, Method, or Field T in
   \(\text{SUBSET(EXPORT(b), IMPORT(a, b))}\) within Scope
   {
      if T in \(\text{SUBSET(EXPORT(b'), IMPORT(a,b))}\) within Scope
      {
         if a modification \(\varepsilon: M\) is performed on T
         {
            Result <- false
         }
      } else
      {
         if T is TypeDef
         {
            SearchScope(T)
         }
      }
   } else
   {
      Result <- false
   }
}

SearchScope(Assembly)
Return Result
```

\textsuperscript{56} A compiler-enforced base case is necessary before maintenance of relative binary compatibility can be guaranteed
The final task of this report section is to formulate all $e: M$ by specifying the conditions $\uparrow e$ and $\downarrow e$. At this stage, $M$ is partitioned into four subsets:

1. Forbidden changes to **fields** - Table 10
2. Forbidden changes to **methods** - Table 11
3. Forbidden changes to **types** - Table 12
4. Forbidden changes to **components**\(^{57}\) - Table 13

Forbidden changes are specified as machine-verifiable **modification codes**.

### Fields

<table>
<thead>
<tr>
<th>Modification Code</th>
<th>$\uparrow e$</th>
<th>$\downarrow e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mf1</td>
<td>Field is <strong>accessible</strong></td>
<td>Field is <strong>not accessible</strong></td>
</tr>
<tr>
<td>Mf2</td>
<td>Field is <strong>static</strong></td>
<td>Field is <strong>non-static</strong></td>
</tr>
<tr>
<td>Mf3</td>
<td>Field is <strong>non-static</strong></td>
<td>Field is <strong>static</strong></td>
</tr>
<tr>
<td>Mf4</td>
<td>Field is <strong>static</strong></td>
<td>Field is <strong>static literal</strong></td>
</tr>
<tr>
<td>Mf5</td>
<td>Field is <strong>not initonly</strong></td>
<td>Field is <strong>initonly</strong></td>
</tr>
<tr>
<td>Mf6</td>
<td>Field type is a type $T$</td>
<td>Field type is <strong>not a type</strong> $T$</td>
</tr>
</tbody>
</table>

**Table 10 – Forbidden changes to fields**

### Methods

<table>
<thead>
<tr>
<th>Modification Code</th>
<th>$\uparrow e$</th>
<th>$\downarrow e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mm1</td>
<td>Method is <strong>accessible</strong></td>
<td>Field is <strong>not accessible</strong></td>
</tr>
<tr>
<td>Mm2</td>
<td>Method is <strong>static</strong></td>
<td>Method is <strong>non-static</strong></td>
</tr>
<tr>
<td>Mm3</td>
<td>Method is <strong>non-static</strong></td>
<td>Method is <strong>static</strong></td>
</tr>
<tr>
<td>Mm4</td>
<td>Method is <strong>non-abstract</strong></td>
<td>Method is <strong>abstract</strong></td>
</tr>
<tr>
<td>Mm5</td>
<td>Method is <strong>non-final</strong></td>
<td>Method is <strong>final</strong></td>
</tr>
<tr>
<td>Mm6</td>
<td>Method return type is a type $T$</td>
<td>Method return type is <strong>not a type</strong> $T$</td>
</tr>
<tr>
<td>Mm7</td>
<td>Method calling convention is <strong>varargs</strong></td>
<td>Method calling convention is <strong>default</strong></td>
</tr>
<tr>
<td>Mm8</td>
<td>Method calling convention is <strong>default</strong></td>
<td>Method calling convention is <strong>varargs</strong></td>
</tr>
<tr>
<td>Mm9</td>
<td>Method is <strong>not instance explicit</strong></td>
<td>Method is <strong>instance explicit</strong></td>
</tr>
<tr>
<td>Mm10</td>
<td>Method is <strong>instance explicit</strong></td>
<td>Method is <strong>not instance explicit</strong></td>
</tr>
</tbody>
</table>

**Table 11 – Forbidden changes to methods**

\(^{57}\) The last subset is added for completeness. It is simple to see how its addition impacts the binary compatibility detection algorithms.
Types  |  ↑℮  |  ↓℮  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mt1</td>
<td>Type is <strong>class</strong></td>
<td>Type is <strong>not class</strong></td>
</tr>
<tr>
<td>Mt2</td>
<td>Type is <strong>enum</strong></td>
<td>Type is <strong>not enum</strong></td>
</tr>
<tr>
<td>Mt3</td>
<td>Type is <strong>struct</strong></td>
<td>Type is <strong>not struct</strong></td>
</tr>
<tr>
<td>Mt4</td>
<td>Type is <strong>interface</strong></td>
<td>Type is <strong>not interface</strong></td>
</tr>
<tr>
<td>Mt5</td>
<td>Type is <strong>delegate</strong></td>
<td>Type is <strong>not delegate</strong></td>
</tr>
<tr>
<td>Mt6</td>
<td>Type is <strong>not abstract</strong></td>
<td>Type is <strong>abstract</strong></td>
</tr>
</tbody>
</table>

Table 12 – Forbidden changes to types

Note that, by Common Type System, each type can be placed into one of five categories:

\[
\text{Type} = \text{Class} \cup \text{Interface} \cup \text{Enum} \cup \text{Struct} \cup \text{Delegate}
\]

Semantics of the different type categories are sufficiently distinct to add significant weight to the claim that, should a client be known to successfully resolve a type reference, changing the category of the type reference results in a loss of binary compatibility relative to this client. Hence modification codes Mt1, Mt2, Mt3, Mt4, and Mt5.

Since inheritance and interface implementation relations between types residing in different components are disregarded, making a class **sealed** has no effect on binary compatibility.
### Table 13 – Forbidden changes to components

<table>
<thead>
<tr>
<th>Components Modification Code</th>
<th>$\uparrow e$</th>
<th>$e \downarrow$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Mc_1$</td>
<td>Assembly has <strong>no</strong> assembly attributes</td>
<td>Assembly has a <strong>nomachine</strong>, <strong>noprocess</strong>, or a <strong>noappdomain</strong> attribute</td>
</tr>
<tr>
<td>$Mc_2$</td>
<td>Assembly has a <strong>nomachine</strong> assembly attribute</td>
<td>Assembly has a <strong>noprocess</strong> assembly attribute</td>
</tr>
<tr>
<td>$Mc_3$</td>
<td>Assembly has a <strong>noprocess</strong> assembly attribute</td>
<td>Assembly has a <strong>noappdomain</strong> assembly attribute</td>
</tr>
<tr>
<td>$Mc_4$</td>
<td>Assembly contains a set of files [introduced by the <code>.file</code> ILASM directive] denoted by $F$</td>
<td>Assembly contains a set of files [introduced by the <code>.file</code> ILASM directive] denoted by $F'$ where $F \subsetneq F'$</td>
</tr>
<tr>
<td>$Mc_5$</td>
<td>Assembly contains a set of manifest resources [introduced by the <code>.mresource</code> ILASM directive] denoted by $MR$</td>
<td>Assembly contains a set of manifest resources [introduced by the <code>.mresource</code> ILASM directive] denoted by $MR'$ where $MR \subsetneq MR'$</td>
</tr>
</tbody>
</table>

There is no formal proof that all modifications $e: M$ are captured by the set of modification codes listed in Tables 10, 11, 12, and 13. Should this set prove inadequate to handle all scenarios encountered in practice, the modular *DejaVU.NET* design provides means to extend the existing modification codes.

---

58 While they were derived from the ILASM syntax, modification codes $Mc_1$, $Mc_2$, and $Mc_3$ are semantical in nature
Section Summary

- Maintenance of binary compatibility can be defined and checked for in a number of ways.
- This project defines binary compatibility in terms of resolution of reference metadata tokens held by a client component.
- This project evaluates whether the evolution of the server component maintains binary compatibility by analyzing the changes in the server component's interface of exported types and type members.
- Relative and absolute binary compatibility must be distinguished. Unlike absolute binary compatibility detection, relative binary compatibility detection requires a compiler-enforced base case.
- Re-conceptualization of the logical view of a .NET component permits low-level CLI implementation details (metadata tables) to be related to concepts encountered in theoretical work [41,39,28,19] (component interfaces).
- Specification of changes in ILASM grammar constructs known to break binary compatibility, enables the formulation of Relative and Absolute Binary Compatibility Detection Algorithms.
SECTION 4 - Binary Compatibility and the Evolution of a Component-based Software System

Introduction

Section 3 defined binary compatibility and outlined mechanical methods of evaluating the effects of a set of modifications made to a server component on the ability of this component to satisfy the requirements imposed by its clients. This section of the report analyzes the impact of changes made to individual components on a complete component-based software system, modeled as a dependency graph. A discussion of the software-system-level operations that must complement individual-component-level operations, in order to guarantee that the state of the software system always meets a set of properties, leads to the development of a Component Installation Algorithm.

Limitations of a client-server model

Binary compatibility detection was previously considered in the context of software maintenance[9]. A client-server model (such as the one shown in Figure 11) was sufficient.

![Figure 11 – A client-server model of binary compatibility detection](image)

In this model, a single client (an executable such as a Java or a C++ application) references functions, types, or methods (depending on the context) defined by a single server (a Java library, a DLL, etc...). This model’s main strength is the ability to reason about software maintenance through the abstraction of the different views of a server developer, a client developer, and a client user. Successive versions of server libraries are examined before informing the server developer whether the modifications made to the server will affect the clients.
Figure 12 illustrates the hierarchical nature of component composition [19]. Each component contains a single EXPORT interface and a number of IMPORT interfaces - it has the potential to take on both the server and the client roles. For example, two graphics packages might be the clients of a component embedding a painting API, which is in turn a client of a component embedding low-level graphics manipulation routines (e.g. a DirectX graphics component), which might be a server for a number of other clients (e.g. a flight simulation package).

The client-server model of Figure 12 is inherently unable to cope with the concept of hierarchical inter-component dependencies. Yet, this project calls for a method of modeling software systems with a large number of inter-dependent components of different versions. In order to produce a model of sufficient complexity to aid the design and implementation of DejaVu.NET, this report section examines the relationships between inter-component dependencies and binary compatibility, outlines a graph-based model for reasoning about evolution of component-based systems in terms of evolution of individual components, and derives a Component Installation Algorithm that encapsulates the functions needed to maintain a component-based system in a state where all clients link against the latest server component version that guarantees the successful resolution of the client’s reference tokens.
Modeling a Component-based system as a Set B

Conceptually, a component-based software system is modeled as a subset of Components, and is represented by an uppercase letter $B^{59}$. Physically, $B$ is represented as a set of physical modules stored in a central component cache. The names and version numbers are embedded both within the assembly manifests and within the directory structure of the component cache. Since they are stored in different file system directories, components of different versions can co-exist in the cache despite the fact that they share a common name. The convention of referring to components using lowercase characters (introduced in Section 3) is maintained – version numbers of components are omitted for brevity.

For a component $c:B$, define:

$$\text{version}(c) = \text{the four-part CLI-compliant version number of } c$$

An implicit assumption relating component evolution, component modifications, component version numbers and time is expressed by the following logical formulas:

$$(a \rightarrow e^{*} \rightarrow a^{'}) ^ (e^{*} \neq 0) \rightarrow \text{version}(a) < \text{version}(a{'})$$

$$(a \rightarrow e^{*} \rightarrow a^{'}) ^ (e^{*} = 0) \rightarrow \text{version}(a) = \text{version}(a{'})$$

Using the plain name of the component refers to a single particular version of the component. Each such plain reference distinguishes components via a combination of names and version numbers ($b^{'} \neq b$). Referring to a component via an underlined component name refers to any version of the component. Underlined component references thus distinguish components solely by their names ($b^{'} = b$), and refer to any member of a set of components that share the given name.

$$\forall c_1,c_2:B.VERR((c_1 = c_2) \rightarrow (\text{version}(c_1) = \text{version}(c_2))]$$

$$\forall c_1,c_2:B.VERR((c_1 = c_2) \rightarrow ((\text{version}(c_1) \neq \text{version}(c_2)) ^ (\exists e^{*}.[(e^{*} \subseteq C) ^ ((c_1 \rightarrow e^{*} \rightarrow c_2) v (c_1 \rightarrow e^{*} \rightarrow c_2)])])$$

59 To avoid any ambiguity, this report section uses upper-case letters to denote sets, and lower-case letters to denote components
60 Single-module assemblies imply a one-to-one-to-one mapping between physical modules, logical assemblies and conceptual components
61 Thus $a$ is the result of a non-deterministic membership function defined on a subset of $B$ containing all components named $a$, regardless of their version numbers
Dependency relation

The formulation of a new model requires a closer look at the notion of a component dependency. Szyperski’s notion of a component’s static dependencies as declarations of what other components it depends on [3] is taken as a starting point.

Dependency is defined in terms of an IMPORT interface of a component, and represented with the \( \sim \rightarrow \) operator:

\[
\forall a,b:B. \left[ \exists \text{IMPORT}(a,b) \rightarrow (a \sim \rightarrow b) \right]
\]

The dependency relation is:

- Non-Reflexive: no assembly \( a \) contains \( \text{IMPORTS}(a,a) \)
- Anti-symmetric: it is impossible to compile \( a \) and \( b \) such that \( a \sim \rightarrow b \) and \( b \sim \rightarrow a \)

By Def4.1, a dependency relation is generally not transitive – if a component \( a \) contains \( \text{IMPORT}(a,b) \) and a component \( b \) contains \( \text{IMPORT}(b,c) \), it is not generally the case (disregarding subclassing, subtyping, and implementation contracts\(^6^{63}\)) that \( a \) contains \( \text{IMPORT}(a,c) \).

Define:

\[
\text{clients}(a) = \{ b:B \mid b \sim \rightarrow a \}
\]

Def4.1 relates two particular versions of client and server components. A weaker statement, relating any version of \( a \) to any version of \( b \) is expressed as \( a \sim \rightarrow b \).

\[
(a \sim \rightarrow b) \rightarrow (a \sim \rightarrow b)

\neg((a \sim \rightarrow b) \rightarrow (a \sim \rightarrow b))
\]

---

\(^{62}\) In multi-module assemblies, a lower-level dependency relation exists between the physical modules that constitute an assembly. This dependency is, according to dynamic linking experiments, reflexive, anti-symmetric and transitive, with an a compiler-enforced requirement to produce a transitive closure.

\(^{63}\) See Appendix A
Another variant relates a particular version of a client to any version of the server - expressed as $a \leadsto b$

**Indirect dependencies**

An indirect dependency is defined to model situations such as that shown in **Figure 13**. Direct dependencies are represented by solid arrows while indirect dependencies are represented by dashed arrows.

![Figure 13 – Indirect dependencies $\leadsto^*$](image)

Indirect dependency is defined recursively:

\[
 a \leadsto^* c \iff (a \leadsto b) \land (b \leadsto c) \land \neg(a \leadsto c) \tag{Def4.3}
\]

\[
 a \leadsto^* c \iff (a \leadsto b) \land (b \leadsto^* c) \land \neg(a \leadsto c) \tag{Def4.3}
\]

**Appendix A** discusses the situations in which contracts between types force client components to have an ‘awareness’ of an indirect dependency.
Dynamic dependencies

An IMPORT interface consists of static (compile-time generated) references to types and type members defined in external components. Since $\sim \sim \sim$ is defined in terms of an IMPORT interface of a component, this definition constrains $\sim \sim \sim$ to static dependencies. The .NET Framework also supports dynamic references that give rise to dynamic dependencies. **Fusion** resolves loading paths of statically referenced assemblies using the properties (i.e. Name, Version, Public Key, etc…) embedded in the AssemblyRef metadata table. In contrast, dynamic referencing implies that properties of the server assembly are supplied as arguments to IL instructions when the loading path of such an assembly is resolved.\(^{64}\) A component is thus ‘aware’ of the server components that it references statically while it has no such ‘awareness’ of dynamically referenced servers.

In a practical view of the framework presented to programmers (which abstracts the lower-level ILASM and metadata views), the two types of dependency are distinguished through a distinction between explicit and implicit type loading \([6]\). Implicit loading results in a compile-time emission of **static references** into the assembly’s metadata, and is subject to compile-time assembly existence and type safety checks. With explicit loading, external components are referenced at run-time via the .NET Managed Reflection API. Evidently, the managed Reflection API classes encapsulate the dynamic linking processes thus making the client component ‘unaware’ of them. As such, the types from dynamically references assemblies are explicitly instantiated by consulting the external assembly’s EXPORT interface - compile-time checks are thus avoided.

Returning to a low-level view of the framework and the binary compatibility detection algorithms discussed in the Section 3, it should become clearer why dynamic dependencies present a problem in satisfying the goals of this project. Since compile-time checks guarantee successful resolution of all reference tokens and since dynamic references avoid compile-checks, **Def3.9** does not hold unless **successful_reference_resolution (a,b)** can be evaluated by means other than execution and other than a compile-time check.

Resorting to a signature-matching approach is of no help either because **Def3.9** cannot be evaluated unless the client-side signatures contained in IMPORT(a, b) contain the information regarding the full context of resolution of the metadata tokens that they embed. Consequentially, unless a low-level IL method body interrogation API is developed, no guarantees can be made on the successful dynamic linking, or successful reference token resolution.

\(^{64}\) Both types of references are still subject to Fusion’s re-direction policy specified in the Xml configuration files.
Def3.8 cannot be evaluated without an ability to evaluate Def3.9. Rather than developing a full IL method body inspection API, this project uses a definition of \( \sim \rightarrow \) which focuses entirely on static dependencies.

Inverse Dependency

Inverse dependency is the philosophical notion of a server component being ‘aware’ of the fact that its clients are ‘aware’ of it. It is represented by the \( \sim \sim \) operator where

\[
(a \sim \rightarrow b) \sim \sim (b \sim \sim a)
\]

Def4.4

Since the inverse dependencies of a component \( b \) are not captured within \( \text{EXPORT}(b) \), inverse dependencies have no physical representation.

Role of Inverse Dependencies in Relative Binary Compatibility Detection

Given \( b \rightarrow e* \rightarrow b' \), it is desired to check whether

\[
\forall a: \text{clients}(b).\text{relativeBC}(b,b',a)
\]

This requires the ability to, given a component \( a \), return \( \text{clients}(a) \) - a set of all of its clients, defined by Def4.2

But, an algorithm that computes \( \text{clients}(a) \) using Def4.2 needs to iterate over every version of every component in the component cache.

Now redefine \( \text{clients}(a) \) using the inverse dependency relation:

\[
\text{clients}(a) = \{ b:B \mid a \sim \sim b \}
\]

Def4.5

The process of finding a set of all clients of a component \( a \) is turned into a simple table look-up process. This saving in time complexity, at the expense of time complexity, is explained in more detail in Section 5.
Distinguishing Between Compile-time and Link-time Server Versions

Given a client component \(a\) and a server component \(b\), \(\text{IMPORT}(a,b)\) contains an \(\text{AssemblyRef}\) metadata table entry which specifies the statically-embedded version of \(b\). As such, the statement \(a \rightarrow b\) relates the particular version of \(a\) to the \textit{particular} version of \(b\) embedded in \(a\)’s metadata. By \textbf{Def3.9}, token resolution against this statically referenced server version is always successful:

\[
\forall a,b:B. \ (a \rightarrow b) \Rightarrow \ (\text{successful_reference_resolution}(a,b))
\]

However, always resolving tokens against such statically determined server versions constrains the evolution of the whole software system \(B\). No formal proof is given, but this is an intuitive notion. Whenever \(b \rightarrow b'\) and a client \(a\) of \(b\) is not updated through a re-compilation, such a client will stay ‘unaware’ of the new server versions - \(\text{IMPORT}(a,b)\) is never replaced with \(\text{IMPORT}(a,b')\). It might well be possible that \(\text{relativeBC}(b,b',a)\), but the client will, whenever invoked, continue to resolve reference tokens against \(b\) - not \(b'\).

It is thus necessary to distinguish between the server version that a client is compiled against (compile-time server version) and the server version that a client resolves reference tokens against (link-time server version). Given that \(a \rightarrow b\), define a function:

\textbf{Def4.6}

\(
\text{linking_server_version}(a,b)\) which returns \(b\) - the particular version\(^{65}\) of the server component \(b\) that is \textit{actually used} by the client \(a\), during execution, to resolve metadata reference tokens with a resolution scope \(b\)

To enable a component-based software system \(B\) to evolve, all components \(c:(A \cup I)\), must be provided with a facility to load a later version of the server than the one they have been compiled against. This requirement is expressed as:

\[
\neg \forall (a,b):B. \ (a \rightarrow b) \Rightarrow (b = \text{linking_server_version}(a,b))
\]

Discussion of physical means used by DejaVu.NET to override Fusion’s versioning policies and to provide such facilities is postponed until \textbf{Section 5}.

\(^{65}\) “\textit{particular version of a component} “refers to the whole component (IL method bodies and metadata tables [which embed version numbers]) - not just a four-part version number
Dependency Graphs

A dependency graph extends the model of a component-based system (previously defined by a set $B$) by explicitly declaring all dependencies between components.

A dependency graph is a directed graph $(B, D)$ where the edges in the graph (members of the set $D$) correspond to the inter-component dependencies. The versioning information contained within dependencies of the set $D$ refers to link-time server versions, and it overrides the versioning information in the statically emitted IMPORT interfaces. Dependency set $D$ is specified as follows:

- **constraint1**
  \[
  \forall a, b \in B. [(a \sim\rightarrow b) \rightarrow (a, b) \in D]
  \]

- **constraint2**
  \[
  \forall a, b, b' \in B. [ ((b \rightarrow c \rightarrow b') \land (a, b) \in D) \rightarrow (a, b') \notin D ]
  \]

- **constraint3**
  \[
  \forall (a, b) \in D. \{ \text{version}(b) = \text{version}(\text{linking_server_version}(a, b)) \}
  \]

- **constraint4**
  \[
  \forall (a, b) \in D. [ (a \sim\rightarrow b') \land (\text{version}(b') \leq \text{version}(b)) ]
  \]

Note that - beyond the claim that $D$ allows compile-time and link-time server versions to differ - this definition of $(B, D)$ puts no additional versioning constraints on the dependency set $D$. A $D$ which satisfies constraint1, constraint2, constraint3, and constraint4 can still lead to failures of the dynamic linking process – $D$ is underspecified. A safety condition must be introduced:

- **constraint5**
  \[
  \forall (a, b) \in D. [ (a \in B) \land (b \in B) \rightarrow \text{successful_token_resolution}(a, b) ]
  \]
Figure 14 shows a component-based .NET software system, and its mapping to a conceptual dependency graph model.

During the compilation of each component, a CLR-compliant compiler automatically introduces a static dependency on the mscorlib.dll library (an implementation of the set of basic built-in types). Figure 14 uses gray arrows to represent these compiler-generated dependencies.
To disregard mscorlib.dll in further discussion, define C and E:

\[ C = B - \{ \text{mscorlib.dll} \} \]

\[ E \subseteq D \]
\[ \forall (a, b) : D. [ -(b = \text{mscorlib.dll}) \rightarrow (a, b) \in E ] \]

Focusing entirely on the user-specified dependencies (modeled by set E), C can be partitioned into three subsets:

- Set A containing the application components, which take only client roles
  \[ A = \{ a \mid a \in C \land \neg \exists b : C. [b \rightarrow a] \land \text{executable}(a) \} \]
- Set L containing leaf library components which take on only server roles
  \[ L = \{ l \mid l \in C \land \neg \exists b : C. [l \rightarrow b] \} \]
- Set I containing library components which take on both client and server roles
  \[ I = \{ i \mid i \in C \land \exists b : C. [b \rightarrow i] \land \exists b : C. [i \rightarrow b] \} \]

The component-based system in Figure 15 can be expressed as follows:

\[ B = \{ \text{A.exe, B.dll, C.exe, D.exe, mscorlib.dll} \} \]
\[ C = \{ \text{A.exe, B.dll, C.exe, D.exe} \} \]
\[ A = \{ \text{A.exe, D.exe} \} \]
\[ L = \{ \text{C.exe} \} \]
\[ I = \{ \text{B.dll} \} \]

\[ D = \{ (\text{A.exe, B.dll}), (\text{B.dll,C.exe}), (\text{A.exe,mscorlib.dll}), (\text{B.dll,mscorlib.dll}), (\text{C.exe,mscorlib.dll}), (\text{D.exe,mscorlib.dll}) \} \]
\[ E = \{ (\text{A.exe, B.dll}), (\text{B.dll,C.exe}) \} \]

66 Executable(a) holds whenever the component the a contains a method marked with the .entrypoint ILASM directive (a static Main method in C#) and its filesystem-visible physical module representation ends with a *.exe extension
DejaVu.NET – A Framework for Evolution of Component-based Software Systems

Execution Safety

The specification of a dependency graph \((C, E)\) does not specify a constraint:

\[ \forall (a,b) : E. [(a \in C) \land (b \in C)] \]

The intentional under-specification is used as a basis for distinguishing between execution safe and execution unsafe dependency graphs. Dependency graphs that meet the execution safety property ensure that each dependency in the set \(E\) relates a client to an existing version of a server.

\[ \text{execution\_safe}((C, E)) \Leftrightarrow \forall (a,b) : E. [(a \in C) \Rightarrow (b \in C)] \]

Def4.7

Dependency Trees

It is necessary to formalize the concept of ‘dynamic composition’ of a set of components into a software application. Each such application is invoked by an execution of a component \(a\) where \(a \in A\). Since \(a\) references types from a number of server components which, in turn, take on client roles and reference types from further components, the act of reasoning about binary compatibility maintenance (within the complete software system modeled by the dependency graph) in the face of a large number of such applications (where all components are taken from a common component pool) is unachievable without an explicit formalization of this ‘dynamic application composition’ concept. The concept is formalized through the definition of dependency trees.

Given \((C, E)\) - a dependency graph representation of a component-based system, and a component \(a\) where executable\((a)\), a dependency tree that arises out of ‘dynamic composition’ of \(a\) and all of its direct and indirect dependencies into an application, is defined by the function \(Dtree(2)\) where \(Dtree((C, E), a)\) returns a graph \((Ca, Ea)\) subject to the following conditions:

\[ \forall c : Ca. [ (c \in (L \cup I)) \lor ((a = c) \vee (a \rightsquigarrow g) \lor (a \rightsquigarrow* g))] \]

\[ \text{cond4.1} \]

\[ \forall (c1, c2) : Ea. [ (c1 \in Ca) \land (c2 \in Ca) \land (c1 \rightsquigarrow c2) \land (c2 = \text{linking\_server\_version}(c1, c2))] \]

\[ \text{cond4.2} \]

\[ ^{67} \text{ Implemented in the .NET framework via the reference token resolution process between pairs of components discussed in Section2} \]
A critical assumption that underlies the derivation of the Component Installation Algorithm can now be expressed:

\[ \forall c_1, c_2 : \text{Ca.} [(c_1 = c_2) \rightarrow (\text{version}(c_1) = \text{version}(c_2))] \]

Given this assumption, the dependency tree model avoids situations where multiple versions of a component are loaded into the same process. Appendix B discusses the problems encountered when this assumption is removed.

Define a predicate \textit{well-versioned} on dependency trees as follows:

\[ \text{well-versioned}(\text{Dtree}((C, E), a)) \leftrightarrow \text{Def4.8} \]

\[ \forall (\text{client}, \text{server}): \text{Ea.} [ \neg \exists \text{server}' : (L \cup I), \{ \text{relativeBC}(\text{server}, \text{server}', \text{client}) \land \text{version}(\text{linking_server_version}(\text{client}, \text{server})) < \text{version}(\text{server}') \} ] \]

Now introduce the definition of \textit{well-versioned} for a complete software system modeled by the dependency graph:

\[ \text{well-versioned}((C, E)) \leftrightarrow \forall a : A. [\text{well_versioned}(\text{Dtree}(C, E), a)] \quad \text{Def4.9} \]

A software system \((C, E)\) for which \textit{well_versioned} holds is said to satisfy the \textit{well-versionedness} property.

Expressed in words, \textit{Def4.7} and \textit{Def4.9} summarize the final goal of this project:

\textit{For all possible applications that can be executed from a common set of components, all client components will be dynamically linked against the latest versions of server components which guarantee successful resolution of reference tokens held by the client components.}
Reconfiguration

The final concept (illustrated in Figure 16) introduced prior to a consideration of the evolution of a whole component-based software system is the concept of dependency reconfiguration.

![Dependency before reconfiguration](image1)

![Dependency after reconfiguration](image2)

Key:
- Dependency before reconfiguration
- Dependency after reconfiguration

Reconfiguration is encapsulated by the function \( \text{reconfigure} \), defined as follows:

Given the preconditions \((a,b) \subseteq E\) and \(b \rightarrow_{e^*} b'\),

\[
\text{reconfigure}(E,a,b,b') = E - \{(a,b)\} \cup \{(a,b')\}
\]

Def4.10

If any of the preconditions is not satisfied,

\[
\text{reconfigure}(E,a,b,b') = E
\]
Modeling the Evolution of .NET Component-based Software Systems

The relationships between software evolution in component-based systems and the graph theory are an active area of research. This project develops and uses a custom dependency graph model to facilitate reasoning about the evolution of a .NET component-based system. Graphs provide a succinct representation of various directions of evolution. Advanced research issues such as higher-order transformations of connections, architecture modeling formalisms, and concurrency issues in dynamic component views are abstracted away.

A dependency graph \((C, E)\) represents the state of a software system at a point in time. The operations that take \((C,E)\) between such states are referred to as its evolution steps. This project assumes that a system \((C,E)\) is evolved solely via evolutionary steps which operate on single components in isolation – there is no notion of concurrent evolution, and the entire software system follows a linear evolutionary path.

Let \(e[c]\) represent an individual evolutionary step. The assumption that evolutionary steps operate on single components in isolation is captured by supplying each evolutionary step with a single parameter \(c: \text{Component}\).

Define

\[
\text{Def4.11} \quad ((C,E) \rightarrow e[c] \rightarrow (C',E')) \iff (\text{an evolutionary step } e[c] \text{ evolves a component-based software system } (C,E) \text{ into a component-based software system } (C',E'))
\]

An evolutionary step \(e[c]\) is valid if it takes the software system from one well-versioned and execution safe state into another well-versioned and execution safe state. The notion of validity is formalized though the \(\Rightarrow e[c] \Rightarrow\) relation (a stronger version of the \(\rightarrow e[c] \rightarrow\) relation):

Given

\[
\text{Def4.12} \quad c: \text{Components},
\text{execution_safe}(C,E),
\text{well_versioned}(C,E),
\]

\[
((C,E) \Rightarrow e[c] \Rightarrow (C',E')) \rightarrow (\text{execution_safe}(C',E') \land \text{well_versioned}(C',E'))
\]
For a single component $c:C$, $(C,E) \rightarrow e[c] \to (C',E')$ and $(C,E) \Rightarrow e[c] \Rightarrow (C',E')$ are higher-level statements than either $c \rightarrow e \to c'$ or $c \rightarrow e^* \to c'$. 

**Defining Evolutionary Steps $e[c]$**

It is assumed that:
- Components are not evolved concurrently
- Components are evolved by component maintainers via a checkin, checkout procedure

∴ All possible evolutionary steps are either component removal operations (set $\mathcal{R}$) or component addition (set $\mathcal{A}$) operations.

\[(e[c] \in \mathcal{A}) \leftrightarrow (e[c] = \{\text{Addition of component c}\})\]  

\[((C,E) \to e[c] \to (C',E')) \land (e[c] \in \mathcal{A}) \leftrightarrow\]
\[
(\forall a:C'.[(a \in C) \lor (a = c)] \land \\
\forall (a,b):E'.[((a,b) \in E) \lor ((a = c) \land (c \rightarrow b))])
\]

*Figure 14 – Evolutionary step $e[c]$: $\mathcal{A}$*
DejaVu.NET – A Framework for Evolution of Component-based Software Systems

\[(e[c] \in \bar{\rho}) \leftrightarrow (e[c] = \{\text{Removal of component } c\}) \quad \text{Def}4.15\]

\[
((C,E) \rightarrow e[c] \rightarrow (C',E') \land (e[c] \in \bar{\rho}) \land (c \in C)) \leftrightarrow
(\forall a:C. [(a \in C') \lor (a = c)] \land \\
\forall (a,b):E. [(a,b) \in E'] \lor ((a = c) \land (c \longrightarrow b))] \quad \text{Def}4.16
\]

![Figure 15 - Evolutionary step e[c]:\bar{\rho}](image)

Def4.13 and Def4.15 refer to the addition and the removal of particular versions of particular components. So, informally, if the set \(C\) contains a component \(c\) where \(\text{version}(c)=1.0.0.0\) as well as a component \(c\) where \(\text{version}(c)=2.0.0.0\), removing one of these two components will not result in a removal of both. Similarly, adding a new \(c\), where \(\text{version}(c)=3.0.0.0\), to such a set \(C\) will result in three elements of \(C\) which share the same names, but have different versions.

---

68 It should be kept in mind that \(C\) distinguished its members through a combination of names and version numbers. Hence, it is possible for \(C\) to contain two elements with identical names.
The semantical link between $\mathfrak{e}[a]$ and $\mathfrak{e}$ is embodied in the following logical equivalence:

\[
\begin{align*}
a & \in C \\
\mathfrak{a} & \xrightarrow{e^\ast} a' \\
\mathfrak{e}[a'] & = \{\text{Addition of component } a'\}
\end{align*}
\]

\[\vdash \mathfrak{e}[a'] = \{\text{Evolution of an existing component } a \text{ into } a'\}\]

Define a set $\mathcal{E}$ where

\[\mathcal{E} \subseteq \mathcal{A} \]

\[(\mathfrak{e}[c] \in \mathcal{E}) \iff (\mathfrak{e}[c] = \{\text{Evolution of an existing component } c \text{ into } c'\})\]

![Figure 16 – Evolutionary step $\mathfrak{e}[c]\mathcal{E}$](image)

Within the context of a component-based system $(C,E)$, the evolution of a component $a$ into $a'$ is thus equivalent to an addition of an evolved version $a'$ to $(C,E)$ which already contains the past version $a$. This approach, where multiple versions of $a$ co-exist in $(C,E)$, is taken because it cannot be assumed that all of $a$'s clients will successfully resolve their reference tokens against $a'$:

\[\neg \forall c: \text{clients}(a).[\text{relativeBC}(a,a',c)]\]
The component addition and the component removal are the two basic evolutionary steps. Formal proofs are outside the scope of this project, but it is an intuitive notion that cross-component type migration, component splitting, and component merging can be broken down and expressed as a sequence of these basic operations, alongside sequences of modifications performed on individual components. Consequently, providing formal definitions of $\vec{e}$, $\vec{e}^*$, $\vec{e}[a]$, $\rightarrow\vec{e}^*$, $\rightarrow\vec{e}[a]$ and $\Rightarrow\vec{e}[a]\Rightarrow$ can provide descriptions for linear evolutionary steps of arbitrary complexity. Informal definitions given in this report are sufficient to aid in the implementation of DejaVu.NET.

Defining Valid Evolutionary Steps

Given a $(C,E)$ where well_versioned((C,E)) and execution_safe((C,E)),

$$\diamond(((C,E) \rightarrow \vec{e}[a] \rightarrow (C',E')) \text{^} \neg (\text{well_versioned}(C',E'))$$

$$\diamond(((C,E) \rightarrow \vec{e}[a] \rightarrow (C',E')) \text{^} \neg (\text{execution_safe}(C',E'))$$

Thus $\rightarrow\vec{e}[a]\rightarrow$ alone cannot guarantee the maintenance of the well-versionedness or the execution safety conditions.

There are two scenarios in which well-versionedness is compromised:

Scenario 1:

$\vec{e}[a] \in \bar{\bar{A}}$

$\exists b': (C \cap C').[ (a \sim \Rightarrow b) \land (b \rightarrow \vec{e}^* \rightarrow b') \land \text{relativeBC}(b,b',a) ]$

Scenario 2:

$(a \in C)$

$(a \rightarrow \vec{e}^* \rightarrow a')$

$(\vec{e}[a'] \in \bar{\bar{A}})$

$\exists b:(C \cap C').[ (b \sim \Rightarrow a) \land \text{relativeBC}(a,a',b) ]$

69 This operation refers to the action of merging two different components – i.e. components with different names. The distinct operation of merging components with same names and different version numbers is outlines in Appendix D as a solution to problems encountered during side-by-side execution.
There are also two scenarios in which execution safety is compromised:

**Scenario 3:**
\[ e[a] \in \bar{A} \]
\[ \exists b:\text{Components}.\left[ (a \rightarrow b) \land (b \notin (C \cap C')) \right] \]

**Scenario 4:**
\[ e[a] \in \bar{R} \]
\[ \exists b:(C \cap C').\left[ b \rightarrow a \right] \]