**Symbols design for NoPia support**

With respect to referenced assemblies, there are two main things compilers have to do to support NoPia feature (<http://devdiv/docs/Dev10/VS/PUs/VSLanguages/Compiler/Specs/No-PIA/NoPIA%20Compiler%20spec.docx>):

1. Unify local types (decorated with TypeIdentifier attribute) from referenced assemblies (section 4.5 of the spec).
2. Prevent instantiations of generic classes or structures closed over local types from crossing assembly boundaries (section 4.4 of the spec).

Doing this work at metadata import phase greatly simplifies subsequent compilation phases: semantic analysis, etc.

**Proposal:**

1. A referenced assembly that contains local types should be imported in a special way, which makes sure that:
2. A local type declaration is hidden, i.e. one will not run into the declaration while traversing Symbols tree by means of GetMembers method (the rational is that, unless we hide local type declaration, merged namespace for a compilation will have a bunch of symbols for local types next to the symbol for their canonical type, which will be awkward).
3. Instantiations of generic classes and structures that refer to a local type in their arguments should surface as ErrorTypeSymbols.
4. References to a local type from other metadata (method signatures, etc) should surface as NamedTypeSymbol for canonical type the local type must be unified against, unless the unification fails (PIA with canonical type is not referenced, unification is ambiguous, etc.). If unification fails, an ErrorTypeSymbol symbol should be returned.
5. A referenced assembly that itself refers to an assembly being /l-ed by the compilation should be imported in a special way, which makes sure that instantiations of generic classes and structures referring to a “to be embedded” type in their arguments should surface as ErrorTypeSymbols.
6. A referenced assembly that is /l-ed by the compilation should be imported in a special way, which makes sure that instantiations of generic classes and structures referring to a “to be embedded” type in their arguments should surface as ErrorTypeSymbols.

**How do we detect that referenced assembly contains local types?**

For assembly file references we should be able to quickly check whether it defines any type with TypeIdentifier attribute applied to it, this can be done without creating any symbols for the assembly.

For referenced compilations, we can make an assumption that its resulting assembly will contain local types if it has a /l reference. However, if compilation doesn’t have /l reference, its resulting assembly may still contain local types explicitly declared in source. We will need a quick way to check for presence of types like that. If resulting assembly is going to contain local types, its SourceAssemblySymbol cannot be used directly.

**When can we reuse symbols for the specially imported assemblies, assuming that the requirement of reference consistency is met?**

Symbols for an assembly of kind 1) (see the proposal), can be reused if:  
- resolution of canonical types is the same;  
- symbols for assemblies defining canonical types are reused as well.

Note that, in general, canonical type for a local type must be looked up in context of the current compilation, i.e. across all assemblies referenced by compilation and, at least for Dev10 VB compiler, inside assembly being compiled. Implementation of the most accurate check for the first requirement (resolution of canonical types is the same) will be rather complex and will require non-trivial amount of upfront work in AssemblyManager. Supporting canonical types inside assembly being compiled (VB Dev10) complicates things even more. Perhaps we should deprecate this support ASAP, before customers have taken dependency on it (opened bug #915767 for Dev11).

Assuming VB behavior is deprecated, we can start with an implementation that reuses symbols only if new compilation is given exactly the same set of references as the one the symbols were initially created for. In this case we will reuse symbols for all references. Scenarios of creating new compilation by applying source changes to an existing compilation will reuse symbols, unless one of the referencing assemblies refers back to the compilation.

Symbols for an assembly of kind 2) can be reused only if all its references that were /l -ed by the compilation, which originated the symbols, are /l-ed by the new compilation and other references are either /r-ed or not referenced.

Symbols for an assembly of kind 3) can be reused only if the assembly was /l-ed by the compilation, which originated the symbols.

For example:

C.DLL : /r:A.DLL, B.DLL

Compilation1 : /r:C.DLL, A.DLL /l:B.DLL

Compilation2 : /r:C.DLL, A.DLL,D.DLL /l:B.DLL

Compilation3 : /r:C.DLL /l:B.DLL

Compilation4 : /r:C.DLL, A.DLL,B.DLL

Compilation5 : /r:C.DLL /l:A.DLL, B.DLL

Compilation6 : /r:A.DLL /l:B.DLL, C.DLL

Compilation7 : /r:C.DLL, A.DLL

Let’s assume that A.DLL, B.DLL and D.DLL do not reference each other and none of the dlls contain local types.

In context of Compilation1, C.DLL is a referenced assembly of kind 2) because it references B.DLL, which is /l-ed by Compilation1. Compiler creates AssemblySymbols A1, B1 and C1 for referenced assemblies.

In context of Compilation2, the same symbols can be reused for A, B and C, AssemblySymbol D2 is created for D.DLL.

Compilation3 can use AssemblySymbols C1 and B1.

Compilation4 can use AssemblySymbols A1, but not C1 or B1 because both C1 and B1 likely contain some error symbols, which shouldn’t be error symbols in context of Compilation4. AssemblySymbols C4 and B4 are created for C.DLL and B.DLL respectively.

Compilation5 can use AssemblySymbols B1, but not A1, C1 or C4 because they don’t expose all “illegal” types as error symbols. C1 and A1 don’t mark generics closed over types from A.DLL as bad. AssemblySymbols A5 and C5 are created for A.DLL and C.DLL.

Compilation6 can use A1, B1, but not C1, C4 or C5 (this is the first time C.DLL is linked). AssemblySymbol C6 is created for C.DLL.

Compilation7 can use A1 and C4. It cannot use C5 because A.DLL is /r-ed. It cannot use C1 because it will likely cause misleading error about “illegal” generic instantiations closed over types from B.DLL.