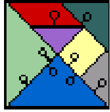


# A Hierarchical Technique for Composing COM based Components

*Author: Chris Peltz*

*Microsoft Corporation*

*Microsoft Research – Advanced Applications (ComApps)*

*chrispe@microsoft.com*

*Last Updated: March 16, 1999*

Abstract: *This is a position paper in response to a call for participation from the Second International Workshop on Component-Based Software Engineering. This paper discusses a technology currently under development to combine arbitrarily complex component structures. We call these structures Assemblies.*

*It will be useful to have knowledge of Microsoft COM technology. This can be obtained from documents available at* http://www.microsoft.com/com/dcom.asp.

**The Problems of Component Composition** *

**Components** *

Assemblies *

Elements *

Connectors *

Binary Connectors *

Ordering of Connections within an Assembly *

Assembly Lifetime Management *

**COMCAD Diagrams for Specifying Assemblies** *

Visual Elements and the COMCAD Diagram Tool *

## The Problems of Component Composition

The concept of software *components* comprises a spectrum of different technologies. Some people think of components as the pieces of large multi-tier distributed systems, such as business objects. Each of the components in these systems is rarely composed of many other smaller components. On the other end of the spectrum, components are used to build all kinds of software not just distributed systems.

At the very extreme end of that spectrum is the idea that components are small (from 100-4000 lines of C) and are composed in a hierarchical fashion to achieve larger and more functional sets of components. This is where the ComApps research team is investigating.

While there are several facets to our research, this position paper only discusses our efforts in solving some of the problems inherent in the hierarchical composition of larger components from smaller components.

In our research, we call a composition of a set of components an *assembly.* Components not composed of other components are called *elements*. In this document, the term *component* refers to both assemblies and elements. A pointer reference from one component to another component is called a *connection.*

In researching the composition of COM components, several problems have been recognized and need to be solved.

1. **Tight Coupling** -- We want to avoid having the connections force a tight coupling between components. Component class A shouldn't be tied to only allowing a connection to objects of class B. This limits reuse. COM's QueryInterface mechanism allows loose coupling between any two elements. The problem becomes more difficult between an assembly and an element, or between two assemblies. In these cases, the assembly object is composed of many different components. QueryInterface only works for a single object identity, not multiple identities.
2. **Versioning Connection Relationships** -- We want to avoid having the assembly mechanism enforce specific semantics on connections. The composition mechanism must allow the components to negotiate the proper relationship between themselves. For instance, consider a component A that can deal with different backward compatible versions of a component B, say B-old and B-new. The components B-old and B-new both implement an interface IBar. However, B-new has extended functionality that is exposed via IBaz. The composition mechanism must allow A to have a fully functional connection to B-old as well as B-new.
3. **Component lifetime management**. – We need to be able to determine when an assembly of components can finally be shut down. Within an assembly and across assemblies there are connections being made between components. Some of these connections may actually cause circular references (that is, object A is holding onto B and B is holding onto A and the objects can't be destroyed until all references are zero). The composition mechanism must provide a solution for all circular references.
4. **Specifying Assemblies** – We need a way to allow the programmer to specify assemblies of components. These assemblies may contain complex connections between components within the assembly as well as connections to components outside of the assembly.

The following sections of this document comprise an overview of the component assembly mechanism and how it addresses these problems.

# Components

As mentioned in the previous section, there are two types of components: Assemblies and Elements. Connections are managed via a mechanism called *connectors*. The following sections describe each of these concepts.

## Assemblies

How does a programmer construct a large reusable component from smaller components? In COM,

aggregation is one such mechanism, but it suffers from some limitations:

- Aggregation can only construct single objects; there is no direct support for part-whole hierarchies. The programmer is responsible for instantiation and management of subordinate objects, with no help.
- The aggregating component must manage the lifetimes of its parts.
- Internal references between parts within the aggregate have special QueryInterface and reference counting rules that need to be followed. This adds complexity and confusion.

In aggregation, the outer component may include code for the semantic behavior of the composite, as well as code for instantiation of parts and lifecycle management. In particular, the outer component knows the classes of parts it will instantiate. This means that if you want to reuse the composite, but modify it to use a different part, then the source code for the outer component must be modified. This limits reuse.

An alternative approach is to implement a part-whole hierarchy, where an object is responsible for managing the objects immediately below it in the hierarchy. As in aggregation, this model puts the code for instantiating and managing the structure into the same components as the code that provides the semantic behavior of (that level of) the composite. Configurations other than pure hierarchies have added complexity in management (e.g. cyclic references or confusion over ownership). Management of the structure becomes one of the hardest parts of programming.

An *assembly* is a connected group of components (including sub-assemblies) that enables connection to its elements by other assemblies, through *connectors (See* Connectors*)*. Assemblies are then candidates for reuse, as well as the constituent components. Connectors provide encapsulation and polymorphism for connecting to components. In addition, assemblies can be parameterized, to give the effect of a framework where you specify one or more partial behaviors. Assemblies thus support hierarchical construction of arbitrarily large composites.

The assembly mechanism separates code for instantiation and connection of components (the *structure-management*) from the code for the semantic behavior of the composite ( *the run-time behavior* of the components). Separating the structure-management from run-time behavior has the advantage of making sure that ordering of connections is independent from the component behavior. The component can easily be re-used within many different assemblies by merely connecting different instances without the worry of ordering dependencies of connections within the bounds of each assembly. Ordering dependencies between connections can limit reuse.

We further factor the lifecycle management problem into two parts:

- a lifetime boundary wrapper that detects when there are no references to any element in the assembly (See Assembly Lifetime Management)
- a generic assembly run-time wiring engine which knows how to cleanly disconnect and shut down all the components. This solves the cyclic reference problem, and provides one clear owner for the components independent of the kind and number of composite behaviors that use the components. (See COMCAD Diagrams for Specifying Assemblies)

## Elements

Elements are simply COM objects. They may or may not expose connectors. They typically do not create other components, although some may instantiate helper components such as collections.

## Connectors

Connections within an assembly are instigated by the assembly. The assembly mechanism will tell one of the components involved in the connection to perform the connection. For instance, the connection between components A and B is actually made by A and not the assembly. Component A can then use QueryInterface to enhance the connection. For instance, consider a connection that is characterized by component A wanting to refer to component B's IBar interface. Component A can extend the meaning of

that connection to optionally include IBaz by simply doing a QueryInterface from IBar to see if IBaz is also supported.

_Connectors_ (sometimes referred to as _Unary Connectors_) are interfaces that facilitate the connection between two components. A connector is implemented by at least one and possibly both components that participate in the connection. Any given connector on a component can specify that the component implements one or more specific interfaces (called connector _out pins_). Additionally, a connector can specify one or more interfaces the component wants to use (called connector _in pins_). The in pins and out pins characterize a given connector implementation. The actual connection that takes place however is performed by the component that implements the connector in pin.

Different out pins on a single connector are not expected to belong to the same object identity, so it is not expected that a call QueryInterface from the interface on one pin, will result in the interface on another pin. This contract is necessary since assemblies are composed of multiple sub-components, connectors are used on assemblies to expose interfaces from the components within.

Its also interesting to note that since connectors are interfaces, entire connectors can be used as the pin of a different connector. In this way, complex connections can be made across assembly borders.

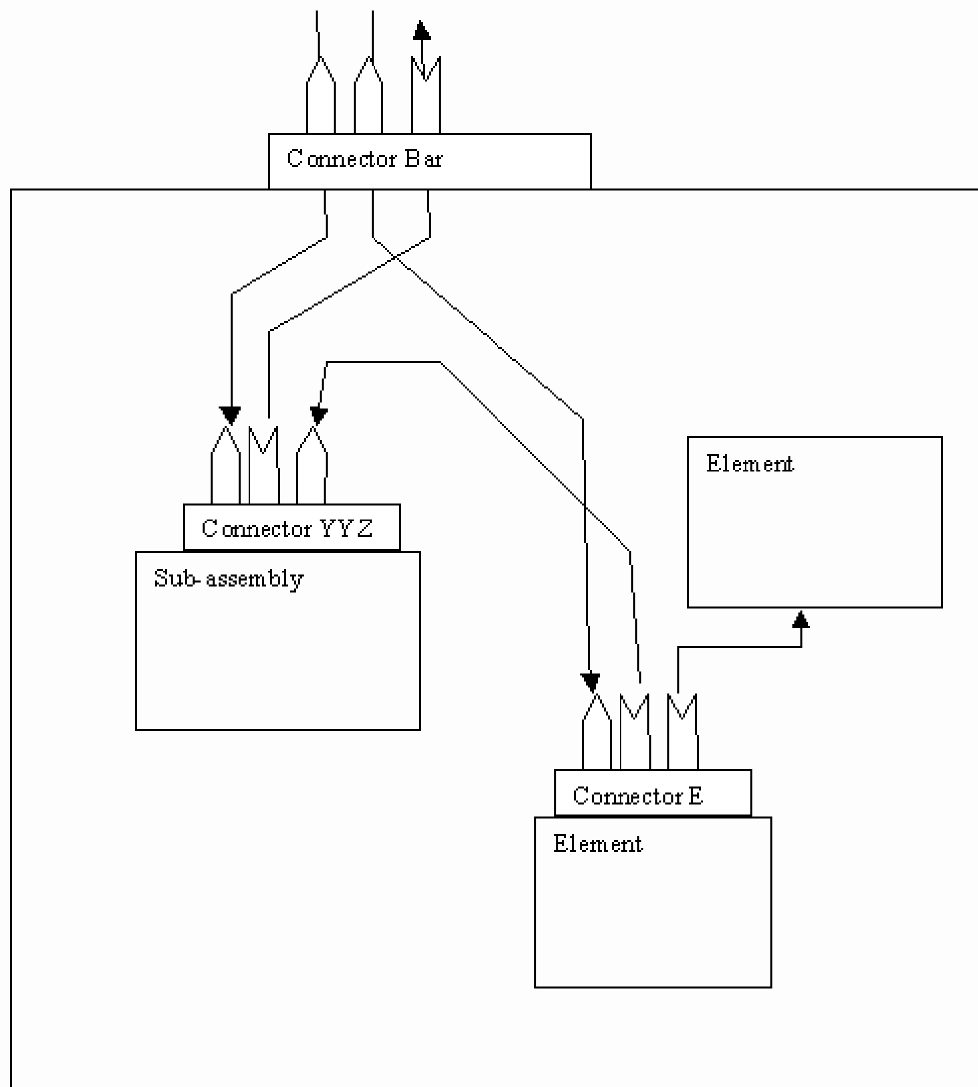


**Figure 1 , An assembly. The out pins on the connectors represent interfaces implemented by the component. The in pins on the connectors represent the interfaces used by the component, the arrows represent actual connections that point toward the implementer of a particular interface.**

Any component may expose multiple connectors. In this way, COM components have static, polymorphic type definitions to facilitate the wiring of components at multiple levels of assembly. Each connector can be considered a specific type contract that extends the individual interface type contracts of its pins.

Essentially, connectors are used to expose parts of an component's sub-structure. It is an encapsulation mechanism.

Two COM interfaces are used to implement connectors on components:

- **IConnectorProvider** – this interface is implemented on the component that supplies a connector interface, typically an assembly. It has one method "GetConnector" on it to retrieve a GUID named IConnector interface from the component. Note that the IConnector interface is not expected to be on the same object instance as IGetConnector.

```
interface IConnectorProvider : IUnknown

{

HRESULT GetConnector (REFCONNECTORNAME conname, REFIID riid,
[out, iid_is(riid)] void **ppvConn);

}
```

- **IConnector – this interface is implemented by a component or by a component within an assembly. Actually this is a templated interface where specific connector types are trivial derivations of IConnector, The derivations are used in order to specify a correct type-contract regarding which pins are present on the connector.**

```
interface IConnector : IUnknown

{

HRESULT GetElement (LONG dwOutPinId, REFIID riid, [out,
iid_is(riid)] void **ppvElement);

HRESULT Connect (LONG dwInPinId, IUnknown *punkOther);

.

.

.

}
```

The interesting methods on this interface are:

- "Connect" indicates a new reference should be made for the given in pin. A generic assembly-wiring engine (See COMCAD Diagrams for Specifying Assemblies) will call this to both connect and disconnect wires between components.
- "GetElement" is used for retrieving an interface reference to the component on the other side of the connection, i.e., the element with the corresponding out pin on the other side of the wire. If the

other side of the wire is not an out pin, then QueryInterface is used to retrieve the necessary interface, rather than GetElement.

### *Binary Connectors*

Binary connectors are a specific flavor of connector that differs from the Unary Connector. They are still driven by an assembly mechanism to perform connections.

Using Unary Connectors, two objects are wired together within an assembly via a connector interface implemented on the object that has the connector in pin.

Binary Connectors enable a third party connection mechanism where neither object contains the actual code to perform the connection. Binary connectors are used:

- when two objects need to be connected in a complex way
- when a component is introduced to an assembly that doesn't use the connector interfaces to facilitate connections… (such as legacy COM components)

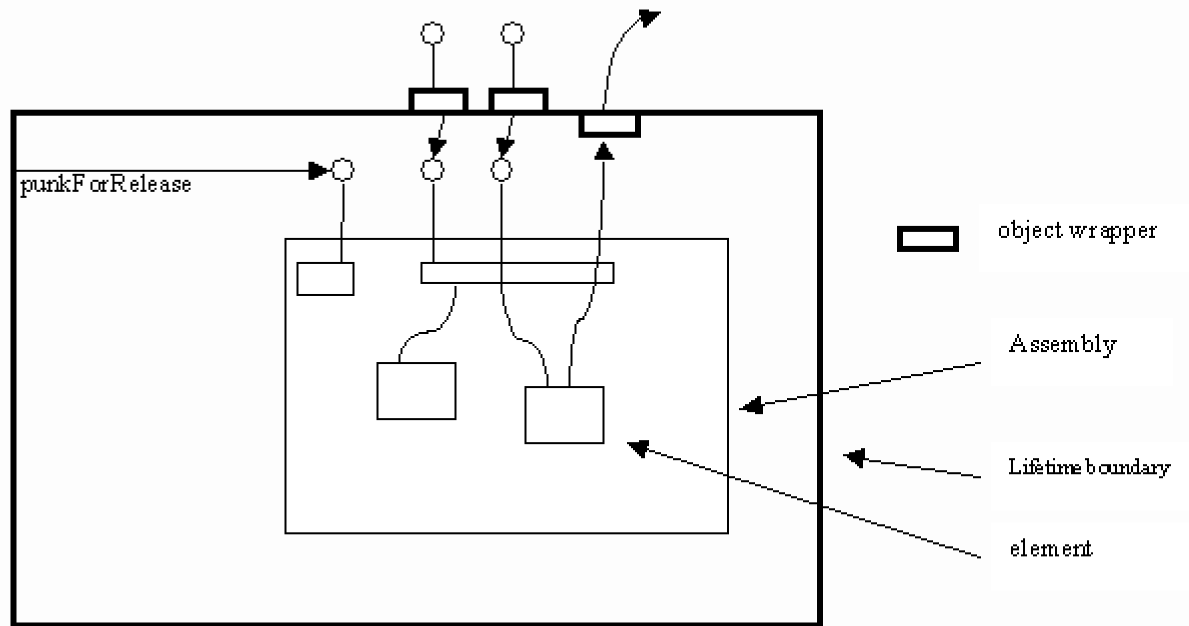
## Ordering of Connections within an Assembly

Within any given assembly, it is required that the order in which connections are made to elements is immaterial. Components should never be designed or built with any assumptions about connection order.

## Assembly Lifetime Management

The lifetime of an assembly is dependent on the assembly object from which it was created. The lifetime wrapper boundary is used to enable all elements of the assembly to contribute their reference counts to the assembly's lifetime. All references going across the boundary are tracked in order to determine when the assembly should be shut down. The assembly can do this since the assembly knows what connections are both internally and externally.

The assembly manages all component creations as well as connections between sub-components via a lifetime boundary:




## COMCAD Diagrams for Specifying Assemblies

Having factored out instantiation and connection (structure-management) behavior from run-time behavior, we can now consider a generic data driven component that can instantiate and manage assemblies. We call this component *the assembly-wiring engine*. We liken assembly diagrams to hardware wiring diagrams, where the wires represent connections between components.

The wiring engine knows how to interpret data that describes an assembly to actually perform the necessary instantiation and connection of sub-components. We call the data a *wiring template*.

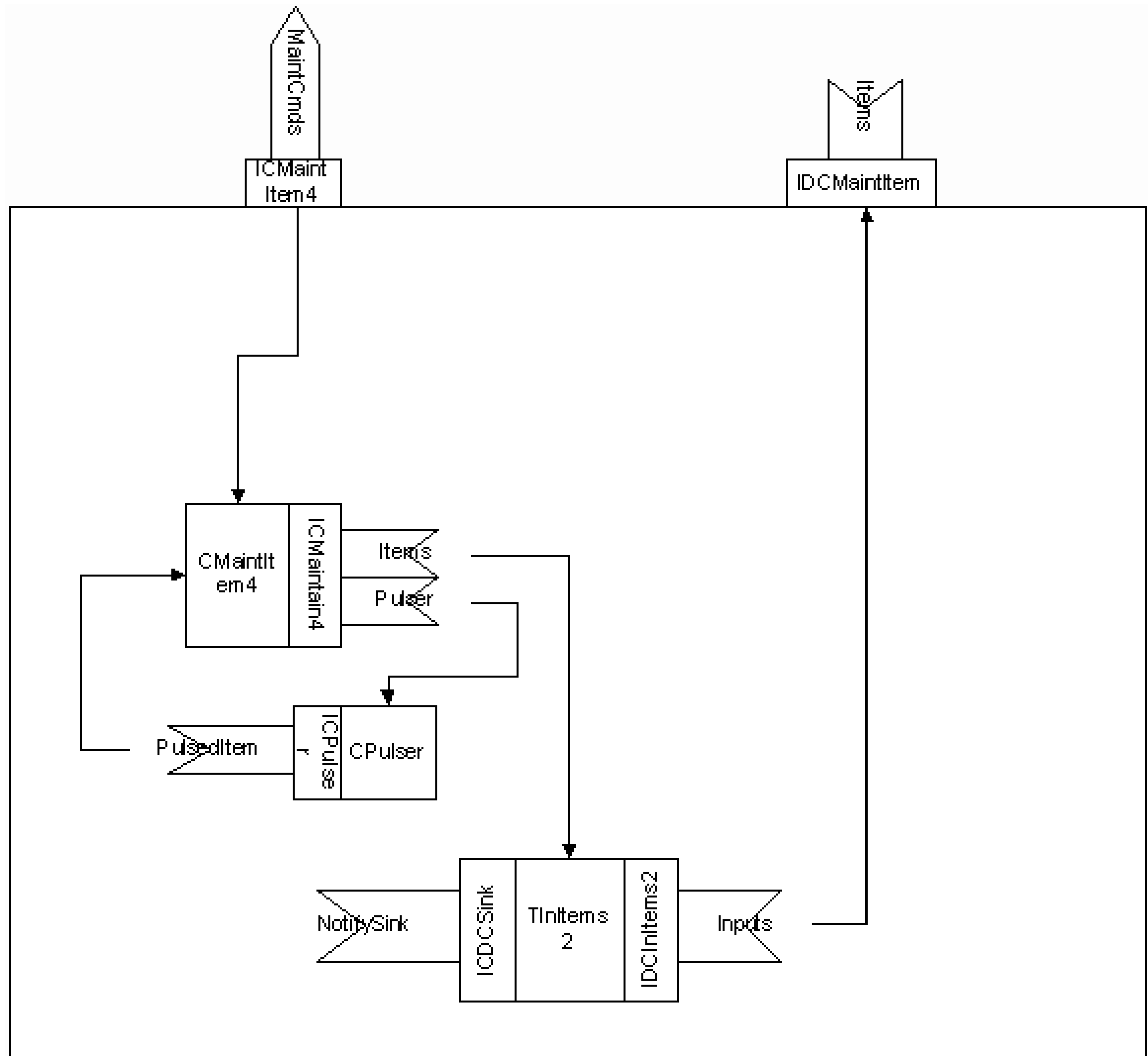
Our team has built a prototype graphical tool for interactively drawing the assemblies. This tool, called COMCAD, then generates a wiring template from an assembly diagram. The wiring template is then executed by the assembly-wiring engine to perform the actual instantiation and connection of the COM components.

Essentially this has become a form of graphical programming or executable diagrams. We believe that that many interesting composites can be constructed out of relatively few parameterized components and sub-assemblies. Wiring templates are an interesting representation for use across networks because they are small, cheap to download, processor neutral, and can be determined to be safe (being data rather than code).

Note that data driven wiring is not an exclusive requirement for implementing assemblies. The architecture defines how any other software can do wiring, and participate in the system in exactly the same way.

### Visual Elements and the COMCAD Diagram Tool

The following diagram is an image of an actual diagram created with our prototype Assembly drawing tool:




This is an image of a single assembly. It contains three sub-components, in this case all of the sub-components are elements that support connectors (although these could have been sub-assemblies as well). There are two export connectors each with a single pin. The arrows (a.k.a., wires) indicate the connections that will be made during instantiation of the assembly.

When the wires point to a box rather than a connector, it means that a IUnknown::QueryInterface is being used rather than IConnector::GetElement to retrieve the out pin side of the wire. The notation is simple and straightforward.

Once an assembly has been saved and built, it can then be used in another assembly diagram as a sub-assembly. In this way, a hierarchy can be built up with progressively more functionality at each level of the hierarchy.

Back to top

Copyright © 1999 Microsoft Corporation. All rights reserved