Interface Negotiation and Efficient Reuse: A Relaxed Theory of the Component Object Model

Kevin J. Sullivan *Mark Marchukov*

Technical Report 97-11
Dept. of Computer Science, University of Virginia
Charlottesville, Virginia 22903 USA
tel. (804) 982-2206
FAX: (804) 982-2214
e-mail: {march, sullivan}@virginia.edu

May, 9 1997

Abstract

Reconciling requirements for (1) the efficient integration of independently developed and evolving components and (2) the evolution of systems built from such components requires novel architectural styles, standards and idioms. Traditional object-oriented approaches have proven inadequate. Two important new mechanisms supporting integration and evolution are dynamic interface negotiation and *aggregation*, an approach to efficient composition. Both feature prominently in the Component Object Model (COM), a *de facto* standard providing the architectural foundation for many important systems. Because these are important mechanisms in general, and because they are central to COM in particular, it is essential that engineers be able to reason effectively about them. In earlier work (Sullivan et al. 1997), we showed that reasoning about them is hard and that formal mathematical theories of such mechanisms can provide a foundation for effective reasoning. In this paper, we present a new theory of interface negotiation and aggregation in COM. Our new theory is based on a relaxed interpretation of the COM specification. Our earlier theory reflected an interpretation of the specification in which components had to be designed to follow COM-specified rules for interface negotiation and aggregation under any possible usage. Our new, strictly weaker theory requires only that actual system executions not manifest any violations of the rules. Architectural styles using mediators that we showed to be untenable under the earlier theory are tenable under this one provided that designers follow certain rules. We derive these necessary and sufficient conditions for legal use of interface negotiation in the presence of aggregation. Our results provide a basis for documenting what engineers must not do to use aggregation and interface negotiation properly.

1. Introduction

Reconciling requirements for (1) the efficient integration of independently developed and evolving components and (2) ease of the evolution of systems built from such components is a demanding task that requires novel architectural standards, styles, mechanisms and idioms [Sullivan 1994, Sullivan & Notkin 1992]. Industrial software designers are beginning to meet this challenge with advanced mechanisms and compositional reuse models. Two important new mechanisms that feature prominently in the Component Object Model [COM 1996] are interface negotiation and aggregation, a mechanism for efficient composition. Unfortunately, the introduction of architectural innovations of this sort without extremely careful analysis can create unintended and subtle design "land-mines" that remain latent until unwary adopters inadvertently detonate them. In earlier work we showed that this concern is not academic: The COM standard has architecturally critical properties that are easy to overlook and hard to reason about [Sullivan et al. 1997].

The importance of having a sound basis for engineers to reason about architectural standards should not be underestimated. First, on the positive side, mechanisms such as interface negotiation and aggregation are both general and extremely useful; so they are worthy of careful scientific study. Interface negotiation appears to be especially important for evolution in a context of independently evolving components, which is perhaps the central feature of a marketplace of reusable components as envisaged by McIlroy at the founding of the software engineering field [McIlroy, 1969]. Second, interface negotiation and aggregation are central to COM, in particular; and because COM is so widely used, it behooves us to understand the mechanisms on which it is based. Third, not understanding these mechanisms puts developers at serious risk. Interface negotiation and aggregation are at the very heart of COM, and COM is the basis for critical, early architectural design decisions. The erroneous use of mechanisms at this stage can severely compromise a system design. Moreover, because these mechanisms provide the basis for late integration of independently developed components, errors can remain undiscovered until disastrously late, after a system is fielded.

In earlier work, we showed that the use of formal methods in-the-small facilitates precise reasoning about such standards. We did not seek a theory of architecture in general, or of COM in its entirety. Rather, we focused on developing a specific theory of interface negotiation and aggregation in COM: the aspects relevant to our design situation. Specifically, we developed a formal theory that showed that an architectural style, intended for use in a commercial multimedia authoring system, and based on mediators [Sullivan 94, Sullivan & Notkin 1992] and COM, was untenable [Sullivan et al. 1997].

That theory of COM formalized key aspects of the published specification of COM [COM 1996]. A key question is whether the theory is a valid model of COM. We tried to validate the theory, having it reviewed by the designers of COM, who agreed that it captured the structure of COM. Nevertheless, one can ask whether our negative results characterize COM, or whether they are artifacts of an artificially restrictive theory.

We believe that our earlier theory captures one widely held view of the meaning of the COM specification. Somewhat astonishingly, however, we have discovered a second interpretation of the specification that leads to a different theory in which our earlier conclusions are weakened. Our earlier theory reflected an interpretation in which components had to be designed to follow COM rules about interface negotiation and aggregation irrespective of any particular operational context. Our new theory reflects an interpretation requiring only that any actual execution of a system not manifest any violations of the rules of COM.

Our earlier work hinted at rules beyond those stipulated in the COM specification that would enable the use of mediators, almost as intended, without problems. Our new theory leads to such a set of rules. We derive necessary and sufficient conditions for the legal use of interface negotiation and aggregation. One result is that the architectural style using mediators that was untenable under the earlier theory is tenable under the new theory, provided that the additional conditions are satisfied. If our new theory is valid, our

results provide a basis for documenting what practicing engineers must do and not do to use COM aggregation and interface negotiation properly. Unfortunately, the new interpretation is even subtler than our earlier theory. The interaction between COM interface negotiation and aggregation appears to be inherently complex. The results we present here bolster our earlier claim that the light-weight use of formal methods can significantly aid in reasoning about subtle but critical aspects of widely used software architectural standards.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts of COM and formalizes them within our new theory. Section 3 gives a new definition of COM component that we use in our dynamic theory. Section 4 formalizes the interface negotiation rules and defines component legality. Section 5 formalizes aggregation and discusses the issues of component identity under aggregation. In section 6 we state and prove three necessary conditions of legality of an aggregated component. Section 7 contains the statement and proof of a sufficient condition of legality of an aggregated component. Section 8 concludes

2. Informal Introduction to COM

COM itself comprises a prescriptive object model for interoperable binary components, and a runtime infrastructure supporting components that conform to the model. The runtime infrastructure provides such services as component creation. In this paper we are concerned only with the object model. When we refer to COM, we are implicitly referring only to the object model and not to the runtime infrastructure.

The heart of COM is a set of rules that define what a component is and how it must behave in certain key areas. For instance, COM specifies that components provide services through multiple interfaces; that components interact with each other only through interfaces; and that components must support certain behaviors enabling components to negotiate with each at runtime for desired kinds of interfaces.

Another key part of the standard defines compositional reuse mechanisms. COM *delegation* is a traditional object-oriented composition mechanism in which the implementation of a service provided by an *outer* component uses the services of encapsulated, *inner* objects. In this paper we are concerned with a novel mechanism called *aggregation.* Aggregation permits an outer object to "pretend" to its clients that an interface implemented by an inner object is actually one of its own interfaces. *Aggregation* supports efficient composition in that requests to "the outer" can be handled by "the inner" without an additional call. Aggregation can also play a role in COM analogous to class inheritance in object-oriented programming.

A component is said to be a legal COM component if and only if it implements these mechanisms according to rules given in the COM specification. Because more and more systems use COM as their foundation it is important to have a clear understanding of the limitations that these rules impose on designers. In particular, an important property that we have been exploring is whether systems of communicating COM components can be encapsulated by outer components such that that the following conditions hold:

1) some services of inner components are made visible as services of the outer component
2) the performance penalty for the encapsulation boundary imposed by the outer is negligible
3) the outer component is a legal COM component, i.e., it satisfies all the rules of COM

In this compositional style a complex is encapsulated, turning it into a new. One might use this style to encapsulate a subsystem comprising off-the-shelf components integrated by component mediators [Sullivan 94] into a new component that could then be used as a basic building block at the next level of composition. The key idea is that the aggregate abstracts the subsystem by hiding some of the services (interfaces) provided by some of the individual subsystem components (Figure 1).

Such a style appeared to be natural, involving little more than an information-hiding encapsulation of complex subsystems. However, our analysis [Sullivan et al. 1997] revealed serious problems that made selective hiding of services of inner objects and the legality of the inner components of such aggregates mutually exclusive.

Something had to give. For example, application designers could be prohibited from selectively hiding the interfaces of aggregated components. However, the designers of COM rejected that option, stating that they required selective hiding of aggregated interfaces to be permitted. We were led to introduce the notion of non-conforming objects (objects that did not follow the rules of COM) into our theory, and to permit non-conforming components inside aggregates. Other options were to change the specification of COM (which we rejected, because we wanted to study COM as it was defined) or to find another interpretation of the specification under which the difficulty would no longer obtain.

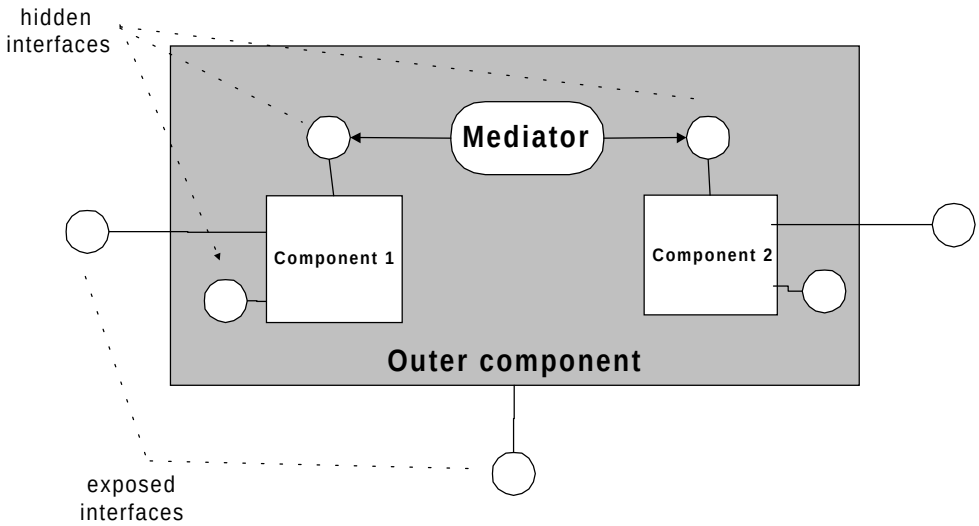


Figure 1. An advanced compositional style

To our surprise we found that the published COM specification does admit two distinct interpretations of the interface negotiation rules. We call these two interpretations "static" and "dynamic." Our earlier work formalized the static interpretation and led to the conclusion that legal COM components could not be aggregated (with selective hiding of interfaces). The notion of legality of a component under the dynamic interpretation is strictly weaker than that under the static interpretation.

In this paper we present a formal theory of COM aggregation and interface negotiation based on the dynamic interpretation. Under the dynamic interpretation, it is possible to assemble aggregates of communicating legal COM components that hide some of their services and still never violate the rules of COM for the duration of their lifetime. However, in some cases to achieve this "legality" requires cooperation from all the components of the aggregate. The cost, then of the relaxed theory is that the legality of components and systems ceases to be a property of individual components alone and becomes instead a non-local system property.

The results presented in this paper do not contradict the basic conclusion presented in our earlier work: By any reasonable measure, COM aggregation and interface negotiation are subtle mechanisms whose proper usage requires careful reasoning. Fortunately, under the dynamic interpretation, we have been able to identify necessary and sufficient conditions for component legality in the face of interactions among

aggregated components. Thus, we have applied formal reasoning to deduce a set of rules conformance to which ensures proper usage of aggregation and interface negotiation. The extent to which these impede architectural design in practice is not yet clear, and will be the subject of future investigations.

3. COM components

In this section we formally define the basic notions of Component Object Model: interface instances, their specifications and unique identifiers, the IUnknown interface and components. For details on the object model we refer the reader to [COM96, Kindel 95, Rogerson 97, Sullivan et al. 97].

3.1. Interfaces

The only way that a COM component provides services and communicates to clients is through one or more *interfaces*. A COM interface is a standardizes binary structure (a pointer to table of function pointers), however this fact is irrelevant for the purposes of our analysis. More importantly, every interface instance corresponds to one or more *interface specifications* or *types* that declare the operations of that interface. A globally unique *interface identifier* or *IID* identifies every interface specification. We formalize interfaces, specifications and IIDs in Z as given sets.

$$[IID, Interface, InterfaceSpec]$$

Every COM interface instance must implement the functions of the IUnknown interface specification. This specification, whose IID also has a special name IID_IUnknown defines the QueryInterface function which is in the hart of the COM interface negotiation mechanism. We formalize the existence of IUnknown and its IID in the following axiom:

$$\begin{array}{|l} IUnknown : InterfaceSpec \\ IID\_IUnknown : IID \\ \hline \end{array}$$

We model the association of each interface specification with its unique *IID* as a total one-to-one function that, in particular, associates *IUnknown* with *IID_IUnknown.*

$$\begin{array}{|l} IIDOfInterfaceSpec : InterfaceSpec \rightarrowtail IID \\ \hline IIDOfInterfaceSpec\ (IUnknown) = IID\_IUnknown \end{array}$$

We use a relation *InterfaceSpecOf* to model the one-to-many relationship between interface instances and specifications. In particular, every interface satisfies at least the IUnknown specification.

$$\begin{array}{|l} InterfaceSpecOf : Interface \leftrightarrow InterfaceSpec \\ \hline Interface \times \{IUnknown\} \subseteq InterfaceSpecOf \end{array}$$

Finally, composing *InterfaceSpecOf* with *IIDOfInterfaceSpec* we get a relation *IIDOfInterface* that maps an interface instance to the set of IIDs of specifications that the instance satisfies. In particular, this relation maps every interface into at least *IID_IUnknown*.

$$\begin{array}{|l} IIDOfInterface : Interface \leftrightarrow IID \\ \hline IIDOfInterface = InterfaceSpecOf \fatsemi IIDOfInterfaceSpec \end{array}$$

Figure 2 illustrates the relationships between interfaces, interface specifications and unique interface identifiers.

3.2. Queries

COM requires that every interface implements a special operation called *QueryInterface*. *QueryInterface* allows a client with a pointer to any interface on an object to obtain pointers to other interfaces on the same object. *QueryInterface* allows objects that were designed independently to negotiate

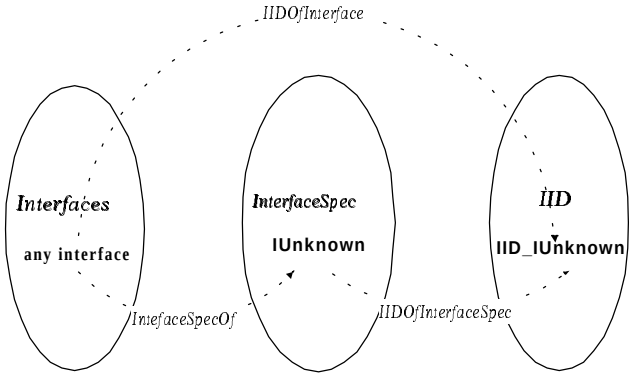


Figure 2 Relationships between interfaces, specifications and IIDs

communication protocols dynamically. *QueryInterface* takes an *IID* as a parameter and returns, through another parameter, a pointer to an interface of the designated type on the same object. If the object does not support the designated type of interface, *QueryInterface* returns a null pointer. The return value indicates whether an interface was returned successfully.

In order to make the interface negotiation process simple and logical, section 3.3.1.1 of COM specification [COM95] defines a set of rules that every implementation of QueryInterface must follow. Somewhat surprisingly, these rules are stated as constraints on *sequences* of calls to QueryInterface functions of interfaces on the same component.

For example, the following paragraph of section 3.3.1.1 defines the "static interface set" or as we call it *stability* property that every QueryInterface implementation must have. Appendix A contains the full text of the QueryInterface specification.

HRESULT IUnknown::QueryInterface(iid, ppv)

It is required that the set of interfaces accessible on an object via QueryInterface be static, not dynamic, in the following precise sense.[1] Suppose we have a pointer to an interface

```
ISomeInterface * psome = (some function returning an ISomeInterface *);
```

where ISomeInterface derives from IUnknown. Suppose further that the following operation is attempted:

```
IOtherInterface * pother;
HRESULT   hr;
hr=psome->QueryInterface(IID_IOtherInterface, &pother);        //line 4
```

[1] While this set of rules may seem surprising to some, they are needed in order that remote access to interface pointers can be provided with a reasonable degree of efficiency (without this, interface pointers could not be cached on a remote machine). Further, as QueryInterface forms the fundamental architectural basis by which clients reason about the capabilities of an object with which they have come in contact, stability is needed to make any sort of reasonable reasoning and capability discovery possible.

Then, the following must be true:

- If hr==S_OK, then if the QueryInterface in "line 4" is attempted a second time from the same psome pointer, then S_OK must be answered again. This is independent of whether or not pother->Release was called in the interim. In short, if you can get to a pointer once, you can get to it again.
- If hr==E_NOINTERFACE, then if the QueryInterface in line 4 is attempted a second time from the same psome pointer, then E_NOINTERFACE must be answered again. In short, if you didn't get it the first time, then you won't get it later.

Effectively, this says that if in a sequence of calls to QueryInterface functions there is a subsequence of two calls through the same interface that ask for the same IID, then both of the calls must succeed or both fail. Unfortunately, although the specification claims to state the required properties of QueryInterface "in the following precise sense" there are at least two distinct interpretations of this rule as well as the other rules. The *static* interpretation assumes that the requirements imposed by the COM specification upon sequences of calls to QueryInterface apply to all *potential* sequences including those not actually executed and even those that could not possibly be executed during the lifetimes of the interfaces in question. The theory of COM that we presented in our earlier work [SSM97] uses this interpretation. It formalizes requirements upon the entire collection of all potential sequences of calls to QueryInterface as axioms of the *QI* function defined as follows:

$$QI : Interface \times IID \pfun Interface$$

For any given pair of interface i and IID d $QI(i, d)$ determines the result of every call to QueryInterface function of interface i asking for IID d. In particular, if $(i, d) \notin dom\ QI$, then every such call will return NULL (object does not have an interface of requested type). If $QI(i, d) = r$, then every such call will successfully return interface r. Thus QI implicitly defines all possible sequences of calls to QueryInterface functions of all interface instances on all components and there is no need to model such sequences explicitly.

An alternative *dynamic* interpretation of the rules of interface negotiation makes a weaker assumption. It assumes that only *actual* sequences of calls to QueryInterface functions of interfaces exposed by an object must follow the rules of the QueryInterface specification. By an actual sequence of calls we mean a sequence of calls that have been made during the lifetime of the object. The dynamic interpretation is the basis for a new theory that we will present for the rest of this paper along with a slightly modified version of the static theory that we first presented in [SSM97].

We begin with formalizing the informal notion of invocation of a QueryInterface function of an interface. We will call such invocations *queries* and model them as elements of set *Query* of triples *(i, d, r)* where *i* is the interface instance whose QueryInterface function was called, *d* is the IID requested and *r* is the result returned by the call to QueryInterface. In order to model both successful and unsuccessful QueryInterface calls we introduce a special interface constant *null*.

$$null : Interface$$

$$null \notin dom\ IIDOfInterface$$

We model an unsuccessful QueryInterface call as an *(i, d, null)* triple. A successful call corresponds to a triple having an interface other than *null* as its third element. Thus *null* roughly corresponds to the NULL interface pointer that QueryInterface returns when the component does not have the requested interface.
In order to emphasize the fact that *null* is a valid query result we introduce a synonym *QueryResult* for the *Interface* type that we will use to designate the type of query result values. We will use the type $Interface$ only for the variables that cannot assume value $null$.

$QueryResult == Interface$

Now we are ready to formally define a call to QueryInterface as an element of the *Query* set.

$$Query == \{\ receiver : Interface; request : IID; result : QueryResult \mid receiver \neq null \bullet (receiver, request, result)\ \}$$

We define *receiver, request* and *result* as accessor functions for queries returning respectively the first, second and third elements of the query triple.

$$receiver : Query \rightarrow Interface$$
$$request : Query \rightarrow IID$$
$$result : Query \rightarrow QueryResult$$

$$\forall i : Interface; d : IID; r : QueryResult \mid (i,d,r) \in Query \bullet receiver\ (i,d,r) = i \land request\ (i,d,r) = d \land result\ (i,d,r) = r$$

3.3. Components

In this section we give formal definitions of components for the static (*QI*-based) and dynamic (query sequence-based) models of COM.

3.3.1. *QI*-based (static) model

A COM component instance (or *component)* is an object that exposes a finite set of interfaces. The set of interfaces that the component exposes is defined recursively. Firstly, *firstInterface*, the interface that the COM component creation routine returned when it created the instance, is exposed. Secondly, every object exposes a distinguished interface (not necessarily distinct from *firstInterface*) that satisfies at least the *IUnknown* specification. In COM, this interface is called the distinguished *IUnknown* of the object. Lastly, if defined, the result of applying *QI* to an interface of an object is another interface on the same object. We define the set of *IIDs* of an object to be equal to the set of *IIDs* of the specifications that are satisfied by the individual interfaces of the object.

$Component_s$

$$interfaces : \mathbb{F}\ Interface$$
$$iids : \mathbb{F}\ IID$$
$$firstInterface,\ iunknown : Interface$$

$$\{firstInterface,\ iunknown\} \subseteq interfaces$$
$$\forall\ i : interfaces; d : IID \mid (i, d) \in dom\ QI \bullet QI(i, d) \in interfaces$$
$$iids = IIDOfInterface\ (\!|\ interfaces\ |\!)$$

Note that the set *interfaces* contains the interfaces that a component can *potentially* expose to its clients. The component does not necessarily expose all of them in its lifetime. Indeed, the static model of COM does not deal at all with the issue of interfaces that clients can potentially get from an object versus those they actually get. This distinction is abstracted out and we assume the rules of QueryInterface to apply to all potentially exposable interfaces, i.e., the members of *interfaces* set.

3.3.2. Sequence-based (dynamic) model

Our dynamic theory also models component instances as objects exposing one or more interfaces. However, in contrast with the static theory, where the emphasis was on the set of exposed interfaces, the essence of an object here is the sequence *queries* of calls to the QueryInterface functions of the interfaces on the object made during the lifetime of the object. Since the lifetime of every component instance is finite, *queries* is a finite sequence. As it was in the static model, *firstInterface* is the interface returned by the routine that created the instance. In addition, *pUnkOuter* is an interface, not necessarily distinct from *firstInterface*, that COM requires an aggregating component to expose to its aggregatees. We assume this variable to be *null* if the component is not an aggregator. In order to simplify our model we assume that every query receiver in the *queries* sequence is either *firstInterface* or *pUnkOuter* or a result of an earlier successful query (C1). In other words we assume that an object does not pass its interfaces to clients by means other than QueryInterface calls. Although not explicitly stated in the COM specification, this assumption has recently been confirmed by the designers of COM [Brockschmidt 1997]. The set *interfaces* is the set of interface instances the component exposes to its clients during its lifetime. As such it contains the interface *firstInterface* that the creator of the component obtains from the creation routine, *pUnkOuter* that the component passes to the objects it aggregates (if any) and the results of all successful queries on the component (C2). Unlike the *QI*-based definition, here it is specifically not the case that *interfaces* contains all the interface instances an object may potentially expose if queried for. The set *interfaces* contains only the interfaces *actually* exposed to some clients of the component during its lifetime. Likewise, *iids* contains the types of all the interfaces that the component actually exposes. (C3) formalizes the COM identity requirement that any query asking for IID_IUnknown must succeed and for any given object two such queries must return the same interface.

$$
\begin{array}{l}
\textit{Component}_d \\
\hline
queries \;:\; seq\ Query \\
firstInterface, pUnkOuter : Interface \\
interfaces \;:\; \mathbb{F}\ Interface \\
iids \;:\; \mathbb{F}\ IID \\
\hline
\forall n_2 : dom\ queries \quad \textbf{(C1)} \\
\quad \bullet\ receiver\ queries(n_2) \in \{firstInterface, pUnkOuter\} \\
\qquad \lor (\exists\ n_1 : \{1..n_2\} \bullet result\ queries(n_1) = receiver\ queries(n_2)) \\
\\
firstInterface \neq null \\
\\
interfaces = \{q : Query \mid q \in ran\ queries \bullet result\ q\ \} \quad \textbf{(C2)} \\
\qquad\qquad \cup \{firstInterface, pUnkOuter\} \setminus \{null\} \\
\\
iids = IIDOfInterface\ (\!|\ interfaces\ |\!) \\
\\
\forall q_1, q_2 : ran\ queries \quad \textbf{(C3)} \\
\quad \mid \{request\ q_1, request\ q_2\} = \{IID\_IUnknown\} \\
\quad \bullet\ result\ q_1 = result\ q_2 \ \land\ result\ q_1 \neq null \\
\hline
\end{array}
$$

3.4. COM identity

In this section we formalize the notion of component identity as defined by COM standard in its static and dynamic interpretations.

3.4.1. Static model

COM object identity is defined in terms of the distinguished *IUnknown* interfaces of components. The basis for identity is the requirement that every call to *QueryInterface* made through any interface of an object, with *IID_IUnknown* as a parameter, always returns the same, distinguished *IUnknown* interface of that object. The *identity axiom* of the static model formalizes this requirement.

$$\forall X : Component; i : Interface \mid i \in X.interfaces \bullet QI(i, IID\_IUnknown) = X.iunknown$$

COM defines object identity as follows: Given any two interfaces, you determine whether they are interfaces on the same object by querying for *IID_IUnknown* through each, then comparing the returned interfaces (pointers). We formalize COM object identity as a binary relation $=_{com,s}$. It is easy to see that $=_{com,s}$ is an equivalence relation.

$$\_ =_{com,s} \_ : Component \leftrightarrow Component$$

$$\forall X, Y : Component \bullet X =_{com,s} Y \iff X.iunknown = Y.iunknown$$

3.4.2. Dynamic model

The dynamic interpretation of the rules of COM naturally leads to a different notion of component identity. The identity *inherent* to a component is no longer applicable and is replaced by *manifested* identity. Under the dynamic interpretation one has no other way to find out whether two components are identical in the COM sense than to compare the results of queries for *IID_IUnknown* in the respective component query sequences. We call such query results *manifested identities.* Since whether a component gets a chance to manifest its identity depends on the sequence of queries that clients make to the component during its lifetime, manifested identity is not defined for every component. Instead we define it through $iunknown_d$, a partial function that maps a component into the common result of all queries for *IID_IUnknown* on its interfaces, if there are any such queries. (C3) guarantees that $iunknown_d$ is indeed a function.

$$iunknown_d : Component_d \nrightarrow Interface$$

$$dom\ iunknown_d = \{ C : Component_d \mid (\exists q : ran\ C.queries \bullet request\ q = IID\_IUnknown) \bullet C \}$$

$$\forall C : dom\ iunknown_d \bullet iunknown_d\ C = (\mu q : ran\ C.queries \mid request\ q = IID\_IUnknown \bullet result\ q)$$

Now we can define a new COM identity predicate.

$$\_ =_{com,d} \_ : dom\ iunknown_d \leftrightarrow dom\ iunknown_d$$

$$\forall X, Y : dom\ iunknown_d \bullet X =_{com,d} Y \iff iunknown_d\ X = iunknown_d\ Y$$

One can interpret the expression $X =_{com,d} Y$ as "components X and Y both have manifested their identities and those identities have been found equal". Obviously, a component that has not manifested its identity during its lifetime is simply not in the domain of the $=_{com,d}$ relation and the result of the comparison is undefined.

4. Formalizing the interface negotiation rules

In this section we give two formal models of the interface negotiation rules of COM. The first model is static (QI-based). It states the interface negotiation rules as constraints on the *QI* function. The second model is based on sequences of queries.

4.1. Static model

In order to be able to reason formally both about interfaces whose QueryInterface functions obey the rules of interface negotiation and those whose QueryInterface functions don't, we limit the domain of those rules to the subset $COMInterfaces_s$ of $Interface$. The elements of $COMInterfaces_s$ are called *legal* COM interfaces. The first requirement on a legal COM interface is that successful invocations of its *QueryInterface* operation always return interfaces that actually have the requested *IIDs* (Appendix A, lines 1,2)

$$\begin{array}{|l} COMInterfaces_s : \mathbb{P}\ Interface \\ \hline \forall i : COMInterfaces_s; d : IID \mid (i, d) \in dom\ QI \bullet QI(i, d) \mapsto d \in IIDOfInterface \end{array} \qquad (S1)$$

The other interface negotiation rules of COM include stability (Appendix A, lines 10-22), reflexivity (line 28), symmetry (lines 29-31) and transitivity (lines 32-37). Because we model QueryInterface as a partial function *QI*, the stability rule applies automatically to all interfaces, including those of $COMInterfaces_s$ set. We formalize the other three rules in the following axioms.

First, COM defines reflexivity to mean that if you have a legal COM interface *a* with type *iidA*, then calling *QueryInterface* on *a* for *iidA* must succeed. It is not required that the returned interface be *a* itself, unless *a* is the distinguished *IUnknown* and *iidA* is *IID_IUnknown*. Recall that *IIDOfInterface* associates an interface with all of the *IIDs* that it satisfies. We formalize the COM notion of reflexivity by stating that the domain of *QI* contains the subrelation of *IIDOfInterface* restricted to the subset of legal COM interfaces.

$$COMInterfaces_s \lhd IIDOfInterface \subseteq dom\ QI \qquad (S2)$$

Second, the symmetry rule in its "static" interpretation means that if you had a legal COM interface *a* of type *iidA*, and if calling QueryInterface on *a* with *iidB* would succeed in returning an interface *b*, then calling QueryInterface on *b* with *iidA* would have to succeed. This condition must hold regardless of whether the sequence of these two calls has actually been executed or it can potentially be executed or even if it cannot possibly ever be executed on a given component.

$$\begin{array}{l} \forall a, b : COMInterfaces_s; iidA, iidB : IID \qquad (S3) \\ \quad \mid (a, iidB) \in dom\ QI \\ \quad \bullet\ a \mapsto iidA \in IIDOfInterface \land QI(a, iidB) = b \implies (b, iidA) \in dom\ QI \end{array}$$

Finally, transitivity under the static interpretation of COM interface negotiation rules means, informally, that if QueryInterface can get you from "here to there" and "there to somewhere else," it can get you "here to somewhere else." The formal statement is similar to those in the preceding paragraphs. [2] Again, the word "can" in this rule implies that the subsequence in question need not be in the sequence of QueryInterface calls actually executed on the component.

[2] The specification actually gives an unorthodox definition of transitivity: informally, that you can get "from elsewhere back to here." The definition is not equivalent to the ordinary definition of transitivity, and it is not strong enough to ensure that *QueryInterface* operations have the required "anywhere-in-one-step" property. We therefore interpret the COM specification as using an erroneous definition of transitivity; and we have used the common definition in place of the unorthodox one.

$$\forall a, b, c : COMInterfaces_s; iidA, iidB, iidC : IID \quad \textbf{(S4)}$$
$$\mid \{(a, iidB), (b, iidC)\} \subseteq dom\ QI$$
$$\bullet\ a \mapsto iidA \in IIDOfInterface$$
$$\wedge\ QI(a, iidB) = b \wedge QI(b, iidC) = c$$
$$\Rightarrow\ (a, iidC) \in dom\ QI$$

Just as we had to distinguish legal COM interfaces, we also had to distinguish legal COM objects. We model legal COM objects as a subset of *Component* whose elements have only legal COM interfaces.

$$COMObjects_s : \mathbb{P}\ Component_s$$

$$\forall\ C : COMObjects_s \bullet C.interfaces \subseteq COMInterfaces_s$$

4.2. Dynamic model

The legality of a component in the static model of COM is determined by the legality of its essential part that is the set of interfaces. Likewise, in the dynamic model a component is considered legal if and only if its essential part is legal. In this case the essential part is the sequence of queries that the component receives in its lifetime. In this section we define a *legal query sequence* as a sequence of queries that satisfy the definition of QueryInterface and all the interface negotiation rules of COM, namely, stability, reflexivity, symmetry and transitivity.

First we formalize the requirement that every successful query in a sequence returns an interface of the requested type. This corresponds to the way COM specification defines QueryInterface.

CorrectResult

$$s : seq\ Query$$

$$\forall\ q : ran\ s \bullet result\ q \neq null \Rightarrow result\ q \mapsto request\ q \in IIDOfInterface \quad \textbf{(L1)}$$

Then we formalize the rule that requires the behavior of QueryInterface be stable (or "static" as the COM specification calls it) in the following sense. If a request for a given IID succeeds (fails) once, then any subsequent call to the QueryInterface function requesting the same IID must succeed (resp. fail). In the static theory this property of legal QueryInterface implementations was implicitly superseded by a stronger assumption. Since *QI* is a mathematical function, it always returns the same result for the same tuple of arguments. The following schema defines the set of stable query sequences. In any such sequence any two calls q_1 and q_2 to the QueryInterface function of the same interface asking for the same IID must both succeed or both fail.

Stable

$$s : seq\ Query$$

$$\forall\ q_1, q_2 : ran\ s \mid receiver\ q_1 = receiver\ q_2 \wedge request\ q_1 = request\ q_2 \quad \textbf{(L2)}$$
$$\bullet\ result\ q_1 = null \Leftrightarrow result\ q_2 = null$$

The next schema is the reflexivity rule. In a reflexive sequence any query that requests an interface of the same type as the query receiver, must succeed.

$$
\begin{array}{|l}
\textit{Reflexive} \\
s : \textit{seq Query} \\
\hline
\forall\ q : \textit{ran}\ s \bullet \textit{receiver}\ q \mapsto \textit{request}\ q \in \textit{IIDOfInterface} \implies \textit{result}\ q \neq \textit{null}
\end{array}
\qquad (L3)
$$

The following schema formalizes the symmetry requirement. If a sequence *s* contains a query to interface *x* for IID *iidY* that successfully returns interface *y*, then any later query to interface *y* asking for an IID of interface *x* in the same query sequence must succeed. Informally, this rule states that if QueryInterface actually gets you from "here" to "there" and you ask it to get you back "here" it must do so or else it is illegal. Compare this with the informal rendition of the symmetry rule in our static theory: "if you *can* get from here to there, you *can* get from there to here".

$$
\begin{array}{|l}
\textit{Symmetric} \\
s : \textit{seq Query} \\
\hline
\forall\ x, y : \textit{Interface};\ \textit{iidX}, \textit{iidY} : \textit{IID};\ r : \textit{QueryResult}; n_1, n_2 : \textit{dom}\ s \\
\quad \mid s(n_1) = (x, \textit{iidY}, y)\ \wedge\ x \mapsto \textit{iidX} \in \textit{IIDOfInterface} \\
\qquad \wedge\ n_1 < n_2\ \wedge\ y \neq \textit{null} \wedge s(n_2) = (y, \textit{iidX}, r) \\
\quad \bullet\ r \neq \textit{null}
\end{array}
\qquad (L4)
$$

And finally we formalize the transitivity rule requiring that for a sequence *s* containing three queries: to interface *x* for *iidY* returning interface *y*, to interface *y* for *iidZ*, returning interface *z*, to interface *x* for *iidZ* returning result *r* (in this order), the third query must succeed. Again the informal interpretation of this rule is that if you did get from here to there and from there to somewhere else, then you must be able to get from somewhere else back here. The dynamic theory does not constrain what must happen if you never actually tried to get from "here" to "there" and then to "somewhere else" but could potentially.

$$
\begin{array}{|l}
\textit{Transitive} \\
s : \textit{seq Query} \\
\hline
\forall\ x,y,z : \textit{Interface}; \textit{iidY},\textit{iidZ} : \textit{IID}; n_1,n_2, n_3 : \textit{dom}\ s; r : \textit{QueryResult} \\
\quad \mid n_1 < n_2 \wedge n_2 < n_3 \wedge x \neq \textit{null} \\
\qquad \wedge\ s(n_1) = (x, \textit{iidY}, y) \wedge s(n_2) = (y, \textit{iidZ},\ z) \wedge\ z \neq \textit{null} \\
\qquad \wedge s(n_3) = (x, \textit{iidZ}, r) \\
\quad \bullet\ r \neq \textit{null}
\end{array}
\qquad (L5)
$$

Now we are ready to define the set of legal query sequences. We define it as a subset of sequences of queries in which every sequence *s* satisfies the five requirements (L1) – (L5).

$$
\begin{array}{|l}
\textit{LegalQuerySequences} : \mathbb{P}\ \textit{seq Query} \\
\hline
\forall\ s : \textit{seq Query} \\
\quad \bullet\ s \in \textit{LegalQuerySequences} \iff \textit{CorrectResult} \wedge \textit{Stable} \wedge \textit{Symmetric} \wedge \textit{Reflexive} \wedge \textit{Transitive}
\end{array}
$$

4.3. Dynamic legality of a component

As one might expect, the dynamic model of COM defines a legal COM object as a component for which the actual sequence of queries made to its interfaces for its entire lifetime is a legal query sequence.

$$COMObjects_d : \mathbb{P}\, Component$$

$$\forall\, C : Component \bullet C \in COMObjects_d \iff C.queries \in LegalQuerySequences$$

Thus a component instance is legal if and only if the sequence of QueryInterface calls (and their results) that the clients of the instance have actually made to its interfaces complies with the rules stated in p. 3.3.1.1 of the COM specification document. Put another way, a component is legal if it does not demonstrate the opposite through its behavior for the duration of its lifetime.

4.4. Relationship between static and dynamic legality of a component

Our static and dynamic theories of COM define the legality of a COM component differently. In this section we will consider the relationship between the two definitions. We will show that a component that is legal under the static interpretation of the rules of COM is also legal under their dynamic interpretation. The converse is not true. Thus the notion of legality as defined in our dynamic theory is weaker than the static legality of components.

Unfortunately we cannot express this statement formally simply as $COMObjects_s \subseteq COMObjects_d$ because the two sets consist of elements of distinct schema types $Component_s$ and $Component_d$. In order to establish a pairwise correspondence between values of these types we define a total one-to-one mapping *Dyn* from $Component_s$ to $Component_d$. This mapping relates static and dynamic models of the same component.

$$Dyn : Component_s \rightarrowtail\!\!\!\!\!\!\rightarrow Component_d$$

$$\begin{array}{l}
\forall\, S : Component_s; D : Component_d \mid D = Dyn\ S \\
\quad \bullet\ S.firstInterface = D.firstInterface \\
\quad \wedge\ \forall\, i : Interface; d : IID; r : QueryResult \\
\qquad \mid (i, d, r) \in D.queries \\
\qquad \bullet\ if\ \ (i, d) \in dom\ QI\ \ then\ \ r = QI(i, d)\ \ else\ \ r = null \qquad \textbf{(D1)} \\
\quad \wedge\ D.pUnkOuter \neq null \Rightarrow D.pUnkOuter \in S.interfaces
\end{array}$$

Now we are ready to formally express the connection between the static and dynamic definitions of component legality.

Theorem 1: Static legality implies dynamic legality
$Dyn\ (\!|\ COMObjects_s\ |\!) \subseteq COMObjects_d$

Proof:
Let $S : Component_s$, $D : Component_d$ such that $S \in COMObjects_s$, $D = Dyn(S)$.
We shall show that $D \in COMObjects_d$ by way of showing that *D.queries* satisfies the axioms (L1) - (L5) of dynamic legality.
(L1) follows from the fact that since *S* is in $COMObjects_s$, all of its interfaces are $COMInterfaces_s$. By (D1) all of interfaces from *D.interfaces* are in *S.interfaces*. Then for all $i : Interface; d : IID;$ $r : QueryResult$ such that $(i, d, r) \in ran\ D.queries$ we have: $r \neq null \ \Rightarrow\ (r \mapsto d) = (QI(i, d) \mapsto d)$ and so by (S1) $r \mapsto d \in IIDOfInterface$.
(L2) follows from the fact that *QI* is a partial function and its result on given parameters, if defined, is always the same.
L3, L4, L5 follow from the reflexivity, symmetry and transitivity properties of *QI* respectively.
Let $i : Interface; d : IID; r : QueryResult \mid (i, d, r) \in ran\ D.queries \wedge\ i \mapsto d \in IIDOfInterface$. By (D1) $D.interfaces \subseteq S.interfaces$ and by (C1) $i \in D.interfaces$. Thus $i \in S.interfaces$ and so

$i \in COMInterfaces_s$ because $S \in COMObjects_s$. By the reflexivity property (C2) $(i, d) \in dom\ QI$ and by (D1) $r = QI(i, d)$. We conclude that $r \neq null$, since $null \notin dom\ IIDOfInterface$ and $r \mapsto d \in IIDOfInterface$ by (S1). Thus the dynamic reflexivity property (L3) holds for $D.queries$.
Now we will show that (L4) holds for $D.queries$.
Let $x, y : Interface;\ iidX, iidY : IID;\ r : QueryResult; n_1, n_2 : dom\ s$ such that $D.queries(n_1) = (x, iidY, y),\ \ x \mapsto iidX \in IIDOfInterface, n_1 < n_2$ and $D.queries(n_2) = (y, iidX, r)$. We need to show that $r \neq null$. First, observe by combining (C1), $D.interfaces \subseteq S.interfaces$ and $S \in COMObjects_s$ that $\{x, y\} \subseteq COMInterfaces_s$. Second, by (D1) $QI(x, iidY) = y$ and so by (S3) $(y, iidX) \in dom\ QI$. Applying (D1) again we get $r = QI(y, iidX)$ that implies $r \neq null$.
We have shown that the dynamic symmetry property (L4) holds for $D.queries$.
The proof of the fact that (L5) also holds for $D.queries$ is similar and is left as an exercise for the reader.
Since all of the schemas $CorrectResult$, $Stable$, $Symmetric$, $Reflexive$ and $Transitive$ (L1) - (L5) hold for $D.queries$, by definition of $LegalQuerySequences$ set, $D.queries \in LegalQuerySequences$ and so $D \in COMObjects_d$.
□

5. Aggregation

In this section we use our dynamic theory to develop a formal model of COM aggregation in terms of sequences of queries.

5.1. Dynamic model of aggregation

The COM specification document defines the rules of aggregation and those of QueryInterface using similar definition styles. The approach used for the QueryInterface rules as we have observed consists of imposing constraints on the behavior of well-implemented QueryInterface functions. COM formulates these constraints as rules (e.g., symmetry, reflexivity, transitivity) that all legal sequences of calls to QueryInterface must follow. Similarly, COM defines aggregation in a set of the following six rules [COM96, section 6.6.2] that constrain the behavior of the outer (aggregating) and the inner (aggregated) components.

1. When creating the inner object, the outer object must pass its own IUnknown to the inner object through the pUnkOuter parameter of IClassFactory::CreateInstance. pUnkOuter in this case is called the "controlling unknown."
2. The inner object must check pUnkOuter in its implementation of CreateInstance. If this parameter is non-NULL, then the inner object knows it is being created as part of an aggregate. If the inner object does not support aggregation, then it must fail with CLASS_E_NOAGGREGATION. If aggregation is supported, the inner object saves pUnkOuter for later use, but does not call AddRef on it. The reason is that the inner object's lifetime is entirely contained within the outer object's lifetime, so there is no need for the call and to do so would create a circular reference.
3. If the inner object detects a non-NULL pUnkOuter in CreateInstance, and the call requests the interface IUnknown itself (as is almost always the case), the inner object must be sure to return its non-delegating IUnknown.
4. If the inner object itself aggregates other objects (which is unknown to the outer object) it must pass the same pUnkOuter pointer it receives down to the next inner object.
5. When the outer object is queried for an interface it exposes from the inner object, the outer object calls QueryInterface in the non-delegating IUnknown to obtain the pointer to return to the client.
6. The inner object must delegate to the controlling unknown, that is, pUnkOuter, all IUnknown calls occurring in any interface it implements other than the non-delegating IUnknown.

In our dynamic theory we render these rules in terms of interfaces and query sequences. We define $Aggregates_d$ as a binary relation on the set of components which in this case is $Component_d$. For every pair of components O and I where O aggregates I we state that they must have at least one interface that they share. In other words their sets of interfaces exposed to clients intersect (A1). In the spirit of our dynamic model we do not consider aggregation a situation where the outer component creates an inner component but never returns its interfaces to clients. Such a set up is of no interest to us because it does not actually *manifest* aggregation and from the client point of view is no different from having no inner

component at all. We specify that according to rule 1 above the outer *pUnkOuter* interface must be non-null (A2). By rule 3 the first interface of the inner must be of type *IUnknown* only (A3). This is the hidden non-delegating iunknown interface of the inner. It is called "hidden" because it is never exposed to clients by the outer (A4). Rule 4 requires that if the inner is itself an aggregator, it must pass down the *pUnkOuter* of the outer component and not its own pUnkOuter (A5). For simplicity we do not consider the situations of multi-level aggregation in our theory, however the theory can easily be generalized to include such aggregates. Finally, (A6) states that a component cannot have more than one aggregator, the inverse of $Aggregates_d$ relation is a partial function.

$$Aggregates_d : Component_d \leftrightarrow Component_d$$

$$\forall\, I, O : Component_d \mid O \mapsto I \in Aggregates_d$$
$$\bullet\ I.interfaces \cap O.interfaces \neq \varnothing \qquad (A1)$$
$$\wedge\ O.pUnkOuter \neq null \qquad (A2)$$
$$\wedge\ InterfaceSpecOf(\{I.firstInterface\}) = \{IUnknown\} \qquad (A3)$$
$$\wedge\ I.firstInterface \notin O.interfaces \qquad (A4)$$
$$\wedge\ I.pUnkOuter = null \qquad (A5)$$

$$Aggregates_d^{-1} \in Component_d \nrightarrow Component_d \qquad (A6)$$

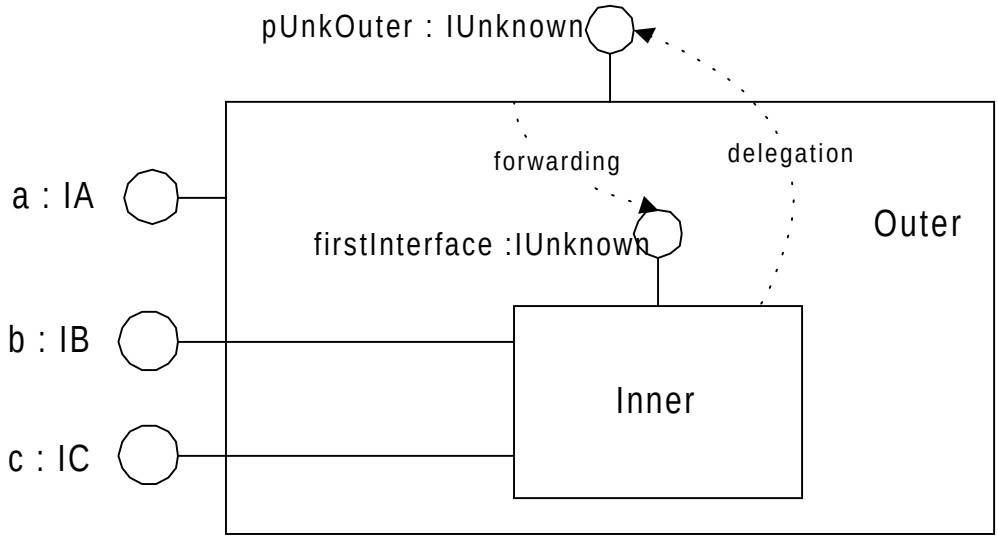


Figure 3: a simple aggregate

We want to emphasize that the mechanism of aggregation allows two or more independent components to *share* their interfaces. This means that the QueryInterface functions of elements of an aggregate may in fact return interfaces implemented by a different component. The outer may return interfaces implemented by the inner and vice versa. Because for the purposes of our analysis we still need to distinguish the sets of interfaces *implemented* by different objects, we define a function *NativeInterfaces* that maps an aggregating component into the set of interfaces it implements and exposes to clients during its lifetime. According to rule 5 above this set includes only those interfaces, that are not obtained from an inner

component by querying its $I.firstInterface$ (A7). We require that $O.pUnkOuter$ be a native interface of O. Likewise, we define $NativeIIDs$ as a function that maps an aggregated component into a set of IIDs interfaces of which that component implements and exposes to clients. The set of native IIDs of an aggregated component consists of the IIDs for which its clients have successfully queried its non-delegating hidden IUnknown interface (A8). Although this last definition looks a little unnatural at first, it follows from the combination of rules 5 and 6 above. For the aggregate on Figure 3 $NativeInterfaces(Outer)$ may, for example, be $\{pUnkOuter, a\}$ and $NativeIIDs(Inner)$ can be $\{ IB, IC \}$, for a suitable sequence of queries to interfaces of *Inner* and *Outer*.

$$
\begin{array}{l}
NativeInterfaces : dom\ Aggregates_d \rightarrow \mathbb{F}\ Interface \\
NativeIIDs : ran\ Aggregates_d \rightarrow \mathbb{F}\ IID \\
\hline
\forall\ O : dom\ Aggregates_d \qquad (A7) \\
\quad \bullet\ NativeInterfaces\ O = \{\ a : O.interfaces \mid (\forall\ I : Aggregates_d(\!|\{O\}|\!);\ iidA : IID \\
\qquad\qquad \bullet\ (I.firstInterface, iidA, a) \notin ran\ I.queries) \bullet a \\
\} \\
\quad \wedge\ O.pUnkOuter \in NativeInterfaces\ O \\
\\
\forall\ I : ran\ Aggregates_d \qquad (A8) \\
\quad \bullet\ NativeIIDs\ I = \{\ q : ran\ I.queries \mid receiver\ q = I.firstInterface \wedge result\ q \neq null \bullet request\ q\ \}
\end{array}
$$

Rules 5 and 6 of aggregation prescribe the necessary actions that QueryInterface functions of the interfaces on the outer and inner components respectively must perform when clients call them. Expressed in the language of our dynamic theory these rules establish 1-1 correspondences between some subsequences of the query sequences of the outer and inner components. *StrictCorrespondence* defines the class of all such correspondences as order preserving 1-1 functions from indices to indices.

$$StrictCorrespondence == \{\ f : \mathbb{N} \rightarrowtail \mathbb{N} \mid (\forall n_1, n_2 : dom\ f \bullet n_1 < n_2 \Rightarrow f\ n_1 < f\ n_2) \bullet f\ \}$$

Rule 5 that we call the *forwarding* rule requires the outer component forward to the hidden non-delegating iunknown interface of the inner every query for an interface of the inner component that the outer wants to expose, and then return the result of that subsequent query to the client. We represent the fact of forwarding as a mapping *Forward* from all pairs *(O, I)* of components in the $Aggregates_d$ relation to strict correspondences $f_{O,I}$ from the indices of the outer query sequence to the indices of the query sequence of the inner component. The domain of $f_{O,I}$ is restricted to the indices of only those queries to interfaces of the outer that are not interfaces of the inner (A9). The image of such an index corresponds to a query from the inner component query sequence that has the same *request* and *result* parts but whose *receiver* is the hidden non-delegating iunknown interface (A10).

$$
\begin{array}{l}
Forward\ :\ Component_d \times Component_d \ \nrightarrow\ StrictCorrespondence \\
\hline
dom\ Forward = Aggregates_d \\
\forall\ I, O : Component_d;\ f_{O,I} : StrictCorrespondence \\
\quad \mid\ O \mapsto I\ \in\ Aggregates_d \\
\qquad \wedge\ f_{O,I} = Forward(O, I) \\
\quad \bullet\ dom\ f_{O,I} = \{i : dom\ O.queries \mid\ receiver(O.queries\ i) \notin NativeInterfaces(O) \bullet i\} \qquad (A9) \\
\qquad \wedge\ ran\ f_{O,I} \subseteq\ dom\ I.queries \\
\qquad \wedge \forall\ n : dom\ f_{O,I} \\
\qquad\quad \bullet\ I.queries(f_{O,I}\ n) = (I.firstInterface, request(O.queries\ n), result(O.queries\ n)) \qquad (A10)
\end{array}
$$

To illustrate the action of $f_{O,I}$ consider the aggregate on Figure 3. Suppose that the sequences of calls that *Outer* and *Inner* components have received are as follows:

$$Outer.queries = \langle\!\langle (a\ IID\_IUnknown\ pUnkOuter)\ (a\ IB\ b)\ (b\ IC\ c)\ (pUnkOuter\ IC\ c)\ (pUnkOuter\ IA\ a) \rangle\!\rangle$$

$f_{O,I}$

$$Inner.queries = \langle\!\langle (firstInterface\ IB\ b)\ (b\ IC\ c)\ (firstInterface\ IC\ c)\ (c\ IA\ a) \rangle\!\rangle$$

Then the arrows indicate the pairs of queries related by $f_{O,I}$.

We model sharing interfaces between the outer and inner components of an aggregate as a mapping *Share* that defines for every pair $O \mapsto I$ in the $Aggregates_d$ relation a strict correspondence $p_{O,I}$ between indices of queries in the query sequences of *I* and *O*. When a client queries a shared interface, an appropriate query appears in both query sequences. $p_{O,I}$ establishes a one-to-one correspondence between such queries in the query sequences of *I* and *O*.

$$Share : Component_d \times Component_d \nrightarrow StrictCorrespondence$$

$$\begin{array}{l}
dom\ Share = Aggregates_d \\
\forall\ I, O : Component_d;\ p_{O,I} : StrictCorrespondence \\
\quad \mid O \mapsto I \in Aggregates_d \\
\qquad \wedge\ p_{O,I} = Share(O, I) \\
\quad \bullet\ dom\ p_{O,I} = \{\ n : dom\ I.queries \mid receiver(I.queries\ n) \in O.interfaces \bullet n\ \} \\
\qquad \wedge\ ran\ p_{O,I} = \{\ m : dom\ O.queries \mid receiver(O.queries\ m) \in I.interfaces \bullet m\ \} \\
\qquad \wedge\ \forall n : dom\ p_{O,I} \bullet O.queries(p_{O,I}\ n) = I.queries(n) \qquad \textbf{(A11)}
\end{array}$$

Again, for the same query sequences of *Outer* and Inner as in the previous example we get the following sharing correspondence $p_{O,I}$:

$$Outer.queries = \langle\!\langle (a\ IID\_IUnknown\ pUnkOuter)\ (a\ IB\ b)\ (b\ IC\ c)\ (pUnkOuter\ IC\ c)\ (pUnkOuter\ IA\ a) \rangle\!\rangle$$

$p_{O,I}$

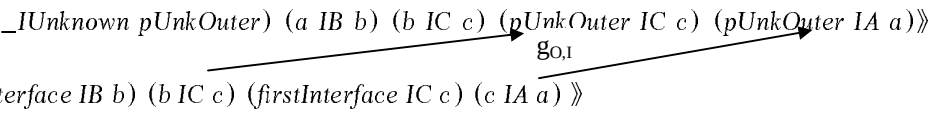
$$Inner.queries = \langle\!\langle (firstInterface\ IB\ b)\ (b\ IC\ c)\ (firstInterface\ IC\ c)\ (c\ IA\ a) \rangle\!\rangle$$

Rule 6 establishes a correspondence between the queries to the delegating interfaces of the inner and the queries to the *pUnkOuter* interface of the outer component. We define a mapping from aggregates $O \mapsto I$ to order preserving 1-1 correspondences $g_{O,I}$ between the indices of the queries to the inner and to the outer in their respective sequences. For every query to a delegating interface of the inner, $g_{O,I}$ gives the index of a query in the outer query sequence that the inner sends to the outer in order to delegate the original query (A12).

$$Delegate\ :\ Component_d \times Component_d \nrightarrow StrictCorrespondence$$

$$\begin{array}{l}
dom\ Delegate = Aggregates_d \\
\forall\ I, O : Component_d;\ g_{O,I} : StrictCorrespondence \\
\quad \mid O \mapsto I \in Aggregates_d \\
\qquad \wedge\ g_{O,I} = Forward(O, I) \\
\quad \bullet\ dom\ g_{O,I} = \{ i : dom\ I.queries \mid receiver(I.queries\ i) \neq I.firstInterface \bullet i \} \\
\qquad \wedge\ ran\ g_{O,I} \subseteq dom\ O.queries \\
\qquad \wedge\ \forall n : dom\ g_{O,I} \\
\qquad\qquad \bullet\ O.queries(g_{O,I} n) = (O.pUnkOuter, request(I.queries\ n), result(I.queries\ n)) \qquad \textbf{(A12)}
\end{array}$$

Once again we use the same pair of query sequences of *Inner* and *Outer* to illustrate the action of the delegation correspondence $g_{O,I}$.



We assume that QueryInterface called in the query to a delegating interface does nothing but delegates the call to *pUnkOuter*, i.e., calls its QueryInterface function. Thus if a client queries an interface of the inner that is delegating and shared with the outer component, then the copy of this query and the delegation query are next to each other in the outer query sequence.

$$\begin{aligned}&\forall\ I, O : Component_d; n : \mathbb{N}\ ; g_{O,I}, p_{O,I} : StrictCorrespondence \qquad (A13)\\ &\quad |\ O \mapsto I\ \in\ Aggregates_d\\ &\quad\ \wedge\ g_{O,I} = Delegate(O,I) \wedge p_{O,I} = Share(O,I)\\ &\quad\ \wedge\ dom\ g_{O,I} \cap dom\ p_{O,I} \neq \varnothing\\ &\quad\ \wedge\ n \in dom\ g_{O,I} \cap dom\ p_{O,I}\\ &\quad \bullet\ g_{O,I}(n) = p_{O,I}(n)+1\end{aligned}$$

5.2. COM identity under aggregation

Theorem 2: Dynamic COM Component Identity
If a component *outer* aggregates a component *inner* and *inner* reveals its identity through an interface other than its hidden non-delegating IUnknown, then *outer* and *inner* share dynamic object identity as defined for COM components.

$$\begin{aligned}&\forall I, O : Component_d\\ &\quad |\ O \mapsto I \in Aggregates_d\\ &\quad\ \wedge\ \exists\ q : ran\ I.queries \bullet request\ q = IID\_IUnknown \wedge\ receiver\ q \neq I.firstInterface \qquad (A14)\\ &\quad \bullet\ I =_{com,d} O\end{aligned}$$

Proof:
First we shall demonstrate that the relation $=_{com,d}$ is indeed defined for components *I* and *O*. This is equivalent to $iunknown_d$ function being defined for both of the components in question. Recall that

$$dom\ iunknown_d = \{\ C : Component_d \mid (\exists\ q : ran\ C.queries\ \mid request\ q = IID\_IUnknown) \bullet C\ \}$$

It follows from (A14) that $I \in dom\ iunknown_d$.
Let *n* be the index of a query $q \in ran\ I.queries$ such that $receiver\ q \neq I.firstInterface$. Such a query exists by (A14). Then *n* is in the domain of the delegation strict correspondence function *g=Delegate(O, I)*. By (A12) the image of *n* under *g* identifies a query $q' = O.queries(g\ n)$ in *O.queries* that requests for the same IID as *q*, namely *IID_IUnknown*. Thus *O* manifests its identity and so $O \in dom\ iunknown_d$.
It remains to show that $iunknown_d\ I = iunknown_d\ O$, in other words that there exists a pair of queries for *IID_IUnknown* in *I.queries* and *O.queries* respectively that have identical results.
Observation that *q* and *q'* are just such two queries completes the proof.
□

The premise (A14) that *inner* manifests its identity through an interface other than its hidden non-delegating IUnknown is essential as the following counterexample illustrates. Consider the aggregate on Figure 3. Suppose that the inner and outer components have the following parameters:

$Outer.firstInterface = Outer.pUnkOuter;$
$Outer.queries = \langle\!\langle (pUnkOuter, IB, b)\ (pUnkOuter, IID\_IUnknown, pUnkOuter) \rangle\!\rangle$

$Inner.pUnkOuter = null;$
$Inner.queries = \langle\!\langle (firstInterface, IB, b)\ (firstInterface, IID\_IUnknown, firstInterface) \rangle\!\rangle$

Although the components *O* and *I* satisfy all the requirements of aggregation in its dynamic interpretation and manifest their identities, so that $iunknown_d$ is defined for both of them, we have

$iunknown_d(Outer) = pUnkOuter$
$iunknown_d(Inner) = firstInterface$

Thus $Outer \neq_{com,d} Inner$.

We conclude that under the dynamic interpretation of the rules of COM the parts of an aggregate are not necessarily COM identical as it was the case under the static interpretation. An extra premise that the inner must manifest its identity through an interface other than the non-delegating IUnknown reveals the mechanics of aggregation. It is the delegation of QueryInterface calls that makes the COM identities of the parts of an aggregate equal. As long as we access the identity of the inner through a non-delegating interface, we may get a result different from the outer's identity and from the identities of all other inner components.

6. Necessary conditions of legality of an aggregated component

Our static theory of the rules of COM predicted that selective hiding of interface types of an inner aggregate component implies that we no longer can assume that the rules of COM in their static interpretation hold for that component. In particular, we cannot assume that the symmetry, reflexivity and transitivity properties hold for all *potential* sequences of queries to the interfaces of that component. This, however, is not a problem if we want to guarantee these properties only for the actual sequence of queries that will be made to the component in its lifetime. Our dynamic model describes just such a case and, as we will soon show, imposes weaker necessary conditions upon the legality of aggregated components.

6.1. Theorem: Reflexivity and selective hiding of interfaces

If a component *O* aggregates a legal COM object *I*, then the sequence of QueryInterface calls to interfaces of *I* does not include a *reflexive call* to a delegating interface requesting an interface type that *O* does not expose. A reflexive call is a call to the QueryInterface function of an interface, requesting an IID of that interface. (Figure 4)

HiddenNotReflexive

$I, O : Component_d$

$\forall\ a : Interface; iidA : IID; r : QueryResult$
$\quad \mid\ iidA \in I.iids \setminus O.iids$
$\quad \wedge\ a \mapsto iidA \in IIDOfInterface$
$\quad \wedge\ a \neq I.firstInterface$
$\quad \bullet\ (a, iidA, r) \notin ran\ I.queries$

$\forall I, O : Component_{\hat{a}}$
$\quad \mid O \mapsto I \in Aggregates_{\hat{a}} \ \wedge \ I \in COMObjects_{\hat{a}}$
$\quad \bullet \ HiddenNotReflexive$

Proof:
By contradiction.
Let $O,I : Component_{\hat{a}}$ such that $O \mapsto I \in Aggregates_{\hat{a}}$ and $I \in COMObjects_{\hat{a}}$.
Let $a : Interface$; $iidA : IID$; $r : QueryResult$ such that $iidA \in I.iids \setminus O.iids$, $a \mapsto iidA \in IIDOfInterface$ and
$a \neq I.firstInterface$. Suppose that $(a, iidA, r) \in ran\ I.queries$ and let $n : \mathbb{N}$ such that $n \in dom\ I.queries$ and $I.queries(n) = (a, iidA, r)$.
Then n is in the domain of the delegation correspondence $g_{O,I} = Delegate(O, I)$ and by property (A12) of aggregation $O.queries(g_{O,I}\, n) = (O.pUnkOuter, iidA, r) \in ran\ O.queries$.
Since $iidA \notin O.iids$, r must be equal to *null* by definition of *interfaces* and *iids* sets of a component.
Thus $(a, iidA, null) \in ran\ I.queries$. On the other hand, since $I \in COMObjects_{\hat{a}}$ we have:
$I.queries \in LegalQuerySequences$ and by the reflexivity rule (L3) $(a, iidA, null) \notin ran\ I.queries$.
We have a contradiction. Our assumption that $(a, iidA, r) \in ran\ I.queries$ is untenable and so
$(a, iidA, r) \notin ran\ I.queries$.
□

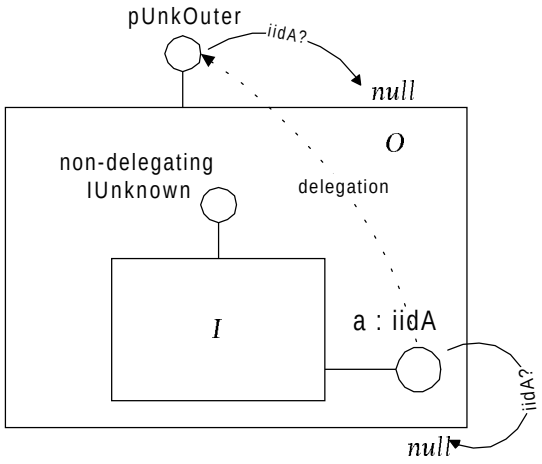


Figure 4: A reflexive query of a hidden interface

6.2. Theorem: Symmetry and selective hiding of interfaces

If a component *O* aggregates a legal COM object *I*, then the sequence of QueryInterface calls to interfaces of *I* does not include a subsequence of the following two calls. The first call is to QueryInterface of a delegating interface *a* of type *iidA*, not exposed by *O*, asking for an IID *iidB* exposed by *O*. This call succeeds and returns interface *b*. The second call, not necessarily immediately following the first one, is a call to QueryInterface of *b* asking for *iidA*. (Figure 5)

InsideOutNotSymmetric

$I, O : Component_{\hat{a}}$

$\forall\ a,b : Interface; iidA, iidB : IID; r : QueryResult; n_1, n_2 : dom\ I.queries$
$\mid iidA \in I.iids \setminus O.iids \land iidB \in O.iids$
$\land \{a \mapsto iidA, b \mapsto iidB\} \subseteq IIDOfInterface$
$\land n_2 > n_1$
$\bullet\ \neg\ (I.queries(n_1) = (a, iidB, b) \land I.queries(n_2) = (b, iidA, r))$

$\forall I, O : Component_{\hat{a}}$
$\mid O \mapsto I \in Aggregates_{\hat{a}} \land I \in COMObjects_{\hat{a}}$
$\bullet\ InsideOutNotSymmetric$

Proof:
By contradiction.
Let $I, O : Component_{\hat{a}}, a,b : Interface; iidA, iidB : IID; r : QueryResult; n_1, n_2 : dom\ I.queries$; such that $iidA \in I.iids \setminus O.iids$, $I \in COMObjects_{\hat{a}}$, $iidB \in O.iids$, $\{a \mapsto iidA, b \mapsto iidB\} \subseteq IIDOfInterface$ and $n_2 > n_1$. Assume further that $I.queries(n_1) = (a, iidB, b)$ and $I.queries(n_2) = (b, iidA, r)$.
First, we shall show that $b \neq I.firstInterface$. Assume the opposite, $b = I.firstInterface$. Then $iidB = IID\_IUnknown$. Since we assume that $iidB \in O.iids$ and $iidA \notin O.iids$, we conclude that $iidA \neq IID\_IUnknown$. Combining this with (A3) we get $a \neq I.firstInterface$. Thus $n_1 \in dom\ g$, where $g = Delegate(O, I)$ and $(O.pUnkOuter, iidB, b) \in ran\ O.queries$ and $b \in O.interfaces$. But by (A4) $I.firstInterface$ cannot be in $O.interfaces$. We have reached a contradiction and so $b \neq I.firstInterface$.
Second, suppose that $r \neq null$. Since $I.queries(n_2) = (b, iidA, r)$ and $I \in COMObjects_{\hat{a}}$, we have: $I.queries \in LegalQuerySequences$ and by (L1) $r \mapsto iidA \in IIDOfInterface$. $b \neq I.firstInterface$ implies that $n_2 \in dom\ g$ and $(O.pUnkOuter, iidA, r) \in ran\ O.interfaces$. Thus $r \in O.interfaces$ and $iidA \in O.iids$ which contradicts our assumption that $iidA \in I.iids \setminus O.iids$. We have shown by contradiction that $r = null$.
Finally, we have established that $I.queries$ contains the following subsequence:
$\langle\!\langle (a, iidB, b)\ (b, iidA, null) \rangle\!\rangle$. This apparently contradicts our premise that $I \in COMObjects_{\hat{a}}$ because it violates the symmetry axiom (L4).
Therefore our hypothesis that $I.queries(n_1) = (a, iidB, b)$ and $I.queries(n_2) = (b, iidA, r)$ is untenable and $InsideOutNotSymmetric$ holds for all aggregates $O \mapsto I$ such that $I \in COMObjects_{\hat{a}}$.
□

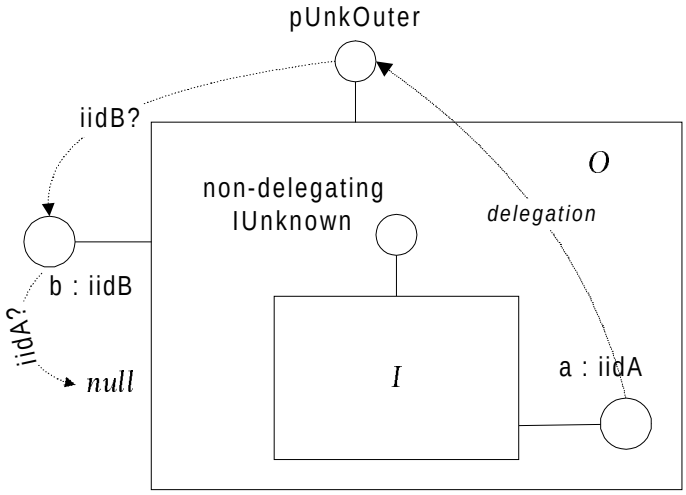


Figure 5: Symmetric pair of queries where the second IID is hidden

6.3. Theorem: Transitivity and the hidden non-delegating IUnknown

If a component *O* aggregates a legal COM object *I*, then the sequence of QueryInterface calls to interfaces of *I* does not include a subsequence of the following three calls, not necessarily next to one another. The first call is to QueryInterface of the hidden non-delegating interface of *I* asking for an IID of a delegating interface on *I*. This call succeeds and returns interface *a*. The second call is to QueryInterface of *a* asking for *iidB*, not exposed by *I*. This call also succeeds. The third call is again to QueryInterface of the hidden non-delegating interface of *I* now asking for *iidB*. (Figure 6)

$$
\begin{array}{l}
\textit{NonDelegatingNotTransitive} \\
I, O : \textit{Component}_d \\
\hline
\forall\, a,b : \textit{Interface}; \textit{iidA}, \textit{iidB} : \textit{IID}; r : \textit{QueryResult}; n_1, n_2, n_3 : \textit{dom}\ I.\textit{queries} \\
\quad \mid \textit{iidA} \in I.\textit{iids} \land \textit{iidB} \notin \textit{NativeIIDs}\ I \\
\quad \land \{a \mapsto \textit{iidA}, b \mapsto \textit{iidB}\} \subseteq \textit{IIDOfInterface} \\
\quad \land\ a \neq I.\textit{firstInterface} \\
\quad \land\ n_1 < n_2 \land\ n_2 < n_3 \\
\quad \bullet \neg\ (I.\textit{queries}(n_1) = (I.\textit{firstInterface}, \textit{iidA}, a) \\
\qquad \land\ I.\textit{queries}(n_2) = (a, \textit{iidB}, b) \\
\qquad \land\ I.\textit{queries}(n_3) = (I.\textit{firstInterface}, \textit{iidB}, r))
\end{array}
$$

$$
\begin{array}{l}
\forall I, O : \textit{Component}_d \\
\quad \mid O \mapsto I \in \textit{Aggregates}_d \land\ I \in \textit{COMObjects}_d \\
\quad \bullet\ \textit{NonDelegatingNotTransitive}
\end{array}
$$

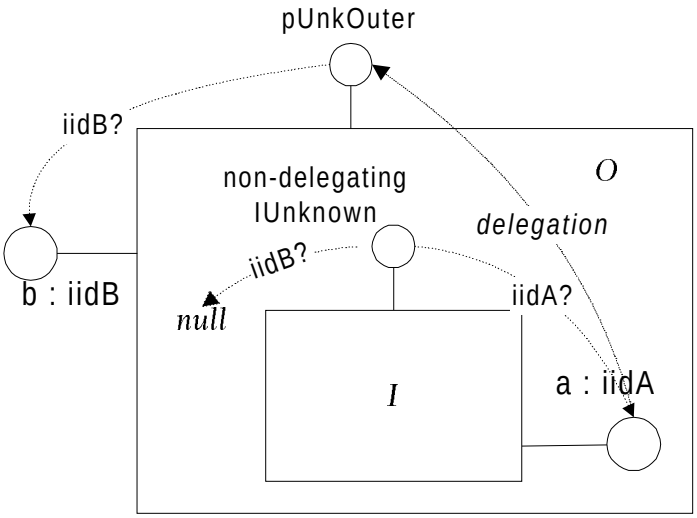


Figure 6: A sequence of two queries going from the non-delegating IUnknown of the inner outside to a native interface on the outer does not allow a shortcut.

Proof:
By contradiction.
Let $I, O : Component_d$, $a,b : Interface$; $iidA, iidB : IID$; $r : QueryResult$; $n_1, n_2, n_3 : dom\ I.queries$ such that
$I \in COMObjects_d$, $iidA \in I.iids$, $iidB \notin NativeIIDs(I)$, $\{a \mapsto iidA, b \mapsto iidB\} \subseteq IIDOfInterface$, $a \neq I.firstInterface$ and $n_1 < n_2 < n_3$. Suppose further that $I.queries(n_1) = (I.firstInterface, iidA, a)$, $I.queries(n_2)=(a, iidB, b)$ and $I.queries(n_3) = (I.firstInterface, iidB, r)$.
First, note that our assumptions imply that $b \neq null$ because $b \mapsto iidB \in IIDOfInterface$ and $null \notin dom\ IIDOfInterface$.
Second, suppose that $r \neq null$. One of our assumptions is that $I \in COMObjects_d$ or equivalently $I.queries \in LegalQuerySequences$. By axiom (L1) $r \mapsto iidB \in dom\ IIDOfInterface$. On the other hand, $I.queries(n_3) = (I.firstInterface, iidB, r)$. By definition of *NativeIIDs* function (A8) $iidB \in NativeIIDs(I)$. This contradicts the assumption that $iidB \notin NativeIIDs(I)$. Our assumption that $r \neq null$ is therefore untenable and we conclude that $r = null$.
We have shown that $I.queries$, contains a subsequence of three queries: $(I.firstInterface, iidA, a)$ $(a, iidB, b)$ and $(I.firstInterface, iidB, null)$ in this order, where $b \neq null$. This contradicts the transitivity axiom (L5) of legal query sequences. We finally conclude that our assumption about the existence of the query subsequence $I.queries(n_1) = (I.firstInterface, iidA, a)$, $I.queries(n_2)=(a, iidB, b)$ and $I.queries(n_3) = (I.firstInterface, iidB, r)$ is untenable. No such subsequence is present in $I.queries$.
□

7. A sufficient condition of legality of an aggregated component

In this section we will state and prove a condition on the outer and a given inner components of an aggregate that guarantees the legality of the inner component. In essence, this condition states that if the outer component is a legal COM object with a static set of interfaces, then every inner component will behave according to the rules of COM unless its clients (including other inner components) provoke it to

an "illegal" action by executing one of the three query sequences ruled out by the necessary conditions of legality.

7.1. Theorem: The legality of an inner component

Let a component *O* aggregate a component *I*.
I is a legal COM object in the dynamic interpretation of the rules of COM if all of the following conditions hold:

(R1) *O* is a legal COM object under the definition of legality given in the dynamic theory of COM
(R2) If *O* exposes an IID *d*, then any query in *O.queries* made to an interface of *O* and asking for *d* succeeds.
(R3) All successful queries to the inner non-delegating IUnknown in the inner query sequence return results of requested type.
(R4) The subsequence of queries to the inner non-delegating IUnknown in the inner query sequence is stable, so that if two queries from this subsequence request the same IID, then they both fail or both succeed.
(R5) The consequents of theorems 6.1, 6.2 and 6.3 (the necessary conditions of dynamic legality) hold for the pair $O \mapsto I$.

$$
\begin{array}{l}
\forall\ O, I : Component_{\hat{a}} \mid O \mapsto I \in Aggregates_{\hat{a}} \\
\quad \bullet\ O \in COMObjects_{\hat{a}} \qquad (R1) \\
\quad \wedge\ (\forall\ d : O.iids; q : ran\ I.queries \bullet request\ q = d \implies result\ q \neq null) \qquad (R2) \\
\quad \wedge\ (\forall\ q_1, q_2 : ran\ I.queries \mid \{receiver\ q_1, receiver\ q_2\} = \{I.firstInterface\} \\
\qquad \bullet\ result\ q_1 \neq null \implies result\ q_1 \mapsto request\ q_1 \in IIDOfInterface \\
\quad (R3) \\
\qquad \wedge\ result\ q_2 = null \iff result\ q_1 = null) \qquad (R4) \\
\quad \wedge\ HiddenNotReflexive \wedge InsideOutNotSymmetric \wedge NonDelegatingNotTransitive \qquad (R5) \\
\implies I \in COMObjects_{\hat{a}}
\end{array}
$$

Proof:
Let $O, I : Component_{\hat{a}} \mid O \mapsto I \in Aggregates_{\hat{a}}$. Suppose that $O \in COMObjects_{\hat{a}}$, (R2), (R3) and (R4) hold for *I.queries* and the predicates of *HiddenNotReflexive, InsideOutNotSymmetric* and *NonDelegatingNotTransitive* schemas hold for the pair $O \mapsto I$.
We shall show that $I.queries \in LegalQuerySequences$.
By definition of *LegalQuerySequences* set
$\forall\ s : seq\ Query$
$\bullet\ s \in LegalQuerySequences \iff CorrectResult \wedge Stable \wedge Symmetric \wedge Reflexive \wedge Transitive$

We need to show that the predicates (L1) - (L5) of schemas *CorrectResult, Stable, Symmetric, Reflexive* and *Transitive* hold for *I.queries*. We will accomplish this in the following five lemmas. For the rest of the proof we let g be the delegation correspondence between the query sequences of I and O, $g = Delegate(O, I)$ and p be the interface sharing correspondence between these sequences, $p = Share(O,I)$

Lemma 1: Successful queries of *I.queries* sequence return results of requested types.
$\forall\ q : ran\ I.queries \bullet result\ q \neq null \implies result\ q \mapsto request\ q \in IIDOfInterface$

Proof:
Let $q : Query \mid q \in ran\ I.queries \wedge result\ q \neq null$. There are two possible cases.
1) $receiver\ q = I.firstInterface$. Then $result\ q \mapsto request\ q \in IIDOfInterface$ by (R3).
2) $receiver\ q \neq I.firstInterface$. Let $n_q : dom\ I.queries \mid I.queries(n_q) = q$. Then by (A12) $O.queries(g\ n_q) = (O.pUnkOuter, request\ q, result\ q)$. By (R1) $O.queries \in LegalQuerySequences$. Thus (L1) holds for $O.queries$ and $result\ q \mapsto request\ q \in IIDOfInterface$.

□

Lemma 2: *I.queries* is stable

$\forall\, q_1, q_2 : ran\ I.queries \mid receiver\ q_1 = receiver\ q_2\ \wedge\ request\ q_1 = request\ q_2$
$\bullet\ result\ q_1 = null \iff result\ q_2 = null$

Proof:

Let $q_1, q_2 : Query \mid \{q_1, q_2\} \subseteq ran\ I.queries \wedge receiver\ q_1 = receiver\ q_2\ \wedge\ request\ q_1 = request\ q_2$.

Let $a : Interface \mid a = receiver\ q_1 = receiver\ q_2$. Again there are two cases to consider.

1) $a = I.firstInterface$. Then $result\ q_1 = null \iff result\ q_2 = null$ by (R4).

2) $a \neq I.firstInterface$. Let $n_1, n_2 : dom\ I.queries \mid I.queries(n_1) = q_1 \wedge I.queries(n_2) = q_2$. Then by (A12) $O.queries(g\ n_1) = (O.pUnkOuter, request\ q_1, result\ q_1)$,
$O.queries(g\ n_2) = (O.pUnkOuter, request\ q_2, result\ q_2)$.
By (R1) $O.queries \in LegalQuerySequences$. Thus (L2) holds for $O.queries$ and so
$result\ q_1 = null \iff result\ q_2 = null$.

□

In the rest of the proof we will demonstrate that, given the assumptions of the theorem, *I.queries* contains no query subsequences that violate the reflexivity, symmetry or transitivity axioms of QueryInterface as defined by our dynamic theory. We will split the set of possible subsequences of *I.queries* into classes and prove for each class that its elements do not violate those rules. A class is identified by a *k*-tuple of mutually exclusive sets of element indices, where every set is one of the following three: $dom\ p$, $(dom\ g \setminus dom\ p)$ and $(dom\ I.queries \setminus dom\ g)$. Note that the last class of the three contains the indices of all those queries in $I.queries$ whose receiver is $I.firstInterface$, the non-delegating hidden IUnknown. The classification is especially simple for a proof of reflexivity property because in this case *k = 1* and the subsequences of interest are singletons. To prove the symmetry property we have to consider 9 classes of two-element subsequences, several of which are easy to show to be empty under the assumptions of the theorem. In the proof of symmetry property we operate three-element sequences and the number of classes to consider threatens to reach 27. However it is not hard to demonstrate that most of those 27 classes are empty.

Lemma 3: *I.queries* is reflexive

$\forall\, q : ran\ I.queries \bullet receiver\ q \mapsto request\ q \in IIDOfInterface\ \Rightarrow\ result\ q \neq null$

Proof:

Let $q : Query; n : \mathbb{N} \mid\ I.queries(n) = q\ \wedge\ receiver\ q \mapsto request\ q \in IIDOfInterface$

Then one of the following is true.

1) $n \in dom\ p$.
Then by (A11) $O.queries(p\ n) = I.queries(n) = q$. By (R1) $O.queries \in LegalQuerySequences$ and so (L3) holds for it and $result\ q \neq null$.

2) $receiver\ q = I.firstInterface$ (or equivalently $n \in dom\ I.queries \setminus dom\ g$)
By (A3) $IIDOfInterface\ (\!|\ I.firstInterface\ |\!) = \{IID\_IUnknown\}$ and so $request\ q = IID\_IUnknown$.
By (C3) $result\ q \neq null$.

Since *HiddenNotReflexive* (theorem 4c) schema holds for the pair $O \mapsto I$ (R5), we need not consider the case $n \notin dom\ p \wedge receiver\ q \neq I.firstInterface$ (or equivalently $n \in dom\ g \setminus dom\ p$), because the predicate of *HiddenNotReflexive* implies that $q \notin ran\ I.queries$ that contradicts our assumptions.

This exhausts all possible cases for *n*. We have shown that $result\ q \neq null$ and (L3) holds for $I.queries$.

□

Lemma 4: *I.queries* is symmetric
$I.queries \in \{ s : seq\ Query \mid Symmetric \bullet s \}$

Proof:
Let $x, y : Interface$; $iidX, iidY : IID$; $r : QueryResult$; $n_1, n_2 : dom\ I.queries$ such that $n_1 < n_2$, $I.queries(n_1) = (x, iidY, y)$, $x \mapsto iidX \in IIDOfInterface$ and $I.queries(n_2) = (y, iidX, r)$.
It is sufficient to show that under these assumptions $r \neq null$.
Again, we will consider several cases.

1) Both x and y are exposed delegating interfaces. $\{n_1, n_2\} \subseteq dom\ p$.
   Then by (A11) $O.queries(p\ n_1) = I.queries(n_1) = (x, iidY, y)$,
   $O.queries(p\ n_2) = I.queries(n_2) = (y, iidX, r)$. Because by (R1) $O.queries \in LegalQuerySequences$, (L4) holds for it and so $r \neq null$.
2) If x is an exposed delegating interface, then y cannot be a hidden interface, because by (A11) $O.queries(p\ n_1) = I.queries(n_1) = (x, iidY, y)$ and so $y \in O.interfaces$ and $iidY \in O.iids$. Thus it cannot be the case that $n_1 \in dom\ p$ and $n_2 \notin dom\ p$. In particular, if $n_1 \in dom\ p$, then $y \neq I.firstInterface$.
3) If x is a delegating hidden interface of I ($n_1 \in dom\ g \setminus dom\ p$) then we only need to consider the case $n_2 \in dom\ p$ because $n_1 \in dom\ g$ along with (A12) imply that $y \in O.interfaces$. Suppose that $n_1 \in dom\ g \setminus dom\ p$ and $n_2 \in dom\ p$. Then if $iidX \notin O.iids$, we have a contradiction with *InsideOutNotSymmetric* schema, since $I.queries(n_1) = (x, iidY, y)$, $I.queries(n_2) = (y, iidX, r)$ and $n_1 < n_2$. If $iidX \in O.iids$, we combine (R2), (A11) and $n_2 \in dom\ p$ to conclude that $r \neq null$.
4) The only case left to consider is when x is the non-delegating hidden interface of I, $x = I.firstInterface$, or equivalently $n_1 \in dom\ I.queries \setminus dom\ g$. Then by (A3) $iidX = IID\_IUnknown$ and no matter what kind of interface y is, (C3) guarantees that the query $(y, iidX, r)$ succeeds and so $r \neq null$.

This exhausts all possible cases for n_1 and n_2. We have shown that $r \neq null$ and so (L4) holds for $I.queries$.
□

Lemma 5: *I.queries* is transitive
$I.queries \in \{ s : seq\ Query \mid Transitive \bullet s \}$

Proof:
Let $x,y,z : Interface$; $iidY,iidZ : IID$; $n_1, n_2, n_3 : dom\ I.queries$; $r : QueryResult$ such that $n_1 < n_2 < n_3$, $I.queries(n_1) = (x, iidY, y)$, $I.queries(n_2) = (y, iidZ, z)$, $z \neq null$ and $I.queries(n_3) = (x, iidZ, r)$.
We shall show that under these assumptions $r \neq null$. Just as we did in the proofs of lemmas 3 and 4 we will partition the set of indices $dom\ I.queries$ into classes and show that no matter what classes n_1, n_2 and n_3 fall into, provided the combination of classes does not contradict our assumptions, the result of the third query is not $null$.

1) Suppose that x is an exposed delegating interface, $n_1 \in dom\ p$. Then it is also the only possible class for y and z because by (A11) $O.queries(p\ n_1) = (x, iidX, y)$ and so $y \in O.interfaces$. Thus we do not need to consider the cases when x is exposed and y is hidden. If both of x and y are exposed delegating interfaces we have: $\{n_1, n_2, n_3\} \subseteq dom\ p$, $O.queries(p\ n_1) = (x, iidX, y)$, $O.queries(p\ n_2) = (y, iidX, z)$,
   $O.queries(p\ n_3) = (x, iidX, z)$ and by assumption (R1) $O.queires \in LegalQuerySequences$. Thus (L5) holds for $O.queries$ and $r \neq null$.

2) Now let x be a hidden delegating interface, $n_1 \in dom\ g \setminus dom\ p$. Then by definition of $g\ n_3 \in dom\ g$. By (A12) we have: $O(g\ n_1) = (O.pUnkOuter, iidY, y)$ and so again $y \in O.interfaces$ which means that y is an exposed delegating interface. By definition of the *Share* and *Delegate* mappings $n_2 \in dom\ p \subseteq dom\ g$. By (A13) and monotonicity of *g,* there exist $m_1, m_2, m_3 : dom\ O.queries$ such that $m_1 < m_2 < m_3$ and
$O.queries(m_1) = (O.pUnkOuter, iidY, y)$, $O.queries(m_2) = (y, iidZ, z)$,
$O.queries(m_3) = (O.pUnkOuter, iidZ, r)$. We can take, for example, $m_1 = g\ n_1$, $m_2 = p\ n_2$, $m_3 = g\ n_3$. Since $O.queries \in LegalQuerySequences$ and $(L5)$ holds for it, we conclude that $r \neq null$.
3) The only case left is x being the hidden non-delegating interface, $x = I.firstInterface$. This corresponds to
$\{n_1, n_3\} \subseteq dom\ I.queries \setminus dom\ g$. There are three subcases to consider.
   a) $n_2 \in dom\ g \wedge\ iidZ \in NativeIIDs\ I$.
   Then by definition of *NativeIIDs* function (A8)
   $\exists\ z_1 \in I.interfaces \setminus \{null\} \mid (I.firstInterface, iidZ, z_1) \in I.queries$. By lemma 2 $I.queries$ is stable and so the query $I.queries(n_3) = (I.firstInterface, iidZ, r)$ must also succeed. $r \neq null$.
   b) $n_2 \in dom\ g \wedge iidZ \notin NativeIIDs\ I$
   In this case we have a contradiction with the assumption that the *NonDelegatingNotTransitive* schema holds for the pair $O \mapsto I$. By that schema the sequence of three queries:
   $(I.firstInterface, iidY, y)$, $(y, iidZ, z)$, $z \neq null$ and $(I.firstInterface, iidZ, r)$ in this order cannot be a subsequence of $I.queries$ provided that $iidZ \notin NativeIIDs\ I$ and $y \neq I.firstInterface$.
   c) $n_2 \notin dom\ g$
   This implies that $y = I.firstInterface$ and since $z \neq null$, we have $iidZ \in NativeIIDs\ I$. Again, as in subcase (a) we use the definition of *NativeIIDs* along with lemma 2 to get $r \neq null$.

This exhausts all possible cases for n_1, n_2, n_3. We have shown that $r \neq null$ and so (L5) holds for $I.queries$.
□

Lemmas 1 through 5 demonstrate that predicates (L1) - (L5) of schemas *CorrectResult, Stable, Symmetric, Reflexive* and *Transitive* hold for *I.queries*. By definition of the $LegalQuerySequences$ set we have:
$I.queries \in LegalQuerySequences$, and finally $I \in COMObjects_d$.
□

7.2. Significance of the sufficient condition result

Armed with a sufficient condition of the legality of inner components, we now produce a set of guidelines for constructing legal COM aggregates from legal COM components. The designers of sophisticated COM-based systems with multiple aggregation and interface hiding can use these guidelines to build aggregates, whose inner and outer components are certain not to violate the interface negotiation rules.

1. The legality of an inner component cannot be guaranteed unless the interface negotiation mechanism of the outer component is implemented according to the rules of COM. Implementing the outer component so that it is legal under the static interpretation of the rules of COM will guarantee that conditions (R1) and (R2) hold for it.
2. Every inner component must implement its hidden non-delegating IUnknown interface correctly, so that its QueryInterface function returns interfaces of the requested types (R3) and satisfies the stability requirement (R4).
3. Every inner component must know which of its interfaces and interfaces of other inner components are of hidden types and not query such interfaces for their hidden types. This guarantees that *HiddenNotReflexive* schema holds for the aggregate. For example, it is reasonable for a mediator to

assume an interface through which it communicates with another inner component to be a hidden interface and not query it reflexively.

4. Inner components must not query hidden interfaces of themselves or other inner components for IIDs exposed by the outer component. If all of them follow this rule, then no sequence of calls prohibited by the *InsideOutNotSymmetric* schema can be executed and so *InsideOutNotSymmetric* holds for the aggregate.
5. Inner components should never query their hidden non-delegating IUnknown interfaces or such interfaces of other inner components. This guarantees that the aggregate satisfies the predicate of the *NonDelegatingNotTransitive* schema.

This set of guidelines appears to be loose enough to allow flexible connection architectures within encapsulated systems. At the same time Theorem 7.1 guarantees that as long as all the components of an aggregate follow these guidelines, they can assume that all components of the aggregate follow the rules of COM interface negotiation.

8. Conclusion

We have presented a new theory of COM aggregation and interface negotiation. The new "dynamic" theory differs fundamentally from earlier "static" theory [Sullivan et al. 1997] in one critical way: Whereas the "static" theory required components to be designed to act legally under any usage situation, the "dynamic" theory requires only that components act legally in each particular system execution. Thus, in the static theory, we formalized the legality of *QueryInterface* as a timeless relation; and in the dynamic theory, we formalize it in terms of sequences of calls to *QueryInterface*.

Under the static theory, we demonstrated that the selective hiding of the interfaces of inner (aggregated) components is incompatible with those inner components being legal under the rules of COM. Under the dynamic theory, by contrast, we showed that this incompatibility disappears, at least in principle. It becomes incumbent on clients of inner objects to follow additional rules that ensure that no violations of the rules of COM are ever manifested. Thus, we used the dynamic theory of COM to derive necessary and sufficient conditions for the legal aggregation of subsystems of interacting components. These additional rules should be of interest to designers who are considering advanced uses of COM aggregation. In addition, it might be appropriate for the designers of COM to consider integrating these rules into the specification of the COM standard, and into adopter-level documentation of the standard.

It is not entirely clear which interpretation people are using in practice. We suspect that people are generally confused about aggregation and, having heard rumors about difficulties with aggregation, that they simply avoid problems by not using except in ways that are well known not to be problematical. We have anecdotal support from several users of COM to support this suspicion, but no results from carefully administered studies.

Far from contradicting the basic conclusion of our earlier work, we take the results presented in this paper as convincing evidence of that conclusion. It is both necessary and profitable to apply formal methods to the analysis of innovative aspects of architectural standards and styles based on such standards. However, until industrial developers and adopters of standards are convinced of the need for and profitability of formal analysis, it is incumbent on the research community to earn its own way by proving the case for the innovative application of formal methods. We can continue to do that by solving problems that are at the heart of highly relevant industrial concerns. In this and in our earlier paper, we have shown that the light-weight use of formal methods "in-the-small" has significant potential to play a role in that endeavor.

Appendix A: COM interface negotiation rules

HRESULT IUnknown::QueryInterface(iid, ppv)

Return a pointer within this object instance that implements the indicated interface. Answer NULL if the receiver does not contain an implementation of the interface.

It is required that any query for the specific interface IUnknown[3] always returns the *same actual pointer value*, no matter through which interface derived from IUnknown it is called. This enables the following identity test algorithm to determine whether two pointers in fact point to the same object: call QueryInterface(IID_IUnknown, ...) on both and compare the results.

In contrast, queries for interfaces *other* than IUnknown are *not* required to return the same actual pointer value each time a QueryInterface returning one of them is called. This, among other things, enables sophisticated object implementors to free individual interfaces on their objects when they are not being used, recreating them on demand (reference counting is a per-interface notion, as is explained further below). This requirement is the basis for what is called *COM identity*.

It is required that the set of interfaces accessible on an object via QueryInterface be static, not dynamic, in the following precise sense.[4] Suppose we have a pointer to an interface

```
ISomeInterface * psome = (some function returning an ISomeInterface *);
```

where ISomeInterface derives from IUnknown. Suppose further that the following operation is attempted:

```
IOtherInterface * pother;
HRESULT   hr;
hr=psome->QueryInterface(IID_IOtherInterface, &pother);        //line 4
```

Then, the following must be true:

- If hr==S_OK, then if the QueryInterface in "line 4" is attempted a second time from the same psome pointer, then S_OK must be answered again. This is independent of whether or not pother->Release was called in the interim. In short, if you can get to a pointer once, you can get to it again.
- If hr==E_NOINTERFACE, then if the QueryInterface in line 4 is attempted a second time from the same psome pointer, then E_NOINTERFACE must be answered again. In short, if you didn't get it the first time, then you won't get it later.

Furthermore, *QueryInterface* must be reflexive, symmetric, and transitive with respect to the set of interfaces that are accessible. That is, given the above definitions, then we have the following:

Symmetric:	psome->QueryInterface(IID_ISomeInterface, ...) must succeed
Reflexive:	If in line 4, pother was successfully obtained, then pother->QueryInterface(IID_ISomeInterface, ...) must succeed.

Transitive:	If in line 4, pother was successfully obtained, and we do
	IYetAnother * pyet;
	pother->QueryInterface(IID_IYetAnother, &pyet); //Line 7
	and pyet is successfully obtained in line 7, then
	pyet->QueryInterface(IID_ISomeInterface, ...)
	must succeed.

Here, "must succeed" means "must succeed barring catastrophic failures." As was mentioned above, it is specifically *not* the case that two QueryInterface calls on the same pointer asking for the same interface must succeed and return exactly the same *pointer value* (except in the IUnknown case as described previously).

REFERENCES

1. Brockschmidt, K., *Inside OLE*, Microsoft Press, 1996.
2. Brockschmidt, K, *What OLE is Really About*, Microsoft Corporation, 1997, available on the world-wide web at *http://www.microsoft.com/oledev/aboutole.htm*
3. Component Object Model Specification, Microsoft Developer Network Library, Microsoft Corporation, 1996 (especially sections 3.3.1 and 6.6.2).

[3] That is, a QueryInterface invocation where iid is 00000000-0000-0000-C000-000000000046.

[4] While this set of rules may seem surprising to some, they are needed in order that remote access to interface pointers can be provided with a reasonable degree of efficiency (without this, interface pointers could not be cached on a remote machine). Further, as QueryInterface forms the fundamental architectural basis by which clients reason about the capabilities of an object with which they have come in contact, stability is needed to make any sort of reasonable reasoning and capability discovery possible.

4. Kindel, C., "The rules of the component object model," Microsoft Developer Network Library, Technical Articles: Windows: OLE COM, Microsoft Corporation, October 20, 1995.
5. McIlroy, M. D., "Mass Produced Software Components", *Software Engineering*, p. 138-150, NATO Science Committee, Jan. 1969.
6. Rogerson, D., *Inside COM*, Microsoft Press, 1997
7. Sullivan, K.J., "Mediators: Easing the Design and Evolution of Integrated Systems," Ph.D. Dissertation, University of Washington Department of Computer Science, August 1994.
8. Sullivan, K.J. and D. Notkin, "Reconciling Environment Integration and Evolution," ACM *Transactions on Software Engineering and Methodology* vol. 1, no. 3, July 1992.
9. Sullivan, K., J. Socha and M. Marchukov, "Using Formal Methods to Reason about Architectural Standards," to appear in Proc. ICSE 97, Boston, Massachusetts, May 1997.
10. Williams, A., *Developing ActiveX Web Controls*, Coriolis Group, Inc, 1996