Compositionality Problems in Microsoft's COM

Kevin J. Sullivan and Mark Marchukov
University of Virginia Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, VA 22903, USA
Tel. (804) 982-2206
sullivan@cs.virginia.edu
http://www.cs.virginia.edu/~sullivan

I. Component Standards

Creating software systems by composing components that were developed by designers whose choices are coordinated only loosely, e.g., through the marketplace, promises improvements in software productivity and quality. Any component-based approach depends on a *component standard*. A standard comprises a set of design rules that when followed ensure the essential preconditions for independently designed components to work together in certain required ways. Conformance to common design rules takes the place of direct communication and agreement among designers as a coordination mechanism [1]. Shared rules are essential to fostering the decentralization of design activities that will be a hallmark of successful component-based software development (CBSD). We note that, in addition to design rules, standards are often supplemented by infrastructure capabilities (or specifications thereof with implementations left to the market). We are concerned with the design and engineering of the shared design rules.

To succeed, CBSD will require standards that are technically adequate and ratified by industry: at least in fact if not by explicit agreement. A successful standard will address a comprehensive range of technical design and evolution issues, otherwise it will not survive as a basis for CBSD in practice. The successful standard is most likely to be developed by industry and for industry.

II. A Role for Researchers

Researchers can contribute to the emergence of a successful component industry by exploring the science and engineering of the shared design rules that such standards promulgate. The standards on which an industry is to be based are critical frameworks whose engineering foundations should be set on a sound basis—at least in areas where innovative aspects raise questions about integrity. Our research is showing the need for such foundations and is contributing to their establishment.

We have taken the position that as researchers our best bet for interesting and useful results is to study and to abstract from existing standards that have already succeeded, rather than designing and analyzing imaginary ones. As mentioned above, real standards simply must address many critical issues. We are working on the assumption that it is possible to abstract from successful standards in order to understand, analyze and even synthesize new or modified CBSD standards, but that imagined standards will overlook critical issues that researchers are unlikely to encounter in the lab.

III. A Conflict in COM

Our approach has produced interesting results to date [12,14]. We have shown that there are subtle but architecturally critical difficulties in the foundations of the most successful standard, Microsoft's COM [2,3,4,5,6,7,15]. These problems prevent satisfactory application of Sullivan's proven compositional, mediator-based design approach [9,10,11,13] to achieve what would be a natural, efficient, hierarchical style with COM objects as components. The problems result from subtle interactions between two mechanisms at the heart of COM. These mechanisms are also general CBSD mechanisms, with potential importance beyond COM. The mechanisms at issue are dynamic interface negotiation (*QueryInterface*) and efficient wrapping (COM *aggregation*).

The design rules for these mechanisms are subtle and complex and the interactions are confusing. It is not possible in this short space to explain them in detail, nor to present our formalizations of these rules and the proofs of theorems that characterize the problems precisely. Rather, we will have to be content to sketch the basic problem.

Substantial detailed knowledge of the COM mechanisms is needed to really understand what the problem is. We can sketch it. First, *QueryInterface (QI)* allows a client with a pointer to any interface on a COM component to ask for another interface on that same component. In figure 1a, the Spellchecker component has two interfaces called *ISpell* and *ISave*. Each interface has a unique interface identifier (*IID*) and supports a *QI* operation (*QI*). *QI* takes an *IID* as a parameter and returns the interface having that *IID* on the same object if that object supports an interface of that type; otherwise *QI* returns *NULL.*

Second, aggregation is a mechanism permitting hierarchical composition of components with an important optimization in the situation where an *outer* object (such as the Spellchecker in Figure 1a) has an interface for which an acceptable implementation is already provided by an *inner* component (such as the Dictionary in Figure 1a). By passing off the *inner's* interface as its own to its clients, the *outer* avoids the memory overhead of representing a virtual function table for the interface, and the runtime overhead of passing procedure calls across the interface boundary. In Figure 1a, the Spellchecker represents that it has an interface *ISave*, which a client could use to save the Spellchecker's state. That state is contained within the dictionary, so Spellchecker just "hands out" the *ISave* interface obtained from the Dictionary, representing as its own interface. That is the essence of COM aggregation.

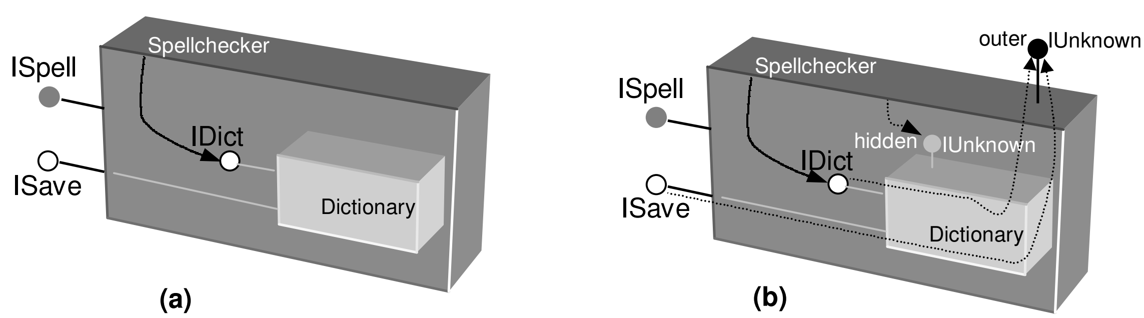
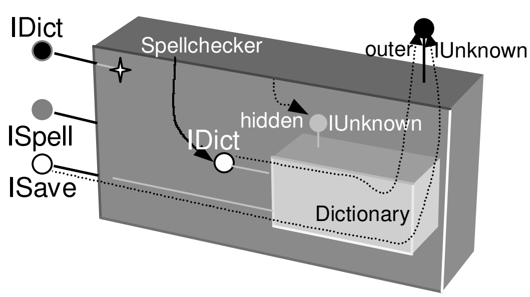


Figure 1. (a) Without cooperation between the outer and inner components, interfaces implemented by the outer (*ISpell*) are unreachable from interfaces of the inner (*ISave*). (b) Dictionary is correctly aggregated (dashed lines denote the necessary forwarding of QI calls).

The COM rules for aggregation have to work with those for *QI*. In particular, the rules of aggregation must (and do) guarantee that "on the outside" any aggregate appears to be a legal COM component. By "legal" we mean that *QI* on exposed interfaces works according to the rules of COM, despite the fact that some interfaces are implemented by inner components. Among other things, that means that if *ISave* on Spellchecker is queried for *ISpell* the query must succeed and return Spellchecker's *ISpell* interface.

To ensure that this is the case, COM stipulates that an aggregated component must work together with its aggregator, in two key ways. First, calls to *QI* on (almost) all interfaces of the aggregated component must be forwarded to an interface of the outer component to be handled. (Figure 1b.) Thus, when *ISave* is queried for *ISpell,* the query will be handled by an interface implemented by the outer, and that interface will be able to return the *ISpell* interface as required. (When created, the inner is given a pointer to an interface on the outer, for forwarding purposes.) The exception to this rule is that the inner component must have one interface by which the outer can obtain the interfaces on the inner that are to be exposed by the outer to its clients. This interface is called the hidden *inner IUnknown*. It's implemented by the inner and not exposed by the outer.

Here's the problem with these design rules. Because Spellchecker wants to prevent clients from performing some Dictionary operations, Spellchecker doesn't expose Dictionary's *IDict* interface. Thus, when a client calls *QI* on any interface on Spellchecker (including *ISave*) and asks for *IDict*, the call must return *NULL*. Yet, the *ISave* interface on Spellchecker is exposed by Dictionary, too. Thus by the rules of *QI*, when a client calls *QI* on any interface of Dictionary, including *ISave,* asking for *IDict*, the call should succeed and return an *IDict* interface. Thus, for the inner to behave legally, the outer must expose an *IDict* interface. In a bizarre twist, it needn't be the *IDict* interface of Dictionary: Any legal *IDict* interface will do, as illustrated in this figure.



The formal structure is intricate, but in essence the problem arises because two components share the *ISave* interface (not merely distinct interfaces of the same type *ISave*), and sharing interfaces does not work well with *QI*. The *QI* rules entitle any client with any interface on a component to obtain interfaces of all types exposed by that component. Thus, we conclude (and have proven) that a holder of a shared interface on two legal COM objects has to be able to get interfaces of all types collectively exposed by the sharers. The contrapositive is that selective hiding makes inner objects illegal with respect to the COM rules of QI: i.e., they are no longer COM objects.

IV. Compositionality Compromised

A composition mechanism is not general unless it allows a system of communicating components to be wrapped up in the form of a larger component. The COM specification does not prohibit the use of aggregation to implement this style of composition. However, the conflict between the rules of aggregation and *QI* leads to serious problems that cannot be ignored.

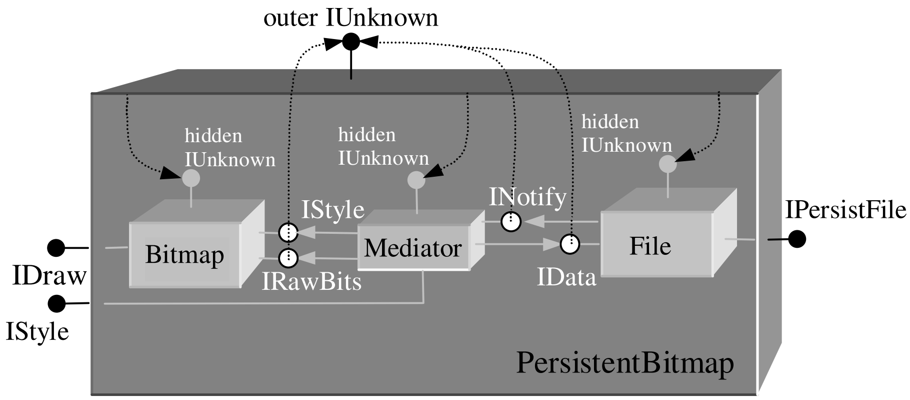


Figure 1. An aggregated system of communicating components

Suppose a multimedia designer wants to build a persistent bitmap component in a compositional multimedia authoring environment by integrating a Bitmap and a File component using Sullivan's approach to component composition [7,10,11] (Figure 1). A persistent bitmap is a bitmap that can be loaded from and saved to a file and displayed in several styles at a specified location on the screen. Since redrawing bitmaps must be fast, Bitmap is aggregated and its *IDraw* interface is exposed to the aggregate's clients. Clients use *IPersistFile* of File to request loading or saving the bitmap. Upon request or when the file changes on disk, File notifies the mediator which in turn sends or receives the raw bitmap data to/from File through its *IData* interface, possibly using a compression algorithm. The mediator uses its *IStyle* interface to correctly render the raw bitmap data from the file. It uses the *IStyle* interface of Bitmap to set its drawing style accordingly.

PersistentBitmap doesn't expose *IRawBits, INotify* or *IData* interfaces. Although it exposes *IStyle*, it is not the same interface as the one exposed by Bitmap. It turns out that under these realistic assumptions Mediator and File can no longer use COM's rules of interface negotiation to locate services on aggregated components that they use. Suppose that Mediator first gets the *IRawBits* interface on Bitmap (somehow). It also needs an *IStyle* interface on that component, so it calls the *QI* function of *IRawBits* to obtain it. But because Bitmap is aggregated, this *QI* function will forward all calls to the *QI* of the outer IUnknown (a dashed line). That *QI* function has no way to distinguish calls from clients inside and outside of the aggregate, so it returns the *IStyle* interface of the aggregate. This is an interface on Mediator itself. Thus, trying to obtain an interface on Bitmap through the standard COM interface negotiation mechanism gets Mediator an interface on itself! Moreover, if Mediator queries the interface (which it assumes to be an interface of Bitmap) for *IRawBits*, the query will fail since *IRawBits* is a hidden type. Another result shows that within the aggregate, COM object identity cannot be distinguished. These are disturbing hidden consequences of the design of COM.

C.6. Research Direction

These results have been fully formalized in Z [8]. It was only by formalizing this standard that we were able to clarify its design. Our work has led to the interesting insight that a (highly) non-standard interpretation of the *existing* informal COM specification leads to a partial resolution of some of the problems [14]. We read the rules for *QI* not as constraining component designs but only as prohibiting *QI* calls that would reveal a component to be unable to follow the *QI* rules. We have formalized this structure. It is harder to reason about, but we have been able to derive necessary and sufficient conditions for the mediator compositional architecture to remain legal.

1. Baldwin, C.Y. and K.B. Clark, "The Microstructure of Designs," Working Paper 98-031, Harvard Business School, 1997.
2. Grimes, R., *Professional DCOM Programming*, Wrox Press Inc, June 1997
3. Goswell, C., "The COM Programmer's Cookbook," Microsoft Office Product Unit, Spring 1995, revised September 13, 1995, Available on the World-Wide Web at the time of submission of this paper through *http://www.microsoft.com/oledev/olecom/com_co.htm*
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. Microsoft Corporation, *The Component Object Model Specification*, Draft Version 0.9, October 24, 1995 Microsoft Corporation and Digital Equipment Corporation. Available on the World-Wide Web at http://www.microsoft.com/oledev/olecom/title.htm
6. Microsoft, http://www.microsoft.com/corpinfo/press/1997/Sept97/COMplspr.htm
7. Rogerson, D., *Inside COM,* Microsoft Press, 1997
8. Spivey, M., *The Z Notation: A Reference Manual,* Prentice-Hall, 1992
9. Sullivan, K.J., "Mediators: Easing the Design and Evolution of Integrated Systems," Ph.D. Dissertation, University of Washington Department of Computer Science, August 1994.
10. Sullivan, K.J. and D. Notkin, "Reconciling Environment Integration and Evolution," ACM *Transactions on Software Engineering and Methodology* vol. 1, no. 3, July 1992.
11. Sullivan, K.J., I.J. Kalet and D. Notkin, "Mediators in a Radiation Treatment Planning System," IEEE *Transactions on Software Engineering*, vol. 22, no. 8, August 1996, pp. 563-579.
12. Sullivan, K.J., J.C. Socha and M. Marchukov, "Using Formal Methods to Reason about Architectural Standards," *Proceedings of the 1997 International Conference on Software Engineering,* Boston, May 17--24, 1997, pp. 503—513 (revised and extended version to be submitted to *IEEE Transactions on Software Engineering* is forthcoming).
13. Sullivan, K.J., Kalet, I.J., and Notkin, D., "Evaluating The Mediator Method: Prism as a Case Study," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, August 1996, pp. 563--579.
14. Sullivan, K.J. and M. Marchukov, "Dynamic Interface Negotiation and Efficient Reuse in Microsoft's COM," University of Virginia Department of Computer Science Technical Report 97-14, May 1997.
15. Williams, A., *Developing ActiveX Web Controls*, Coriolis Group, 1996