Shaky Foundations? Using Formal Methods to Reason about Architectural Standards

Kevin Sullivan
John Socha

Computer Science Report No. CS-96-18
November 1996

Shaky Foundations? Using Formal Methods to Reason about Architectural Standards

Kevin Sullivan and John Socha
University of Virginia Computer Science Department
Charlottesville, VA 22903
tel. (804) 982-2206 e-mail: sullivan@virginia.edu
and Socha Computing, Inc., Kirkland, WA, (206) 822-9300 jsocha@socha.com

ABSTRACT
We present a case study in which we applied formal methods in evaluating a novel architectural style that combined mediators and Microsoft's Component Object Model (COM). To verify conformance with the COM specification, we built a formal model of key aspects of COM. That led to an effort to understand and *validate* key properties of COM. We averted an architectural disaster by discovering that our proposed architecture was illegal. The problem was in architecturally important but previously overlooked subtleties in the design of the COM standard. Such widely used ***architectural standards*** are critical infrastructure systems. Formal methods have a significant role to play in practical validation and verification efforts.

Keywords
Software architecture, mediators, COM, formal methods

INTRODUCTION

The critical architectural designs of a vast number of important systems will depend on widely used **architectural standards** such as Microsoft's Component Object Model (COM) [1] and the Object Management Group's Common Object Request Broker [7]. Such standards should therefore be treated as critical infrastructure systems. Standards that provide a basis for application interoperation are especially critical in that design errors based on improper or unanticipated use of such standards may go unnoticed until interactions among fully deployed applications finally reveal "killer" design faults.

Any lack of clear guidance in the proper application of an architectural standard puts users at risk of making errors in the critical, early architectural design stage. To the extent practicable, users should be relieved of the burden of having to reason about subtle but important aspects of such standards. Key standards therefore should be carefully *validated* and the results of validation efforts (e.g., tools or theorems) should be made available to users. These kinds of products of *validation efforts* can be especially important in helping users to *verify* the legality of proposed uses of an architectural standard.

This paper presents a case study in which *formal methods* [8,12,16] played as key role in validating the specifications of architectural standards, and in verifying conformance of designs to such standards. We report on a modest application of *formal methods* that averted a costly commitment to an exciting but flawed *architectural style* based on mediators [17,18,19] and the COM mechanism of *aggregation.* At stake was the integrity of Socha Computing, Inc.'s multimedia authoring system (called Herman). Our initial attempt to verify that our proposed style used COM legally led us to formalize aspects of COM that we found relevant but subtle and hard to reason about. Our results characterize architecturally crucial but previously unexplained or misunderstood properties of COM.

Section 2 summarizes COM and architectural style we proposed. Section 3 summarizes our results. Section 4 formalizes key aspects of COM and proves our main theorem. Because of length restrictions, our formal model in Z [16] will be available separately. Section 5 presents our results in detail. Section 6 evaluates this work. Section 7 discusses related work. Section 8 concludes.

A NOVEL ARCHITECTURAL STYLE

COM is the foundation of Microsoft's component-based software architecture. It is the lowest level of OLE [4], and a foundation for systems of major importance to large segments of industry, government and military. COM defines the form of components and several composition mechanisms. For our purposes, components have two key features. First, each one can expose multiple interfaces, each of which groups operations for some service, and each of which is of a type indicated by a unique *interface identifier (IID).* See Fig. 1. Components interact through pointers to interfaces (arrows and circles in Fig. 1). Second, each interface implements a *QueryInterface* operation that a client with a pointer to one interface uses to get pointers to other interfaces on the same object. *QueryInterface* takes an *IID* and returns a pointer to the desired kind of interface, or null if there isn't one.

COM's composition mechanisms are mostly traditional: explicit and implicit invocation, containment, delegation. The exception is *aggregation.* In aggregation, one or more interfaces of a component are obtained from hidden, contained components. See Fig. 2. *QueryInterface*

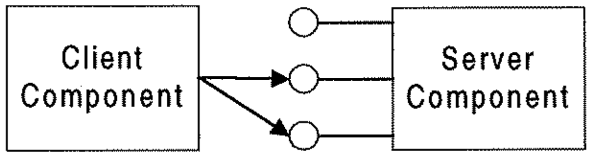

Fig. 1. Components expose multiple interfaces that are accessed through pointers.

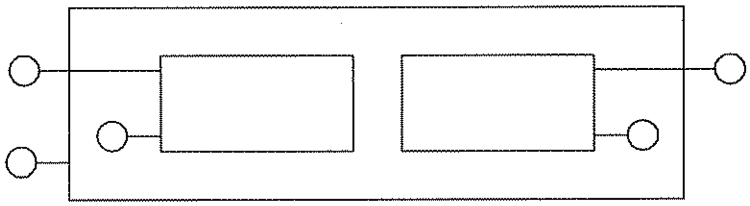

Fig. 2. The *aggregation* approach to composition in COM

operations on outer objects can return pointers to interfaces on inner objects. One advantage is to reduce the overhead from chained calls (as in delegation). In Fig. 2, two of the three interfaces exported by an outer component are actually interfaces on a hidden, inner (aggregated) components.

We intended to combine COM components, mediators, and aggregation to define a highly compositional architectural style in which we would aggregate a subsystem whose basic components were integrated by separate mediator components in order to build a new component that could then be used as a basic component at the next level. Fig. 3 illustrates the idea. The mediator (round-cornered rectangle) integrates the basic components (rectangles) into a subsystem which is aggregated to make a larger component. This approach was meant to support the abstraction of subsystems by *selectively hiding and exposing of aggregated interfaces* (circles). By also supporting the selection of variant components and mediators, such an aggregate would define a reference architecture [2,3] for a family of multimedia subsystems or applications—an interesting idea that we can't pursue further in this paper.

SUMMARY OF RESULTS

Our concern for the legality of the proposed architectural style led us to attempt a careful verification. We found it hard to reason about some non-obvious aspects of COM. Our need to understand the subtleties led us to use concepts from discrete mathematics to build an abstract model of the key structures at issue.

To cut to the chase, our modest use of formal methods showed that our architectural style violated the COM standard, and, moreover, that many seemingly natural designs do, too. In particular, we disproved the putative theorem that *aggregation in COM supports abstraction through selective hiding of interfaces on aggregated objects.* COM appears not to support such an abstraction mechanism.

A corollary is that our architectural style does not work. In Fig. 3, the absence of *IInterfaceB* and *IInterfaceC* interfaces on the *CFileDib* component turns out to be illegal. A related result is that an aggregated (inner) component generally cannot be treated as a normal component by another aggregated component such as a mediator.

Our difficulties resulted from a subtle interaction of *QueryInterface* and *aggregation.* The COM specification acknowledges a complication here, and prescribes a solution. Because calls to *QueryInterface* must return pointers to interfaces on the *same* component, *QueryInterface* operations of aggregated interfaces exposed by the aggregator must only return interfaces on the aggregator. COM therefore prescribes that *QueryInterface* operation of aggregated interfaces be delegated to *QueryInterface* operations of the aggregator—which are assumed to return pointers only to aggregator interfaces. On the other hand, there seems to be nothing in the COM standard or related authoritative documents even hinting at our architecturally critical "theorems of COM."

FORMAL MODEL AND REASONING

In this section we present (an English translation of) a formal model of the basic COM concepts of interface, component, and aggregation using basic concepts from discrete mathematics (set theory and first order predicate logic). We then use this model to draw our conclusions about COM.

Interfaces

Each COM *interface instance* (or just *interface*) belongs to one component and satisfies an *interface specification* (or just *specification*). Like a C++ abstract class, a specification

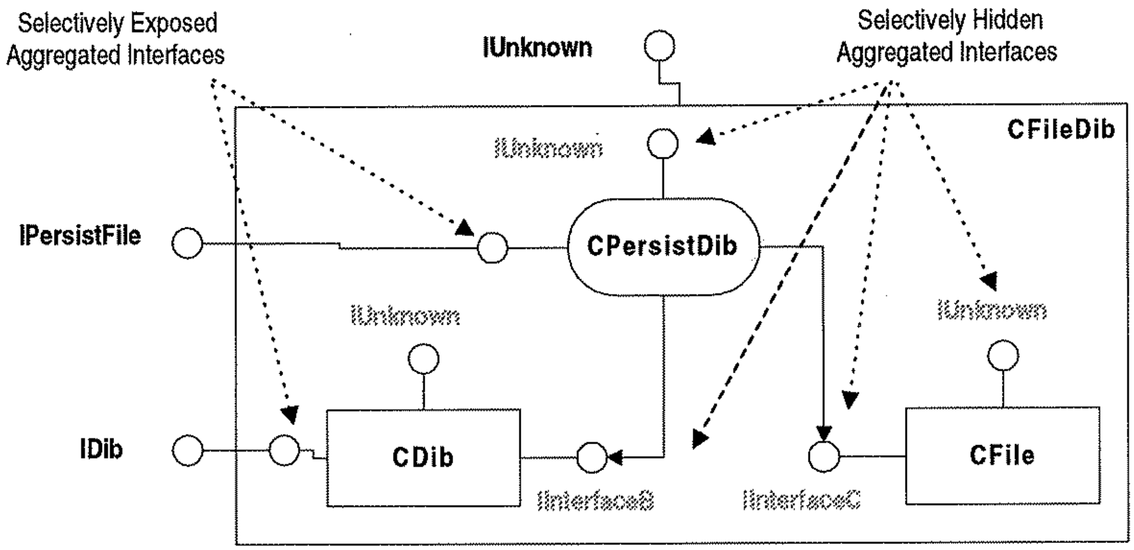

Fig. 3. Subsystem of multimedia authoring tool using proposed architectural style

declares the operations on interfaces of that kind. Each specification is identified by a unique *interface identifier* called an *IID*. Components bind implementations to the operations of their interfaces, and are accessed solely through *interface pointers*—pointers to their interfaces.

There is a special specification called *IUnknown*, with an *IID* called *IID_IUnknown*. The first operation declared by *IUnknown* is *QueryInterface*, taking an *IID* as an in parameter and returning an interface. Each specification inherits from and is polymorphic with *IUnknown*. Therefore a pointer to any interface has type pointer to *IUnknown;* and *QueryInterface* is the first operation of every interface. *QueryInterface* implementations allow a client with one interface pointer to ask for another on the same component. *QueryInterface* returns a null pointer if the component doesn't expose an interface with the given *IID*.

Our model abstracts from all operations on an interface except *QueryInterface*. We model the implementation of each *QueryInterface* operation of each interface on a given component as a partial injection from *IIDs* to interface pointers for interfaces on the same component. The COM standard requires that an interface pointer of type *IID_IUnknown* be available through every implementation of *QueryInterface*. We model this requirement with the predicate that *IID_IUnknown* is in the domain of every *QueryInterface* mapping.

We model the association of an interface specification with a unique *IID* as a bijection. We model the polymorphism of interfaces with *IUnknown* abstractly, with bimorphism in place of polymorphism. (This abstraction models common practice quite closely.) Specifically, we define *InterfaceSpecOf* as a relation mapping interfaces to the interface specifications that they satisfy; we add a predicate stating that every interface maps to *IUnknown* and to at most one other specification. More formally, restricting the range of the relation to exclude *IUnknown* yields a partial injection. We use relational composition to define a relation, *IIDOfInterface,* relating each interface to the *IIDs* of the specifications it satisfies. A simple result is that *IIDOfInterface* maps each interface at least to *IID_IUnknown.*

Components

For our purposes, a legal COM component is an object that supports multiple interfaces, pointers to which are provided to clients through *QueryInterface* operations (and possibly by other means). All interfaces on any component must provide legal *QueryInterface* implementations (mappings); and every component must expose a distinguished interface of type *IID_IUnknown*. This distinguished *IUnknown* interface (as it is called) defines an object's identity: Given any two interface pointers, calling *QueryInterface* with *IID_IUnknown* on each, possibly at different times, returns the same pointer values if and only if the interface pointers referred to interfaces on the same object.

The subtle issue is the conditions for legal *QueryInterface* implementations. COM requires *QueryInterface* operations on a component to be reflexive, symmetric and transitive. Contrary to intuition (and to what we strongly believe to be common understanding) these conditions say nothing about being able to get from one *interface* to another, but only about getting from one *type* of interface to another.

Moreover, that different interfaces can be reached given the same *IID* is not represented as a key property, yet it is in essential to the very consistency of the COM standard. The COM specification explains only that, among other things, the ability to return different interface pointers for the same *IID* allows a smart component to manage memory associated with its interfaces. This is a case where the formalization revealed a major subtlety that is not explicit in the standard. To understand the subtlety requires details. The details are in

English and pseudo-code in the COM specification. We model them as predicates over the relations modeling the implementations of *QueryInterface* operations.

Reflexivity means that if you have a pointer to an interface of type *IID_Some,* then calling *QueryInterface* through that pointer with the same *IID* must return a pointer to an interface of that same type on the same object. It is not required that the returned pointer be equal to the one through which the call was made. In more formal terms, if *IID_Some* is in the domain of the *QueryInterface* mapping for a given interface and if p is the image of *IID_Some* under that mapping, then *IID_Some* must be in the range of *QueryInterface* on the interface pointed to by p (although the image of *IID_Some* under that mapping need not be p).

Symmetry means that if you have a pointer to an interface of type *IID_Some* on an object, and if calling *QueryInterface* with *IID_Other* returns a non-null pointer, then calling *QueryInterface* through that pointer with *IID_Some* must also return a non-null pointer. Informally, if you can get from "here to there," you can get from "there to here." The subtlety, again, is that, in this context, "here" and "there" refer only to interface *types*. The formal statement is similar in form to the one described in the previous paragraph.

Finally, transitivity means, informally, that if *QueryInterface* can get you from "here to there" and "there to somewhere else," it can get you "from somewhere else back here." The pseudo-code and formal statements are similar to those in the preceding paragraphs.

We can now state and prove (with modest rigor in this paper) the following lemma: *QueryInterface must be a total algebra on the set of interface types exposed by a component.* Informally, you can get from anywhere to anywhere in one *QueryInterface* step. This simplifying result follows from reflexivity, symmetry and transitivity combined with the reachability of *IID_IUnknown* from any interface. This statement of the lemma is not new: Goswell asserts, "The set of interface IDs accessible via **QueryInterface** is the same for every interface.... [10]" However, rigorous justification of Goswell's paraphrase of the standard does appear to be new.

Aggregation

In this section, we model and reason about COM aggregation. We introduce component hierarchy and the idea of delegating and non-delegating inner interfaces of type *IID_IUnknown*. A key lemma shows that the availability of an interface of an arbitrary type *IID_ISome* on an aggregated (inner) component implies the presence of an interface of that same type on the aggregator. To build confidence in the lemma, we explain why it is that the common aggregation-based design idiom does not violate the COM standard. The bridge to our main theorem is the definition of *abstraction* as the property of an inner aggregated objecting having an interface of a type not present on the aggregator (i.e., not visible to its clients). This definition is a completely natural one based on the concept of information hiding [13]. The proof of the central theorem, *that aggregation does not support abstraction through the selective hiding of the types of aggregated interfaces* follows from the lemma by contradiction.

First, we model an *aggregate* as a collection of components. We impose a hierarchical relation that organizes them into a tree, whose root is called the *outer* component, and all of whose other components are called *inner* components.

Next we model key COM rules governing the interfaces on inner components and the implementations of their *QueryInterface* operations. First, on each inner object we require a distinguished interface of type *IID_IUnknown*, the implementation of whose *QueryInterface* operation maps *IIDs* of interfaces on the inner object to interface pointers on the inner object. The outer uses this interface to obtain pointers to inner interfaces exposed to outer clients. Second, for each inner object we restrict the implementations of *QueryInterface* operations on all interfaces other than the distinguished *IUnknown* to be equal to the *QueryInterface* implementation on the outer object's *IID_IUnknown* interface. These are the so-called *delegating* implementations of the *IUnknown* operations—the specification calls them *"delegating IUnknowns."* That is our simple, formal model of aggregation.

Next we consider the legality of what we understand to be the common use of aggregation—what we call the "OLE control and container" design idiom. In this idiom, an outer object serves as a transparent wrapper for one or more aggregated objects (frequently OLE controls). The outer provides access to all of the interfaces of the inner objects (except for their non-delegating *IUnknowns*) but augments those interfaces with additional information, e.g., by exporting an additional interface. In the case of OLE controls (such as menus or buttons), the outer wrapper enables the inner object to be managed by a *container* object, such as a Visual Basic form [21]. In this case, the *outer* wrapper object augments the inner with placement-on-form information.

One might object that this use of aggregation is illegal because, although there is a way to get from the non-delegating inner *IUnknown* to another inner interface, and from there (via that interface's delegating *IUnknown*) to an interface on the outer object, there is no way to get from the outer interface back to the non-delegating inner *IUnknown.* The problem with this argument is that the rules don't govern reachability of *interfaces* via *QueryInterface* calls, but only reachability of the *types* of interfaces. There is nothing wrong with the scenario because *QueryInterface* takes you from an interface of type *IID_IUnknown* to an interface of some other type *ISome*, and from there back to one of type *IID_IUnknown.*

We prove, however, that if *inner* is an inner component with an interface of type *IID_ISome* reachable via *QueryInterface* on the inner non-delegating *IUnknown,* then the outer component must expose an interface of type *IID_ISome.* To our knowledge, both the statement and proof of this architecturally crucial "lemma of COM" are new. The proof is by contradiction. Assume that the outer does not provide an interface of type *IID_ISome*. By hypothesis we can get from the inner *IUnknown* to the (inner) interface of type *IID_ISome*. The *QueryInterface* implementation on that interface is delegating (equal to the outer *IUnknown's QueryInterface* mapping). Therefore, it is possible to get from that interface to the *IUnknown* interface on the outer object. The symmetry rule requires that we be able to get from there "back" to an interface of type *IID_ISome*. But that contradicts the assumption that the outer object does not export an interface of type *IID_ISome*. The assumption was untenable and so the lemma is proved.

Thus, an outer object must export an interface of each type (*IID*) obtainable through *QueryInterface* on a non-delegating inner *IUnknown* interface. Given our definition of abstraction, the proof of our theorem follows immediately: COM does not support abstraction via selective hiding of *the kinds of* interfaces on inner objects. As a corollary, our proposed architectural style is illegal, since it depends on that kind of selective hiding.

RESULTS

We present a number of results. First, our proposed architectural style appeared to be a novel, natural and powerful new way to use COM. Although it does not work, we are confident that something like it will. We are using the machinery developed in this paper to facilitating our exploration and evaluation of the designs of related architectural styles.

Second, we showed that our proposed architectural styles and others that depend on abstraction though information hiding using aggregation are illegal. Third, through a careful verification effort, we caught a serious architectural design flaw in a particular commercial development project before the cost really hurt the corporation. Fourth, we used *formal methods* profitably in a commercial setting to verify the legality of a proposed architectural design style based on the standard. Fifth, we document a profitable application of formal methods to the validation of important, non-obvious properties of a widely used architectural design standard.

Our conclusion and lemmas and theorem behind it were unexpected outcomes of our initial work on the design of an architectural style. The critical implications of what we now see as major subtleties in the COM standard were not obvious to us.

Moreover, evidence suggests that these subtleties may not be obvious even to COM experts. For example, in "The Rules of the Component Object Model," Charlie Kindel, Program Manager for Windows NT, states that "**QueryInterface** must be reflexive, symmetric, and transitive with respect to the set of interfaces that are accessible [11]." Our use of formal methods shows that this should say, *QueryInterface* must be reflexive, symmetric, and transitive with respect to the set of *interface identifiers (IIDs) for which* interfaces are accessible. Indeed, if the rule were as Kindel put it, any use of aggregation would violate the rules of COM: Specifically, our explanation for why the common use is legal would not work.

Thus, even the internal consistency of the COM standard turns out to be subtle. COM "works" *because* it does not require closure on the set of interfaces, a degree of freedom is presented as merely an opportunity for low-level optimization. "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.... [6]"

Finally, and perhaps most importantly, we documented a case that supports the claim that we should treat widely used architectural standards as critical infrastructure systems. The well-being of a small corporation was at risk in our case. The wide and early use of such standards can easily put others at similar, serious risk. It should be possible for adopters to use standards (their specifications, design and implementation in the case of runtime support) without an undue burden of risk. Our case focused on the need to make architecturally important implications of subtleties explicit, but other validation concerns will arise (e.g., ambiguity or inconsistency).

Our concern for the integrity of key standards is not unprecedented, of course. For example, Ardis et al. report that ambiguities in the specification of a telecommunications protocol make it "...possible to completely defeat the protection switching protocol, causing the communication link to fail, even though there was at least one working line in each direction [1]." Ardis et al. then express the hope that "...future authors of standards will consider using formal languages, so that ambiguity can be minimized [1]." Our claim that formal methods have an important role to play in validating *widely used architectural standards* agrees with the view of Ardis et al.

EVALUATION

We believe that we have characterized critical but previously unrecognized architectural implications of subtleties in the design of COM. We hope that our results will warn designers away from a tempting but forbidden corner of the design space. More generally, we believe that our results can help ease the task of reasoning about COM. We believe the machinery we have developed can help to answer a range of important questions that we do not address explicitly. Issues concerning object identity and the grouping of interfaces come to mind. We expect that our work can help to rationalize the investigation of COM-based design idioms, such as Goswell's [10].

An important question that we cannot yet answer adequately is what degrees of coverage and rigor are most appropriate in the validation of architectural standards and verifications of their use. We used concepts and notations from discrete mathematics, but not advanced tools, fully formal specification languages or mechanized theorem provers or checkers. Our application of formal methods was restricted to early life cycle phases. Moreover, we formalized just a few (but critical) elements of the standard. Finally our method was ad hoc—guided by the demands of a particular situation—not part of a systematic process. In terms of Rushby's taxonomy of approaches to the use of formal methods [14] our levels of coverage and rigor were modest.

Ultimately, we present a single data point, so strong, broad generalizations are unwarranted. Nevertheless, we are confident that formal methods should be brought to bear on the validation of widely used architectural standards. It is critical to relieve users of the risky burden of reasoning about subtleties in such standards. The message for developers is that, until such time as key standards are subjected to validation efforts commensurate with their role as critical systems, it is in the developers' interest to take a rigorous approach to verifying that their architectures conform to possibly subtle requirements. Formal methods can play an important, profitable role in such verification efforts, as well.

Finally, we want to comment on COM. The bottom line is that while the discoveries that we made in this work came as real surprises, nothing that we found has changed the decision by Socha Computing, Inc. to use COM as the basis for its Herman multimedia authoring tool. The design of COM is elegant and innovative. Moreover it is very useful, widely used, and the foundation for many applications in extremely wide use. COM is clearly not fatally broken. Users will however benefit by the results of careful validation and characterization of COM based on the judicious use of formal methods.

RELATED WORK

Related works falls in several categories: software architecture, object models, formal methods. At the intersection of architecture and formal methods are efforts such as those of Garlan [9] and Luckham [12]. By contrast, our focus is on *widely used architectural standards*. Both Luckham and Garlan model dynamic computational semantics at the architectural level (e.g., event ordering in Rapide or fair scheduling in pipe and filter architectures). We address only certain key structures. Unlike Shaw and Garlan [9], we do not assume a general systems theory-based entity/relation (component and connector) framework for describing architecture.

The obvious related work in the area of object models is that on OMG's CORBA. Our decision to focus on COM followed from the choice of COM for use in Herman. A common contrast is that CORBA supports implementation inheritance while COM does not. We will not comment on that debate in this paper. We are aware of but have not yet studied carefully a published paper on the formal specification of CORBA's core object model [5]. It appears that the primary impetus for the specification was the need for a concrete focus for consensus-making among the members of the committee designing the CORBA standard, and not for validation to aid end-user verification efforts.

CONCLUSION

The few aspects of COM we have discussed—multiple interfaces, *QueryInterface*, and (to a lesser extent) aggregation—are foundations of the COM standard. Brockenschmidt calls multiple interfaces, "one of OLE's strongest architectural features... [4]," while the COM specification emphasizes that "... [T]he powerful and important *QueryInterface* mechanism ... for all intents and purposes is the single most important aspect of true system component software [6]." Aggregation is presented as one of two basic component reuse mechanisms in COM.

Because COM and other such standards are widely used foundations for vast numbers of important systems, the critical implications of any major subtleties at the core of such standards must be explicated with great clarity. Widely used standards should be subjected to rigorous scrutiny employing at least modest (and probably aggressive) use of formalism. The care with which standards are designed, specified and implemented must be commensurate with their status as critical infrastructure systems

ACKNOWLEDGEMENTS

Sullivan's interest in COM was bolstered by several conversations, some with David Notkin, others with people who may wish to remain anonymous. David Socha came up with the idea of using aggregation as a mechanism to abstract mediator-based subsystems. John Knight provided useful pointers into the formal specification literature. This work was supported by the National Science Foundation.

REFERENCES

1. Ardis, M.A., J.A. Chaves, L. J. Jagadeesan, P. Mataga, C. Puchol, M.g. Staskauskas and J. Von Onnhausen, "A framework for evaluating specification methods for reactive systems," *Transactions on Software Engineering*, vol 22, no. 6, June, 1996, pp. 378-389.
2. Batory, D. and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Transactions on Software Engineering and Methodology 1*,4, pp. 355-398, Oct. 1992.
3. Batory, D., L. Coglianese, M. Goodwin and S. Shafer, "Creating reference architectures: an example from avionics," Proceedings of SSR'95, *Software Engineering Notes*, April 28-30, 1995, pp. 27-37.
4. Brockschmidt, K., *Inside OLE*, Microsoft Press, 1996.
5. Bryant, T. and A. Evans, "Formalizing the object management group's core object model," *Computer Standards and Interfaces*, vol 17, no. 5-6, pp. 481-9, September 30, 1995.
6. Common [sic] Object Model Specification, Microsoft Developer Network Library, Microsoft Corporation, 1996 (especially sections 3.3.1 and 6.6.2).
7. Common Object Request Broker Architecture, Object Management Group, Inc.
8. Craigen, D., S. Gerhart and T. Ralston, "An international survey of Industrial Applications of Formal Methods, Volumes 1 and 2," NIST GCR 92/626, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, March, 1993.
9. Garlan, D. and M. Shaw, "An introduction to software architecture," *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Publishing, 1993.
10. 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.

11. Kindel, C., "The rules of the component object model," Microsoft Developer Network Library, Technical Articles: Windows: OLE COM, Microsoft Corporation, October 20, 1995.
12. Luckham, IEEE Transactions on Software Engineering, 1995.
13. Parnas, D., "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, December, 1972, pp. 1053-1058.
14. Rushby, J., *Formal Methods and Digital Systems Validation for Airborne Systems,* NASA Contractor Report 4551, National Aeronautics and Space Administration, Office of Management, Scientific and Technical Information Program, 1993.
15. Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline,* Prentice-Hall, 1996.
16. Spivey, *The Z Specification Language,* Prentice-Hall.
17. Sullivan, K.J., "Mediators: Easing the Design and Evolution of Integrated Systems," Ph.D. Dissertation, University of Washington Department of Computer Science, August, 1994.
18. Sullivan, K.J. and D. Notkin, "Reconciling Environment Integration and Evolution," ACM *Transactions on Software Engineering and Methodology* vol. 1, no. 3, July, 1992.
19. Sullivan, K.J., I.J. Kalet and D. Notkin, "Mediators in a Radiation Treatment Planning System," IEEE *Transactions on Software Engineering*, to appear.
20. Taylor, R.N., N Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy and D.L. Dubrow, "A component- and message-based architectural style for GUI software," *IEEE Transactions on Software Engineering* 22,6, pp. 390-406, June, 1996.
21. Visual Basic 4 User's Manual, Microsoft Corporation, 1996.