Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse

Kevin J. Sullivan         John C. Knight
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
(sullivan | knight)@virginia.edu

Abstract

Systematic reuse of large-scale software components promises rapid, low cost development of high-quality software through the straightforward integration of existing software assets. To date this promise remains largely unrealized, owing to technical, managerial, cultural, and legal barriers. One important technical barrier is architectural mismatch. Recently, several component integration architectures have been developed that purport to promote large-scale reuse. Microsoft's OLE technology and associated applications are representative of this trend. To understand the potential of these architectures to enable large-scale reuse, we evaluated OLE by using it to develop a novel fault-tree analysis tool. Although difficulties remain, the approach appears to overcome architectural impediments that have hindered some previous large-scale reuse attempts, to be practical for use in many domains, and to represent significant progress towards realizing the promise of large-scale systematic reuse.

1. Introduction

Large-scale, systematic software reuse promises rapid, low-cost development of major systems through straightforward integration of existing assets [1, 3]. The realization of this promise would provide benefits in many areas. One benefit of direct interest to the software engineering research community would be a means to rapidly and inexpensively develop software tools to make new research results useful to practicing engineers in the field.

To date the promise of large-scale reuse remains largely unrealized. Although some success has been achieved, barriers remain in a variety of areas: technical, managerial, cultural, legal.

In this paper we address the important technical barrier to large-scale reuse called architectural mismatch [5]. Garlan et al. coined this term to describe a variety of difficulties encountered in building an application from a collection of large-scale existing software components. Their experience enabled them to characterize the difficulties and to systematically analyze their causes from an architectural stance [6, 7, 13].

Garlan et al. identified four main categories of architectural mismatch: incompatibilities in assumptions about the nature of components; the nature of connectors; the global architectural structure; and the construction process. In each category mismatches occur because of conflicts in assumptions made by developers that are reasonable in isolation but that lead to mismatches in a reuse situation.

Recently, several architectures have been developed that purport to enable system development through the integration of large-scale, reusable components. Most notable are Microsoft's OLE [2] (a proprietary, de facto standard) and CORBA [9,10,11] (a non-proprietary, de jure standard developed by an industry consortium). The advent of such architectures, along with a variety of architecture-compliant component applications, raises the question of how well they support reuse and how they deal with the elements of architectural mismatch identified by Garlan et al.

In this paper we report an evaluation of the architectural approach to large-scale reuse represented by OLE. Our evaluation follows an experiment in which we developed a substantial application by integrating large-scale reusable components based on OLE. The result of our evaluation is that, although difficulties remain, the approach appears to have made significant progress towards realizing the promise of rapid development of sophisticated systems through reuse. In particular, the designers of OLE appear to have avoided many of the elements of architectural mismatch identified by Garlan et al. In addition, we observe that the approach is practical now in many domains.

We selected OLE for our experiment for two reasons. First, for our purposes in this paper, OLE appeared to be representative of the component architecture approach in general. Second, OLE and a variety of OLE applications were readily available and familiar to us. We believe that our work and conclusions are applicable to component integration architectures more generally.

In the next section of this paper we review OLE. Following that, we describe the application software system that we built and how it was developed. Next we present the results of the development process and finally we present our conclusions.
2. The OLE architectural framework

We begin by introducing OLE, as an exemplar of a component integration architecture. In Inside OLE, Craig Brockschmidt defines OLE as “a unified environment of object-based services with the capability of both customizing those services and arbitrarily extending the architecture through custom services, with the overall purpose of enabling rich integration between components.” A component is “made of one or more objects, where each object then provides its functionality through one or more interfaces [2, p. 9].”

The components of interest here are large-scale applications, such as spreadsheets, drawing and word processing programs, and the like. These applications comprise object hierarchies that can be manipulated by external programs. Of particular relevance to us are OLE Automation (OA) applications. OA permits one program or application (called an Automation client) to manipulate another application (called an Automation server). Clients can be written in interpreted languages such as Visual Basic or in compiled languages such as C++. Support for interpreted languages is based on OA’s run-time binding of operation names.

While a comprehensive overview of OLE is unnecessary and infeasible in this paper, a brief enumeration of key, relevant features will help the reader understand both why this kind of technology represents an important architectural advance, and how the Automation capabilities referenced in this paper work [2].

- **OLE is an object-based architectural framework.**
  OLE supports a wide range of object services, not all of which we can describe in this limited space. The services most relevant to our work on tool integration are support for explicit and implicit invocation within and across process boundaries. In principle, these services should enable tight integration of appropriately designed component applications. This aspect of OLE suggested to us the use of the mediator approach to tool integration that we describe below.

- **OLE objects conform to binary interface standards.**
  OLE objects conform to a binary as opposed to source-code interface standard. This property of OLE obviates source code incompatibilities and supports interoperability of separately developed, separately marketed component applications.

- **OLE objects support a multiple interface model.**
  In contrast to the traditional object model in which an object exports a single interface, OLE objects export multiple interfaces. Each interface supports a possibly complex service. To supports implicit invocation, for example, an object exports and implements the IConnectionPoint interface—among interfaces for other services, such as structured, persistent storage.

- **OLE Automation supports a single interface model.**
  Automation does not support multiple interfaces. Rather, an Automation interface has a traditional single interface structure. The whole Automation interface for an object is in fact implemented through one standard OLE interface, called IDispatch.

- **Calls to OLE interfaces are bound at compile time.**
  Clients of an object access its services through an interface. Clients do not have direct access to the object’s underlying data. An OLE interface is implemented using a structure called a vtable. A vtable contains a sequence of pointers to implementations of the operations defined in the interface. The structure of a vtable (i.e., the ordering of these pointers) is fixed at compile time. Client references to operations are resolved at compile time using this ordering information. Convenient support for runtime binding of operation thus requires an added level of indirection.

- **IDispatch provides an added level of indirection.**
  The IDispatch interface provides for the late binding needed by OA. IDispatch exports an operation called Dispatch. Dispatch uses an assignment of integer identifiers (called dispIDs) to the operations exported by other interfaces. A DispID of 1 might identify the event announcement operation exported by the IConnectionPoint interface, for example. Dispatch takes a DispID and invokes the designated operation. A compile-time binding to Dispatch thus permits bindings of other calls to be delayed until runtime.

- **Visual Basic hides calls to Dispatch.**
  The Automation capability of Visual Basic encapsulates calls to Dispatch. Calls to an Automation object are translated by Basic into calls to Dispatch. This encapsulated, late binding of procedure calls makes it easy to write Visual Basic programs that drive Automation-enabled applications. Given a variable shpSObj referring to the collection of shapes in an open Visio drawing, for example, you can obtain the first shape in the collection with the Visual Basic statement Set shpSObj = shpsObj.Item(1). Visual Basic translates the right hand side of the assignment statement into an OA call to the Item operation of the shpSObj object.
3. Evaluating the reuse architecture

In order to gain insight into the potential for component integration architectures to support large-scale reuse, we decided to use OLE and associated, volume-priced application components to develop an industrial-quality software tool to support an innovative reliability analysis technique developed by Joanne Dugan at the University of Virginia. Given claims made on behalf of OLE, it seemed reasonable to try using mediators [15, 16] to build a full-featured tool by integrating volume-priced application components, such as Shapeware’s Visio drawing tool and Microsoft’s Access database program.

A secondary goal in developing this tool was to examine the potential for large-scale reuse to help deliver promising results from the laboratory to designers and engineers in the field. The high cost of software delivery vehicles—of all the superstructure needed to make a new technique truly useful in practice—impedes the transfer of innovations to market. Consequently, potentially profitable technologies may languish in the laboratory, depressing the returns on investments in research and denying benefits to practitioners. Mary Shaw has observed that,

Most applications devote less than 10% of their code to the overt function of the system; the other 90% goes into system or administrative code: input and output; user interfaces, text editing, basic graphics, and standard dialogs; communications; data validation and audit trails; basic definitions for the domain such as mathematical or statistical libraries; and so on [14, pp. 4-5].

If large-scale reuse enables the construction of sophisticated packaging at low cost, the resulting improvements in productivity and quality could provide considerable benefits by removing a major barrier to practical application of novel research results.

3.1. The application

The application we developed as part of this research is a tool for supporting a technique called fault-tree analysis, a technique that is frequently used in systems engineering of safety-critical applications. A fault tree is a structure that shows how events that can occur in a system can lead to hazards. Fault trees can be analyzed in a variety of ways to yield information such as the probability of a hazard arising given the probabilities of occurrence of the various events. Figure 1 presents an example a fragment of a fault tree from a nuclear reactor application.

Recently, novel techniques for the analysis of fault trees have been developed [4]. These techniques promise much faster and more accurate analyses, and they add new capabilities such as the ability to deal with circumstances in which the order of events in a fault tree is significant. The goal of the software development project we undertook was to produce a tool that would make these techniques available quickly and in a form with features and performance characteristics that would be similar to commercial products. We note that a number of commercial fault-tree analysis packages are available—RISKMAN [12] for example—but such tools lack the new analysis techniques.

Although the fault-tree analysis function is coded in under ten thousand lines of C++, providing it to users requires a vastly more complex delivery vehicle. Industrial users will look for such features as a database for fault tree storage; graphical rendering and direct manipulation user interfaces; inclusion of graphics, database views and analysis results in reports prepared for technical and managerial audiences; and clean integration of the tool into the organization’s overall engineering process. Thus, we framed the following requirements for our tool:

- The ability to manipulate the graphic representation of a fault tree via a “point-and-click” interface including the creation and deletion of nodes and arcs, zooming in and out to change the view presented, and the ability to encapsulate a subtree into a single node so as to permit hierarchic tree definitions.
- The ability to annotate the nodes of the fault tree via the graphic representation with a variety of information types, including numeric (such as probabilities) and textual (such as event descriptions).
- The ability to apply layout algorithms to the graphic representation of a fault tree so as to arrange the display in an aesthetic manner. Given the specialized nature of the displays in this case, the layout algorithms might well have to be developed as separate packages. It is important however, that the “hooks” for such processing be available.
- The ability to store and retrieve fault trees so as to retain all structural information and annotations.
- The ability to manipulate collections of fault trees.
- The ability to invoke simple analyses on fault trees such as the preparation of lists of nodes with particular properties.
- The ability to invoke specialized analyses programmed as separate packages. The ability to integrate a revised implementation of the analyses quickly so as to permit experimentation by the developers of the analysis techniques.
- The ability to include formatted fault trees and the results of analyses in documents.
Using conventional software development methods (such as object-oriented programming with class libraries) to build industrially viable tools is so costly that it is generally feasible only for the most promising technologies. Moreover, commercial developers are unlikely to incur high development costs to deliver specialized technologies to what may be small or ephemeral markets.

We saw the possibility to overcome these problems by crafting a sophisticated delivery vehicle by reusing OLE-compliant application components. Some of these components are remarkably powerful, and taken together they appear to span the spectrum of capabilities needed for delivery vehicles for a wide variety of applications.

The capabilities include relational database management, graphical user interface construction, compound document design and storage, constraint-based structured interactive graphics, and diverse computational models, including spreadsheets and general-purpose imperative programming in languages such as C++. In reviewing these components, we observed that the rich features they offer are applicable to many domains—and not merely management information systems or data processing.

Moreover, many of the applications are ostensibly designed for integration and reuse. Microsoft presents OLE and related applications as enabling rapid development of applications from components. Component technologies, especially those based on Visual Basic Custom Controls, have been used extensively to build a variety of business and other applications. We asked whether we might not take the same approach to deliver an advanced reliability engineering technique into practical use.

3.2. Developing the tool

In this section, we discuss the design and implementation of our tool. First, we discuss our decision to build the tool as an environment in which Visio and Access are tightly integrated component applications. Next, we present a straightforward design for this environment using a mediator to integrate Visio and Access. Finally, we discuss the ways in which we were compelled by architectural difficulties with OLE and our component applications to diverge from our straightforward design.

Behavioral Entity-Relationship Model: We observed that each of several applications seemed well suited to handle a particular aspect of the requirements. This led us to adopt an “integrated environment” approach in our system design. Each part of the problem is handled by a particular application. Our task, then, is to integrate the applications to solve the overall problem.

This approach implies the distribution of related data over multiple components. For example, we represent a hazard in a fault-tree as a record in an Access database and as a shape in a Visio drawing. The hazard being modeled is represented by correlated data in both applications.

Each component thus maintains a projection of the “phenomena of interest,” and the conjunction of the projections comprises the overall representation of the phenomena. There is no representation of the phenomena other than as the conjunction of the related, possibly overlapping, partial representations. This multiplicity of representations requires consistency maintenance. Relieving users of the consistency maintenance burden demands behavioral integration of the component applications.
First, it is necessary to integrate the separate, partial representations into a coherent representation of the phenomena. Our prototype tool has no central representation of a fault tree; so to represent associations between views of a fault tree, we explicitly link the data in separate applications. For example, we associate Visio shapes in a fault tree drawing with corresponding Access database records.

Second, it is necessary to maintain system-wide coherency of this distributed representation in the face of user manipulation of the views presented by separate applications. We have to ensure that all data are or can be made consistent with the fault tree intended by the user. If the user indicates the addition of a hazard by adding a shape to a Visio drawing, a corresponding record should be added to the Access database, and the correspondence between the two must be represented. The same considerations apply to gates, entire fault trees, collections of trees, etc.

Figure 2 depicts the integration requirements in the form of a behavioral entity-relationship (ER) model [15, 16]. This model represents the behavior of the system as a network of visible, independent behaviors (Visio and Access) integrated by a behavioral relationship that models how the applications work together (the arrow). The behavioral ER model is a very high-level architectural view with application behaviors as the components and behavioral relationships responsible for integration as the connectors.

Ideal Mediator-Based Design: Since the applications are given, our task is to implement the behavioral relationship. The question is how to implement it using available mechanisms and component interfaces. The mediator approach [15, 16] appears to be an ideal solution. We thus tried to implement the relationship as a mediator that integrates the applications without compromising their visibility or independence.

We did this by designing the mediator as a component that interacts with the applications through a carefully engineered combination of implicit and explicit invocations (event notification and procedure call). The approach seemed natural because we had understood these mechanisms to be supported by the basic OLE technology.

Figure 3 depicts the mediator architecture that we posited. The mediator registers with the application events to be invoked implicitly when component activities require consistency maintenance. Once invoked, the mediator explicitly invokes the other applications as needed to maintain consistency. We intended to use Visual Basic OLE Automation for both explicit and implicit invocation.

For example, consider how the mediator would handle the addition of a hazard shape to the Visio drawing, or a change to the name of a hazard using Access. The mediator responds to an event from Visio indicating the addition of a shape by adding a corresponding record to the Access table and storing the association between the shape and the record. Notification of a change in the name field of a hazard record in Access prompts the mediator to look up the corresponding shape and to update its name in Visio.

The mediator also provides a user interface. This interface presents a view of the system as a whole, including menu items to invoke the fault-tree analysis function, to open and close collections of fault trees in a coordinated fashion, to add and delete new fault trees, and so forth.

The Actual Compromised Design: When we tried to implement our design, we ran into several problems that required us to compromise the mediator architecture. The good news is, first, that we succeeded in building (almost) the system we wanted; and, second, with one major exception, the problems we had were not with OLE itself, but in the designs of the component applications. We highlight the architectural compromises we had to make by illustrating the structural differences between the ideal mediator design and our actual design. We then discuss the underlying architectural problems in terms of the compromises we had to accept.

To illustrate the architectural compromises, we use graphical conventions similar to those used in reflexion models [8]. Specifically, we present a figure that highlights where the ideal mediator structure and the actual structure agree and where they differ. We highlight both the absence of expected structures and the presence of unexpected structures in the actual architecture.
Figure 4 presents this comparison and contrast. The heavy black line indicates the structure that is present in the mediator architecture and also in the actual system. The heavy dashed lines indicate structures present in the actual system but missing from the ideal one. The light dashed lines indicate structures present in the ideal architecture but missing from the actual system. The shading indicates a change in a component definition. We elide the analysis component for the sake of clarity in Figure 4.

These discrepancies reflect architectural difficulties. We quickly discovered that Access 3.0 does not export operations, events, or a data model suitable for use by an OLE Automation client.† This restriction prevents the mediator from invoking or being invoked by Access. The light dashed lines between the mediator and Access indicate that the designated implicit and explicit invocations are missing from the actual system architecture.

We worked around this problem by using the Visual Basic language’s built-in database functions, which use the same underlying “database engine” as Access. Our mediator communicates with Access by writing to a low-level database shared between Basic and Access. Access is triggered when the underlying database changes—through a standard implicit invocation mechanism, we presume.

The light dashed line from the Visual Basic mediator to Visio is more interesting. The missing implicit invocation from Visio to Visual Basic reflects two problems—one with OLE, and one with the design of the Visio interface.

The more serious problem is that Visual Basic 3.0 does not support procedure pointers as a data type, precluding the use of “call-backs” to implement implicit invocation. In fact, Visual Basic 3.0 does not support explicit or implicit invocation of components written in Visual Basic using OLE Automation. While Visual Basic 3.0 is a flexible Automation client, able to drive other programs effectively, other programs appear to have no easy way to way to drive components written in Visual Basic.††

This is a serious and unfortunate problem because it precludes the use of mediators, which have proven to be a highly effective structuring method for integrated systems. Moreover, we see no easy workarounds in the current version of the language. This deficiency essentially defeated our hopes to define a mediator that would eagerly and incrementally maintain consistency in the face of fine-grained operations on Visio, such as shape addition and deletion.

The missing implicit invocation between the mediator and Visio is due not only to shortcomings of Visual Basic but also to the insufficiently rich event interface provided.

†. Access 4.0 does support OLE Automation. We highlight our difficulty with Access 3.0 because it illustrates a general problem that component-based system architects must consider and watch out for: absence of suitable interfaces.

††. Visual Basic 4.0 (which shipped between the time this paper was submitted for review and for publication) provides support for explicit invocation of Visual Basic objects using OLE Automation. We have begun exploring implementations of implicit invocation mechanisms for Visual Basic using this new capability.

---

Fig. 3 - Proposed mediator-based architecture.
by Visio 3.0. The events that Visio announces are insufficient to support our integration requirements. Visio announces events to signal that shapes have moved, have had their names changed, and have been double clicked; but incrementally maintaining global consistency also requires events signalling deletions, which Visio does not announce.

This combination of difficulties led us to give up on the idea of tool-triggered, eager, incremental consistency maintenance. Instead, we settled for occasional consistency restoration triggered by the user’s selection of a menu item in the master control window. When the user selects this item, a procedure uses OLE Automation to inspect the set of shapes and connections in the active Visio picture. This set is compared with the records in the database; and appropriate additions, deletions and modifications are made to the database.

A final interesting problem that we had was lack of unique identifiers for Visio shapes. Visio 3.0 assigns each shape an identification number, but the number is not guaranteed to be unique within a drawing, only within a page. Because a single drawing can span multiple pages, two shapes in the same drawing can end up with the same identification number. Nor are these numbers guaranteed to be unique across time: Deleting one shape and then adding another can result in the new shape having the same number as the old one. These properties of shape identification numbers make them unsuitable as keys in external representations of associations between Visio shapes and database records.

Our solution exploited a useful extensibility property of Visio shape objects: They provide several empty slots in which one can store arbitrary data. We use one of these to store the key of the database record corresponding to a given shape. It is critical for integration that tools provide means by which associations between their data and data maintained in other tools can be represented.

We conclude this section with the observation that, although we encountered “serious” architectural impediments, we were still largely able to build our system in about a person week by integrating large-scale, reusable components. The result is a fully functional tool at a cost that is extraordinarily low by historical standards.
4. Results

The results we obtained from this activity fall broadly into two areas. One area is the relatively successful performance of the effort—a useful tool was built quickly and at low cost. The second area is architectural assessment. We uncovered a number of subtle difficulties with both the OLE architecture and the application components. The difficulties we encountered highlight important general problems for designers and users of component integration architectures and applications. In the remainder of these sections we discuss the specifics of these two areas.

- **A very high level of productivity achieved.**
  
  With about a person week of effort, we developed a tool that met virtually all the requirements outlined above using existing capabilities of the reusable application components. We wrote a small amount of software to implement the integration and overall control mechanisms. The sophisticated analysis components were developed from scratch. However, we note that the analysis package is less than 10,000 lines of code, yet is available within a framework of functionality that is implemented by probably several million lines.

- **Industrial-strength capability demonstrated.**
  
  The tool we have developed provides a wealth of functionality as well as the essential analysis capability. For example, the full capabilities of Visio and Access are at the user’s disposal thereby enabling tremendous graphic and reporting facilities. Compared with commercial tools in the same domain, the tool we built could be enhanced to include comparable analytic capabilities in at most a few person months.

- **Familiar “look and feel” provided.**
  
  Because the application components that we used are in common use, the appearance of the tool will be familiar to the user. In addition, since the underlying computing platform is Microsoft’s WindowsTM, the entire operating environment will be familiar. An important inference from this is that entire collections of tools can be developed that will all maintain the same look and feel.

- **Adequate performance achieved.**
  
  We have no measured performance information for the tool we have built. However, when executing on a 90 MHz Intel-Pentium-based computer, the performance is “adequate” for normal use. This is important since performance could easily become quite unacceptable with the techniques used. OLE demands late binding of remote procedure calls so quite a lot has to go on at execution time. In our experience to date, this has not been a significant issue.

- **Incomplete event interfaces.**
  
  A key element to the successful integration of application components is that component interfaces provide complete, consistent, and *timely* information about their state via events. We found a significant deficiency in this area. The Visio 3.0 event interface, is not sufficiently rich to support eager incremental consistency maintenance. For example, there are no events to signal impending object deletions. As a result the changes needed in other components have to be delayed.

- **Incomplete OLE support.**
  
  One of the assets that we used, Microsoft’s Access, did not provide an interface supporting OLE automation at all, in the version available to us. The result was that we had to integrate Access into the tool using a completely different mechanism. It was fortunate that we were able to work around the problem in this case, but this is not going to be possible in every case.

- **Inconsistent OLE support.**
  
  The ability to integrate tools rapidly using a scripting language that includes a reasonable programming environment is very valuable. However, we found a serious difficulty with the Visual Basic 3.0 instantiation of this concept. The difficulty was that while Visual Basic 3.0 is an excellent OLE Automation client, it cannot be used to implement OLE Automation servers. Thus it cannot be used to implement a mediator that responds to events from application components. It is very hard (essentially impossible) to work around this difficulty. This forced us to adopt a batch consistency model rather than our preferred incremental consistency maintenance approach.

- **Inconsistent object naming.**
  
  Visio makes it harder than necessary to corollate Visio objects with external objects because it fails to provide light-weight object identifiers. It does assign objects integer ID’s and those ID’s are guaranteed to be unique per page of a drawing but not across pages nor through time. Thus we had to implement our own.

- **Insufficient component adaptability.**
  
  The application components that we used are rich pieces of software. In some cases, we needed to disable certain functions because they either were not necessary or because they provided ways to circumvent essential functions such as consistency mainte-
nance. We found that we were not able to disable functionality with the degree of flexibility that we desired. In a similar vein, we needed to change the user interfaces integral to the components to “distinguish” our tool from its component applications. The provision for this form of tailoring in the application components was, in general, far less than we needed.

- **Only functioning software is produced.**

Although we have developed a relatively sophisticated tool, all we have is the software. We have no documentation, no users’ guide, and so on. In terms of the software development lifecycle cost, it is not clear how much of an advantage we have achieved.

5. Conclusion

In the work described in this paper, we evaluated the potential for component integration architectures to realize the promise of large-scale systematic reuse. We took OLE as a representative architecture. We conclude that the approach provides a powerful technique for reuse-based software development. Although difficulties remain, such an approach is practical now in many domains. It substantially overcomes the architectural impediments that have hindered some previous large-scale reuse attempts. It appears to represent significant progress towards realizing the promise of rapid software development through integration of large-scale, reusable application components.

In essence, a key reason that the OLE technology worked in this case, and that we did not face the complex architectural mismatch difficulties reported by Garlan et al., is that the component integration architecture defines a complete framework of design standards intended to support integration. Although we experienced some architectural difficulties, the various assumptions that precipitated the severe architectural mismatches experienced by Garlan et al. were not present in our case. Since most of the components were designed for use with OLE, they were built with the same architectural techniques.

A key lesson is that, if components are to be composable, they have to be designed for it. Although our experience was largely positive, the technology will be more difficult to exploit than it needs to be unless this lesson is heeded. Furthermore, reuse will be restricted unless new application components are designed both to support the architecture from the outset, and to export interfaces intended for integration. A good analogy is that of the early days of the railroads when every railroad company defined its own track gauge. The effect was to require extensive effort at the “interfaces”.

It is important to note that the experience we report is based on the development of only a modestly ambitious system. This raises the issue of whether the reuse technology that we used will scale up more ambitious systems.

Assumptions or assertions of scalability from known cases to uncharted areas of design space are dangerous and rarely warranted. However, it is reasonable to take “scalability to more complex applications” as a working hypothesis, given both our experience to date and our understanding of the OLE architecture. A wide variety of OLE-compliant application components either exist or could be developed. **OLE per se** supports the implicit and explicit invocation mechanisms needed for effective component integration. Support is forthcoming for OLE Automation-based invocation of objects coded in Visual Basic. In addition, OLE supports a suite of other object services, services that we did not evaluate in this paper. We note that competing component standards such as CORBA provide comparable, attractive technical features.

6. Acknowledgments

It is a pleasure to acknowledge the extensive advice we received on the analysis of fault trees from Joanne Dugan. We are also pleased to acknowledge the programming assistance that we received from Mike Lee.

This work was funded in part by Motorola, in part by the National Science Foundation under grant numbers CCR-9213427 and CCR-9502029, and in part by NASA under grant number NAG1-1123-FDP.

7. References


