# ASSESSING AN ARCHITECTURAL APPROACH TO LARGE-SCALE SYSTEMATIC REUSE†

Kevin J. Sullivan and John C. Knight

(sullivan | knight)@virginia.EDU

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Contact Author:

Kevin J. Sullivan
Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903

(804)982-2216
sullivan@Virginia.EDU

Keywords: Software reuse, software architecture, software design.

---

†. Supported 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.

# ABSTRACT

Large-scale systematic reuse promises rapid development of significant systems through straightforward composition of large-scale existing assets. The realization of this promise would provide major benefits in many areas. For example, sophisticated software-engineering tools could be developed rapidly and inexpensively to deliver promising software engineering research results into practice.

To date the promise of large-scale reuse remains largely unrealized. Although some successes have been achieved, barriers remain in a variety of areas: technical, managerial, cultural, and legal. In this paper we address an important technical barrier: architectural mismatch.

Architectural mismatch has been identified as an important barrier to large-scale reuse. Recently architectural frameworks that purport to enable large-scale reuse have been developed. Among them is Microsoft's OLE technology, comprising both an architectural framework and a suite of reusable component applications. The manufacturer presents this technology as a toolkit for the rapid development of applications from components.

In reviewing these component applications, we observed that they offer rich features applicable to a number of domains—and not merely management information systems or data processing. The components are both designed for reuse and also seem to span the spectrum of capabilities required to build 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 this paper we evaluate the approach to large-scale systematic reuse represented by OLE. We conclude that, although difficulties remain, such an approach is practical now in many domains, that it substantially overcomes the architectural impediments that have hindered some previous attempts at large-scale reuse, and that it represents significant progress towards realizing the promise of rapid development of sophisticated systems.

We report on our prototyping approach to the evaluation of this technology. Our evaluation focused on the ability of the technology to support the development of software-engineering tools. We define our evaluation framework, describe our experience developing a specific tool, and present conclusions of our evaluation.

# INTRODUCTION

Large-scale, systematic software reuse promises rapid development of major systems through straightforward composition of large-scale existing software assets [BP89, BS92]. The realization of this promise would provide major benefits in many areas. One benefit of direct interest to the software engineering community would be that sophisticated software-engineering tools could be developed rapidly and inexpensively to deliver the results of software engineering research to practice.

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, and legal.

In this paper we address architectural mismatch, an important technical barrier to large-scale reuse [GAO95]. Garlan et al. report a variety of difficulties in developing an application using a collection of large-scale existing components. Their experience enabled them to characterize the difficulties and offer a systematic analysis of causes of the problems from an architectural stance [PW92, GS93, GHJV94].

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

Recently, architectural frameworks that purport to enable large-scale reuse have been developed. Among them is Microsoft's OLE technology [Bro95], comprising both an architectural framework and a suite of reusable component applications. The capabilities of these component applications 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++.

The advent of such frameworks raises the question of how well they support reuse and how they deal with the various elements of architectural mismatch identified by Garlan et al. In this paper we report an evaluation of the approach to large-scale systematic 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.

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

# THE OLE ARCHITECTURAL FRAMEWORK

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***," where a component is "made of one or more *objects*, where each object then provides its functionality through one or more *interfaces* [Bro95, p. 9]."

The components of interest in this paper are applications that export objects and operations to manipulate them. Of particular relevance are *OLE Automation* objects and applications. The key to Automation is that the names of operations are bound at runtime, permitting Automation objects (called *Automation servers*) to be manipulated by programs (called *Automation clients*), which are often written in interpreted scripting languages such as Visual Basic. Our intention in the experiment we document here was to use Automation, as supported by Visual Basic, to tightly integrate off-the-shelf applications to produce a tool supporting a new systems engineering technology.

While a comprehensive overview of OLE is unnecessary and infeasible for 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 actually work [Bro95].

- *OLE is a comprehensive object-oriented 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, separate applications. This aspect of OLE suggested to us the use of the mediator-based approach to tool integration that we describe below.

- *OLE objects conform to binary not source-code 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 components.

- *OLE objects export multiple interfaces.*

  In contrast to the traditional object model in which an object exports a single interface, OLE objects can export multiple interfaces, each defining a possibly complex service. An object that supports implicit invocation, for example, implements the standard *IConnectionPoint* interface, in addition to interfaces for the other services it provides.

- *Automation reflects a more traditional interface mode.*

  Automation does not support multiple interfaces. Rather, an Automation interface has a traditional, monolithic structure. Indeed, the whole Automation interface for an object is implemented through one standard OLE interface called *IDispatch.*

- *The key difference between standard and Automation interfaces is binding time.*

  Standard OLE interfaces are implemented using structures called *vtables*. A *vtable* is a sequence of pointers to implementations of the operations in the interface. The structure of a *vtable* (i.e., the ordering of pointers) is fixed at compile time. Client references to operations must therefore have this compile time information. Convenient support for runtime binding of operation requires an added level of indirection.

- *IDispatch provide the added level of indirection.*

  Automation is based on the late binding of operations as effected by a standard interface called *IDispatch*. This interface exports an operation called *Dispatch*. The *Dispatch* operation depends on the assignment of integer identifiers (*dispIDs*) to operations exported through other interfaces. A *DispID* of 1 might identify the operation for announcing an event exported through the *IConnectionPoint* interface, for example. The *Dispatch* operation maps *DispIDs* to invocations of the indicated operations. A compile-time binding of the *Dispatch* operation thus permits other bindings to be deferred until runtime. Richer implementations of *Dispatch* enable operation names to be dynamically mapped to *DispIDs*, and from there to invocations.

- *Automation as supported by Visual Basic encapsulates IDispatch.*

  The Automation capability of Visual Basic rests on and encapsulates calls to *Dispatch* operations. Calls to an Automations objects are translated by the Basic runtime system into calls to *Dispatch*. The encapsulated, late binding of procedure calls based on their names makes it easy to write Visual Basic programs that drive Automation-enabled applications. This Automation capability, as supported by Visual Basic, was the integration mechanism we were counting on for rapid composition of our tool from existing, reusable component applications.

## EVALUATING THE REUSE ARCHITECTURE

In order to gain insight into the potential of this technology to support large-scale reuse, we decided to develop an industrial-quality software tool to support an innovative reliability analysis technique being developed by a colleague at the University of Virginia. Given claims made on behalf of the OLE component approach, it seemed that it would be plausible to use the mediator design method [SN92, Sul94] to integrate industrial-strength, volume-priced application components, such as Shapeware's Visio drawing tool and Microsoft's Access database program, into a high quality product.

One of our goals in developing this tool was to examine the potential for large-scale reuse to help deliver promising new research results to practitioners. We believe that the high cost of *software delivery vehicles*—of all of the super-structure required to make a technique useful in an industrial setting—tends to impede the transfer of technological innovations to market. The result of this is that potentially profitable technologies sometimes languish in the laboratory, depressing the returns on investments in research and denying the benefits of gains to practitioners. If this situation could be mitigated, the ben-

efits could be considerable.

## 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 as 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 [DD95]. 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 tool development project that we undertook was to produce a tool that would make these powerful new 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 for example [PLG], but such tools lack the new analysis techniques.

Although the fault-tree analysis technology developed by Dugan is coded in under ten thousand lines of C++, delivering the technology to users requires a vastly more complex delivery vehicle. Industrial users will look for such features as a transactional database for fault tree storage; graphical rendering and direct manipulation user interfaces; inclusion of graphic renderings, 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.

Indeed, the general functional requirements for the tool that we wanted to build were:

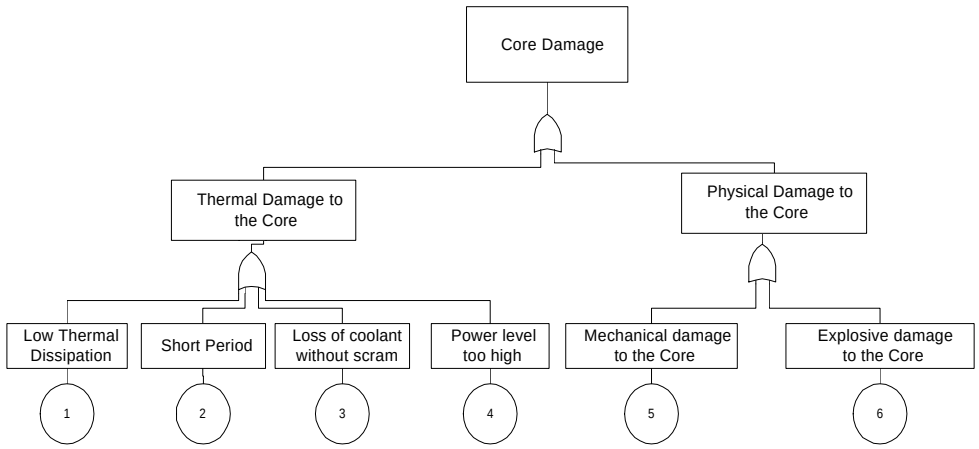


**Fig. 1. - Simple fault tree example.**

- 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 information (such as probabilities) and textual information (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 reusable 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 might be small or ephemeral markets.

We perceived the possibility to overcome these problems by crafting a sophisticated technology delivery vehicle out of reusable application components based on OLE. Some of the applications that have been developed with support for the OLE technology are remarkably powerful, and taken together appear to span the spectrum of capabilities required to build 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 component applications, we observed that the very rich features which they offer are applicable to a number of domains---and not merely management information systems or data processing.

Moreover, many of the applications are designed for integration and reuse. The manu-

facturer of OLE presents the technology as a toolkit for the 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 engineering technique into practical use.

## Developing the Tool

In this section, we discuss the architectural design and implementation of our tool. First, we discuss our decision to build the tool as an environment in which the Visio and Access tools 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 to diverge from the straightforward design by certain architectural difficulties with the OLE-based components.

### Behavioral Entity-Relationship Model of the Desired System

We observed that each of several applications seemed well suited to handle a particular aspect of requirements. This view led us to adopt an "integrated environment" approach to the design of our system. Each part of the problem is handled by a particular application. Our task was to integrate the applications to solve the overall problem.

This integrated systems approach implies the distribution of logically related data over multiple, largely independent components. For example, we represent an instance of the fault-tree notion of a hazard as a record in an Access database and as a shape in a Visio drawing. The existence of the hazard in the abstract set of a fault tree is reflected simultaneously in both applications.

Each component thus maintains a projection of the "phenomena of interest," and the conjunction of the projections comprises the full representation of the phenomena. Indeed, there is no representation of the phenomena other than as the conjunction of the related (possibly overlapping) partial representations in the component applications. This division of labor creates both static (data) and dynamic (behavioral) integration requirements.

First, it is necessary to integrate the separate, partial representations into a coherent representation of the overall phenomena. Because we have no explicit representation of logical phenomena (such as hazards), representing associations between the views of the phenomena requires that we link related views explicitly. Thus, for example, we will want to associate shapes in Visio drawings with records in an Access database.

Second, it is necessary to maintain system-wide coherency of this distributed representation in the face of direct user manipulation of the partial views maintained by separate tools. We have to ensure that data are consistent with each other and with the logical model intended by the user. If the user indicates the addition of a new hazard by adding a shape to a Visio drawing, then a corresponding record should be added to the database maintained by Access; and the relationship between the two must be represented somewhere. The same sorts of considerations apply to gates, trees, collections of trees, and so

forth..

Figure 2 depicts the overall integration requirements in the form of a behavioral entity-relationship (ER) model [SN92, Sul94]. The behavioral ER model represents the behavior of the system as a network of visible, independent behaviors (Visio and Access) integrated by a behavioral relationship (the arrow) that models how the applications work together. The behavioral ER model thus defines the architecture of our tool with application as components and behavioral relationships as connectors.

**A Mediator-Based Realization the Behavioral ER Model**

Since the applications were given, our task is to integrate the them by implementing the behavioral relationship. The question is how to implement it using available mechanisms. The mediator approach [SN92, Sul94] appears to be an ideal solution. The basic idea is to implement the relationship as a component which ensures that the applications work together without compromising their visibility or independence.

It is possible to do this by designing the mediator as a component that interacts with the applications through a carefully engineered combination of implicit and explicit invocations. The approach seemed natural because we had understood these mechanisms to be supported by the basic OLE technology.

Figure 3 depicts the mediator-based architecture that we posited. The mediator would register with the applications to be invoked implicitly whenever individual tool activities might require global consistency maintenance. Once invoked, the mediator would then use explicit invocation to perform its consistency maintenance tasks. Invocations were to be performed using Visual Basic's Automation capability.

As an example of the operation of this mediator, consider the processing required (a) when a user adds a hazard shape to the graphic representation maintained by Visio and (b) changes the label for a hazard using Access. The mediator would respond to an event from Visio indicating the addition of a *hazard* shape to a drawing by adding a corresponding record to the hazard table in Access and by storing in an internal table the association between the shape and the record for later reference. Notification of a change in the label of a *hazard* shape would prompt a mediator to look up the corresponding record in the table and to update its name field.

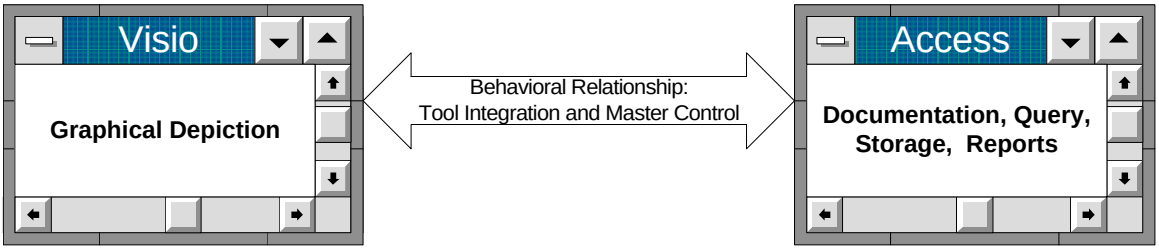


**Fig. 2. - Overall integration requirements for the tool.**

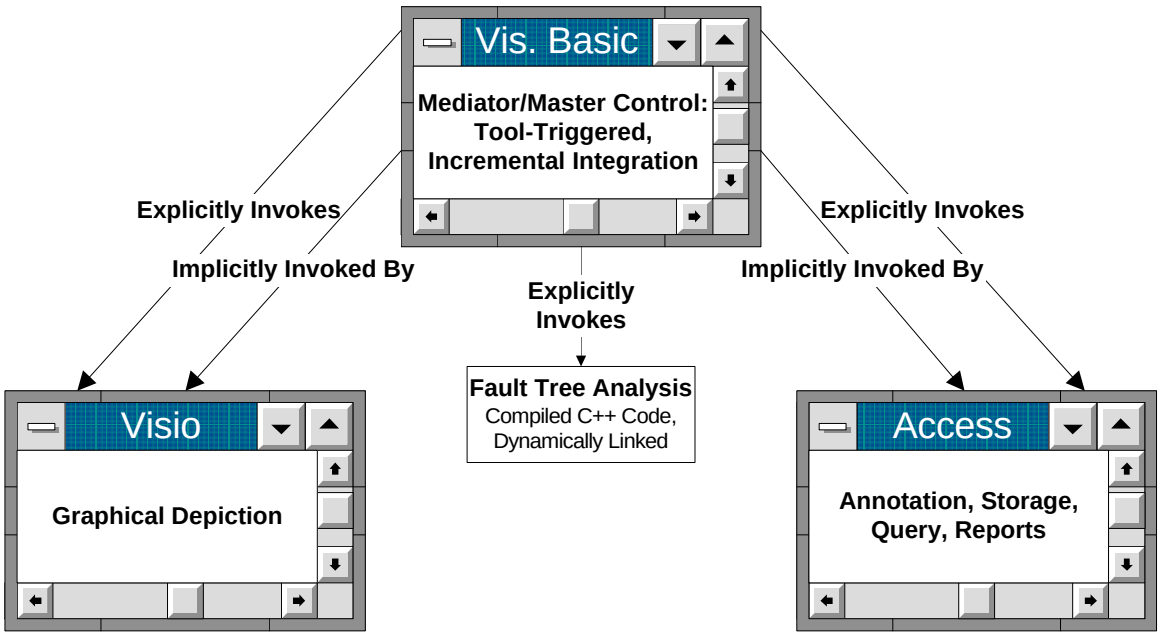


**Fig. 3. - Proposed mediator-based architecture.**

The mediator was also intended to provide its own user interface to the user. This interface presents a view of the system as a whole. It includes menu items to invoke the specialized fault-tree analysis codes, to open and close collections of fault trees in a coordinated fashion, to add and delete new trees and new collections of trees, and so forth.

### The Actual Design Reflecting Several Architectural Compromises

When we tried to implement the mediator design, we ran into several problems that compelled us to compromise our 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 using an adaptation of the idea of the reflexion model [MNS95]. We then discuss the underlying architectural problems in terms of these compromises.

A reflexion model compares two structures, highlighting where the structures agree and where they differ both by the absence of expected arcs and by the presence of unexpected ones. One structure might be a hypothesized, high-level relational model of a software system (such as an architectural diagram); the other, a low-level model derived by analysis of the actual source code. Here we use the same idea to highlight differences between our ideal mediator architecture and the system we built.

Figure 4 presents our reflexion model. Black lines and text indicate structures in the ideal architecture present in the actual system. Red indicates structures that are missing

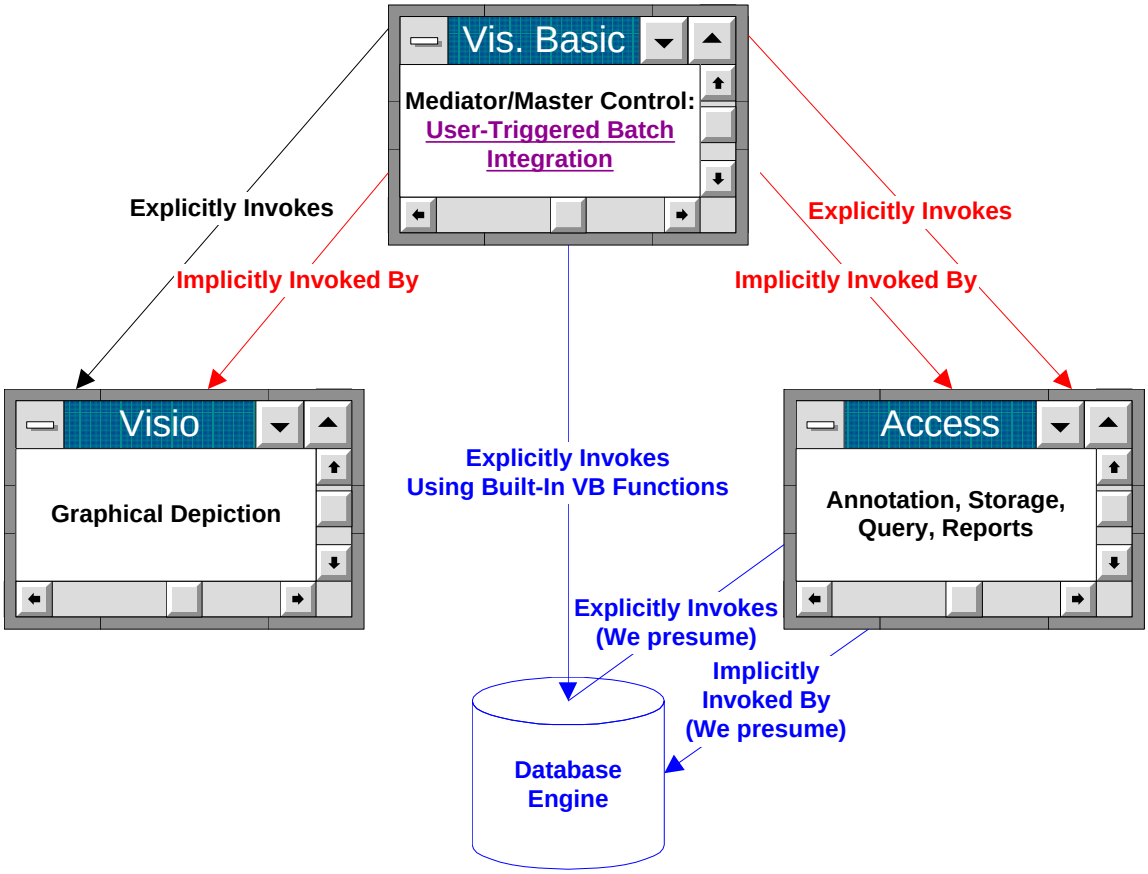


**Fig. 4. - Actual tool architecture.**

from the actual system; blue indicates unexpected structures. Purple signifies changes.

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 (explaining the red lines between the mediator and Access: the expected implicit and explicit invocation connections are missing from the architecture). We worked around this problem by using the Visual Basic language's built-in database functions, which happen to use the same underlying "database engine" as Access. Our mediator communicates with Access by writing to a shared database. Access reacts when the underlying shared database changes, we presume by using a standard implicit invocation mechanism.

The red line from our mediator to Visio is more interesting. The missing implicit invocation from Visio to Visual Basic reflects two serious problems, one with the basic OLE technology, and one with the design of the Visio interface.

†. We expect that Access 4.0, to ship 8/95, will support OLE Automation. We highlight the problem not because it is interesting in itself, but because it is one of the key classes of problems that applications architects should consider.

The more serious problem is that Visual Basic provides no support for the all-important call-back structures needed to implement implicit invocation. It does not support subprogram pointers as a programming mechanism, and does not even support explicit invocation by other components using Automation. While it 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 using OLE Automation.

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 building integrated systems. Moreover, we see no elegant 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 second problem contributing to the missing implicit invocation from the mediator is the insufficiently rich event interface provided by Visio. Even if there were a workaround that would deliver Visio events to our Visual Basic mediator, it is still the case that the events Visio announces would not be enough to support our integration requirements. Events are announced to signal that shapes have moved, have had their names changed, and that they have been selected by double clicking; but incrementally maintaining the consistency of the set of records with the set of shapes in a picture would additionally require events signalling shape deletions.

This combination of shortcomings 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 corresponding set of records in the database; and appropriate additions, deletions and modifications are made to the database records.

A final interesting problem that we had was lack of unique identifiers for Visio shapes. Visio assigns each shape an identification number, but the number is guaranteed to be unique only on a per-page basis, and not across time. This makes the numbers unsuitable as keys for representing associations between Visio shapes and database records. Our solution exploited a very useful 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 the 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 some "serious" architectural impediments, we were still able to build a 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.

# RESULTS

The results we obtained from this activity fall broadly into two areas. One area is the relatively successful performance of the effort—a very useful tool was built very quickly. The second area is architectural assessment. We uncovered a number of subtle difficulties with both the OLE architecture and the application components. In the remainder of these sections we discuss the specifics of these two areas.

## Performance

- *A very high level of productivity achieved.*

  With about a person week of effort, a tool that met virtually all the requirements outlined above was developed. Essentially all of the requirements were met using existing capabilities of the application components. A small amount of software was written to implement the overall control mechanism, and, of course, the sophisticated analysis components were developed from scratch. However, we note that the entire analysis package is less than 10,000 lines yet it is available within a framework of functionality that is implemented by probably several million lines of code.

- *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 have 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 were 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 Windows™, 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 (by definition) 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 event

interface, for example, is not sufficiently rich to support eager incremental consistence maintenance. For example, there are no events to signal impending object deletions and 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, does not support OLE automation at all in the version available to us. The result was that Access had to be integrated into the tool using a completely different mechanism. It was fortunate that we were able to work around the problem in this application but this is not going to be possible in every case.

- *Inconsistent OLE support.*

  The ability to integrate tools rapidly using an interpreted scripting language that includes a reasonable programming environment is very valuable. However, we found a serious difficulty with the instantiation of this concept that we used (Visual Basic). The difficulty was that while Visual Basic is an excellent OLE Automation client, it cannot be used as an OLE Automation server. Thus it cannot be used as a mediator to respond to events being passed back from application components. It is very hard (essentially impossible) to work around this difficulty because Visual Basic does not support pointers to subprograms in any sense. This forced a batch consistency model to be implemented rather than other preferred approaches (such as incremental).

- *Inconsistent object naming.*

  Visio makes it harder than necessary to name consistent 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 very powerful pieces of software. In some cases, we needed to disable certain functions because they either were not necessary or because they provided ways of circumventing essential functions such as consistency maintenance. 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 many of the components so as to permit our tool to be "distinguished" from its component parts. 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 available is the software itself. We have no documentation, no users' guide, and so on. In terms of the lifecycle cost of software development, it is not clear how much of an advantage we have achieved.

# CONCLUSION

In the work described in this paper, we have evaluated the approach to large-scale systematic reuse represented by OLE. We conclude that it is an extremely powerful reuse-based software development approach; although difficulties remain, such an approach is practical now in many domains; it substantially overcomes the architectural impediments that have hindered some previous attempts at large-scale reuse; and it represents significant progress towards realizing the promise of rapid development of sophisticated systems.

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 OLE 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.

The lesson here 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. In addition, 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 a modestly ambitious application. This raises the issue of whether the large-scale reuse technology that we used will scale up. We hypothesize that it will given: (a) that a wide variety of OLE-compliant application components either exist or could be developed; (b) that OLE supports the implicit and explicit invocation mechanisms needed for rich tool integration; and (c) that in addition OLE supports a suite of other object services, services that we did not evaluate in this paper.

# 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.

# REFERENCES

[BP89] Biggerstaff, T.J. and A.J. Perlis, *Software Reusability,* ACM Press, 1989.

[Bro95]Brockschmidt, K., *Inside OLE, Second Edition,* Microsoft Press, Redmond WA, 1995.

[BS92] Boehm, B and W. Scherlis, "Megaprogramming", *Proceedings of Software Technology Conference, DARPA*, ARPA, 1992.

[DD95] Doyle, S.A., and J.B. Dugan, "Dependability assessment using binary decision diagrams (BDDs)", Proceedings of FTCS-25, *Twenty-Fifth International Symposium on Fault-tolerant Computing*, Pasadena, CA June 1995.

[GAO95] Garlan, D., R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts", *Proceedings of ICSE 17, Seventeenth International Conference on Software Engineering*, Seattle, WA, June 1995.

[GHJV94] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Micro-Architectures for Reusable, Object-Oriented Design*, Addison-Wesley, 1994.

[GS93] Garlan, D. and M. Shaw, "An introduction to software architecture", in V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume 1*, New Jersey, 1993, World Scientific Publishing Company.

[MNS95] Murphy, G.C., D. Notkin, and K.J. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models", *Proceedings of the SIGSOFT Conference on the Foundations of Software Engineering*, Washington, DC, October 1995.

[PLG] RISKMAN: PRA Workstation Software, PLG Inc.

[PW92] Perry, D.E. and A.L. Wolf, "Foundations for the study of software architecture", *ACM SIGSOFT Software Engineering Notes*, 17(4), pp 40-52, October 1992.

[SN92] Sullivan, K.J., and D. Notkin, "Reconciling environment integration and software evolution", *ACM Trans. on Software Engineering and Methodology,* 1(3), July 1992.

[Sul94] Sullivan, K.J., "Mediators: Easing the design and evolution of integrated systems", Ph.D. dissertation and technical report CSE-TR 94-08-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1994.