

THE ROLE OF ACTIVEX AND COM IN ATE

Kirk G. Fertitta
Vektrex Electronic Systems
8675 Miralani Drive
San Diego, CA 92126
619-578-6787
kfertitta@vektrex.com

John M. Harvey
Hewlett-Packard Company
815 14th Street S.W.
Loveland, CO 80537
970-679-3535
John_M_Harvey@hp.com

Abstract -- The Component Object Model (COM) is an object-oriented standard for reusable binary components. COM reusability is based not only on component reuse, but on interface reuse. Client programs access COM objects only through defined interfaces, which may be reused from object to object. Interfaces hide implementation details (encapsulation). Interface reuse and inheritance provide runtime polymorphism. Interfaces can be precisely described with type libraries to development environments, and can be accessed remotely with little additional effort.

COM technology has incredible potential for ATE. This paper focuses on ways that COM will be used to transform instrument driver technology, and on the resulting benefits for test systems in general. Runtime polymorphism supports syntactical interchangeability. The IVI Foundation is currently pursuing technologies that promise to deliver this level of interchangeability to the ATE market. COM will also provide natural solutions for driver deployment/integration, remote access, and a variety of other benefits as well. Ultimately, COM will be used "in the box" to combine the functionality of driver and firmware.

COM PRIMER

Effective Code Reuse

When it comes to component reuse, the software world has achieved nothing close to the level of success the hardware world has enjoyed for decades. Software reuse has become one of the most important topics in the software industry. Fueled by rising software development costs, increasingly complex software requirements, and widely disparate usage models, companies have committed vast resources to address software reuse issues. C++ did provide increased source-code reuse, but this has proven to

be inadequate for large-scale reuse and broad-based distribution. Effective source-code reuse is often feasible within a single project, but even then requires significant forethought and up-front design. With the notable exception of well-established source libraries like the Microsoft Foundation Classes (MFC) and the Standard Template Library (STL), effective reuse between companies or large organizations is quite rare.

Distribution standards such as Windows dynamic link libraries (DLLs) provide relief from some of the problems in source-code level reuse models, but other difficulties arise. Compiler vendors often elect to implement language features in proprietary manners that render modules "untouchable" by code generated by any other compilers. C++ exception handling is an excellent example of such a feature. A C++ exception thrown from a function compiled with one compiler cannot be reliably caught by client code compiled with a different compiler. Finally, hardware platform dependencies like fundamental data representation and organization make seamless communication between code modules on different platforms extremely difficult.

All of these problems point to the lack of a *binary* standard for component interaction.

COM Basics

COM is a specification that describes a standard for binary, executable software objects. COM is *not* a computer programming language, nor is COM a development environment. COM components can be developed in many languages and on any hardware platform, as long as the component is compiled as a COM binary object. COM is not a C++ class library

like MFC. Similarly, COM is also not primarily an application programming interface (API) or set of functions like the Win32 API. The Win32 API itself is not required to develop COM components, and though there is a COM library that provides essential COM services, it occupies a small part of the larger COM picture. Finally, COM does not compete with or replace DLLs. DLLs are a distribution standard that house COM's binary components.

The vision of COM is that complete applications can be built of reusable binary components, each potentially from different software vendors using different compilers, languages, and/or different hardware platforms. Developers using COM components from outside sources are shielded from the implementation details of the component provider.

Benefits of COM

COM provides a number of very compelling benefits:

- **Language Independence.** COM is a binary standard, and, as such, allows any combination of language and compiler to be employed.
- **Interoperability.** Components from different software vendors can interoperate safely.
- **Location Transparency.** COM and DCOM allow clients to communicate with components without regard to whether those components are running in the same process, in a different process, or even on another computer.
- **Robust Versioning.** COM allows server components a great deal of flexibility to change and evolve. At the same time, COM restricts the ways that COM components can change to protect client programs from breaking when new components are introduced into a system.
- **Security.** COM and DCOM provide several types of security for authentication of potential clients, authorization for specific component operations, data integrity for protecting data in transit from modification, and data privacy.

COM Interfaces

Interface Principles

COM achieves its reuse goals by strict separation of interfaces and implementation. In COM, interfaces are everything. In C++, good designs often separate interfaces from implementation, but most designs do not. By contrast, COM enforces this separation.

A COM interface is simply a collection of method prototypes. These methods represent functionality that a COM component implements and that client programs can use. Clients communicate with COM components *only* through the component's interfaces. This achieves an important feature of object-oriented designs – *encapsulation*. With only knowledge of a component's interfaces, the client need not worry about any implementation details associated with the component. As long as the interface remains the same (and COM rigidly mandates this), then the implementation can change radically without breaking the client program. New interfaces to COM components can be introduced and have no adverse effects on a client application as long as the previous interfaces are still supported.

Interface definitions include syntax and semantics. The syntax is captured in Interface Definition Language (IDL), and is unambiguous. Semantics describe what each method does. There is no standard mechanism for defining the semantics of a definition, but this should not be an excuse for ignoring them, or doing a poor job of defining them.

Physically, a COM interface is a specific binary structure of memory. The precise layout of this structure is part of the COM specification – indeed, it is the heart of the specification. This structure contains an array of function pointers (called a vtable), where each function pointer contains the address of a function implemented by the COM component. Specifying a binary interface standard is what allows COM components to be language independent. In C++, it happens that the binary structure for interfaces specified by COM is precisely the layout produced by compilers when processing abstract C++ class definitions. This natural affinity is part of the reason C++ is the overwhelming language of choice for COM component development. However, C++ is by no means the only practical language for COM component development. As long as a compiler can produce this simple binary layout, that compiler can be used to develop COM components.

Polymorphism

COM objects may implement more than one interface. When a COM object exposes more than one interface, it is polymorphic. For instance, a stereo system COM object might have a radio interface, a CD interface, and an amplifier interface all in one object. To treat the stereo system like a radio, the client program uses the radio interface. To treat it like a CD player, the client program uses the CD interface. In fact, because the implementation is hidden, the client program does not know how the stereo system is implemented. It could be inextricably mixed together in one “boom box”, or it could be three distinct components, one for each interface. [1]

COM and Inheritance

One common criticism of COM is that it does not support inheritance. While it is true that it does not support *implementation inheritance*, it does support *interface inheritance*. Implementation inheritance allows one object to inherit code from a parent object. A method executed on a child object actually invokes the code of the parent object’s method. With interface inheritance, however, the child interface (not object!) only inherits the methods of the parent interface. It is up to the class that implements the child to provide the implementation code for handling any method calls it receives.

C++ supports implementation inheritance. COM, however, *intentionally* supports only interface inheritance. Remember that in COM, the entire functionality “contract” between server and client is in the interface definition. By definition, there is nothing in the interface definition that is implementation specific, and that is why COM can successfully focus solely on interface inheritance [2]. Inheriting a well-defined interface from a parent interface ensures that a client program can still use the interface when a new implementation is provided by the server class.

There are situations where something like implementation inheritance is desirable. Developers can still leverage existing “parent” classes into “child” classes. To accomplish this, COM supports *containment* and *aggregation*. Both mechanisms represent binary reuse, as opposed to the classic source level reuse of implementation inheritance. With containment, a COM object simply delegates method calls to another object that provides the implementation of the method. To the client program, the COM object appears to support the desired interface methods, even though the COM object does

not directly expose the interfaces of the contained object (e.g. the one providing the implementation). By contrast, aggregation allows one COM object to present the interfaces of another COM object as if they were its own. Aggregation is more involved than containment, but both provide effective mechanisms to build on existing COM objects.

Types of COM Interfaces

COM interfaces come in three varieties: 1) COM custom interfaces, 2) automation interfaces, and 3) dual interfaces. Each interface type has unique characteristics with regard to performance, supported data types, and ease-of-use.

COM Custom Interfaces

Custom interfaces use the vtable mechanism described in the previous section to directly access all interface methods. Custom interfaces can offer the highest performance and greatest flexibility. The performance of custom interface method calls varies greatly depending upon whether the COM component providing the interface resides in the client’s address space (in-process server), in another process’s address space (out-of-process server), or on another machine (remote server).

When a client program calls a member function of a custom interface provided by an in-process server, the client is actually performing a simple, efficient indirect function call. The call overhead associated with an in-process method is of the same order of magnitude as a local function call. For out-of-process method calls, the performance suffers significantly. An “empty” out-of-process method call (i.e. one with no parameters) executes 1000 times slower than the same in-process method call. When passing parameters across process boundaries, the method COM uses to package and transport method parameters (termed *marshalling*) introduces additional overhead that will increase the method invocation time as well. Method calls performed on remote COM servers execute slower still, being subject to the limitations of the physical network, the transport protocol in use, and the network traffic at the time of the call.

Custom interfaces provide the greatest flexibility in data types. Virtually any data type that can be represented in C can be passed as a parameter to a custom interface. Arbitrarily complex user-defined data types can be supported by custom interfaces.

Automation Interfaces

Custom interfaces require that the client environment support function pointers that can be resolved at compile time. For interpreted languages, macro languages and scripting environments, this is not practical. To service these important and popular use cases, COM defines a standard interface for calling a component's methods indirectly. This interface is called *IDispatch*, and interfaces that derive from *IDispatch* to implement the set of functions defined by the standard are collectively referred to as *dispinterfaces*. Client calls to *IDispatch* methods are directly through the *vtable*, but calls to the other methods in the interface are made indirectly via the *IDispatch Invoke* method.

While automation interfaces currently allow access from the largest pool of client programs, this comes at a price. The call overhead of an in-process automation interface method is approximately 100 times slower than a similar custom interface method. For out-of-process method calls, the performance difference becomes less important. For remote method calls, the increased call overhead of an automation interface becomes negligible since the network communication involved in the call is by far the most time-consuming portion of the operation.

Automation interfaces also have limited data types that can be passed as parameters to methods. Automation clients can only pass data types that can be represented in a standard data structure known as a *variant*. A variant is simply a large C-language style union that can house various integer and floating-point data, strings, and arrays thereof. Thus, a user-defined data structure, such as a C-language *struct*, can not be passed as a parameter to an automation interface method. Aside from reduced flexibility, an additional consequence of this is that automation interfaces are awkward to use in C++ client programs. Automation interfaces are easily called from scripting clients and other languages, such as Visual Basic, but custom interfaces are much more natural in a C++ development environment.

Use of automation interfaces implies one interface per COM class -- a significant restriction compared to custom interfaces. COM classes can have only one implementation of the *IDispatch* methods, and the implementations are interface dependent. As a result, multiple automation interfaces must be implemented in multiple COM classes. These classes may be tied together with references from one to the other -- this is standard technique in applications such

as Microsoft Excel and Microsoft Word. The resulting polymorphism is not as clean as it is when a single class implements multiple custom interfaces, but the net results are very similar.

Dual Interfaces

A dual interface provides a compromise between custom interfaces and automation interfaces. Dual interfaces make their methods available through both *IDispatch* (as in pure automation interfaces) as well as through direct function pointers (as in custom interfaces). This allows clients that support custom interfaces to make interface method calls with execution speeds similar to custom interface method calls. It also allows pure automation clients like VBScript to access the methods of the interface. One drawback of the dual interface is that it has the same limitation on data types that pure automation interfaces have. Specifically, dual interfaces only support parameter data types that can be contained in a variant. In spite of this limitation, dual interfaces have become the standard choice for developers seeking to make a single COM component that is both available to automation clients and efficient for *vtable* use.

Type Libraries

Type libraries provide a computer readable description of the interfaces associated with a COM class. Type libraries may be packaged with the COM classes in the class DLL or OCX file, or they may be delivered separately. Many programming tools can read type libraries to discover class and interface definitions. This information can then be presented to the user (as in the Object Browser and OLE Viewer) or used to assist programmers (as with IntelliSense).

ActiveX and COM

In recent years, ActiveX technology has become one of the most pervasive software technologies in the world. Understandably, this has given rise to a great deal of confusion. At present, ActiveX and its relationship to COM is one of the most commonly misunderstood topics in the software industry. This section will attempt to provide a brief clarification.

ActiveX actually refers to an entire suite of technologies built on top of COM. ActiveX defines a large number of standard interfaces for building interactive components. The most important ActiveX

technologies include ActiveX automation, ActiveX controls, ActiveX Documents, and ActiveX Scripting. In spite of their common ActiveX nomenclature, these are all very different technologies with very distinct roles in the universe of software development. However, the underlying framework for all of these technologies is, indeed, COM.

ActiveX Automation

ActiveX automation (or simply *automation*) refers to the standard mechanism by which one application can be controlled by another application. An automation server exposes its functionality through an automation interface. An automation client accesses interfaces on an automation server to control the server application. For instance, a LabVIEW application can programmatically create a Microsoft Outlook task request, fill in the fields on the task request form, address the recipient, and send the task request. In this scenario, LabVIEW is the automation client and Outlook is the automation server. ActiveX automation provides the standard COM interfaces by which this is accomplished.

ActiveX Controls

An ActiveX control is a COM object that supports a customizable, programmatic interface. An ActiveX control is a COM object, plus a large number of additional standard interfaces for implementing the interactive capabilities required of client applications that have modern graphical user interface (GUI) elements. ActiveX defines standard interfaces for such things as property pages, user events, as well as non-GUI interfaces for security and run-time licensing.

Historically, ActiveX controls evolved from OLE controls. OLE controls were an attractive component technology based on COM, but the size of a typical OLE control made it impractical for use in Internet applications. With the exploding popularity of the Internet, the demands for lightweight reusable components spurred Microsoft to trim the OLE control specification and coin the term ActiveX control.

ActiveX Documents and ActiveX Scripting

ActiveX Documents and ActiveX Scripting are both technologies targeted at Internet applications and browsers. ActiveX Documents build on OLE compound document technology to allow one type of application document to be embedded in another application. Though the distinction between OLE controls and ActiveX controls has disappeared, OLE

Documents remain distinct from ActiveX Documents. In fact, the term OLE has now been relegated to refer exclusively to compound document technology. The most common example of ActiveX Document technology is Internet Explorer, where various types of non-HTML documents can be opened directly from within the browser. ActiveX Scripting allows users to add script commands in any scripting language to an application, such as Internet Explorer. ActiveX Scripting allows script hosts to invoke scripting services with script engines. The hosts and engines can be from different software vendors and can implement different languages, since COM handles the “plumbing” between the two.

Distributed COM (DCOM)

Distributed COM (DCOM) extends the COM specification to allow clients and servers to communicate over a network transparently. DCOM delivers the benefit of COM that is termed *location transparency*. DCOM provides features that allow clients and components to interact without either having any knowledge of the physical location of the other. When client and component reside on different machines, DCOM merely replaces the local interprocess communication with a network protocol. This occurs automatically, completely unbeknownst to both the client and the component. DCOM further manages the connection between a component and a client, monitoring for broken connections and automatically managing the associated resources. In addition, DCOM treats the network transport as an abstraction, so that any network protocol can be used without requiring any code changes to either the component or the client.

Independent of the transport protocol used, DCOM provides a security mechanism. DCOM can make distributed applications secure without any security-specific coding in either client application code or component code. Just as DCOM hides the physical location of a component, it hides the security requirements of the component as well. Security settings can be configured on a component-wide basis by developers and administrators in the registry, or they may be configured programmatically for greater flexibility and finer control over component access.

COM Terminology

This section briefly covers some standard terminology to clarify discussions of COM topics.

ActiveX – a suite of technologies built on top of COM with additional standard interfaces for building interactive components.

ActiveX Automation – COM-based standard for controlling one application programmatically from within another application.

Automation Dispatch Interface – an interface that derives from IDispatch. You call methods in a component's interface through IDispatch.

Automation Dual Interface – an interface that derives from IDispatch, but which is defined in such a way that it may be called either through the vtable or by using IDispatch.

ActiveX Controls – COM object that supports a customizable, programmatic interface.

COM Class – an implementation of one or more COM interfaces.

COM Component – a piece of compiled code that provides some services to a client application or module.

COM Custom Interface – a COM interface that is accessed through the vtable. Custom interfaces can support any legal IDL data type.

COM Object – an instance of a COM class.

COM Interface – a collection of logically related methods that express a single functionality.

COM Library – a standard API that provides run-time services for managing COM components.

Distributed COM – an extension of the COM specification that allows COM clients and servers to communicate over a network. DCOM is the portion of COM that provides location transparency, so that neither client nor server need be aware of the other's location to successfully communicate.

Interface Description Language (IDL) – platform-independent language used for defining COM interfaces.

Globally Unique Identifiers (GUIDs) – 128-bit integers used by COM to uniquely identify every COM interface and every COM component class. These identifiers are UUIDs (universally unique identifiers) as defined by the Open Software Foundation's

Distributed Computing Environment. GUIDs are guaranteed to be unique in the world across both space and time.

In-process Server – a COM component that resides in the same process as the client.

Method – a single function of a COM interface.

OLE – COM-based compound document technology.

OLE Automation – synonymous with ActiveX automation.

Out-of-process Server – a COM component that resides in a different process than the client.

Remote Server – a COM component that resides on a different machine than the client.

Server – in the context of COM, a generic term for a COM component. An application or code module that uses a COM component's interfaces is a client and the COM component it uses is the server.

Type Library – Computer readable type information for COM classes and interfaces that allows COM components to be self-describing.

Variant – standard data structure used by automation clients and servers to exchange data.

Virtual Function Table (vtable) – data structure containing pointers to the methods of a COM component.

ATE BACKGROUND

Beginnings of Automated Test

There are two paradigms for human-instrument interaction. One is controlling an instrument using a physical front panel. The other is controlling an instrument using a software interface accessible via a standard I/O mechanism.

While front panel control is necessary for many tasks, software control provides speed, repeatability, and the ability to operate remotely. Software control enables timely coordination of multiple instruments in a test system and unattended test execution.

The first "connected" instruments appeared on the market in the early sixties[3]. Early on there was little

concern for ATE standardization. I/O was implemented inconsistently. There were no API standards, and control was either via conventional programming languages or dedicated instrument controllers with proprietary languages.

In 1975, HP's HP-IB interface was standardized as IEEE 488.1, commonly known as GPIB. This was a major milestone in ATE. GPIB provided needed performance and features tuned to use with instruments. GPIB instruments implemented an API of ASCII string commands. The companion software standard, IEEE 488.2, was approved in 1985. IEEE 488.2 standardized some commands and described a method for forming instrument command mnemonics. The SCPI Consortium, founded in 1990, took on the job of standardizing the way that ASCII APIs were created for specific instruments.

This standardization has served the industry well. GPIB continues to be successful with cards available from a variety of vendors for a variety of platforms, all with accompanying software libraries. It also set the precedent for the industry. No test and measurement I/O protocol since has broadly succeeded without some level of industry standardization.

Automated Test Environments

The first automated test environments were dedicated instrument controllers. These controllers provided both a development environment and an execution environment for test systems. They implemented proprietary languages for accessing instruments. For example, Hewlett-Packard controllers implemented several proprietary variations of Basic attuned to instrument I/O. As these controllers moved to more standards-based implementations, it became easier to use them with a variety of instruments designed to the standards, irrespective of vendor.

In general, the move to standards for instrument I/O and command sets benefited developers of systems. Standards enabled predictable I/O and instrument behavior. Reusable software could be written to take advantage of this predictable behavior, and could be used industry wide.

The advent of desktop computers (both PCs and Unix workstations) made it possible to implement such environments on standard computer platforms. HP leveraged its experience with Basic into Rocky Mountain Basic. Other popular automated test development/execution environments include HP-VEE

and National Instrument's LabWindows/CVI and LabVIEW.

Some of these environments were really test-specific extensions to standard programming idioms. However, the electrical engineers and technicians who best understood the problem domain were used to thinking in terms of electrical schematics.

The idea of software ICs had been advanced in the software community a number of years before, and was one motivation for the move to object oriented programming. As windowing systems enabled the development of sophisticated GUIs, people also began to play with graphical programming. Graphical programming has seen limited success in the general software development market and is generally not considered suitable for large-scale development. However, the graphical programming paradigm has become pervasive in the ATE world, as illustrated by the success of National Instrument's LabVIEW and HP VEE. In both environments it is easy to compare the graphical elements of a program to ICs, and the data flow between them to wiring. The resulting application code looks very much like a schematic.

Although there was some prior demand for instrument drivers (e.g. HP's ITG drivers), the idea of an instrument driver flourished in the world of graphical languages. Indeed, it was necessary to represent instrument functionality as graphical component(s). Instrument operations were abstracted into one or more graphical components, and the collection became the instrument driver. The most important limitation of these drivers was that VEE drivers only worked in VEE and LabVIEW drivers only worked in LabVIEW.

Despite the popularity of VEE, LabVIEW, and other automation test development environments, many test system developers prefer general-purpose programming environments. This is particularly true for large, complex systems. C, C++, and, increasingly, Visual Basic are popular development environments for ATE and are supported by a variety of tools and standards.

VXI and VXIplug&play Drivers

The VXI standard promised dramatic performance improvements by connecting instruments and controller(s) over a high-speed backplane. Additionally, the standard promised decreased cost and a more compact form factor by eliminating the physical front panel in favor of a software interface.

VEE and LabVIEW had already shown how instruments could be controlled from a software GUI. VXI vendors quickly found that customers expected both VEE drivers and LabVIEW drivers for their VXI products. In addition, since VXI could only be controlled through software, people expected the convenience of drivers in other development environments as well – chiefly C, LabWindows/CVI, C++, and Visual Basic.

Drivers

The VXI Systems Alliance took on the task of creating a driver standard, *VXIplug&play*, that would allow a single driver to work with a variety of I/O protocols and programming environments. *VXIplug&play* drivers are developed in G (a proprietary LabVIEW language) or ANSI C and can be used in multiple development environments (C, C++, LabWindows/CVI, Visual Basic, LabVIEW, and HP VEE), on multiple platforms (Windows 3.x, Windows 95/98/NT, Sun Solaris, and HP-UX).

It is important to emphasize, though, that one standard does not mean one driver. The standard describes several C driver frameworks for Windows 3.1, Win32 (Windows 95, 98, and NT), HP-UX, and Sun Solaris. Developers must deliver drivers for each of these frameworks to get full platform coverage, though most of the C code for a specific instrument's driver can be leveraged across frameworks. Of these frameworks, Win32 is by far the current favorite, and in many cases the only driver delivered.

The driver executable is a function library consisting of seven standard functions and any number of additional driver-specific functions. Aside from the seven required functions, function syntax and semantics are not specified. All functions are prefixed with a driver-specific string to insure that function names are unique across drivers. All functions return a status code and include a parameter that identifies the VISA I/O session.

Source code, help, a function panel file, and a Visual Basic header file (for Win32) are delivered with the driver.

I/O

The *VXIplug&play* I/O standard, VISA, specifies an API capable of handling GPIB, RS-232, and VXI I/O. VISA implementations are specific to a particular vendor's I/O hardware. For instance, if you are using an HP HP-IB card, you must have an HP VISA library.

However, with the correct VISA library installed, you can use a compliant *VXIplug&play* driver to control an instrument, regardless of the driver vendor.

Issues

While *VXIplug&play* provided advances on several fronts, it also had significant shortcomings:

1. **Syntax & Semantics.** A specification of syntax and semantics like the one provided by SCPI for instrument APIs was totally missing from the *VXIplug&play* standard. Driver users can not reliably leverage knowledge of function syntax from one driver to the next.
2. **Reuse.** There is virtually no reuse with *VXIplug&play* instrument drivers.
3. **Quality.** Users complain that *VXIplug&play* drivers are buggy and incomplete, and, in general, that they require source to address both problems.
4. **General Functionality.** *VXIplug&play* does not say anything about how to implement driver functions. Drivers vary widely with respect to efficiency, thread safety, error handling, and other features.
5. **Application Development Environment (ADE) Fit.** C framework *VXIplug&play* drivers work in Visual Basic, LabVIEW, and HP VEE, but they don't really look like they belong, and are difficult to use.
6. **Remote Access.** You can't directly access a *VXIplug&play* driver remotely.
7. **I/O.** The VISA standard must be changed to accommodate new I/O protocols such as USB and IEEE 1394. VISA libraries from different vendors cannot coexist on the same PC, and there are also slight functional inconsistencies between them.

IVI Instrument Drivers

The IVI (Interchangeable Virtual Instruments) Foundation was publicly announced in the summer of 1998, and membership opened to any interested company. It proposed creating a new driver standard to address some of the shortcomings of *VXIplug&play* drivers.

At the time of writing (May 1999), the IVI Foundation has voted and approved five instrument class specifications (see below for details). With membership opened to instrument vendors, these specifications are undergoing some revision. Architecture and general driver requirement specifications are in process, as are a few additional instrument class specs [4]. At this time, the IVI specification is very much a work in process, and details below are subject to change.

Instrument Syntax

The IVI Foundation's ongoing work centers around specifying APIs for several *instrument classes*. An instrument class defines a generalization of a particular type of instrument. To date IVI specifications have been published for digital multi-meters[5], oscilloscopes[6], power supplies[7], switches[8], and function generators[9].

Instrument class specifications break instrument functionality into *capability groups*. Each class includes one *fundamental capability group* that defines the minimum functionality that an instrument driver must implement to be class compliant. Each class also includes several optional *extension groups*. As the name implies, these groups extend the fundamental capabilities. Though support for an extension group is, by definition, optional for a particular instrument driver, once the driver is published as supporting an extension group, it must support all functions in that group. Some classes require drivers to implement the fundamental capability groups and one extension from a set [10].

Class specifications define functions and attributes, which usually reflect instrument state variables. Functions are defined using ANSI C prototypes, as in the C-based *VXIplug&play* frameworks. One requirement of IVI function names is that they have a driver-specific prefix to make them unique. To illustrate the need for the prefix, consider two instrument drivers, one for a power supply and one for a DMM, being used in the same test program. Both drivers have a `reset()` function. When a C compiler sees

```
reset();
```

in the test program, what does it do? It has two `reset()` functions to choose from, and it cannot resolve the ambiguity. Prefixes are used to resolve the ambiguity, and the test programmer can specify exactly the function he or she requires, for example:

```
dmm_reset();  
or  
power_reset();
```

Class Drivers

At first glance, it appears that instrument-specific drivers (or *specific drivers* in IVI terminology) should be polymorphic. That is, if you only use class-defined functions in one specific driver, you should be able to swap in another specific driver that implements the same class functions without changing your test program code. Unfortunately, because of function prefixes, all test program code written using one specific driver must be rewritten to use a different specific driver.

IVI *class drivers* are used to resolve this naming dilemma. Class drivers implement all of the class-defined functions, which are prefixed with a class-defined prefix, for example `dmm_` for the DMM class). Based on configuration information, the class driver dynamically loads a specific driver when the `init()` function is called and maps its functions to whatever class-defined functions are implemented by the specific driver.

Interchangeability

The combined use of class drivers with specific drivers results in a "simulated" polymorphism. That is, the specific driver API and the class driver API can be exposed, backed by an implementation in the specific driver. This polymorphic behavior is the core of IVI interchangeability. However, it has some limitations. For instance, class drivers must implement the entire class definition, whereas specific drivers need not implement optional extension groups. When a test program uses a class function that is not included in the specific driver being used, the class driver must return an appropriate error.

Note that the function syntax for extension groups is specified just as precisely as functions in the fundamental capability group. The polymorphism of an extension group, if present at all, is at the same level as that for the fundamental capabilities.

Some limitations are subtler. For instance, class drivers must support the widest possible ranges for instrument state variables. Specific drivers, which support specific instruments, will likely operate over narrower ranges, with the exact range varying from instrument to instrument. Test programmers cannot substitute one instrument for another without first

verifying that both are capable of performing the desired measurement.

The fact that two instruments are theoretically capable of performing the same measurement does not automatically mean that a test program that works with one of the instruments will work with the other. For example, the default setup of the instruments may be different. If the test program does not fully specify the state of the instrument, the differences in default setup may affect the results. This is a simple example of how problems related to instrument state can affect measurement results.

In this context, it is clear that the polymorphic behavior of the class driver provides *syntactical interchangeability*, that is, the ability to use two different specific drivers in one test program without recompiling code. It does not provide *functional equivalence* – that is, a guarantee that the two specific drivers will provide the same result.

Several features are being discussed in the IVI Foundation to support interchangeability. *Interchangeability checking* warns test programmers when relevant state variables do not have values that were explicitly set by the test program. Class drivers set state variables in unused extension groups to default values that simulate the behavior of instruments without the extension group at all.

Note that even with good class definitions, class drivers, and all of the available interchangeability features, subtle differences in the way instruments work may prevent functional equivalence. There is no substitute for proper testing of test systems with all target instruments against the expected range of test conditions.

Finally, the IVI Foundation is considering the concept of Measurement Subsystems Architecture technology as a means of guaranteeing interchangeability at the measurement level. The authors refer the reader to previous papers on the subject [11].

Other Features

At the present time, there are only five IVI instrument classes published. For some types of instruments, class specifications may take considerable time to develop, and for others there may never be a class specification at all. There is no class driver or interchangeability without an instrument class. However, even in the absence of a class specification IVI drivers will still add considerable value. The

following features are being discussed as potential IVI features.

1. **Simulation.** Test programmers want to be able to develop and test software without having the instrument attached. While a simulator will never substitute for the instrument itself, different levels of simulation are possible and helpful.
2. **Thread-Safety.** Multiple threads in a single process should be able to safely access the driver.
3. **Range Checking.** Optional range checking in the driver may save I/O to the instrument in cases where the range checks fail. However, care must be taken with range checking, as it is often affected by instrument state.
4. **State Caching.** Optional caching of the instrument state in the driver may save I/O to the instrument in cases where the driver attempts to set an instrument state variable to a value to which it is already set. State caching is difficult to do correctly when state variables are *coupled*, particularly in large instruments with complex couplings. Coupling occurs when changes to one state variable result in changes to one or more related state variables. Firmware experts understand the couplings, and driver writers may not have the benefit of the firmware engineer's knowledge.
5. **Deferred Updates.** In normal operation, the driver will send commands to the instrument one-by-one as the driver executes. However, the driver may queue up ASCII commands to send to an instrument. This is an optional, internal feature.
6. **Instrument Status Checking.** Application developers can choose whether the driver will query the instrument for its status after each operation. The tradeoff is between good status information (status checking turned on) and good performance. While asynchronous event notification may be possible in some cases, instrument status checking can be used more generally.
7. **Channels.** IVI specifications can define one repeated instrument feature as a channel. The obvious application is to features that have historically been called channels (such as oscilloscope channels), but other repeated features could be used as well.

Delivery

The IVI C-based driver installation will look similar to the VXI*plug&play* package. In addition to the DLL, the package includes source code, help, a function panel file, and a Visual Basic header file (for Win32).

Issues

C-based IVI represents an improvement on VXI*plug&play*. To compare C-based IVI drivers with C-based VXI*plug&play* drivers, consider the list of VXI*plug&play* issues:

1. **Syntax & Semantics.** IVI Class specifications allow syntactical interchangeability between specific drivers that support the same extension groups.
2. **Reuse.** IVI class definitions and class drivers provide a higher level of reuse than VXI*plug&play*. However, reuse comes at the cost of architectural complexity.
3. **Quality.** There is not enough history with C-based IVI drivers to accumulate empirical data about quality. However, IVI class specifications provide minimum standards for completeness. Other IVI features may be considered either standard, or standard approaches to optional features. On the other hand, C-based IVI drivers are currently more complex than VXI*plug&play*, and some features such as range checking and state caching may be error prone for complex instruments.
4. **General Functionality.** IVI does attempt to mandate some functionality. Though details may change, drivers should be more predictable with respect to efficiency, thread safety, error handling, and other features.
5. **ADE Fit.** C-based IVI drivers look just like VXI*plug&play* drivers in Visual Basic, LabVIEW, and HP VEE.
6. **Remote Access.** You can't directly access an IVI driver remotely.
7. **I/O.** C-based IVI is more flexible than VXI*plug&play* with regard to I/O. IVI drivers are free to use any form of I/O, but, at the time of this writing, the IVI infrastructure still had some minor VISA dependencies.

ATE ISSUES AND PROBLEMS

The nature of ATE software has evolved considerably over the years. Some of the most vigorous activity in ATE software and hardware development has historically been in military and aerospace applications, which place unique demands on test systems in terms of use cases, lifetime, costs, scalability, maintainability, and backwards compatibility. Commercial applications seeking to exploit the latest advances in technology further broaden the spectrum of ATE usage models. Military and aerospace applications may last for many years, and new hardware must be integrated into the system over time. The new hardware may come with both new driver technology and new underlying software technologies. In contrast, commercial applications for consumer electronics must change every three to six months to accommodate new designs. But test programmers cannot afford to change test hardware with each design cycle – they must reuse their test hardware (and drivers) as the test application changes. Though these demands are seemingly orthogonal, the underlying problem is the same – independent versioning of hardware and software components. Further complicating the issue, are a number of unique challenges faced by the ATE developer

- Various I/O technologies
- Demand for COTS
- Full Functionality and ease of use
- Wide variety of ADEs (application development environments)
- Legacy support

Various I/O Technologies

Automated test systems require that the test instrumentation be interfaced to the computer. A great deal of work has gone into various standards to accomplish this and to make these interfaces fast, efficient, reliable, and pervasive. GPIB, RS-232, VXI, and PXI are all examples of open standards for instrument I/O. Newer instrumentation even presents networked interfaces, bringing into play the immense array of network protocols as legitimate instrument interfaces. This presents a challenge to instrument driver developers in particular, which, at present, is a task shared by instrument vendors, system integrators, and end users. An end user's computer system may support a particular type of interface, while the instruments it uses support several. This often results in substantial programming effort when either the capabilities of the control computer or those

of the instruments themselves change. ATE applications that make direct I/O calls suffer the most when I/O requirements and/or capabilities change.

The VISA standard introduced an I/O framework for providing some degree of encapsulation of the underlying I/O hardware. There are, however, still implementation differences between the various VISA libraries available. These differences can be manifest in the end user's fundamental application behavior. Furthermore, multi-threading and asynchronous event handling are increasingly important developer topics that are simply not dealt with in a standard way in some VISA implementations, if they are dealt with at all.

Demand for COTS

Both commercial and military applications are increasingly looking to COTS (commercial off-the-shelf) test solutions to lower costs and provide maximum capability, flexibility, and maintainability. System integrators and end users alike desire the extra design degree-of-freedom in selecting best-of-breed instrumentation. Indeed, this trend had become so pervasive as to spur the development of such instrumentation standards as VXI and PXI. Modern test systems are now often composed of multiple types of instruments, each potentially from a different hardware vendor.

Though the hardware standardization allowed instruments from various suppliers to be used together in a single system, a reliable software framework to support that instrumentation does not exist. In the ATE world, this is most clearly manifest in the instrument drivers that accompany COTS instrumentation. Vendors providing COTS solutions must choose a predominant software technology for shipping with their instruments or they must provide multiple types of drivers. Both situations have serious drawbacks. Supporting multiple pieces of code that perform the same functionality almost invariably results in less-used versions being lower quality and poorly maintained. Vendors choosing to support one type of software technology for their instruments will almost certainly alienate a broad segment of their potential market.

COTS also does not imply standardization of any sort. LabVIEW and HP VEE are both COTS products, but neither are open software standards.

Further complicating the situation is that COTS components must interoperate reliably under a single

system, irrespective of the driver technology employed by the various hardware vendors. The best-of-breed decisions faced by integrators and end users are all too often affected by the availability of specific driver technologies, for fear that the integration tasks associated with widely dissimilar driver implementations is too costly and too risky. This simply should not be the case.

Full Functionality and Ease of Use

Software technologies for ATE struggle with the challenge of adequately servicing the needs of widely disparate classes of users. For general-purpose drivers, instrument vendors must allow access to the full functionality of their products. Software developers for instrument vendors strive for completeness and accuracy in creating the software that will serve as the "personality" of their hardware. On the other hand, system integrators and end users place high value on ease-of-use and interoperability. Within the focused needs of a particular test system, full functionality drivers often appear overly complex and unwieldy. Consider SCPI (an example from the firmware world). SCPI provided a predictable way to represent full functionality, but it failed to provide an intuitive or easy-to-use model for instrument control [12].

Wide Variety of ADEs

Related to the issue of satisfying the requirements of a broad class of users and developers is the issue of developing instrumentation software that may be conveniently utilized in a wide variety of ADEs. These ADEs vary from full-featured C++ compilers like Visual C++, to rapid application development (RAD) environments like Visual Basic and LabVIEW, to scripting languages like VBScript and JavaScript. Development environments span computer platforms as well. This further complicates the issue of ATE software development and distribution, as no single distribution or language standard exists across multiple platforms.

System integrators and end users need the freedom to select a development environment that is appropriate for solving their domain-specific problems. This may indeed be very different from the environment an instrument vendor finds effective in populating a massive product catalog with instrument drivers and other ATE software. Solution providers and system integrators often elect RAD environments to deliver productive and cost-effective solutions. In addition, ATE software product companies often present some

of the most demanding requirements for instrument software technologies. Product development may involve a “mixed-mode” environment for development, where certain software components of a particular product are developed in one language or environment, while other components of the same product are developed in a completely different language or environment. For example, product developers may elect to develop GUI components in Visual Basic or LabVIEW, while time-critical components are developed in Visual C++. These various components must interoperate seamlessly and reliably with each other as well as with instrument drivers and other ATE software components. No syntactic, semantic, or component standard exists for the instrument vendors to employ or for the ATE developers to rely upon.

Legacy Support

ATE hardware is notoriously long-lived. By contrast, ATE software is increasingly short-lived. Investments made in initial software development *must* be leveraged when new test requirements appear or increased capabilities are desired. At present, the introduction of new equipment that invariably accompanies system upgrades is often quite disruptive. Application code must often be modified, at best, or completely rewritten, at worst. In either case, the costs associated with these modifications can be exorbitant.

Upgrade paths for ATE systems are often difficult to foresee. Some of the reasons for this relate to domain-specific issues. The solutions for these are often the most difficult to solve. Though it is not the subject of this paper, architectures to address these problems are being developed and demonstrated. However, the greatest cost impact occurs in the ATE application software and the cause is most often associated with integration of new software components. Existing test applications represent a tremendous investment in domain-specific programming, and if the introduction of a new instrument driver technology is not easily incorporated into the existing code, the integration costs can easily exceed the original development costs.

ATE AND COM

Motivation

COM interfaces are a combination of syntax and semantics. The syntax is captured precisely in IDL, and the IDL is unambiguous. Semantics must have some ambiguity to allow multiple objects to use the same interface [13]. The challenge is to find just the right amount of ambiguity. In this sense, COM interfaces are ideal for implementing IVI class definitions that are syntactically precise, but capable of modeling multiple instruments in the class.

Because prefixes are needed to qualify names in C APIs, IVI requires two components to achieve polymorphism – a specific instrument driver that implements class functions and a corresponding class driver. With COM interfaces, method names are always qualified by the object name. The following Visual Basic code shows how object names qualify the `reset()` function, so that prefixes are not needed.

```
` Fluke45 = Driver for Fluke DMM
Dim dmm As Fluke45
` HP3631A = Driver for HP power supply
Dim power as HP3631A
...
dmm.reset()
power.reset()
```

COM offers the possibility of syntactical interchangeability without requiring class drivers.

COM interfaces hide the implementation from client test programs. Implementation language is hidden. Detailed algorithms are hidden and are irrelevant as long as they implement the interface’s semantics. This means that instrument vendors, who presumably know their instruments better than anyone, can implement interfaces in various ways appropriate to their instruments.

COM interfaces are implemented in COM classes and instantiated in COM objects. This means that COM drivers can implement instance data internally. This helps to eliminate one awkward feature of both *VXIplug&play* and IVI drivers. Each method in a *VXIplug&play* or IVI driver must include a session ID as a parameter. The session ID is used by libraries called by the driver to maintain session information. With COM, implementing one COM driver per session allows the session to be stored as instance data.

Since implementation differences are hidden, versioning is easier. A COM object can change, but as long as it exposes the expected interfaces, it can be integrated into existing client test programs without compilation.

COM includes built in remoting capabilities. This creates a whole new set of possibilities to test system designers, capabilities that were only possible previously with custom RPC (remote procedure call) programming. Test programs can access drivers located across the building, the city, or the country. Furthermore, there is no reason why the instrument itself cannot implement the COM object directly. This avoids the price of two remote I/O calls – one from client to driver and one from driver to instrument – in favor of one call from the client to the instrument. As we'll see, this also results in development efficiencies and insures that the quality of the driver rivals that of the firmware.

In VXIplug&play and C-based IVI drivers, the API is flat and could be quite large. The hierarchy is in the function panel file, which is only accessible from test and measurement ADEs. COM allows developers to build hierarchies into their interfaces by including in one interface references to other interfaces.

Another advantage to programmers is the type library. The type library is included with a COM component, and describes the interfaces exposed by the component. This means that ADEs can read the type library and convey the interface to the programmer in the form of programming aids. IntelliSense in Visual Basic and Visual C++, the Visual Basic Object Browser, the OLE viewer, and the LabVIEW ActiveX dropdowns are all examples of this kind of help.

COM has a variety of other features that hold promise for driver developers and test programmers. Collections allow a standard way of implementing channels and other multiple-instance features. Connection points offer a standard way to implement events. COM has standard error handling. It is possible to implement a variety of approaches to concurrency and thread safety. And all of this is designed to work well remotely as well as on a single computer.

IVI and COM

The IVI Foundation is currently investigating implementing drivers as COM objects. This is not intended as a replacement for C-based IVI drivers, but as a complement to them. In this section we'll look at

the advantages to using COM. In the next section we'll look at some issues.

ADE Requirements

IVI-COM drivers must be usable from all popular ATE application development environments (ADEs). These include LabVIEW, HP VEE, Visual Basic and VBA, C, C++, and LabWindows/CVI. Each of these environments has its own peculiarities with regard to the level of COM support provided. The nature of ATE and the instrument drivers that form the backbone of ATE applications is such that developers will need to be able to access drivers from low-performance scripting environments as well as from high-performance C and C++-based environments. IVI-COM drivers offer the promise of a solution for developers in both of these categories.

LabVIEW and HP VEE require automation, which means that drivers must implement automation interfaces, and are restricted to automation data types. Although Visual Basic is no longer restricted to automation interfaces, it is restricted to automation types. Some of the scripting languages have further restrictions on data types. While this may impose serious limitations in more general-purpose applications, IVI-COM drivers would have relatively simple data typing requirements. Indeed, all of the data types required by existing IVI instrument classes can be represented in variants. This ensures that IVI drivers will be usable from all ADEs.

It is important to note that even though IVI-COM data types are restricted when passing parameters to an IVI-COM driver, this imposes no restrictions on the data types that may be used internally by the driver developer. Recall that COM rigidly enforces separation of interface from implementation. The data typing issue is an excellent example of how important this separation can be. The implementation of an IVI-COM driver has only the data type restrictions of the development language. It is only when the IVI-COM driver exposes an interface with parameters that those parameters be of a type appropriate for the COM interface and ADE in use. Were the driver implementation restricted in the same way that interfaces are, COM would be virtually useless for even the most modest applications.

Interfaces

Conventional IVI drivers for classes defined in the existing IVI Foundation specifications rely on ANSI C function prototypes to establish each driver interface.

Alternatively, IVI-COM drivers would expose their functionality through interfaces, just like other COM objects. Recall that COM interfaces come in essentially three varieties – 1) COM custom interfaces, 2) automation interfaces, and 3) dual interfaces. Since IVI drivers are specifications, IVI-COM drivers must be specified against a particular COM interface type. The choice of interface type is an extremely important one. Pure automation interfaces would make ATE application development awkward in languages like C and C++. Additionally, performance would be severely compromised, potentially alienating a large user community of “power users.” Conversely, high-performance custom interfaces are not accessible from the enormous community of Visual Basic, LabVIEW, and VBScript users. Excluding such a large class of users would be antithetical to the broad acceptance goals of the IVI Foundation. Thus, the IVI Foundation has elected to utilize dual interfaces for specifying COM interfaces. This provides speedy vtable access for C++ users and it makes the drivers available to all the popular automation clients.

Use of automation interfaces implies one interface per COM class, as discussed previously. Drivers are built as multiple COM classes tied together with references, in such a way to form a hierarchy. The root of the hierarchy is the only COM class that the client can create – all of the other functionality is accessed by traversing the references that are used to build the hierarchy. The resulting model is very much like Microsoft Excel and Microsoft Word interfaces.

Integration with ADEs

The focus of modern ADEs is chiefly to increase developer productivity. The COM specification enables reuse, but productivity involves a number of other factors. Software technologies must be easily usable within the development *environment*, not just accessible from a development *language*. The prevalence of COM has spurred the development of ADE features that make the use of COM objects more efficient.

The IntelliSense feature built into Microsoft ADEs like Visual Basic, VBScript, VBA, and Visual C++ provides a valuable tool for easily accessing the methods of COM objects. Having to toggle between a component vendor’s documentation and the ADE to determine the correct calling parameters for a method can consume a surprisingly large amount of time. With IntelliSense, ATE developers using IVI-COM drivers will immediately see driver functions and parameters

appear inline. No extra effort is required on the part of the IVI driver developer for the IntelliSense activation to operate for their IVI-COM driver on any of the ADEs listed above.

Object browsers are also a very popular and valuable tool when using COM objects in an application. Object browsers use standardized type library information that the COM component developer ships with the component. IVI-COM drivers for sophisticated instruments could contain an extensive and complicated function hierarchy. The object browsers built into most modern ADEs would provide a very convenient utility assisting a developer to correctly traverse complex hierarchies.

IVI-COM Vision

The goal of IVI-COM would be the proliferation of standard COM instrument interfaces. It’s instructive to consider the analogous infrastructure of conventional ActiveX controls. ActiveX controls are COM objects with a large number of standard interfaces for implementing user interface features. When a developer wishes to implement property pages for his or her ActiveX control, he or she utilizes a standard set of property page interfaces with a defined syntax and semantic. Similarly, an ATE developer using an IVI-COM driver would have available a standard interface when performing a 4-wire DMM resistance measurement.

The end goal for IVI drivers in general is that every instrument ships with a standard driver interface that can be used with any number of popular ADEs. This puts the onus on the instrument vendors to develop and maintain IVI drivers. With IVI-COM drivers, vendors can develop and publish a single driver interface and be able to accommodate most all users. A single IVI-COM driver could be utilized in an Excel spreadsheet in the same way it is utilized in a Visual Basic or C++ program. The same set of functions and function parameters would be visible to the Excel spreadsheet macro programmer as the Visual C++ developer. For example, using an IVI-COM driver, an engineer can take some simple DMM measurements, perform analysis, and populate cells in a worksheet – all within Excel. Any peculiarities of the instrument or of the driver itself could all be exposed in this simple “mock-up” environment, before expensive application development ensued.

COM IN THE BOX

The process of interacting with instrument hardware from a test program typically requires that information pass through a number of levels of software and hardware. For example, consider an application written in HP VEE or LabVIEW.

1. The application executes a statement that attempts to communicate with the instrument via a driver.
2. The ADE processes the statement, calling the driver entry point specified.
3. The driver calls an I/O library such as VISA.
4. The I/O library calls specific I/O APIs that deconstruct the information into packets that can be physically sent by the computer's I/O hardware, and data goes across the I/O connection.
5. The instrument's I/O hardware receives the data and specific I/O libraries reconstruct the packets into a form recognizable to the instrument.
6. The instrument's command string parser parses the command and makes an appropriate function call to the measurement firmware.
7. The measurement firmware interacts with the measurement hardware to perform the requested operation.
8. The flow of results goes in the reverse direction back to the test program.

Consider the components that must be individually managed to communicate this way:

1. *Drivers.* The current industry expectation is that instrument vendors will supply drivers.
2. *Test I/O Library.* VISA, covering GPIB, VXI, and RS-232, is one popular ATE standard. Other forms of I/O specific to ATE (for example, PXI) require I/O libraries specific to ATE. Other computer standard forms of I/O (parallel, USB, IEEE 1394) may also require I/O libraries specific to ATE.
3. *Specific I/O Library.* These are ATE specific, where the underlying I/O protocol is ATE specific. GPIB, VXI, and PXI are examples.
4. *Instrument command string parser.* This must be implemented in the instrument to transform ASCII strings into a form that the measurement firmware can understand.

Since instrument vendors are expected to provide drivers, they must be concerned with each of these four items. Yet these items are typically outside their core competency. Furthermore, since the driver interface looks suspiciously like the function interface

to the instrument firmware, these four steps begin to look like performance-consuming busy-work.

Consider the process again, only this time, assume that the measurement software exposes COM interfaces that look suspiciously like a driver interface.

1. The application executes a statement that attempts to communicate with the instrument via a driver.
2. The ADE/program processes the statement, calling the driver entry point specified.
3. The driver calls a DCOM interface on the instrument's measurement firmware.
4. The measurement firmware interacts with the measurement hardware to perform the requested operation.
5. The flow of results goes in the reverse direction back to the test program.

Now the instrument vendor does not need to give any attention to drivers, ATE specific I/O libraries, or command string parsers in the instrument. Some attention must still be given to DCOM I/O, but a companion assumption is that these instruments would use computer industry standards for I/O, and that DCOM would be supported *by the computer industry* for these standards.

We call this approach "COM in the box".

If instrument vendors move to computer standards for I/O and object software, they will be free to focus on their core competency – measurement technology. ATE specific I/O libraries and instrument drivers, both sources of considerable dissatisfaction in the industry, would be eliminated. The instrument's COM interfaces will get the same level of attention as the rest of the instrument firmware, improving the quality of what test programmers see to the level of firmware.

These ideas are not new. RPC has been well understood for quite a few years. The problem has been that RPC programming is not a general-purpose solution. It is not programmer friendly; it is not implemented or supported as a common standard; and it is not well supported by ATE environments.

COM, on the other hand, is programmer friendly (at least on the client side), is well supported as a standard, and is supported by a wide variety of ATE development environments. As such, it is well positioned to contribute to a new level of programming convenience and value in the ATE world.

[1] Note that this is slightly different from polymorphism in a source-code based object-oriented system like C++. In C++, polymorphism is based on source-code object inheritance, where inheritance can include implementation inheritance as well as interface inheritance. C++ objects are polymorphic because they inherit from other objects, not because they expose multiple, distinct interfaces. The result is that C++ polymorphism is often seen in the light of the sub/super type relationships inherent in inheritance. COM polymorphism is much broader.

[2] Interface inheritance enforces encapsulation. Implementation inheritance exposes the internals of a parent object's implementation. Coupling the child object with the parent object's implementation can actually break the child (and, hence, the client program), because implementation inheritance does not make clear and precise the interface between child and parent.

[3] The earliest published references found by the authors were to BCD instruments in the 1961 HP Catalogue: HP 1961 Electronic Test Instruments, Hewlett-Packard Company, Palo Alto, CA, 1961, p. 198.

[4] National Instruments released LabWindows/CVI 5.0 with "IVI" technology included. Their work will clearly be influential in the content of the architecture and general driver requirement specifications when they are complete, but expect some significant changes.

[5] *IVI-5: IviDmm Class Specification*, Revision 1.0, IVI Foundation, August 1998. All IVI specifications cited are available from the IVI web site at <http://www.ivifoundation.org/>

[6] *IVI-4: IviScope Class Specification*, Revision 1.0, IVI Foundation, August 1998.

[7] *IVI-7: IviPower Class Specification*, Revision 1.0, IVI Foundation, August 1998.

[8] *IVI-8: IviSwitch Class Specification*, Revision 1.1, IVI Foundation, August 1998.

[9] *IVI-6: IviFgen Class Specification*, Revision 1.0, IVI Foundation, August 1998.

[10] The power supply spec is an example.

[11] Oblad, Roger P., "Applying New Software Technologies to Solve Key System Integration Issues," *Proceedings of Systems Readiness Technology Conference, Autotestcon, 1997*, IEEE, 1997 (ISBN 0-7803-4162-7), p. 181 ff.

[12] The SCPI CONFigure/MEASure model was created to try to address ease of use and transfer of learning issues, but has been hard to implement consistently and was not as intuitive for test programmers as was hoped.

[13] Box, Don, Keith Brown, Tim Ewald, and Chris Sells, *Effective COM*, Addison Wesley, Reading, Mass., 1999 (ISBN 0-201-37968-6), p. 32.