

# THE STATE OF INTERCHANGEABILITY IN ATE

**Kirk Fertitta**  
**Pacific Software Group**  
**3435 Lebon Drive, Suite 1134**  
**San Diego, CA 92122**  
**(858) 554-0721**  
[fertitta@san.rr.com](mailto:fertitta@san.rr.com)

**Daniel Eriksson**  
**Vektrex Electronic Systems, Inc**  
**8675 Miralani Drive, Suite 150**  
**San Diego, CA 92126**  
**(858) 458-5822**  
[Daniel.Eriksson@computer.org](mailto:Daniel.Eriksson@computer.org)

*Abstract - The Interchangeable Virtual Instrument (IVI) Foundation formed in August of 1998 to address the challenge of instrument interchangeability in the test and measurement industry. Since its inception, the IVI Foundation has engaged in vigorous standardization activities along several different, though coherent, axes. Instrument class specifications are continuously under development for some of the most pervasive types of instruments. Architectural standards establishing IVI conformance criteria will broaden the capability of vendors, end-users, and system integrators to innovate while simultaneously allowing them to integrate into a heterogeneous IVI-based system. IVI Measurement Stimulus Subsystems (IVI-MSS) standards under development offer the promise of a higher level of interchangeability than instrument-centric standards can offer. Shared IVI infrastructure components will ensure interoperability in IVI systems by providing a common model for services such as configuration, installation, resource sharing, event notification and management, and component instantiation. In addition, the Foundation has embraced COM technology as a powerful means for delivering interchangeable ATE software services.*

## EVOLUTION OF IVI

As is the case with most rapidly evolving technologies, understanding the state-of-the-art in IVI and interchangeability requires a solid grasp of the evolution of the IVI Foundation itself and the standardization efforts and accomplishments recorded thus far. The early efforts of the Foundation centered upon the domain problem of instrumentation and interchangeability, as opposed to some of the pure computer science issues of software interchangeability. Most of the activity early on

was devoted to the definition of some of the more common classes of instruments. Focused working groups within the Foundation would work to identify and define the salient features of instrument classes, like digital multimeters (DMMs), oscilloscopes, and function generators. For each instrument class, the instrument working groups would produce a specification that enumerated all of the required and optional functions and attributes that an instrument driver for that class would need to support in order to be “IVI compliant.” The working groups expressed the class functions and attributes in a formal specification using ANSI C-language prototypes.

The precise reason why C was chosen as the language for instrument specifications is a continual subject of debate both inside and outside the IVI Foundation. C enjoys the advantages of being familiar to a large class of test engineers and software developers as well as being platform-neutral. It is also reasonable to attribute the use of C in the IVI Foundation to one of the most active member companies – National Instruments (NI). NI’s LabWindows/CVI development environment is a popular C-language alternative for a large class of test engineers. An aggressive offering of instrument driver development tools from NI hosted in the CVI integrated development environment (IDE) made C-language specifications the most natural choice for the Foundation.

With the domain problem of defining instrument classes and their features reasonably well in hand, discussions related to pure software issues of standardization and interchangeability began to be a prime focus of the IVI Foundation. In spite of well-advertised “IVI-compliant” drivers available on the market, questions remained as to the best software technology to employ in delivering IVI

drivers. The first available IVI drivers were simply Windows Dynamic Link Libraries (DLLs), but concerns about the maintainability, robustness, and usability of simple DLLs made many wondering if a better software component strategy should be employed. Instrument vendors faced with the mammoth task of revamping entire catalogs to meet a new instrument driver standard were particularly interested in examining a better software strategy for delivering IVI drivers. From this need came a very aggressive movement within the Foundation to investigate Microsoft's Component Object Model (COM) as the software component technology for IVI drivers.

Within IVI, COM-based solutions are being employed to attack a variety of technical problems. COM was first considered in addressing some issues IVI had with ANSI-C as the syntax specification language for IVI instrument classes. C-language, though familiar to many, is not ideal as a pure interface specification language. COM uses Interface Definition Language (IDL) to completely specify the syntax of an interface unambiguously. As an example of how ambiguities can arise in interfaces, examine the following simple C function prototype:

```
void foo(char* SomeParm);
```

It is unclear from this prototype whether the parameter `SomeParm` is an input or output parameter. Is it an array of `char`'s, and if so, how many elements does it contain? IDL was designed for precisely this sort of problem and it has explicit constructs to remove the type of ambiguity demonstrated in the C-language prototype example above.

Though the benefits COM IDL provide as a pure interface specification language were appreciated, many of the member companies saw even greater potential in what COM offered for driver developers and for end users writing test applications. A thorough discussion of the advantages of COM is beyond the scope of this paper, but a brief examination of some of the benefits most commonly cited by major IVI members is informative. As an interface standard that does not address implementation, COM is truly language-neutral. This allows vendors to develop drivers using the language and application development environment (ADE) of choice, while simultaneously allowing their end-user customers to author test applications in their ADE of choice. Not

only does this give greater design degrees of freedom to both sets of parties, it leverages the built-in COM support found in almost all Windows-based ADEs. ADEs are increasingly providing fuller and more seamless support for COM, so that both driver authors and test application developers have a more productive, richer programming experience. This ease of driver use is very important to instrument vendor because customers associate that ease of use with the instrument hardware.

Other trends in the test and measurement industry prompted IVI member companies to embrace COM as a component technology for drivers. Companies are increasingly striving to reduce product test times, as test time directly influences production capacity. Some popular tactics employed to produce faster tests include writing multithreaded test applications that can perform portions of tests "in parallel", thereby reducing the overall test time. Additionally, test applications are becoming inherently more complex and often distributed in nature. Applications need to seamlessly share data with other threads, other processes and even other machines on a network. COM provides infrastructure and standards to elegantly deal with these complex issues that would otherwise require custom solutions. Writing multithreaded code requires developers to work with tricky operating system synchronization primitives, while custom distributed applications require intimate knowledge of specific interprocess communication protocols.

Bringing COM into the IVI standardization efforts resulted in two main choices for drivers – ANSI-C or COM. Vendors will be providing IVI drivers with their instruments, so it will be the vendor who ultimately determines which interface technology will be used with their drivers. Most vendors (including the very largest ones) are committed, in many cases exclusively, to COM technology for their instruments. Nevertheless, the Foundation maintains its ANSI-C heritage, and most likely users will have to be educated on terms such as "IVI-C" drivers versus "IVI-COM" drivers. Indeed, that is the subject of a later section in this paper.

The IVI Foundation has now turned a good deal of its attention to what is commonly referred to as "IVI infrastructure" issues. The idea is to take a system-level perspective on what drivers need to do to operate in a test

system. The Foundation is looking closely at how test engineers use drivers in a complete system. This is considerably more involved than a single application on a test station talking to a single instrument through a single driver. What the Foundation has determined is that there are common services and patterns of usage that lend themselves well to standardization. Drivers themselves have certain requirements with regards to installation, configuration, versioning, and so on. Additionally, drivers provide services that must be done in a standard way if they are to be at all useful in a heterogeneous test system that employs multiple instruments from a variety of vendors. For example, test applications are often event driven, in that they perform certain functions in response to the occurrence of an asynchronous event from a driver. Indeed, they may also fire events that a driver must detect. If there is not a common mechanism amongst vendors for handling asynchronous events, then users will not be able to take advantage of these capabilities in a heterogeneous test system.

To address the need for system-level standardization, or “IVI infrastructure”, the Foundation has undertaken the task of specifying and implementing specific infrastructure components. The vision is that the IVI Foundation will take an “open source” approach to these components, so that they will be freely available to the public. The legal and logistical challenges associated with this have yet to be fully resolved. Nevertheless, the Foundation is moving ahead vigorously in specifying, standardizing, and implementing what it believes to be the most critical IVI infrastructure components. These components address the need for common, standardized configuration services, event services, and activation services. They are discussed in detail in later in this paper.

Ever since its inception, IVI has struggled, both internally and externally, with what interchangeability really means and what users can expect. A good deal of industry press has been devoted to this topic elsewhere, so this paper will not examine the issue in detail. However, to properly understand what IVI will offer in terms of interchangeability, it is important to have at least a minimal grasp of IVI’s notion of “levels of interchangeability.” The present IVI class specifications define interfaces that allow instruments within a class to be interchanged without recompiling the test application. That is to say, a digital multimeter (DMM) from one

vendor can be swapped with a DMM from a different vendor without breaking the syntactic integrity of the test application. This offers interchangeability within an instrument class only. Even though an oscilloscope, for example, is capable of making some of the same measurements as a DMM, the oscilloscope could not be used interchangeably in an application that had been written to the DMM class interface specification. Two working groups within the IVI Foundation are now working on standards that do not take an instrument-centric view of interchangeability at all. These working groups are the Measurement Stimulus Subsystem (MSS) working group and the Signal Interface working group. With the MSS approach, a system integrator or other solution provider defines an abstract (but fixed) measurement interface that provides some logical measurement service. Beneath this interface, any combination of instruments, custom code, and any other required resources conspire to provide the service specified by the interface. In this way, the test application is shielded from the specifics of the underlying instrumentation, just as with conventional IVI drivers. However, the key difference is that with MSS, the interface against which the program is written is totally abstract, so that any combination of software and hardware that can deliver the required measurement service can be supplied without disrupting the test application. Whereas conventional IVI drivers only allow interchanging individual instrument tasks within an instrument class, MSS allows interchangeability of entire logical measurements, irrespective of instrument class. This provides what IVI terms a higher “level of interchangeability.”

The Signal Interface working group shares much of the same philosophy as MSS, in that it completely abstracts away the instrument altogether in its interface specifications. A popular programming paradigm for test applications views a test program as a series of signal source operations and signal measurement operations. Indeed, this signal-based model has been the cornerstone of ATLAS programming for decades, and a great many complex engineering problems have been (and are being) solved using signal-based programming. The idea is simple -- a test program logically wants to test properties of a signal, not to exercise features of an instrument. A common task might be to measure the rise time of an AC pulse signal. Rather than define instrument-specific commands to perform the measurement, the programmer only has to specify that the signal of interest was an AC pulse and that the required measurement parameter was the rise time. Any instrument, or collection of

instruments and software algorithms, that are physically capable of performing that measurement can be introduced into the test system interchangeably without requiring any changes to the application source code. To achieve that level of interchangeability, the Signal Interface working group is charged with defining standards and specifications for building signal-based IVI components.

## IVI ARCHITECTURE

Though much of the work of the IVI Foundation is devoted to defining instrument classes and their salient features and functions, a great deal of work is being devoted to the system-level view of how drivers will be used in practice. For drivers from different vendors to interoperate well together in the same system, the drivers must employ a standard mechanism for performing common operations in a test system. IVI has taken the approach to deliver these common services in separate software components that will be specified and implemented by the IVI Foundation itself. It is unclear at this point if “implemented by the IVI Foundation” means that member companies will volunteer resources directly or whether an outside party will provide resources. Nevertheless, the Foundation is vigorously pursuing development of some of these common components.

With the identification of some standard system-level services that must be in place for test engineers to take advantage of IVI drivers, the concept of *IVI architecture* emerges. Broadly speaking, the collection of common components for system-level services together with the IVI drivers and the test application itself constitute the IVI architecture. Specification of the common components (also referred to as *infrastructure components*) is a more recent and, hence, less understood, activity undertaken by the IVI Foundation. However, a great deal of the value of IVI itself lies in this common driver architecture that allows IVI drivers from multiple vendors to interoperate not only peaceably together in the same system, but with rich functionality. This section discusses in detail two of the common components currently being developed by the IVI Foundation – the IVI Config Store and the IVI Event Server.

## IVI Config Store

Most modern computer applications rely on some sort of configuration data file to supply information that would otherwise reside in the application code itself. The purpose of separating an application’s configuration data from the application source code is to extend to the end user a prescribed amount of configurability in the application’s behavior. Changes made to specific pieces of information in the configuration file can effect changes in program behavior without editing, recompiling, or relinking application source code. Additionally, configuration files often serve as centralized information repositories for related applications requiring a well-known location to store and retrieve data.

The IVI Config Store is a standard IVI-defined infrastructure component that serves as a centralized repository for IVI driver configuration information. The information contained in the store serves a variety of purposes, as will be discussed shortly. In many respects, the IVI Config Store is analogous to the Windows registry. Application programs, individual components, and users store and retrieve information from the registry for different purposes. IVI uses a similar model, but with a much tighter scope.

### *Config Store Structure and Contents*

Physically, the IVI Config Store is an XML file that contains textual information about all of the IVI drivers installed on a particular test system. XML was chosen as the storage format for a number of reasons: it is human readable, it is truly cross-platform, good parsers are available on virtually all platforms, and it has tremendous momentum in the computer industry. The XML file contains groups of name-value pairs. Each group may describe an IVI driver, an installed piece of support software, a hardware asset, or another IVI-defined component. The name-value pairs are IVI-defined attributes and their values. Drivers, test applications, and IVI infrastructure components store and retrieve attribute values for a number of purposes:

- **Interchangeability Settings**

IVI relies on the use of Logical Names for instruments to achieve interchangeability. Test

application source code refers to a user-specified logical name for an instrument, rather than referring directly to a specific instrument. In the Config Store, the logical names are mapped to the specific instruments in use. When a new instrument is introduced into the system in place of an existing instrument, it is the logical name mapping (not the application source code) that is edited by the user to allow the test application to function interchangeably with the new instrument.

- **Executable Code Paths**

Test applications and certain IVI infrastructure components must know the file paths of the executable code for every driver. Typically, the executable code takes the form of a Windows DLL. These paths are stored in the Config Store.

- **Static Driver Characteristics**

Each driver has associated with it a certain amount of static information that a vendor's installation program will typically write in the Config Store. This information includes the vendor name, the instrument model, the IVI-defined class(es) supported, the driver version number, and other data.

- **Driver Initialization Options**

IVI defines a number of attributes in the Config Store that the user employs to control driver's behavior. For instance, state-caching and range-checking are two well-known features of IVI drivers, and both of these options are enabled by values set in the IVI Config Store. When a driver is initialized for use in a test application, the driver inherently knows how to retrieve these initialization options from the Config Store and how to modify its dynamic behavior accordingly.

- **Custom Driver Attributes**

One of the prevalent themes in IVI is extensibility. Vendors must maintain the ability to innovate and differentiate while maintaining full IVI compliance. This means drivers must be able to expose vendor-specific functionality. IVI provides guidelines and, in many cases, specific requirements for how vendors and driver authors incorporate non-IVI defined functionality in their drivers. The Config Store is one such area where driver authors and even end users have the capability to define their own

attributes for their own purposes. These custom attributes may be simple flags that enable or disable special driver features, or they may be more complex data that links together software modules that add special capabilities to drivers.

## ***Config Store API***

The Config Store specification being developed by the IVI Foundation actually consists of two parts – a syntax specification and an application programming interface (API). The first part of the Config Store specification deals with the definition of the XML syntax for the attributes contained within the Config Store file itself. Given this published syntax specification, vendors and end users can interact with the Config Store by modifying the values of attributes within the store directly. Since XML is text-based, it is a simple matter to interactively edit values in the Config Store – if one is very familiar with the impact direct edits to the Config Store may have. Additionally, Microsoft's Internet Explorer ships with a COM object that parses XML directly and provides a programming interface for modifying an XML file.

In spite of the ease with which an XML file can be edited directly, this would not be an advisable means for changing Config Store attributes. This is due to the fact that entries in the Config Store contain dependencies and links to other entries. A mistake made in directly editing the Config Store could adversely affect a number of components throughout the test station. Maintaining the referential integrity of the Config Store is of paramount importance, and so the IVI Foundation has begun work in specifying an API for modifying the Config Store. The API will be exposed by a COM object that implements the editing logic that ensures referential integrity of Config Store elements and guarantees that the XML data remains well-formed. The Config Store API also allows instrument driver authors and test programmers to interact with the Config Store using IVI concepts and constructs, such as "driver" and "hardware asset", rather than generic XML constructs.

Specifying a standard API for the IVI Config Store also has the advantage of completely abstracting away the nature and technology of the underlying storage medium. If, at some point in the future, it becomes desirable to use an alternative medium of the Config Store API, such as a binary structured storage file or an SQL database, then applications and drivers written to the standard API will

not have to be modified at all. New storage technologies that offer advantages in terms of speed or networking capabilities can be introduced without disrupting the considerable investment in test applications and IVI instrument drivers.

## **IVI Event Server**

As both test applications and test instrumentation become more complex, test programs are becoming more parallel in nature. Multithreaded programming techniques are often employed to maximize both CPU and instrument utilization, thereby reducing overall test time. As a result, test programs need to employ mechanism for generating and responding to asynchronous events. For example, a test application may wish to trigger a time-consuming measurement on an instrument and then perform some useful software calculations or other work until the instrument has completed its measurement operation. For this to work, both the instrument driver and the test program must agree on how the measurement completion event is formatted, fired, and received. In order to provide a common infrastructure for drivers, test programs, and other components to share asynchronous events, the IVI Foundation has begun development of the IVI Event Server.

The Event Server is a COM component that exposes standard IVI-defined interfaces for software components to source and sink events. The Event Server supports the classic publisher-subscriber design pattern, whereby a software component can publish a list of events that it may fire and any number of other software components can subscribe to any of those events and trigger some useful operation upon their occurrence. The Event Server offers advanced features such as filtering, which allows a software component to select the specific type(s) of events to which it wants to subscribe. Internally, the Event Server uses advanced multithreading techniques to fire events essentially simultaneously, irrespective of the number of, or relationships between, subscribers. Event logging is also a supported feature.

The architecture of the IVI Event Server separates functionality into two essential components. An inprocess event server component resides in the process space of a particular application. This allows each application to communicate very efficiently with the event server, as no process boundaries are being crossed, and hence, no interprocess communication mechanisms are required. The second element of the Event Server

architecture is a centralized Event Server Manager that acts as the central hub for all incoming and outgoing events on a particular machine. The Event Server Manager resides in its own process space and communicates with the individual inprocess event server components to properly route events between publishers and subscribers. This architecture has a number of advantages. It allows events to be shared across process boundaries seamlessly, without requiring test applications or drivers to employ any interprocess communication schemes. Additionally, it decouples event publishers from event subscribers, so that if one component is trying to communicate with another component that is not yet running, the Event Server Manager maintains the link until both components are activated.

## **TYPES OF IVI DRIVERS**

The introduction of COM-based IVI drivers coupled with the presence of IVI infrastructure components like the IVI Config Store and the IVI Event Server has added a good deal of confusion as to what users can expect from IVI drivers. Fundamental questions often arise as to exactly what is an IVI driver. The key to understanding the answer is understanding that there are now different flavors of IVI drivers. Though the IVI Foundation itself has yet (at the time of this writing) to agree upon the precise nomenclature for type different types of IVI drivers, the names used here are close candidates.

There are currently four types of IVI drivers. These are derived from two possibilities for interface technology (ANSI-C or COM) and two possibilities for class compliance (IVI class compliant or custom class compliant). At first glance, it may seem that it is contradictory to define a standard like IVI and then allow variations on that standard. What really binds IVI drivers together is their ability to work together in a single system. The common IVI architecture described earlier is the framework upon which all four types of IVI drivers are built.

## **Inherent Capabilities**

IVI divides the universe of instrumentation into classes of instruments, such as digital multimeters and oscilloscopes. The IVI Foundation publishes separate specifications for each instrument class that enumerate the required functions and properties a class-compliant instrument driver must provide. However, there are

certain driver functions and properties that are common to all instrument classes. IVI terms these *inherent capabilities*. Examples of functions that have been standardized across instrument classes are those that reset the instrument, initialize the instrument, retrieve manufacturer information, and retrieve version information. These inherent capabilities are also common across all four types of IVI drivers. Any type of IVI driver must conform to these inherent capabilities.

## **IVI-C Class-Compliant Drivers**

An IVI-C Class-Compliant Driver is an instrument driver that exposes an ANSI-C interface to access its functionality. The “Class-Compliant” designation indicates that the driver complies with one of the IVI-defined instrument classes, such as the DMM class specification, the function generator class specification, or the oscilloscope class specification. It may at first seem strange that the “Class-Compliant” designation is included, as many people expect that every IVI driver must comply with an instrument class. This is not true, as the discussion on custom IVI drivers illustrates below.

IVI-C Class-Compliant Drivers must also conform to a special IVI specification (currently under development) known informally as the IVI-C Architecture Specification. This document details ANSI-C interfaces and infrastructure components that drivers must support to be IVI-compliant drivers. Compliance with the architecture specifications ensures that IVI-C drivers from different vendors can interoperate with each other as well as with IVI-COM drivers from multiple vendors. As with all IVI drivers, the IVI-C Class-Compliant Driver must support the IVI-defined inherent capabilities.

## **IVI-COM Class-Compliant Drivers**

An IVI-COM Class-Compliant Driver is an instrument driver that exposes a COM interface to access its functionality. Similar to the IVI-C Class-Compliant Driver, and IVI-COM Class-Compliant Driver complies with one of the IVI-defined instrument classes. Also like the IVI-C Class-Compliant Driver, the IVI-COM Class-Compliant Driver must also comply with a separate architecture specification that addresses COM-specific issues. COM is a broad-based technology with a rich feature set. In order to maximize interchangeability and COM drivers interoperable, the COM architecture specification details how IVI-COM drivers must handle COM errors, how IVI-COM interfaces hierarchies will be structured, which specific type of COM interface will be

employed, which COM threading model must be supported, and so on. As with all IVI drivers, the IVI-COM Class-Compliant Driver must support the IVI-defined inherent capabilities as well.

It is expected that the IVI-COM Class-Compliant Driver will be the dominant type of IVI driver in the near future. Most vendor companies (including the very largest ones) are exclusively committed to COM technology for driver development. The enormous momentum behind COM technology should make the IVI-COM driver a powerful tool for test application developers.

## **IVI-COM and IVI-C Custom Drivers**

The discussion above on the IVI architecture is intended to give insights into the benefits IVI offers even in the absence of an IVI-defined class specification. There are certain types of instruments that are so complex or so specialized that the IVI Foundation is unlikely to compose class specifications for those instruments. In these situations, a vendor or end-user may choose to define their own custom instrument class. Such types of drivers are still IVI drivers. If a vendor supplies an ANSI-C interface to the driver, it is known as an IVI-C Custom Driver. If a vendor supplies a COM interface to the driver, it is known as an IVI-COM Custom Driver.

There are a number of benefits vendors and end users realize with IVI Custom Drivers. First and foremost, these drivers must comply with the IVI architecture. This means that they must provide all of the IVI-defined inherent capabilities; they must know how to interact with the IVI Config Store and the IVI Event Server, and they must follow the conventions laid out in either the C Architecture Specification or the COM Architecture Specification (depending upon which interface technology is employed). Secondly, IVI Custom Drivers could still provide some of the popular features that many people intrinsically associate with IVI, such as state-caching and range-checking. For IVI-COM Custom Drivers, end users and vendors alike would still reap all of the benefits inherent in COM technology, such as rich ADE support, multithreading, and transparent remote access capabilities.

Another interesting application of IVI Custom Drivers is in the definition of custom classes for the purpose of achieving interchangeability only within a single company’s product lines. For instance, the classic example is a custom class definition that a company like Nokia composes for its line of mobile phones. If Nokia’s test programmers internally develop drivers compliant

with their custom phone class definition, then they could achieve interchangeability within different models of their phones. This could reduce test program development costs for certain types of tests. Additionally, training for test station operators could be simplified, and retraining could be potentially reduced.

## IVI MYTHS

There are many misconceptions about what an IVI driver really offers. Exactly why this has happened is unclear, but a combination of salesmen without proper knowledge, general hype and a lack of information would all be good guesses.

### State caching

Many end users believe that state caching, as offered by IVI drivers, will magically speed up their test programs by an order of a magnitude. Unless their test programs are poorly written they will probably be disappointed by the result.

State caching offers performance benefits, but only when redundant commands are sent over the bus to begin with. If a test program is written to always initialize an instrument to a known state before taking a measurement there might be a noticeable performance increase simply because less bus traffic is generated. Another benefit of state caching is when an operation takes a long time for the instrument to complete. An example of such an operation might be the switching between frequency- and time-domain on a spectrum analyzer.

State caching allows the driver to verify the value of the parameter the user is trying to set prior to issuing the command to the instrument. If the user-supplied value is equal to the current value held in the state cache, no command is issued. If the two values differ then a command is issued to the instrument and the cache is updated.

While state caching is advertised as something inherent to IVI drivers, in reality state caching is optional to implement. The only requirement is that the driver supports the enable/disable state cache method call. A driver that does not support state caching will simply ignore the call to turn state caching on. A driver can also choose to only cache a limited number of parameters.

Also, state caching is made more complicated by the fact that many parameters in the instrument are coupled to each other. Changing the resolution bandwidth might invalidate the video bandwidth setting and so on. These couplings make implementing state caching for all parameters a time consuming and difficult task.

### Range checking

Range checking is used to verify that parameters are within bounds before they are sent to the instrument. Being able to trap these error conditions in the driver instead of in the instrument firmware makes debugging a test program easier. However, since the algorithms for determining out-of-bounds conditions in the firmware are sometimes quite complicated, the driver writer often chooses to use less complicated algorithms to test the parameters. This discrepancy between driver and firmware might make life more confusing for the end user.

Range checking can be combined with coercion. Coercion forces out-of-bounds values to within the limits described by the driver. This allows the test program to continue without an error condition. However, using this as a programming paradigm is questionable since it allows for successful completion of a measurement without exact knowledge of how the instrument was set up. In most industries this is not acceptable.

### Interchangeability means “same answer”

Being able to replace one instrument with another instrument and still get the same result when performing a measurement is often referred to as “same answer”. The fact that the interchangeability provided by IVI Class drivers does not guarantee “same answer” is something that unfortunately has been lost somewhere in the marketing hype.

Interchangeability does not necessarily mean “same answer”. To get a guaranteed “same answer” there needs to be some extra code that compensates for the differences between the instruments. This is where Measurement and Stimulus Subsystem (MSS) comes into the picture. MSS provides a place to put the code to do this compensation, and it provides a nomenclature and naming standard useful when discussing the architecture.

## **Drivers come with simulation**

Simulation has been advertised as one of the benefits with IVI drivers. While it is true that some drivers will provide simulation, it is not required by a driver to be IVI compliant. Also, IVI does not define how and what to simulate. If a driver supports simulation it might do anything from simply returning a static value, asking the user for what to return, or reading the values from a user-supplied file.

Different levels of simulation might very well be used as a marketing advantage when trying to sell instruments. The ability to run test programs without having to connect any instruments is very useful, so expect the big companies to compete with each other not only on instrument performance, but also on driver capabilities in the future.

## **CONCLUSION**

This paper has discussed many aspects of the standards emerging from the IVI Foundation. Though there remains a significant amount of work to be done, the momentum behind the technology and the energy of the participants is encouraging. The scope of the effort has indeed expanded from that envisioned when the Foundation was formed. Much more attention is being paid to system-level issues and to pure software technology issues. A standardized architecture is emerging that has at its heart a collection of cooperating infrastructure components that factor out of the test program common operations and features. This promises to make IVI a more robust, more comprehensive, driver standard that delivers to the industry something of truly enduring value.