Polymorphic Analog Instrument Environment

IVI for Analog Instrumentation GUIs

David Lind
Mike Haney
Ryan Healey
Assembly Test Division
Teradyne, Inc.
North Reading, MA

***Abstract*— this paper discusses the architecture of an environment for analog tests that is designed to work on a variety of hardware platforms. It employs a resource manager that understands the resources that are available and manages their allocation. The particular brand of instrument is not important, as long as the instrument contains the necessary capabilities.**

The Analog Instrument Environment contains instrument graphical user interfaces that morph to reflect the capabilities of the target hardware. There is a UI with a common look and feel that queries the hardware for its capabilities, and allows real time interaction with the hardware using the particular instrument's native nomenclature. The environment lets you define analog test steps consisting of a sequence of steps that control and read back results from the instrumentation available. You can run and debug the step, and you can generate the code necessary to run the steps in a Test Program Set (TPS).

Keywords- Analog; IVI;GUI; Debug;

I. INTRODUCTION

The Interchangeable Virtual Instrument (IVI) Foundation has over the last decade defined standards for programming analog instruments in various software languages[1]. While IVI provides an instrument interchangeable method for programming instruments in a TPS, until now there has been no graphical equivalent for IVI. This paper discusses an implementation of an Analog Test Environment that provides common analog instrument panels that can morph into depicting the capabilities of a specific instance of an instrument. For the purposes of this discussion, the types of instrumentation being considered are Digital Multi-Meter, Waveform Generator, Timer/Counter, and Digitizer. In this paper we will use as an example a graphical display for a waveform generator.

There are several problems to overcome in such an implementation. They include:

- Managing available resources and connecting them out to the unit under test
- Defining common capabilities of analog instruments that may be exposed by the GUI
- Communication of settings information between the common GUI and an instrument specific runtime code layer.
- Handling instrument specific capabilities.

This paper will discuss how each of these problems can be overcome to provide an instrument independent common GUI for analog instrumentation control.

At the highest level, the environment runs a Test Program Set (TPS) and allows control of a variety of analog instrumentation used in that TPS. The environment uses several instrument agnostic GUI layers to display the setup and results of various kinds of analog instruments. These GUI layers contain generic instrument panels that access a set of instrument specific runtime layers. The instrument specific runtime layers handle the specific driver calls necessary to setup the specific instance of hardware. The environment also employs a resource manager used to assign a qualifying resource to a specific need. Both the Resource Manager and the Instrument Runtime Layers have access to IVI drivers for specific instruments, the rest of the system is insulated from the drivers.

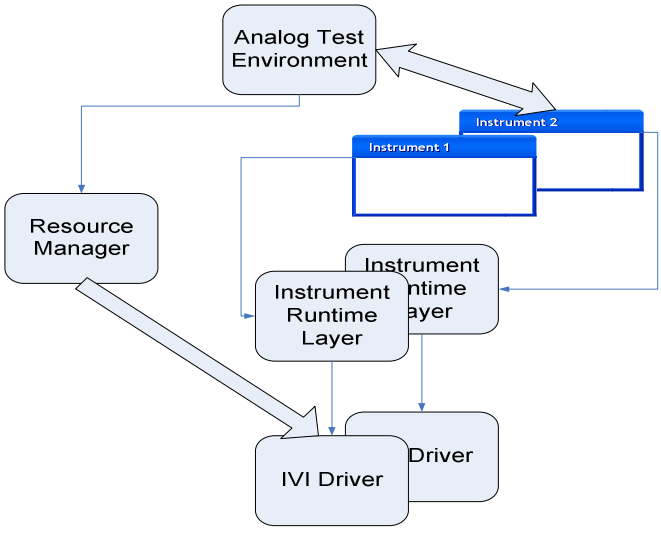

Figure 1. High Level Overview

[1] IVI Foundation, http://www.ivifoundation.org/about/Overview.aspx

978-1-4244-9363-0/11/$26.00 ©2011 IEEE

A user can access the Analog Test Environment using an application that contains various test editors. This central application allows resources to be managed across multiple editors.

II. Resource Management

The Analog Test Environment can display multiple instrument panels that are connected to unique physical instruments. These various physical resources must be managed to allow the Analog Test Environment to display the available resources and to avoid attempts to use the same physical resource in multiple instrument panels. This management is handled within the Resource Manager Service.

When the Analog Test Environment is started, it performs a discovery process of the instruments it supports. It then asks the Resource Manager for the available hardware resources that match those supported by the Analog Test Environment. If hardware resources exist for a supported instrument then a button is displayed on the instrument toolbar (with a matching icon) which the user can click to invoke a GUI instance of that instrument.

When a user selects an instrument, the Analog Test Environment prompts the user with a list of available hardware resources that can be chosen. After a hardware resource is selected, the instrument GUI is created and the hardware resource is then bound to that GUI. The Resource Manager keeps track of the allocated resources to prevent resource conflicts.

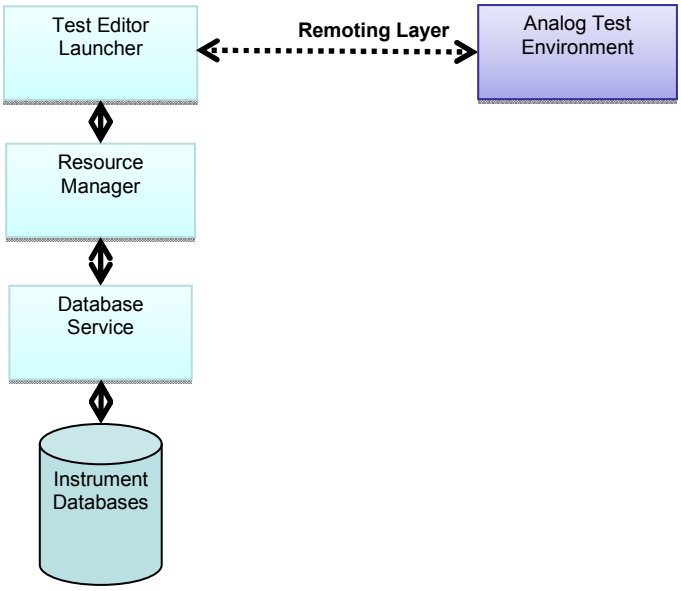

Figure 2. Physical Resource Management

When an instrument is selected from the Analog Test Environment's instruments toolbar, an ID for matching resources is passed into the Resource Manager. The Resource Manager then returns a list of resources that can be used with the particular instrument. When the user selects a resource it is then reserved. The Resource Manager then broadcasts an event to all instrument GUIs that care about this resource so that their resource menus are updated to reflect this change. When a resource is unreserved, i.e. the user selects a different physical resource; the menus are updated across all instrument GUIs indicating that the resource was released

The Resource Manager has a major role when restoring a saved session in the Analog Test Environment. When restoring, the Resource Manager validates that the physical resources are still available. If the validation fails the Analog Test Environment requests a list of available resources from the Resource Manager and allows the user to select an alternate physical resource.

III. Run-Time Interface

The GUI must be able to query the actual hardware for its state so that the current settings of the specific instance of hardware can be displayed. Then when the user instructs the GUI to perform an action on the hardware, the GUI must be able to communicate the settings to the instrument using nomenclature native to the particular instrument. In addition the GUI needs to do this with no knowledge of the specific hardware programming layer in order to maintain its ability to morph itself into a graphical control for any brand of instrument class.

To allow the Analog Test Environment to have no knowledge of instrument and runtime information, there is a separation of instrument panels, runtime layers and runtime execution. The instrument panels contain controls to manage the display and change information relevant to a particular class of instrument. The runtime layers contain all the information to communicate with a physical instance of an instrument. The runtime engine manages the sequencing and execution flow of interactive execution. Figure 3 below shows this separation of hardware knowledge in the Analog Test Environment. The boxes shaded in purple are involved in graphical control and have no knowledge of the specific hardware used. The green boxes interface with the hardware drivers directly. The Analog Test Environment can instantiate and interface to a set of Instrument Panels. The Analog Test Environment also interfaces to a runtime engine. The Runtime Engine manages the activity of the instrument as directed by the environment. For example it manages configuring and running an instrument, and fetching measurement values. This runtime engine still has no knowledge about programming the specific hardware. The direct hardware programming is done by the Instrument Runtime. And the Instrument Code Generator has knowledge of the specific instrument so that it can generate code for it.

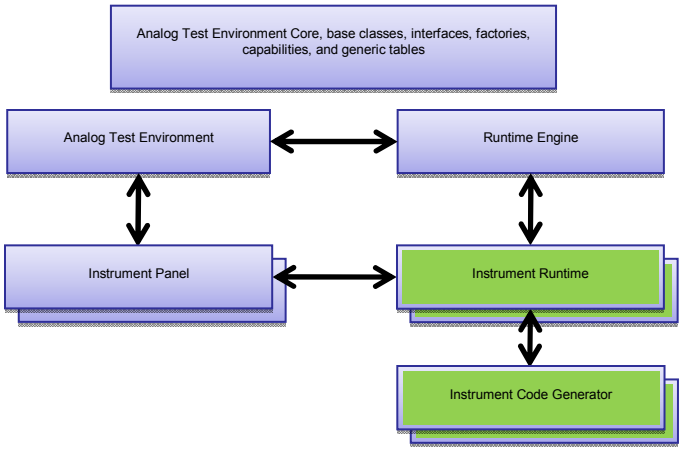

Figure 3. Layers Within the Analog Test Environment

A. *Generic GUI to Instrument Specific Communication*

In order to accomplish this instrument independent query and instrument setup, we use a set of instrument settings objects. Each instrument settings object contains information about a specific setting of the hardware. An example is Signal Generator Waveform Type. The Instrument Panel supports a set of common signal generator waveform types such as sine, square, pulse, triangle, etc. The table provided by the Instrument Runtime maps these generic waveform types to the instrument specific nomenclature. There are a set of such tables for each instrument type. The Instrument Runtime for each instrument creates and manages the tables according to a set of rules enforced by the Analog Test Environment. Each table is named, and the GUI expects each specific instrument client to contain tables with the prescribed names. For instruments that do not have any settings for a specific table, for example some more complex capabilities such as waveform modulation modes, there is a capability query mechanism that allows the instrument specific runtime layer to report whether it supports the capability. This is discussed further in the next section.

Using our Signal Generator Waveform Type example, a sample settings table class for the waveform type would comprise

- A name
- A description
- A settings table comprising entries that contain
  - A generic representation of the entry that is understood by the GUI
  - The instrument specific setting value that must be applied to the hardware by the runtime layer at instrument setup time.
  - A string that represents the meaning of the setting in the instrument's nomenclature.
  - A string that can be used by the GUI for context sensitive help.

The name is an agreed upon string between the GUI layer and the runtime layer so that the GUI can access the table by name. The description is used by the GUI in the display so that even the description seen by the user is expressed in the specific instrument's nomenclature. The table allows the runtime layer both to supply legal values to the GUI for display, and to provide the translation between a generic representation of the setting known by the GUI and the instrument specific value required by the specific instrument implementation.

All of this allows the GUI to display a meaningful set of options to the user in the specific instrument's nomenclature, while providing a common look and feel display for many different brands of instruments in a single application.

B. *Instrument Runtime Responsibilities*

Now that the GUI and the instruments are able to speak a common language without the GUI needing to know the specifics of the instrument, the GUI is able to communicate settings information and request specific actions of the instrument layer. Each instrument runtime layer provides a common interface comprising the following:

- IsSupported – instrument runtime layer reports whether a particular capability is supported.
- CanGenerateCode – the instrument runtime layer reports if it is capable of generating API code for the instrument settings.
- InitInstrument – instrument is initialized.
- ConfigureInstrument – Perform the instrument configuration specified in the instrument settings object.
- ConfigureTrigger – Perform the instrument trigger configuration specified in the instrument settings object.
- RunInstrument – Results in the instrument sourcing or measuring as it has been configured.
- StopInstrument – Ceases instrument operation.
- CloseInstrument – Ends the instrument session.
- Synchronize – Update the instrument settings to reflect the current state of the physical hardware.

The GUI is in charge of modifying the instrument settings object based on user preferences. It knows nothing about the meaning of these values. When the user requests an action such as running the instrument, the GUI passes the instrument settings object to the runtime layer that is associated with the instance of the GUI panel and the runtime layer takes care of applying the instrument settings to the hardware.

IV. Defining Common Capabilities

In order to accommodate all the various instrument features that are available, the Analog Instrument Environment needs to create a set of common capabilities. It then needs a way to query the instrument regarding what capabilities are supported.

Each instrument class defines a set of capabilities that the runtime layer for the instrument may support. This is a predefined list for each class and the list is known by both the GUI and the runtime layer. These capabilities are represented as properties in the instrument settings. Each runtime layer for an instrument can define which capabilities it supports. The GUI asks the instrument layer whether a capability is supported, and uses the response to determine how to populate the GUI. If a particular capability is supported, the GUI may ask the runtime layer for the settings table that contains the instrument specific definitions for the capability. As was described in a prior section, these table values are then used to update the graphical controls in the GUI. The following figures show screen shots from two different signal generator instrument panels. Note the similarities, and the differences between the screen shots. One has basic capabilities, the other shows the panel layout for an instrument with advanced capabilities. The Basic Signal Generator shown in Figure 4 supports only sine, triangle, pulse, and DC waveforms. The Advanced Signal Generator in Figure 5 adds square, ramp, double pulse, etc. In addition the Advanced Signal Generator adds Modulation and Generation mode capabilities. The Instrument Panel queries the Instrument Runtime for its capabilities and uses this information to determine which buttons to enable and which control groups to display.

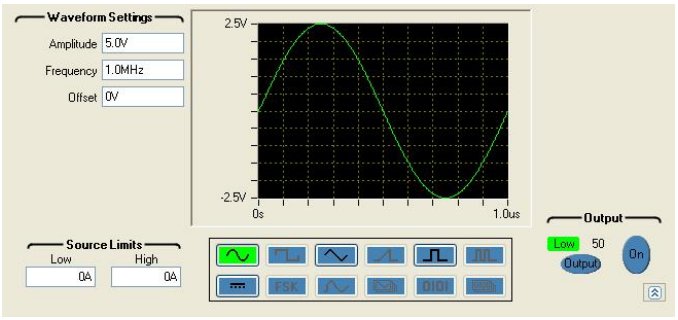

Figure 4. Basic Signal Generator

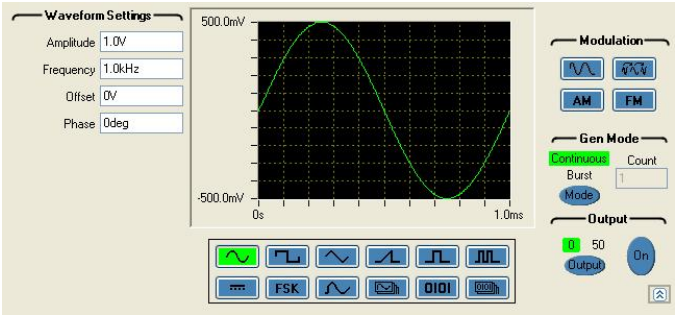

Figure 5. Advanced Signal Generator

Some capabilities imply sub-capabilities. For instance a signal generator capability might be modulation modes on a signal. Sub-capabilities might include AM, FM, whether modulation sweep is supported, whether modulation hopping is supported. Further if Modulation Sweep is supported there are sub-capabilities of Start Frequency, Stop Frequency, and Sweep Time. Further, there are required capabilities and optional capabilities that are influenced by this hierarchy. For instance, if Modulation Sweep is supported then Start Frequency and Stop Frequency must be supported. This hierarchy and the requirement to register each capability allows the GUI and runtime layers to grow over time as more instruments are included in the family of instruments supported by the Analog Test Environment.

V. Interactive Execution

The Analog Test Environment can be used as an interactive control panel for one or more physical instruments. The Analog Test Environment contains a runtime engine that will call methods in the runtime layers to initialize, configure and run a source or measurement cycle. The execution order of the instrument panels and their associated runtime layers can be controlled by the user. The user can select single or continuous execution within the runtime engine. In continuous execution mode, the instrument panels will respond similar to benchtop instrumentation.

The Analog Test Environment can also be used as a debugging window for an existing TPS. When used in this manner a user can open an instrument panel and connect it to an instrument that is part of a TPS. The instrument panel can be synchronized with the physical hardware. This way a user can step through their TPS in a separate Test Program development environment and observe the settings of the physical instrument in the graphical windows of the Analog Test Environment. The controls in the Analog Test Environment can be used to modify instrument settings or fetch measurement results.

VI. Use Within a Test Program

The architecture of the Analog Test Environment supports two methods for use in a Test Program; code generation and runtime execution.

A. Code Generation

The instrument runtime layers can optionally provide methods to generate code in the native API for the physical instrument. When the user selects “generate code” in the Analog Test Environment, each runtime layer is queried to determine if it can generate code and which languages are supported.

When the Analog Test Environment is placed in code generation mode, calls to the instrument runtime layers are redirected to the code generator for the runtime. The runtime layer may be directed to generate code in either ANSI C or C#. If an instrument panel is bound to an instrument runtime layer that does not support code generation, no calls will be made to its runtime while the Analog Test Environment is in code generation mode.

B. Runtime Execution

The runtime engine of the Analog Test Environment is available as a separate component. This allows the creation of a test container for the runtime engine. This test container can be used to load a saved session in the Analog Test Environment and call methods on the runtime engine to initialize, configure and run a source or measurement cycle.

Runtime execution will perform in a similar manner to interactive execution, but there will be no visible instrument panels. Runtime execution has the advantage that all instruments are available, even those that do not support code generation.

VII. Future Work

The current instrument panels represent the capabilities of a composite of instruments. Some of the more advanced instrument specific capabilities have been omitted to simplify the presentation in the instrument panel. In the future this will be addressed on multiple levels:

- Allow runtime layers to add to the menus of the Analog Test Environment
- Allow runtime layers to provide properties pages for display in the Analog Test Environment
- Allow runtime layers to provide graphical panels to add to the controls displayed in the instrument panels

All of these extensions will rely on the runtime layer to manage the additional settings. These settings will still be saved as part of a session in the Analog Test Environment.

VIII. Summary

Just as it is useful for test developers to have a common API for their instrumentation programming language, it is just as powerful to be able to provide a common GUI look and feel to the Soft Front Panels for those instruments. This environment provides the common look and feel GUI, and allows for a common debugging and code generation experience. This advance can help test developers to debug their TPS and provide a method to more quickly learn the capabilities and programming of new instruments.