Source of Acquisition
NASA Goddard Space Flight Center

Instrument Remote Control Application Framework

Troy J. Ames* and Carl F. Hostetter†
NASA Goddard Space Flight Center, Greenbelt, MD, 20771

The Instrument Remote Control (IRC) architecture is a flexible, platform-independent application framework that is well suited for the control and monitoring of remote devices and sensors. IRC enables significant savings in development costs by utilizing eXtensible Markup Language (XML) descriptions to configure the framework for a specific application. The Instrument Markup Language (IML) is used to describe the commands used by an instrument, the data streams produced, the rules for formatting commands and parsing the data, and the method of communication. Often no custom code is needed to communicate with a new instrument or device. An IRC instance can advertise and publish a description about a device or subscribe to another device's description on a network. This simple capability of dynamically publishing and subscribing to interfaces enables a very flexible, self-adapting architecture for monitoring and control of complex instruments in diverse environments.

Nomenclature

Italicized text represent names of interfaces or classes from the framework (e.g., *EventBus*).

I. Introduction

NASA Goddard Space Flight Center, led by the Advanced Architectures and Automation Branch (Code 588), has developed an extensible application framework for instrument command and control, known as Instrument Remote Control (IRC). The IRC architecture is a flexible, platform-independent application framework that is well suited for the control and monitoring of remote devices and sensors. Working with instrument engineers and scientists as well as past experience with distributed systems we have tried to come up with an architecture that balances simplicity and flexibility. The architecture has to be simple enough to use and maintain as well as flexible enough to be useful in a wide variety of applications and domains. The architecture emphasizes the capability to configure itself based on eXtensible Markup Language (XML) descriptions. There are descriptions to tell the framework which application components to plug in, what the Graphical User Interface (GUI) should look like, what devices to connect to and how to communicate with them, what algorithms to include in the application, and what interface to present to other peers. To enable a dynamic discovery and configuration capability for a collection of devices, each IRC instance can advertise and publish a description of itself on a virtual network. This simple capability of dynamically publishing and subscribing to interfaces enables a very flexible, self-adapting architecture for monitoring and control of complex instruments in diverse environments.

IRC enables significant savings in development costs by utilizing the XML descriptions to configure the framework for a specific application. Each of the descriptions will be outlined in greater detail in the following sections as we present the structure of the framework as well as the structure of a typical application using the framework.

II. Application Architecture

The IRC framework is implemented in Java for cross-platform portability. It is currently being used on Windows, Mac OS, Solaris, and several variants of Linux operating systems. The framework includes a growing library of components, algorithms, and visualizations making it possible to construct an application entirely from existing components. The framework also consists of several managers and factories which collectively are responsible for creating an application instance configured for a specific application. Each manager or factory is implemented by an interface, an abstract implementation, and a concrete default class. This common pattern in the

* Computer Engineer, Advanced Architectures and Automation Branch, Code 588.
† Computer Engineer, Advanced Architectures and Automation Branch, Code 588.

framework allows developers to easily create variants of built-in functionality or create new implementations and plug them into the framework. The existing managers and factories are designed to be general purpose and typically will work as is for most target applications. Example managers include, but are not limited to, a *ResourceManager* responsible for locating needed resources, a *PreferenceManager* that maintains user preferences, a *ComponentManager* for maintaining a list of current components in the application, and a *GuiFactory* that is responsible for constructing the GUI from an XML description. The *IrcManager* is the application manager responsible for initializing the application on startup and creating the other managers and factories as needed.

The high-level architecture of an application using the IRC framework consists of one or more components communicating information or data using the Publish-Subscribe[1] pattern. At this level there are two primary methods that components can communicate with each other, either through an *EventBus* or, for high data rates, through a *DataSpace* (Fig 1). The default *EventBus* implementation allows components to subscribe to all events or to those matching specified criteria facilitating a dynamic and flexible flow of information. This also isolates components from the transient nature of some components. Device proxies subscribe to receive message events from the bus that are directed to their external device and publish message events or data from their external device. GUIs can come and go based on user demands and may publish and subscribe to events on the bus.

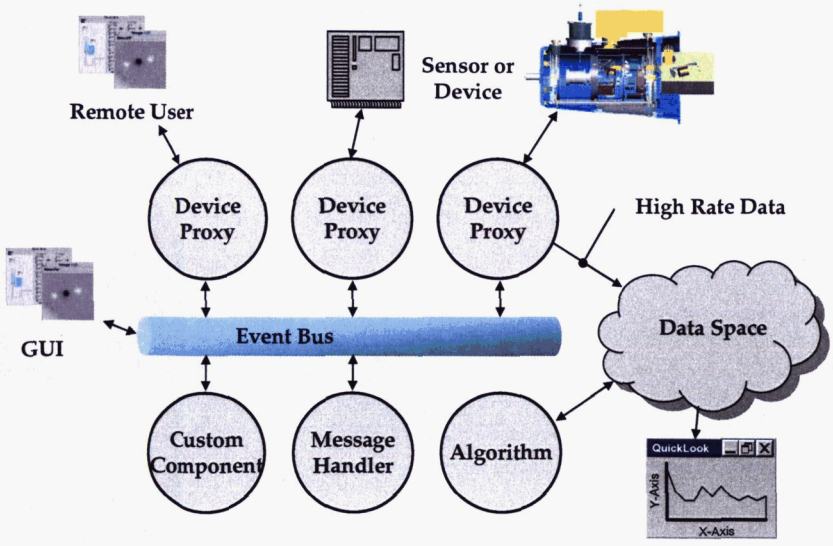

Figure 1. Representative Application Architecture.

The *DataSpace* is a catalog of published data products, called *BasisBundles* in IRC, available from data sources. A *BasisBundle* is a collection of one or more data buffers that share a common *BasisBuffer*. The common basis values in a *BasisBundle* typically represent time associated with the elements of each data buffer; however the basis can be any type such as a counter or frequency. The structure of a *BasisBundle* can be described with XML or dynamically constructed or changed. A *BasisBundle* publishes data written to it, by a data source, to all registered subscribers. The *BasisBundle* architecture is optimized for higher rate data with end-to-end throughput of hundreds of Megabytes per second achievable in some configurations.

Since the configuration of the framework for a specific application is centered on XML descriptions, we have developed corresponding XML schema definitions that enable an XML parser to validate the files, thereby guaranteeing that the descriptions are complete and correct. The following sections describe the XML based descriptions including the Type Map and Component descriptions, the Instrument description, and the GUI description.

A. Type Map Description

The Type Map description contains several tables of key-value pairs that map type identifiers to actual implementation classes. A default Type Map description is read by the framework on startup that defines the default implementations as well as a library of available components. This file is central to the framework in that many of the default implementations can be replaced by simply overriding entries in this file. The user can create an application specific Type Map that will override identical keys in the default or augment the map with new key value pairs. Listing 1 is a partial example of a type map file.

Listing 1. Example Type Map Description

```
<LookupTable name="GlobalTypeMap" >

    <NamespaceTable name="DeviceType">
        <Mapping name="Default" value="gov.nasa.gsfc.irc.devices.DefaultDeviceProxy"/>
        <Mapping name="Stateful" value="gov.nasa.gsfc.irc.devices.StatefulDeviceProxy"/>
    </NamespaceTable>
    <NamespaceTable name="ManagerType">
        <Mapping name="TaskManager" value="gov.nasa.gsfc.commons.processing.tasks.DefaultTaskManager"/>
        <Mapping name="DataSpaceManager" value="gov.nasa.gsfc.irc.data.DefaultDataSpace"/>
        <Mapping name="ScriptEvaluator" value="gov.nasa.gsfc.irc.scripts.DefaultScriptEvaluator"/>
        ...
    </NamespaceTable>
    <NamespaceTable name="ComponentType">
        <Mapping name="SkySubtraction" value="gov.nasa.gsfc.hawc.algorithms.SkySubtraction"/>
        <Mapping name="RawDataArchiver" value="gov.nasa.gsfc.hawc.archiving.RawDataFitsArchiver"/>
        <Mapping name="Demuxer" value="gov.nasa.gsfc.hawc.algorithms.DemultiplexAlgorithm"/>
        <Mapping name="Phaser" value="gov.nasa.gsfc.hawc.algorithms.Phaser"/>
        <Mapping name="Level0FitsArchiver" value="gov.nasa.gsfc.hawc.archiving.ChopRateFitsArchiver"/>
        <Mapping name="ClientMessageHandler" value="gov.nasa.gsfc.hawc.app.ClientMessageHandler"/>
    </NamespaceTable>
    ...
</LookupTable>
```

B. Component Description

A Component Description file defines a set of components for the framework to create and configure on startup. An example description is given in Listing 2. The framework uses the Type Map description to resolve the component types specified in a Component Description. For example the "Message Handler" component with type "ClientMessageHandler" will result in an instance of the class "gov.nasa.gsfc.hawc.app.ClientMessageHandler" being created.

Listing 2. Example Component Description

```
<?xml version="1.0" encoding="UTF-8"?>
<ComponentSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://aaa.gsfc.nasa.gov/cml cml.xsd"

    <Component name="SkySubtraction" type="SkySubtraction" start="true"/>
    <Component name="Message Handler" type="ClientMessageHandler" start="true"/>
    <Component name="Raw Data Archiver" type="RawDataArchiver"/>
    <Component name="Demuxer" type="Demuxer" start="true"/>
    <Component name="Phaser" type="Phaser" start="true"/>
    <Component name="Chop Rate FITS Archiver" type="Level0FitsArchiver"/>
</ComponentSet>
```

The content of a component description is used by the framework to configure the new component. The "start" attribute in the "SkySubtraction" component description tells the framework to start this component after it is created. The framework also passes the complete Component description to the created component allowing the component to configure itself in some component-specific way if desired.

Typically all components, including algorithms and custom components needed for an application, are specified in the Component Description, with the exception of Device Proxies which are handled by a Device description described in the next section. If the application has a GUI the user can add/remove or configure components dynamically. This is accomplished through Java's ability to dynamically load classes at run-time. Optionally this can also be done remotely by sending messages.

C. Instrument Description and Device Proxy Architecture

We developed the Instrument Markup Language[2] (IML) as a means to describe an instrument or device. IML is a vocabulary of XML dating back before XML became a W3C standard and has gone through much iteration over the years based on lessons learned. The attributes of a device that can be described by IML include:

- Device Proxy component to use
- Device subsystems
- State model component to use
- Logical message or command set
- Message arguments (including data types, valid values/ranges, and units)
- Message formats
- Logical script set
- Script arguments (including data types, valid values/ranges, and units)
- Logical data streams (e.g., science data, housekeeping, message responses)
- Data fields (including data types, valid values/ranges, and units)
- Data formats
- Communication mechanisms

The IML can describe a hierarchy of sub-devices, and each subsystem (sub-device) may use a different communication mechanism or protocol. For example, one subsystem may have a TCP/IP interface with binary messages and another subsystem may have an RS232 interface with ASCII messages. Each subsystem in the IML description is represented by its own *DeviceProxy*, which receives message objects, formats them according to the rules specified in the IML file, and then sends them to the actual instrument.

Although at this stage in the evolution of IML it is primarily the software engineers who are writing the descriptions, we envision the hardware engineers taking on this task. Not only do hardware engineers know the instrument details best, but they traditionally provide significant contributions to a formal Interface Control Document (ICD). The IML documents can serve a similar role – that is, communicating the intricate details of an instrument's interface – in a much more structured, formal, and easily manipulated way. An example IML file for an existing rover is given in Listing 3.

Listing 3. Example IML file

```
<?xml version="1.0" encoding="UTF-8"?>
<Device xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://aaa.gsfc.nasa.gov/iml iml.xsd"
        name="PgRover" displayName="PG Rover" type="Default">
    <Port name="Port" type="Simple" >
        <MessageInterface name="Rover Messages" >
            <Message name="?C" displayName="Toggle Camera">
                <Field name="cameraMode" displayName="Camera Mode" type="Integer" default="1">
                    < ListConstraint name="mode">
                        < Choice  name="on" displayName="On" value="1"/>
                        < Choice  name="off" displayName="Off" value="0"/>
                    </ ListConstraint>
                </Field>
            </Message>
            <Message name="!F" displayName="Forward">
                <Field name="Value" displayName="Value" type="Integer" default="10">
                    <RangeConstraint name="" low="0" high="47"/>
                </Field>/>
            </Message>
            ... <!-- Other messages not shown -->
        </MessageInterface>

        <OutputAdapter name="messageFormatter" displayName="Message Formatter" type="MessageFormatter">
            ... <!-- Output message format description goes here (see Listing 4) -->
        </OutputAdapter>
```

```
        <InputAdapter name="messageParser" displayName="Message Parser" type="SystemOut"/>

        <Connection name="TCP" type="TCP Client">
            <Parameter name="hostname" value="PgRover1.gsfc.nasa.gov"/>
            <Parameter name="port" value="23"/>
        </Connection>
        <Connection name="Standard Out" type="STDOUT" />
    </Port>
</Device>
```

The framework reads the IML for a specific device, in this case a "PG Rover", and based on this description creates the necessary *DeviceProxy* component. The "<Device>" root element in the example specifies a device of type "Default". The framework looks up the device type in the Type Map and an instance of the *DefaultDeviceProxy* class is created as the *DeviceProxy* component. This proxy component will then be passed the device description to configure itself. The *DefaultDeviceProxy* is a *Composite* component in the framework library that does not implement any device specific knowledge, rather it creates, manages, and delegates device specifics to subcomponents based on the IML description (see Fig. 2).

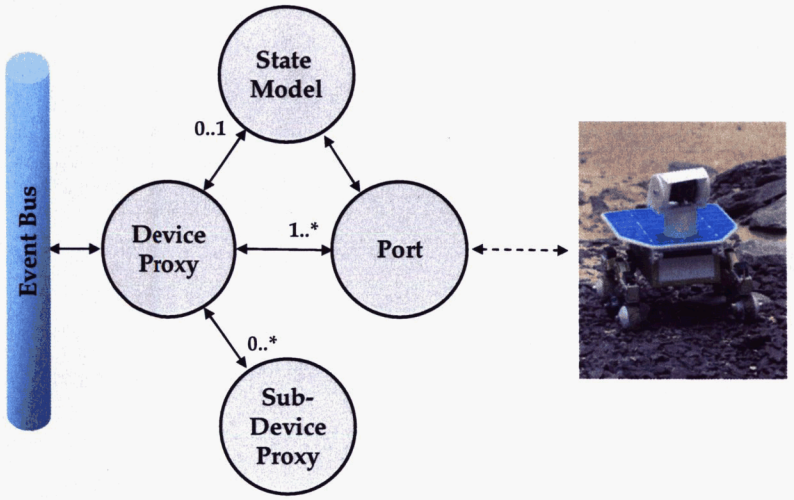

Figure 2. Default Device Proxy Component Structure.

The *DefaultDeviceProxy* class constructs one *Port* component for each "<Port>" element in the IML description, one *DeviceProxy* component for each nested "<Device>" element (if any), and a *StateModel* component if specified. Each component is connected as needed to listen for message events from the *DefaultDeviceProxy* class. The *DeviceProxy* receives messages from the *EventBus* and passes them onto all listening managed components. The *StateModel* can be configured to listen for messages going to the device as well as messages received from the device. Figure 3. shows the resulting component structure created for the "PG Rover" description.

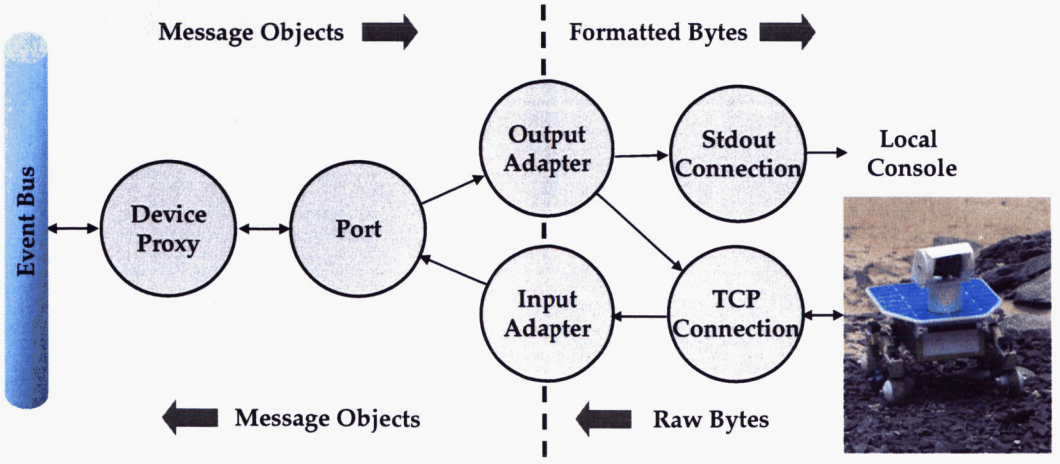

Figure 3. Rover Device Proxy Component Structure.

In the example shown in Fig. 3, the *DeviceProxy* receives rover messages from the *EventBus* and passes them onto the *Port* component. A *Port* component's primary responsibility is to manage the transition between internal *Message* objects to and from a device-specific representation of the message. The *Port* implementations available in the framework all accomplish this by creating and managing *OutputAdapter*, *InputAdapter*, and *Connection* components. A *Port* can also do some message filtering or validation based on the message descriptions, however since there is only one port defined in the rover description and it specifies a "Simple" type, all outgoing messages are simply passed onto the *OutputAdapter*. The *OutputAdapter* is responsible for converting a *Message* object into a buffer of bytes. The method for doing this depends on the implementation type of the *OutputAdapter*. Listing 4 shows the *OutputAdapter* description for the rover expanded to include the transformation rules for converting a *Message* into the ASCII bytes expected by the rover.

Listing 4. Output Adapter Example Description.

```
<OutputAdapter name="messageFormatter" displayName="Message Formatter" type="MessageFormatter">
    <Transformation>
        <Format>
            <!--
                We simply write out the message name followed by the
                value of each field in the input message as ascii values,
                terminated by a CR.
            -->
            <Record useDataNameAsInitiator="true">
                <Field applyToRemainingFields="true">
                    <Value type="printf" pattern="%02d"/>
                </Field>
                <Terminator value="&#13;"/>
            </Record>
        </Format>
    </Transformation>
</OutputAdapter>
```

The *OutputAdapter* type specified in the IML for the rover is "MessageFormatter" which resolves from the Type Map to the concrete class *MessageFormatter* from the framework component library. The *MessageFormatter* class uses the transformations given in the IML description to transform each message into bytes. In this case the transformation will result in short, four-character commands such as "!F23" for a "Forward" command or "?C01" to turn the camera on. The formatted bytes from the *OutputAdapter* are sent to all listeners.

In the case of the rover the listeners are two *Connections*. One connection simply writes out the bytes to the console for debugging purposes and the other writes all bytes received to a TCP socket connection. This decoupling of output format and connection medium allows the user to easily change either without impacting the other. To change the connection from TCP to a serial connection the type in the *Connection* description simply has to be changed from "TCP Client" to "Serial".

The responsibility of the *InputAdapter* is to do the reverse of the *OutputAdapter* and transform the bytes received by a *Connection* into a *Message* object or, in the case where the bytes represent a data stream, into the *DataSpace*. The IML description for the *InputAdapter* may contain parsing rules for doing this transformation. An alternative for both the *OutputAdapter* and *InputAdapter* is to plug in an implementation that performs the transformation in code.

Although the IML description of a device does not represent visual information, a message editor component does utilize the IML description to present the available messages or commands, and scripts for a device. The visual view of the PG Rover IML description from Listing 3 is presented in the message editor as in Fig.4.

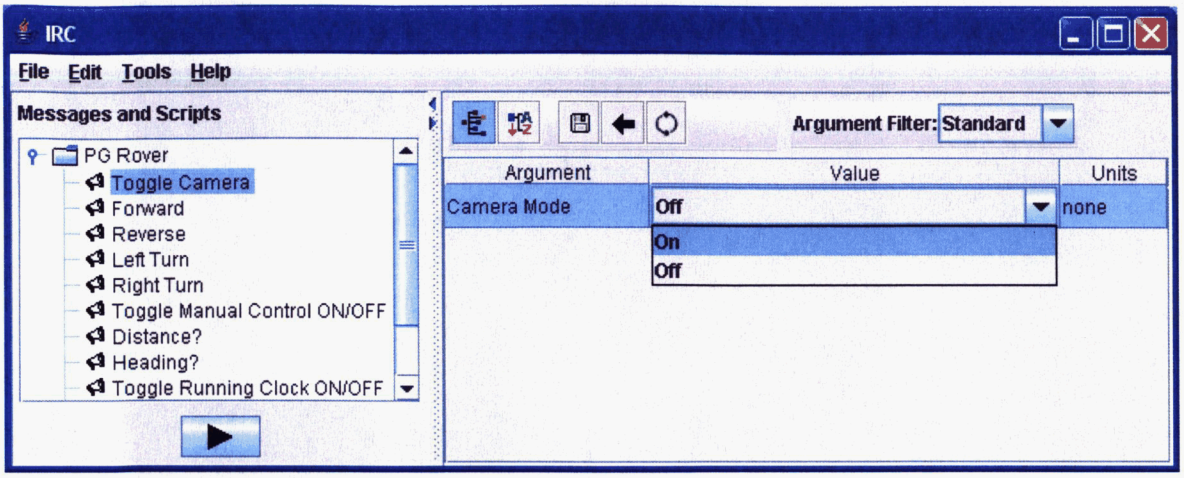

Figure 4. Message Editor representing the "PG Rover" IML description.

Although the rover example is fairly simple it does represent a real application of IML and is a common pattern seen for controlling a device. It only took 10 minutes to create the complete IML description and use it within the framework to control the PG Rover without any additional coding.

Since the instrument description is read at runtime, IML enables rapid iterative development and prototyping. For example, suppose the hardware engineer decides that he wants to make available a new rover command. He can simply define his new command in the IML and load the revised IML file, and the new command will appear in the GUI. No new code must be written, nor is recompilation necessary.

While there are many advantages to using IML, one of the most significant is the ability to defer some of the hardware implementation details as long as necessary during the development period. Software often needs to be developed in parallel with the hardware it is to control. Since hardware engineers may need to change various details as their subsystems are integrated, or as new hardware components with different characteristics are manufactured, it is crucial that the software architecture provide a degree of separation between the objects that represent the system and the hardware nuts-and-bolts.

D. User Interface Description

The IRC framework supports specifying the layout and content of the Graphical User Interface (GUI) using XML. While describing a User Interface is not new, many implementations take a minimalist approach by only mapping a few components that are common to several platforms or GUI libraries. This limits the flexibility and scope of user interfaces that can be created. The XML dialect that the IRC framework uses is an expanded version of SwiXML[‡]. A GUI generating engine parses the XML representation at runtime and instantiates the necessary classes to render the GUI. The GUI description can be loaded dynamically from a file packaged with the code, from a remote server, or from a remote device that has published a specialized GUI.

The XML dialect of SwiXML is closely tied to the Java Swing components contained in the Java Foundation Classes (JFC). For example the XML element "<frame>" corresponds to the *JFrame* Swing component and the "<panel>" element to the *JPanel* Swing component, etc. In addition an XML attribute of an element corresponds to a method call on the Swing component. For example in Listing 5 the attribute "title" in the element "<frame>" corresponds to the "setTitle" method in the *JFrame* class.

[‡] http://www.swixml.org/

Listing 5. Simple SwiXML description.

```
<?xml version="1.0" encoding="UTF-8" ?>
<frame size="640,480" title="Hello SWIXML World" DefaultCloseOperation="JFrame.EXIT_ON_CLOSE">
    <panel constraints="BorderLayout.CENTER">
        <label LabelFor="tf" Font="Comic Sans MS-BOLD-12" Foreground="blue" text="Hello World!" />
        <textfield id="tf" Columns="20" Text="Swixml" />
        <button Text="Click Here" Action="submit" />
    </panel>
</frame>
```

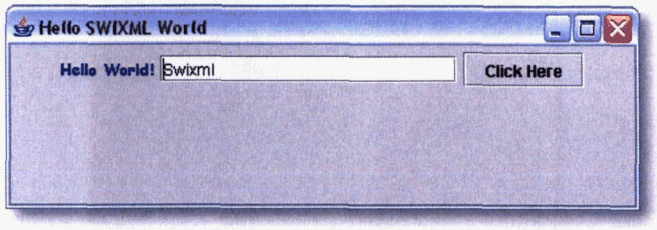

Figure 5. Swing JFrame rendering of SwiXML description.

SwiXML addresses the View in the common Model-View-Controller[3] (MVC) software pattern. The MVC pattern divides the user interface into three parts, the model represents the context, the view is one representation of that context, and the controller defines how the view reacts to user interaction. Since SwiXML is limited with respect to the models and controllers, we have extended the syntax to support instantiating models and controllers and associating them with one or more views. In Listing 6 a "<ControlClass/>" element instantiates a controller that is then associated with a panel and popup menu by a "controlclass" attribute. The controller identified in this example as "Tree_Controller" will be registered to user events from the tree view component and popup menu.

Listing 6. GUI description using the ControlClass Element.

```
<frame id="Browser_Frame" name="frame" size="600,400" title="Component Browser" layout="BorderLayout">

    <ControlClass id="Tree_Controller"
            class="gov.nasa.gsfc.irc.gui.browser.ComponentTreeController"/>

    <splitpane Orientation="1" DividerLocation="250"
            BottomComponent="Property_Table_Panel" TopComponent="Tree_Panel">
        <panel id="Tree_Panel" Layout="borderlayout">
            <scrollpane>
                <tree id="Component_Tree" SelectionRow="0"
                        initclass="gov.nasa.gsfc.irc.gui.browser.ComponentTreeModel"
                        controlclass="Tree_Controller" constraints="BorderLayout.NORTH">
                    <popupmenu controlclass="Component_Tree_Controller">
                        <menuitem text="Start" controlclass="Tree_Controller" actionCommand="START"/>
                        <menuitem text="Stop" controlclass="Tree_Controller" actionCommand="STOP"/>
                        <separator/>
                        ... <!-- Other menu items not shown -->
                    </popupmenu>
                </tree>
            </scrollpane>
        </panel>
        ... <!-- Other items not shown -->
    </splitpane>
</frame>
```

The rendering of Listing 6 is shown below in Fig. 6 with the popup menu displayed. There is a similar mechanism for instantiating a model for a view. A GUI description can specify multiple models and controllers linked to GUI objects. These simple extensions enables a self contained MVC description that can be instantiated, or included in a larger GUI description, or even published to remote clients.

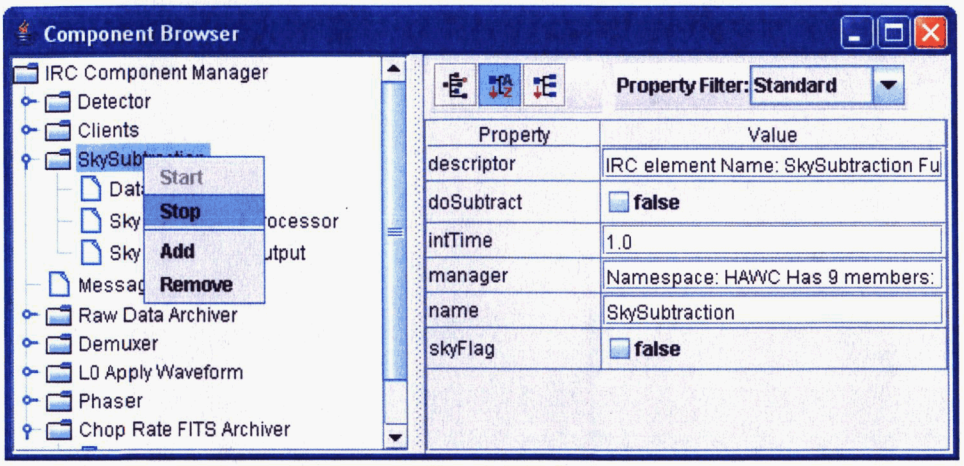

Figure 6. Swing JFrame rendering of the "Component Browser" GUI description.

We have also extended the description syntax to support the frameworks library of visualization components. A visualization can be described in XML and constructed using one or more light-weight renderers that are combined or overlaid to create a complete custom visualization. A visualization can also reference a Scalable Vector Graphics (SVG) description to create a dynamic data driven interface. Figures 7 and 8 are examples of these types of visualizations that can be described in XML and constructed with the IRC framework.

The IRC framework has several predefined GUI descriptions for view components such as the "Component Browser" previously shown in Fig. 6, and the "Message Editor" in Fig. 4 that can be customized or used as is.

E. Distributed Architecture

The previous sections described the internals of the IRC Framework and how an IML description is used to communicate with an instrument. With a distributed environment we need to take a broader view of an IRC architecture based on multiple instances of the IRC framework. A single IRC instance can use IML descriptions in two different contexts. The first, as outlined in section C, is a description of the private interface to an instrument that a device will be communicating with. The second context is a description of a device's own public interfaces that other external clients can use to communicate with it.

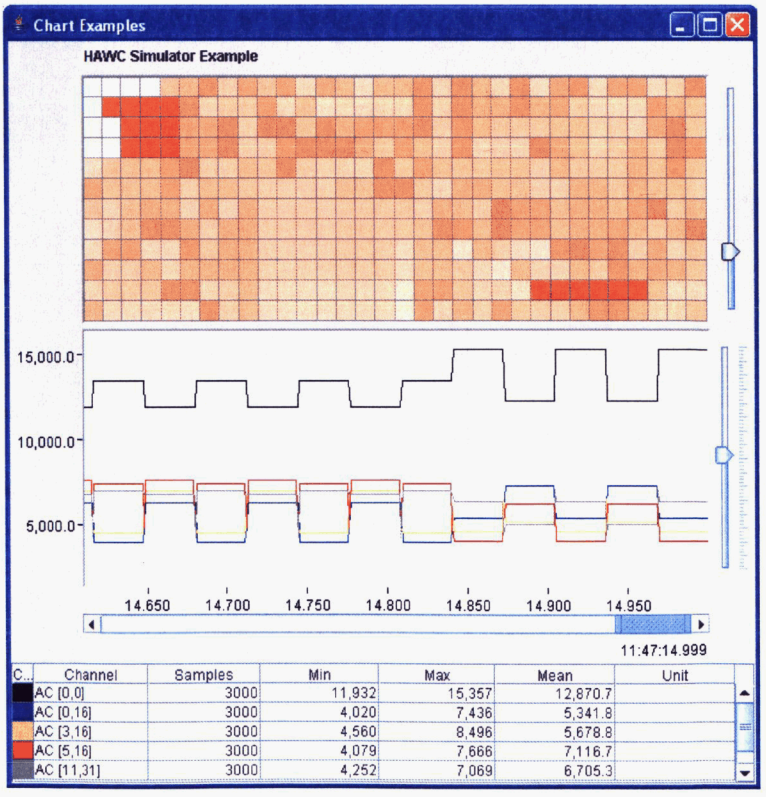

Figure 7. Example visualizations.

The IRC framework provides the ability to dynamically discover and communicate with other devices anywhere on a network in a peer-to-peer manner. To enable a dynamic discovery and configuration capability for a collection of devices, each IRC instance (referred to as an IRC Device) can advertise and publish information about itself on a virtual network[§]. A virtual network allows devices to communicate and organize independently from the physical network. For scoping and security the virtual network can be divided into virtual peer groups. Devices can join or leave a virtual peer group and thus join or leave the instrument control environment of IRC.

An IRC Device can advertise and publish several public IML interface descriptions. A device may want to split up its public descriptions (commanding vs. data) or publish more than one version (novice vs. expert). An IRC Device can also publish a GUI description for a customized command panel or visualization of the device.

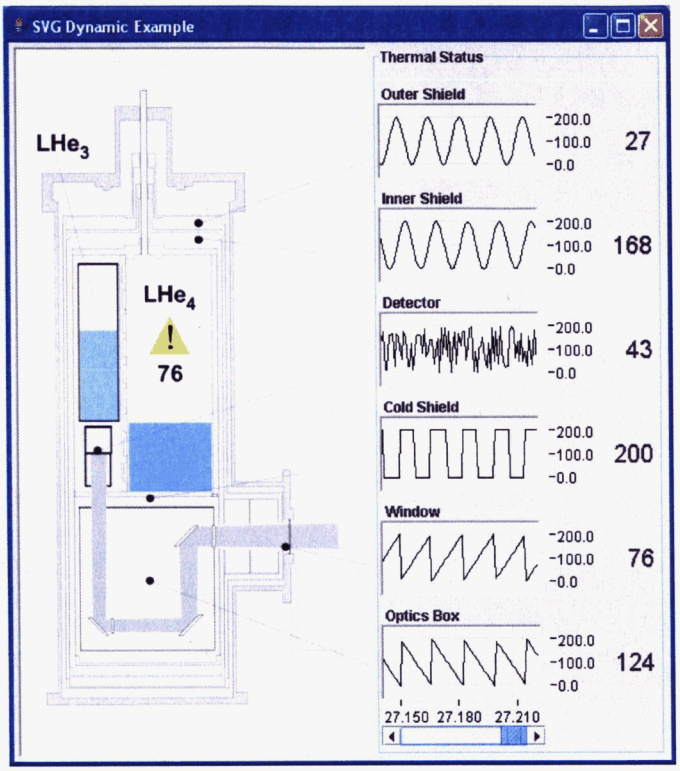

Figure 8. Example SVG visualization.

1. Example Distributed IRC Architecture

The High-resolution Airborne Wideband Camera[**] (HAWC) instrument will be a facility instrument for NASA's Stratospheric Observatory For Infrared Astronomy[††] (SOFIA) mission, a Boeing 747SP aircraft modified to accommodate a 2.5m reflecting telescope. The HAWC control and monitor software will be configured in a distributed hierarchal peer architecture as illustrated in Fig. 9.

Each of the subsystems (ADR, Thermal, Optics, etc.) will have dedicated IRC devices to control them and function as subsystem proxies. The proxies will primarily encapsulate and perform subsystem-specific functions and advertise the subsystem on the network to a "HAWC Subsystem" peer group. The type of specific functions that each proxy may perform includes but is not limited to closed loop control, data translation or calibration, and command translation. The "HAWC" IRC Device will join the "HAWC Subsystem" peer group as a trusted peer and request the published IML interface for all subsystems. The "HAWC" device will also join a "HAWC Instrument" peer group and publish its IML interface to the group. Astronomers and engineers will be able to start client IRC Devices anywhere on the network, join the "HAWC Instrument" group, and request the public IML description from the "HAWC" device. The "HAWC" device may publish more than one version of the interface depending on the type or authorization of the user.

2. Distributed IML Example

Using the distributed architecture of IRC, IML descriptions can be much more independent of the physical location of the devices. Listing 7 shows a complete description that the HAWC device will use to connect with each subsystem.

[§] IRC transitioned from an in-house developed peer-to-peer framework called WorkPlace to JXTA (see http://www.jxta.org). JXTA defines a set of open XML protocols for finding and organizing a virtual network of peers.

[**] http://astro.uchicago.edu/hawc/hawc.htm

[††] http://www.sofia.usra.edu/

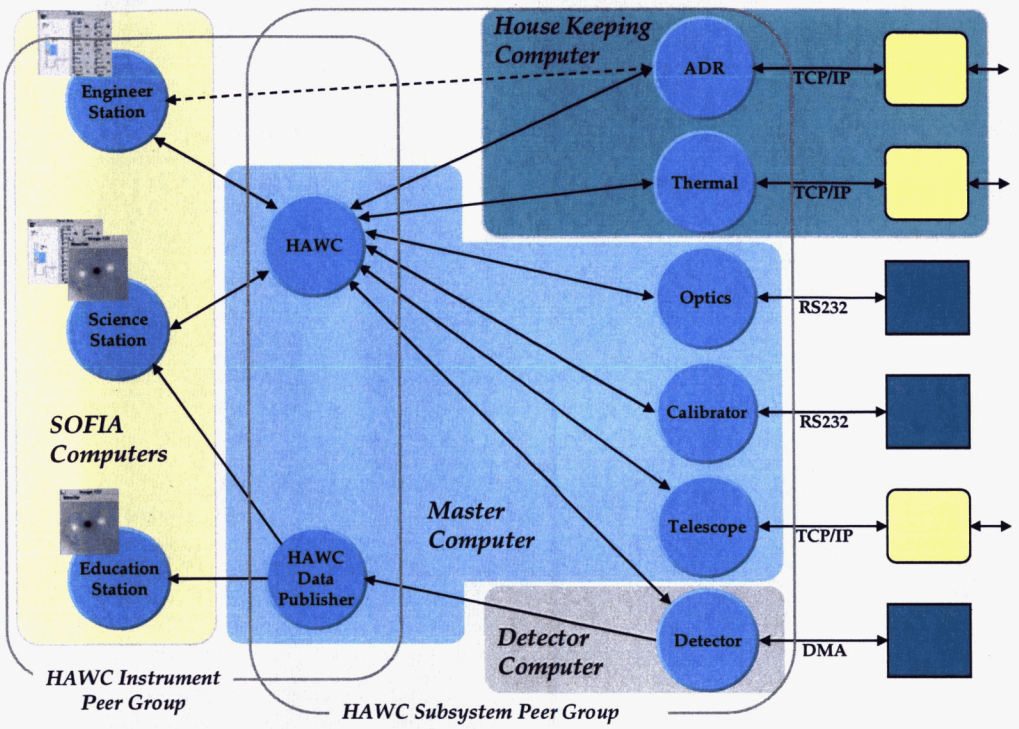

Figure 9. HAWC IRC Device Architecture.

The dynamic discovery mechanisms of IRC are used to find the IML associated with each subsystem. With this approach, the HAWC instrument can be unaware of the physical location of the other peers on the network. It simply knows the names of the peers. Other variants of this allow for IRC to search for all peers in a particular group or for all peers of a particular name, regardless of the group.

For the HAWC instrument the environment changes as do the control needs when the instrument is rolled on the airplane versus a test lab before flight. For example, the description of the telescope that the software will receive on the airplane will describe how to communicate with the actual telescope, while on the ground in the lab the description could be for a telescope simulator or some other piece of test equipment.

Listing 7. IML Description of HAWC Device and Subsystems.

```
<Device name="HAWC">
    <DevicePeer group="HAWC Subsystem" description="Optics"/>
    <DevicePeer group="HAWC Subsystem" description="ADR"/>
    <DevicePeer group="HAWC Subsystem" description="Thermal"/>
    <DevicePeer group="HAWC Subsystem" description="Calibrator"/>
    <DevicePeer group="HAWC Subsystem" description="Detector"/>
    <DevicePeer group="HAWC Subsystem" description="Telescope"/>
</Device>
```

F. Algorithms

IRC provides several general-purpose algorithms, and aims to make it easy to develop instrument-specific algorithms. Taking advantage of Java's dynamic class loading, the IRC framework does not have to know about the algorithm implementation class until runtime; by referencing the location of the Java byte code, the framework is able to create instances of algorithms as needed. Also, by using the Java Native Interface (JNI), algorithms can be implemented in any native language such as C, C++, or FORTRAN.

To simplify the implementation of custom algorithms the framework has class implementations and interfaces that provide the algorithm architecture shown in Fig. 10. *Algorithm* is a composite object that manages a set of *Input*, *Output*, and *Processor* Objects. The *Input* and *Output* components know how to interact with the *DataSpace*. The *Input* component can filter, down-sample, or queue the data for the algorithm if needed. The *Processor*

component is the algorithm implementation and is typically the only component from the algorithm architecture that needs to be developed for a custom algorithm.

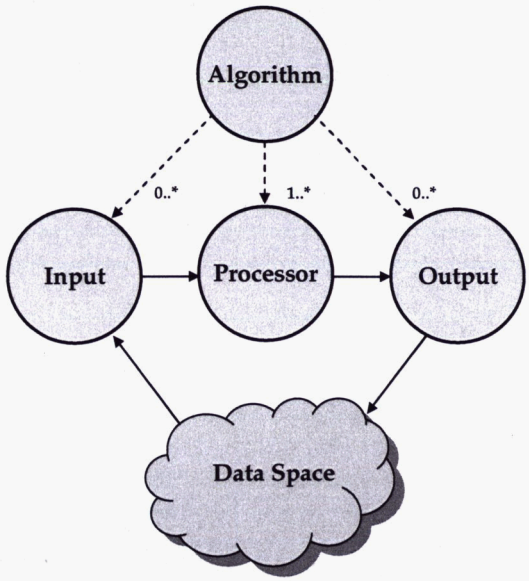

Figure 10. Algorithm Architecture.

G. Scripting

The ability to write scripts to embed in the instrument control software is an important feature of the IRC framework. It provides the user with a way to sequence common tasks. Currently, scripts must be written in Jython[‡‡] or JavaScript; however, the IRC architecture allows for support of any scripting language that supports the Bean Scripting Framework[§§]. Jython is a Java implementation of Python, an interpreted, object-oriented programming language

A script that configures or commands an instrument can be written easily. Such a script is shown in Listing 8. This script sends two messages to a device. The "getMessageDescriptor" call returns the IML descriptor of the "setRegister" method for the "Detector" device. The descriptor uniquely identifies a specific message including any specification of arguments and constraints. This descriptor is used by the "publishMessage" call to validate the arguments before actually publishing the message to the *EventBus*.

A script can also prompt the user for input, and can add, remove, and configure Algorithms. Support for looping and control flow is included. Using more advanced capabilities of Jython and JavaScript, a script can extend the IRC framework in interesting ways, since they have access to all Java packages and can extend Java classes. These features have been used to create scripts that implement algorithms that connect themselves to the *EventBus*, issue commands based on the analysis of incoming data, and then remove themselves from the framework.

Listing 8. Sample Jython Script.

```
REG_TM_RST_FIBR = 257
REG_TM_CLK = 263

descriptor = getMessageDescriptor("setRegister.SI1.Detector")

# Reset fiber interface
publishMessage(descriptor, REG_TM_RST_FIBR, 0)
# Set clock to 15 KHz
publishMessage(descriptor, REG_TM_CLK, 15)
```

To make a script available to the system, a fragment of IML must be created that describes the script, its arguments (including data types and valid values), and any documentation for the script. The IML fragment shown in Listing 9 can be added to a library of scripts or to the description of a subsystem to make the script appear as a primitive command to the user.

Listing 9. Sample IML Script Element.

```
<Script name="resetFiberOpticLink"
   displayName="Reset Fiber Optic Link"
   description="Resets the fiber optic link."
   file="detector/resetFiberLink.py"
   language="Jython"/>
```

[‡‡] http://www.jython.org/
[§§] http://jakarta.apache.org/bsf/

III. Role in Missions

IRC is well suited for the control of instruments and devices and is typically used as the primary means of remotely controlling a device; however it can also be used within the context of a larger mission system or framework. The most common roles for IRC in a mission framework are the following:

1) As an adapter between the mission framework and devices. In many cases it is not feasible to modify an existing device, sensor, or application to be compatible with a mission framework. IRC and an IML description can facilitate the integration of a wide variety of devices and applications.
2) As a customizable user interface to the mission framework or to devices and applications attached to the system. IRC provides a flexible architecture for describing a user interface using XML as well as a library of visualization components.
3) As an application framework for building customized client or service applications connected to the mission framework. The XML based component description in the IRC framework allows an application to be assembled and configured from a library of plug-in components. IRC has been used in this way for creating device simulators, autopilots, environmental monitors, and smarter devices.
4) As a standalone controller for test equipment or subsystems supporting lab tests and validation.

IV. Future

The IRC framework has evolved based on lessons learned applying it to device control to minimize the amount of custom, device-specific development that must be completed. One area that still requires custom code for complex devices is representing and operating on device state. IML does have a "<StateModel>" element that allows the developer to plug in and connect a device state model component, however the framework and IML provide only limited help in implementing or describing a device's state. Currently this is done by coding custom models either as algorithms or components. An important enhancement to the IRC framework will be to enable behavior state models described by XML to be assembled and synthesized into the framework. This will also provide the ability to quickly develop models that can be used for simulating device operations with whatever degree of fidelity is deemed necessary. Simulations also allow many activities to be performed long before instrument development has been completed. Instrument designers can develop, validate, and modify designs quickly and efficiently. Scientists can begin science planning and data analysis algorithm development; data archival, retrieval, and publication scenarios can be worked out; and support staff can begin training for instrument operations very early in the program. These state or simulation models can also be used operationally for detecting fault conditions and recovery.

V. Summary

The IRC framework design supports a high degree of configurability, allowing it to be tuned for specific domains. Processes can be run on a single computer or on multiple heterogeneous computers, ranging from small, low cost hardware components to high-end workstations. Processes can be run either locally, at an observatory for example, or remotely over the Internet (or both). This provides an instrument development team the flexibility to use the hardware components that best fit the operating environment and instrument requirements. The framework supports the cross-platform migration of functions and necessary reconfiguration if these requirements change. This flexibility enables a design in which small, embedded software components are placed at the point of origin of the generated data (smart sensors) and at the point of device control (smart actuators). The configurable IRC framework enables these software solutions to be easily developed, enhanced, maintained, and reused for different devices, different instruments, and different domains.

The IRC framework, utilizing descriptions in XML, supports instrument development from early design through operations and maintenance to minimize software development time, minimize development costs, maximize reuse of software components, and maximize flexibility of instrument architectures. The plug-in nature of the framework maximizes the ability to incorporate emerging technologies. Thus, as new instruments are added to a system, or as specifications for existing instruments are modified, or as new requirements are added, the effort to adapt the software to these changes will be incremental rather than major.

IRC has been successfully used to control, monitor, or simulate instruments from simple sensors, lab equipment, sensor webs, autonomous boats, to large telescopes.

References

[1] Gamma, E., Helm, R., Johnson, R., and Vliddides, J, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass. 1995.

[2] Ames, T. J., Sall, K. B., & Warsaw, C. E.," NASA's Instrument Control Markup Language," *Astronomical Data Analysis Software and Systems VIII*, eds. D. M. Mehringer, R. L. Plante, & D. A. Roberts, ASP Conf. Ser., Vol. 172, Astronomical Society of the Pacific, San Francisco, California, 1999, p103.

[3] Krasner, G. E., and Pope, S. T., "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, August/September 1988, pp. 26-49.