Symbolic Ai

Design Guide

Version 2.0

September 17, 2020

ILS Automation, Inc.

Table of Contents

1. INTRODUCTION

The *Symbolic Ai* is designed to detect, manage, annunciate, and respond to events. The output of *Symbolic Ai* is a diagnosis, a recommended response to the problem.

Symbolic Ai is implemented using Ignition™ from Inductive Automation. Ignition is an extensible platform that provides OPC connectivity, database integration and a graphics library for interface development. It is designed to be extensible via Python scripting and custom Java modules. *Symbolic Ai* consists of a Java module, Vision windows, Python scripts, User-Defined Tag templates (UDTs), and a SQL*Server database schema.

1.1 Licensing

1.1.1 Ignition

A *Symbolic Ai* installation requires a commercial Ignition license to be obtained from Induction Automation.

1.1.2 Toolkit

Symbolic Ai is the joint property of ILS Automation and ExxonMobil. Any applications developed by an end customer based on *Symbolic Ai* remain the property of the end customer.

The application is heavily dependent on various open-source packages. The packages are listed in the license statement that is displayed when loading the module. Open source packages have been carefully selected to include only those with licenses that allows free and unfettered use of the package.

1.2 Prerequisites

1.2.1 Java

Ignition 7.7 requires Java JDK1.8. It is the customer's responsibility to install the Java JDK on any system that will be running the application.

1.2.2 Eclipse

For development support, compilation of Java 1.8 code requires Eclipse Luna 4.4 or newer. Any systems to be used for development require a Java SDK installation.

1.2.3 Ignition

This application requires Ignition 7.7.0 or newer.

1.2.4 Internationalization

There is no requirement for localized text. The application will be presented in English.

2. ARCHITECTURE

An Ignition project implements, by its very nature, a client-server architecture. The server is called the “Gateway”. It supports autonomous processing without need for clients.

Client views are provided for observation and/or control of the application. Clients can either co-reside on the server platform or be remote.

The Ignition platform is typically customized using Python scripting combined with Ignition-provided user-interface widgets. The specific Python implementation is 2.6 via a Java-based translator called Jython 2.5. Since Jython is Java-, it has the ability to utilize Java modules, classes and methods in a very straight-forward manner. Python/Jython scripts execute either within a client/designer or gateway scopes. Client scripts are intended for functionality associated with user-interaction. Gateway scripts execute without user-interaction.

Additionally, the Ignition platform is extensible via customized Java packages called modules. Based on the Ignition Software Development Kit (SDK), modules are tightly integrated with other parts of the platform. For the purposes of the toolkit, a module was developed to take advantage of Ignition’s block-and-connector interface and to implement the execution engine as a completely autonomous entity. The *Symbolic Ai* executes completely within the Ignition gateway scope, but offers hooks via RPC calls to designer and client scopes as well. It also provides classes that support the Designer-view of a diagram.

The following diagram shows a functional breakdown and communications between the application scopes. Code written as part of the Java module is shown in blue, Python in green

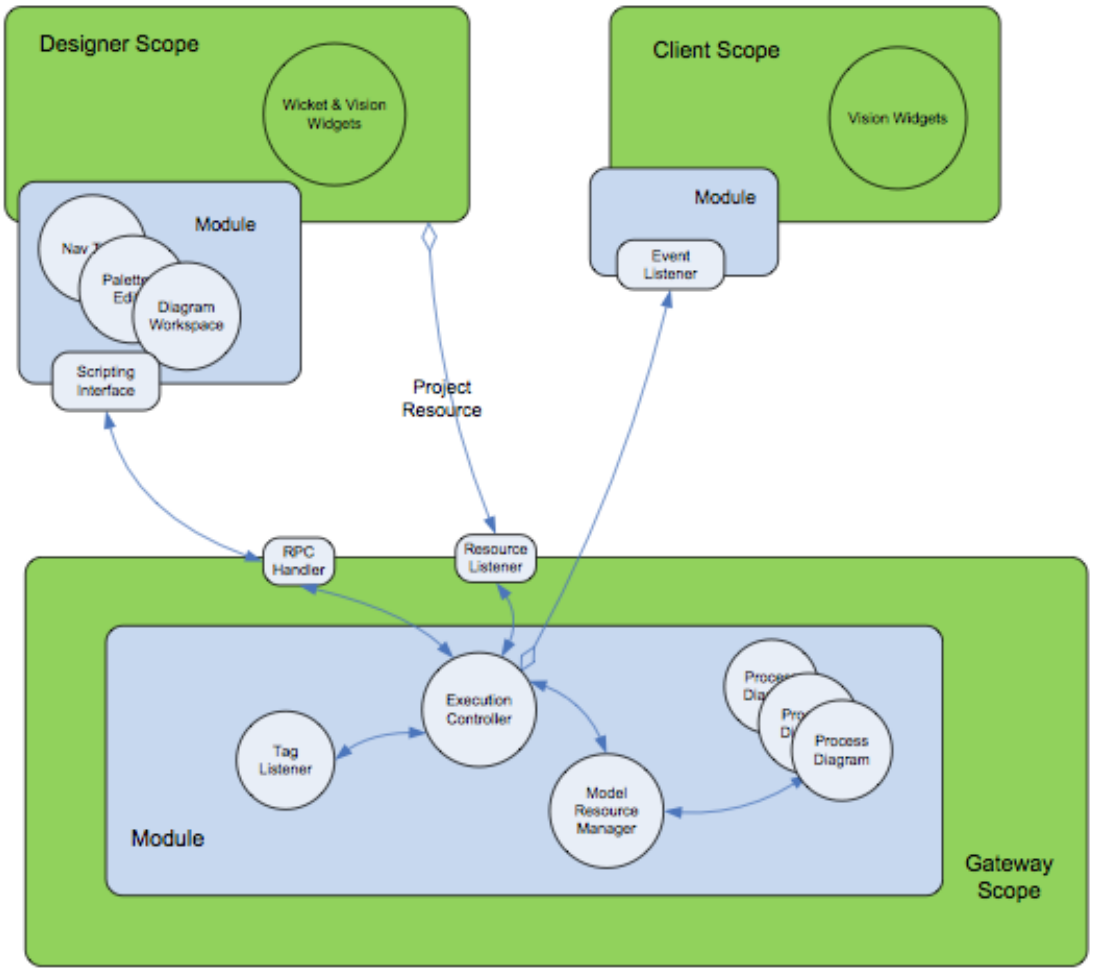


Figure 1 – Collaboration Diagram

2.1 Prototypes

The design decision for the current architecture came about after testing of a series of prototypes.

An initial prototype was developed based on the Vision™ graphics module that is distributed as part of the Ignition platform. This prototype ran into serious, time-consuming obstacles. Namely that the concept of connections was not supported and that the graphics classes were not easily extensible.

A second prototype was developed based on the JgraphX open-source graphics package. It was highly extensible and supported the concept of connections but only at a very primitive

level. A high level effort would have been required to complete the customizations necessary to meet the toolkit requirements.

The final architecture makes use of a new feature of Ignition 7.7, blocks-and-connectors. This interface has proven to be easily customizable and is well-integrated into the Ignition framework. It has the additional advantage of being supported by Inductive Automation. It is made available as part of the Ignition core platform at no additional cost.

2.2 Serialization

Serialization refers to the process of converting Java objects into a format suitable for storage and/or network transmission. The process of recovering the Java objects from storage is called “serializing”.

Serialization is an important consideration in the toolkit design because, unlike the legacy application, the Ignition scopes (Designer, Client and Gateway) are (or can be) hosted on separate systems. They do not share virtual machines. Serialization of the model is required to synchronize between scopes. It is also used when projects are saved.

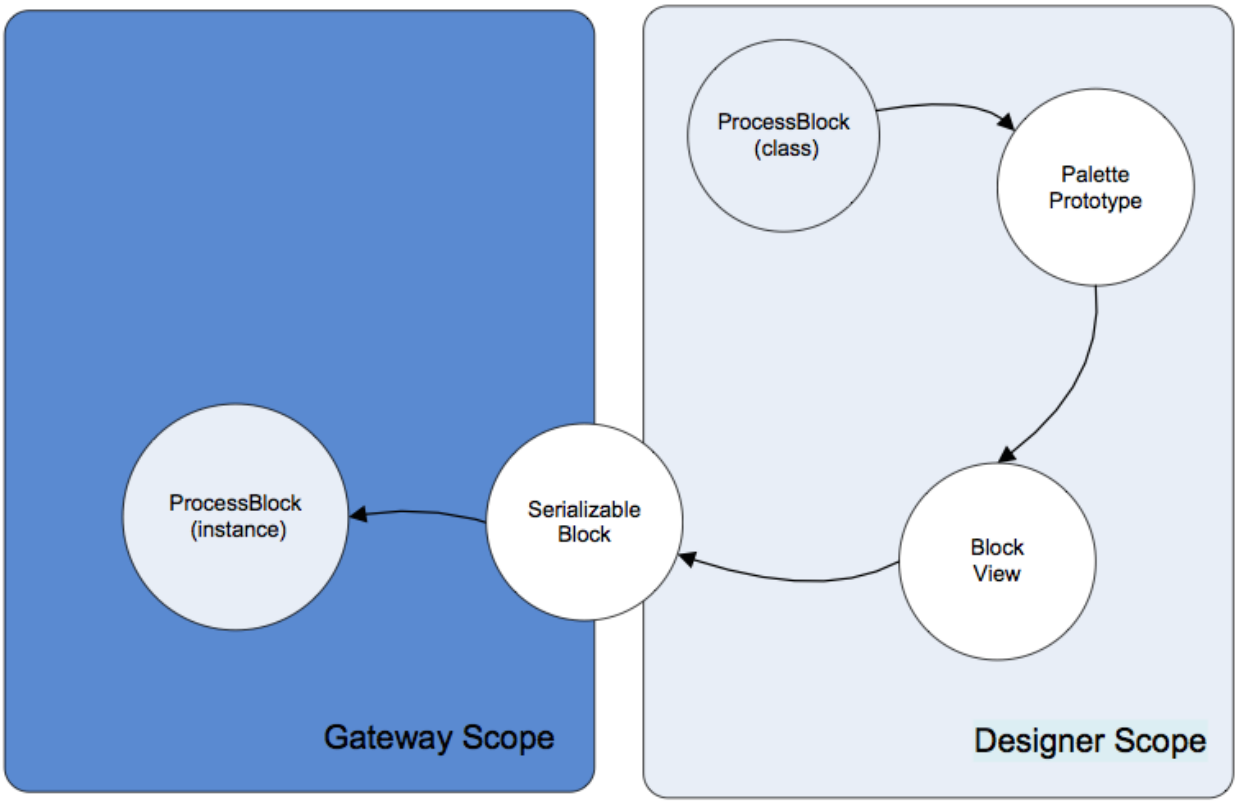


Figure 2 – Serialization Cycle

From a macro level, an Ignition project follows a standard Model-View-Controller paradigm. Objects in the Designer support the view, objects in the Gateway are the model, and Ignition, itself, is the controller.

The figure above illustrates the sequence of operations involved in following a processing block through its different environments.

1. Palette Prototype. When the Designer creates the block palette it queries the Gateway for a list of block classes that are annotated as "*ExecutableBlock*". From these it obtains prototype objects that contain enough information to create viewable blocks in the Designer view of a diagram.

   Each prototype contains:
   a. Block class
   b. Icon to use in the palette
   c. A label
   d. Basic rendering style (Square ,Diamond ,Entry, Circle . . .)
   e. Anchor points (stubs)
   f. Icon embedded in drawing, if any
   g. Label rendered on drawing, if any

2. Block View. Selection from the palette creates a *ProcessBlockView*. This is the visible rendering of a process block. When editing a block, the model in the Gateway is queried to obtain a list of properties for that block. The property list is not transferred via the palette prototype.
3. Serializable Block. When a "save" operation is requested, the *ProcessBlockView* objects are converted into *SerializableBlock* instances and transferred to the Gateway. Serialization is required because the transfer may span different systems. If a project is exported, the export file contains *SerializableBlock* instances.
4. Process Block. When the Gateway receives a project update, the *SerializableBlock* instances contained in the project resource are converted into *ProcessBlock* instances and added to the "live" diagram.

Ignition provides an XML-based mechanism for serialization. However, this was not compatible with Java-generics (strongly typed lists) and was abandoned for that reason.

JavaScript Object Notation (JSON) is an alternative text-based solution. Serialization/deserialization is handled by an open-source package named "Jackson". It is distributed with an Apache license. JSON is used for all serialization within the toolkit.

2.3 Custom Blocks

The design supports user-extension of the block repertoire built into the BLT module. These user-designed blocks are written in Python and stored within a separate Ignition project that can be viewed as a library of custom blocks and may be re-used in multiple applications.

2.4 Model Definition

In a Model-View-Controller design, the model contains the definition of the core data structures, in our case, a block diagram. For the purposes of this application, the *model* is a *ProcessDiagram* object. It completely describes a diagram and can be used to render its display. It becomes serialized into a project resource. This resource is the single unifying structure that binds knowledge of a diagram among all three Ignition scopes.

In the Designer, the *File->Export* dialog lists all the resources connected with the current project. This dialog has been modified to include the custom toolkit model resource.

2.5 Auxiliary Data

The toolkit may require accompanying information that is not used in the execution of the blocks themselves (e.g. database entries). As appropriate, a block definition may include specification of a custom editor to define and modify these properties. Auxiliary data are not visible except through these editors. We assume that these data are present in both the toolkit blocks and an external system (database). The question arises as to how to keep the two entities in-synch with each other. Issues arise when moving a toolkit diagram from one system to another.

“ExternalData” is a Python-friendly dictionary structure that can be part of any block. This auxiliary configuration information in no way affects the execution of the block. The block is simply used as a convenient repository when moving a diagram from one system to another.

Consider the following cases:

2.5.1 Normal operation/Editing

Under normal circumstances, the external system (database) and toolkit are presumed to be in-synch. There is no special mechanism to keep them that way. The operating premise is that the normal way to edit these attributes is with the custom editors provided as part of the toolkit. On “Save” these editors will update both the external system and the auxiliary data stored in the toolkit. If the external system is edited directly, this is considered to be “out-of-bounds”. The person making the edits is also then expected to select a “Refresh Auxiliary Data from Database” menu choice on each affected application in the toolkit to synchronize the toolkit.

2.5.2 Transfer a project

A project containing toolkit resources is moved to a different environment. Perhaps this is from production to off-site for test/debug/analysis. Perhaps this is from a development area to a production system. Perhaps the database connection has been redefined in the Gateway.

The toolkit has no way of "knowing" when the project is started whether or not this is a different environment from when it was last saved. In this scenario, the user must explicitly select a "Refresh Database from Auxiliary Data" menu choice on each application in the toolkit where a refresh is desired.

2.5.3 Export/Import

In a similar manner to the situation described above, an exported diagram or entire application tree is moved between environments. However instead of the transfer being inside an Ignition project, the application has been exported in .json format from one system and imported into a second. In this instance, the toolkit framework automatically refreshes the database on the second system.

2.5.4 Isolation Mode

On a transition to "Isolation" mode, the toolkit framework automatically updates the isolation-mode database from the information stored in the diagram. When returning to "Production" mode, no action is taken. The production database remains unchanged.

2.5.5 New Blocks

A required part of creating new block instances in the toolkit is to visit any associated custom editors. Other than completing configuration through these editors, there is no additional work required on the users' part to synchronize the external database with the diagram.

2.6 Gateway Scope

For the purposes of the toolkit, the *Gateway* contains the engine that executes the logic blocks. Block logic is retained in separate Java or Python class instances. These classes are the focus of any custom development to extend the toolkit.

This design allows the block logic to be saved and restored just as any other project resource. The gateway is a listener on project resource changes. This is the mechanism for remaining in synch with the Designer. A project update is transmitted to the Gateway upon a "save" action in the Designer.

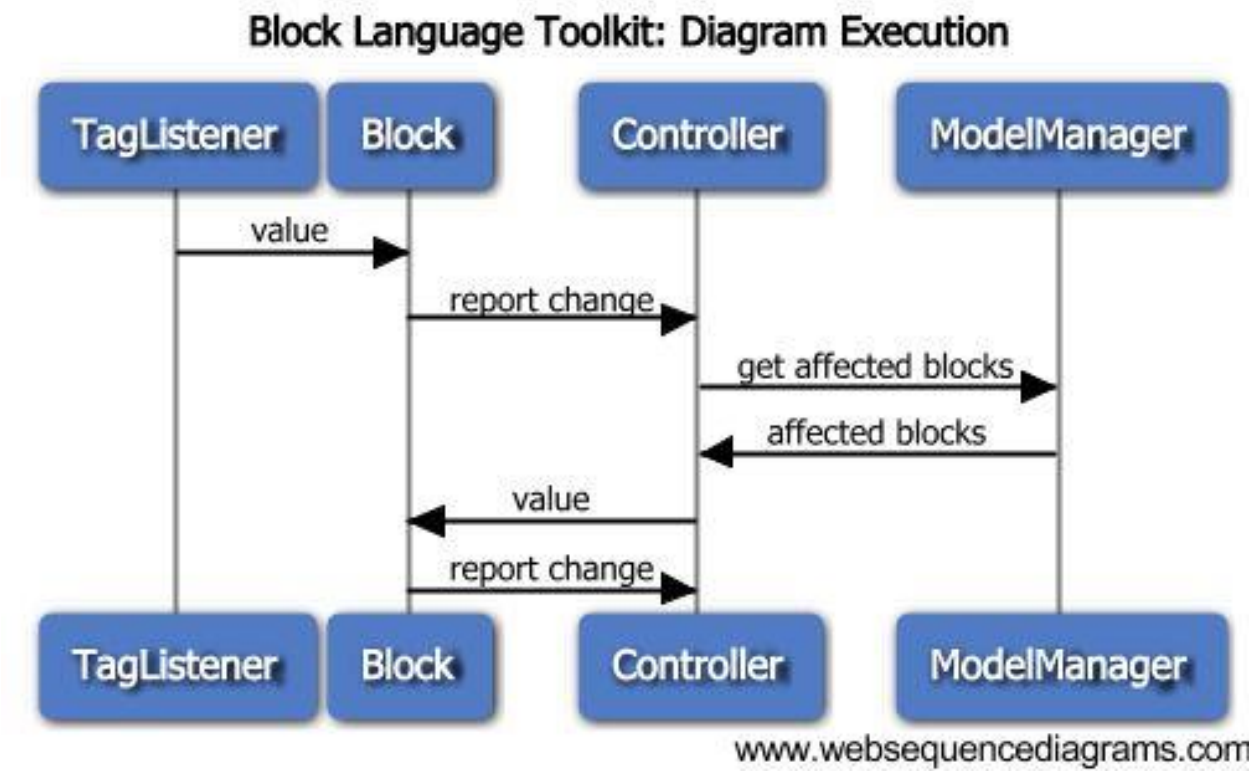


Figure 3 – Sequence Diagram

The diagram above depicts the sequence of operations when the tag listener detects a change that is bound to a block property. The activities all take place within the Gateway scope. The primary actors are:

- TagListener – The TagListener is configured to listen for changes to all tags that are bound to properties of blocks within a diagram. On detection of a change, the appropriate block is notified.
- Block – a process block. On detection of a new value on its input, the block processes the new value and, if appropriate, places a new value on its output.
- Controller – Block Execution Controller. The controller is the core dispatcher, accepting inputs and deciding what happens next. In this case, the controller asks the ModelResourceManager which blocks are connected to the output of the block that reported the change. It then notifies those blocks of the new value on their inputs.
- ModelManager = Model Resource Manager. This instance maintains collections of diagrams. The diagram instances are generated whenever a block-model project resource change (or addition) is detected. On request from the controller. The manager asks a diagram for a list of blocks downstream from a given block.

It should be noted that, in the sequence above, the block that receives the initial value update from the tag listener, is probably not to be the same block that receives the value change from the controller.

2.7 Designer

The *Designer* contains all code for creating and modifying diagrams.

2.7.1 Navigation Tree

Whenever the *Symbolic Ai* module is loaded into the Gateway, a “Symbolic Ai” node appears in the project resource area. This is the root of a tree structure that supplies access to the components (Applications, Families, Problems, Folders) of a *Symbolic Ai* application, or set of applications,

When any component of the navigation tree is selected, the *Symbolic Ai* workspace is displayed. This workspace is distinct and separate from the workspace that presents window components.

The state of any diagram and whether or not it is open is reflected by the icon in the tree. Additionally, any component that has a final diagnosis that is `TRUE` anywhere on it or among its offspring is annotated with a "bell" badge.

2.7.2 Saving

Saving refers to synchronization of the diagram view in the designer with the “live” version of that diagram and its blocks in the gateway scope. As shown in the diagram below, there are 3 components, each representing the diagram, which must be kept in synch.

- Diagram View – the Designer scope picture of the diagram
- Diagram Resource – the resource that is saved and restored with the project
- Diagram – “live” executing diagram, Gateway scope

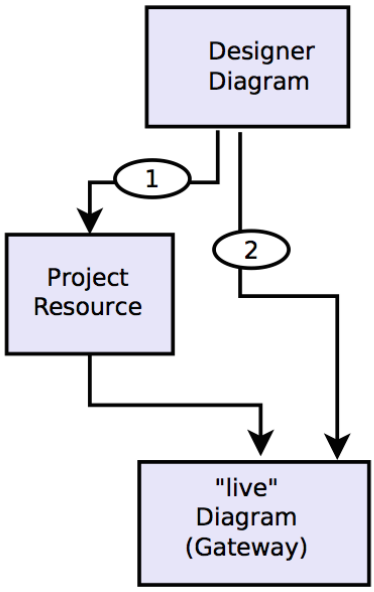


Figure 4 – Save Paths

The normal save path is the route indicated as (1). A “save” action in the Designer converts the view into a project resource, saving it in the project file. The same action generates an event in the gateway prompting the model manager to update the currently executing diagram object.

The alternate update route (2) is used when the user makes non-structural changes to the diagram or its blocks, such as property value changes. These are transmitted directly to the executing copy of the diagram, but leave the project resource out-of-synch.

The need to save is indicated with italicized names in the navigation tree. This indicates a mismatch between the designer view of the node and the project resource. In the case of a diagram view, a mustard background indicates that the designer view is out-of-synch with the actual diagram executing in the gateway.

There are several menu selections that “Save”, each with different behaviors.

- “Save and Publish”. This menu on the main menu bar saves all project resources, including those resources associated with diagnostic toolkit blocks. Unlike the saving of Vision resources, the toolkit resources are saved whether or not they are currently displayed on a workspace tab.
- “Save All”, is a menu selection on the top node of the Diagnostic Toolkit navigation tree. It results in a save of all applications and diagrams to the project. Note that this action has no effect on Vision or other resources not associated with the Block Language Toolkit, nor is any action taken if the designer and project resource are in synch. The menu selection is always enabled. Open diagrams are saved as they currently appear in the editor.
- “Save”, on an Application, saves of all diagrams and blocks associated with that application. As with the previous, only resources associated with the Block Language Toolkit are saved. This selection is enabled only if one or more resource in the Application tree has a pending change.
- “Save”, on a Diagram updates the project with properties of that Diagram and of its all blocks. This selection is disabled until the parent application has been saved for the first time. The reason for this behavior is that, until the parent Application is saved, the controller has no knowledge of that diagram. The menu selection is also disabled if there has been no structural change to the diagram or any of its blocks.
- “Save”, on a Block updates properties of that Block directly to the controller. This selection is disabled if there has been no change to the block’s properties. It is also disabled if the diagram is not known to the engine (it has never been saved).
- On closing a dialog tab the user is asked whether or not the diagram should be saved as shown or reverted to its state before the tab was opened..

- Editing block properties in the generic block editor has an immediate effect on the "live" block, but is not permanently saved with the project until the block, parent diagram or application is saved. * is this true? cjl
- When a node (application, folder, family or diagram) is added or deleted, the change is propagated to the gateway immediately and also updated in the project resource.
- A newly imported dialog is set to "disabled" to allow a review of its contents before going "live". A newly imported application is immediately "live".

2.7.3 Visual Synchronization

In order to create the illusion that the user is viewing the "real" blocks in the Designer, the Designer view must be carefully synchronized with the Gateway, both when initially displayed and in real time.

The mechanism to accomplish this coordination is as follows:

1) In the designer there is a push notification listener and cache. It detects changes as a diagram executes in the gateway. The listener is notified whenever a value is placed on the input of a block. In the designer this is equivalent to a connection receiving a value.
2) When a diagram is open in the designer, each of its connections subscribes to the push listener/cache for relevant changes.
3) When a diagram is opened in the designer, each of its connections queries the push listener/cache for current value.
4) When the designer is opened, the push listener/cache queries every diagram in the gateway for the current state of each of its blocks.
5) Each block in the gateway, when queried, responds with the last-known value on each of its outputs.
    a) Some blocks have a "value" property bound to the ENGINE that persists when the block is saved (e.g. an input). These blocks are able to respond with state even after a gateway restart.
    b) Some blocks (like observation blocks) retain state as an internal (transient) variable. After a gateway reset, these blocks have no current state (until a new value arrives)
    c) Some blocks (like junction blocks, test points) are simply pass thru and don't retain any state. After a designer restart, these blocks cannot report connection state to the drawing.

2.8 Client

Several Client views are provided for monitoring the state of logic blocks and other results.

3. GATEWAY

The *Gateway* “runs” diagrams defined in the *Designer* scope. It is the keeper of the “model”, which is a description of the blocks in the diagrams, their attributes and states, and the connections between them.

While it may be tempting to think of the Gateway as a “running engine”, in fact, the Gateway code merely listens to asynchronous events and responds accordingly. This is shown in the sequence diagram, Figure 3.

There are two live elements of the engine, a watchdog thread used for block input synchronization, and a tag subscription thread. The engine itself runs a bounded buffer that collects input and processes the resultant output.

3.1 Gateway Functions

The subsections below describe the major controller classes in the Gateway scope.

3.1.1 Dispatcher

The *GatewayRpcDispatcher* registers on startup as the receiver of RPC requests from client or designer components.

3.1.2 Resource Changes

The *ModelResourceManager* is a project change listener. It detects updates to project resources that hold diagram model definitions. On resource change, it deserializes the model and informs the engine of the changes. Resource changes are propagated by Ignition whenever the user selects “Save” on an Application, or “Save All” on the browser root node. In the case of “Save All”, a synchronization takes place to guarantee that there are no missing or extraneous resources in the Gateway due to some abnormal condition.

3.1.3 Block Execution

The *BlockExecutionController* follows the Singleton design pattern. It is the “engine”. Being a Singleton provides a well-known address for the object from anywhere in the Gateway. The engine is called when a block completes evaluation. Its function is to determine the block or blocks that are next to execute. The selected blocks are provided with the new output value, their input, then the *acceptValue()* method is invoked.

3.1.4 Tag Changes

The Gateway *TagListener* subscribes to tags that are identified as block inputs. When the tags change, the handler informs the appropriate block instance of a property change.

3.1.5 Block State

All blocks have a state, a truth-value. A truth-value has possible values of `UNSET`, `UNKNOWN`, `TRUE`, or `FALSE`. In practice, the state is meaningful only for those blocks that conclude an output truth-value. For other blocks the value remains `UNSET`. Note: The state `UNKNOWN` often indicates the presence of bad quality data.

Block state is accessible through the scripting interface and is visible in the user-interface through the “View Internals” menu selection on an individual block.

4. DESIGNER

The *Designer* is the only scope where changes to a diagram are supported.

4.1 NavTree

The Designer's navigation tree contains a "Diagnostic Toolkit" node. Use this root node to create new applications and under them, new families and diagrams.

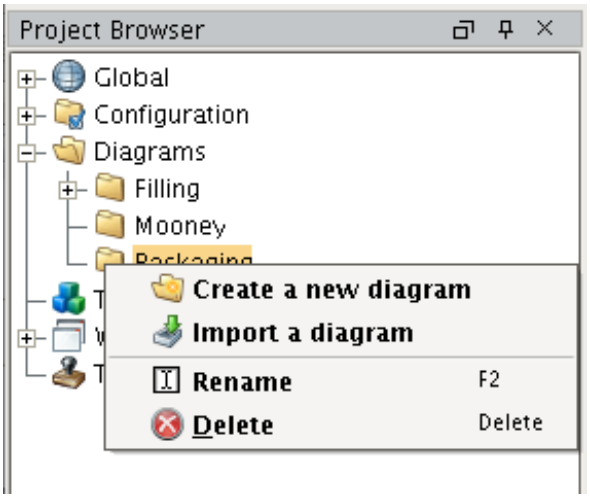


Figure 5 – Navigation Tree

The sections below summarize the available menu options for each node type.

4.1.1 Root Node

The root node is a single top-level entry in the project browser.

- *Create a new application* – create a folder node that will contain a collection of families or diagram nodes.
- *Start engine/Shutdown* – only one of these options is enabled at a time. Starting the engine enables tag subscriptions for bound properties, starts the watchdog timer thread and establishes the bound buffer that accepts block output actions. Shutting down unsubscribes to tags, terminates the watchdog thread and shuts down the processing buffer within the engine.
- *Clear controller*. This selection is present purely for debugging. It deletes any diagrams in the controller. An application "Save" can then be used to add a single application to the execution engine. This expedient makes debugging easier as it removes confusion

due to other live applications that may even belong to different projects. This action has no permanent effect – a Gateway restart will restore the prior state.

- *Debug to log* – write a description of current project resources to the Ignition designer log. Write a similar description of resources known to the Gateway. This comparison is useful only during development.

4.1.2 Application Nodes

Children of the root node are “Applications”.

- *Create a new family* – create a family node. A family is a container for problem nodes. Multiple problem nodes may be created under the same application.
- *Create Folder* – create a folder that can contain families. Useful for organizing large sets of diagrams.
- *Store Aux Data in Database* – covered above
- *Restore Aux Data from Database* – covered above
- *Copy* – copy the selected family to the clipboard
- *Paste* – paste a copy of the current family or folder in the clipboard
- *Configure* – bring up a custom edit dialog to change isolation mode, set outputs, etc.
- *Export Application Hierarchy* – Not sure – look into this - cjl
- *Save Application* – push changes to this application to the gateway.
- *Rename* – change the name of the application.
- *Delete* – remove the application node and all diagrams below it.

4.1.3 Family Nodes

Family nodes appear between “Application” and “Problem” nodes. There may be an arbitrary number of intervening folder nodes on either side of a family.

- *Create Diagram* – create a new problem node and accompanying diagram. A workspace is a container for blocks that will make an executable diagram. Multiple diagrams may be created under the same application.
- *Import Diagram* – display a file browser that allows entry of a diagram name. On selection of a file, attempt to marshal it and create a new diagram.
- *Create Folder* – create a folder that can contain diagrams or more levels of folders. Useful for organizing large sets of diagrams.
- *Copy* – copy the current active diagram to the clipboard. This differs from an import in that the states of the blocks in the cloned diagram are identical to those in the original.
- *Paste* – paste a copy of the selected diagram family or folder in the clipboard
- *Configure* – bring up a custom edit dialog to change isolation mode, set outputs, etc.
- *Save Family* – Push selected family data to the gateway
- *Rename* – change the name of the application.
- *Delete* – remove the application node and all diagrams below it.

4.1.4 Folder Nodes

Folder nodes are simple containers.

- *Create a new diagram* – create a new problem node and accompanying diagram. A workspace is a container for blocks that will make an executable diagram. Multiple diagrams may be created under the same application.
- *Import a diagram* – display a file browser that allows entry of a diagram name. On selection of a file, attempt to marshal it and create a new diagram.
- Clone diagram – clone the current active diagram. This differs from an import in that the states of the blocks in the cloned diagram are identical to those in the original.
- *Rename* – change the name of the application.

- *Delete* – remove the application node and all diagrams below it.

4.1.5 Problem Nodes

Each Problem node contains a diagram that schematically defines the problem analysis.

- double-click – opens the diagram workspace.
- *Export diagram* – display a file selection dialog. On selection of a file, serialize the diagram and write it to the specified file path.
- *Set State* – change the selected diagram to active, disable, or isolated. See 4.5.1
- *Save Diagram* – push the current diagram to the gateway
- *Copy* – copy the current diagram to the clipboard
- *Rename* – change the name of the diagram.
- *Delete* – remove the diagram node and associated workspace.
- *Debug to Log* – dump the state of the system to the gateway log. Useful to compare the state of blocks in the gateway vs. client/designer
- *Reset Diagram – See 4.5.3 Reset Action*

4.2 Menu

The Designer menu has been enhanced with an additional function under the View menu. When any node in the “Diagrams” tree is selected, the “Reset Panels” sub-menu will reset the visible workspace for the Block Language Toolkit. This includes hiding Vision windows, showing a tabbed pane center area that holds diagrams and also showing the block palette.

4.3 Palette

The palette is a Designer (Wicket) panel made viewable by a double-click action anywhere in the toolkit Nav tree or by the menu selection described above. The palette is a tabbed pane. The tab designations are determined by whatever is specified in the block prototype definitions.

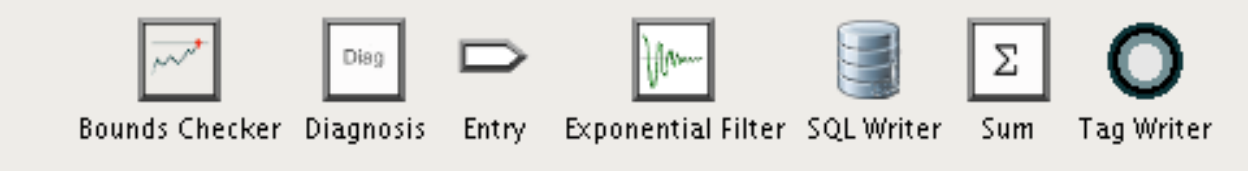


Figure 6 – Palette

4.4 Icons

All icons used in the toolkit reside in the Gateway (they are not project-specific). They may be accessed via the Ignition Designer “Icon Management” tool and, through it, be freely imported from and exported to files. This provides for easy user-modification.

The icon path is of the form: Block/icons/*nn*/name_*nn*.png, where nn is the icon dimension in pixels. In general icons that look good at a 32x32 pixel resolution are appropriate for the application. Icons of different pixel dimensions will be translated as needed.

4.5 Diagrams

The diagram window is tabbed pane located in the center of the Designer workspace area.

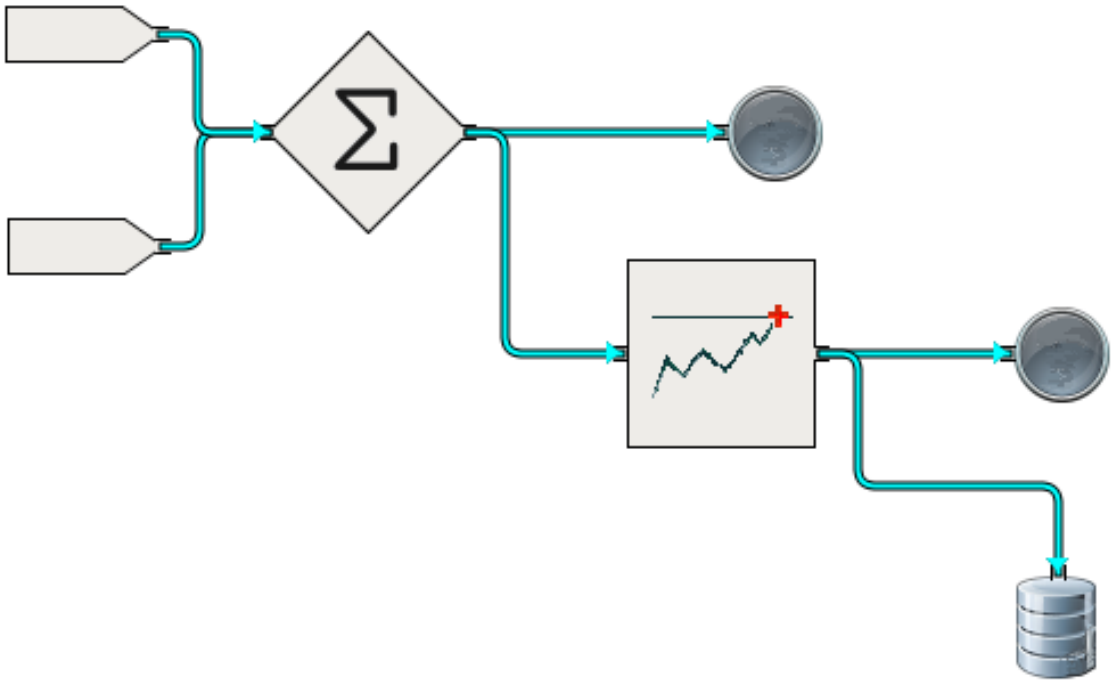


Figure 7 – Sample Diagram

4.5.1 Diagram State

The underlying block execution engine is, by default, “running”. In this condition, tag subscriptions are active and inter-block traffic is delivered to connected blocks.

The engine is started when the *Symbolic Ai* module is loaded and shutdown when the module is unloaded (usually during Ignition startup/shutdown). The engine running state may be also controlled via a designer menu selection. This action is typically used only during development.

Individual diagrams exist in one of three states:

1. Active. This is the normal state. The blocks in the diagram respond to tag value updates that modify bound properties. They respond to incoming data at their inputs, execute accordingly and post results on their outputs.

When gateway and designer versions of that diagram are “in-sync”, that is there have been no changes in the designer version since the last “save” action, then block state changes are transmitted to the Designer scope via the Ignition “push notification” mechanism. These are used to animate the diagram view.

2. *Disabled.* The user may explicitly set the state of a diagram to “disabled” via a menu selection in the Navigation tree. When a diagram is disabled, block properties bound to tags cease to be updated and no data/messages are delivered to blocks within that diagram. When the diagram is moved out of the” disabled” state, the diagram is reset. The disabled state in no way inhibits the ability of a user to edit the diagram.
3. Isolated. When in an “isolated” state, blocks in the diagram interact with a completely separate database and set of tags from production. In this way a diagram in this state is prevented from making “live” updates. The isolation database connection and tag provider are configured in the Designer *View* menu. When the Block Language Toolkit module is loaded this menu will contain a custom entry for “External Interface Configuration”. This allows definition of production and isolation database connections and tag providers. Once set, these values persist in the Ignition internal database.

4.5.2 Dirty

A diagram becomes “dirty” when an edit action changes block and/or connection configuration. In a “dirty” state, the version of the diagram executing in the Gateway does not correspond to the version in the Designer. In this state, notifications coming from the Gateway are ignored. Additionally, the diagram background is muddied to indicate this state to the user.

The “dirty” condition is cleared when the user “saves” the diagram either individually, via a parent application or via the main menu (which saves all project resources/diagrams). New

diagrams and newly imported diagrams are treated as “dirty” because, until a “save” action, there is no corresponding entity in the Gateway to actually execute.

A workspace tab that shows a “dirty” diagram cannot be closed without the user agreeing to save or discard the changes. In this instance, “save” means that the diagram changes are recorded into the project resource.

4.5.3 Reset Action

A “reset” action on a diagram results in the following sequence of actions:

1. Disable: The diagram is disabled, thereby canceling all tag subscriptions related properties of blocks within the diagram.
2. Block reset: Each block
   within the diagram is programmatically reset. Each block is expected to appropriately set its internal state. This usually results in setting its state to `UNSET` and clearing any internal data caches.
3. Connection reset: The connections downstream of each block are colored to indicate their “empty” status. No values are propagated.
4. Enable: The diagram is finally re-enabled (returning to its previous active or isolated state). Tag subscriptions are restored and each subscribing property receives the current value. Input blocks propagate the current tag value on their outputs.

Note: If the diagram is currently in a “disabled” state, a reset action has no effect.

4.5.4 Save

Saving changes is complicated. The philosophy is that a structural change, such as adding a block or deleting a block will make a diagram “dirty” and the diagram will remain “dirty” until the project is saved. Non-structural changes, such as setting a property value like the limit of a high limit block, is automatically saved immediately. When there is a structural change, making the diagram “dirty”, followed by a non-structural change; the automatic save will save both the structural and non-structural change and the diagram will return to a clean state.

Note from Pete: I think that the perception of saving non-structural changes immediately was that this was some sort of quick, lightweight save. In reality it appears to be a normal save, just performed automatically.

4.6 Blocks

This section describes characteristics of blocks within the block language.

4.6.1 Block Behavior

With few exceptions, all blocks respond to the following action commands:

- propagate – force propagation of the latest values on all outputs
- force – place a value on the selected output. Once propagated, the block is placed into a locked state.
- lock/unlock – a *locked* block will not propagate a value on its output(s)
- reset – initiate a *reset* action on the block. Reset behavior is dependent on the block type.

These actions can be triggered via a user menu associated with the block or via a signal connected directly to the block. The signal stub is enabled/disabled interactively by the operator. By default it is not displayed.

4.6.2 Block Definitions

This section describes important implementation details of specific blocks.

4.6.2.1 Input/Output and Source/Sink Blocks

There are four blocks that are the main way to bring data into a diagram and to get data out of a diagram. It is important to understand the similarities and differences between the blocks which are highlighted in the following table. All four of the blocks are configured with a tagpath.

Block	Direction	Tag Type	Reset Propagates	Follow Connections
Input	In	OPC, Memory, Expression or Query	No	No
Output	Out	OPC or Memory	No	No
Source	In	Memory tag in connection folder	Yes	Yes
Sink	Out	Memory tag in connection folder	Yes	Yes

The Input and Source blocks both use a tag listener to get the value of a tag and propagate it to its output connection whenever the tag receives a value.

Input / Output blocks are intended to bring data into the Symbolic Ai environment from an external system via an OPC tag or from a derived value via an expression tag or from some other facility in Ignition via a memory tag.

Source / Sink blocks are intended to provide a connection between diagrams. It is a common practice for a diagram to produce a value as a combination of blocks and then that value is

pushed to multiple other diagrams. Using memory tags to facilitate the communication between the Sink and the Source provides a convenient debugging capability and still utilizes the well-known tag listener technique. The normal workflow is to create a sink and then create a source and configure it to reference a sink.

4.6.2.1.1 CREATION SHORTCUT

There are two ways to create an Input / Output or Source / Sink block. They can be dragged from the palette onto the designer canvas or tag can be dragged from the tag browser onto the designer canvas. If the tag is from the Symbolic Ai/Connection folder, then a source / sink will be created any other tag will create an input or output. If the tag is dropped on the left side of the diagram then an input or source will be created, if dropped on the right ride then an output or sink will be created.

4.6.2.1.2 IMPORTANT FEATURES

These blocks have the following important features:

- A blocks connection type is determined by the type of the tag that it is bound to. This is nice automatic capability to make it faster to create and configure a diagram.
- The blocks connection type is “locked” in once it has been connected to another block.
- A block can be reconfigured by dragging a tag onto it. If the block has “locked in” its data type, then the data type of the new tag must be the same as the old tag.
- Blocks can be created by dragging them from the palette or by dragging a tag from the tag browser.
- Creating a block by dragging a tag from the tag browser has the following features:
    - Dragging to the left side of the diagram creates an input.
    - Dragging to the right side of the diagram creates an output.
    - When dragging a tag onto a block, or dragging a tag to create a block, will name the block with the tag name rather than some arbitrary unique name. The name should still be checked for uniqueness on the diagram and be morphed by adding a numeric suffix if necessary.
    - When dragging a tag onto the canvas, the type of the connection is determined by the type of the tag. A string, float, or integer will create an input block with a Data connection, a Boolean tag will create an input block with a truth value connection.
- Source / Sink Blocks have the following feature. (I think that the normal workflow is to create the sink, name the sink, then create a source, and select the name of the sink. If you want to create the source first you can, but you won’t be able to configure it until the sink is created.):
    - Dragging a memory tag from the Symbolic Ai/Connections folder will create a Source / Sink subject to the same left / right rules.

- Dragging a tag that is NOT in the Symbolic Ai/Connections folder onto a Source / Sink will be rejected.
- Multiple Sinks cannot reference the same tag
- Multiple Sources can reference the same tag
- Source / Sink blocks can only reference memory tags in the Symbolic Ai/Connections folder.
- Dragging a sink from the palette should uniquely name the sink and create a tag. If the user names the sink it will rename the tag.
- Dragging a source from the palette will leave the source unnamed. Configuring the Source should be as simple as selecting the name from a list of existing sinks.
- Neither a source or sink exposes the tag path or a tree for selecting a tag.
- Deleting a sink should delete the tag – extra credit if you check for references and prevent a referenced sink from being deleted
- Renaming a sink should rename the tag and referenced sources.
- Manually deleting connection tags will leave any referenced source and sinks in an unusable state.
- Sources and sinks must support reset propagation, this only applies when the path is a truth-value. (Reset starts with a Final Diagnosis and goes upstream to any observations and resets them. The observations will set their output to UNKNOWN which must then propagate forward.) In order to propagate an UNKNOWN from a sink to a source, truth value connections must use string tags. This provides some ambiguity when determining the type of connection for a source because a string tag could be used for a data connection or a truth-value connection. The strategy should be to set the connection type of the source to the same as the connection type of the sink.

4.6.3 Bad Data Handling

The information transmitted between blocks is encapsulated in *QualifiedValue* objects. These contain a value, a quality and a timestamp. In general, when a data value of bad quality is received, the resulting output will also be marked as having bad quality. In addition, the result of any calculation involving the bad data will return a value of not-a-number or `UNKNOWN` as appropriate to its type.

Exceptions to this pattern occur when it is possible to deduce an answer without involving the spurious input. For example, an “And” block can conclude a `FALSE` with one `FALSE` input of good quality, independent of the value or quality of other inputs.

4.6.4 Block Locking

Any block may be placed in a *locked* state. When *locked*, no further results are transmitted on its output, however all input processing occurs as normal. When the block becomes *unlocked*,

output is again enabled. The block will output a value during its next normal evaluation. Depending on the block this may be as a result of a timer expiring or receipt of input.

The locked state is activated via a menu choice that is available on all blocks. A locked block is annotated with a lock badge in the designer. Locked state is always cleared during a save.

Even when locked, the block will respond to a “forcedPost” request,. This causes the block to place a specified value on its output. This allows a tester/developer to isolate portions of a diagram during development/debugging.

4.6.5 Block Reset

All blocks support a *reset()* command. The intent of a *reset*() is to clear the block state and any internal data structures, thereby forcing calculations to begin from scratch. A reset will place the block in an `UNSET` state. This state is not propagated downstream. However, the downstream connection will be painted in such a way as to mark the reset action.

The `UNSET` state guarantees that the receipt of the next value will cause the block to evaluate and propagate a value, if appropriate. Blocks, such as `Compare`, `And` or `Or`, (these accept multiple inputs) retain the latest values received on each input. This allows a computation to be made on the very next arrival, no matter which input.

On a diagram reset, *Input* blocks propagate their values (the value of the bound tag). Individual blocks in the diagram are reset first before any propagation from input blocks.

4.6.6 Timestamp Handling

In general, if a data value is passed through a block, even if there has been an operation on it (e.g. exponential), the timestamp of the result will be the same as the timestamp of the input. However, if the value has been combined in any way with other inputs, then the timestamp of the result is set to the time at which the output was propagated.

4.6.7 Connection Type Change

Several of the block classes operate on any of the possible data types. In order to properly configure connections for each individual situation, a control-click menu option is supplied to “Change Connection Types”. This option alters all of the block’s stubs to the chosen type. The blocks for which this feature is available include:

- Input
- Junction
- Output
- Parameter
- Readout

- Sink
- Source

Once a diagram has been saved, this option is no longer available.

4.6.8 Type Coercion

Blocks are aware of the types of data they process. The type of the input connector guarantees the data type of an input. (In the case of Input blocks, the incoming tag value is coerced to the expected type). Blocks also guarantee the correct data type on output.

Where coercion is necessary, the following steps are taken:

1. The value is converted to a string. This provides the conversions with a common starting point.
2. For strings no conversion is necessary. However, a null will generate an error.
3. For real numbers, the conversion is `Double.parseDouble(`*`value`*`)`.
4. For integers, the conversion is `Integer.parseInt(`*`value`*`)`.
5. For booleans, any non-zero numeric value or the string "`TRUE`" (case-insensitive) is taken as true. All others are false.

If the conversion step fails, then the reading is given a BAD quality and propagated.

4.6.9 Block Programming Interface

All executable blocks, both those implemented in Python and those implemented in Java, must support the following interface:

```
/**
 * Notify the block that a new value has appeared on one of its
 * input anchors. The notification contains the upstream source
 * block, the port and value.
 * @param vcn
 */
public void acceptValue(IncomingNotification vcn);

/**
 * Notify the block that it is the recipient of a signal from
 * "the ether". This signal is not associated with a connection.
 * This method is meaningful only for blocks that are "receivers".
 * @param sn
 */
public void acceptValue(SignalNotification sn);

/**
 * If true, the "engine" will delay calling the start() method
 * of this block until all other blocks that do not indicate
 * a delay have been started.
 * @return true if this block should be ordered at the end
 *         of the startup process.
 */
public boolean delayBlockStart();
```

```
    /**
     * In the case where the block has specified a coalescing time,
     * this method will be called by the engine after receipt of input
     * once the coalescing "quiet" time has passed without further input.
     */
    public void evaluate();

    /**
     * Place a value on a named output port of a block.
     * This action does not change the internal state of the block.
     * It's intended use is to debug a diagram.
     * @param port the port on which to insert the specified value
     * @param value a new value to be propagated along an
     *        output connection. The string value will be coerced
     *        into a data type appropriate to the connection.
     */
    public void forcePost(String port, String value);

    /**
     * @return a list of anchor prototypes for the block.
     */
    public List<AnchorPrototype> getAnchors();

    /**
     * @return the universally unique Id of the block.
     */
    public UUID getBlockId();

    /**
     * @return information necessary to populate the block
     *          palette and subsequently paint a new block
     *          dropped on the workspace.
     */
    public PalettePrototype getBlockPrototype();

    /**
     * @return the fully qualified path name of this block.
     */
    public String getClassName();

    /**
     * @return information related to the workings of the block.
     *        The information returned varies depending on the
     *        block. At the very least the data contains the
     *        block UUID and class. The data is read-only.
     */
    public SerializableBlockStateDescriptor getInternalStatus();

    /**
     * @return the block's label
     */
    public String getName();

    /**
     * @return the Id of the block's diagram (parent).
     */
    public UUID getParentId();

    /**
     * @return the id of the project under which this block was created.
     */
    public long getProjectId() ;
```

```
	/**
	 * @return all properties of the block. The array may be used
	 *                  to updated properties directly.
	 */
	public BlockProperty[] getProperties();

	/**
	 * @return a particular property by name.
	 */
	public BlockProperty getProperty(String name);

	/**
	 * @return a list of names of properties known to this class.
	 */
	public Set<String> getPropertyNames() ;

	/**
	 * @return the current state of the block
	 */
	public TruthValue getState();

	/**
	 * @return a string describing the status of the block. This
	 *            string is used for the dynamic block display.
	 */
	public String getStatusText();

	/**
	 * @return true if this block is locked for debugging purposes.
	 */
	public boolean isLocked();

	/**
	 * @return true if this block is a candidate for signal messages.
	 */
	public boolean isReceiver();

	/**
	 * @return true if this block publishes signal messages.
	 */
	public boolean isTransmitter();

	/**
	 * Send status update notifications for any properties
	 * or output connections known to the designer.
	 *
	 * In practice, the block properties are all updated
	 * when a diagram is opened. It's the connection
	 * notification for animation that is most necessary.
	 */
	public void notifyOfStatus();

	//====================== PropertyChangeListener ======================
	/**
	 * This is a stricter implementation that enforces QualifiedValue data.
	 */
	public void propertyChange(BlockPropertyChangeEvent event);

	/**
	 * Reset the internal state of the block.
	 */
	public void reset();
```

```
    /**
     * Set the anchor descriptors.
     * @param prototypes
     */
    public void setAnchors(List<AnchorPrototype> prototypes);

    /**
     * Set or clear the locked state of a block.
     * @param flag True to lock the block.
     */
    public void setLocked(boolean flag);

    /**
     * @param name the name of the block. The name
     *        is guaranteed to be unique within a
     *        diagram.
     */
    public void setName(String name);

    /**
     * @param id is the project to which this block belongs.
     */
    public void setProjectId(long id);

    /**
     * Accept a new value for a block property. It is up to the
     * block to determine whether or not this triggers block
     * evaluation.
     * @param name of the property to update
     * @param value new value of the property
     */
    public void setProperty(String name,Object value);

    /**
     * Set the current state of the block.
     * @param state
     */
    public void setState(TruthValue state);

    /**
     * @param text the current status of the block
     */
    public void setStatusText(String text);

    /**
     * Specify the timer to be used for all block-
     * internal timings.
     *
     * @param timer
     */
    public void setTimer(WatchdogTimer timer);

    /**
     * Start any active monitoring or processing within the block.
     */
    public void start();

    /**
     * Terminate any active operations within the block.
     */
    public void stop();

    /**
```

```
	 * Convert the block into a portable, serializable description.
	 * The descriptor holds common attributes of the block.
	 * @return the descriptor
	 */
	public SerializableBlockStateDescriptor toDescriptor();

	/**
	 * @param tagpath
	 * @return true if any property of the block is bound to
	 *         the supplied tagpath. The comparison does not
	 *         consider the provider portion of the path.
	 */
	public boolean usesTag(String tagpath);

	/**
	 * Check the block configuration for missing or conflicting
	 * information.
	 * @return a validation summary. Null if everything checks out.
	 */
	public String validate();
```

4.6.10 Block Icons

Each block (both custom blocks and the blocks implemented in Java) needs to provide two icon images, preferably in PNG format. One for the palette and another for the designer workspace. The image for the palette should be 32x32 and loaded into Block/icons/palette. The image for the designer workspace should be 100x100 and loaded into Block/icons/embedded. Image files are loaded in the designers Image Management system, accessible through the Tools menu (Tools/Image Management).

Note: ILS uses the “Paint S” package on a Macintosh to create and edit images. After the image editing is complete the paint file is saved as a PNG and loaded into Ignition. Both the paint and PNG files are checked into subversion.

4.6.11 Creating Custom Blocks

It is possible for users to create custom blocks entirely in Python without requiring a rebuild of the Symbolic Ai module. The Python for user created custom blocks are located in *user-lib\pylib\ils\user\block* under the Ignition install directory. ILS also provides several blocks that are implemented in Python. These are located in *user-lib\pylib\ils\block.* This will help differentiate between standard and custom blocks as well as preventing overwrite during system updates.

A gateway (and designer) restart is required after changes have been made to any block python files.

The arithmetic block (*ils.block.arithmetic.py*) is used as an example to help understand how to define a new block. The block definition is contained entirely in this single Python file. The definition uses object oriented Python with a single class definition.

The class definition statement and the __init__ method are shown below. The class will inherit from basicBlock.BasicBlock. This is seen in the class statement and in the *__init__()* method.

```
class Arithmetic(basicblock.BasicBlock):
    def __init__(self):
        basicblock.BasicBlock.__init__(self)
        self.initialize()
```

The initialize method defines the block specific properties which will appear in the property editor. It also defines the input and output connections. It is worth pointing out that the standard behavior for connections is that they will accept multiple connections. While this may make sense for some blocks, for a block that implements a math function of one argument it does not. The *'allowMultiple': False* clause can be added enforce a single connection.

```
# Set attributes custom to this class
def initialize(self):
    self.className = 'ils.block.arithmetic.Arithmetic'
    self.properties['Function'] = {'value':'','editable':'True','bindingType':'OPTION',
                                   'binding':'ABS,CEILING,COSINE,FLOOR,SINE,TANGENT,TO_RADIAN,ROUND'}
    self.inports = [{'name':'in','type':'data','allowMultiple':False}]
    self.outports= [{'name':'out','type':'data'}]
```

The getPrototype() method primarily specifies the appearance of the block on the palette and on the designer workspace. It references two image files. If the images do not exist then the block will not appear on the palette.

```
# Return a dictionary describing how to draw an icon
# in the palette and how to create a view from it.
def getPrototype(self):
    proto = {}
    proto['iconPath']= "Block/icons/palette/function.png"
    proto['label']   = "Arithmetic"
    proto['tooltip']         = "Execute a user-defined function on the input"
    proto['tabName']         = 'Arithmetic'
    proto['viewBackgroundColor'] = '0xF0F0F0'
    proto['viewIcon']       = "Block/icons/embedded/fx.png"
    proto['blockClass']      = self.getClassName()
    proto['blockStyle']      = 'square'
    proto['viewHeight']      = 70
    proto['viewWidth']       = 70
    proto['inports']         = self.getInputPorts()
    proto['outports']        = self.getOutputPorts()
    proto['receiveEnabled']  = 'false'
    proto['transmitEnabled'] = 'false'
    return proto
```

The *acceptValue()* method is called whenever a value is received on its input. This method implements the functionality of the block.

```
# Called when a value has arrived on one of our input ports
# Compute the result, then propagate on the output.
def acceptValue(self,port,invalue,quality,time):
    function = self.properties.get('Function',{}).get("value","")
    if len(invalue) == 0:
        return

    dbl = Double(str(invalue))
    value = dbl.doubleValue()       # Default behavior is a pass-through
    #value = float(invalue)
    if len(function)>0:
        if function == 'ABS':
            absolute = Abs()
            value = absolute.value(value)
        elif function == 'CEILING':
            ceiling = Ceiling()
            value = ceiling.value(value)
        elif function == 'COSINE':
            cosine = Cosine()
            value = cosine.value(value)
        elif function == 'FLOOR':
            floor = Floor()
            value = floor.value(value)
        elif function == 'ROUND':
            value = Precision.round(value,0)    # Static function, this is OK
        elif function == 'SINE':
            sine = Sine()
            value = sine.value(value)
        elif function == 'TANGENT':
            tan = Tangent()
            value = tan.value(value)
        elif function == 'TO_RADIAN':
            value = 0.0174532925*value;

        log.tracef("Arithmetic.acceptValue: %s(%s) = %s",function,str(invalue),str(value))

        self.value = value
        self.quality=quality
        self.time = time
        self.state = "TRUE"
        self.postValue('out',str(value),'good',time)
```

The *propagate()* method implements the mechanics of the value that is passed to downstream blocks via the output connection.

```
# Propagate the most recent state of the block.
def propagate(self):
    if self.state <> "UNSET":
        self.postValue('out',str(self.state),self.quality,self.time)
```

The complete list of methods that are available can be seen in *ils/actions/block/basicblock.py*.

4.6.11.1 Init File Requirement

New block names need to be added to the __init.py__ file located in the same folder as the block python file. The new block will not appear in the palette if this entry is not added. Please note that the name and file name must match and be all lower case in order for it to be picked up by the system. The __init.py__ entry should be similar to this:

```
# Define the modules that are executable block classes
# This is required for "from ils.block import *" to give the desired answer
#
# **** ALL NAMES MUST BE ALL LOWER CASE OR IT WILL FAIL ****
#
__all__ = ["action","arithmetic","finaldiagnosis","sqcdiagnosis","subdiagnosis"]
```

4.6.12 Obsolete Blocks

The following blocks were not implemented for one reason or another. They are left in this document in the event that they are needed at some point in the future.

4.6.12.1 Control Counter

Tracks the number of times it receives a control signal and displays the total on the block in the UI. It passes the control signal on its output control path and the count on its output data path. On reset, the count is set to zero.

Attribute Display:

Count - The number of signals received

Connections:

- in – data connection
- out – passthru of the input
- count – value of count, incremented each execution

4.6.12.2 Data Pump

Emit a fixed (or bound) value on a configured interval. The output is stamped with the current time. The block is initially inactive waiting until the interval property receives a value. Additionally, the block will not cycle if the the interval is less than or equal to zero.

The value property may be updated at any time. An new value is propagated immediately. On reset, the block restarts its cycle (assuming that both interval and value are valid. If the value is bound to a tag, then the block essentially polls the value of the tag.

Properties:

Interval – interval at which a value is generated on the output ~ secs. The default value is Double.NaN. The block will not begin polling until a valid interval is configured.

LiveOnStart – If the interval and value are defined, and this parameter is TRUE, the block will begin emitting values when it is started. Otherwise the value will not be emitted until there is a change to either value or interval properties.

Value – the value to be emitted by the block. Datatype is `OBJECT`, meaning it is alterable based on the connection type.

Connections:

- out – output anchor. The connection type may be set after the block is placed on the workspace. The emitted value is coerced to a datatype compatible with the connection.

4.6.12.3 Encapsulation

An Encapsulation block represents an EncapsulatedDiagram in the daigram superior to it. An Encapsulation is the only block that has runtime-configurable anchor points. On creation of an Encapsulation an associated EncapsulationDiagram is created and linked into the Navigation tree. When anchor points are configured on the encapsulation, associated Entry and/or Exit Connections are created on the sub-diagram.

Connections: There are a variable number of connections, configured on the encapsulation at run-time.

4.6.12.4 Entry Connection

This is a connection post that is automatically generated when an Encapsulation block is created. It appears on the associated EncapsulatedDiagram associated with the encapsulation. There is one EntryConnection for each incoming connection to the encapsulation.

Properties:

Label – This label is read-only. It matches the name of the anchor point of the parent encapsulation associated with this block.

Connections:

- out –value derived from the parent encapsulation.

4.6.12.5 Exit Connection

This is a connection post that is automatically generated when an Encapsulation block is created. It appears on the associated EncapsulatedDiagram associated with the encapsulation. There is one EntryConnection for each outgoing connection on the encapsulation.

Properties:

Label – This label is read-only. It matches the name of the anchor point of the parent encapsiulation associated with this block.

Connections:

- in –value to be routed to the parent encapsulation.

4.6.12.6 Inference Inhibitor – DEPRECATED Use Inhibitor

The Inference Inhibit block enables and disables a truth-value path. You can use it to control entire flow diagrams or an important subsection of one.

When the status value of the top inference path (control) matches the value of the attribute Trigger, the block inhibits the bottom input inference value (in). When the block is inhibiting the inference path, it does not propagate any input.

When the status value of the top inference path (control) does not match Trigger, the block passes the bottom input inference value normally. On a change of the trigger input to a mismatch, the last value received is propagated.

If the InitialValue property has a value, the block passes that value on startup or reset; otherwise it passes nothing when initialized or reset. Also, if the block has an InitialValue, and the top inference path value matches the Trigger, the block passes InitialValue.

Properties:

InitialValue – truth-value, an optional value to be propagated on startup or block reset if the trigger input matches.

Trigger – truth-value, propagate the input if the trigger line matches this value.

Connections:

- in – truth-value, incoming value.
- control – truth-value, the control line.
- out – truth-value, output connection.

4.6.12.7 Inference Memory

The Inference Memory block remembers whether the top input (marked “S”) has ever been `TRUE since either the block has been started or reset path activated`. The top input, marked “S,” is the set path, and the bottom input, marked “R,” is the reset path. When the set path becomes `TRUE` and the reset path is already `FALSE`, the block’s output becomes `TRUE`. When the reset path becomes `TRUE`, the block’s value is “reset” to `FALSE`.

When the block evaluates its inputs, it goes through the steps in this list in the order shown to determine what value to pass. Inputs that have never received a value are treated as “UNKNOWN”.

1. If R is TRUE, it passes `FALSE`.
2. If R is `UNKNOWN`, it passes `UNKNOWN`.
3. If S is TRUE, it passes `UNKNOWN`.
4. If previous output is TRUE, it passes TRUE.
5. If S is `UNKNOWN`, it passes `UNKNOWN`.
6. On startup or on block reset, it passes `UNKNOWN`.

The Inference Memory block has no configurable properties.

Connections:

- set – truth-value, the set input.
- reset – truth-value, the reset line
- out – truth-value, the result

4.6.12.8 PID

Perform PID control on an incoming data stream.

Properties:

SetPoint – the target setting.

Interval – the time interval at which control calculations are made ~ milliseconds.

Kd – the derivative constant.

Ki – the integral constant.

Kp – the proportional constant .

Connections:

- in – raw data stream to be controlled.
- out – truthvalue, initialized to UNKNOWN. Value is emitted on a state change.
- recv – signal. If the command is “RESET”, then any accumulated data will be cleared, and the calculations will proceed from scratch.

4.6.12.9 SQL

Write data received on the input connection to a database. The SQL is a block property, assumed to be a prepared statement with a single argument. The input value of the block is that value. The database is the appropriate instance depending on the state (isolation or production) of the parent diagram.

Properties:

SQL – A SQL insert or update statement that can be compiled into a prepared statement taking exactly one argument.

Connections:

- in – data, truthvalue, signal or text connection.

4.7 Block Properties

Each block possesses a pre-defined list of properties. Properties are persistent attributes specified by a name, a binding type and a value. Additionally, a property value may be configured for display in the designer by using a right mouse click on a block and selecting Display Block Properties. This will bring up a dialog box where available properties can be selected for display. Property values do NOT have a quality attribute.

4.7.1 Data Types

Properties are pre-assigned a data type from one of the following options:

- BOOLEAN
- INTEGER
- HTML (gets an HTML editor)
- HYSTERESIS (TRUE, FALSE, ALWAYS, NEVER)
- LIMIT (HIGH, LOW, BOTH, NONE, CONSECUTIVE)
- LIST (multiple user-entered values)
- OBJECT (matches any type)
- OPTION (block-defined selection list)
- SCOPE (GLOBAL, APPLICATION, FAMILY, LOCAL)
- STRING
- TIME
- TRUTHVALUE (TRUE, FALSE, UNKNOWN, UNSET)

4.7.2 Binding

Block properties may be bound to tags or other elements in the application. The binding options are as follows:

- NONE
- ENGINE
- OPTION
- TAG_MONITOR
- TAG_READ
- TAG_READWRITE
- TAG_WRITE

Except for properties that are pre-defined as bound to the `ENGINE` or `OPTION`, the binding definition can be changed dynamically via the property editor.

The `ENGINE` designation is determined exclusively by the block and cannot be changed. That is, once a property is designated as `ENGINE` it cannot be set to a different binding type, nor

can a property that is not bound to ENGINE be converted to ENGINE. This behavior is enforced by the property editor.

In a similar way, the OPTION binding contains a fixed list of options known to the block and cannot be changed.

The TAG_MONITOR binding describes a property that subscribes to a tag and changes its value when the tag changes value. An Input block is the most straightforward example.

A property bound to a tag with TAG_READ actively retrieves the tag value on command.

TAG_READWRITE is used for a property that both writes to a tag on value change and refreshes its value when the tag changes. This is used, for example, in the Parameter block. The block is essentially a stand-in for the tag in the diagram.

TAG_WRITE refers to a property binding where its value is written to a tag each time it changes.

The value of any property can, optionally, be displayed in the Designer by using the Display Block Properties option from the block's right click popup menu. Properties that have an ENGINE binding are of particular interest as they reflect some internal state of the block and will update dynamically as the block executes.

4.7.3 Activity Log

Every block has the potential to record a list of activities in a fixed-size buffer. Initially this capability is disabled. It can be activated simply by editing the block's ActivityBufferSize property, setting the value to something greater than zero.

Contents of the log are available in the "*View Internal State*" popup menu on the block. Activities that are logged include:

- RESET – marks a reset action on the block
- STATE – records an internal block state change
- *port* – records data propagated on the named output port

Additional activites may be available on specific classes of blocks.

4.8 Property Editor

Whenever a block is selected, an editor for its properties appears in a Designer (Wicket) window. The editor displays read-only block attributes, and then a panel for each custom property. When edited, the block properties are immediately transmitted to the Gateway.

Note: Entries into a text field are not recorded until an ENTER is typed.

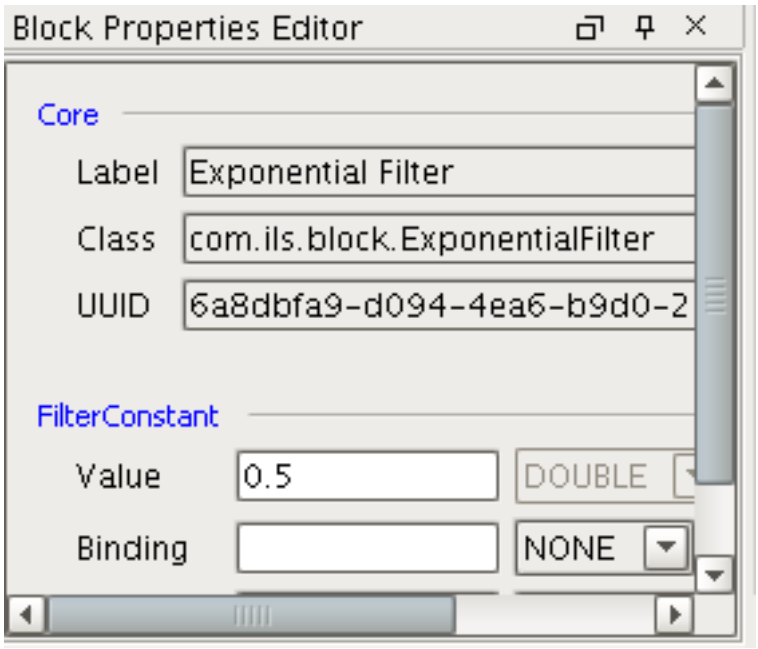
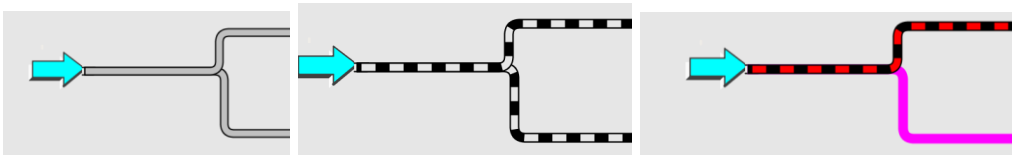


Figure 8 – Properties Editor

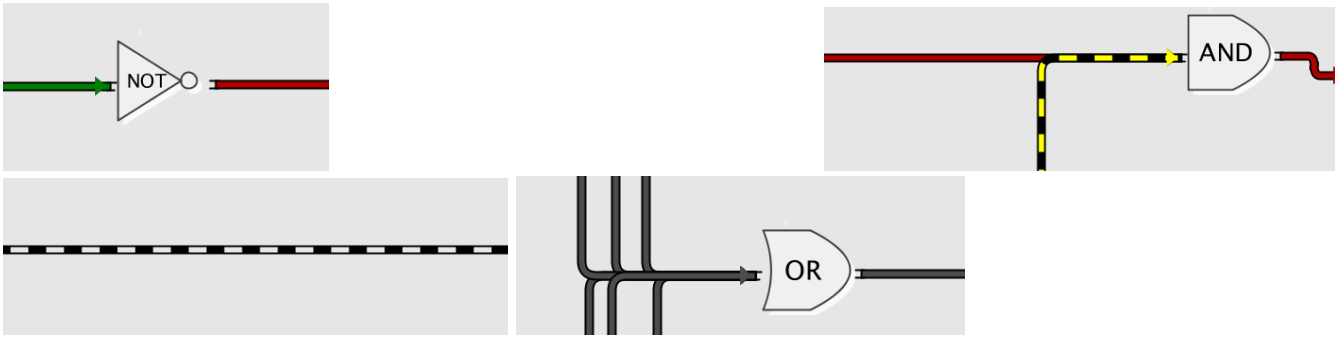
4.9 Connections

There are four types of connections:

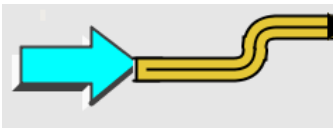
- Numerical (data)
- Truth-value (true,false,unknown,unset)
- Signal
- Text (diagnoses, recommendations)



DATA – normal, unset, not-a-number, selected (magenta)



TRUTH-VALUE – true (green), false (red), bad quality(yellow/black), unset (gray/black), unknown (gray)



TEXT

Figure 9 – Connection Coloring

4.10 Transmit/Receive

The ability to send and receive signals is a design-time configurable property of a block. Blocks enabled to receive broadcast signals are marked with a receiver “badge”. Blocks that send signals are similarly marked with a transmitter badge.

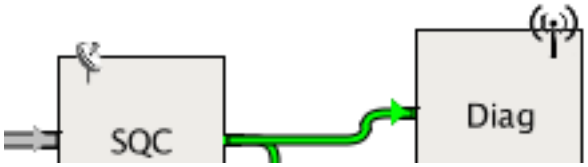


Figure 10 – Receive/transmit Badges

4.10.1 Signals

Signals are commands that can be used to control block state, e.g. “start” or “reset”. In addition to originating from blocks within a diagram, signals can be sent from the outside, e.g. from a Vision window. Signals have a property of “scope” that determines the collection of blocks to which they may be delivered.

There are four scoping levels:

- Local – the signal is delivered only blocks in the diagram from which it originated
- Family – the signal is delivered blocks in diagrams in the same family as the originator
- Application – the signal is delivered to blocks in diagrams in the same application as the originator
- Global – the signal is delivered to all blocks (with receiving enabled) in the current project

A signal has the following attributes:

- Command – a string value containing a well-known command

- Argument - a string argument or parameter name
- Payload – an optional payload, a string
- Pattern – a string used by a Receiver block to determine whether or not to process the signal. A '*' configured in the block matches all input. For other types of blocks, the pattern must match the block name.

4.10.2 Standard Signals

All blocks are expected to respond to the following signal commands as they arrive on the signal input stub that is present for every block.

- configure – use information in the signal to set a property value. Trigger any evaluations would normally occur.
- propagate – execute the propagate() method of the block. This forces emission of the last value on the outputs.
- lock – set the block to a locked state. It will not pass any values.
- reset – clear any internal storage, set the current state, if appropriate, to UNKNOWN.
- unlock – set the block to an unlocked state

4.11 Find/Replace

The standard Ignition find/replace capabilities have been extended to include the Block Language Toolkit. There are three elements of this feature: finding, replacing and locating.

The search may be restricted to any or all of the categories: application names, family names, diagram names, block names, block property names and block property values. In the case of bound properties that search is made on the binding rather than the current value. For example we return a tag path rather than the current value of that tag.

The current implementation does not allow editing via the find/replace mechanism. Any edits must be made through the normal Designer interfaces.

Double-clicking on the icon associated with a discovery will display the diagram containing that element. For this display to appear, the block language toolkit workspace must be already selected in the Designer.

4.12 Gateway Helper Functions

There are a set of utilities designed to be accessible from either Java or Python in the designer space. These methods are defined in the BLT_Common project in com.ils.blt.common.ToolkitRequestHandler.java and implemented in ApplicationRequestHandler.java. In the Python realm there is a definition file, ils.blt.diagram.py, to make them easily visible through Eclipse or another IDE

On the gateway side they are in system.ils.blt.diagram and in designer scope accessed through com.ils.blt.common.ApplicationRequestHandler.

NOTE: On the gateway side, these methods are accessed through GatewayScriptFunctions.java

```
This class exposes python-callable functions in the Gateway scope. These mirror the
functions available in Client/Designer scope.  All requests are delegated to the same
request handler as the GatewayRpcDispatcher.
```

5. PYTHON

The toolkit may be extended in several important ways via Python. The Python interfaces described in this section are intended as opportunities for the user to develop custom links with the toolkit core. The categories of extensions are as follows:

- Custom Blocks – In addition to the collection of blocks distributed with the toolkit, the user may develop blocks completely in Python, completely external to the official distribution. Blocks that written in this manner behave within the diagram in the same way as the blocks written in Java that are distributed as part of the BLT module.
- Scripting Interface – The toolkit exposes a collection of Python scripts that may be accessed from any Vision widget.
- Extension functions – The toolkit exposes extension points where the engine calls Python scripts (if they are defined) at specified points in the life cycle of an Application or Family object.

5.1 Custom Blocks

This section discusses the features in Symbolic Ai that allow blocks to be implemented in Python and also how to define a block in Python.

5.1.1 Custom Block Implementation

The utility package `ils.blt.util` contains all of the methods defined to interface into the Python classes. These are called from a block class in the engine that serves as a proxy in the Gateway “engine” code.

The Python interface includes:

```
def createBlockInstance(className,parent,uid,result):
def evaluate(block):
def getBlockProperties(block,properties):
def getNewBlockInstances():
def getNewBlockInstance(className):
def getBlockPrototypes(prototypes):
def setBlockProperty(block,prop):
def setValue(block,port,value,quality):
```

5.1.1.1 CreateBlockInstance

Return an instance of the specified Python class. The method accepts a single argument, the class of the block to create. The shared variable, ‘result’ is sent to the method as an empty dictionary. On return, it should contain an element ‘instance’ that contains the newly created block.

5.1.1.2 GetBlockProperties

Return the defined properties for a specified block. Typically these are configuration parameters as opposed to real-time state variables. The block is specified as the ‘block’ argument of the method. The shared variable, ‘properties’ is sent to the method as an empty list. On return, the list should contain dictionaries one for each block property . The dictionaries contain the following keys:

- binding - path to the .
- bindingType - type of binding for this property, if any. Options are: `NONE`, `ENGINE`, `TAG_READ`, `TAG_WRITE` and `TAG_MONITOR`. Default is `NONE`. Refer to the JavaDoc for *BindingType* for the most current list.
- editable - `TRUE` if the property can be edited, otherwise `FALSE`.
- maximum - a double value specifying the permissible maximum of the property. Default is `Double.MAX_VALUE`.
- minimum - a double value specifying the permissible minimum of the property. Default is `Double.MIN_VALUE`.
- name - name of the property. This must be unique among the properties for a class.
- quality - current quality of the value of the property.
- type - data type of the property. Options are: `STRING`, `DOUBLE`, `INTEGER`, and `OBJECT`. Default is `STRING`. Refer to the JavaDoc for *PropertyType* for the most current list.
- value - current value of the property

5.1.1.3 GetBlockPrototypes

Return the information necessary to create palette prototypes. The method accepts no arguments. The shared variable, ‘prototypes’ is sent to the method as an empty list. On return, the list should contain dictionaries one for each block class implemented in Python. The dictionaries contain the following keys:

- blockClass - class name of the block, e.g. “app.block.Custom.Custom”. The repeated class names are an artifact of storing each Python class in its own file.
- blockStyle - name of the graphical layout of the block in a view. Options are: `CLAMP, DIAMOND, ENTRY, ICON, ROUND,` and `SQUARE`. Refer to the JavaDoc for *BlockStyle* for the most current list.
- iconPath - this is the path to the icon used for the block in the palette.
- label - this is the label that appears under the block in the palette
- tabName - name of the palette tab on which this block appears. If the name does not match an existing tab, a new tab will be created.
- tooltip - text that appears when the mouse hovers over this block on the palette.
- viewBlockIcon - path to an icon that is to be used for the entire block.
- viewFontSize - set the size of the font for text embedded in the block view, if any. Default size is 24 points.
- viewHeight - overrides the default height of the block when drawn in a diagram. The default is a function of block style. ~ pixels.
- viewIcon - icon to appear inside a block when drawn in a diagram.
- viewLabel - text to appear inside a block when drawn in a diagram.
- viewWidth - overrides the default width of the block when drawn in a diagram. The default is a function of block style. ~ pixels.
- inports - a list of dictionaries describing the ports that are inputs, that terminate a connection.

- outports - a list of dictionaries describing the ports that are outputs, that originate a connection.

The keys for the dictionaries used to describe ports are:

- name - name of the port
- type - datatype of the connection connected to the port. Options are: `ANY`, `DATA`, `INFORMATION`, `SIGNAL` or `TRUTHVALUE`. Refer to the JavaDoc for *ConnectionType* for the most current list.

5.1.1.4 Report Results

Return block output values to the execution engine for dissemination to downstream blocks.

```
/**
 * Handle the block placing a new value on its output.
 *
 * @param parent identifier for the parent, a string version of a UUID
 * @param id block identifier a string version of the UUID
 * @param port the output port on which to insert the result
 * @param value the result of the block's computation
 * @param quality of the reported output
 */
void send(String parent,String id,String port,String value,String
quality);
```

5.1.2 Creating a Python Block

Several of the blocks that are distributed with Symbolic Ai have been written in Python. It is transparent to the user whether a block is implemented in Java or in Python. The python blocks that have been written by ILS Automation are distributed in the installer and placed into the `user-lib/pylib/ils/block` subdirectory of the Ignition install location. These Python files will be distributed with every Symbolic Ai release so any changes made to these files will be lost. Python for user defined blocks should be placed into the `user-lib/pylib/ils/user/block` subdirectory of the Ignition install location. These files will not be overwritten. New blocks placed into either of these locations will automatically recognized upon a Gateway restart.

5.1.2.1 Examples

The distribution contains the following classes implemented in Python:

- Action - ils//block/action.py
- Arithmetic - ils/block/arithmetic.py
- FinalDiagnosis - ils/block/finaldiagnosis.py
- SQCDiagnosis - ils/block/sqcdiagnosis.py
- SubDiagnosis - ils/block/subdiagnosis.py

5.2 Scripting Interfaces

5.2.1 Installation of Documentation

The scripting interface and, for that matter all of the toolkit Java classes, are documented in a system called "javadoc". The documentation consists of .html files that are automatically generated from structured source code. This ensures that the documentation remains up-to-date as it is generated fresh with each build.

Previously the *javadoc* was only accessible from the Ignition Gateway web page by selecting the "documentation" link next to the BLT module listing. This link is still available.

The *javadoc* is also distributed as a separate collection of files and may be installed on the user development system for more convenient access. To install:

- From the installer's documentation page, scroll to the *Block Language Toolkit – Scripting Interface* link. Download the zip file.
- Unzip the downloaded file to a convenient location. The result of the unzipping will be a directory named "javadoc".
- Inside the *javadoc* directory, double-click on the file *index.html*. This will launch a browser session pointed at the root of the documentation tree. Bookmark this location for future reference.

5.2.2 Designer/Client

When accessing the Block Language Toolkit from Designer or Client scope, the actual block and diagram objects are not directly accessible. We are forced to place remote procedure calls into the Gateway in order to effect control. Return objects are often "descriptors" which contain read-only snapshots of the actual object state and properties.

Refer to the javadoc *ApplicationScriptFunctions* class for a complete list of methods available. This class defines the python scripting interface for Designer and Client-scope objects to control engine execution and to access or modify values from executing diagrams and blocks. All of these methods are located in the Python package:

system.ils.blt.diagram.

So, to execute a method from Python, for example:

```
import system
import system.ils.blt.diagram as script
descriptors =
script.getDiagramDescriptors(system.util.getProjectName())
for diagDescriptor in descriptors:
     if diagDescriptor.getPath()==”Application/Folder/DiagramName”:

     script.sendSignal(diagDescriptor.getId(),”TargetBlock”,”reset”,”
”)
```

The script above iterates over all known diagrams looking for the one with a particular path, then signals a named block on that diagram to reset.

Scripting access to the name of the currently configured datasource and tag providers are available through the *getToolkitProperty*() call. The following property names are of particular interest:

- `Database` – the production datasource
- `SecondaryDatabase` – the isolation datasource
- `Provider` – production tag provider
- `SecondaryProvider` – isolation tag provider

5.2.3 Gateway

Unlike in the Designer or Client, scripts executed in Gateway scope have direct access to the objects that are part of the Block Language Toolkit. Reliance on a scripting interface can often be replaced by more robust and functional direct interaction with toolkit objects and their properties. The possibilities are all documented in the *javadoc*.

Similar to Designer/Client scopes, there is a scripting interface. It uses the same package:

system.ils.blt.diagram.

The available methods are implemented in the *GatewayScriptFunctions* class. Using these methods, the same code listed above could be executed in Gateway scope – with one exception. The Ignition function, system.util.getProjectName() is not available in Gateway scope. In the Gateway, the getDiagramDescriptors() method takes no arguments and returns a list of all diagrams, independent of project.

An alternative style involves direct object interaction. The *PythonRequestHandler* object is available to any block implemented in python and is also available via the scripting interface. This same code example might also be implemented as follows:

```
import system
import system.ils.blt.diagram as script
descriptors = script.getDiagramDescriptors()
handler = script.getHandler()
for diagDescriptor in descriptors:
     if diagDescriptor.getPath()=="Application/Folder/DiagramName":
          diagram = handler.getBlock(diagDescriptor.getId())
          block = diagram.getBlockByName("TargetBlock")
          block.reset()
```

As can be seen in the example, actual block and diagram objects are available through *PythonRequestHandler* which is obtained from the script function *getHandler()*. Refer to *ProcessDiagram* and *ProcessBlock* interfaces documents to view all the methods available directly on a diagram or block.

Any block written in Python has access to the *PythonRequestHandler* object as a "handler" class member. The handler provides all services necessary for operation of the block and its integration into the diagram.

5.3 Classes with Dual Representations

Application, Family, Final Diagnosis, and SQC Diagnosis all have dual representations. The first is internal to the BLT module / gateway. This is where the entities are created, controlled, and configured. There is a parallel representation in the SQL*Server database. It is vitally important that the gateway representation and the database representation are consistent. The reason for the dual representation is that the BLT module is responsible for what happens up to the final diagnosis but the management of the recommendations made by the final diagnosis is implemented in Python based on the configuration of the application, family, and final diagnosis. The application, family, and diagram exist only in the Project tree. They provide a structure for prioritizing and managing final diagnosis. An application is also used to define the set of outputs and the constraints on those outputs that can be referenced by the final diagnosis belonging to the application.

5.3.1 Extension Functions.

The toolkit provides extension functions that will be called at opportune spots in the life cycle of specific blocks to maintain the synchronization of the gateway and the SQL*Server database. It is important to note that all of the functions are called due to some gesture in the designer but many of the functions run in gateway scope. This is important to consider when writing the Python that implements the function. It is also important to realize where to look for debug and error information: the wrapper log of the designer console log.

An extension is provided for each of these classes:

- application - com.ils.application
- family - com.ils.family
- diagram - com.ils.diagram
- final diagnosis - ils.block.finaldiagnosis.FinalDiagnosis
- SQC Diagnosis - ils.block.sqcdiagnosis.SQCDiagnosis.

A “handler” is supplied as a means of obtaining context within the script. The class *com.ils.blt.gateway.PythonRequestHandler* may be instantiated with a no-argument constructor within the extension scripts. The chart below summarizes actions that trigger calling the scripts:

Action	Extension Function	Objects Affected	Runs in Scope
File: Save	appProperties:save() famProperties:save() diaSave:save() fdProperties:save()	All, even if not modified	Gateway
App: Save Application	appProperties:save() famProperties:save() diaSave:save() fdProperties:save()	All, even if not modified and even if not a member of the target application.	Gateway
App: Duplicate	famProperties:save() diaSave:save()	Families and diagrams in the new application, but not the application, itself.	
App: Delete	appProperties:delete()	The target application	
Fam: Save Family	appProperties:save() famProperties:save() diaSave:save() fdProperties:save()	All, even if not modified and even if not a member of the target family.	
Fam: Duplicate	diaSave:save()	Diagrams in the new family, but not the family itself.	
Fam: Delete	famProperties:delete()	The target family	
Diagram: Save Diagram Diagram: Set State Diagram: Duplicate	diaSave:save() fdProperties:save()	The target diagram and any final diagnoses in it.	

Action	Extension Function	Objects Affected	Runs in Scope
Diagram: Delete	diaSave:delete() fdProperties:delete()	The target diagram and any final diagnoses on it.	
Delete final diagnosis on diagram, save diagram	diaSave:save() fdProperties:save() fdProperties:delete()	The target diagram and any final diagnoses remaining on it. The "delete" method is called on the final diagnosis that was removed.	
Final diagnosis: rename using property editor	fdProperties.rename()	The target final diagnosis.	Designer
SQC diagnosis: rename using property editor	sqcProperties.rename()	The target SQC diagnosis.	Designer

Figure 11 – Triggers for Extension Functions

5.3.2 Configuration Dialog Support

The following extension functions support the configuration editors. These editors are implemented in Swing and are accessed by selecting *Configure* after right-clicking on the node in the project tree for applications and families and by selecting *Configure Block* after right-clicking on block the final diagnosis and sqc diagnosis. The chart below defines the methods that are used to configure these blocks. It is a bit confusing that the Final Diagnosis and SQC diagnosis have both a property editor and a configuration dialog.

Entity	Property Editor	Configuration Editor
Application	No	Yes
Family	No	Yes
Diagram	No	No
Final Diagnosis	Yes	Yes
SQC Diagnosis	Yes	Yes

This table defines the functions that are called when configuring the objects. The *getAux()* function is called when the Swing dialog is launched. It fetches the configuration of the object from the database using the UUID of the block. The *setAux()* method is called when the user presses OK to save the configuration from the Swing dialog to the SQL*Server database. All of these functions run in designer scope.

Target	Extension Function
Application	appProperties.py
Family	famProperties.py
Final Diagnosis	fdProperties.py
SQC Diagnosis	sqcProperties.py

Appendix A - Debugging

A.1 Logging

The Ignition *LoggerEx* class is used throughout to implement logging for purposes of debugging code. This class, based on Log4j, is configurable at run-time by changing log levels in any of Gateway, Designer or Client scopes as appropriate. Separately-configurable loggers are created for each Java package. Available log levels are: `ERROR`, `WARN`, `INFO`, `DEBUG`, and `TRACE`. `INFO` is the default.

Appendix B - Functional Testing

foo

B.1 Block Test Project

A functional test project named *blt_block_test.proj* is supplied as part of project deliverables. Its goal is to execute PyUnit tests that comprehensively exercise interfaces of each block within the Block Language Toolkit. The test project has the following prerequisites:

- BLT Module – this contains the core toolkit as well as the blocks to be tested.
- BLTT Module – this contains a mock diagram, a "test harness" that encapsulates the block under test.
- Unit-Test Module – this module provides facilities for setting tag values from CSV-formatted files.
- SQLTags provider named "TAG"

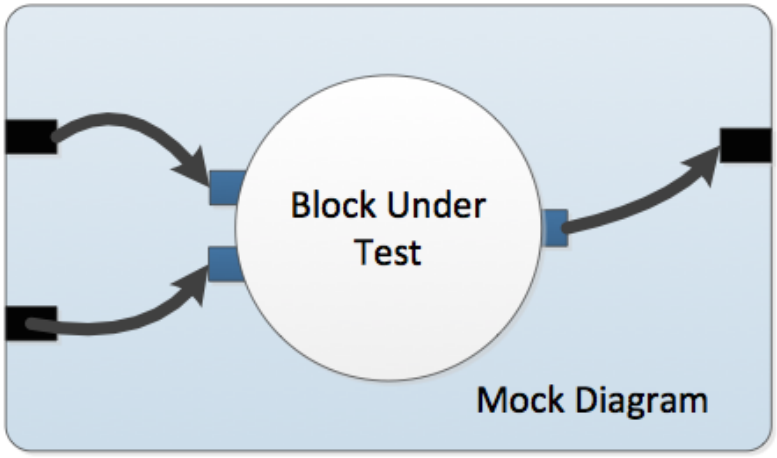
B.2 Block Language Toolkit Test Module

The Unit-Test module is a general-purpose Ignition module developed by ILS Automation to assist with PyUnit tests. In particular, it provides a scripting interface for the introduction of test-data into the project.

B.2.1 Mock Diagram

The core premise of the functional testing is that a block can be completely tested through its well-defined interfaces. This is accomplished using a special "mock" diagram with the following characteristics:

- Accepts exactly one block, the "block under test"
- Stubs connections to all block inputs and output
- Arbitrarily presents data on inputs
- Monitors all "transmissions" from the block
- Injects pertinent "transmissions" for the block to receive
- Reads results at output stubs



- **Figure 12 – Mock Diagram**

B.2.2 Scripting Interface

The following functions are provided by the BLT-test module “data” scripting interface. These functions provide for interaction between the Python functional test script and the mock diagram.

system.ils.test.mock. createMockDiagram

Description

Create a new mock diagram and add it to the list of diagrams known to the BlockController. The diagram has no valid resourceId and so is never saved permanently. It never shows in the designer. This call does not start subscriptions to tag changes. Subscriptions are triggered in response to a "start" call. This should be made after all to mock inputs and outputs are defined.
The diagram holds exactly one block, the "Unit Under Test".

Syntax

uuid = createMockDiagram(blockClass)

Parameters

String blockClass – the class of block to be tested.

Return

UUID – the unique ID of the newly created mock diagram.

system.ils.test.mock.addMockInput

Description

Define an input connected to the named port. This input is held as part of the mock diagram. Once defined, the input cannot be deleted. A separate (duplicate) input should be defined for every connection coming into the named port.

Syntax

addMockInput (diagram,path,type,port)

Parameters

UUID diagram – the unique ID of the mock diagram
String path – the fully qualified tag path of the tag that triggers this input.
String type – the data type of the tag `(STRING,INTEGER,DOUBLE, BOOLEAN,OBJECT)`
String port – name of the port on the block-under-test to which this input must connect.

system.ils.test.mock.addMockOutput

Description

Define an output connected to the named port. This output is held as part of the mock diagram. Once defined, the output cannot be deleted. It does not make sense to define more than one output for each outgoing port on the block-under-test.

Syntax

addMockOutput (diagram,path,type,port)

Parameters

UUID diagram – the unique ID of the mock diagram
String path – the fully qualified tag path of the tag that is set as a result of this output.
String type – the data type of the tag `(STRING,INTEGER,DOUBLE, BOOLEAN,OBJECT)`
String port – name of the port on the block under test to which this output must connect.

system.ils.test.mock.deleteMockDiagram

Description

Remove the specified test harness from the execution engine (block controller). The harness is stopped before being deleted.

Syntax

deleteMockDiagram (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock.forcePost

Description

Force the block under test to present a specified value on a specified output port.

Syntax

forcePost (diagram,port,value,)

Parameters

UUID diagram – the unique ID of the mock diagram

String port – name of the output port of the block-under-test on which the value is to be sent.

Object value – the new value. The value should be of type Boolean, Double, Integer, or String. In addition the value must be appropriate to the port.

system.ils.test.mock.getState

Description

Return the execution state of the block-under-test. This is used in conjunction with tests of the *reset* function.

Syntax

getState (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

Return

String state, the execution state of the diagram. `ACTIVE` or `INITIALIZED`.

system.ils.test.mock.isLocked

Description

Determine whether or not the block-under-test is locked.

Syntax

isLocked (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

Return

boolean value – true if the block is currently locked, otherwise false.

system.ils.test.mock.readValue

Description

Read the current value captured by the named output port.

Syntax

value = readValue (diagram,port)

Parameters

UUID diagram – the unique ID of the mock diagram

String port – name of the output port on the block-under-test on which to read the current value.

Return

QualifiedValue value – the current value held by the specified port

system.ils.test.mock.reset

Description

Reset the block-under-test. This is accomplished by calling its reset() method.

Syntax

reset (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock.setLocked

Description

Lock or unlock the block-under-test.

Syntax

setLocked (diagram,flag)

Parameters

UUID diagram – the unique ID of the mock diagram

boolean flag – true to lock the block-under-test, false to unlock it.

system.ils.test.mock.setProperty

Description

Set the value of the named property. This value ignores any type of binding. If the property is bound to a tag, then the value should be set by writing to that tag.

Syntax

setProperty (diagram,name,value)

Parameters

UUID diagram – the unique ID of the mock diagram

String name – name of the property of the block-under-test which is to be set.
Integer index – index of the connection into the named port. The index is zero-based.
Object value – the new value. The value should be of type Boolean, Double, Integer, or String.

system.ils.test.mock. startMockDiagram

Description

Start the test harness by activating subscriptions for bound properties and mock inputs.

Syntax

startMockDiagram (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock. stopMockDiagram

Description

Stop all property updates and input receipt by canceling all active subscriptions involving the harness.

Syntax

stopMockDiagram (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock.writeValue

Description

Simulate data arriving on the nth connection into the named input port.

Syntax

writeValue (diagram,port,index,value,quality)

Parameters

UUID diagram – the unique ID of the mock diagram
String port – name of the output port on the block-under-test on which to read the current value.
Integer index – index of the connection into the named port. The index is zero-based.
String value – the new value
String quality – the quality of the new value

Return

long timestamp – the current system in millisecs (time since start of the Unix epoch) corresponding to the creation of the value being presented on the block input. This value is useful for testing the propagation of timestamps through the block.

Appendix C - Application Testing

C.1 Introduction

The functional testing described in the previous section is designed to exercise each interface of each block class in isolation. The application test facility, on the other hand, tests the completed application as a whole. It is designed as an external harness that drives and monitors the production application at an accelerated rate.

This section describes features of the Block Language Toolkit implemented specifically in support of this facility.

C.2 Time Acceleration

All blocks use the WatchDogTimer class for timing ....

Appendix D - Adding Math Functions

D.1 Arithmetic Block

The *Arithmetic* block allows the customer developer to extend its repertoire with additional functions from the Apache Commons-Math library. Functions can also be custom coded. This section presents a discussion of the prototype block provided by ILS Automation.

D.2 Math Library

The Apache Commons-Math library is an open source collections of mathematical functions. It is already linked into the Block Language Toolkit and available for use. Documentation in the form of Javadoc may be found at

http://commons.apache.org/proper/commons-math/javadocs/api-3.3/index.html.

D.3 Example Modification

The prototype Arithmetic block can be found in Ignition external python in package *ils.block.Arithmetic.* The steps below highlight areas of the code which must be changed in order to add of new functions. These functions are not currently available with the built-in blocks. Additionally only functions available in the Apache Commons-Math library are available for inclusion in this particular block. Moreover these functions must be of the type that accept a single input and return a single output.

D.4 Function List

The list of available functions is presented to the block-user in a pull-down list inside the property editor.

Extension of the function list is a straightforward modification to the property binding in the following initialization method:

```
def initialize(self):
        self.className = 'ils.block.arithmetic.Arithmetic'
        self.properties['Function'] =
             {'value':'','editable':'True','bindingType':'OPTION',
'binding':'ABS,CEILING,COSINE,FLOOR,SINE,TANGENT,
 TO_RADIAN'}
```

D.5 Import Statement

Addition of a new function will require a new import statement to include the code from the math library. The sample below shows the statement used for the *Abs()* function.

```
import org.apache.commons.math3.analysis.function.Abs as Abs
```

D.6 Accept Function

When a block receives a new value on its input connection, its *acceptValue()* function is executed. The code below shows this code in its entirety for the prototype function. Additional functions involve creation of more *elif* clauses.

```
# Called when a value has arrived on one of our input ports.
# Compute the result, then propagate on the output.
    def acceptValue(self,invalue,quality,port):
        function = self.properties.get('Function',{}).get("value","")
        value = invalue       # Default behavior is a pass-through
        if len(function)>0:
            if function == 'ABS':
                absolute = Abs()
                value = absolute.value(value)
            elif function == 'CEILING':
                ceiling = Ceiling()
                value = ceiling.value(value)
            elif function == 'COSINE':
                cosine = Cosine()
                value = cosine.value(value)
            elif function == 'FLOOR':
                floor = Floor()
                value = floor.value(value)
            elif function == 'SINE':
                sine = Sine()
                value = sine.value(value)
            elif function == 'TANGENT':
                tan = Tangent()
                value = tan.value(value)
            elif function == 'TO_RADIAN':
                value = 0.0174532925*invalue;

           self.postValue('out',value,'good')
```

Appendix E - Custom Actions

E.1 Action Block

The *Action* block allows a customer-developer to extend the existing block language with custom python scripts. The configured script is executed when the input to the block newly matches a specified truth-state.

The python package must be in the external python area. The block does not have access to *project* or *shared* package scopes, which are the packages that are available via the Designer editor. Instead the recommended development environment is *Eclipse*. The actions scripts will be accessible to all projects.

E.2 Call Signature

The script is given the object that represents the originating block as its sole argument. The block is a derivative of class `ils.block.basicblock.BasicBlock`. A rich environment can be created from this object.

E.3 Request Handler

Each python block instance is supplied with a *handler* object. The handler is of class `com.ils.blt.gateway.PythonRequestHandler`. It offers access to a large number of project variables. For details of the methods supported see section 7.1.2.5.

E.4 Example

The example below demonstrates acquisition of contextual parameters via the block argument. If the *Action* block "script" property held the value:

Ils.actions.demo.act

then the file containing the code should be located in;

<*IgnitionInstallation*>/lib/pylibs/ils/actions/demo.py

Each level of the directory hierarchy should contain an empty `__init__.py` file to aid the python parser in traversing the path.

Demonstration of a custom action module:

```
def act(block):
    print block.getClassName()
    print block.uuid
    print block.parentuuid
    # The handler is a com.ils.blt.gateway.PythonRequestHandler
```

```
        print block.handler.getDefaultDatabase(block.parentuuid)
        print block.handler.getDefaultTagProvider(block.parentuuid)
```