**Block Language Toolkit**

**High Level Design**

**ExxonMobil - Baton Rouge Chemical Plant**

**Document Version: 0.3**

**Feb 27, 2014**

Table of Contents

1. Change History 3

1.1 References 3

1.2 Disclaimer 3

2. Introduction 4

2.1 Licensing 4

2.1.1 Ignition 4

2.1.2 Toolkit 4

2.2 Prerequisites 4

2.2.1 Java 4

2.2.2 Eclipse 5

2.2.3 Ignition 5

2.3 Internationalization 5

3. Architecture 6

3.1 Prototypes 7

3.2 Serialization 8

3.3 Custom Blocks 9

3.4 Model Definition 10

3.5 Gateway Scope 10

3.6 Designer 11

3.7 Client 11

4. Gateway 12

4.1 Gateway Functions 12

4.1.1 Dispatcher 12

4.1.2 Resource Changes 12

4.1.3 Block Execution 12

4.1.4 Tag Changes 12

4.2 Python Blocks 12

4.2.1 Scripting Interface 14

5. Designer 15

5.1 NavTree 15

5.1.1 Root Node 15

5.1.2 Application Nodes 16

5.1.3 Diagram Nodes 16

5.2 Menu 16

5.3 Palette 16

5.4 Icons 17

5.5 Diagrams 17

5.6 Blocks 18

5.7 Property Editor 18

5.8 Connections 18

5.9 RPC Interface 18

6. Client 20

6.1 Operator 20

6.2 Engineer 20

6.3 Communication with the Gateway 20

7. Migration 21

7.1 G2 Export Module 21

7.2 Migration Tool 22

8. Use Cases 23

8.1 Basic Diagram 23

8.1.1 Description 23

8.1.2 Key Requirements 23

# Change History

| Version | Date | Author | Changes |
| --- | --- | --- | --- |
| 0.1 | Jan 27, 2013 | C. Coughlin (ILS) | Initial version |
| 0.2 | Jan 28, 2014 | C. Coughlin (ILS) | Application based on block & connector interface |
| 0.3 | Feb 27, 2014 | C. Coughlin (ILS) | Added section on migration |

## References

| Version | Date | Title | Author |
| --- | --- | --- | --- |
| 2.0r1 | May 2006 | ExxonMobil – BRCP – Toolkit Migration Project | ILS Automation for EM - BRCP |
| 2.0r1 | March 2007 | ExxonMobil – BRCP – Diagnostics Toolkit, Engineer’s Documentation | ILS Automation for EM - BRCP |

## Disclaimer

The application and its documentation are works in-progress. At times, this document may describe features not-yet-implemented. In these cases the text will be annotated in *italics*.

# Introduction

The ExxonMobil Chemicals Diagnostic Toolkit is a collection of applications designed for the detection, management, annunciation, and response to events. The output of any of the constituent tool is a diagnosis, a recommended response to the problem. Details of existing applications may be found in the references.

ExxonMobil Chemicals requires that the existing Diagnostic Toolkit be ported from the current platform. Ignition™ from Inductive Automation has been chosen as the target platform. 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.

The Block Language Toolkit (BLT), the subject of this document, is just such a module. It has been designed as a hosting platform for the Diagnostics applications. The overarching requirement of the BLT project is to achieve operational equivalency with the current Diagnostic Toolkit on a G2™ platform. Additionally, because of the large number of legacy applications, a conversion utility is required. This utility must export individual legacy diagrams into a form ready for import into the new platform.

## Licensing

### Ignition

The application requires a commercial Ignition license to be obtained by ExxonMobil from Induction Automation. ExxonMobil is responsible for its installation on application servers.

### Toolkit

The Block Language Toolkit is the joint property of ILS Automation and ExxonMobil Chemicals. Any applications based on the toolkit in collaboration with ExxonMobil Chemicals remain the property of ExxonMobilChemicals.

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 included only those with licenses that allows free and unfettered use of the package.

## Prerequisites

### Java

The application requires Java JDK1.7. It is the customer’s responsibility to install the Java JDK on any system that will be running the application.

### Eclipse

For development support, compilation of Java 1.7 code requires Eclipse Juno, Kepler or newer. Any systems to be used for development require a Java SDK installation.

### Ignition

This application requires Ignition 7.7.0 or newer.

## Internationalization

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

# Architecture

An Ignition project is, 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 may be extended by 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 the block-and-connector interface and to implement the execution engine as a completely autonomous entity. The BLT 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

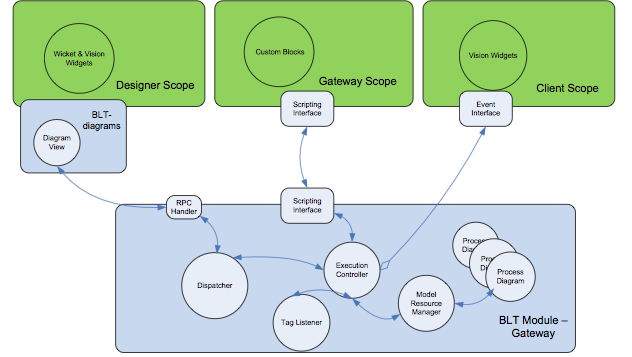


Figure 1 – Collaboration Diagram

## 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 be 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 well-integrated into the Ignition framework.

## Serialization

Serialization refers to the process of converting Java objects into a format suitable for storage. The process of recovering the Java objects from storage is called “marshalling”.

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.

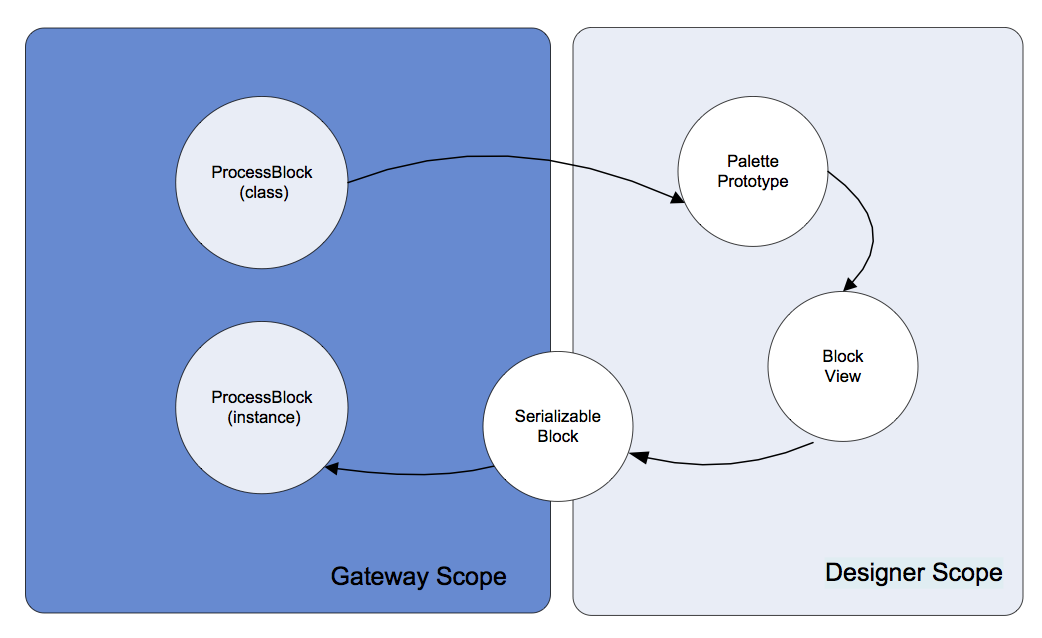


Figure 2 – Serialization Cycle

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:

* 1. Block class
  2. Icon to use in the palette
  3. A label
  4. Basic rendering style (Square ,Diamond ,Entry, Circle . . .)
  5. Anchor points (stubs)
  6. Icon embedded in drawing, if any
  7. Label rendered on drawing, if any

1. 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 the block. The property list is not transferred via the palette prototype.
2. 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.
3. 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.

## 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 and may be re-used in multiple applications.

## Model Definition

In a Model-View-Controller design, the model contains the definition of the core data structures, in this 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.

## 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 the project resource changes. This is the mechanism for remaining in synch with the Designer.

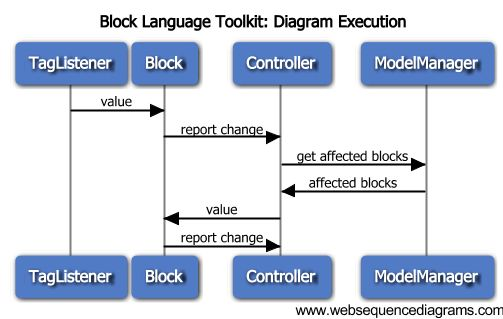


Figure 3 – Sequence Diagram

The diagram above depicts the sequence of operation 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 – 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.

## Designer

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

## Client

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

# 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.

## Gateway Functions

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

### Dispatcher

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

### 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” in the main Designer menu.

### 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 *evaluate()* method is invoked.

### 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.

## Python Blocks

*The toolkit may be extended with an Ignition Python-project that registers itself with the BLT module as a process block provider. Custom blocks may then be implemented within this project. The project is then usable within any application using the toolkit. Blocks that are a part of this project behave within the diagram in the same way as blocks that are distributed as part of the BLT module.*

*The code exhibit below shows the base class from which all Python-blocks must be derived.*

*# Copyright 2014 ILS Automation*

*#*

*# This class an abstract base class for all blocks in*

*# diagnostic diagrams.*

*#*

*class BasicBlock():*

*# Properties are a dictionary of attributes keyed by name*

*properties = \*

*{'class':{'value':'app.diagnostics.classes.BasicBlock','readonly':True}}*

*# Input ports are named stubs for incoming connections*

*inports = []*

*# Outports are named stubs for outgoing connections*

*outports = []*

*# Return a list of property names.*

*def getPropertyNames():*

*return properties.getKeys()*

*# Return a specified property. The property*

*# is a dictionary guaranteed to have a "value".*

*def getPropertyName(name):*

*properties.get(name,{})*

*# Accept notification that a value has arrived on an input*

*# The default implementation does nothing*

*def setValue(value,port):*

*pass*

*# Evaluate the block. This default implementation*

*# does nothing.*

*def evaluate():*

*pass*

Figure 4 – Python Base Class

### Scripting Interface

*The module supports a scripting interface for communication with the custom blocks that have been implemented in Python. The python-callable functions are:*

*system.ils.diagnostics.reportBlockCompletion*

*Description*

*Inform the gateway when a block completes processing and has placed a result value on one of its output paths.*

*Syntax*

*reportBlockCompletion (path, index,value,output)*

*Parameters*

*String path – tree path to the diagram. The path is a colon-delimited string derived from the Designer navigation tree.*

*int index – the index of the block within the diagram. This value in conjunction with the tree-path uniquely identifies the block.*

*object value – the result of the block calculation that is to be propagated to blocks connected to its output.*

*String output – the name of the output connection upon which to propagate the result.*

# Designer

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

## NavTree

The Designer’s navigation tree contains a “Diagram” node. Use this root node to create new applications and under them, new diagrams.

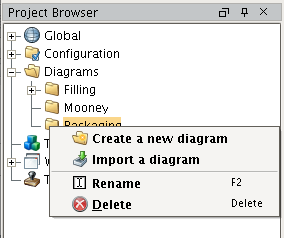


Figure 5 – Navigation Tree

`The sections below summarize the available menu options for each of three node levels.

### Root Node

* *Create a new application* – create a folder node that will contain a collection of 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.
* *Debug to log* – write a description of all current project resources to the Ignition designer log. This is useful only during development.

### Application Nodes

* *Create a new diagram* – create a diagram node and workspace. 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.

### Diagram Nodes

* double-click – opens the diagram workspace.
* *Export diagram* – display a file. On selection of a file, serialize the diagram and write it to the specified file path.
* *Rename* – change the name of the diagram.
* *Delete* – remove the diagram node and associated workspace.

## Menu

The Designer main menu has an additional function. 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.

## Palette

The palette is a Designer (Wicket) panel made viewable by a double-click in the Nav tree or by the menu selection described above.

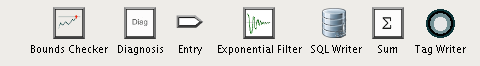


Figure 6 – Palette

## 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 pixel dimensions will be translated as needed.

## Diagrams

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

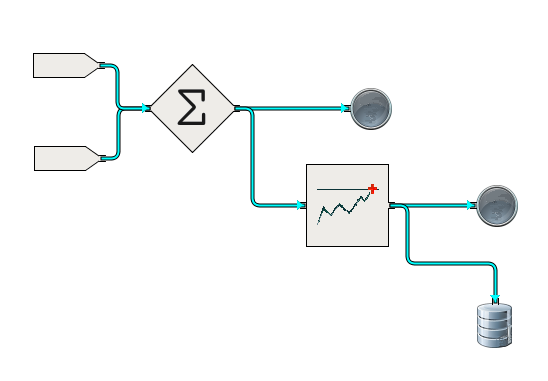


Figure 7 – Sample Diagram

## Blocks

This section describes each block individually.

## 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 panel for each custom property. Once changed, the block properties are not transmitted to the Gateway until the user executes a “Save”.

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

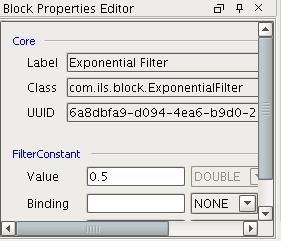


Figure 8 – Properties Editor

## Connections

There are four types of connections:

* Numerical (data)
* Truth-value (true,false,unknown)
* Control (signal)
* Text (diagnoses, recommendations)

## RPC Interface

The following interface is supported to allow Designer- or Client-scope objects to access values from the executing model in the Gateway.

getBlockProperties

Description

Return the current properties of a specified processing block. If the block does not exist, return the properties of a newly created block.

Syntax

List<String> getBlockProperties (projectId,resourceId,blockId,className)

Long projectId – the project to which the diagram belongs

Long resourceId – the project resource corresponding to this diagram

String blockId – the UUID of the block as a String

String className – class of the block. This is used in the case where the desired block does not yet exist in the Gateway,

Return

A list of block properties. The list contains strings of JSON-encoded BlockProperty objects.

startController

Description

Start the Gateway “engine”. Enable subscriptions and process block output.

Syntax

startController ()

stopController

Description

Shutdown the Gateway “engine”. Unsubscribe from any tag subscriptions. No longer process block output.

Syntax

stopController ()

# Client

*A Client view is the receiver of state or results for a diagram. The client comes in two flavors. The Engineer view is directed toward simulation and debugging. It features an animated “preview” view of a diagram that displays current status of the blocks. It cannot be edited.*

*The operator view emphasizes the display of conclusions and recommendations from the executing diagram. The operator’s windows feature standard Ignition components.*

## Operator

The operator’s view ...

## Engineer

The engineer’s view

## Communication with the Gateway

Event listener ...

# Migration

Due to the large number of applications built on the existing G2 platform, it is essential that a facility be developed to handle conversion of these applications to the Block Language Toolkit. The conversion takes place in 3 steps:

1. Export from G2. This requires a new G2 module that traverses a diagram workspace and extracts information about its blocks and connections. The output is written as a JSON document.

For each block the following is recorded:

* 1. Block class
  2. Block unique ID
  3. Block position – x,y in workspace coordinates
  4. Block attributes – name-value pairs, these can be nested

For each connection:

1. Upstream block unique ID
2. Upstream block connection port name
3. Downstream block unique ID
4. Downstream block connection port name
5. Connection class
6. Convert. The JSON document exported from G2 contains only references to G2 entities. It has no toolkit-specific information.

Conversion to BLT-specific JSON is performed by a *blt\_migrate* tool. The tool accepts a G2-JSON document on its standard input and writes BLT-JSON on its standard output. (This is done to facilitate bulk processing with scripts). The *blt\_migrate* tool connects to a database that contains translation tables mapping such things as G2 class names to BLT class names, and G2 method names to BLT procedures.

1. Import into Ignition. This step makes use of the standard import mechanism for diagrams. This is a manual operation.

Note that the manual activity of coding Ignition replacements for G2 blocks, procedures and methods must precede first of these steps.

## G2 Export Module

From a high level, the format of the JSON output is:

{“blocks”: [ { *serialized\_block1* }, …],

“connections”: [ { *serialized\_connection1* }, ….] }

As in Python syntax, {} denotes a dictionary, [] denotes a list. Whitespace is ignored. The details of the serialized blocks and connection are flexible.

During development, JSON syntax can be validated via the following free website: <http://www.freeformatter.com/json-validator.html>.

## Migration Tool

The migration tool accesses a SQLite database that contains the translation tables. (SQLite was chosen because it is easy to script, has no license restrictions and no installation. There is also an easy-to-use GUI editor for Windows/Linux or OSX available at <http://sourceforge.net/projects/sqlitedbrowser/files>).

# Use Cases

The section describes use cases that have helped shape the design of the tool kit.

## Basic Diagram



### Description

**T**his diagram has two numeric inputs. The inputs feed a sum block that adds the two inputs and puts the result on its output. The output of the sum block is the input to a high limit observation block configured with a high limit. The output of the observation block is a Boolean value that is true if the value is greater than the limit and false otherwise. The Boolean output is the input to a Final Diagnosis block that inserts a record into the database on the rising edge.

### Key Requirements

1. The value of the inputs must incorporate standard Ignition binding, so the inputs can come from an OPC tag, an expression, a database query, or the result of a script.
2. The sum block does not have any special properties.
3. The Hi Limit Observation block as a simple property which can be bound to anything
4. The input is event driven
5. Near simultaneous updates to the two inputs must be evaluated in a consistent manner.
6. For the purpose of this use case, the final diagnosis inserts a record into some database table.