Signal I/O Domain Workbook

A Pycca Translation

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	June 17, 2016	Initial draft.	GAM
1.1	August 20, 2016	Model translation completed and running.	GAM
1.2	October 10, 2016	Rework model to bring it into line with naming convensions used in the book.	GAM
1.2	October 11, 2016	Addition clean up to match naming conventions.	GAM
1.3	March 6, 2017	Rework in preparation for book release.	GAM

Contents

List of Figures

List of Tables

Introduction

This document describes the Signal I/O domain. The domain presented here is an example that is contained in the book, *Models To Code*.

This document is also a literate program which means that it contains both the descriptive material for the domain as well as the `pycca` source code that implements the domain. A document, in many different formats (*e.g.* PDF), can be generated from the source using `asciidoc`[1]. The source file is a valid `asciidoc` file. The `pycca` source code that implements the model can be extracted from the source using a literate programming tool named, `atangle`[2].

Important
This literate program document intermixes model elements and implementation elements. This makes the correspondence between the model and its translation very clear and simplifies propagating model changes into the translation. However, we must be clear that producing a running domain is a two step process. The model **must** be completed fully before the translation can be derived from it. Do not let the appearance of the final document suggest that the process of obtaining it was one of incremental refinement or that translating the model took place at the same time as formulating the model. Just as one does not write a novel from beginning to end in a single pass, one does not construct a domain in the same order as it is presented here.

Domain Mission

The mission of the Signal I/O domain is to provide an idealized view of control of the world external to the system.

Background

Much of the usefulness of a computer depends upon its ability to interact with the external environment. This may be a simple as displaying text on a graphics screen or as complicated as controlling a nuclear reactor. Interacting with the external environment always involves some type of *transducer*. A tranducer, for our purposes, is a device that converts a physical quantity to an electrical signal in such a way that measurements of the signal can be directly related to the physical quantity. A simple example of a transducer is a microphone. The microphone converts sound waves into an electrical signal. That signal can be brought into a computer system as a sequence of digtal values by converting the electrical signal using an *analog to digital converter* (ADC).

There are many different types of transducers and the signals they produce are brought into a computing environment in many different ways. Electrical engineers have developed a vast array of techniques for interfacing electrical signals to computers. However, there are some common aspects to all of them. Part of the purpose of the Signal I/O domain is to exploit the commonality of interfacing techniques to provide its clients with a simpler, consistent view of the external environment.

The Signal I/O domain itself does not directly read or write values to hardware. Projects will need some type of hardware access layer to access the physical hardware. Signal I/O does deal with *device units*. Handling device units implies that the bit pattern input or output by Signal I/O are appropriate for directly transfering to the hardware controls. Signal I/O delegates the actual transfer of the raw device value to the hardware because there are many ways that hardware values are interfaced to a system. Sometimes hardware access is mapped to memory. Other times there are intermediary busses such as SPI or I2C. Because access to hardware is so system specific, Signal I/O does not concern itself with that level of system operations in order to support better reusablity across systems.

Signal I/O does perform any necessary conversions between device units and *engineering units*. Engineering units are common units used in algorithms, such as volts or amps or kPa. Clients of Signal I/O need not be concerned with how the transducers of the system scale their indication of the measured physical quantities.

Finally, the Signal I/O domain presented here does not perform every conceivable function on signals. The model given here provides only those capabilities that are useful for the Automatic Lubrication domain to which it is bridged. We will point out below areas where this domain may be expanded to meet other requirements.

[1] http://www.methods.co.nz/asciidoc/
[2] http://repos.modelrealization.com/cgi-bin/fossil/mrtools

Limitations

This domain contains the capability to sample analog values from the external world. It uses delayed events to realize the sampling period. This implies that the model shown here is **not** suitable for high rate, precisely sampled data as might be required for a signal processing application. The time unit for delayed events is milliseconds and so the maximum rate of sampling can be no more frequent than 1000 Hz. To model high rate sampling requires other techniques. Most modern microcontrollers have peripherals that can accomplish precise high rate data sampling but the techniques involve coordinating the use of timers to initiate the conversion, a high clock rate into an ADC and often the use of a DMA controller to transfer the converted data out of the ADC. Managing multiple peripherals used togther along with a data buffering scheme requires a more sophisticated model and hardware access layer than is presented here.

The same limitation apply when attempting to generate high speed pulsed output such as a pulse width modulated (PWM) square wave that might be used to run an electric motor. The frequency required for such wave generation is such that most modern microcontrollers have specialized modes of timers to generate such signals (motor control being a very common application). Again the complexity of use of such peripherals will fall on the hardware access code and attempting to use delayed events to toggle output signals will, in general, not function as required. The ultimate performance of the code derived from this model is more suitable for blinking an LED rather than running a DC motor.

Data Types

Point ID
A small non-negative integer that is used to identify I/O Points.

```
<<external data types>>=
typedef uint8_t sio_Point_ID ;
```

Point Value
The Point Value data type is the set of all values for I/O Points. I/O Points typically have values that are real numbers and the Point Value type must be able to represent real numbers to the extent of the precision of the underlying hardware.

```
<<external data types>>=
typedef int32_t sio_Point_Value ;
```

Point Value Mask
The Point Value Mask data type is an unsigned version of the Point Value type.

```
<<internal data types>>=
typedef uint32_t Point_Value_Mask ;
```

Count
The Count data type holds general ordinal numbers.

```
<<internal data types>>=
typedef int Count ;
```

Msec
An integer value of time duration in the units of milliseconds.

```
<<internal data types>>=
typedef uint32_t Msec ;
```

Boolean
The Boolean type has two allowed values: false and true. The standard “C” `bool` type has the proper characteristics and we use it directly.

Converter ID
A small non-negative integer that is used to identify Signal Converters.

```
<<external data types>>=
typedef uint8_t sio_Converter_ID ;
```

Group ID
A small non-negative integer that is used to identify Conversion Groups.

```
<<external data types>>=
typedef uint8_t sio_Group_ID ;
```

Threshold ID
A small non-negative integer that is used to identify Point Thresholds.

```
<<external data types>>=
typedef uint8_t sio_Threshold_ID ;
```

Access Type
Hardware registers may or may not be able to act as ordinary RAM memory does. The Access Type data type has three values:

ReadWrite
Implies the hardware register may be read and written.

ReadOnly
Implies that the hardware register may only be read.

WriteOnly
Implies that the hardware register may only be written.

```
<<internal data types>>=
typedef enum {ReadWrite, ReadOnly, WriteOnly} Access_Type ;
```

Memory Type
Hardware registers can have side effects when they are accessed. The Memory Type data type has three values:

Memory
Implies that the hardware register behaves like ordinary memory.

ReadClear
Implies that the hardware register is bit encoded. Reading the register gives the status of the underlying hardware. Writing one to a particular bit clears the corresponding status. Writing zero to a particular bit does nothing.

ReadClearZero
Implies that the hardware register is bit encoded. Reading the register gives the status of the underlying hardware. Writing zero to a particular bit clears the corresponding status. Writing one to a particular bit does nothing.

```
<<internal data types>>=
typedef enum {Memory, ReadClear, ReadClearZero} Memory_Type ;
```

Edge Type
Asynchronous signals may be of interest when they transition from inactive to active, active to inactive or both. The Edge Type data type has three values:

Active
Implies only the transition from inactive to active is of interest.

Inactive
Implies only the transition from active to inactive is of interest.

BothActive
Implies both transitions are of interest.

```
<<internal data types>>=
typedef enum {Inactive, Active, BothActive} Edge_Type ;
```

Bit Count
The Bit Count data type has values that can count the number of bit in the binary representation of a raw hardware value.

```
<<internal data types>>=
typedef uint8_t Bit_Count ;
```

Bit Offset
The Bit Offset data type has values that describe the offset from the least significant bit of the binary representation of a raw hardware value.

```
<<internal data types>>=
typedef uint8_t Bit_Offset ;
```

Excursion Direction
Values of the Excursion_Direction type can have the values of Rising or Falling which indicate the direction across a threshold which causes an alarm.

```
<<internal data types>>=
typedef enum {Rising, Falling} Excursion_Direction ;
```

Classes and Relationships

External signals and actuators are represented by an **I/O Point**. An **I/O Point** can be one of three types (**R1**). A **Discrete Point** is a point that can take on a set of values that can be enumerated. A **Signalling Point** is an input point whose change of state can be detected asynchronously. A **Continuous Point** represents a signal that takes on values that can be represented by real numbers.

A **Discrete Point** may be contained on it own as a **Control Point** or may be a **Packed Point** (**R8**) which are bit fields within a **Control Point** (**R9**).

A **Signalling Point** indicates a change of state in the external world for devices that can be in two states. For example, a simple switch can be on or off.

A **Continuous Point** is scaled from device units to engineering units (or *vice versa*) by a **Point Scaling** (**R3**). **Continuous Points** are also either input or output (**R2**). **Continuous Input Points** may be monitored by a **Range Limitation** to determine if they exceed the threshold specified by a **Point Threshold** (**R7**). **Continuous Input Points** are collected into a **Conversion Group** (**R4**) as a set of points that must to be sampled at the same time. A **Signal Converter** represents a device that can obtain a digital value for input points. All the input points in a **Conversion Group** are wired to the same **Signal Converter** (**R5**). At any given point in time, a **Signal Converter** may be converting the values of the points for a **Conversion Group** (**R6**).

The figure below shows the class diagram for the Signal I/O domain.

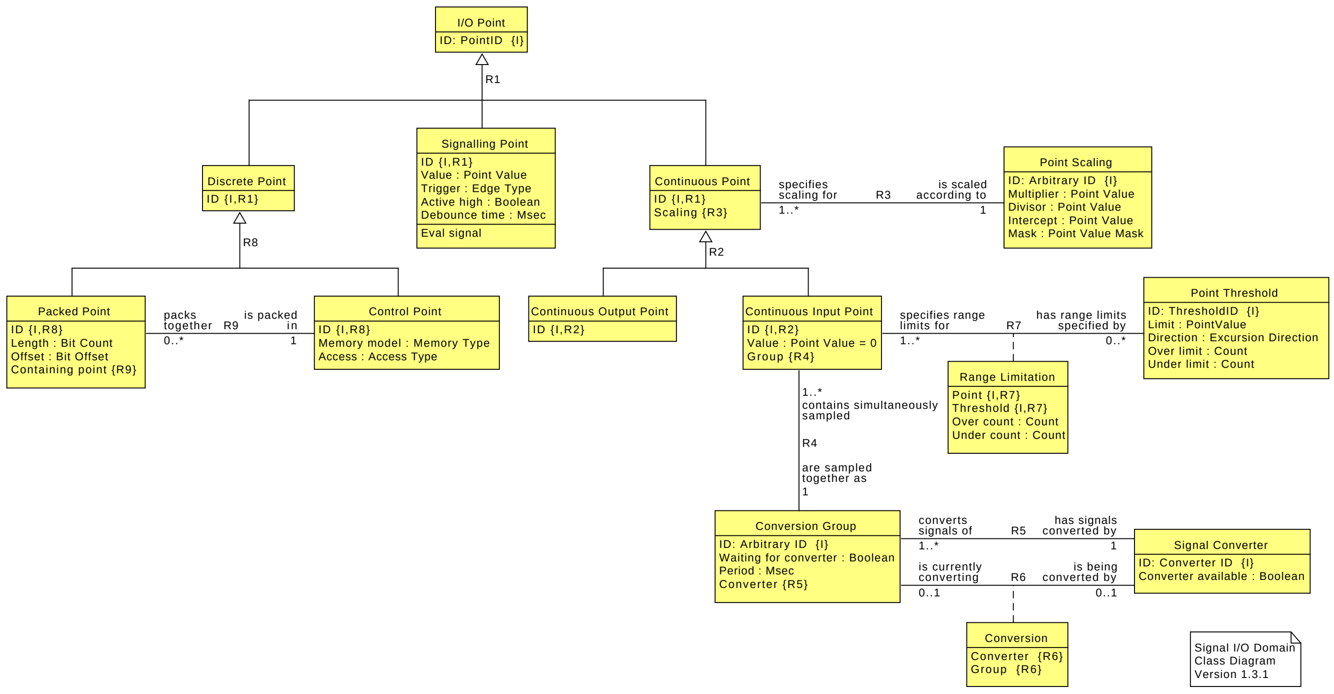


Figure 1: Signal I/O Domain Class Diagram

I/O Point

The fundamental abstraction provided by the Signal I/O domain is that of an I/O Point. Client domains may treat I/O Points as a single value that they can read or write to know about or control the external world.

For example, a client domain may wish to know the temperature of some piece of equipment. This information would be presented to the client domain bridge as the value read from an I/O Point.

So, an I/O Point represents an incoming signal or an outgoing control. Reading and/or writing the values of I/O Points is the means by which a client domain can affect the outside world. Each I/O Point represents a single, self-contained value that is a meaningful signal or control in the external world.

ID {I}

I/O Points are identified by small non-negative numbers.

Data Type	Point ID

R1 Generalization

- **I/O Point** is a **Discrete Point**, **Signalling Point** or a **Continuous Point**.

All points are categorized by whether their values represent a discrete enumerable set, an asychronous state change or represent values whose domain is the set of real numbers. This distinction is necessary because of the way system hardware treats the interfaces of the points.

Implementation

```
<<io point references>>=
subtype R1 union
```

```
    Discrete_Point
    Signalling_Point
    Continuous_Point
end
```

Discrete Point

Many types of point values in the external world can be represented as a small set of discrete values. For example, a motor may only run a three speeds: slow, medium and fast. Typically these would be encoded a small integers, say 0, 1 and 2. Points which can take on a finite number of discrete values are known as Discrete Points.

ID {I,R1}
The identfier of the Discrete Point.

Data Type	Refers to I/O Point.ID

Signalling Point

Some devices in the external world only have two states and the device can change from one state to the other asynchronously. For example, a simple pushbutton switch is like this. Hardware design often allows these state changes to be transported directly into the system, usually via an interrupt, without having to poll the current state of the device on a regular basis. A Signalling Point represents this type of device interface.

ID {I,R1}
The identfier of the Signalling Point.

Data Type	Refers to I/O Point.ID

Value
The current value of the signalling point

Data Type	Point Value

Trigger
The Trigger attribute records which signal edge we are interested in. For example, we may only be interested when the point changes from inactive to active. In some cases we may be interested in both transitions.

Data Type	Edge

Active High
The Active High attribute records which value of the point is considered to be the active value. If Active High is true, then a point value of 1 is consider the active state of the point for the purpose of determining our interest in the transition. Conversely, an Active High value of false indicates the point value of 0 is the active state.

Data Type	Boolean

Debounce time
When some signals change state, they may require some time for their values to settle into a stable arrangement. During

the settling time the value of the point may change multiple times and we do not want those changes to be given any significance. The Debounce Time is the number of milliseconds to wait before deciding if the state change of the point is real or just the result of some noise.

Data Type	Msec

Implementation

```
<<signalling point attributes>>=
attribute (sio_Point_Value Value) default {0}
attribute (Edge_Type Trigger)
attribute (bool Active_high)
attribute (Msec Debounce_time)
```

Continuous Point

Other types of point values can only be represented by real numbers. Quantities such as temperature and pressure can take on any value in a range. Points with these characteristics are known as Continuous Points.

It is certainly the case that since we are dealing with values held in a digital computer that there will be limits in the implementation of the precision of continuously varying values. The implementation must choose representations of Continuous Points that accomodate the needs of the application logic, but for the point of view of the domain we can treat Continuous Point values as simple real numbers.

ID {I,R1}
The identfier of the Continuous Point.

Data Type	Refers to I/O Point.ID

Scaling {R3}
The identifier of the Point Scaling instance used to convert the Continuous Point between engineering units and device units.

Data Type	Refers to Point Scaling.ID.

R8 Generalization

- **Discrete Point** is a **Packed Point** or a **Control Point**.

By hardware design, a Discrete Point is either wholly contained in a single control register or consists of a set of points packed together as bit fields in a control register.

Implementation

```
<<discrete point references>>=
subtype R8 union
    Packed_Point
    Control_Point
end
```

Packed Point

It is common for hardware designers to place several related peripheral controls in the same hardware register. A Packed Point recognizes such an arrangement and records the information necessary to treat the point as a distinct entity regardless of how the point value is placed in a hardware control register. A Packed Point can be described as a contiguous bit field within a larger control register.

ID {I,R8}
The identfier of the Packed Point.

Data Type	Refers to Discrete Point.ID

Length
The number of continuous bits occupied by the point.

Data Type	Bit Count

Offset
The offset in bits from the beginning of the control register to the least significant bit of the point value. We assume that bits are labeled such that bit number zero is the least significant bit of the control register.

Data Type	Bit Offset

Containing point {R9}
The identifier of the Control Point into which the Packed Point is placed.

Data Type	Refers to Control Point.ID.

Implementation

```
<<packed point attributes>>=
attribute (Bit_Count Length)
attribute (Bit_Offset Offset)
```

Control Point

A Control Point models a hardware register that provides an interface to the external world.

ID {I,R8}
The identfier of the Control Point.

Data Type	Refers to Discrete Point.ID

Memory model
It is a common hardware design practice to have control registers behave differently than ordinary RAM. This is necessary since sometimes it is desirable for the act of reading or writing a control register to have some side effect in the hardware. For example, consider a register that has a set of status bits that indicate if a particular condition in the peripheral is causing an interrupt. This might be a bit that indicates that a timer has expired. It may be convenient to pack several of these status bits together in the same register but each one must be controllable so it may be reset after it is serviced. The Memory

Model attribute describes the characteristics of the hardware register with respect to any side effects that access to the register might have. See the description of the Memory Type data type for the values and their meanings.

Data Type	Memory_Type

Access

Hardware control registers also differ from conventional RAM memory in the set of access operations that may be allowed. Most modern hardware peripherals are controlled with registers that may be read or written, with reading returning the last value written to the register. Other times, the semantics of the control register may be such that it can only be read or only be written. The Access attribute distinguishes these different types of access to hardware control registers. See the description of the Access Type data type for the values and their meanings.

Data Type	Access Type

Implementation

```
<<control point attributes>>=
attribute (Memory_Type Memory_model)
attribute (Access_Type Access)
```

R9 — Packed Point ⇒ Control Point

- **Packed Point** is packed in *exactly one* **Control Point**
- **Control Point** packs together *zero or more* **Packed Point**

Constraints on hardware design sometimes make it convenient to place several related hardware controls into the same hardware register. This arises from limitations in the register addressing space or because software interaction with the hardware is more convenient if several pieces of information may be accessed in a single physical register access. We do not allow a point to be spread among several registers. Not all Control Points hold multiple controls as it is sometimes a better interface design if it holds only a single hardware control.

Implementation

```
<<packed point references>>=
reference R9 -> Control_Point
```

Point Scaling

A Point Scaling represents the information required to convert a point value from device units to engineering units or *vice versa*. Hardware always works in device units. Such units represent an encoding that is meaningful to the external world. For example, a 12-bit analog to digital converter always reports values between 0 and 4095. These are device units in that they represent values specific to the underlying hardware. Client domains of Signal I/O always work in engineering units. Hardware design will then attribute an engineering unit significance to the device value. Say in this case each bit of the ADC value represents 0.25 kPa of pressure, then the range of the device units can represent 0 to 1023 kPa in engineering units.

The scaling supported by the domain is restricted to be linear. Most hardware operates linearly within the range of interest. Non-linear scaling would require a different approach than is presented here.

Continuous Input Points are converted from device units to engineering units. Continous Output Points are converted from engineering units to device units. For inputs, it is assumed that the following simple conversion formula applies.

$$eng = \left[\frac{dev * Multiplier}{Divisor} + Intercept \right] AND\, Mask$$

EQUATION 0.1: Conversion from Device to Engineering Units

where *AND* is the bit wise AND operation.

For outputs, the conversion from engineering units to device units is accomplished by:

$$dev = \left[\frac{(eng - Intercept) * Multiplier}{Divisor} \right] AND\, Mask$$

EQUATION 0.2: Conversion from Engineering to Device Units

ID {I}
A Point Scaling is identified arbitrarily.

Multiplier
The Multiplier and Divisor attributes form the slope of the implied line used for conversion. They are held as separate values to insure that values can be chose to avoid arithmetic overflow in the above equations.

Data Type	Point Value

Divisor
The Divisor atttribute along with the Multiplier attribute form the slope of the linear scaling.

Data Type	Point Value

Intercept
The Intercept attribute holds the value of the Y-axis intercept of the implied line used for conversion.

Data Type	Point Value

Mask
The Mask attribute is used to insure that converted values are contained within a fixed number of bits.

Data Type	Point_Value_Mask

Implementation

```
<<point scaling attributes>>=
attribute (sio_Point_Value Multiplier)
attribute (sio_Point_Value Divisor)
attribute (sio_Point_Value Intercept)
attribute (Point_Value_Mask Mask)
```

R3 — Continuous Point ⇒ Point Scaling

- **Continuous Point** is scaled according to *exactly one* **Point Scaling**

- **Point Scaling** specifies scaling for *one or more* **Continous Point**

Converting between device and engineering units is part of the primary service offered to Signal I/O domain clients. Each Continuous Point must be scaled from either engineering units to device units or *vice versa*. Since many Continuous Points may represent different instance of the same type of value, it is possible to use the same Point Scaling to scale them all.

Implementation

```
<<continuous point references>>=
reference R3 -> Point_Scaling
```

R2 Generalization

- **Continuous Point** is a **Continuous Output Point** or a **Continuous Input Point**

By hardware design, continuous values are transfered between the system and the external world in only one direction, either as inputs or outputs.

Implementation

```
<<continuous point references>>=
subtype R2 union
    Continuous_Output_Point
    Continuous_Input_Point
end
```

Continuous Output Point

A Continuous Output Point is that type of Continuous Point where values flow out to the external world. Typically, these types of points are implemented in hardware using a digital to analog converter (DAC) or a modulated square wave (PWM) with appropriate external hardware filtering circuitry.

ID {I,R8}
The identfier of the Continuous Output Point.

Data Type	Refers to Continuous Point.ID

Continuous Input Point

A Continuous Input Point is that type of Continuous Point where values flow in from the external world. Typically, these types of points are implemented in hardware using an analog to digital converter (ADC).

ID {I,R8}
The identfier of the Continuous Input Point.

Data Type	Refers to Continuous Point.ID

Value
The value of the point scaled to engineering units.

Data Type	Point Value

Group {I,R4}

The identfier of the Conversion Group to which the point belongs.

Data Type	Refers to Conversion Group.ID

Implementation

```
<<continuous input point attributes>>=
attribute (sio_Point_Value Value) default {0}
```

Point Threshold

Frequently, we are not so concerned with the specific value of Continuous Input Point. Rather we wish to know if the value has exceeded some boundary value. The Point Threshold class models a value threshold for an input point. This class also contains attributes to allow simple hysterisis of the point value to prevent declaring threshold excursions in the face of noisy input values.

ID {I,R8}

Point Threshold instances are identified by small non-negative integers.

Data Type	Threshold ID

Limit

The point value that defines the threshold limit.

Data Type	Point Value

Direction

The direction of change in the point value that is of interest. Threshold violations may be cause when the point value goes above the Limit or falls below the Limit.

Data Type	Excursion_Direction

Over limit

The number of times a point value must exceed the Limit to be declared out of range.

Data Type	Count

Under limit

The number of times a point value must be below the Limit to be declared in range.

Data Type	Count

Implementation

```
<<point threshold attributes>>=
attribute (sio_Point_Value Limit)
attribute (Excursion_Direction Direction)
attribute (Count Over_limit)
attribute (Count Under_limit)
```

R7 — Continuous Input Point ⇒ Point Threshold

- **Range Limitation** is an instance of **Continuous Input Point** has range limits specified by *zero or more* **Point Threshold**
- **Range Limitation** is an instance of **Point Threshold** specified range limits for *one or more* **Continuous Input Point**

A Continuous Input Point value may be monitored for being outside of prescribed ranges. It is possible to monitor several ranges for a point, including none at all. The specification of a threshold must be associated with some Continuous Input Point otherwise there would be no reason to state the value limits.

Implementation

```
<<range limitation references>>=
reference R7_PT -> Point_Threshold
reference R7_CIP -> Continuous_Input_Point

<<continuous input point references>>=
reference R7 ->>c Range_Limitation
```

Range Limitation

The Range Limitation class represents the monitoring of input values to determine if they have exceeded any threshold limits set for them.

Point {I,R7}
The identifier of the Continuous Input Point which is monitored.

Data Type	Refers to Continuous Input Point.ID.

Threshold {I,R7}
The identifier of the Point Threshold which supplies the limits for the point value.

Data Type	Refers to Point Threshold.ID.

Over count
The number of times the point value has exceeded it limit. This attribute is used as part of the hysterisis logic when diagnosing a threshold excursion.

Under count
The number of times the point value is less than its limit. This attribute is used as part of the hysterisis logic when diagnosing a threshold excursion.

Implementation

```
<<range limitation attributes>>=
attribute (Count Over_count) default {0}
attribute (Count Under_count) default {0}
```

Conversion Group

The **Conversion Group** class models a set of input values that must be sampled at the same time. As an example, signals that represents the voltage and current across a load can be used to compute the resistance of the load by Ohms law. However, the two signals must be sampled at the same time. In truth, a Signal Converter cannot sample two signals simultaneously. Most modern ADC hardware allows for sampling multiplexed inputs back to back as quickly as possible given the clock signal driving the ADC. This is usually fast enough that the signal may be considered sampled at the same time.

ID {I}

An arbitrary identifier for the conversion group.

Data Type	Group ID

Waiting For Converter

The **Waiting For Converter** attribute records whether the **Conversion Group** currently is currently waiting for a **Signal Converter** to be assigned to it.

Data Type	Boolean

Period

The number of milliseconds between each attempt to acquire the **Convertion Group** input values. This value determines the frequency of the sampled values.

Data Type	Msec

Converter {R5}

The identifier of the **Signal Converter** to which the **Conversion Group** is attached. All points that belong to a **Conversion Group** must be attached, by hardware design, to the same **Signal Converter**.

Data Type	Refers to Signal Converter.ID

Implementation

```
<<conversion group attributes>>=
attribute (bool Waiting_for_converter) default {false}
attribute (Msec Period)
```

R4 — Continuous Input Point ⇒ Conversion Group

- **Continuous Input Point** are sampled together as *exactly one* **Conversion Group**
- **Conversion Group** contains simultaneously sampled *one or more* **Continuous Input Point**

All Continuous Input Points are deemed to belong to some Conversion Group. For those cases where several points must be sampled together, then those points are placed in the same Convertion Group so the Sample Converter will sequence their conversion properly.

Implementation

```
<<continuous input point references>>=
reference R4 -> Conversion_Group

<<conversion group references>>=
reference R4 ->>c Continuous_Input_Point
```

Signal Converter

The Signal Converter class models the means by which values for the points in Conversion Groups are obtained. Typically, this represents some type of analog to digital conversion (ADC) hardware that is capable of sampling an analog voltage value and converting it to a digital number. ADC hardware functions by comparing an input voltage to a reference voltage and reports the fraction of the measured input relative to the reference. The Signal Converter is *not* aware of what the values of the signal represent. The usual arrangement is for a transducer to represent a physical quantity as an electrical voltage that is less than the reference voltage of the ADC. Usually the transducer voltage is scaled by external hardware to match the range implied by the ADC reference voltage and most transducers produce a linear relationship between the measured physical quantity and the voltage presented to the ADC.

ID {I}

Signal Converters are identified by small, non-negative integer values.

Data Type	Converter ID

Converter available

The **Converter Available** attribute records whether or not the **Signal Converter** is available to be used to convert input signals.

Data Type	Boolean

Implementation

```
<<signal converter attributes>>=
attribute (bool Converter_available) default {true}
```

Conversion

The Conversion class models an ongoing conversion of the points in a Conversion Group by a Signal Converter.

Converter {R6}

The idenifier of the Signal Converter performing this conversion.

Data Type	Refers to Signal Converter.ID

Group {R6}

The idenifier of the Conversion Group which is being converted.

Data Type	Refers to Conversion Group.ID

R5 — Conversion Group ⇒ Signal Converter

- **Conversion Group** has signals converted by *exactly one* **Signal Converter**
- **Signal Converter** converts signals of *one or more* **Conversion Group**

By hardware design, certain input points are physically wired to a specific **Signal Converter**. Any points that must be sampled as a group must all be wired to the same **Signal Converter**.

Implementation

```
<<conversion group references>>=
reference R5 -> Signal_Converter

<<signal converter references>>=
reference R5 ->>c Conversion_Group
```

R6 — Conversion Group ⇒ Signal Converter

- **Conversion** is an instance of **Conversion Group** is being converted by *at most one* **Signal Converter**
- **Conversion** is an instance of **Signal Converter** is currently converting *at most one* **Conversion Group**

At any point in time, a **Signal Converter** is actually converting at most one set of input points that form a **Conversion Group**. This rule arises from the design of the hardware ADC.

Implementation

```
<<signal converter references>>=
reference R6 -> Conversion_Group
```

Because the **Conversion** class has no behavior associated with it (*i.e.* no state model or class methods), we implement **R6** with a simple singular reference in the **Signal Converter** class and dispense with **Conversion** class all together.

Active Classes

The diagram below shows the collaboration of several classes in the domain to accomplish sampling Continuous Input Points and detecting any threshold excursions that might be associated with the points.

Clients may initiate and stop sampling of Conversion Groups and can be notified when new point values are available. Clients may also be notified when an input point goes into and out of specified value ranges. Because relationship, **R6**, is competitive, there is an assigner associated with each Signal Converter. The assigner uses a typical assigner protocol of events to sequence the assignment of a Signal Converter to a Conversion Group allowing the points in the group to be converted and read into the domain. When values are available, they are updated into the Continuous Input Point which, in turn, uses the Range Limitation to determine any excursion in the value of the point.

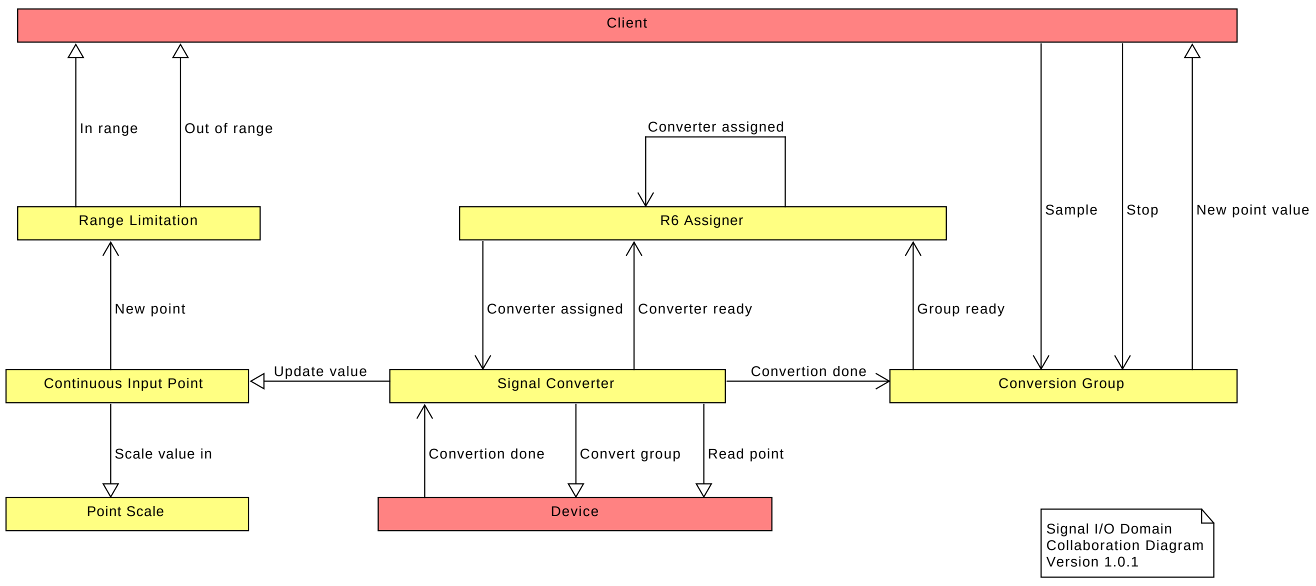


Figure 2: Signal I/O Domain Collaboration Diagram

R6 Assigner

By hardware design, a **Signal Converter** can only operate on at most one set of input points at a time. Since there may be several **Conversion Groups** connected to a **Signal Converter** and each **Conversion Group** may wish to be sampled at times asynchronous to the other groups, there is contention for the R6 relationship. We define an assigner to R6 to resolve the contention and insure that **Conversion Group** needs to use the **Signal Converter** are serialized properly. Note this is a multi-assigner and there must be one instance of the assigner for each instance of **Signal Converter**. It is a rule that conversion groups must only be converted by signal converters to which they are related by R5 and this reflects the physical arrangement of the hardware.

R6 Assigner State Model

The state model for the R6 Assigner is a typical design for insuring that a **Conversion Group** is paired properly with a **Signal Converter**.

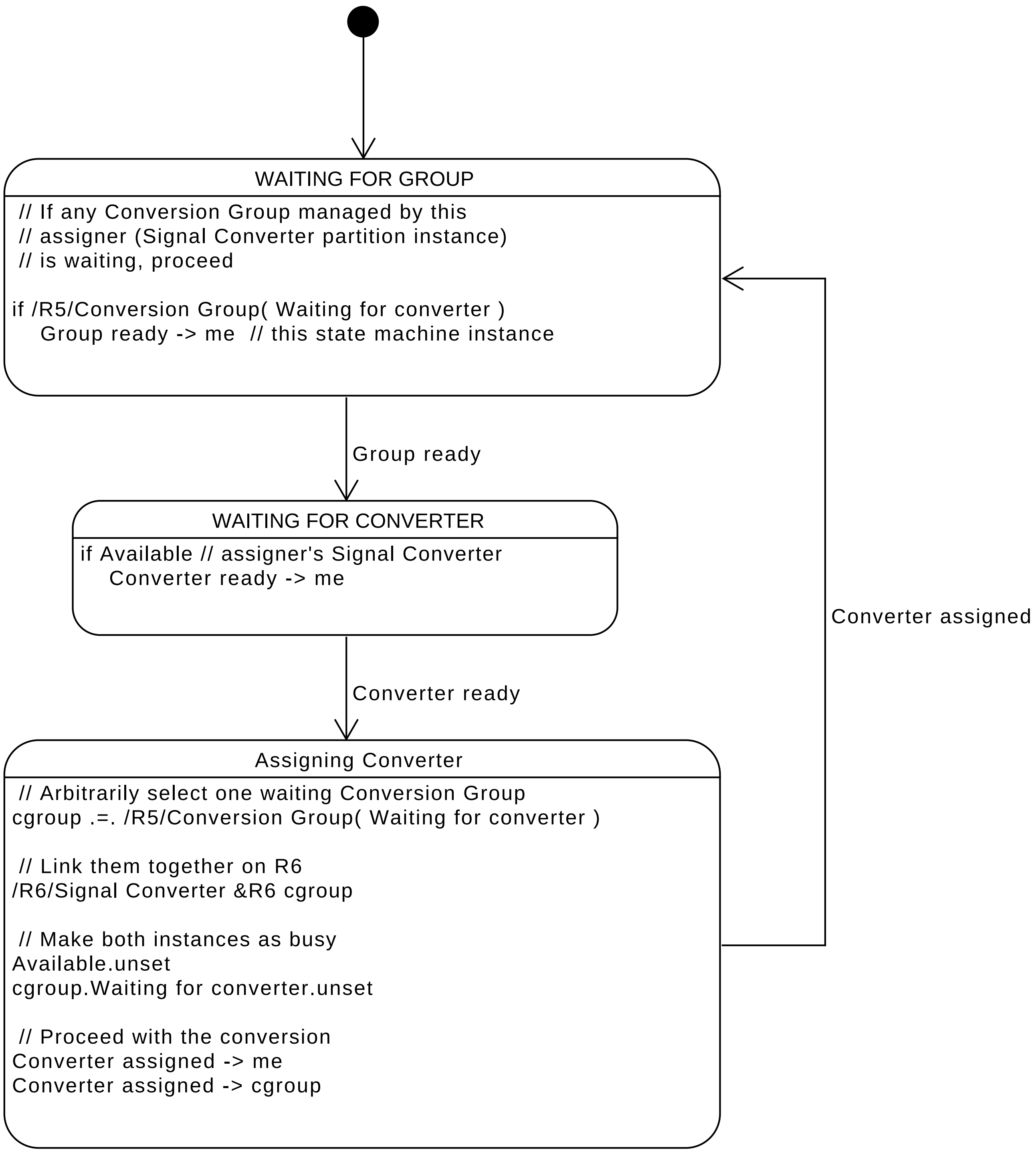

WAITING FOR GROUP
// If any Conversion Group managed by this
// assigner (Signal Converter partition instance)
// is waiting, proceed
if /R5/Conversion Group(Waiting for converter)
Group ready -> me // this state machine instance
Group ready
WAITING FOR CONVERTER
if Available // assigner's Signal Converter
Converter ready -> me
Converter assigned
Converter ready
Assigning Converter
// Arbitrarily select one waiting Conversion Group
cgroup .=. /R5/Conversion Group(Waiting for converter)
// Link them together on R6
/R6/Signal Converter &R6 cgroup
// Make both instances as busy
Available.unset
cgroup.Waiting for converter.unset
// Proceed with the conversion
Converter assigned -> me
Converter assigned -> cgroup

Figure 3: R6 Assigner State Model

Table 1: R6 Assigner Transition Matrix

	Group ready	**Converter ready**	**Assigning Converter**
WAITING FOR GROUP	WAITING FOR CONVERTER	IG	CH

Table 1: (continued)

	Group ready	Converter ready	Assigning Converter
WAITING FOR CONVERTER	IG	Assigning Converter	CH
Assigning Converter	CH	CH	WAITING FOR GROUP

R6 Assigner Pycca Transitions

```
<<R6 assigner state model>>=
default transition CH
initial state WAITING_FOR_GROUP

transition WAITING_FOR_GROUP - Group_ready -> WAITING_FOR_CONVERTER
transition WAITING_FOR_GROUP - Converter_ready -> IG

transition WAITING_FOR_CONVERTER - Group_ready -> IG
transition WAITING_FOR_CONVERTER - Converter_ready -> Assigning_Converter

transition Assigning_Converter - Group_ready -> IG
transition Assigning_Converter - Converter_ready -> IG
transition Assigning_Converter - Converter_assigned -> WAITING_FOR_GROUP
```

WAITING FOR GROUP State

The R6 Assigner remains in the **WAITING FOR GROUP** state until at least one **Conversion Group** has signaled that it needs to be converted.

Activity

```
 // If any Conversion Group managed by this
 // assigner (Signal Converter partition instance)
 // is waiting, proceed

if /R5/Conversion Group( Waiting for converter )
    Group ready -> assigner  // this state machine instance
```

Implementation

```
<<R6 assigner state model>>=
state WAITING_FOR_GROUP () {
    ClassRefVar(Signal_Converter, sc) = self->idclass ;

    ClassRefConstSetVar(Conversion_Group, cgset) ;
    PYCCA_forAllRelated(cgset, sc, R5) {
        ClassRefVar(Conversion_Group, cg) = *cgset ;
        if (cg->Waiting_for_converter) {
            PYCCA_generateToSelf(Group_ready) ;
            return ;
        }
    }
}
```

WAITING FOR CONVERTER State

The R6 Assigner remains in the **WAITING FOR CONVERTER** state until the **Signal Converter** associated with the assigner is ready to perform a conversion.

Activity

```
if Available // assigner’s Signal Converter
    Converter ready -> assigner
```

Implementation

```
<<R6 assigner state model>>=
state WAITING_FOR_CONVERTER () {
    ClassRefVar(Signal_Converter, sc) = self->idclass ;
    if (sc->Converter_available) {
        PYCCA_generateToSelf(Converter_ready) ;
    }
}
```

Assigning Converter State

The **Assigning Converter** state is where the R6 Assigner creates an instance of **R6** and initiates the conversion process.

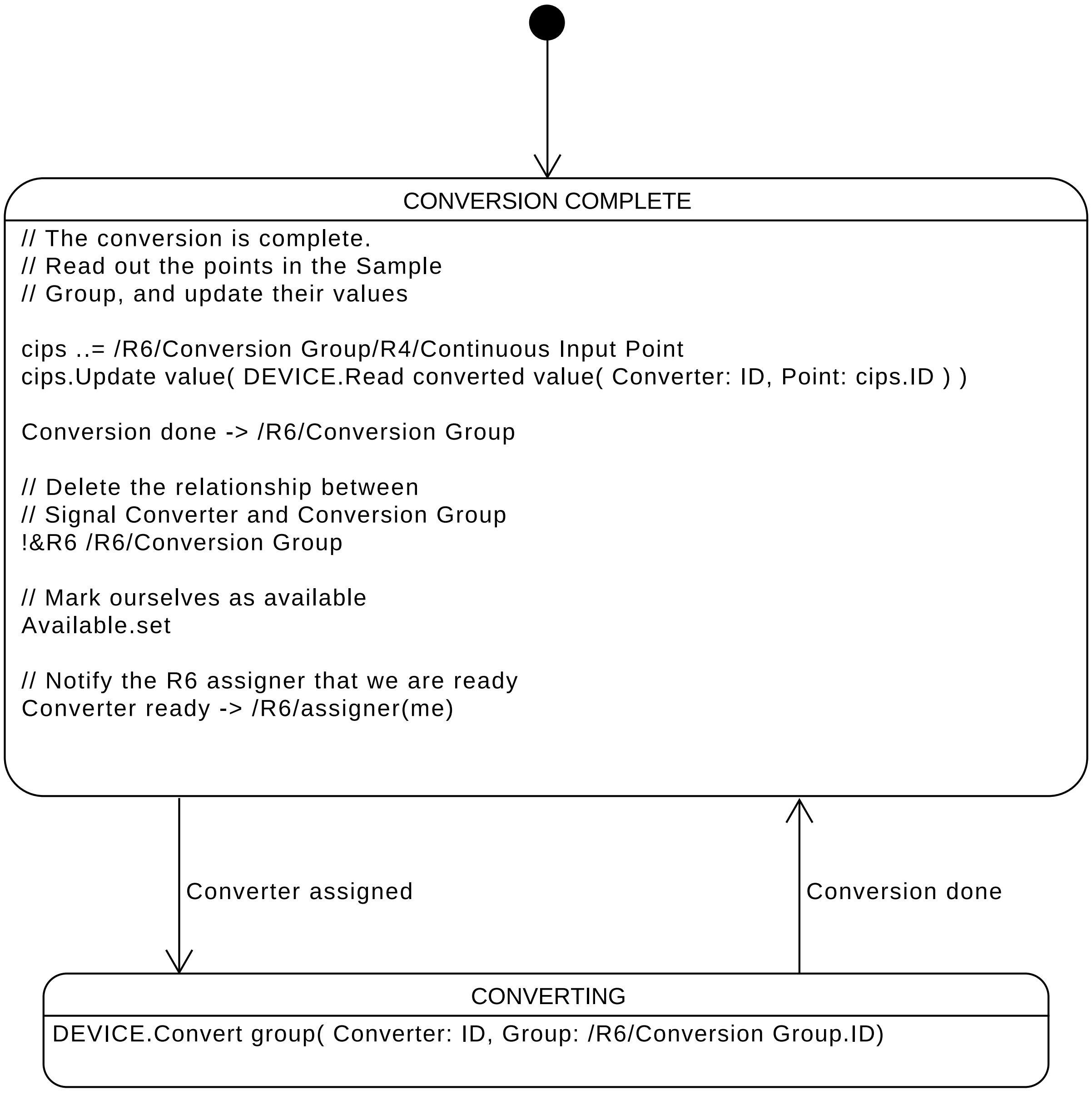
Activity

```
 // Select one waiting Conversion Group
cgroup .=. /R5/Conversion Group( Waiting for converter )

 // Link them together on R6
/R6/Signal Converter &R6 cgroup

 // Make both instances as busy
Available = false
cgroup.Waiting for converter = false

 // Proceed with the conversion
Converter assigned -> assigner
Converter assigned -> cgroup
```

Implementation

```
<<R6 assigner state model>>=
state Assigning_Converter () {
    ClassRefVar(Signal_Converter, sc) = self->idclass ;
    assert(sc->Converter_available) ;
    ClassRefConstSetVar(Conversion_Group, cgset) ;
    PYCCA_forAllRelated(cgset, sc, R5) {
        ClassRefVar(Conversion_Group, cg) = *cgset ;
        if (cg->Waiting_for_converter) {
            sc->R6 = cg ;

            sc->Converter_available = false ;
            cg->Waiting_for_converter = false ;
            PYCCA_generateToSelf(Converter_assigned) ;
            PYCCA_generate(Converter_assigned, Signal_Converter, sc, self) ;
            return ;
        }
    }
}
```

Signal Converter State Model



```
// The conversion is complete.
// Read out the points in the Sample
// Group, and update their values

cips ..= /R6/Conversion Group/R4/Continuous Input Point
cips.Update value( DEVICE.Read converted value( Converter: ID, Point: cips.ID ) )

Conversion done -> /R6/Conversion Group

// Delete the relationship between
// Signal Converter and Conversion Group
!&R6 /R6/Conversion Group

// Mark ourselves as available
Available.set

// Notify the R6 assigner that we are ready
Converter ready -> /R6/assigner(me)
```

```
DEVICE.Convert group( Converter: ID, Group: /R6/Conversion Group.ID)
```

Figure 4: Signal Converter State Model

Table 2: Signal Converter Transition Table

	Converter assigned	**Conversion done**
CONVERSION COMPLETE	CONVERTING	CH
CONVERTING	CH	CONVERSION COMPLETE

Implementation

```
<<signal converter state model>>=
initial state CONVERSION_COMPLETE
default transition CH

transition CONVERSION_COMPLETE - Converter_assigned -> CONVERTING

transition CONVERTING - Conversion_done -> CONVERSION_COMPLETE
```

CONVERSION COMPLETE State

Activity

```
 // The conversion is complete.
 // Read out the points in the Sample
 // Group, and update their values

 cips ..= /R6/Conversion Group/R4/Continuous Input Point
 cips.Update value( DEVICE.Read converted value( Converter: ID, Point: cips.ID ) )

 Conversion done -> /R6/Conversion Group

 // Delete the relationship between
 // Sample Converter and Conversion Group
 !&R6 /R6/Conversion Group

 // Mark ourselves as available
 Available = true

 // Notify the R6 assigner that we are ready
 Converter ready -> /R6/assigner(me)
```

Implementation

```
<<signal converter state model>>=
state CONVERSION_COMPLETE () {
    ClassRefVar(Conversion_Group, cg) ;
    cg = self->R6 ;
    assert(cg != NULL) ;
    ClassRefConstSetVar(Continuous_Input_Point, cipset) ;
    PYCCA_forAllRelated(cipset, cg, R4) {
        ClassRefVar(Continuous_Input_Point, cip) = *cipset ;
        ClassRefVar(IO_Point, iop) = PYCCA_unionSupertype(
            PYCCA_unionSupertype(cip, Continuous_Point, R2), IO_Point, R1) ;
        sio_Point_Value value = ExternalOp(DEVICE_Read_converted_value)(
                PYCCA_idOfSelf, PYCCA_idOfRef(IO_Point, iop)) ;
        InstOp(Continuous_Input_Point, updateValue)(cip, value) ;
    }
    PYCCA_generate(Conversion_done, Conversion_Group, cg, self) ;

    self->R6 = NULL ;

    self->Converter_available = true ;
    ClassRefVar(R6_Assigner, r6asgn) = self->assigner ;
    PYCCA_generate(Converter_ready, R6_Assigner, r6asgn, self) ;
}
```

CONVERTING State

Activity

```
DEVICE.Convert group( Converter: ID, Group: /R6/Conversion Group.ID)
```

Implementation

```
<<signal converter state model>>=
state CONVERTING () {
    ClassRefVar(Conversion_Group, cg) = self->R6 ;
    assert(cg != NULL) ;
    ExternalOp(DEVICE_Convert_group)(PYCCA_idOfSelf,
        PYCCA_idOfRef(Conversion_Group, cg)) ;
}
```

Conversion Group State Model

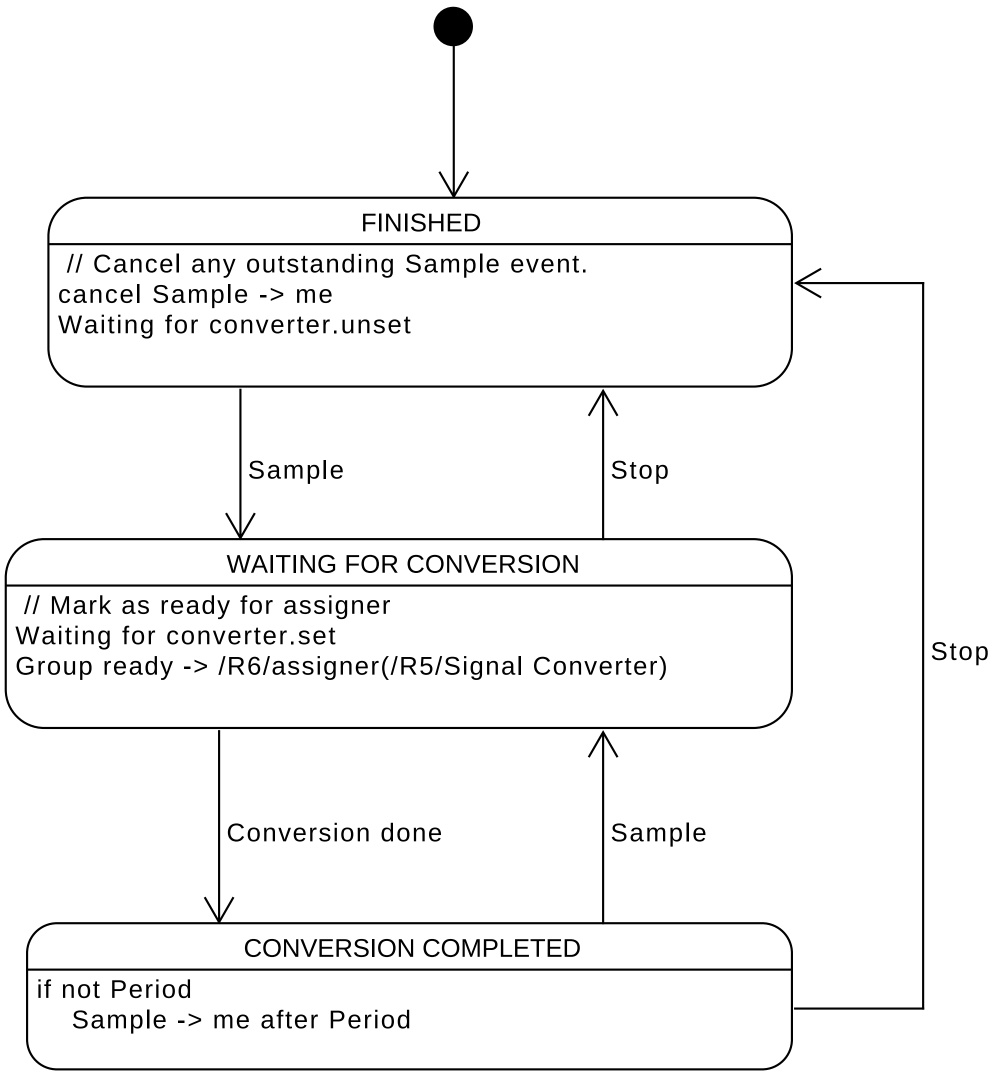

FINISHED
// Cancel any outstanding Sample event.
cancel Sample -> me
Waiting for converter.unset
Sample
Stop
WAITING FOR CONVERSION
// Mark as ready for assigner
Waiting for converter.set
Group ready -> /R6/assigner(/R5/Signal Converter)
Stop
Conversion done
Sample
CONVERSION COMPLETED
if not Period
Sample -> me after Period

Figure 5: Conversion Group State Model

Table 3: Conversion Group Transition Table

	Sample	Stop	Conversion done
FINISHED	WAITING FOR CONVERSION	IG	IG
WAITING FOR CONVERSION	WAITING FOR CONVERSION	FINISHED	CONVERSION COMPLETED
CONVERSION COMPLETED	WAITING FOR CONVERSION	FINISHED	CH

Implementation

```
<<conversion group state model>>=
initial state FINISHED
default transition CH

transition FINISHED - Sample -> WAITING_FOR_CONVERSION
transition FINISHED - Stop -> IG
transition FINISHED - Conversion_done -> IG

transition WAITING_FOR_CONVERSION - Sample -> WAITING_FOR_CONVERSION
transition WAITING_FOR_CONVERSION - Stop -> FINISHED
transition WAITING_FOR_CONVERSION - Conversion_done -> CONVERSION_COMPLETED

transition CONVERSION_COMPLETED - Sample -> WAITING_FOR_CONVERSION
transition CONVERSION_COMPLETED - Stop -> FINISHED
```

FINISHED State

Activity

```
 // Cancel any outstanding Sample event.
cancel Sample -> me
Waiting for converter = false
```

Implementation

```
<<conversion group state model>>=
state FINISHED() {
    PYCCA_cancelDelayedToSelf(Sample) ;
    self->Waiting_for_converter = false ;
}
```

WAITING FOR CONVERSION State

Activity

```
 // Mark as ready for assigner
Waiting for converter = true
Group ready -> /R6/assigner(/R5/Signal Converter)
```

Implementation

```
<<conversion group state model>>=
state WAITING_FOR_CONVERSION() {
    self->Waiting_for_converter = true ;
    ClassRefVar(R6_Assigner, r6asgn) = self->R5->assigner ;
    PYCCA_generate(Group_ready, R6_Assigner, r6asgn, self) ;
}
```

CONVERSION COMPLETED State

Activity

```
if not Period
    Sample -> me after Period
```

Implementation

```
<<conversion group state model>>=
state CONVERSION_COMPLETED() {
    if (self->Period != 0) {
        PYCCA_generateDelayedToSelf(Sample, self->Period) ;
    }
}
```

Range Limitation State Model

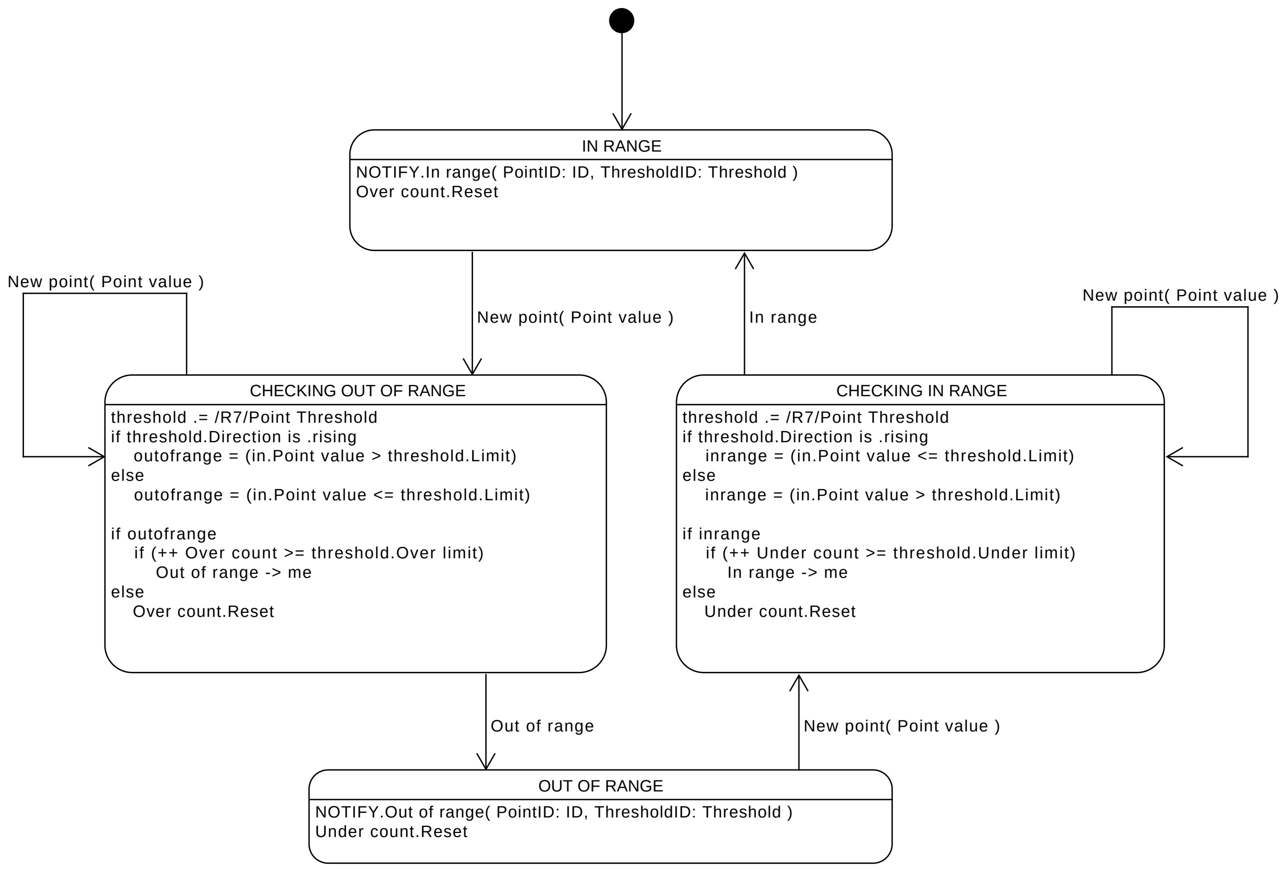

IN RANGE
NOTIFY.In range(PointID: ID, ThresholdID: Threshold)
Over count.Reset
New point(Point value)
In range
New point(Point value)
New point(Point value)
CHECKING OUT OF RANGE
threshold .= /R7/Point Threshold
if threshold.Direction is .rising
outofrange = (in.Point value > threshold.Limit)
else
outofrange = (in.Point value <= threshold.Limit)
if outofrange
if (++ Over count >= threshold.Over limit)
Out of range -> me
else
Over count.Reset
CHECKING IN RANGE
threshold .= /R7/Point Threshold
if threshold.Direction is .rising
inrange = (in.Point value <= threshold.Limit)
else
inrange = (in.Point value > threshold.Limit)
if inrange
if (++ Under count >= threshold.Under limit)
In range -> me
else
Under count.Reset
Out of range
New point(Point value)
OUT OF RANGE
NOTIFY.Out of range(PointID: ID, ThresholdID: Threshold)
Under count.Reset

Figure 6: Range Limitation State Model

Table 4: Range Limitation Transition Table

	New point	**In range**	**Out of range**
IN RANGE	CHECKING OUT OF RANGE	CH	CH
CHECKING OUT OF RANGE	CHECKING OUT OF RANGE	CH	OUT OF RANGE
OUT OF RANGE	CHECKING IN RANGE	CH	CH
CHECKING IN RANGE	CHECKING IN RANGE	IN RANGE	CH

Implementation

```
<<range limitation state model>>=
initial state IN_RANGE
default transition CH

transition IN_RANGE - New_point -> CHECKING_OUT_OF_RANGE

transition CHECKING_OUT_OF_RANGE - New_point -> CHECKING_OUT_OF_RANGE
transition CHECKING_OUT_OF_RANGE - Out_of_range -> OUT_OF_RANGE

transition OUT_OF_RANGE - New_point -> CHECKING_IN_RANGE

transition CHECKING_IN_RANGE - New_point -> CHECKING_IN_RANGE
transition CHECKING_IN_RANGE - In_range -> IN_RANGE
```

IN RANGE State

Activity

```
NOTIFY.In range( PointID: ID, ThresholdID: Threshold )
Over count.Reset
```

Implementation

```
<<range limitation state model>>=
state IN_RANGE() {
    ClassRefVar(Continuous_Input_Point, cip) = self->R7_CIP ;
    ClassRefVar(Continuous_Point, cp) = PYCCA_unionSupertype(cip,
        Continuous_Point, R2) ;
    ClassRefVar(IO_Point, iop) = PYCCA_unionSupertype(cp,
        IO_Point, R1) ;
    ExternalOp(NOTIFY_In_range)(PYCCA_idOfRef(IO_Point, iop),
            PYCCA_idOfRef(Point_Threshold, self->R7_PT)) ;
    self->Over_count = 0 ;
}
```

CHECKING OUT OF RANGE State

Activity

```
threshold .= /R7/Point Threshold
if threshold.Direction is .rising
    outofrange = (in.Point value > threshold.Limit)
else
    outofrange = (in.Point value <= threshold.Limit)

if outofrange
    if (++ Over count >= threshold.Over limit)
        Out of range -> me
else
    Over count.Reset
```

Implementation

```
<<range limitation state model>>=
state CHECKING_OUT_OF_RANGE(
    sio_Point_Value pointValue) {
    ClassRefVar(Point_Threshold, pt) = self->R7_PT ;

    bool outRange = pt->Direction == Rising ?
            rcvd_evt->pointValue > pt->Limit :
            rcvd_evt->pointValue <= pt->Limit ;
    if (outRange) {
        if (++self->Over_count >= pt->Over_limit) {
            PYCCA_generateToSelf(Out_of_range) ;
        }
    } else {
        self->Over_count = 0 ;
    }
}
```

OUT OF RANGE State

Activity

```
NOTIFY.Out of range( PointID: ID, ThresholdID: Threshold )
Under count.Reset
```

Implementation

```
<<range limitation state model>>=
state OUT_OF_RANGE() {
    ClassRefVar(Continuous_Input_Point, cip) = self->R7_CIP ;
    ClassRefVar(Continuous_Point, cp) = PYCCA_unionSupertype(cip,
        Continuous_Point, R2) ;
    ClassRefVar(IO_Point, iop) = PYCCA_unionSupertype(cp,
        IO_Point, R1) ;
    ExternalOp(NOTIFY_Out_of_range)(PYCCA_idOfRef(IO_Point, iop),
            PYCCA_idOfRef(Point_Threshold, self->R7_PT)) ;
    self->Under_count = 0 ;
}
```

CHECKING IN RANGE State

Activity

```
threshold .= /R7/Point Threshold
if threshold.Direction is .rising
    inrange = (in.Point value <= threshold.Limit)
else
    inrange = (in.Point value > threshold.Limit)

if inrange
    if (++ Under count >= threshold.Under limit)
        In range -> me
else
    Under count.Reset
```

Implementation

```
<<range limitation state model>>=
state CHECKING_IN_RANGE(
    sio_Point_Value pointValue) {
    ClassRefVar(Point_Threshold, pt) = self->R7_PT ;

    bool inRange = pt->Direction == Rising ?
            rcvd_evt->pointValue <= pt->Limit :
            rcvd_evt->pointValue > pt->Limit ;
    if (inRange) {
        if (++self->Under_count >= pt->Under_limit) {
            PYCCA_generateToSelf(In_range) ;
        }
    } else {
        self->Under_count = 0 ;
    }
}
```

Signalling Point State Model

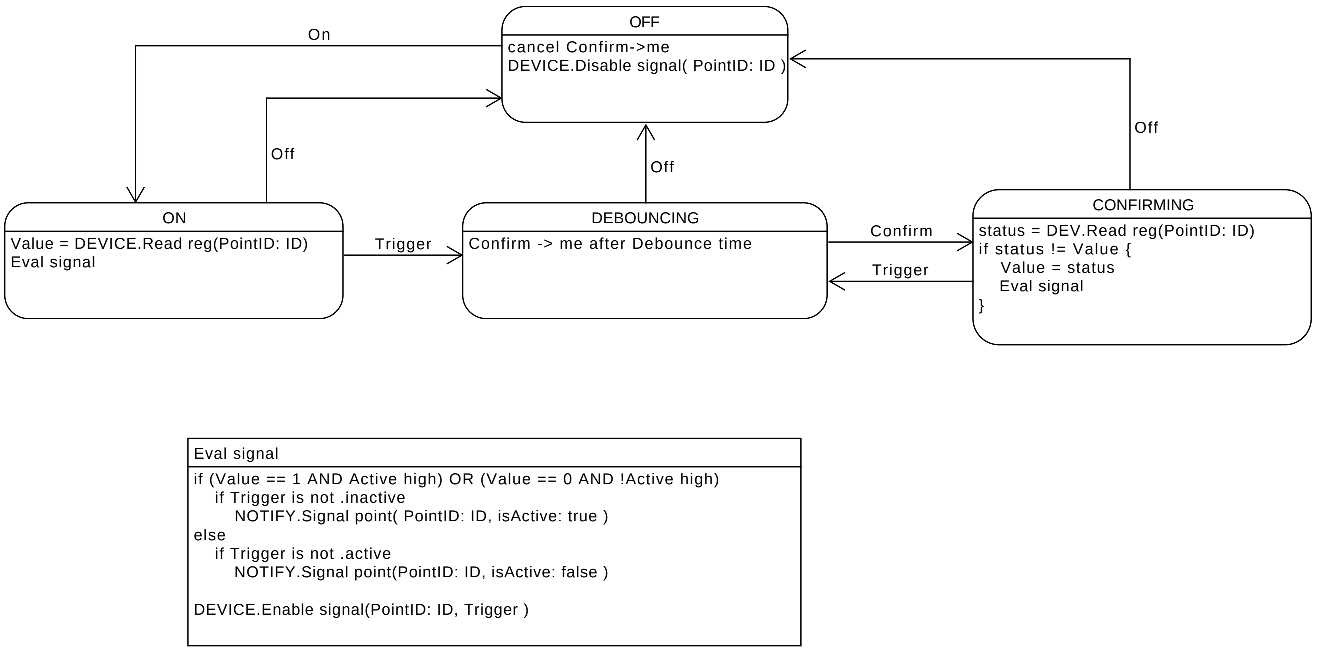

OFF
cancel Confirm->me
DEVICE.Disable signal(PointID: ID)
On
Off
Off
Off
ON
Value = DEVICE.Read reg(PointID: ID)
Eval signal
Trigger
DEBOUNCING
Confirm -> me after Debounce time
Confirm
Trigger
CONFIRMING
status = DEV.Read reg(PointID: ID)
if status != Value {
Value = status
Eval signal
}
Eval signal
if (Value == 1 AND Active high) OR (Value == 0 AND !Active high)
if Trigger is not .inactive
NOTIFY.Signal point(PointID: ID, isActive: true)
else
if Trigger is not .active
NOTIFY.Signal point(PointID: ID, isActive: false)
DEVICE.Enable signal(PointID: ID, Trigger)

Figure 7: Signalling Point State Model

Table 5: Signalling Point Transition Table

	On	Off	Confirm	Trigger
OFF	ON	IG	CH	IG
ON	IG	OFF	CH	DEBOUNCING
DEBOUNCING	IG	OFF	CONFIRMING	IG
CONFIRMING	IG	OFF	CH	DEBOUNCING

Implementation

```
<<signalling point state model>>=
initial state OFF
default transition IG

transition OFF - On -> ON
transition OFF - Confirm -> CH

transition ON - Off -> OFF
transition ON - Confirm -> CH
transition ON - Trigger -> DEBOUNCING

transition DEBOUNCING - Off -> OFF
transition DEBOUNCING - Confirm -> CONFIRMING

transition CONFIRMING - Off -> OFF
transition CONFIRMING - Confirm -> CH
transition CONFIRMING - Trigger -> DEBOUNCING
```

OFF

Activity

```
cancel Confirm->me
DEVICE.Disable signal( PointID: ID )
```

Implementation

```
<<signalling point state model>>=
state OFF() {
    PYCCA_cancelDelayedToSelf(Confirm) ;
    ExternalOp(DEVICE_Disable_signal)(
            PYCCA_idOfRef(IO_Point, PYCCA_unionSupertype(self, IO_Point, R1))) ;
}
```

ON

Activity

```
Value = DEVICE.Read reg(PointID: ID)
Eval signal
```

Implementation

```
<<signalling point state model>>=
state ON() {
    sio_Point_ID ptid = PYCCA_idOfRef(IO_Point,
            PYCCA_unionSupertype(self, IO_Point, R1)) ;
    self->Value = ExternalOp(DEVICE_Read_reg)(ptid) ;
    ThisClassInstOp(evalSignal)(self) ;
}
```

DEBOUNCING

Activity

```
Confirm -> me after Debounce time
```

Implementation

```
<<signalling point state model>>=
state DEBOUNCING() {
    PYCCA_generateDelayedToSelf(Confirm, self->Debounce_time) ;
}
```

CONFIRMING

Activity

```
status = DEV.Read reg(PointID: ID)
if status != Value {
    Value = status
    Eval signal
}
```

Implementation

```
<<signalling point state model>>=
state CONFIRMING() {
    sio_Point_ID ptid = PYCCA_idOfRef(IO_Point,
            PYCCA_unionSupertype(self, IO_Point, R1)) ;
    sio_Point_Value status = ExternalOp(DEVICE_Read_reg)(ptid) ;
    if (status != self->Value) {
        self->Value = status ;
        ThisClassInstOp(evalSignal)(self) ;
    }
}
```

Signalling Poing::evalSignal

Activity

```
if (Value == 1 AND Active high) OR (Value == 0 AND !Active high)
    if Trigger is not .inactive
        NOTIFY.Signal point( PointID: ID, isActive: true )
else
    if Trigger is not .active
        NOTIFY.Signal point(PointID: ID, isActive: false )

DEV.Enable signal(PointID: ID, Trigger )
```

Implementation

```
<<signalling point operations>>=
instance operation
evalSignal()
{
    sio_Point_ID ptid = PYCCA_idOfRef(IO_Point,
            PYCCA_unionSupertype(self, IO_Point, R1)) ;
    if ((self->Value == 1 && self->Active_high) ||
            (self->Value == 0 && !self->Active_high)) {
        if (self->Trigger != Inactive) {
            ExternalOp(NOTIFY_Signal_point)(ptid, true) ;
        }
    } else {
        if (self->Trigger != Active) {
            ExternalOp(NOTIFY_Signal_point)(ptid, false) ;
        }
    }
    ExternalOp(DEVICE_Enable_signal)(ptid) ;
}
```

I/O Point Operations

I/O Point::readPoint

Activity

```
I/O Point::readPoint() : Point Value
[
    self->[R1]Discrete_Point > ~dpt | None? !True !False
    !True: [
        self->[R1]Signalling_Point > ~spt | None? !True !False
        !True: [
            self->[R1]Continuous_Point] > ~cpt
            ~cpt.readPoint(~pid => pid) | return
        ]
        !False: [
            ~spt.readPoint(~pid => pid) | return
        ]
    ]
    !False: [
        ~dpt.readPoint(~pid => pid) | return
    ]
]
```

Implementation

```
<<io point operations>>=
instance operation
readPoint() : (sio_Point_Value)
{
    assert(PYCCA_isSubtypeRelated(self, IO_Point, R1, Discrete_Point) ||
            PYCCA_isSubtypeRelated(self, IO_Point, R1, Continuous_Point)) ;

    switch (self->SubCodeMember(R1)) {
    case SubCodeValue(IO_Point, R1, Discrete_Point):
        return InstOp(Discrete_Point, readPoint)(
                PYCCA_unionSubtype(self, R1, Discrete_Point)) ;

    case SubCodeValue(IO_Point, R1, Signalling_Point):
        return InstOp(Signalling_Point, readPoint)(
                PYCCA_unionSubtype(self, R1, Signalling_Point)) ;

    case SubCodeValue(IO_Point, R1, Continuous_Point):
        return InstOp(Continuous_Point, readPoint)(
                PYCCA_unionSubtype(self, R1, Continuous_Point)) ;

    default:
        return 0 ;
    }
}
```

I/O Point::writePoint

Activity

```
I/O Point::writePoint(
    Point Value value)
[
    self->[R1]Discrete_Point > ~dpt | None? !True !False
    !True: [
        self->[R1]Continuous_Point] > ~cpt
        ~cpt.writePoint(~pid => pid)
    ]
    !False: [
        ~dpt.writePoint(~pid => pid)
    ]
]
```

Implementation

```
<<io point operations>>=
instance operation
writePoint(
    sio_Point_Value value)
{
    assert(PYCCA_isSubtypeRelated(self, IO_Point, R1, Discrete_Point) ||
            PYCCA_isSubtypeRelated(self, IO_Point, R1, Continuous_Point)) ;

    switch (self->SubCodeMember(R1)) {
    case SubCodeValue(IO_Point, R1, Discrete_Point):
        InstOp(Discrete_Point, writePoint)(
            PYCCA_unionSubtype(self, R1, Discrete_Point), value) ;
        break ;

    case SubCodeValue(IO_Point, R1, Continuous_Point):
        InstOp(Continuous_Point, writePoint)(
            PYCCA_unionSubtype(self, R1, Continuous_Point), value) ;
        break ;

    // N.B. no default
    }
}
```

Discrete Point Operations

Discrete Point::readPoint

Activity

```
Discrete Point::readPoint() : Point Value
[
    self->[R8]Packed Point > ~ppt | None? !True !False
    !True: [
        self->[R8]Control Point > ~ctpt
        ~ctpt.readPoint() | return
    ]
    !False: [
        ~ppt.readPoint() | return
    ]
]
```

Implementation

```
<<discrete point operations>>=
instance operation
readPoint() : (sio_Point_Value)
{
    assert(PYCCA_isSubtypeRelated(self, Discrete_Point, R8, Packed_Point) ||
            PYCCA_isSubtypeRelated(self, Discrete_Point, R8, Control_Point)) ;

    switch (self->SubCodeMember(R8)) {
    case SubCodeValue(Discrete_Point, R8, Packed_Point):
        return InstOp(Packed_Point, readPoint)(
                PYCCA_unionSubtype(self, R8, Packed_Point)) ;

    case SubCodeValue(Discrete_Point, R8, Control_Point):
        return InstOp(Control_Point, readPoint)(
                PYCCA_unionSubtype(self, R8, Control_Point)) ;

    default:
        return 0 ;
    }
}
```

Discrete Point::writePoint

Activity

```
Discrete Point::writePoint()
[
    self->[R8]Packed Point > ~ppt | None? !True !False
    !True: [
        self->[R8]Control Point > ~ctpt
        ~ctpt.writePoint()
    ]
    !False: [
        ~ppt.writePoint()
    ]
]
```

Implementation

```
<<discrete point operations>>=
instance operation
writePoint(
    sio_Point_Value value)
{
    assert(PYCCA_isSubtypeRelated(self, Discrete_Point, R8, Packed_Point) ||
            PYCCA_isSubtypeRelated(self, Discrete_Point, R8, Control_Point)) ;

    switch (self->SubCodeMember(R8)) {
    case SubCodeValue(Discrete_Point, R8, Packed_Point):
        InstOp(Packed_Point, writePoint)(PYCCA_unionSubtype(self, R8,
                Packed_Point), value) ;
        break ;

    case SubCodeValue(Discrete_Point, R8, Control_Point):
        InstOp(Control_Point, writePoint)(PYCCA_unionSubtype(self, R8,
                Control_Point), value) ;
        break ;

    // N.B. no default
    }
}
```

Continuous Point Operations

Continuous Point::readPoint

Activity

```
Continuous Point::readPoint() : Point Value
[
    self->[R2]Continuous_Input_Point > ~cip | None? !True !False
    !True: [
        0 | return
    ]
    !False: [
        ~cip.Value | return
    ]
]
```

Implementation

```
<<continuous point operations>>=
instance operation
readPoint() : (sio_Point_Value)
{
    assert(PYCCA_isSubtypeRelated(self, Continuous_Point, R2,
            Continuous_Output_Point) ||
        PYCCA_isSubtypeRelated(self, Continuous_Point, R2,
            Continuous_Input_Point)) ;

    return PYCCA_isSubtypeRelated(self, Continuous_Point, R2,
            Continuous_Input_Point) ?
        PYCCA_unionSubtype(self, R2, Continuous_Input_Point)->Value : 0 ;
}
```

Continuous Point::writePoint

Activity

```
Continuous Point::writePoint(
    Point Value value)
[
    self->[R2]Continuous_Output_Point > ~cop | None? !False
    !False: [
        self->[R3]Point_Scaling > ~ps
        ~ps.scaleValueOut(~value => value) > ~scaled
        DEVICE::Write reg(~cop.PointId => point, ~scaled => value)
    ]
]
```

Implementation

```
<<continuous point operations>>=
instance operation
writePoint(
    sio_Point_Value value)
{
    assert(PYCCA_isSubtypeRelated(self, Continuous_Point, R2,
            Continuous_Output_Point) ||
        PYCCA_isSubtypeRelated(self, Continuous_Point, R2,
            Continuous_Input_Point)) ;

    if (PYCCA_isSubtypeRelated(self, Continuous_Point, R2,
            Continuous_Output_Point)) {
        ClassRefVar(Point_Scaling, ps) = self->R3 ;
        assert(ps != NULL) ;
        sio_Point_ID pid =
            PYCCA_idOfRef(IO_Point, PYCCA_unionSupertype(self, IO_Point, R1)) ;

        sio_Point_Value scaled = InstOp(Point_Scaling, scaleValueOut)(ps, value) ;
        ExternalOp(DEVICE_Write_reg)(pid, scaled) ;
    }
    /*
     * Writes to Continuous Input Points are silently ignored.
     */
}
```

Packed Point Operations

Packed Point::readPoint

Activity

```
Packed Point::readPoint() : Point Value
[
    (1 << self.Length) - 1 > ~mask
    self->[R9]Control_Point > ~ctpt
    ~ctpt.readPoint() > ~value
    (~value >> self.Offset) BITAND ~mask | return
]
```

Implementation

```
<<packed point operations>>=
instance operation
readPoint() : (sio_Point_Value)
{
    sio_Point_Value mask = (1 << self->Length) - 1 ;
    ClassRefVar(Control_Point, ctpt) = self->R9 ;
    assert(ctpt != NULL) ;
    sio_Point_Value value = InstOp(Control_Point, readPoint)(ctpt) ;
    return (value >> self->Offset) & mask ;
}
```

Packed Point::writePoint

Activity

```
Packed Point::writePoint(
    Point Value value)
[
    ((1 << self.Length) - 1) << self.Offset > ~mask
    self->[R9]Control_Point > ~ctpt
    case ~ctpt.Memory_model
    !MEM: [
        ~ctpt.readPoint() BITAND (NOT ~mask) > ~base
    ]
    !RC: [
        0 > ~base
    ]
    !RCZ: [
        NOT 0 > ~base
    ]
    ((~value << self.Offset) BITAND ~mask) BITOR ~base > ~new
    ~ctpt.writePoint(~new => value)
]
```

Implementation

```
<<packed point operations>>=
instance operation
writePoint(
    sio_Point_Value value)
{
    sio_Point_Value mask = ((1 << self->Length) - 1) << self->Offset ;
    ClassRefVar(Control_Point, ctpt) = self->R9 ;
    assert(ctpt != NULL) ;
    sio_Point_Value base ;
    switch (ctpt->Memory_model) {
    case Memory:
    default:
        base = InstOp(Control_Point, readPoint)(ctpt) & ~mask ;
        break ;

    case ReadClear:
        base = 0 ;
        break ;

    case ReadClearZero:
        base = ~0 ;
        break ;
    }

    InstOp(Control_Point, writePoint)(ctpt,
            ((value << self->Offset) & mask) | base) ;
}
```

Control Point Operations

Control Point::readPoint

Activity

```
Control Point::readPoint() : Point Value
[
    self.Access != WriteOnly ? !True
    !True: [
        DEVICE::Read reg(self.PointId => point) | return
    ]
]
```

Implementation

```
<<control point operations>>=
instance operation
readPoint() : (sio_Point_Value)
{
    sio_Point_Value value = 0 ;
    if (self->Access != WriteOnly) {
        sio_Point_ID pid =
            PYCCA_idOfRef(IO_Point,
            PYCCA_unionSupertype(PYCCA_unionSupertype(self, Discrete_Point, R8),
                IO_Point, R1)) ;
        value = ExternalOp(DEVICE_Read_reg)(pid) ;
    }

    return value ;
}
```

Control Point::writePoint

Activity

```
Control Point::writePoint(
    Point Value value)
```

```
[
    self.Access != ReadOnly ? !True
    !True: [
        DEVICE::Write reg(self.PointId => point, ~value => value)
    ]
]
```

Implementation

```
<<control point operations>>=
instance operation
writePoint(
    sio_Point_Value value)
{
    if (self->Access != ReadOnly) {
        sio_Point_ID pid =
            PYCCA_idOfRef(IO_Point,
            PYCCA_unionSupertype(PYCCA_unionSupertype(self, Discrete_Point, R8),
                IO_Point, R1)) ;
        ExternalOp(DEVICE_Write_reg)(pid, value) ;
    }
}
```

Signalling Point Operations

Signalling Point::readPoint

Activity

```
Signalling Point::readPoint() : Point Value
[
    DEVICE::Read reg(self.PointId => point) | return
]
```

Implementation

```
<<signalling point operations>>=
instance operation
readPoint() : (sio_Point_Value)
{
    sio_Point_ID pid =
        PYCCA_idOfRef(IO_Point, PYCCA_unionSupertype(self, IO_Point, R1)) ;
    return ExternalOp(DEVICE_Read_reg)(pid) ;
}
```

Continuous Input Point Operations

Continuous_Input_Point::updateValue

Activity

```
Continuous_Input_Point::updateValue(
    Point Value value)
[
    self->[R2->R3]Point_Scaling > ~ps
    ~ps.scaleValueIn(value => value) > self.Value
    # Notify clients of new point values
    NOTIFY::newPointValue(ID => self.ID, Value => value)
    # Check the point thresholds against the new value
    self->[R7]Range Limitation | generate New_point(self.Value)
]
```

Implementation

```
<<continuous input point operations>>=
instance operation
updateValue(
    sio_Point_Value value)
{
    ClassRefVar(Continuous_Point, cpt) = PYCCA_unionSupertype(self, Continuous_Point, R2) ;
    ClassRefVar(Point_Scaling, ps) = cpt->R3 ;
    assert(ps != NULL) ;
    self->Value = InstOp(Point_Scaling, scaleValueIn)(ps, value) ;

    ClassRefVar(IO_Point, iop) = PYCCA_unionSupertype(cpt, IO_Point, R1) ;
    ExternalOp(NOTIFY_New_point_value)(
        PYCCA_idOfRef(IO_Point, iop), self->Value) ;

    ClassRefConstSetVar(Range_Limitation, tdset) ;
    PYCCA_forAllRelated(tdset, self, R7) {
        ClassRefVar(Range_Limitation, td) = *tdset ;
        MechEcb np = PYCCA_newEvent(New_point, Range_Limitation, td, self) ;
        PYCCA_eventParam(np, Range_Limitation, New_point, pointValue) = value ;
        PYCCA_postEvent(np) ;
    }
}
```

Point Scaling Operations

Point Scaling::scaleValueIn

Activity

```
Point Scaling::scaleValueIn(
    Point Value value) : PointValue
{
    ~value * self.Multiplier / self->Divisor +
        self.Intercept | MIN(self.Mask) | return
}
```

Implementation

```
<<point scaling operations>>=
instance operation
scaleValueIn(
    sio_Point_Value value) : (sio_Point_Value)
{
    /*
     * Convert, rounding to deal with division truncation.
     */
    sio_Point_Value in = ((value * self->Multiplier + self->Divisor / 2) /
            self->Divisor) + self->Intercept ;
    if (in > self->Mask) {
        in = self->Mask ;
    }
    return in ;
}
```

Point Scaling::scaleValueOut

Activity

```
Point Scaling::scaleValueIn(
    Point Value value) : PointValue
{
    MAX(~value, self.Intercept) - self.Intercept |
        ~ * self.Multiplier / self->Divisor |
        MIN(self.Mask) | return
}
```

Implementation

```
<<point scaling operations>>=
instance operation
scaleValueOut(
    sio_Point_Value value) : (sio_Point_Value)
{
    if (value < self->Intercept) {
        value = self->Intercept ;
    }
    /*
     * Convert, rounding to deal with division truncation.
     */
    sio_Point_Value out = (((value - self->Intercept) * self->Multiplier) +
            self->Divisor / 2) / self->Divisor ;
    if (out > self->Mask) {
        out = self->Mask ;
    }
    return out ;
}
```

Domain Operations

Read point

Domain clients use the `Read_point` operation to obtain the value for a point. Point values are returned in engineering units.

Activity

```
Read_point(
    Point_ID pid) : Point Value
[
    IO_Point(one, PointId = pid) > ~iop
    ~iop.readPoint() | return
]
```

Implementation

```
<<domain operations>>=
domain operation
Read_point(
    sio_Point_ID pid) : (sio_Point_Value)
{
    PYCCA_checkId(IO_Point, pid) ;
    ClassRefVar(IO_Point, iop) = PYCCA_refOfId(IO_Point, pid) ;
    return InstOp(IO_Point, readPoint)(iop) ;
}
```

Write point

Domain clients use the `Write_point` operation to set the value of an output point. Values are assumed to be in engineering units and are scaled as necessary before output.

Activity

```
Write_point(
    PointI_D pid,
    Point Value value)
[
    IO_Point(one, PointId = pid) > ~iop
    ~iop.writePoint(value) | return
]
```

Implementation

```
<<domain operations>>=
domain operation
Write_point(
    sio_Point_ID pid,
    sio_Point_Value value)
{
    PYCCA_checkId(IO_Point, pid) ;
    ClassRefVar(IO_Point, iop) = PYCCA_refOfId(IO_Point, pid) ;
    InstOp(IO_Point, writePoint)(iop, value) ;
}
```

Init

The Signal I/O domain supports the notion of turning on and off a Signalling Point. However, the Lube domain does not use that notion for its Machinery lockout or Reservoir fluid levels. So will turn on all the Signalling Points at initialization time.

Implementation

```
<<domain operations>>=
domain operation
init()
{
    ClassRefVar(IO_Point, iopt) ;
    PYCCA_forAllInst(iopt, IO_Point) {
        if (iopt->SubCodeMember(R1) ==
                SubCodeValue(IO_Point, R1, Signalling_Point)) {
            ClassRefVar(Signalling_Point, spt) =
                    PYCCA_unionSubtype(iopt, R1, Signalling_Point) ;
            PYCCA_generate(On, Signalling_Point, spt, NULL) ;
        }
    }
}
```

External Entities

Notify Entity

The SIO domain assumes the existance of an entity that can be used to notify domain clients that a new value for a Continuous Input Point is available.

NOTIFY::New point value

The `New point value` operation is called when the sampling for a Sample Group is complete. Domain clients are notified of the change in point value.

Interface

```
NOTIFY::New point value(Point_ID point, Point Value value)
```

Implementation

```
<<external operations>>=
external operation
NOTIFY_New_point_value(
    sio_Point_ID point,
    sio_Point_Value value)
{
}
```

NOTIFY::In range

Interface

```
NOTIFY::In range(Point_ID point, Threshold ID threshold)
```

Implementation

```
<<external operations>>=
external operation
NOTIFY_In_range(
    sio_Point_ID point,
    sio_Threshold_ID threshold)
{
}
```

NOTIFY::Out of range

Interface

```
NOTIFY::Out of range(Point_ID point, Threshold ID threshold)
```

Implementation

```
<<external operations>>=
external operation
NOTIFY_Out_of_range(
    sio_Point_ID point,
    sio_Threshold_ID threshold)
{
}
```

NOTIFY::Signal point

Interface

```
NOTIFY::Signal point(Point_ID point, bool isActive)
```

Implementation

```
<<external operations>>=
external operation
NOTIFY_Signal_point(
    sio_Point_ID point,
    bool isActive)
{
}
```

Device Entity

The SIO assumes the existance of an entity that can be used to read and write hardware values to registers.

DEVICE::Convert group

The `Convert group` operation is invoked to start the conversion of the input points in a sample group.

Interface

```
DEVICE::Convert group(ConverterIdType converter, GroupIdType group)
```

Implementation

```
<<external operations>>=
external operation
DEVICE_Convert_group(
    sio_Converter_ID converter,
    sio_Group_ID group)
{
}
```

To drive the processing along, we will stub the `DEVICE_Convert_group` operation to send the **Conversion Done** event after a short delay.

```
<<external operation stubs>>=
void
eop_sio_DEVICE_Convert_group(
    sio_Converter_ID converter,
    sio_Group_ID group)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain sio eop DEVICE_Convert_group "
            "parameters {converter %" PRIu8 " group %" PRIu8 "}",
            converter, group) ;
#       endif /* TACK */

    assert(converter < SIO_SIGNAL_CONVERTER_INST_COUNT) ;
    int pcode = pycca_generate_delayed_event(&sio_portal,
        SIO_SIGNAL_CONVERTER_CLASS_ID,
        converter,
        NormalEvent,
        SIO_SIGNAL_CONVERTER_CONVERSION_DONE_EVENT_ID,
        NULL,
        10) ;
    assert(pcode == 0) ;
    (void)pcode ;
}
```

DEVICE::Read converted value

The `Read converted value` operation is invoked to obtain the input point values after a conversion has completed.

Interface

```
DEVICE::Read converted value(ConverterIdType converter, Point ID point) : Point Value
```

Implementation

```
<<external operations>>=
external operation
DEVICE_Read_converted_value(
    sio_Converter_ID converter,
    sio_Point_ID point) : (sio_Point_Value)
{
}
```

For the purposes of testing and simulation, we need to have somewhat meaningful values returned from the signal conversion. For intended purpose of linking up with the Lube domain, the values we need here are for injector pressure. We'll return values from a stored set of values placed in the code. Alternatively, more extensive testing could use a file of stored values and `Read_converted_value` could return values from the file.

```
<<external operation stubs>>=
typedef struct {
    int valueIndex ;
    int numValues ;
    sio_Point_Value *values ;
} ValueStore ;
```

The `valueIndex` is used to keep our place in the array of `values`. We will iterate through the values at `valueIndex` modulo `numValues`.

We will need a separate store of values for each Signal Converter and for each Continuous Input Point attached to the Signal Converter.

```
<<external operation stubs>>=
static sio_Point_Value
vs_nextValue(
    ValueStore *vs)
{
    if (vs->numValues <= 0) {
        return 0 ;
    }

    sio_Point_Value cval = vs->values[vs->valueIndex] ;
    if (++vs->valueIndex >= vs->numValues) {
        vs->valueIndex = 0 ;
    }

    return cval ;
}
```

```
<<external operation stubs>>=
static sio_Point_Value inj1PresValues[] = {20, 21, 22, 26, 27, 16, 15, 14} ;
static sio_Point_Value inj2PresValues[] = {20, 21, 22, 26, 27, 23, 19, 18} ;
static sio_Point_Value inj3PresValues[] = {20, 21, 22, 26, 27, 23, 19, 18} ;

static ValueStore converterValues
        [SIO_SIGNAL_CONVERTER_INST_COUNT]
        [SIO_CONTINUOUS_INPUT_POINT_INST_COUNT] = {
    [SIO_SIGNAL_CONVERTER_CVT1_INST_ID] = {
        [SIO_CONTINUOUS_INPUT_POINT_IOP1_INST_ID] = {
            .valueIndex = 0,
            .numValues = COUNTOF(inj1PresValues),
            .values = inj1PresValues
        },
        [SIO_CONTINUOUS_INPUT_POINT_IOP2_INST_ID] = {
            .valueIndex = 0,
            .numValues = COUNTOF(inj2PresValues),
            .values = inj2PresValues
        },
        [SIO_CONTINUOUS_INPUT_POINT_IOP3_INST_ID] = {
            .valueIndex = 0,
            .numValues = COUNTOF(inj2PresValues),
            .values = inj3PresValues
        },
    },
} ;
```

Stub

```
<<external operation stubs>>=
sio_Point_Value
eop_sio_DEVICE_Read_converted_value(
    sio_Converter_ID converter,
    sio_Point_ID point)
{
#       ifdef TACK
    harness_stub_printf("stub", "domain sio eop DEVICE_Read_converted_value "
            "parameters {converter %" PRIu8 " point %" PRIu8 "}",
            converter, point) ;
#       endif /* TACK */

    assert(converter <= SIO_SIGNAL_CONVERTER_INST_COUNT) ;
    assert(point < SIO_IO_POINT_INST_COUNT) ;

    /*
     * Offset the point id to reorg to the beginning of Continuous Input
     * Point numbering.
     */
    sio_Point_ID cip = point - SIO_IO_POINT_IOP1_INST_ID ;

    return vs_nextValue(&converterValues[converter][cip]) ;
}
```

DEVICE::Write reg

The `Write reg` operation is invoked when this domain needs to set the current value of a hardware register associated with some I/O Point.

Interface

```
DEVICE::Write reg(Point ID pid, Point Value value)
```

Implementation

```
<<external operations>>=
external operation
DEVICE_Write_reg(
    sio_Point_ID pid,
    sio_Point_Value value)
{
}
```

DEVICE::Read reg

The `Read reg` operation is invoked when this domain needs to obtain the current value of a hardware register associated with some I/O Point.

Interface

```
DEVICE::Read reg(Point ID pid) : Point Value
```

Implementation

```
<<external operations>>=
external operation
DEVICE_Read_reg(
    sio_Point_ID pid) : (sio_Point_Value)
{
    return 0 ;
}
```

Stub

```
<<external operation stubs>>=
static sio_Point_Value m1LockoutValues[] = {0, 1} ;
static sio_Point_Value m2LockoutValues[] = {0, 1} ;
static sio_Point_Value m3LockoutValues[] = {0, 1} ;
static sio_Point_Value rsv1LevelValues[] = {1, 0} ;
static sio_Point_Value rsv2LevelValues[] = {1, 0} ;

static ValueStore registerValues[SIO_IO_POINT_INST_COUNT] = {
    [SIO_IO_POINT_IOP7_INST_ID] = {
        .valueIndex = 0,
        .numValues = COUNTOF(m1LockoutValues),
        .values = m1LockoutValues
    },
    [SIO_IO_POINT_IOP8_INST_ID] = {
        .valueIndex = 0,
        .numValues = COUNTOF(m2LockoutValues),
        .values = m2LockoutValues
    },
    [SIO_IO_POINT_IOP9_INST_ID] = {
        .valueIndex = 0,
        .numValues = COUNTOF(m3LockoutValues),
        .values = m3LockoutValues
    },
    [SIO_IO_POINT_IOP10_INST_ID] = {
        .valueIndex = 0,
        .numValues = COUNTOF(rsv1LevelValues),
        .values = rsv1LevelValues
    },
    [SIO_IO_POINT_IOP11_INST_ID] = {
        .valueIndex = 0,
        .numValues = COUNTOF(rsv2LevelValues),
        .values = rsv2LevelValues
    },
} ;
```

```
<<external operation stubs>>=
sio_Point_Value
eop_sio_DEVICE_Read_reg(
    sio_Point_ID pid)
{
#       ifdef TACK
```

```
    harness_stub_printf("stub", "domain sio eop DEVICE_Read_reg "
            "parameters {pid %" PRIu8 "}", pid) ;
#       endif /* TACK */

    assert(pid < SIO_IO_POINT_INST_COUNT) ;

    return vs_nextValue(&registerValues[pid]) ;
}
```

DEVICE::Enable signal

Interface

```
DEVICE::Enable signal(Point_ID point)
```

Implementation

```
<<external operations>>=
external operation
DEVICE_Enable_signal(
    sio_Point_ID pid)
{
}
```

DEVICE::Disable signal

Interface

```
DEVICE::Disable signal(Point_ID point)
```

Implementation

```
<<external operations>>=
external operation
DEVICE_Disable_signal(
    sio_Point_ID pid)
{
}
```

Initial Instance Population

Note
We are including the example instance population directly in the domain workbook. This is convenient for the case of this example, but generally, instance populations should be kept separate from the domain itself. During actual development, many populations will be used for testing and deployment and keeping them separate makes that task easier. Also, if the domain should be reused, separating out the population from the domain makes reuse easier.

We will populate the model to match the needs of the Lubrication Domain to which SIO provides services. The Lubrication Domain classes that are related to SIO services are:

Machinery
There are three instances of Machinery and each has a single sensor indicating the lockout status. We will use three instances of Signalling Point for the lockout sensors.

Reservoir
: There are two instance of Reservoir and each has a single sensor indicating lube level. We will use two instances of Signalling Point for the lube level sensors.

Injector
: There are three instances of Injector. Each injector has a pressure sensor so we will use three Continuous Input Point instances to measure the pressure. Each point will be in a separate Conversion Group and we will assume that there is only a single Signal Converter in the system. The Injectors need to know if the pressure is above the dissipation pressure and if the pressure is above or below the inject pressure. So we must configure the necessary instances of Point Threshold and Range Limitation to monitor the limits on injector pressure. The Injector also has an actuator to control the delivery of lubricant, so we will need three Discrete Points to be able to control the lubricant injection.

I/O Point Population

Three pieces of machinery, two reservoirs and three injectors gives us eleven I/O Points total.

Table 6: I/O Point Population

ID
iop1
iop2
iop3
iop4
iop5
iop6
iop7
iop8
iop9
iop10
iop11

```
<<io point population>>=
table IO_Point
    R1
@iop1
    -> Continuous_Point.iop1
@iop2
    -> Continuous_Point.iop2
@iop3
    -> Continuous_Point.iop3
@iop4
    -> Discrete_Point.iop4
@iop5
    -> Discrete_Point.iop5
@iop6
    -> Discrete_Point.iop6
@iop7
    -> Signalling_Point.iop7
@iop8
    -> Signalling_Point.iop8
@iop9
    -> Signalling_Point.iop9
@iop10
    -> Signalling_Point.iop10
@iop11
```

```
    -> Signalling_Point.iop11
end
```

Discrete Point Population

Table 7: Discrete Point Population

ID
iop4
iop5
iop6

```
<<discrete point population>>=
table Discrete_Point
    R8
@iop4
    -> Control_Point.iop4
@iop5
    -> Control_Point.iop5
@iop6
    -> Control_Point.iop6
end
```

Control Point Population

Table 8: Control Point Population

ID	Memory model	Access
iop4	Memory	ReadWrite
iop5	Memory	ReadWrite
iop6	Memory	ReadWrite

```
<<control point population>>=
table Control_Point
    (Memory_Type Memory_model)
    (Access_Type Access)
@iop4
    {Memory} {ReadWrite}
@iop5
    {Memory} {ReadWrite}
@iop6
    {Memory} {ReadWrite}
end
```

Signalling Point Population

Table 9: Signalling Point Population

ID	Trigger	Active high	Debounce time
iop7	BothActive	true	500
iop8	BothActive	true	500
iop9	BothActive	true	500
iop10	BothActive	true	2000
iop11	BothActive	true	2000

```
<<signalling point population>>=
table Signalling_Point
    (Edge_Type Trigger)
    (bool Active_high)
    (Msec Debounce_time)
@iop7
    {BothActive}    {true}      {500}
@iop8
    {BothActive}    {true}      {500}
@iop9
    {BothActive}    {true}      {500}
@iop10
    {BothActive}    {true}      {2000}
@iop11
    {BothActive}    {true}      {2000}
end
```

Continuous Point Population

Table 10: Continuous Point Population

ID	Scaling
iop1	ix77b_pres_scale
iop2	ihn4_pres_scale
iop3	ix77b_pres_scale

```
<<continuous point population>>=
table Continuous_Point
    R2
    R3
@iop1
    -> Continuous_Input_Point.iop1
    -> ix77b_pres_scale
@iop2
    -> Continuous_Input_Point.iop2
    -> ihn4_pres_scale
@iop3
    -> Continuous_Input_Point.iop3
    -> ix77b_pres_scale
end
```

Point Scaling Population

Since there are two different injector designs being used, we will assume the pressure transducer for each has a different scaling from device units to engineering units. However, without any additional hardware information about the scaling, we use the identity scaling for the values. Once real hardware scaling is known, then it needs to be substituted here.

Table 11: Point Scaling Population

ID	Multiplier	Divisor	Intercept	Mask
ihn4_pres_scale	1	1	0	~0
ix774_pres_scale	1	1	0	~0

```
<<point scaling population>>=
table Point_Scaling
    (sio_Point_Value Multiplier)
    (sio_Point_Value Divisor)
    (sio_Point_Value Intercept)
    (Point_Value_Mask Mask)
@ihn4_pres_scale
    {1} {1} {0} {~0}
@ix77b_pres_scale
    {1} {1} {0} {~0}
end
```

Continuous Input Point Population

Table 12: Continuous Input Point Population

ID	Value	Group
iop1	0	inj1_cg
iop2	0	inj2_cg
iop3	0	ing3_cg

```
<<continuous input point population>>=
table Continuous_Input_Point
    R4
    R7
@iop1
    -> inj1_cg
    ->>
        above_disp_iop1
        above_inj_iop1
        max_pres_iop1
    end
@iop2
    -> inj2_cg
    ->>
        above_disp_iop2
        above_inj_iop2
        max_pres_iop2
    end
```

```
@iop3
    -> inj3_cg
    ->>
        above_disp_iop3
        above_inj_iop3
        max_pres_iop3
    end
end
```

Point Threshold Population

Table 13: Point Threshold Population

ID	Limit	Direction	Over limit	Under limit
ix77b_above_disp	26	Rising	2	2
ix77b_above_inj	15	Rising	2	2
ix77b_max_pres	26	Rising	1	2
ihn4_above_disp	32	Rising	2	2
ihn4_above_inj	19	Rising	2	2
ihn4_max_pres	35	Rising	1	2

```
<<point threshold population>>=
table Point_Threshold
    (sio_Point_Value Limit)
    (Excursion_Direction Direction)
    (Count Over_limit)
    (Count Under_limit)
# IX77B design
@ix77b_above_disp
    {26}    {Rising}    {2}     {2}
@ix77b_above_inj
    {15}    {Rising}    {2}     {2}
@ix77b_max_pres
    {26}    {Rising}    {1}     {2}

# IHN4 design
@ihn4_above_disp
    {32}    {Rising}    {2}     {2}
@ihn4_above_inj
    {19}    {Rising}    {2}     {2}
@ihn4_max_pres
    {35}    {Rising}    {1}     {2}
end
```

Range Limitation Population

Table 14: Range Limitation Population

Point	Threshold	Over count	Under count
iop1	ix77b_above_disp	0	0
iop1	ix77b_above_inj	0	0
iop1	ix77b_max_pres	0	0

Table 14: (continued)

Point	Threshold	Over count	Under count
iop2	ihn4_above_disp	0	0
iop2	ihn4_above_inj	0	0
iop2	ihn4_max_pres	0	0
iop3	ix77b_above_disp	0	0
iop3	ix77b_above_inj	0	0
iop3	ix77b_max_pres	0	0

```
<<range limitation population>>=
table Range_Limitation
    R7_PT
    R7_CIP
@above_disp_iop1
    -> ix77b_above_disp
    -> iop1
@above_inj_iop1
    -> ix77b_above_inj
    -> iop1
@max_pres_iop1
    -> ix77b_max_pres
    -> iop1
@above_disp_iop2
    -> ihn4_above_disp
    -> iop2
@above_inj_iop2
    -> ihn4_above_inj
    -> iop2
@max_pres_iop2
    -> ihn4_max_pres
    -> iop2
@above_disp_iop3
    -> ix77b_above_disp
    -> iop3
@above_inj_iop3
    -> ix77b_above_inj
    -> iop3
@max_pres_iop3
    -> ix77b_max_pres
    -> iop3
end
```

Conversion Group Population

Table 15: Conversion Group Population

ID	Waiting for converter	Period	Converter
inj1_cg	false	500	cvt1
inj2_cg	false	500	cvt1
inj3_cg	false	500	cvt1

```
<<conversion group population>>=
table Conversion_Group
    (Msec Period)
    R4
    R5
@inj1_cg
    {500}   ->> iop1 end   -> cvt1
@inj2_cg
    {500}   ->> iop2 end   -> cvt1
@inj3_cg
    {500}   ->> iop3 end   -> cvt1
end
```

Signal Converter Population

Table 16: Signal Converter Population

ID	Converter available
cvt1	true

```
<<signal converter population>>=
instance Signal_Converter@cvt1
    R5 ->>
        inj1_cg
        inj2_cg
        inj3_cg
    end
    assigner -> r6asgn1
end
```

R6 Assigner Population

```
<<R6 assigner population>>=
instance R6_Assigner@r6asgn1
    idclass -> cvt1
end
```

Code Layout

The order of components in a `pycca` file is somewhat arbitrary. The only order imposed by `pycca` itself is that class definitions must precede class populations. The generated “C” file is reordered by `pycca` to meet the needs of the compiler. Generally this means that definitions appear before their use and thus inverts the more natural order of code presentation. This is only significant because it is usually the “C” file that is viewed in a debugger.

In literate programming terminology, a *chunk* is a named part of the final program. The program chunks form a tree and the root of that tree is named *by default. Here we follow the convention of naming the root the same as the output file name. The process of extracting the program tree formed by the chunks is called* tangle*. By the default the program, *atangle*, extracts the root chunk to produce the `pycca` source file.

Root Chunk

```
<<sio.pycca>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS GENERATED FROM THE SOURCE OF A LITERATE PROGRAM.
# YOU MUST EDIT THE ORIGINAL SOURCE TO MODIFY THIS FILE.
#*++
# Copyright 2017 by Leon Starr, Andrew Mangogna and Stephen Mellor
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#
# Project:
#   Models to Code Book
#
# Module:
#   Signal I/O Domain pycca file
#*--

domain sio
    interface prolog {
    <<interface prolog>>
    <<external data types>>
    }
    interface epilog {
    <<interface epilog>>
    }
    <<domain operations>>
    <<external operations>>
    <<classes>>
    <<population>>
    implementation prolog {
    #include <assert.h>
    #include "sio.h"
    <<implementation prolog>>
    <<internal data types>>
    }
    implementation epilog {
    <<implementation epilog>>
    }
end
```

```
<<sio_eop.c>>=
/*
# DO NOT EDIT THIS FILE!
# THIS FILE IS GENERATED FROM THE SOURCE OF A LITERATE PROGRAM.
# YOU MUST EDIT THE ORIGINAL SOURCE TO MODIFY THIS FILE.
#*++
# Copyright 2017 by Leon Starr, Andrew Mangogna and Stephen Mellor
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#
# Project:
#   Models to Code Book
#
# Module:
#   Signal I/O External Operations Stubs
#*--
*/

#include <stdbool.h>
#include <inttypes.h>
#include <stddef.h>
#include <assert.h>
#include "pycca_portal.h"
#include "sio.h"
#ifdef TACK
#   include "harness.h"
#endif /* TACK */

#ifndef COUNTOF
#define COUNTOF(a)  (sizeof(a) / sizeof(a[0]))
#endif /* COUNTOF */

<<external operation stubs>>
```

Class Chunks

For each of the classes in the domain, we define how the chunks are composed into the class specification. We have followed a naming convention that defines a chunk for attributes, references and state model for each class. Below we compose the components of the class definition into `pycca` syntax.

```
<<classes>>=
<<io point class>>
<<discrete point class>>
<<signalling point class>>
<<continuous point class>>
<<packed point class>>
<<control point class>>
<<continuous output point class>>
<<continuous input point class>>
<<point scaling class>>
```

```
<<point threshold class>>
<<range limitation class>>
<<signal converter class>>
<<conversion group class>>
<<R6 assigner>>
```

I/O Point Pycca Class

```
<<io point class>>=
class IO_Point
    population static
    <<io point references>>
    <<io point operations>>
end
```

Discrete Point Pycca Class

```
<<discrete point class>>=
class Discrete_Point
    population static
    <<discrete point references>>
    <<discrete point operations>>
end
```

Signalling Point Pycca Class

```
<<signalling point class>>=
class Signalling_Point
    population static
    <<signalling point attributes>>
    <<signalling point operations>>
    machine
        <<signalling point state model>>
    end
end
```

Continuous Point Pycca Class

```
<<continuous point class>>=
class Continuous_Point
    population static
    <<continuous point references>>
    <<continuous point operations>>
end
```

Packed Point Pycca Class

```
<<packed point class>>=
class Packed_Point
    population static
    <<packed point attributes>>
    <<packed point references>>
    <<packed point operations>>
end
```

Control Point Pycca Class

```
<<control point class>>=
class Control_Point
    population static
    <<control point attributes>>
    <<control point references>>
    <<control point operations>>
end
```

Continuous Output Point Pycca Class

```
<<continuous output point class>>=
class Continuous_Output_Point
    population static
    <<continuous output point references>>
    <<continuous output point operations>>
end
```

Continuous Input Point Pycca Class

```
<<continuous input point class>>=
class Continuous_Input_Point
    population static
    <<continuous input point attributes>>
    <<continuous input point references>>
    <<continuous input point operations>>
end
```

Point Scaling Pycca Class

```
<<point scaling class>>=
class Point_Scaling
    population static
    <<point scaling attributes>>
    <<point scaling references>>
    <<point scaling operations>>
end
```

Point Threshold Pycca Class

```
<<point threshold class>>=
class Point_Threshold
    population static
    <<point threshold attributes>>
    <<point threshold references>>
    <<point threshold operations>>
end
```

Range Limitation Pycca Class

```
<<range limitation class>>=
class Range_Limitation
    population static
    <<range limitation attributes>>
    <<range limitation references>>
    <<range limitation operations>>
    machine
        <<range limitation state model>>
    end
end
```

Signal Converter Pycca Class

```
<<signal converter class>>=
class Signal_Converter
    population static
    <<signal converter attributes>>
    <<signal converter references>>
    reference assigner -> R6_Assigner
    machine
        <<signal converter state model>>
    end
end
```

Conversion Group Pycca Class

```
<<conversion group class>>=
class Conversion_Group
    population static
    <<conversion group attributes>>
    <<conversion group references>>
    machine
        <<conversion group state model>>
    end
end
```

R6 Assigner

```
<<R6 assigner>>=
class R6_Assigner
    reference idclass -> Signal_Converter
    machine
        <<R6 assigner state model>>
    end
end
```

Population Chunks

```
<<population>>=
<<io point population>>
<<discrete point population>>
<<signalling point population>>
<<continuous point population>>
<<point scaling population>>
<<packed point population>>
<<control point population>>
<<continuous output point population>>
<<continuous input point population>>
<<point threshold population>>
<<range limitation population>>
<<conversion group population>>
<<signal converter population>>
<<R6 assigner population>>
```

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the Tcl interpreter. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the Tcl code in an order suitable for the Tcl interpreter. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to further processing.

Index