**Architecture and Design**

08

**Fall**

Abhishek Shiroor

Albert Pham

John Lee

Mari Masuda

Project 4 - Technical Report

Team Facade

Table of Contents

Executive Summary 4

Project Background 4

Sensor Visualization 4

Virtual Sensor Editor 4

System Enhancements 4

Dashboard Visualization 5

Control Flows 5

Taverna Integration 5

Geo Fencing 5

Architecture and Design 5

High Level Architecture of Sensor Data Service Platform Application 5

System Functional View 7

Strategy Pattern: Virtual sensor serialization to XML 9

Design Pattern for Control Flow (Decision) 9

Architecture Tradeoff Analysis Method (ATAM) Analysis 10

Project Background 10

Architectural Approaches 10

ATAM Utility Tree 11

Scenario Generation and Prioritization 13

VSE Enhancements 14

Sensor dashboard visualization 14

Control flows 14

Taverna Integration 16

Why XML? 16

What is Taverna? 16

JSON to XML 17

XML to Taverna (.t2flow) 17

Importing to Taverna 23

Implementing FUNCTION Functionality 24

Geofencing 34

GPS Coordinates 37

Source Code 40

Future Work 40

Taverna Plugin for running JavaScript in Java using Rhino 40

Unit Testing with Jasmine 41

Integration with rest of Sensor Data Platform 41

Figure 1: High Level Architecture of Sensor Visualization Platform 6

Figure 2: System Functional View 8

Figure 3: Strategy Pattern (Data & Object Factory, LLC. 2013) 9

Figure 4: Observer Pattern (Hong 2013) 10

Figure 5: Sensor Monitor Dashboard 14

Figure 6: Decision Control Flow in Sensor Editor – Status Monitor True 15

Figure 7: Decision Control Flow in Sensor Editor – Status Monitor False 16

Figure 8: Sample Virtual Sensor 18

Figure 9: Taverna Workbench 23

Figure 10: Taverna Function 24

Figure 11: Ray Casting Algorithm 35

Figure 12: Jordan Curve Theorem 37

Figure 13: CMR1 Room 110 and Room 118 GPS Coordinates 38

Figure 14: Algorithm Results 39

Figure 15: Test Simple Polygon 40

Table 1: ATAM Utility Tree 12

Table 2: Scenario Generation and Prioritization 13

# Executive Summary

The purpose of this document is to provide detailed technical information that can aid the continued development of the Virtual Sensor Editor and Visualization Dashboard.

The Virtual Sensor Editor is a web application that can be used to manage virtual sensors. Virtual sensors are logical combinations of data from multiple physical sensors that are part of the Sensor Data Platform. The Visualization Dashboard is a web application that provides topological, geographical and graphical views of the data from the Sensor Data Platform. As part of Project 4 Team Facade enhanced the editor with features such as control flows that can be used to create complex workflow based virtual sensors. Team Facade also implemented modifications to the Visualization Dashboard so it could be compatible with a new backend system.

As part of Project 4 Team Facade developed basic integration with Taverna – An open source and domain independent workflow management system. Team Facade also developed a location awareness system by implementing Geo Fencing algorithms.

This document provide detailed explanation of the features developed, their design and architecture as well as next steps for future development of the application.

# Project Background

Team Facade was focused on two aspects of the Sensor Data Service Platform — sensor visualization and the virtual sensor editor.

# Sensor Visualization

The sensor dashboard graphically displays the status of all of the sensors via color-coding. It consists of two views—a topographical view and a geographical view. The topological view shows the logical layout of the sensors and the geographical view shows the location of the sensors on a map.

# Virtual Sensor Editor

The virtual sensor editor (VSE) is a GUI-based tool that allows you to create virtual sensor output from physical sensor input. For example, you can create a virtual sensor that displays the real-time average temperature of all of the rooms on the first floor of Carnegie Mellon Silicon Valley’s (CMUSV) building 23. This can be accomplished by dragging representations of the physical sensors onto a canvas and then drawing connections between them and a function box. The function box contains a JavaScript expression that is evaluated using the connected physical sensors’ data as arguments. The resulting value can either be connected to further function boxes for further processing or saved as a virtual sensor.

# System Enhancements

Team Facade was tasked with enhancing the dashboard and VSE by adding new features as described in the following sections.

## Dashboard Visualization

When the course started, the Sensor Data Service Platform was in a state of transition to new hardware and software. The old API was hosted on Heroku, the new one on einstein.sv.cmu.edu. At the time Team Facade forked the code from the gh-pages branch of the main repo on GitHub, the dashboard code was calling the Heroku API, which was no longer active. Team Facade started by updating the code to call the Einstein API. Next, the team worked to color code the sensors so that working sensors are displayed in blue, and sensors that are not working are displayed in red. Finally, the team fixed the data graph so that it displays historical data for a given sensor.

## Control Flows

Team Facade implemented a true/false decision control flow object within the VSE to make it possible to make a decision based on the value of a physical or virtual sensor.

## Taverna Integration

Team Facade implemented the ability for the VSE to export a virtual sensor to XML for saving and reloading in Taverna (Taverna Workflow Management System 2009), an open-source workflow management program.

## Geo Fencing

In the future, the Sensor Data Service Platform will be able to make use of mobile devices as ad hoc sensors. To make this work, it is necessary to know when a device is within a room so that its data can be associated with the correct location. Team Facade implemented two algorithms to detect when a point is within a polygon so that future teams can determine when a mobile device is within a given geofence.

# Architecture and Design

## High Level Architecture of Sensor Data Service Platform Application

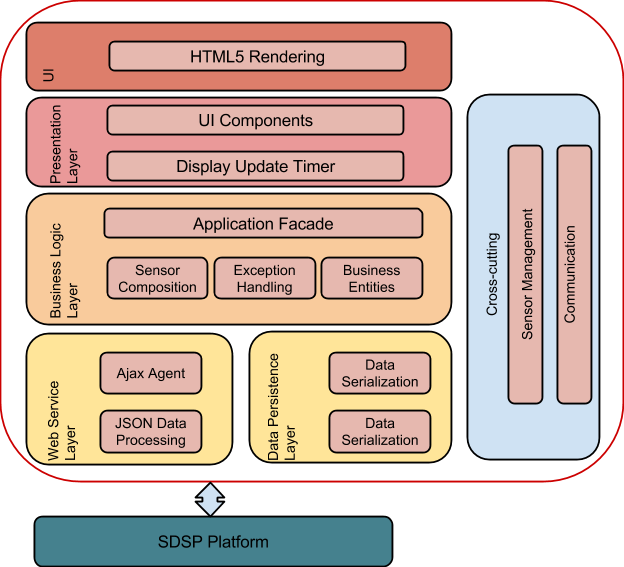


Figure : High Level Architecture of Sensor Visualization Platform

The above diagram (VirtualSensorEditor-Presentation-07262013.pptx n.d.) shows the overall architecture of the Virtual Sensor Editor Platform.

The figure shows the Sensor Data Service Provider Platform acting as the service layer to the Virtual Sensor Editor. From there, the various layers (Data Persistence, Web Service, Business Logic, Presentation Layer, User Interface (UI), and Cross-Cutting) contain key components, features, and functionality.

As can be inferred from the project background, the team’s major contributions fall within the Presentation, UI, and Business Logic Layers as exemplified in improving the sensor visualization dashboard, adding new components/control flows to the virtual sensor editor, exporting virtual sensor data serialization to XML, and implementing geofencing algorithms to support the future work of adding mobile devices as ad-hoc sensors.

## System Functional View

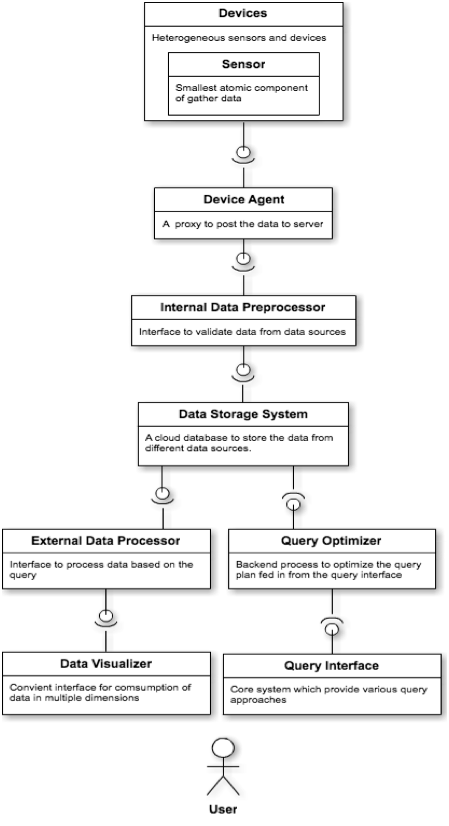


Figure : System Functional View

As seen from the figure, the devices and sensors collect data and get processed, stored, optimized, and finally visualized through the means of the virtual sensor editor for the end user. Team Facade’s project and tasks focuses within the data visualizer area depicted on the figure.

## Strategy Pattern: Virtual sensor serialization to XML

To facilitate the export of valid Extensible Markup Language (XML) to Taverna and Vistrails, both open source workflow management packages, the team utilized behavioral patterns that allow for the realization of common communication patterns between objects. In particular, the team made use of the strategy pattern. The benefit of the strategy pattern is that it separates out algorithms into classes that can then be plugged in at runtime. The figure below depicts the notion of a strategy pattern.

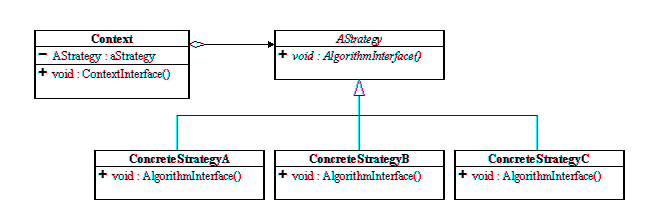


Figure : Strategy Pattern (Data & Object Factory, LLC. 2013)

The team’s task to convert the virtual sensor serialization to XML required many distinct behaviors where each behavior was implemented as a strategy. The serialization of the sensor data, originally expressed is JavaScript Object Notation (JSON) was first converted to XML using the x2js library with modifications needed to be made to the JSON structure.

The next strategy involved an Extensible Stylesheet Language (XSL) transformation to a format that can be imported into Taverna. Finally, a command line tool (xsltproc) was used for applying Extensible Stylesheet Language Transform (XSLT) stylesheets to the XML that allowed for the successful import into the Taverna workflow.

The use of the strategy patterns allows the changing of an object’s behavior dynamically without changing the object itself. A more detailed description of the team’s implementation of this task using this pattern is found later in the report.

## Design Pattern for Control Flow (Decision)

For the implementation of a control flow within the VSE application, the team made use of the observer design pattern. Because the team decided to implement a decision as the type of control flow, the inputs received from the virtual sensors directly corresponded to the output of the decision box. This allowed the team to utilize the update method in receiving sensor information and using that information to toggle the outputs of the decision and have this be displayed in real-time on the virtual sensor editor canvas.

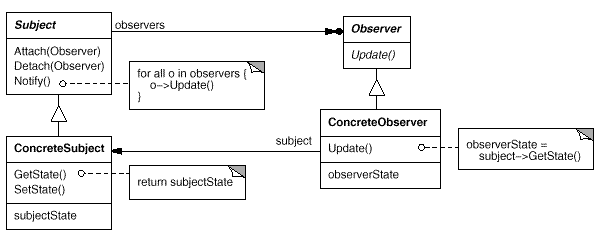


Figure : Observer Pattern (Hong 2013)

# Architecture Tradeoff Analysis Method (ATAM) Analysis

## Project Background

The Sensor Data Service Platform includes a Virtual Sensor Editor (VSE) that allows you to design and construct virtual sensors out of available physical sensor data. The virtual sensors can be saved and used within the VSE in the same way as physical sensors in order to construct other virtual sensors.

The VSE is a browser-based web application built with JavaScript and HTML5.

## Architectural Approaches

The following architectural approaches were used in designing the Virtual Sensor Editor and the Visualization Dashboard:

1. Tiered Architecture - As described in the diagram the architecture of the Virtual Sensor Platform is tiered, consisting of functional layers. Tiered architectures help in abstracting functionality that results in benefits such as easier refactoring and greater testability.
2. RESTful API - Both the editor and the dashboard render data from RESTful API services. RESTful clients are simpler to implement and are lighter on bandwidth. However REST does not cover other web services standards such as transactions, security etc.
3. Modularization - The Module pattern is a popular JS design pattern used in rich client side applications. This pattern helps to encapsulate private, state and organization using closures. Using this pattern only the public API is available keeping everything else within the closure private.
4. Object Literal Notation: The Virtual Sensor Editor and the Visualization Dashboard uses several third party JavaScript libraries to render complex client side graphics. The use of object literal notation helps to avoid polluting the global namespace as well as in good JavaScript code organization.
5. Model View Presenter (MVP) Pattern - This pattern was used in the design of the Virtual Sensor Editor. In this pattern the Presenter contains the UI business logic for the View. All invocations from the View delegate directly to Presenter. The presenter is decoupled from the view via an interface. The MVP pattern also consists of a lot two-way interactions between the view and the presenter. The MVP pattern shields the view from major API/data changes as the business logic can be easily handled in the presenter. However a drawback of the MVP pattern is that it results in additional work in creating setter properties etc. as well as the data binding between the presenter and view.
6. Mediation - This pattern was used to send events in background for thread processing in the Virtual Sensor Virtualization. Use of this pattern reduces the dependencies between communicating objects.
7. Client-Side Rendering: The project utilized client side rendering. Benefits of doing this include being more interactive with the user. However, if there is too much data, the client may be unresponsive.

## ATAM Utility Tree

|  |  |
| --- | --- |
| **Quality Attribute** | **Modifiability** |
| Attribute Concerns | The code should be easily modifiable by future teams |
| Scenarios | 1. The project will continue beyond the scope of one class so future teams should be able to easily modify the code in order to keep the project going. |
| **Quality Attribute** | **Extensibility** |
| Attribute Concerns | Virtual Sensor Editor should be extensible to address future business needs and stakeholders |
| Scenarios | 1. System should support new enhancements 2. VSE should handle new integrations to additional 3rd party systems 3. System should support changes in hardware and software infrastructure |
| **Quality Attribute** | **Performance** |
| Attribute Concerns | No degradation in response time of Virtual Sensor Editor |
| Scenarios | 1. Dozens of physical sensors are displayed in editor 2. User creates dozens of virtual sensors |
| Attribute Concerns | Dashboard should perform well under load |
| Scenarios | 1. User expands multiple sensor nodes |
| Attribute Concerns | Data Latency |
| Scenarios | 1. Virtual Sensor Editor data should be refreshed continuously every 10 seconds 2. Dashboard sensor data refreshed every 10 seconds |
| **Quality Attribute** | **Reliability** |
| Attribute Concerns | The Virtual Sensor Editor should be able to detect and recover from faults so that no “bad” data is collected. |
| Scenarios | 1. A sensor or sensor component fails. The Virtual Sensor dashboard is able to recognize a failure of the sensor and display status to the user. |
| Attribute Concerns | Graceful degradation in presence of a hardware sensor failure. |
| Scenarios | 1. The graph on the virtual sensor visualization dashboard shows a real-time data for a particular sensor. If no data is present, graph will be blank. |
| **Quality Attribute** | **Interoperability** |
| Attribute Concerns | The converted XML from the sensor serialization should work within different workflows |
| Scenarios | 1. A t2flow file imported into Taverna and a workflow is generated 2. A generated XML is compatible with VisTrails |
| Attribute Concerns | Geofencing algorithms work among different mobile devices |
| Scenarios | 1. An Android device enters (or leaves) the geofence and the point in polygon algorithm (Ray Casting or Jordan Curve theorems) detects the device is within (or outside) the area 2. An iOS device enters (or leaves) the geofence and the point in polygon algorithm (Ray Casting or Jordan Curve theorems) detects the device is within (or outside) the area |

Table : ATAM Utility Tree

## Scenario Generation and Prioritization

|  |  |  |
| --- | --- | --- |
| **Number** | **Scenario** | **Priority (1-high, 5-low)** |
| **1** | The project will continue beyond the scope of one class so future teams should be able to easily modify the code in order to keep the project going. | **1** |
| **2** | System should support new enhancements | **1** |
| **3** | VSE should handle new integrations to additional 3rd party systems | **2** |
| **4** | System should support changes in hardware and software infrastructure | **1** |
| **5** | Dozens of physical sensors are displayed in editor | **2** |
| **6** | User creates dozens of virtual sensors | **2** |
| **7** | User expands multiple sensor nodes | **2** |
| **8** | Virtual Sensor Editor data should be refreshed continuously every 10 seconds | **5** |
| **9** | Dashboard sensor data refreshed every 10 seconds | **5** |
| **10** | A sensor or sensor component fails. The Virtual Sensor dashboard is able to recognize a failure of the sensor and display status to the user. | **1** |
| **11** | The graph on the virtual sensor visualization dashboard shows a real-time data for a particular sensor. If no data is present, graph will be blank. | **1** |
| **12** | A t2flow file imported into Taverna and a workflow is generated | **2** |
| **13** | A generated XML is compatible with VisTrails | **5** |
| **14** | An Android device enters (or leaves) the geofence and the point in polygon algorithm (Ray Casting or Jordan Curve theorems) detects the device is within (or outside) the area | **3** |
| **15** | An iOS device enters (or leaves) the geofence and the point in polygon algorithm (Ray Casting or Jordan Curve theorems) detects the device is within (or outside) the area | **3** |

Table : Scenario Generation and Prioritization

# VSE Enhancements

## Sensor dashboard visualization

One of the team’s first tasks was to improve the current sensor monitoring dashboard. When the team started the project, the sensor nodes representing the physical sensors located within buildings at the CMUSV campus did not depict the sensor’s status. The team made use of developed APIs to poll the sensors to receive the status of the sensors. If the sensors were active, the nodes of the sensors would be colored blue. Inactive or non-responsive sensors were marked as red to allow users to easily see the status of each sensor. In addition to adding the sensor status to the dashboard, the team also added a time series data graph. The graph would allow a user to graphically view real-time data from the sensor that was selected on the dashboard.

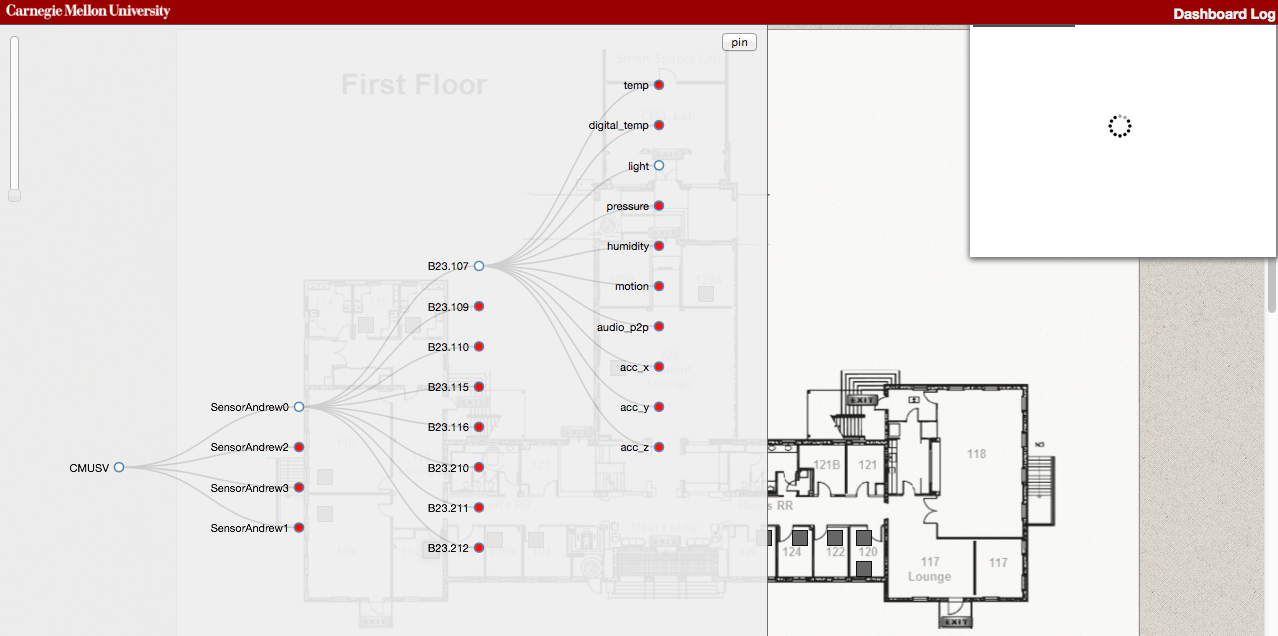


Figure : Sensor Monitor Dashboard

Because of known issues causing the sensors to be unresponsive in providing real-time data, the team used known past data to implement the functionality for completing this task.

## Control flows

The team was also tasked with implementing a basic control flow to the virtual sensor composition. The team decided to implement a basic decision structure for this task. This involved writing a new function to create a new control flow on the virtual sensor editor canvas. The decision structure takes in an input from a sensor, and depending upon the input and the explicitly created JavaScript function, toggles between two outputs, true and false.

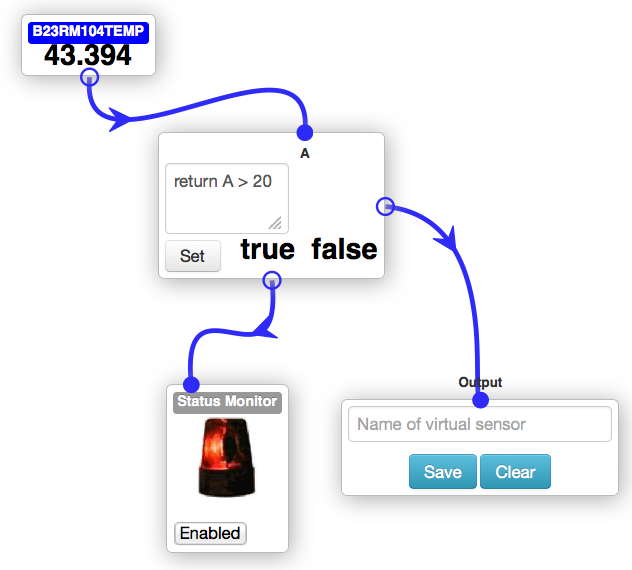


Figure : Decision Control Flow in Sensor Editor – Status Monitor True

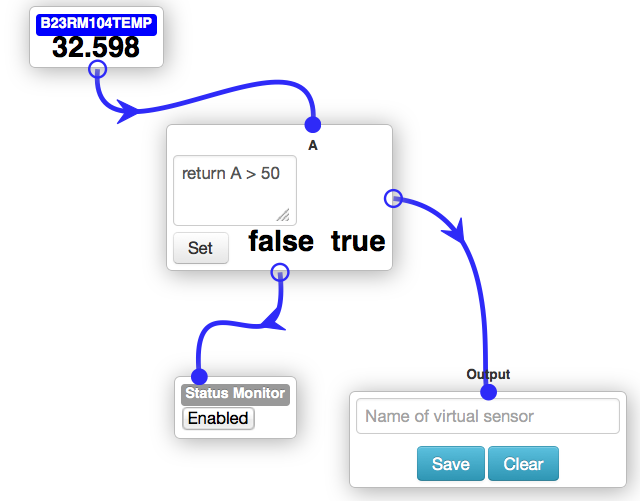


Figure : Decision Control Flow in Sensor Editor – Status Monitor False

## Taverna Integration

The VSE stores the virtual sensor definitions as JSON. Team Facade was tasked with exporting virtual sensors to XML for storage and later retrieval. Another team was tasked with storing the XML to the database; Team Facade’s implementation saves the XML in the browser’s local storage.

### Why XML?

In addition to being human-readable, XML can be associated with a schema, thus making machine understanding possible. This gives XML an advantage over JSON, which is schema-less.

### What is Taverna?

Taverna (<http://www.taverna.org.uk>) is an open-source workflow management package. It allows you to create workflows, assign values to inputs, run the workflow, and receive the output. Taverna workflows can be exported to XML (.t2flow) for storage and later retrieval. Taverna workflow XML has a well-documented schema.

### JSON to XML

As noted, the virtual sensors were already being saved as JSON internally by the VSE. The structure of the JSON was as follows

{

“VIRTUAL\_SENSORS”: {...},

“SETTINGS”: {}

}

To convert the JSON to XML we found the x2js (<https://code.google.com/p/x2js/>) JavaScript library. We found that the initial structure of the JSON did not produce valid XML because there was not a single root object. Therefore, we modified the JSON to have this structure:

{

“ROOT”: {

“VIRTUAL\_SENSORS”: {...},

“SETTINGS”:{}

}

}

This resulted in XML like the following:

<?xml version=“1.0”?>

<ROOT>

<VIRTUAL\_SENSORS>

...

</VIRTUAL\_SENSORS>

<SETTINGS/>

</ROOT>

The original JSON representation of the virtual sensor is saved to the browser’s local storage as storageObj. The output of the x2js library is stored as storageObjXml.

### XML to Taverna (.t2flow)

Next, Team Facade’s strategy was to import the XML into Taverna because Taverna already has a well-documented schema. To make this work the team wrote an XSL transformation to massage the x2js output into the format required by Taverna. As this was just a proof-of-concept transformation, a simple virtual sensor consisting of two physical sensors combined by a function box was used (see following figure).

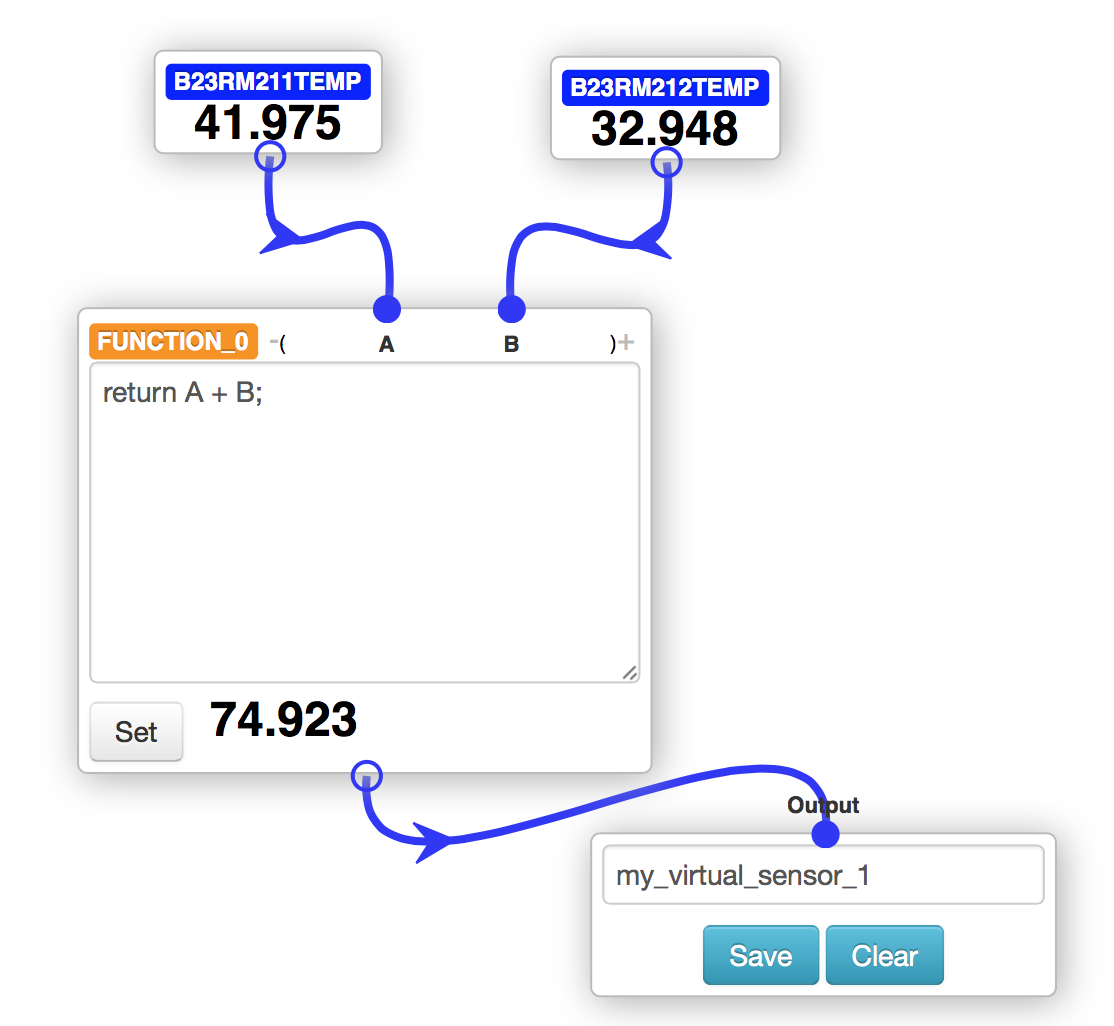


Figure : Sample Virtual Sensor

The first step in writing the transformation was to adjust the output of the JavaScript-generated UUIDs to be in Taverna format. The old code generated a 16-digit UUID composed of a randomly generated string of the first 16 letters of the alphabet:

function createUUID(prefix) {

var s = [];

var hexDigits = "abcdefghijklmnop";

for (var i = 0; i < 16; i++) {

s[i] = hexDigits.substr(Math.floor(Math.random() \* 0x10), 1);

}

var uuid = prefix + s.join("");

return uuid;

}

Taverna-formatted UUIDs consist of the more traditional hex digits [0-9a-f] grouped into strings of 8, 4, 4, 4, and 12. For example: ab56bb3d-f8ee-ac45-831e-61657f49f981. To update this, we changed designtool\_BLL.js as follows:

function createUUID(prefix) {

var strings = [];

strings[0] = generateRandomHexString(8);

strings[1] = generateRandomHexString(4);

strings[2] = generateRandomHexString(4);

strings[3] = generateRandomHexString(4);

strings[4] = generateRandomHexString(12);

return prefix + strings.join("-");

}

function generateRandomHexString(numDigitsToOutput) {

var s = [];

var hexDigits = "0123456789abcdef";

for (var i = 0; i < numDigitsToOutput; i++) {

s[i] = hexDigits.substr(Math.floor(Math.random() \* 0x10), 1);

}

return s.join("");

}

After updating the UUIDs the next step was to add a “name” element to the JSON that contained the name of the virtual sensor so that it would appear in the XML created by x2js. Previously, the name of each virtual sensor was an immediate child of the <VIRTUAL\_SENSORS> element. For example, if there were two virtual sensors named VIRTUAL\_SENSOR\_1 and VIRTUAL\_SENSOR\_2, the XML would look like this:

<?xml version=“1.0”?>

<ROOT>

<VIRTUAL\_SENSORS>

<VIRTUAL\_SENSOR\_1>

...

</VIRTUAL\_SENSOR\_1>

<VIRTUAL\_SENSOR\_2>

...

</VIRTUAL\_SENSOR\_2>

</VIRTUAL\_SENSORS>

<SETTINGS/>

</ROOT>

We did not know how to use XPath to address the dynamically named elements, so we decided to modify the JSON/XML to produce the following XML:

<?xml version=“1.0”?>

<ROOT>

<VIRTUAL\_SENSORS>

<VIRTUAL\_SENSOR\_1>

...

<name>VIRTUAL\_SENSOR\_1</name>

</VIRTUAL\_SENSOR\_1>

<VIRTUAL\_SENSOR\_2>

...

<name>VIRTUAL\_SENSOR\_2</name>

</VIRTUAL\_SENSOR\_2>

</VIRTUAL\_SENSORS>

<SETTINGS/>

</ROOT>

With this change in place, we took a sample of storageObjXml to see what it looked like:

<?xml version="1.0"?>

<ROOT>

<VIRTUAL\_SENSORS>

<VIRTUAL\_SENSOR\_1>

<components>

<uuid>7e7332f3-cb1d-811f-d51a-843afbe287d0</uuid>

<value>

<category>custom</category>

<name>FUNCTION</name>

<setIntervalId>14</setIntervalId>

<expression>return A + B;</expression>

<children>ab56bb3d-f8ee-ac45-831e-61657f49f981</children>

<children>727a45b1-cacc-3a49-3487-f8d5ba84ce41</children>

<top>234.70666980743408</top>

<left>337.6153259277344</left>

</value>

</components>

<components>

<uuid>727a45b1-cacc-3a49-3487-f8d5ba84ce41</uuid>

<value>

<category>physical</category>

<sensorType>digital\_temp</sensorType>

<deviceID>10170003</deviceID>

<name>B23RM116TEMP</name>

<setIntervalId>8</setIntervalId>

<top>120.7066650390625</top>

<left>557.6152954101562</left>

</value>

</components>

<components>

<uuid>ab56bb3d-f8ee-ac45-831e-61657f49f981</uuid>

<value>

<category>physical</category>

<sensorType>digital\_temp</sensorType>

<deviceID>10170002</deviceID>

<name>B23RM115TEMP</name>

<setIntervalId>6</setIntervalId>

<top>125.7066650390625</top>

<left>282.6153259277344</left>

</value>

</components>

<source>7e7332f3-cb1d-811f-d51a-843afbe287d0</source>

<version>1</version>

<name>VIRTUAL\_SENSOR\_1</name>

</VIRTUAL\_SENSOR\_1>

</VIRTUAL\_SENSORS>

<SETTINGS/>

</ROOT>

Next, we created a sample workflow in Taverna and saved it so we could find the minimum amount of data needed to be able to recreate the saved workflow. This was a trial and error process as we commented out different sections of data. Once we narrowed down the minimum amount of data needed to successfully import a .t2flow file into Taverna, we set out to transform our storageObjXml into a viable .t2flow file.

The process we used to create the XSLT file was to save the output of storageObjXml into a file on the Mac desktop called input.xml. We found that on the Mac there is a built-in command line program called xsltproc, which is a command line XSLT processor. We created an XSLT file called Taverna.xsl and also saved it to the Mac desktop. We wanted to verify that the data output from xsltproc was valid in the eyes of the Taverna schema so we also made use of xmllint, another command line program. The Taverna schema is available at <http://taverna.googlecode.com/svn/taverna/dev/xsd/trunk/t2flow/t2flow.xsd>, which we saved to the Mac desktop as t2flow.xsd. Assuming the current directory is ~/Desktop, the final command we used to transform the XML and check the output against the schema was:

$ xsltproc -o transformed.xml Taverna.xsl input.xml; xmllint --format --schema t2flow.xsd transformed.xml

The XSLT to get a basic importable .t2flow file is shown here:

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">

<workflow xmlns="http://taverna.sf.net/2008/xml/t2flow" version="1" producedBy="CMUSV">

<xsl:for-each select="ROOT/VIRTUAL\_SENSORS/\*">

<dataflow>

<xsl:attribute name="id">

<xsl:value-of select="source"/>

</xsl:attribute>

<xsl:attribute name="role">top</xsl:attribute>

<xsl:variable name="uuid" select="source"/>

<name><xsl:value-of select="name"/></name>

<inputPorts>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<port>

<name><xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/></name>

<depth>0</depth><!-- TODO: hard coded to 0 for now -->

<granularDepth>0</granularDepth><!-- TODO: hard coded to 0 for now -->

</port>

</xsl:for-each>

</inputPorts>

<outputPorts>

<port>

<name><xsl:value-of select="name"/></name>

<annotations></annotations>

</port>

</outputPorts>

<processors>

<processor>

<name><xsl:value-of select="components/value/name"/></name>

<inputPorts>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<port>

<name><xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/></name>

<depth>0</depth><!-- TODO: hard coded for now -->

</port>

</xsl:for-each>

</inputPorts>

<outputPorts>

<port>

<name><xsl:value-of select="components/value/name"/></name>

<depth>0</depth><!-- TODO: hard coded for now -->

<granularDepth>0</granularDepth><!-- TODO: hard coded to 0 for now -->

</port>

</outputPorts>

<annotations></annotations>

<activities></activities>

<dispatchStack></dispatchStack>

<iterationStrategyStack>

<iteration>

<strategy></strategy>

</iteration>

</iterationStrategyStack>

</processor>

</processors>

<conditions></conditions>

<datalinks>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<datalink>

<sink type="processor">

<processor><xsl:value-of select="../name"/></processor>

<port><xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/></port>

</sink>

<source type="dataflow">

<port><xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/></port>

</source>

</datalink>

</xsl:for-each>

<datalink>

<sink type="dataflow">

<port><xsl:value-of select="name"/></port>

</sink>

<source type="processor">

<processor><xsl:value-of select="components/value/name"/></processor>

<port><xsl:value-of select="components/value/name"/></port>

</source>

</datalink>

</datalinks>

<annotations></annotations>

</dataflow>

</xsl:for-each>

</workflow>

</xsl:template>

</xsl:stylesheet>

Once the transformation was working we added JavaScript code to do the processing and store the result to storageObjTaverna in the browser’s local storage. The user can also copy this converted XML from the local storage and save it as a .t2flow file. This .t2flow file can then be opened in Taverna.

### Importing to Taverna

At this point the output of xsltproc, transformed.xml, is a valid Taverna .t2flow file that can be imported into Taverna (see following figure):

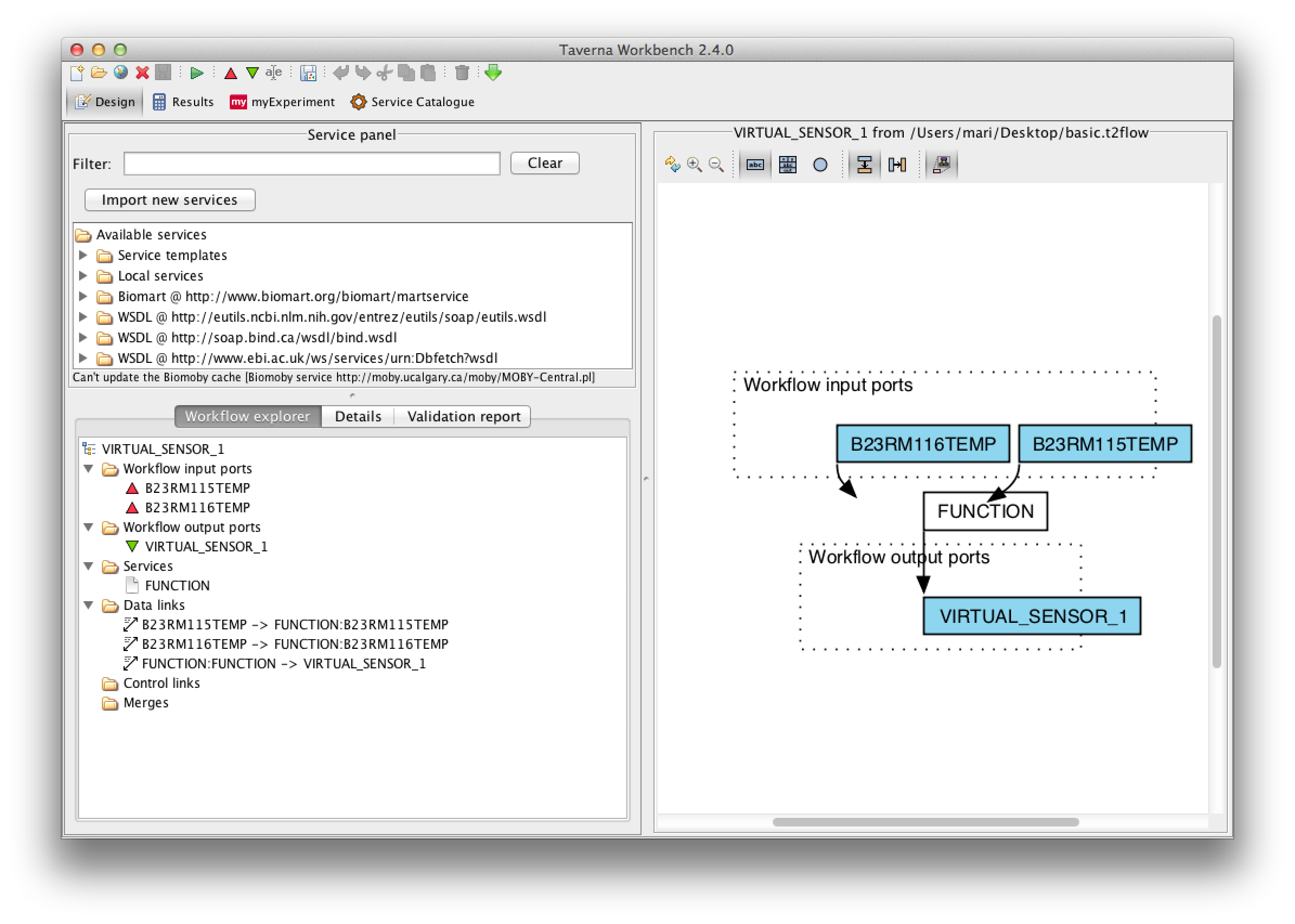


Figure : Taverna Workbench

However, the box in the picture labeled “FUNCTION” is not functional and does not do anything as indicated by the white color. Our next goal was to figure out how to make “FUNCTION” be able to do simple arithmetic. One obstacle is that the VSE is written in JavaScript and the Taverna system is written in Java. Taverna uses the Beanshell scripting language, which is Java-based. Since we could not see an easy way to convert JavaScript to Java, we decided to just hard code an addition function that would add the two inputs (represented by B23RM116TEMP and B23RM115TEMP), but would serve as a proof-of-concept showing that it is possible to create an XML file that contains embedded workflow processing instructions.

### Implementing FUNCTION Functionality

To implement the FUNCTION functionality, the team needed an example to work from. First we created a Taverna workflow using a Beanshell script that simply added two numbers and examined the resulting .t2flow file. We could not decipher how it worked, so we looked for other solutions. The team found Taverna Online at <http://onlinehpc.com/site/main> and one of the team members signed up for an account.

The Taverna Online tool is similar to the VSE in that it allows you to use a browser-based GUI to put together workflows. The resulting workflow can be exported to a .t2flow file. We used Taverna Online to create a simple adder that adds two inputs and gives one output and exported it to a .t2flow file. We then examined and experimented with this file to see what the essential parts were. Once we knew which parts were essential to making our FUNCTION work, we modified Taverna.xsl to add the new information to the transformation. Our modified output allowed the FUNCTION box in Taverna to become active, as shown by the magenta color:

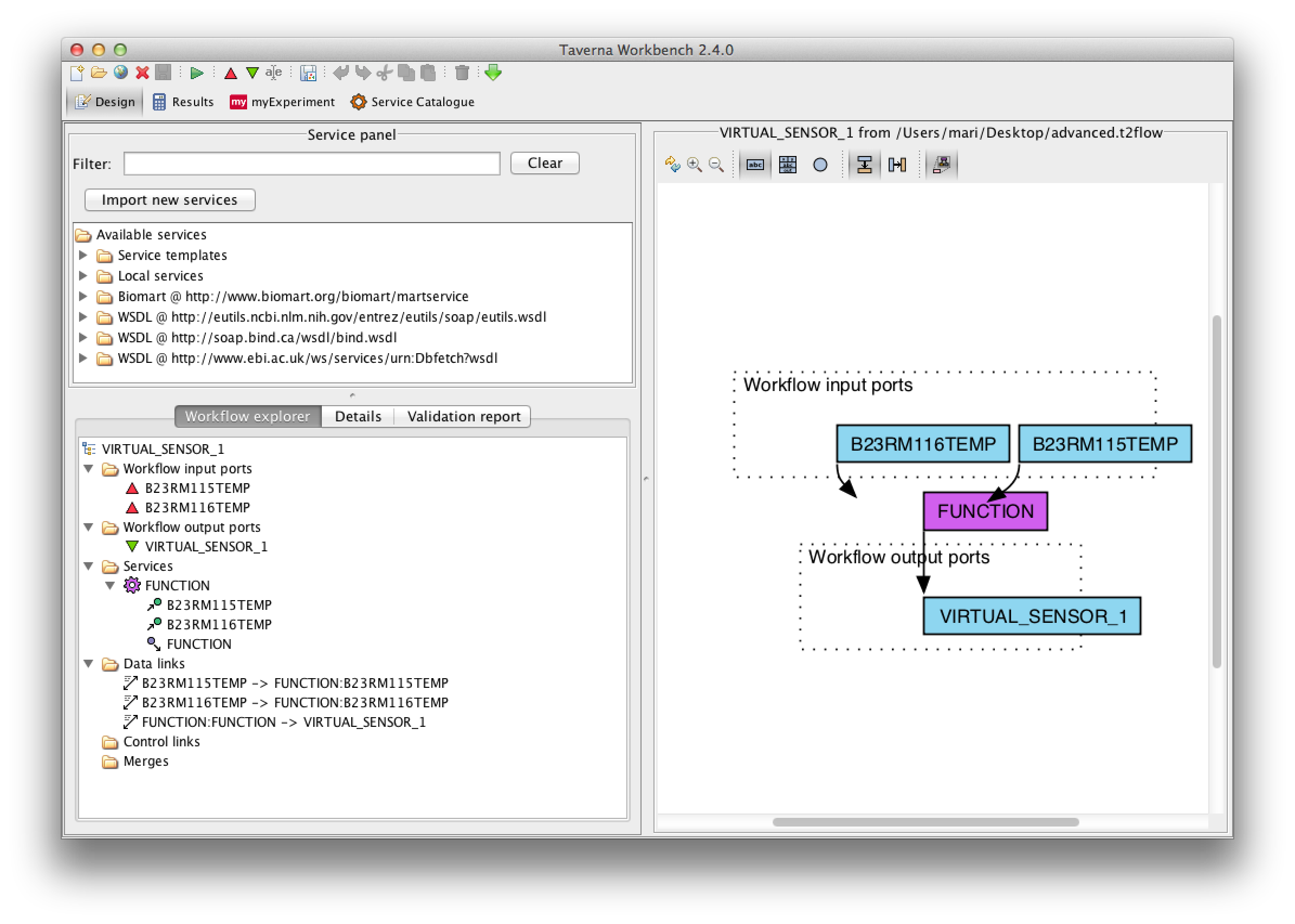
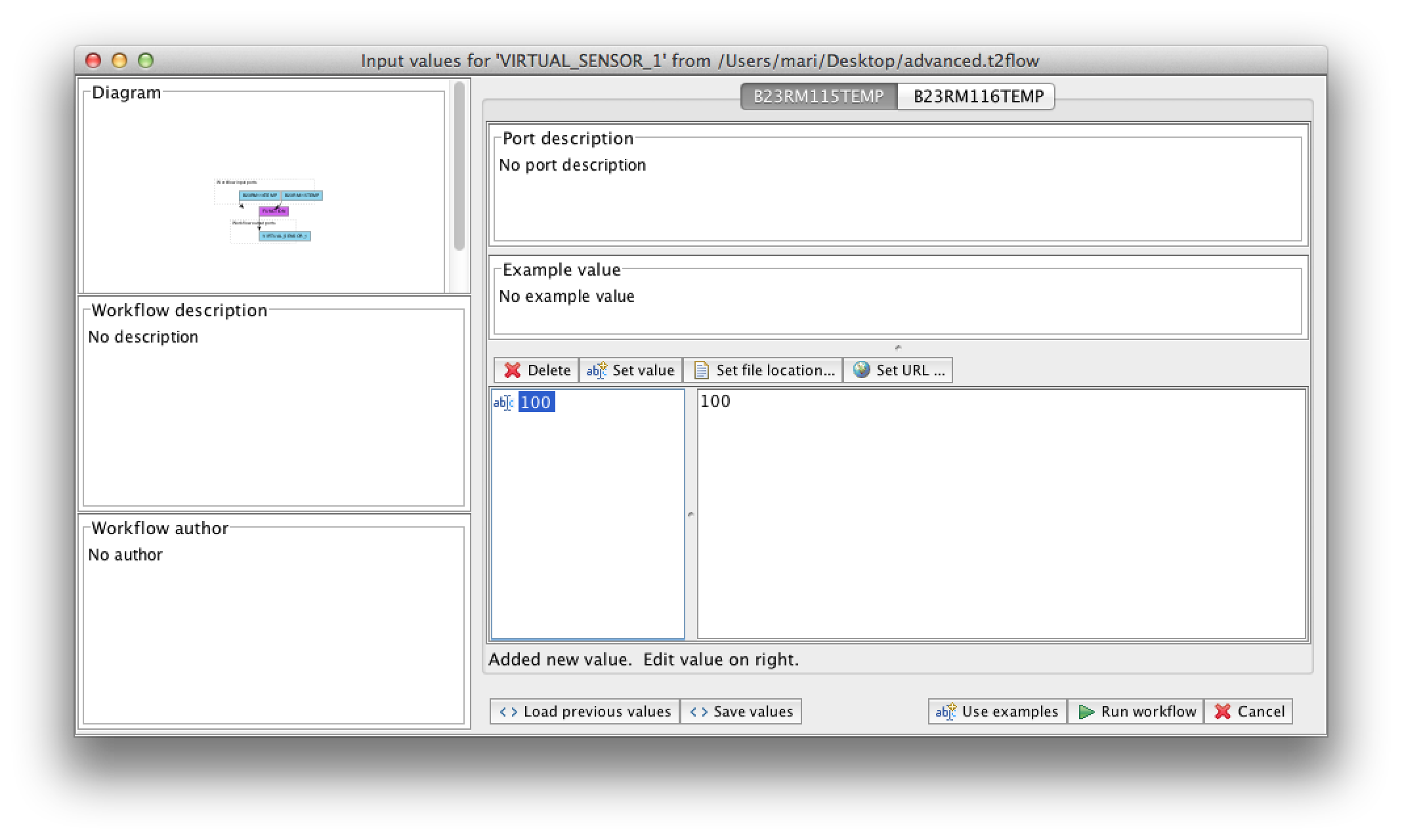


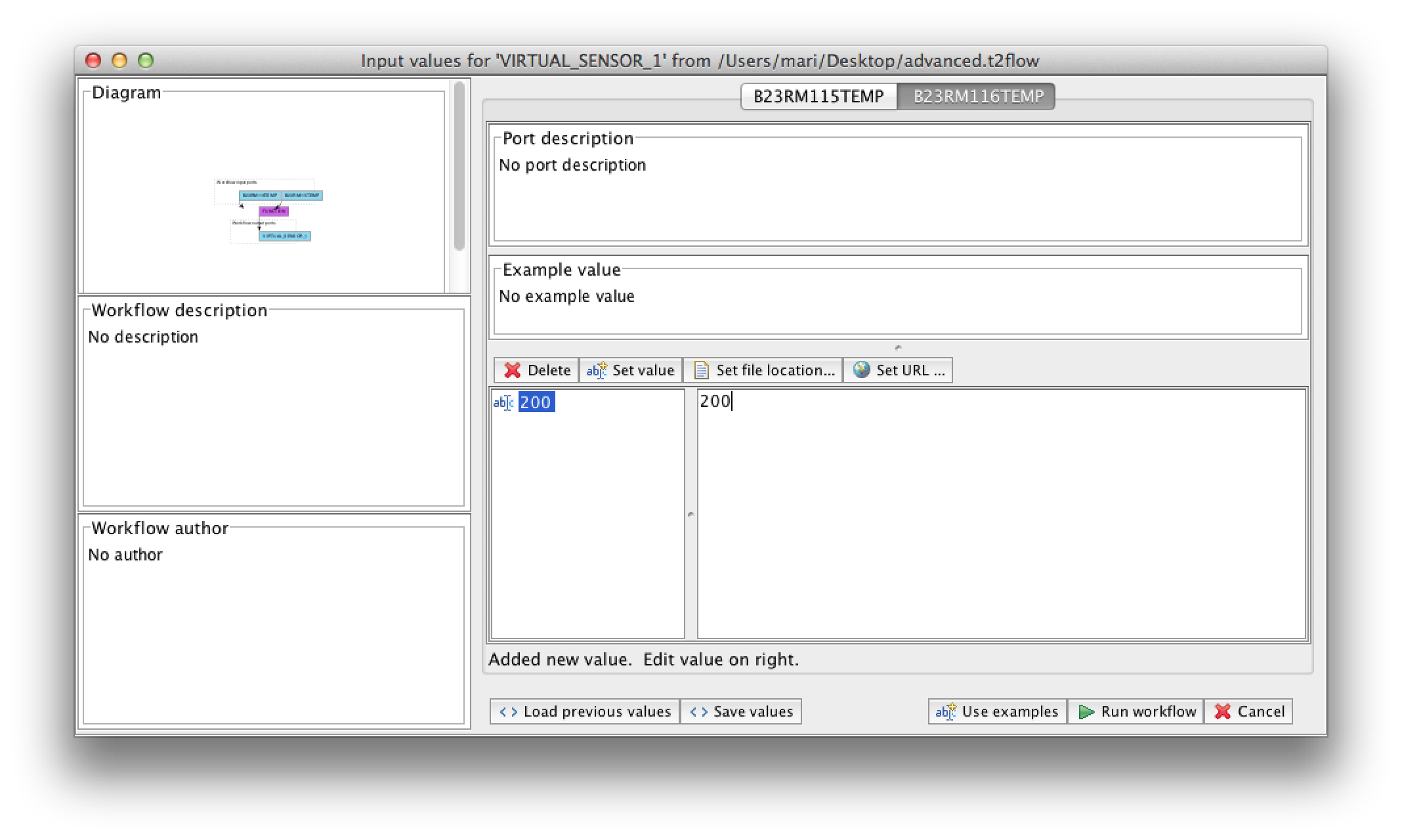
Figure : Taverna Function

To verify that our modified workflow actually worked, we assigned values to the input ports and ran the workflow.

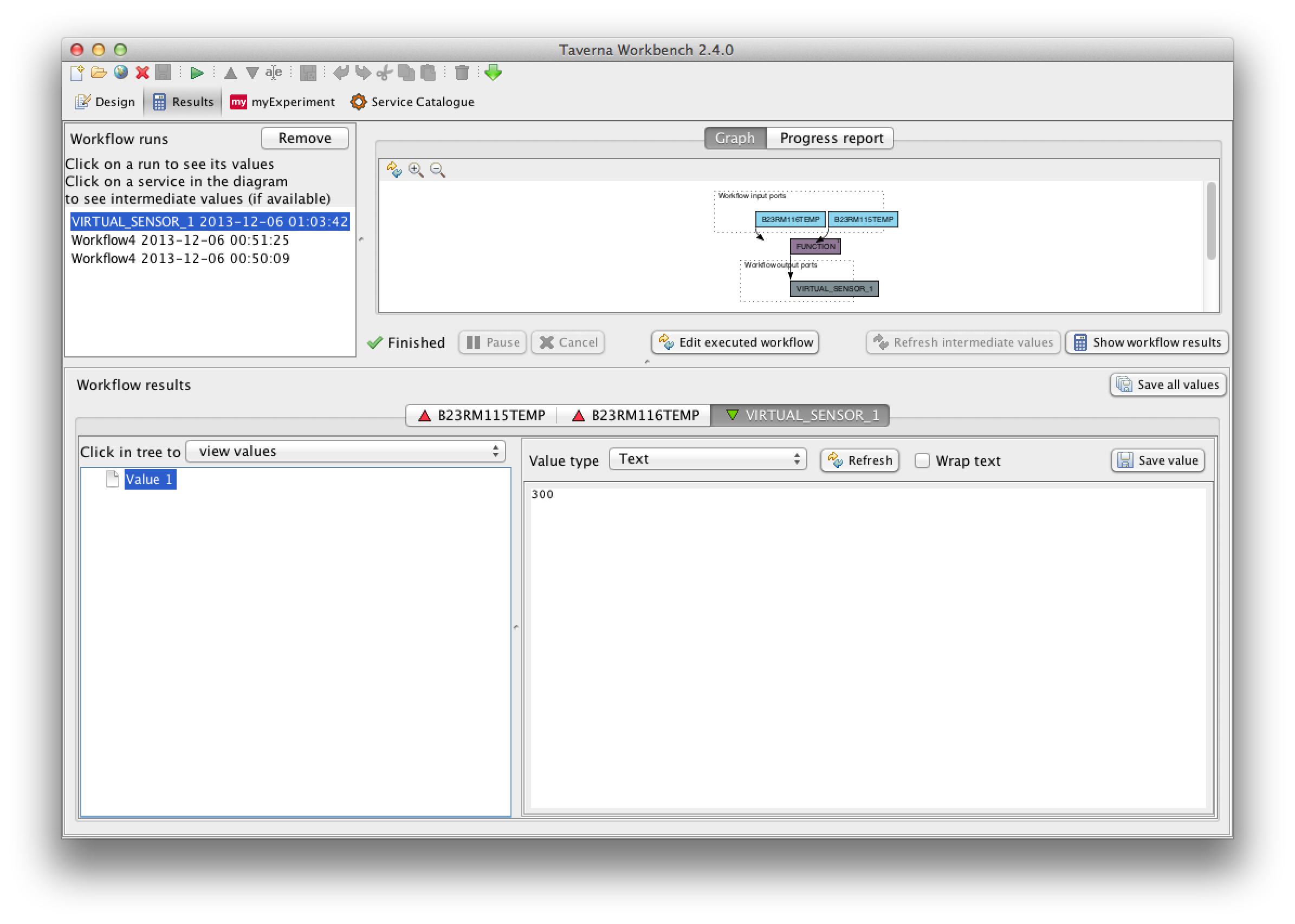
First, we assigned the value of 100 to the B23RM115TEMP input:



Next, we assigned the value of 200 to the B23RM116TEMP input:



Next, we clicked the “Run workflow” button and received the result of 300:



The updated Tavernal.xsl is more complex than the non-functional version:

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="xml" indent="yes" encoding="UTF-8"/>

<xsl:template match="/">

<workflow xmlns="http://taverna.sf.net/2008/xml/t2flow" version="1" producedBy="CMUSV">

<xsl:for-each select="ROOT/VIRTUAL\_SENSORS/\*">

<dataflow>

<xsl:attribute name="id">

<xsl:value-of select="source"/>

</xsl:attribute>

<xsl:attribute name="role">top</xsl:attribute>

<xsl:variable name="uuid" select="source"/>

<name>

<xsl:value-of select="name"/>

</name>

<inputPorts>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<port>

<name>

<xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/>

</name>

<depth>0</depth>

<!-- TODO: hard coded to 0 for now -->

<granularDepth>0</granularDepth>

<!-- TODO: hard coded to 0 for now -->

</port>

</xsl:for-each>

</inputPorts>

<outputPorts>

<port>

<name>

<xsl:value-of select="name"/>

</name>

<annotations/>

</port>

</outputPorts>

<processors>

<processor>

<name>

<xsl:value-of select="components/value/name"/>

</name>

<inputPorts>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<port>

<name>

<xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/>

</name>

<depth>0</depth>

<!-- TODO: hard coded for now -->

</port>

</xsl:for-each>

</inputPorts>

<outputPorts>

<port>

<name>

<xsl:value-of select="components/value/name"/>

</name>

<depth>0</depth>

<!-- TODO: hard coded for now -->

<granularDepth>0</granularDepth>

<!-- TODO: hard coded to 0 for now -->

</port>

</outputPorts>

<annotations/>

<activities>

<activity>

<raven>

<group>net.sf.taverna.t2.activities</group>

<artifact>localworker-activity</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.activities.localworker.LocalworkerActivity

</class>

<inputMap>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<map>

<xsl:attribute name="from">

<xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/>

</xsl:attribute>

<xsl:attribute name="to">

<xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/>

</xsl:attribute>

</map>

</xsl:for-each>

</inputMap>

<outputMap>

<map>

<xsl:attribute name="from">

<xsl:value-of select="components/value/name"/>

</xsl:attribute>

<xsl:attribute name="to">

<xsl:value-of select="components/value/name"/>

</xsl:attribute>

</map>

</outputMap>

<configBean encoding="xstream">

<net.sf.taverna.t2.activities.localworker.LocalworkerActivityConfigurationBean xmlns="">

<inputs>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityInputPortDefinitionBean>

<name><xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/></name>

<depth>0</depth>

<mimeTypes>

<string>'text/plain'</string>

</mimeTypes>

<handledReferenceSchemes/>

<translatedElementType>java.lang.String</translatedElementType>

<allowsLiteralValues>true</allowsLiteralValues>

</net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityInputPortDefinitionBean>

</xsl:for-each>

</inputs>

<outputs>

<net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityOutputPortDefinitionBean>

<name><xsl:value-of select="components/value/name"/></name>

<depth>0</depth>

<mimeTypes>

<string>'text/plain'</string>

</mimeTypes>

<granularDepth>0</granularDepth>

</net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityOutputPortDefinitionBean>

</outputs>

<classLoaderSharing>workflow</classLoaderSharing>

<localDependencies/>

<artifactDependencies/>

<script><xsl:value-of select="components/value/name"/> = Integer.parseInt(<xsl:value-of select="components[2]/value/name"/>) + Integer.parseInt(<xsl:value-of select="components[3]/value/name"/>);</script>

<dependencies/>

<localworkerName>ru.iitp.cluster.taverna.Localworker</localworkerName>

</net.sf.taverna.t2.activities.localworker.LocalworkerActivityConfigurationBean>

</configBean>

<annotations/>

</activity>

</activities>

<dispatchStack>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Parallelize

</class>

<configBean encoding="xstream">

<net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.ParallelizeConfig xmlns="">

<maxJobs>1</maxJobs>

</net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.ParallelizeConfig>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.ErrorBounce

</class>

<configBean encoding="xstream">

<null xmlns=""/>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Failover

</class>

<configBean encoding="xstream">

<null xmlns=""/>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Retry

</class>

<configBean encoding="xstream">

<net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.RetryConfig xmlns="">

<backoffFactor>1</backoffFactor>

<initialDelay>1000</initialDelay>

<maxDelay>5000</maxDelay>

<maxRetries>0</maxRetries>

</net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.RetryConfig>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Invoke

</class>

<configBean encoding="xstream">

<null xmlns=""/>

</configBean>

</dispatchLayer>

</dispatchStack>

<iterationStrategyStack>

<iteration>

<strategy>

<cross>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<port>

<xsl:attribute name="name">

<xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/>

</xsl:attribute>

<xsl:attribute name="depth">0</xsl:attribute>

</port>

</xsl:for-each>

</cross>

</strategy>

</iteration>

</iterationStrategyStack>

</processor>

</processors>

<conditions/>

<datalinks>

<xsl:for-each select="components/value/children">

<xsl:variable name="childUuid" select="text()"/>

<datalink>

<sink type="processor">

<processor>

<xsl:value-of select="../name"/>

</processor>

<port>

<xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/>

</port>

</sink>

<source type="dataflow">

<port>

<xsl:value-of select="/ROOT/VIRTUAL\_SENSORS/\*/components[uuid=$childUuid]/value/name"/>

</port>

</source>

</datalink>

</xsl:for-each>

<datalink>

<sink type="dataflow">

<port>

<xsl:value-of select="name"/>

</port>

</sink>

<source type="processor">

<processor>

<xsl:value-of select="components/value/name"/>

</processor>

<port>

<xsl:value-of select="components/value/name"/>

</port>

</source>

</datalink>

</datalinks>

<annotations/>

</dataflow>

</xsl:for-each>

</workflow>

</xsl:template>

</xsl:stylesheet>

The XML it produces is as follows:

<?xml version="1.0" encoding="UTF-8"?>

<workflow xmlns="http://taverna.sf.net/2008/xml/t2flow" version="1" producedBy="CMUSV">

<dataflow id="7e7332f3-cb1d-811f-d51a-843afbe287d0" role="top">

<name>VIRTUAL\_SENSOR\_1</name>

<inputPorts>

<port>

<name>B23RM115TEMP</name>

<depth>0</depth>

<granularDepth>0</granularDepth>

</port>

<port>

<name>B23RM116TEMP</name>

<depth>0</depth>

<granularDepth>0</granularDepth>

</port>

</inputPorts>

<outputPorts>

<port>

<name>VIRTUAL\_SENSOR\_1</name>

<annotations/>

</port>

</outputPorts>

<processors>

<processor>

<name>FUNCTION</name>

<inputPorts>

<port>

<name>B23RM115TEMP</name>

<depth>0</depth>

</port>

<port>

<name>B23RM116TEMP</name>

<depth>0</depth>

</port>

</inputPorts>

<outputPorts>

<port>

<name>FUNCTION</name>

<depth>0</depth>

<granularDepth>0</granularDepth>

</port>

</outputPorts>

<annotations/>

<activities>

<activity>

<raven>

<group>net.sf.taverna.t2.activities</group>

<artifact>localworker-activity</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.activities.localworker.LocalworkerActivity

</class>

<inputMap>

<map from="B23RM115TEMP" to="B23RM115TEMP"/>

<map from="B23RM116TEMP" to="B23RM116TEMP"/>

</inputMap>

<outputMap>

<map from="FUNCTION" to="FUNCTION"/>

</outputMap>

<configBean encoding="xstream">

<net.sf.taverna.t2.activities.localworker.LocalworkerActivityConfigurationBean xmlns="">

<inputs>

<net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityInputPortDefinitionBean>

<name>B23RM115TEMP</name>

<depth>0</depth>

<mimeTypes>

<string>'text/plain'</string>

</mimeTypes>

<handledReferenceSchemes/>

<translatedElementType>java.lang.String</translatedElementType>

<allowsLiteralValues>true</allowsLiteralValues>

</net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityInputPortDefinitionBean>

<net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityInputPortDefinitionBean>

<name>B23RM116TEMP</name>

<depth>0</depth>

<mimeTypes>

<string>'text/plain'</string>

</mimeTypes>

<handledReferenceSchemes/>

<translatedElementType>java.lang.String</translatedElementType>

<allowsLiteralValues>true</allowsLiteralValues>

</net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityInputPortDefinitionBean>

</inputs>

<outputs>

<net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityOutputPortDefinitionBean>

<name>FUNCTION</name>

<depth>0</depth>

<mimeTypes>

<string>'text/plain'</string>

</mimeTypes>

<granularDepth>0</granularDepth>

</net.sf.taverna.t2.workflowmodel.processor.activity.config.ActivityOutputPortDefinitionBean>

</outputs>

<classLoaderSharing>workflow</classLoaderSharing>

<localDependencies/>

<artifactDependencies/>

<script>FUNCTION = Integer.parseInt(B23RM116TEMP) + Integer.parseInt(B23RM115TEMP);</script>

<dependencies/>

<localworkerName>ru.iitp.cluster.taverna.Localworker</localworkerName>

</net.sf.taverna.t2.activities.localworker.LocalworkerActivityConfigurationBean>

</configBean>

<annotations/>

</activity>

</activities>

<dispatchStack>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Parallelize

</class>

<configBean encoding="xstream">

<net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.ParallelizeConfig xmlns="">

<maxJobs>1</maxJobs>

</net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.ParallelizeConfig>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.ErrorBounce

</class>

<configBean encoding="xstream">

<null xmlns=""/>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Failover

</class>

<configBean encoding="xstream">

<null xmlns=""/>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Retry

</class>

<configBean encoding="xstream">

<net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.RetryConfig xmlns="">

<backoffFactor>1</backoffFactor>

<initialDelay>1000</initialDelay>

<maxDelay>5000</maxDelay>

<maxRetries>0</maxRetries>

</net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.RetryConfig>

</configBean>

</dispatchLayer>

<dispatchLayer>

<raven>

<group>net.sf.taverna.t2.core</group>

<artifact>workflowmodel-impl</artifact>

<version>1.4</version>

</raven>

<class>

net.sf.taverna.t2.workflowmodel.processor.dispatch.layers.Invoke

</class>

<configBean encoding="xstream">

<null xmlns=""/>

</configBean>

</dispatchLayer>

</dispatchStack>

<iterationStrategyStack>

<iteration>

<strategy>

<cross>

<port name="B23RM115TEMP" depth="0"/>

<port name="B23RM116TEMP" depth="0"/>

</cross>

</strategy>

</iteration>

</iterationStrategyStack>

</processor>

</processors>

<conditions/>

<datalinks>

<datalink>

<sink type="processor">

<processor>FUNCTION</processor>

<port>B23RM115TEMP</port>

</sink>

<source type="dataflow">

<port>B23RM115TEMP</port>

</source>

</datalink>

<datalink>

<sink type="processor">

<processor>FUNCTION</processor>

<port>B23RM116TEMP</port>

</sink>

<source type="dataflow">

<port>B23RM116TEMP</port>

</source>

</datalink>

<datalink>

<sink type="dataflow">

<port>VIRTUAL\_SENSOR\_1</port>

</sink>

<source type="processor">

<processor>FUNCTION</processor>

<port>FUNCTION</port>

</source>

</datalink>

</datalinks>

<annotations/>

</dataflow>

</workflow>

Again, the transformed XML output (.t2flow formatted) is stored in the browser’s local storage as storageObjTaverna.

## Geofencing

Currently there are physical sensors distributed around the CMUSV campus that push information to the Sensor Data Platform. But now there are more and more location-aware devices such as smartphones and tablets also have sensor data. The concept is to utilize these devices by building a geofence or virtual perimeter for the Carnegie Mellon Silicon Valley campus. When the location-aware devices enters this geofence, the device will automatically send sensor data to the Sensor Data Platform.

**Point-In-Polygon Algorithms**

Another team is responsible for creating a smartphone app and pushing the data to the Sensor Data Platform. Team Facade is responsible for defining the algorithm for determining when a device enters or exits a geofence. We are implementing our geofence as a simple array of latitude and longitude points that form polygon. Once we have our list of latitude and longitude points, we use two point-in-polygon algorithms to check if a given point is inside or outside the polygon. The two algorithms are the Ray Casting Algorithm and Jordan Curve Theorem.

#### Ray Casting Algorithm

Starting from outside a polygon, we “cast” a ray towards the point in question. If the ray crosses a odd number of lines (borders) to reach the point, we are inside the polygon, otherwise we are outside (see fig. Ray). This algorithm is also known as the crossing number algorithm or the even-odd rule algorithm. (Wikipedia 2013)

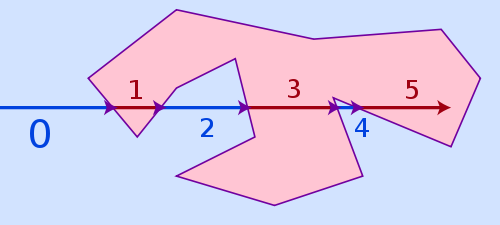
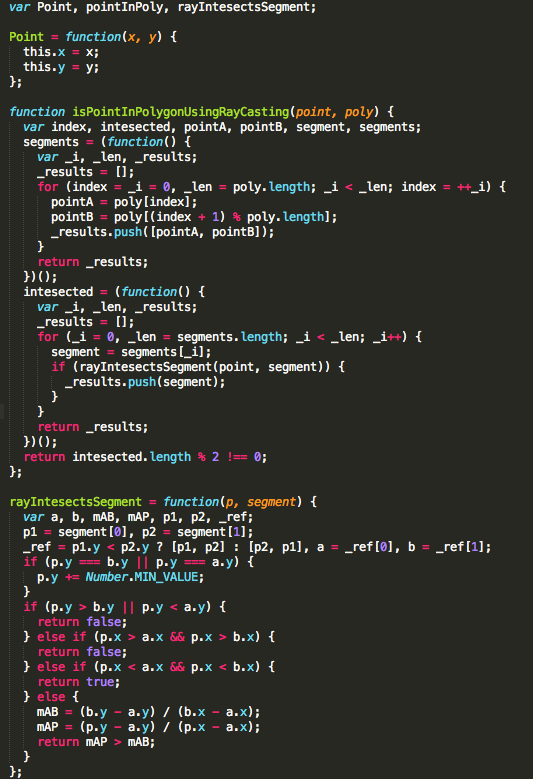


Figure : Ray Casting Algorithm

The point-in-polygon algorithms are implemented as JavaScript Functions in “pointInPolygon.js”. The Ray Casting function is defined as “isPointInPolygonUsingRayCasting” and is detailed below. Sample code to invoke this function can be found in “PointInPolygon.html”.



#### Jordan Curve Theorem

The Jordan curve theorem states that any simple closed curve C divides the points of the plane not on C into two distinct domains (with no points in common) of which C is the common boundary. (wikipedia n.d.)

To apply this to point-in-polygon, we run a semi-infinite ray horizontally (increasing x, fixed y) out from the test point, and count how many edges it crosses. At each crossing, the ray switches between inside and outside. This is called the Jordan curve theorem. (Franklin) If it crosses odd number of edges it is inside the polygon, otherwise if even it is outside the polygon.

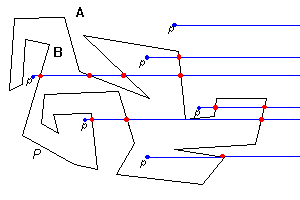
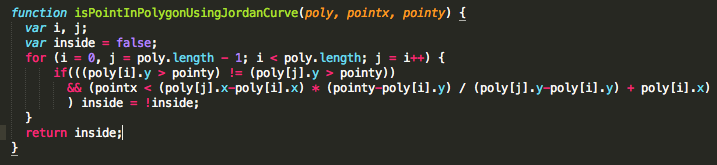


Figure : Jordan Curve Theorem

The Jordan Curve Theorem is implemented as a JavaScript function in pointInPolygon.js. The function is defined as “isPointInPolygonUsingJordanCurve” and is detailed below. Sample code to invoked this function can be found in PointInPolygon.html.



### GPS Coordinates

To test the point-in-polygon algorithms, we created a PointInPolygon.html test page that can be viewed at <http://localhost:8000/PointInPolygon.html>. The test page contains Room 110 and Room 118 GPS coordinates (fig. CMR1). Using Google Maps, we were able to determine the northeast, southeast, southwest and northwest corners of the room. We also defaulted the “Enter X & Y” coordinates to the center of the room so when “Is Point In Room 110” is clicked, the results will be true (fig. CMR2) The user can also enter different GPS coordinates such as other points in and outside the room (geofence) to determine if the algorithms are accurate.

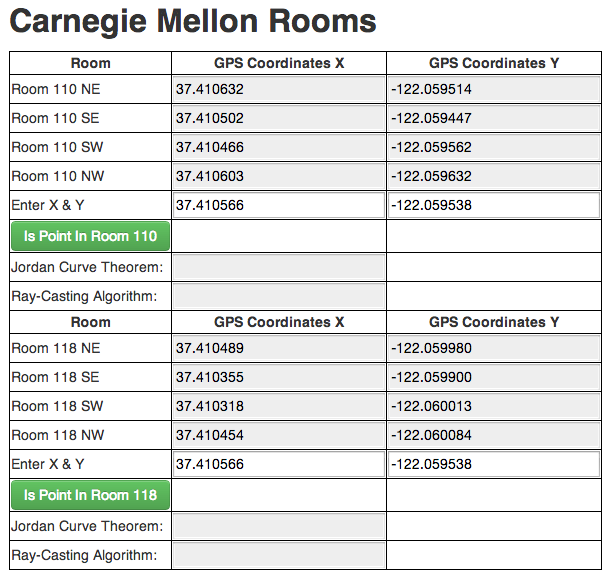


Figure : CMR1 Room 110 and Room 118 GPS Coordinates

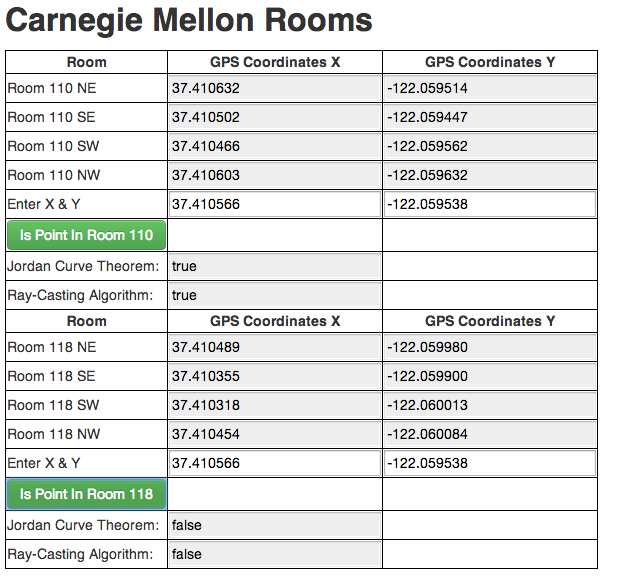


Figure : Algorithm Results

In addition to GPS coordinates testing, a simple polygon was also created where the user can enter X and Y coordinates to test both the Jordan Curve Theorem and Ray-Casting Algorithms. (Figure XY) The X and Y coordinates entered by the user is displayed as a small blue box on the canvas.

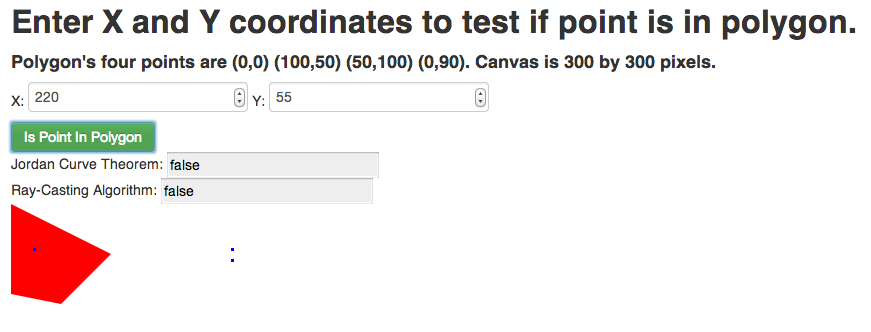


Figure : Test Simple Polygon

# Source Code

The source code of this application is available on Github. <https://github.com/johnleee/VirtualSensorEditor>

# Future Work

Team Facade’s recommendations for future work address interoperability and reliability quality attributes. The future recommendations for this project are to support seamless integration with Taverna and the Sensor Data platform. We also propose to improve the reliability and quality of the system by implementing a more robust unit-testing framework using Jasmine.

## Taverna Plugin for running JavaScript in Java using Rhino

As described in a previous section, during the conversion of JSON to XML, Team Facade encountered a technical issue as the VSE is written in JavaScript and the Taverna system is written in Java. Therefore the functions defined in VSE were unable to run in Taverna. To resolve this we propose creating a Taverna Plugin that invokes Rhino.

Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end-users. It is embedded in J2SE 6 as the default Java scripting engine. Rhino converts JavaScript scripts into classes and is intended to be used in server-side applications such as Taverna. (Mgjbot, et al. 2009)

Taverna Plugin

<http://dev.mygrid.org.uk/wiki/display/developer/Creating+plugins+for+Taverna+2>

Rhino

<https://developer.mozilla.org/en-US/docs/Rhino>

## Unit Testing with Jasmine

For testing the point-in-polygon JavaScript functions, a static html page with predefined GPS coordinates was created for our proof of concept. But ideally a robust test suite where multiple room and GPS coordinates can be run automatically tested is needed. We propose to test the point-in-polygon algorithms using the Jasmine test framework.

Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests. (Pivotal Labs n.d.)

## Integration with rest of Sensor Data Platform

The visualization dashboard and VSE are components of the overall Sensor Data Platform. There is also another team is working on enhancements to the VSE in parallel. As they were not in a state where their work and our work can be integrated, in the future our VSE fork will need to be merged with their repository.

References

Data & Object Factory, LLC. *PatternStrategy.aspx.* LLC. Data & Object Factory. 2013. http://www.dofactory.com/Patterns/PatternStrategy.aspx (accessed 12 8, 2013).

Hong, K. *observer.php.* 2013. http://www.bogotobogo.com/DesignPatterns/observer.php (accessed 12 8, 2103).

Mgjbot, Nickolay Ponomarev, Janet Swisher, and Norrisboyd. *Rhino.* 08 9, 2009. https://developer.mozilla.org/en-US/docs/Rhino (accessed 12 8, 2013).

Pivotal Labs. *jasmine.* http://pivotal.github.io/jasmine/ (accessed 12 8, 2013).

*Taverna Workflow Management System.* University of Manchester, UK. School of Computer Science. 2009. http://www.taverna.org.uk/ (accessed 12 8, 2013).

"VirtualSensorEditor-Presentation-07262013.pptx."

wikipedia. *Jordan\_curve\_theorem.* http://en.wikipedia.org/wiki/Jordan\_curve\_theorem (accessed 12 8, 2013).

Wikipedia. *Point\_in\_polygon.* 12 7, 2013. http://en.wikipedia.org/wiki/Point\_in\_polygon (accessed 12 8, 2013).