Philips Technical Report, April 30, 2011.

Rapid-Prototyping Control Implementation using the Building Controls Virtual Test Bed

Yao-Jung Wen

Philips Research North America, 345 Scarborough Road, Briarcliff Manor, NY 10510
yao-jung.wen@philips.com

Abstract

This report documents the development of a rapid-prototyping control framework based on the Building Controls Virtual Test Bed co-simulation software. The objective of the developed framework is to establish the separation between the control algorithm and the physical systems such that the control algorithm can be rapidly revised and implemented without having to physically swap the controllers. The corresponding protocols and interfaces are designed for maximal flexibility, easy generalization and straightforward implementation. An instance of such control framework has been realized in the Advanced Windows Testing Facility at the Lawrence Berkeley National Laboratory and is used as a case study throughout this report.

Keyword: control; closed-loop; co-simulation; Building Control Virtual Test Bed; rapid-prototyping.

1. Introduction

This document reports the development of a rapid-prototyping implementation framework for the control of building systems and components. The framework was realized in the Advanced Windows Testing Facility at the Lawrence Berkeley National Laboratory for testing integrated electric lighting and venetian blind control algorithms developed under the Philips-LBNL partnership. The experiences and lessons learned are also documented in this report.

1.1. Objective

The objective of the rapid-prototyping control framework is to create a platform for testing and tuning control algorithms on building systems or components in the most efficient and the least intrusive manner. This framework should easily allow multiple users to implement different controls on the same hardware setup at the designated time slot without physically plugging in or unplugging their controllers. The control algorithms can even be implemented remotely and communicate with the systems through the Internet.

1.2. Problem description

This work was motivated by the need of using the Advanced Windows Testing Facility for evaluating the control algorithms developed under the Philips-LBNL partnership without disturbing the well-established routines and setups that have already been in place. Scheduled tests run in this particular facility are cycled every few days throughout a solstice-to-solstice period. Sharing the same hardware, including electric lights, venetian blind drivers, sensors, etc. with other test setups, it presents a challenge to physically disconnect and reconnect a customized controller quickly and correctly every time. Moreover, reprogramming a controller in order to populate a revision or correction to a control algorithm may also be tedious. Therefore, a setup for rapidly implementing and switching control algorithms is desirable for running customized control algorithms in such testing facility.

1.3. Solution approach

The Building Controls Virtual Test Bed (BCVTB) developed by the Simulation Group at LBNL was adopted for establishing the rapid-prototyping control platform. BCVTB is essentially a simulation tool that enables co-simulation of multiple domain-specific programs by coordinating and synchronizing runtime data exchange. In this particular application, the hardware setup and the corresponding drivers in the testing facility are treated as a program that has its own input/output. For instance, the input can be the signals to set the electric light level, blind slat angle, etc., and the output may be the task illuminances, temperatures, and so on, acquired by the sensors. In the meantime, the control algorithm is implemented as another program, and BCVTB manages the data exchange between the two programs to establish a complete control loop.

Section 2 of this report provides an overview of BCVTB and the basic idea of exploiting it for real-time control purposes. Section 3 details the development of each component for establishing the BCVTB control platform, including the protocol and interface. The realization of such control framework in the Advanced Windows Testing Facility is described in section 4, and the challenges encountered during the implementation are documented in section 5. Section 6 concludes the work and points out potential future applications.

In the rest of this report, there will be a few terms that are used repetitively. These terms represent very different things although they may seem very similar. The following table lists all the terms upfront along with short descriptions so that the readers can always refer back to this table should any confusion occur.

Table 1 Terms used in the report.

Terms	**Description**
BCVTB control framework	This is the core of this report. It refers to how the BCVTB co-simulation concept is used for physical implementation.
BCVTB configuration	The configuration refers to a diagram representing how the data are routed among all the actors. See Figure 2 and Figure 15 for example.
BCVTB server	This is a server built exclusively for the BCVTB control framework, which connects to the hardware and can be accessed through the Internet. See section 3.1 for details.
BCVTB tag	BCVTB tags are used in the protocol for communicating with the BCVTB server designed exclusively for the BCVTB control framework. See section 3.3 for details and Table 2 for some sampled tags.
BCVTB (surrogate) interface	This interface is a program hosted by the *Simulator* actor, and is created exclusively for the BCVTB control framework. It establishes the connection between the BCVTB server and the *Simulator* actor. The details are described in section 3.4.

1.4. Intended audience

This document is intended to be a reference for establishing a rapid-prototyping control implementation framework. It is for people who are interested in an alternative way of implementing and testing control algorithms. A basic understanding of controls may be required in order to understand the overall concepts introduced herein. This report provides the fundamental knowledge of building such a control platform and uses the realization in the Advanced Windows Testing Facility as an example. However, it is not a tutorial of BCVTB or any control program, and one will still need to acquire the ability of using BCVTB and possibly other script languages before being able to establishing the control platform.

2. Building controls virtual test bed

The Building Controls Virtual Test Bed (BCVTB) is the key element of the rapid-prototyping control implementation platform. This section provides a general description of BCVTB and how it is used for constructing the control platform.

2.1. General idea and description

BCVTB is a software configuration that coordinates and synchronizes multiple simulation programs for runtime data exchange and co-simulation [1]. It is based on Ptolemy II, a software framework supporting actor-oriented design developed at UC Berkeley [2]. BCVTB creates a socket to the program it connects to such that the outputs of the program are sent to BCVTB by writing to the socket while the program reads from the socket to get its inputs from BCVTB. By creating multiple sockets, one to each program, data from the programs can be exchanged through BCVTB. All programs essentially send their outputs to BCVTB and get their inputs from BCVTB. Figure 1 shows the high-level illustration of a BCVTB co-simulation of several programs. The *Simulator*, a type of *Actor* in Ptolemy II, creates a socket to the simulation programs that allows them to read/write data from/to BCVTB.

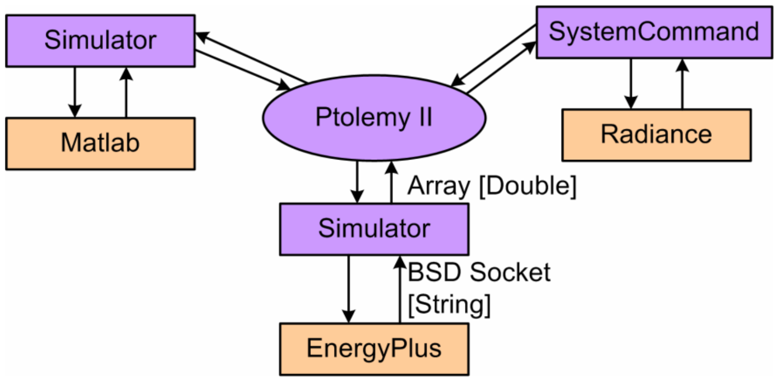


Figure 1 Illustration of a BCVTB co-simulation.

Although the input/output values are certainly meaningful to each co-simulated program, they are treated equally as plain double-precision floating-point numbers in BCVTB. The *BCVTB configuration* determines which number is exchanged among which programs. Figure 2 demonstrates an example of a BCVTB configuration, where three programs are co-simulated. Each of the rectangles with white background is a *Simulator* that hosts a domain-specific program, i.e. creates a socket to that program. Suppose program 1 takes three inputs and generates two outputs; program 2 takes two inputs and generates three outputs; program 3 takes five inputs and generates five outputs. In this particular configuration, the outputs from programs 1 and 2 are read by program 3 as its input through BCVTB, and the first two outputs of program 3 are routed as the inputs of program 2 while the last three outputs are fed into program 1 as its inputs. The slim blue rectangles are vector assemblers and dissemblers in that the data to/from a *Simulator* must be in the form of a vector rather than a scalar.

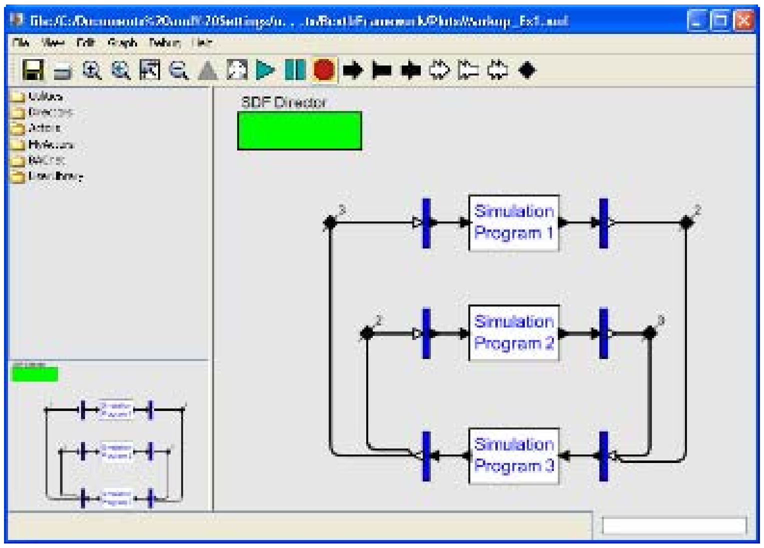


Figure 2 Markup of a BCVTB configuration.

At the beginning of each time step, BCVTB blocks all the co-simulated programs and performs data exchange. Each program sends its outputs to BCVTB and gets its inputs from BCVTB through *Simulator* socket writing and reading respectively. As soon as the data exchange process is completed, BCVTB unblocks the operation of the programs, and each program calculates and generates new outputs during the remaining of the time step based on the newly acquired inputs. The new outputs will then get exchanged at the beginning of the next time step. The same process repeats until the specified simulation time duration is achieved. Figure 3 illustrates the data exchange between two simulation programs through BCVTB. The output of one program becomes the input to the other program; for example, output 1 (*Out 1* in red) from *Program 1* at the first time step becomes the input (*In 2* in red) to *Program 2* at the second time step.

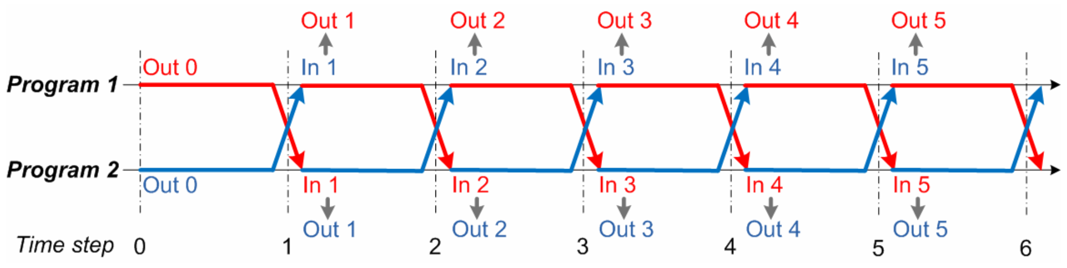


Figure 3 BCVTB data exchange at each time step.

2.2. *Synchronized real-time simulation*

As described in the previous section that data exchange in a BCVTB co-simulation takes place at the beginning of each time step, and the simulation programs return to their normal operation right after that. It is desirable for the programs to run as fast as possible in the simulated virtual world. However, for the implementation of control algorithms, everything must be adhered to real time in order to account for the interactions with the physical world. BCVTB conveniently allows the option of ‘synchronizing to real time’. It will be perfectly synchronized to real time if all data exchange and calculations can be finished within the specified duration of a time step. Should any delay longer than a time step occur due to various reasons, BCVTB will slightly speed up the following few time steps in an attempt to catch up with the real time.

2.3. BCVTB for control implementation

There are quite a few different ways to use BCVTB for control implementation, and the method adopted in the development of the framework reported herein is to use an interface that is capable of creating a socket to the *Simulator* while connecting and communicating with the hardware. This interface is essentially a surrogate that can be hosted by the *Simulator* to form a proper BCVTB configuration. The detail of this interface will be described in the later section.

Figure 4 shows the basic idea of using BCVTB for real-world control implementations. This approach completely separates the controller and the hardware. The controller can be realized in any program that can be hosted by the *Simulator*, and Matlab is used here as an example. The surrogate interface needs to be able to speak in the same language as the hardware in order to relay the data to/from BCVTB through the *Simulator*. In each time step, the surrogate interface gets the sensor readings in the test facility and puts them to BCVTB so that the controller can read in the readings and generates control decisions accordingly. In the meantime, the control commands are sent from the controller to BCVTB, and the surrogate interface picks up the commands and sends to the driver in the test facility to actuate the hardware.

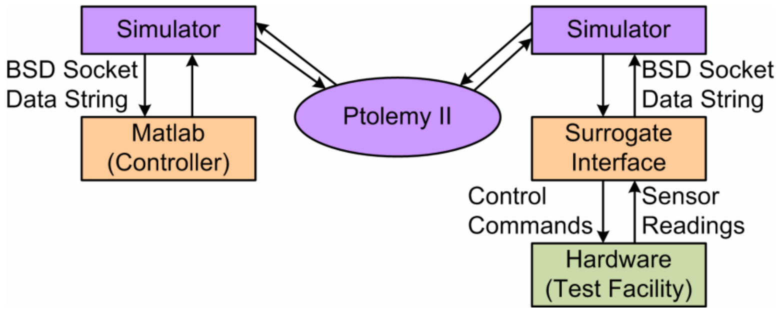


Figure 4 BCVTB control architecture.

3. Retrofit the testing facility for BCVTB control

This section describes the design and implementation of the BCVTB framework in the Advanced Windows Testing Facility. Although this particular realization was specific to the testing facility, the design was meant for being generalized to other instances in a straightforward fashion.

3.1. Design requirements

The Advanced Windows Testing Facility is operated with standalone data acquisition and control infrastructures, both of which are established in LabVIEW environments. Dedicated LabVIEW programs in the data acquisition machine acquire readings from monitoring sensors and record the data on a minute by minute basis. The LabVIEW control programs connect to the control sensors, acquire readings, make control decisions and actuate the hardware. For the purpose of implementing the BCVTB framework, the following requirements were identified.

- Access the sensor readings on the existing data acquisition infrastructure.
- Inject control commands to actuate the hardware drivers on the existing control infrastructure.
- Minimal intrusion to the routines that are already running on the infrastructures.
- Preferably remote access to the facility.
- Maximal flexibility on selecting specific sensors and drivers.
- Accommodate future expansion of the data acquisition and control infrastructure.

In order to access the closed LabVIEW data acquisition and control environment, an additional server machine (designated as the ‘BCVTB server’ thereafter) along with a LabVIEW program were added for interfacing with the outside world through the Internet. Sensor readings are put out on the Internet for external access in response to specific URL requests, and control commands in an established URL format are accepted by the LabVIEW control programs to actuate the hardware. The following subsections describe the design of the components of the BCVTB framework addressing each of the requirements.

3.2. Available information

The information available from the original monitoring and control infrastructure in the Advanced Windows Testing Facility include

- 16 illuminance readings at various locations in each test cell;
- 15 temperature readings at various locations in each test cell;
- 11 wattage-related readings of various equipment in each test cell;
- 2 outdoor temperature readings;
- 14 external solar-related readings;
- 20 control-related readings for each test cell.

There are also spared channels for additional sensors in the future. This is the information that can be put out on the Internet by the BCVTB server.

For each of the test cells in the facility, there are basically two systems that can be controlled, namely dimmable electric lighting and window shading systems. Furthermore, the upper and lower parts of a shading system can be controlled separately if the shades have the capability, and thus the heights (and slat angles for venetian blinds) can be specified separately. In other words at most five control parameters can be specified in each test cell as follows.

- Lower shade (blind) height;
- Lower blind slat angle;
- Upper shade (blind) height;
- Upper blind slat angle;
- Electric light level.

3.3. Protocol

It was determined that getting each single reading one-by-one would be too cumbersome and could easily eat up network bandwidth if accessed through the Internet. Therefore, the readings are classified into a few groups and put out on the Internet as a string, one for each group. Each group is tagged with a 5-letter label. The clients that request the readings are responsible for parsing the string and extracting only the readings of interest. The tags of each group and their associated members are summarized in Table 2.

Table 2 Sensor reading groups.

Tag Name	Members
Luxs**X**	The 16 illuminance readings. **X** represents the test cell number (A, B or C).
Temp**X**	The 15 temperature readings and the two outdoor temperature readings. **X** represents the test cell number (A, B, or C).
Watt**X**	The 11 wattage-related readings. **X** represents the test cell number (A, B or C).
Sun__	The 14 external solar-related readings. Same for each test cell.
Cntl**X**	The 20 control-related readings. **X** represents the test cell number (A, B or C).
Xtra_	Spared channels for additional sensor readings in the future.
All_**X**	All the above readings. **X** represents the test cell number (A, B or C).

To add a layer of security and ownership when accessing the sensor readings from the Internet, a 4-letter password is added to the protocol, and the BCVTB server only responds with the requested string of readings if the password matches the record on file. In short, the URL for requesting a specific string of sensor readings takes the form in Figure 5, where *131.243.168.15* is the IP address of the BCVTB server, which can also be a domain name if available; *7100* is the port number opened for the communication (if different from the conventional HTTP port 80); *pswd* is the 4-letter password, and *Tag__* is one of the 5-letter tags in Table 2.

```
http:// 131.243.168.15:7100/pswd?Tag__
```

Figure 5 Sensor reading request URL format.

The BCVTB server responds to a URL request with a string of sensor readings in the following format (Figure 6), where *Tag__* is the same 5-letter tag in the request, and the readings from the *N* sensors in the specific group are delimited by commas. The last semicolon in the string signifies the end of the group of readings. As mentioned earlier, it is up to the requesting client to extract only the numbers of interest from the group of readings. The purpose of repeating the 5-letter tag in the string is to help the requesting client verify its validity and distinguish among the responses if more than one is expected (in the case of the *All_X* tag). The commas and semicolon should help the client to easily and correctly parse the string into *N* separate readings.

```
Tag__ rdg1,rdg2,rdg3,…,rdgN,;
```

Figure 6 Responding string format for a specific tag (except the All_X tag); rdg stands for "reading".

Notice that there is no mechanism in the protocol for mapping a specific sensor in a group to the order of its reading in the string, and this relationship must be pre-established and agreed upon between the server and client sides.
A special tag, the *All_X* tag in the last row of Table 2, is created when at least one sensor reading in each group of sensors is needed. In this case, the client can get all the readings at once with a single request and simply look for the readings of interest. This, again, relieves the burden on the network and speeds up the process for acquiring necessary readings from the server. The responding string will not start with the tag *All_X* but will return the concatenation of all responding strings for room ***X*** as illustrated in Figure 7.

```
CntlX rdg1,…,rdgN_C,;Lux_X rdg1,…,rdgN_L,;TempX rdg1,…,rdgN_T,;WattX rdg1,…,rdgN_W,;SunX
rdg1,…,rdgN_S,;XtraX rdg1,…,rdgN_X,;
```

Figure 7 Responding string format for All_X tag; rdg stands for "reading".

The protocol for issuing control commands from the client side takes a similar format as the sensor readings requesting URL as shown in Figure 8. The tag *Oper**X*** signifies that it is a command for actuating the system drivers, and *cmd1* through *cmd5* are the control commands for lower shade/blind height, lower blind tilt, upper shade/blind height, upper blind tilt and electric light level in the case of the particular testing facility.

```
http:// 131.243.168.15:7100/pswd?OperXcmd1,cmd2,cmd3,cmd4,cmd5;
```

Figure 8 Control command URL format; cmd stands for "command".

Upon receiving the command, the BCVTB server will echo the command part of the requesting URL back to the client as illustrated in Figure 9. It is up to the client to make use of the acknowledgement from the server for verification or other purposes.

```
OperXcmd1,cmd2,cmd3,cmd4,cmd5;
```

Figure 9 Control command acknowledgement format; cmd stands for "command".

This protocol should provide maximal flexibility for future extension or for being generalized to other projects. More groups of sensors can be added with different tags, more members can be assigned to the same tagging group, and more control commands are also allowed for actuating more systems.

3.4. BCVTB interface

The BCVTB interface is the surrogate interface that communicates with the BCVTB server as a client while connecting to BCVTB through the *Simulator* actor (see section 2.3). Based on the protocol developed in the previous section, the development objective of this BCVTB interface is to ensure maximal flexibility so that it can talk to any BCVTB server that complies with the protocol. In other words, the interface itself should not require any revision in the case of more tags, different tag names, different sensor grouping, or more control commands.
The BCVTB interface is written in Java and packaged into an executable JAR file. The flexibility is provided through a separate XML configuration file. Figure 10 shows an example of the BCVTB control framework and illustrates where the surrogate interface and the associated configuration file fit in. The interface is thus responsible for performing the following three tasks.

- Interpret the configuration file for operational instructions.
- Construct and transmit URL's to the BCVTB server based on the information from its hosting *Simulator*.
- Parse the information from BCVTB server and relay to its hosting *Simulator*.

In addition, a *time step* and *start time* are also required in a BCVTB configuration so that the *Simulators* can be synchronized, properly started and ended. The *start time* and *time step* must be consistent across all *Simulators* and the hosted programs in the same BCVTB configuration. These two values are specified as input arguments to the surrogate interface JAR file.

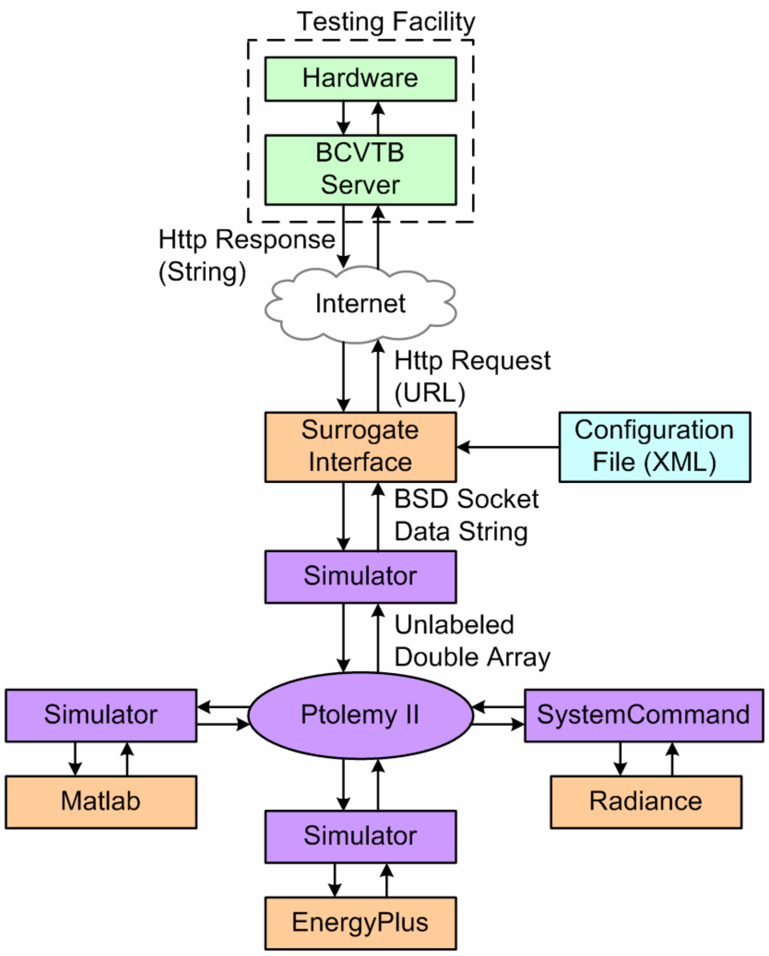
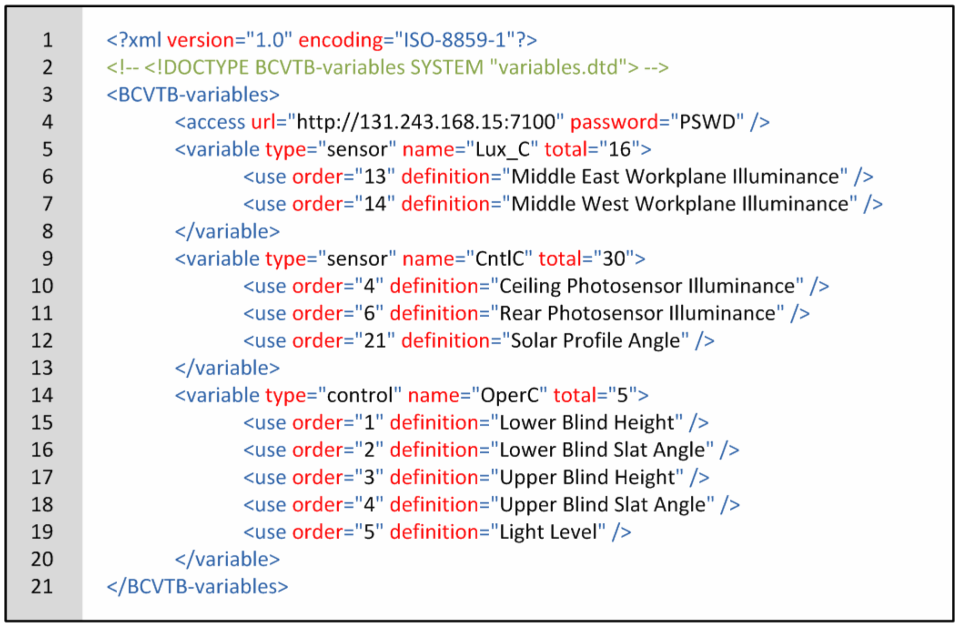


Figure 10 Example of the BCVTB framework with the surrogate interface.

The configuration file is in XML (extensible markup language) format and instructs the interface how to access the BCVTB server, what tags are used, and which specific variables are of interest. It is this configuration file that makes the surrogate interface flexible work with any BCVTB server that follows the protocol described in section 3.3. The file must be named 'variables.cfg' in order for the interface program to locate and interpret it correctly. Figure 11 shows a sample configuration file. Lines 1 and 2 are standard XML declarations and the actual instructions are enclosed in between the <BCVTB-variables> </BCVTB-variables> tags[1] between lines 3 and 21. In this particular configuration file, three tags are used: Lux_C, CntlC and OperC; the former two are for acquiring sensor information and the last one is for sending control commands to actuate the venetian blind and electric lights. These are specified in the *variable* tag (<variable></variable>) in the XML file.

[1]A markup construct enclosed in a pair of angle brackets, e.g. <label> or </label>, is referred to as a tag in XML, which should not be confused with the 'tag' introduced in section 3.3. Therefore, 'BCVTB tag', instead of just 'tag', will be used wherever possible when referring to the tags used in the BCVTB protocol.



```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <!-- <!DOCTYPE BCVTB-variables SYSTEM "variables.dtd"> -->
3   <BCVTB-variables>
4           <access url="http://131.243.168.15:7100" password="PSWD" />
5           <variable type="sensor" name="Lux_C" total="16">
6                   <use order="13" definition="Middle East Workplane Illuminance" />
7                   <use order="14" definition="Middle West Workplane Illuminance" />
8           </variable>
9           <variable type="sensor" name="CntlC" total="30">
10                  <use order="4" definition="Ceiling Photosensor Illuminance" />
11                  <use order="6" definition="Rear Photosensor Illuminance" />
12                  <use order="21" definition="Solar Profile Angle" />
13          </variable>
14          <variable type="control" name="OperC" total="5">
15                  <use order="1" definition="Lower Blind Height" />
16                  <use order="2" definition="Lower Blind Slat Angle" />
17                  <use order="3" definition="Upper Blind Height" />
18                  <use order="4" definition="Upper Blind Slat Angle" />
19                  <use order="5" definition="Light Level" />
20          </variable>
21  </BCVTB-variables>
```

Figure 11 Sample configuration file (variable.cfg).

Starting from line 4, the *access* tag contains the information about the BCVTB server. The *url* attribute specifies the IP address and port number (if different from standard HTTP port 80) for establishing the connection to the BCVTB server. The *password* attribute stores the password required to access the BCVTB server. Lines 5 to 8 specify how the BCVTB tag *Lux_C* should be used. The *type* attribute in the *variable* tag indicates whether the value in the *name* attribute is a BCVTB tag for acquiring sensor readings or for sending control command. Only two values are recognized as the *type*: *sensor* for sensor readings or *control* for control commands. The *total* attribute must be equal to the total number of readings/commands attached to the BCVTB tag. In this case, *Lux_C* is a BCVTB tag for acquiring light sensor readings, and there are 16 sensor readings attached to this particular BCVTB tag, i.e. N=16 in Figure 6. However, in this example, not all 16 readings are needed for the application, but only the 13th and 14th readings are of interest. This is specified in the *use* tag in lines 6 and 7. The *order* attribute indicates the order of the variable of interest in the readings string following the BCVTB tag. The *definition* attribute is not actually interpreted by the surrogate interface but is there for users' convenience to note the name or purpose of the particular reading. Lines 9 to 13, similar to lines 5 to 8, contain information on how the BCVTB tag *CntlC* should be used. Lines 14 to 19 are also very similar to that just described, but this section is for sending control commands as the value in the *type* attribute suggests. A critical difference is that for a *control* type, the values attached to the corresponding BCVTB tag cannot be selectively omitted like the case of *sensor* type. In other words, the number of *use* tags must be equal to the value in the *total* attribute. This is because there is no way for the surrogate interface to anticipate a missing control command and still assemble a correct command string to send to the BCVTB server. It should become clear by now that the BCVTB tag and the number of readings attached to it can be quite arbitrary as long as they are correctly specified in the configuration file, and the BCVTB tags mentioned in Table 2 are merely a sample instantiation.

Another important functionality of the surrogate BCVTB interface is to create a socket to the *Simulator* so that the data can be exchanged through Ptolemy II, the core of BCVTB. This is realized following the message format specified by the *Simulator* [1].

Using the information in the configuration file, the surrogate interface assembles the HTTP requests in the form of Figure 5 and sends it to the BCVTB server for acquiring sensor readings. In the example in Figure 11, two URLs will be assembled and sent as shown in the first two rows in
Figure 12. The BCVTB server will in turn respond with two reading strings corresponding to the *Lux_C* and *CntlC* tags back to the interface like the ones in the 3rd and 4th row in
Figure 12. Upon receiving the two responses, the interface parses and extracts only the readings of interest and assembles them into a string in the message format as illustrated in the 5th row in

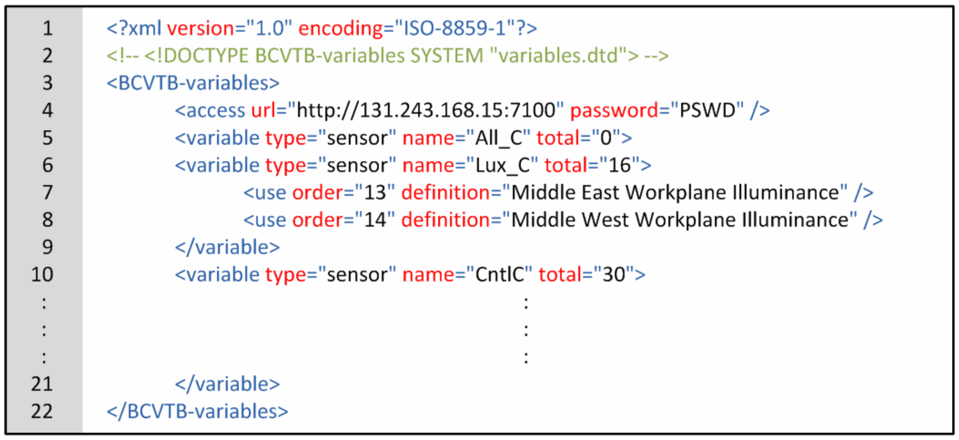
Figure 12, where the first five numbers are the designated header for the message string. The string is then written to the socket established between the interface and the *Simulator*. The *Simulator* subsequently transforms the received string message into an array of double precision floating-point numbers, as shown in the 6th row in
Figure 12, and puts them to Ptolemy II. These values are then routed to other co-simulated programs according to the BCVTB configuration such as the one in Figure 2 for making control decisions.

The control decisions generated by other co-simulated programs are routed back to the *Simulator* in the BCVTB configuration as an array of double precision floating-point numbers (7th row in
Figure 12). The *Simulator* transforms the numbers in the array into a string message with a prescribed header as illustrated in the 8th row in
Figure 12. The surrogate interface reads the string message from the established socket. A control URL, last row in Figure 12, is then assembled by the interface based on the information contained in the configuration file and sent to the BCVTB server to actuate the hardware.

HTTP requests from the surrogate interface to the BCVTB server	http:// 131.243.168.15:7100/PSWD?Lux_C
	http:// 131.243.168.15:7100/PSWD?CntlC
HTTP responses from the BCVTB server to the surrogate interface	Lux_C rdg_L1,rdg_L2,rdg_L3,…,rdg_L16,;
	CntlC rdg_C1,rdg_C2,rdg_C3,…,rdg_C30,;
A string of readings from the surrogate interface to the Simulator	1 0 5 0 0 rdg_L13 rdg_L14 rdg_C4 rdg_C6 rdg_C21
An array of double values from the Simulator to Ptolemy II	[rdg_L13 rdg_L14 rdg_C4 rdg_C6 rdg_C21]
An array of double values from Ptolemy II to the Simulator	[cmd1 cmd2 cmd3 cmd4 cmd5]
A string of readings from the simulator to the surrogate interface	1 0 5 0 0 cmd1 cmd2 cmd3 cmd4 cmd5
HTTP request from the surrogate interface to the BCVTB server	http:// 131.243.168.15:7100/PSWD?OperCcmd1,cmd2,cmd3,cmd4,cmd5;

Figure 12 Sample data processing in the surrogate interface.

Since the communication between the interface and the BCVTB server is through the Internet, the computer running BCVTB and the corresponding co-simulated programs does not need to physically be in the testing facility or in the same place as the BCVTB server. This also fulfills one of the design requirements identified in section 3.1. Special attention should be given to the BCVTB tag *All_X* as an exception has been implemented should this specific tag be used for the reason explained in section 3.3. Figure 13 shows a slightly different configuration file compared to that in
Figure 12 but with the same functionality. Notice that the *All_C* tag is specified in line 5 with a value of 0 for the *total* attribute. When the surrogate interface sees this line, it will only generate a single HTTP request using the *All_C* tag to acquire all sensor readings from the BCVTB server. The interface subsequently uses the rest of the information in the configuration file to process the returned readings in a similar manner as described before.



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- <!DOCTYPE BCVTB-variables SYSTEM "variables.dtd"> -->
<BCVTB-variables>
        <access url="http://131.243.168.15:7100" password="PSWD" />
        <variable type="sensor" name="All_C" total="0">
        <variable type="sensor" name="Lux_C" total="16">
                <use order="13" definition="Middle East Workplane Illuminance" />
                <use order="14" definition="Middle West Workplane Illuminance" />
        </variable>
        <variable type="sensor" name="CntlC" total="30">
                                            :
                                            :
                                            :
        </variable>
</BCVTB-variables>
```

Figure 13 Sample configuration file (variables.cfg) including the All_C tag.

4. Control algorithm implementation

Based on the development in section 3, this section describes a specific instantiation of such a BCVTB control application that has been implemented and tested in the Advanced Windows Testing Facility.

4.1. BCVTB configuration and setup

The main objective for the implemented application was to regulate a constant task illuminance in the testing room with a combined control of electric light and venetian blind, hence daylight. The task illuminance certainly should not be too low so as to impair the performance of visual tasks. On the other hand, glare and unnecessary energy consumption may also accompany excessive levels of task illuminance. In particular, the control algorithm was designed to maintain the task illuminance at 500 lux[2].

The dimmable electric lights that have originally been in place were utilized as the source of artificial light. An interior venetian blind with 2-inch slat and matte white color was connected to the blind driver for regulating admitted daylight. The block diagram in Figure 14 shows the actual setup of the BCVTB control framework in the Advanced Windows Testing Facility. Blue arrows in the diagram indicate the sensor data flow, while red arrows represent the flow of control commands. Blocks enclosed in the green dashed box are the delicate monitoring and control infrastructure that has already been in place for various tests conducted in the facility prior to this implementation. A dedicated LabVIEW program interfaces the newly created BCVTB server with the existing infrastructure. As mentioned before, several sensors that are not originally deployed in the existing infrastructure were added to an extra channel, tagged *Xtra_*, and the BCVTB server directly acquires those sensor readings without going through the data acquisition machine. This is for the purpose of not disturbing the original data acquisition setup in the facility.

[2] The task illuminance in this test was referred to and assessed against the two photometers at desk height (30 inches) close to the two desks, one of which was as against the west wall and the other was against the east wall.

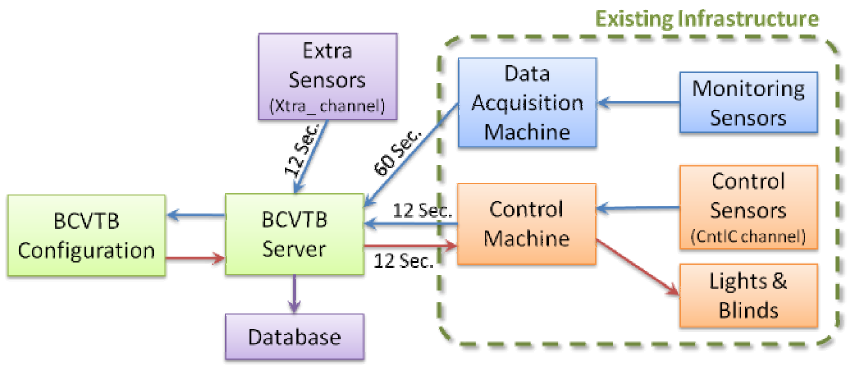


Figure 14 BCVTB control setup in the testing facility.

In this particular setup, the control and extra data (sensor data attached to the CntlC and Xtra_ tags in Table 2) are updated every 12 seconds, and the corresponding data strings stored on the BCVTB server are updated accordingly. In the mean time, the latest control commands from the BCVTB surrogate interface will only be sent to the control machine for execution at the multiple of 12 seconds according to the server's clock no matter when they were actually received. The other monitoring sensor data (sensor readings attached to other tags in Table 2) are acquired every 60 seconds by a dedicated data acquisition machine as established before the BCVTB framework was implemented, and the corresponding data strings on the BCVTB server are then updated in accordance.
In addition, a database was commissioned, as shown in Figure 14, to store all the control and sensor data for future analysis. A standalone Java program, also packaged into a JAR file, was created to query the BCVTB server for sensor data in the same way as the BCVTB interface, i.e. using the URL in Figure 5 with the *All_X* tag, and store the data into the database.

Figure 15 shows the actual BCVTB configuration for this implementation that realizes the *BCVTB configuration* block in Figure 14. Two *Simulator* actors are used for co-simulation of Matlab and the BCVTB surrogate interface. The *Matlab* block is a *Simulator* hosting Matlab, which runs the control algorithm script. The other *Simulator*, the *LabVIEW* block, hosts the BCVTB interface that communicates with the BCVTB server for sensor readings and hardware actuation commands. The time step of this configuration is set to be 12 seconds, same as the minimal data update rate at the BCVTB server. For the convenience of runtime visualization, the sensor readings output from the BCVTB interface are displayed in a separate window using a *Display* actor (the *sensor readings* block). Similarly, the control decisions calculated by the Matlab script are also shown in a separate window by another *Display* actor (the Control Values block). The blocks at the bottom of Figure 15 show two time information; the 'Time in Simulation' block carries the theoretical time passed since the program starts, i.e. number of complete time steps multiplied by the duration of a time step (12 seconds), and the 'Execution Time' block shows the actual time that has elapsed. Notice that the 'Execution Time' should always be greater or equal to the 'Time in Simulation' due to real-world delays caused by computation, network, etc.

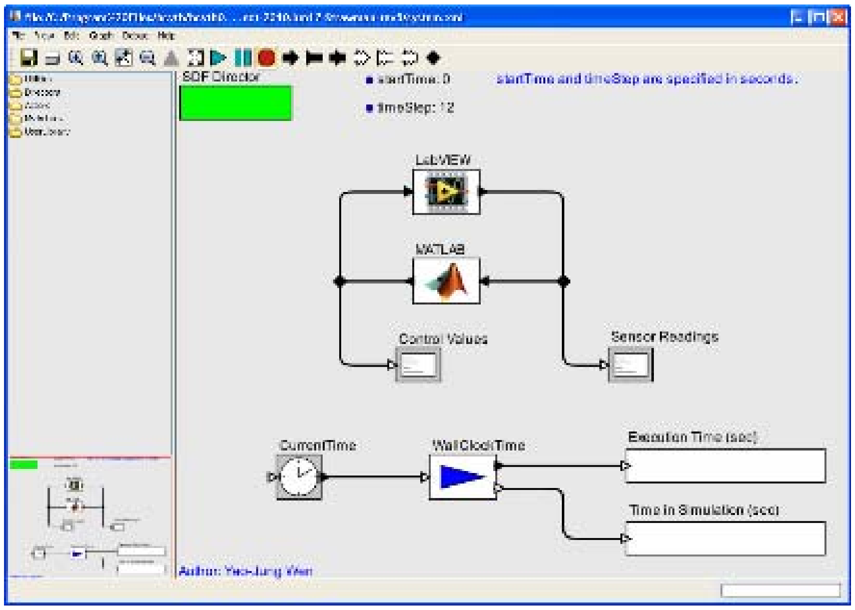


Figure 15 BCVTB configuration for actual implementation in the facility.

4.2. Closed-loop control

The control system realized using the BCVTB framework was implemented in a closed-loop fashion. The benefits of closed-loop control are two-fold: the control decisions can be made based on the outcomes of previous decisions, and the requirements on hardware resolution and precision can be relieved. Take venetian blinds for instance, it will be extremely difficult, if not impossible, to tilt the slats to the exact desired angle due to the difference in motor, tilting mechanism, etc., intrinsic to each individual blind. Furthermore, it can be even more challenging or computationally expensive to figure out the exact amount of daylight that will be admitted into the space at a given slat angle.

The most important feedback information was the ceiling-mounted control photosensor readings. Based on the readings from the particular sensor, the control algorithm in Matlab determined how to actuate the electric lights and the venetian blind slats to maintain the 500 lux task illuminance. The exterior global and direct illuminance readings were also incorporated to help the control algorithm make decisions on deploying/retracting the blind.
Other than the above functional feedback, several operational feedbacks were also found to be crucial in order for a successful feedback lighting control system with short time step, such as a time step less than one minute. There will always be discrepancies between the time when the control commands were sent to the BCVTB server and the time when the commands were actually executed. The delay, as will be discussed in depth in the next section, could span the duration of several time steps in the worst-case scenario. Therefore, blindly making control decisions without knowing that the previous commands have been in effect could easily cause the system to oscillate or even become unstable. Therefore, the dimming ballast voltage and the venetian blind slat angle on the control sensor channel (data with *CntlC* tag) were used as the operational feedbacks. The Matlab control algorithm will keep track of the previous commands, compare to the operational feedback data, and only calculate and issue new commands after the previous ones have been executed.

5. Potential challenges

Several potential challenges have been identified during the process of implementing the BCVTB control framework at the Advanced Windows Testing Facility. This section will discuss the challenges in detail and recommend possible improvements.

5.1. *Latency*

Latency can be an issue when the BCVTB time step is too close to the data acquiring/updating frequency. As hinted in the previous section, there are several sources of delays: the delay caused by the BCVTB server implementing commands, unsynchronized/staggered timing for sensor data acquisition and network delay. In this particular implementation, the BCVTB server only executes the commands every 12 seconds according to its own clock. In other words, control commands received at any time in between will not be in effect until next multiple of 12 seconds. For example, a command received by the server at the 13th second will be executed at the 24th second, which then results in a delay of 11 seconds.

The unsynchronized/staggered timing for sensor data acquisition may also cause latency. It is a very practical and inevitable problem since each single data acquisition process requires a little amount of time, and thus it is impossible to acquire all sensor data simultaneously. It is potentially more challenging to synchronize the data acquisition time with the control command execution cycle. For both the functional and operational feedback described in section 4.2, the control algorithm may still see the old sensor data from the BCVTB server if an updated reading has not been acquired even though the control command has been executed. In this case, a new control decision can only be made at the next time step, causing at least one time step delay (12 seconds) on top of other latencies.

Network delay can be the most unpredictable source of delay, especially when the data/commands have to go through several routers between the control machine and the BCVTB server. During the testing period, a delay of several seconds was commonly observed. Notice that these delays actually happened even though both machines were on the LBNL Ethernet without many intermediate routers in between.

It is recommended that the 12-second updating cycle of the BCVTB server be reduced, if possible, in order to respond to control commands more promptly. One possible way to mitigate the delay caused by unsynchronized sensor data acquisition timing could be to make the control sensor data acquisitions follow the execution of control commands. In other words, the BCVTB server should acquire the control sensor data immediately after it executes the latest control commands (or with a slight delay to allow the hardware to settle).

5.2. *Control acknowledgement*

This issue is closely related to the latency issue discussed above. The acknowledgement from the BCVTB server upon receiving control commands does not imply that the commands have been executed. In the current setup, when a control command is issued from the BCVTB interface (Figure 8), the BCVTB server echoes back the command string (Figure 9) immediately as a means for acknowledging a successful network communication. This mechanism, however, does not give the client any information about whether the commands have been in effect. As a result, a separately operational feedback as described in section 4.2 has to be in place so as to indicate the timing and successfulness of executing the control commands.

For future implementation, the BCVTB server acknowledgement mechanism should include both network communication and command execution. The most straightforward way would be for the BCVTB server to only echo back the received commands or send back any indicative message after the commands have been executed. This approach could, however, result in hanging of the BCVTB operation waiting for the acknowledgement from the server, and the duration of hanging depends on the execution cycle as discussed in previous subsection.

5.3. *Constrained time step*

The time step of the BCVTB control framework is constrained by the updating and execution cycle of the BCVTB server. In the particular implementation in the Advanced Windows Testing Facility, the time step cannot be smaller than the 12-second BCVTB server updating/execution cycle. This constraint should not be an issue for systems that do not need to be actuated frequently, such as the venetian blinds. However, for the electric lights to perform daylight dimming, it is desirable to dim the light continuously and smoothly in response to available daylight. A smooth dimming could not be achieved with a minimum interval of 12 seconds although the electric lighting system does possess such capability and resolution. As a result, electric lighting dimming was divided into 20 steps with a 5% change in light output each step increment/decrement. Therefore, the response of electric light to daylight variation was slow, especially when the latencies mentioned in section 5.1 were added on top of the relatively large time step.

5.4. Restart with prescribed initial condition

An initial condition has to be prescribed in each co-simulated program for BCVTB to start the first data exchange at time step 0. The state of final simulation results do not carry over to a new BCVTB simulation. When a BCVTB configuration is used for pure simulation, it makes perfect sense to start with certain initial condition since each simulation instance will be a complete run and the effects of initial conditions can eventually be negligible. For the BCVTB control framework, this requirement could pose a potential inconvenience. During a day-long control performance testing, many things can go wrong unexpectedly and force the tests to be temporarily stopped, including loss/interruption of network connection, flaws in the control algorithm, problems with hardware and computers, etc. While most of the issues can be corrected within minutes, BCVTB control always has to be restarted with the prescribed initial condition (electric lights off and blind retracted in this particular implementation) instead of being resumed from where it was stopped.

One possible solution would be to add a mechanism in the control algorithm, the Matlab script in this particular case, to record the last status. The control algorithm can create and update a separate file every time new sensor readings are received and new control commands are sent. This static file will last even when the control script is stopped or closed. When a new BCVTB run is started, the control algorithm will then have the option to initialize itself with the last state by reading the status file.

6. Conclusions

A control framework built on the Building Controls Virtual Test Bed has been established and implemented in the Advanced Windows Testing Facility at the Lawrence Berkeley National Laboratory. The BCVTB, originally developed for co-simulation of domain-specific programs, was utilized as a means for rapid-prototyping control design and testing with physical systems. The corresponding protocol and interface developed makes this framework flexible for easy implementation. A 6-month test of an integrated electric lighting and venetian blind control algorithm has been successfully realized and tested in the facility leveraging this framework, and a detailed data analysis assessing the control performance is documented in a separate report.

During the testing and implementation of this BCVTB control framework, several potential challenges have also been identified, including latency, insufficient acknowledging mechanism, constrained time step and fixed initial conditions. These challenges serve as considerations for further improvements in the future.

While Matlab was the only co-simulated program running the control algorithm in the Advanced Windows Testing Facility implementation, other domain-specific program can potentially be used with this BCVTB control framework for more versatile building energy management. For example, EnergyPlus can be added for co-simulation under the control framework. The sensor readings from the facility may be used to real-time calibrate the EnergyPlus building model. On the other hand, the control commands can simultaneously go into both the facility and a calibrated EnergyPlus model, and the resulting condition in the real and the virtual buildings can be compared for fault detection and other continuous commissioning purposes.

Acknowledgement

We would like to acknowledge Mr. Dennis DiBartolomeo for his significant contribution on building the BCVTB server and the corresponding LabVIEW interfacing with the existing sensing and control infrastructures. He also provided valuable suggestions and insight into the development of the BCVTB rapid-prototyping control framework. We also thank Dr. Michael Wetter, the original developer of BCVTB, for his consultancy during the development of this work. We are also grateful for Ms. Eleanor Lee’s support on implementing the developed framework in the Advanced Windows Testing Facility.

References

[1] M. Wetter, Building Controls Virtual Test Bed. [Computer software]. Available at https://gaia.lbl.gov/bcvtb.

[2] J. Eker, J. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, 91(1), 2003, pp. 127-144.