Co-Simulation of Building Energy and Control Systems with the Building Controls Virtual Test Bed

Michael Wetter

Lawrence Berkeley National Laboratory

August 2011

DISCLAIMER

Published in *Journal of Building Performance Simulation*, 4(3):185-203, 2011.

Co-Simulation of Building Energy and Control Systems with the Building Controls Virtual Test Bed

Michael Wetter
Lawrence Berkeley National Laboratory
Environmental Energy Technologies Division
Building Technologies Department
Berkeley, CA 94720

August 31, 2011

Abstract

This paper describes the implementation of the Building Controls Virtual Test Bed (BCVTB). The BCVTB is a software environment that allows connecting different simulation programs to exchange data during the time integration, and that allows conducting hardware in the loop simulation. The software architecture is a modular design based on Ptolemy II, a software environment for design and analysis of heterogeneous systems. Ptolemy II provides a graphical model building environment, synchronizes the exchanged data and visualizes the system evolution during run-time. The BCVTB provides additions to Ptolemy II that allow the run-time coupling of different simulation programs for data exchange, including EnergyPlus, MATLAB, Simulink and the Modelica modeling and simulation environment Dymola. The additions also allow executing system commands, such as a script that executes a Radiance simulation.

In this paper, the software architecture is presented and the mathematical model used to implement the co-simulation is discussed. The simulation program interface that the BCVTB provides is explained. The paper concludes by presenting applications in which different state of the art simulation programs are linked for run-time data exchange. This link allows the use of the simulation program that is best suited for the particular problem to model building heat transfer, HVAC system dynamics and control algorithms, and to compute a solution to the coupled problem using co-simulation.

1 Introduction

This paper presents the Building Controls Virtual Test Bed (BCVTB), which is a software environment for co-simulation and for real-time simulation. By co-simulation, we mean applications in which at least two simulators, each solving an initial-value differential or difference equation, are coupled to exchange data that depend on state variables. By real-time simulation, we mean applications in which the simulation time is synchronized to the wall clock time, and the simulator may be coupled to hardware.

The BCVTB is a modular, extensible open-source software platform available from Lawrence Berkeley National Laboratory that allows designers, engineers and researchers of building energy and control systems to interface different simulation programs with each other. Interfaces to Building Automation Systems (BAS) are also available but will not be discussed in this paper. The intention of the BCVTB is to give users the option to use the simulation programs that are best suited to model various aspects of building energy and control systems, or to use programs in which they already acquired significant expertise. The BCVTB exchanges data between these programs as the simulation progresses, for example to close feedback control loops or to solve coupled differential equations. Typical applications in building controls include the performance assessment of integrated building energy and control systems, either through co-simulation or real-time simulation, the development of new control algorithms, the testing of control hardware and software in an emulated environment, and the simulation-based verification of control algorithms prior to deployment in a building. Applications in energy systems include the extension of the simulation capability of building simulation programs such as EnergyPlus [Crawley et al., 2001] through the use of more modern languages such as Modelica [Fritzson and Engelson, 1998] that allow rapid virtual prototyping of new components and systems, integrating models from different domains such as electrical systems, thermal systems and control systems and generation of code that can be uploaded to control hardware [Wetter, 2009a; Elmqvist et al., 2009]. While the use of co-simulation allows extending the capabilities of different domain-specific simulation programs through the run-time coupling with other simulation programs, a direct implementation of all equations in one simulator may be favorable for reasons of computing time and ease-of-use. This is particularly the case if the rate of change in the exchanged data varies significantly during the simulation, i.e, if the coupled system of differential equations is stiff. However, the work for such an implementation can be enormous and may not be practical to do for an individual simulation project.

Different building energy analysis programs have also been coupled by several others [Hensen, 1999; Lam et al., 2002; Trčka et al., 2006; Zhai and Chen, 2005; Trčka et al., 2007]. In most cases, the coupling is done by a direct link between two simulators. The software architecture of the BCVTB differs from the above approaches in that it uses a modular middleware to couple any number of simulation programs, instead of coupling two simulators directly. The coupling can be done either locally on one host computer, or remotely over the internet, possibly using different operating systems. The middleware allows users to couple simulators and control interfaces graphically, and it provides

a library so users can add their own system models directly within the middleware. Such system models can be used to model control algorithms, physical systems (such as HVAC systems) or communication networks, using different models of computation, such as synchronous data flow, continuous time, discrete time and finite state machine. It also provides models for data processing, such as output analysis, online visualization and reporting. The middleware can simulate systems as fast as possible, synchronized to real-time or at any speed in between. One of the design goals of the BCVTB was to provide to users a platform that allows them to link their own simulation program or control interface. This has been accomplished by providing libraries that can be used to integrate other simulation programs.

The BCVTB does not attempt to solve the problems of data and process interoperability [Augenbroe et al., 2004; Bazjanac, 2004; Lam et al., 2004], but it provides a modular software framework that can be used to interface different simulation tools and BAS for run-time data exchange. Earlier work conducted by the Lawrence Berkeley National Laboratory led to a prototype link between building control systems and EnergyPlus, using SPARK models embedded in EnergyPlus to communicate with the control system via digital/analog converters [Haves and Xu, 2007]. The BCVTB presented here is a complete redesign with a different software architecture.

Significant work on using simulation to evaluate the performance of local loop and supervisory control has been performed in the International Energy Agency (IEA) Annex 17 [Lebrun, 1992]. Control strategies were implemented in simulation and in real, commercial, control hardware coupled via analog interfaces to building envelope and system simulations [Haves et al., 1998, 1991; Kelly et al., 1991; Laitila et al., 1991; Vaezi-Nejad et al., 1991; Wang et al., 1994; Decious et al., 1997; Wittwer et al., 2001]. The techniques developed in the IEA Annex 17 were further developed in the ASHRAE Research Project 825-RP and resulted in a simulation testbed for control algorithms [Haves et al., 1996]. The US National Institute for Standards and Technology has been developing a Virtual Cybernetic Building Testbed (VCBT) that uses the Common Object Request Broker Architecture (CORBA) and BACnet, to link computer models for building envelope, HVAC systems, fire and smoke propagation to control hardware [Bushby et al., 2010].

Recently, a standardization effort was started to facilitate co-simulation, integration of models into different simulation environments and execution of models on embedded systems: In 2010, the MODELISAR consortium released a first specification of the Functional Mock-up Interface for Model Exchange (FMI), which standardizes the interface of models that contain differential, algebraic and discrete equations [MODELISAR]. Such models can then be encapsulated into a Functional Mock-up Unit (FMU). An FMU may be self-integrating if used for co-simulation, or may require the simulator to perform the numerical integration. This effort is expected to further facilitate the development and interoperability of tools for co-simulation. However, the BCVTB does not yet support the FMI specification.

2 Nomenclature

2.1 Conventions

1. Subscripts denote time steps in a time series.
2. Superscripts denote elements of vectors.
3. We denote by $y = \lim_{s \uparrow t} x(s)$ the one-sided limit as $s \to t$ with $s < t$.

2.2 Variables

C	heat capacity
$\dot{m}_w$	water mass flow rate
$\dot{Q}_0$	nominal heating power
$\dot{Q}_s$	sensible heat flow rate
$\dot{Q}_l$	latent heat flow rate
T_{out}	outside temperature
T_{set}	setpoint temperature
T	room temperature
x	state variable
X_w	water vapor mass fraction
u	input signal
y	control signal
UA	conductance times surface area
Δt	time step
φ	relative humidity of room air
φ_{out}	relative humidity of outside air
γ	control gain

2.3 Sets

$\mathbb{R}$	real numbers
$\mathbb{N}$	natural numbers

3 Intended Applications and Requirements

The BCVTB allows expert users of simulation to couple different simulation programs to enable an integrated analysis of building systems that require the use of multiple simulation programs because their functionalities complement each other. For example, for a researcher who is interested in assessing the performance of new HVAC system architectures, a typical use case would be to couple an EnergyPlus model of the building envelope with a Modelica simulation environment such as Dymola [Brück et al., 2002] that enables the researcher to modularly build a model of a new HVAC system using the Modelica Buildings library [Wetter, 2009c,b] and to couple the two programs for data exchange during run-time. A control engineer who needs to develop a new supervisory control sequence, such as for an active facade of a naturally ventilated building, may develop a control sequence in MATLAB/Simulink [Mathworks, 2010] and couple MATLAB/Simulink with EnergyPlus through the BCVTB. When developing standard control sequences for HVAC systems, such as the ones published by ASHRAE [ASHRAE, 2006], a Modelica model of an HVAC system that allows detailed controls analysis may be coupled to an EnergyPlus building model to assess and improve the performance of control algorithms in order to reduce energy and increase comfort. If assessment of daylight availability and glare is of interest, such a system may also require the use of Radiance [Ward, 1994] in a feedback loop in which the input to Radiance may be a control signal for the facade and the output from Radiance may be the illuminance measured by a photosensor in the room. To verify that a supervisory control algorithm works as intended, a control technician may couple a BACnet [ASHRAE, 2004] compliant BAS to a Modelica model, which is then used as a virtual representation of the HVAC system that will be controlled by the BAS. This may help reduce commissioning time and eliminate errors in the control sequence. To conduct model-based fault detection and diagnostics (FDD), a simulation model and an FDD algorithm may be coupled to a BAS through the BCVTB, which may store data in a database.

Based on the above use cases and on interviews with stakeholders from industry, we developed the BCVTB to support the following capabilities:

1. The BCVTB allows users to couple different new clients, i.e., a new simulation program, with code modifications required only in the new client.

2. The computing time for data transfer between simulation programs should be small compared to the computing time spent in the individual simulation programs when performing a co-simulation for a whole building.

3. The BCVTB is modular and simulation tool independent so that different clients can be coupled to it. Examples of clients are EnergyPlus, MATLAB, Simulink, simulation environments for Modelica, visualization tools for the online plot of variables, data bases and a BACnet compatible BAS.

4. The BCVTB allows communication with clients over the internet and across different operating systems.

5. The BCVTB runs on Microsoft Windows, Linux and Mac OS X.

6. The BCVTB can be run with a graphical user interface or as a console application without user interaction.

4 Implementation

We implemented the BCVTB using a modular architecture in which a middleware links the different clients. To simplify the discussion, *client* will refer to a simulation program or a BAS. Using a middleware, as opposed to linking clients directly to each other, allows the coupling of an arbitrary number of clients. It also provides a central point for starting the simulation of all clients, establishing the communication channels, synchronizing the simulation time and stopping the clients. Fig. 1 shows the architecture of the BCVTB.

The dotted line marks the middleware that is used to implement the BCVTB. The middleware has a director that fires off each actor at the synchronization time step. The director also organizes the data exchange between the actors. Each actor is responsible for one simulation program. Prior to the simulation, the actor writes a configuration file that specifies how the simulator can connect to the actor. Finally, it fires off a process invocation to start the simulation. It also starts a server that uses the Berkeley Software Distribution socket (BSD socket). The simulation program reads the configuration file and connects to the actor through a BSD socket using TCP/IP. This socket is used to exchange data between the simulator and the actor.

The following sections explain the implementation as follows: Sec. 4.1 explains the middleware, Sec. 4.2 explains the additions to the middleware, Sec. 4.3 discusses the mathematics of the implemented co-simulation, Sec. 4.4 explains the libraries that are available to implement new clients. Sec. 4.5 provides a simple example and Sec. 4.6 discusses the sequence of data exchange. Sec. 4.7 explains the various interfaces that are implemented in the BCVTB to couple different programs.

4.1 Middleware

To implement the middleware, we used the Ptolemy II software [Brooks et al., 2007]. Ptolemy II is a Java-based open-source software framework developed by the University of California at Berkeley to study modeling, simulation and design of concurrent heterogeneous real-time systems. In Ptolemy II, models can be built by instantiating *actors*. Actors encapsulate the action performed on input data to produce output data. Ptolemy II wraps these data into so-called *tokens*, which can have various data types. In the BCVTB, an actor may be a Java class, i.e., a group of Java methods and variables, that communicates during run-time with a simulation program. Its input token may be a control signal to be sent to an actuator in the simulation program, and an output token may be a sensor value received from the simulation program. In Ptolemy II's graphical model editor, actors are encapsulated by graphical icons. To send output tokens to input ports of another actor, a user can draw a connection line between the ports of the actors.

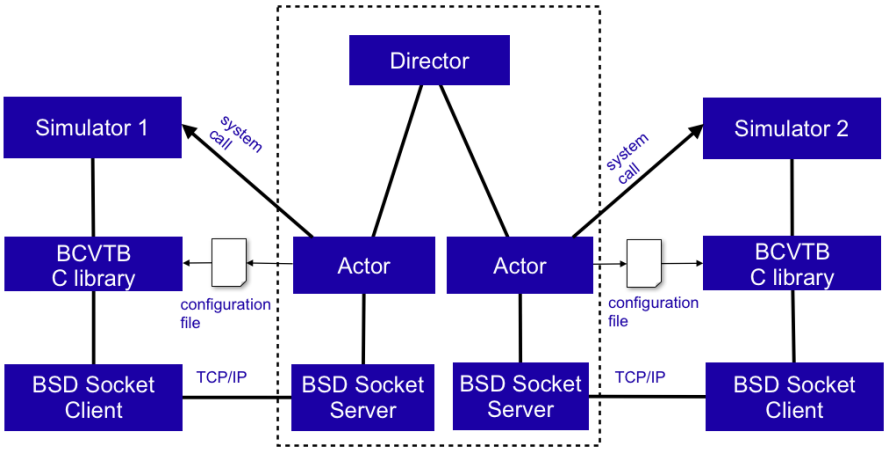


Figure 1: *Architecture of the BCVTB with the middleware shown in the dotted box that connects two clients.*

The interactions between the actors are defined by a *Model of Computation* (MoC). The MoC specifies the communication semantics among ports. Examples that are of particular interest for the BCVTB are *Synchronous Data Flow* (SDF) and *Finite State Machines* (FSM). We use SDF to control the communication of actors that connect to simulation programs. In SDF, each actor is fired when a fixed, pre-specified number of tokens is available on each of its input ports. In the SDF domain, each actor produces a fixed, pre-specified number of output tokens at each firing. Another MoC that is frequently used in the BCVTB is the FSM domain. In FSM, entities are not actors but rather *states*, and the connections among the entities represent transitions between states. FSM are useful for expressing control logic in which different control laws are used depending on the state of the system, such as sequencing control strategies for air-handling units [Seem et al., 1999].

Ptolemy II has a modular extensible structure that allows the use of the BCVTB for other research applications, such as analyzing the effect of lost packets or latency in a communication network on the performance of a building control system, or the integration of optimization algorithms for electricity demand response control.

4.2 Additions to Ptolemy II

To enable co-simulation, we added a new Java package to Ptolemy II. This package contains the actors `Simulator` and `SystemCommand`. It also contains the classes `Server` and `XMLWriter`. The class `Simulator` is an actor that starts a simulation program, sends its input tokens to the simulation program, receives new values from the simulation program and sends these values to its output port. The class `Simulator` makes an instance of the class `Server` for the interprocess communication with the simulation program, which is the client. The interprocess communication is implemented using the Berkeley Software Distribution socket interface (BSD sockets). To connect to the server,

the client needs to know the number of the port on which it needs to connect. To pass the port number from the server to the client, the class `Simulator` writes an XML file using `XMLWriter`, which is then read by the client at program start.

This implementation allows the actor `Simulator` to use a system call to start any executable (such as a batch file on Windows, a shell script in Mac OS X or Linux) or to start directly any executable program that may have been compiled by the user.

The program that is started by the actor `Simulator` keeps running until the co-simulation finishes. In contrast, the actor `SystemCommand` calls a user-specified system command and waits until the command terminates. The system command can be any executable. A typical application where one may prefer to use `SystemCommand` instead of `Simulator` is the use of a program that does not have any state variables and whose start-up time is small compared to its computing time. In this situation, it is sometimes easier to implement the communication through command line arguments and text files instead of integrating the socket interface into the simulation program, and the additional overhead for file I/O may be negligible. The use of Radiance presents such a situation.

4.3 Mathematics of the Implemented Co-Simulation

In the BCVTB, data is exchanged between the different clients using a fixed synchronization time step. There is no iteration between the clients. In the co-simulation literature, this coupling scheme is referred to as *quasi-dynamic coupling*, *loose coupling* or *ping-pong coupling* [Hensen, 1999; Zhai and Chen, 2005].

The algorithm for exchanging data between clients is as follows: Suppose we have a system with two clients, where each client solves an initial value ordinary differential equation that is coupled to the ordinary differential equation of the other client. Let $N \in \mathbb{N}$ denote the number of time steps and let $k \in \{0, \ldots, N\}$ denote the time steps. For some $n_1, n_2 \in \mathbb{N}$, let $f_1 : \mathbb{R}^{n_1} \times \mathbb{R}^{n_2} \to \mathbb{R}^{n_1}$ and $f_2 : \mathbb{R}^{n_2} \times \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$ denote the functions that compute the next value of the state variables in simulator 1 and 2. Note that these functions are defined by the sequence of code instructions executed in the respective simulator. The simulator 1 computes, for $k \in \{0, \ldots, N-1\}$, the sequence

$$x_1(k+1) = f_1(x_1(k), x_2(k)) \tag{1}$$

and, similarly, the simulator 2 computes the sequence

$$x_2(k+1) = f_2(x_2(k), x_1(k)) \tag{2}$$

with initial conditions $x_1(0) = x_{1,0}$ and $x_2(0) = x_{2,0}$. An implementation difficulty is presented by the situation that $f_1(\cdot, \cdot)$ and $f_2(\cdot, \cdot)$ need to know the initial value of the other simulator. Thus, at $k = 0$, both simulators exchange their initial value $x_{1,0}$ and $x_{2,0}$. To advance from time k to $k+1$, each simulator uses its own time integration algorithm. At the end of the time step, the simulator 1 sends the new state $x_1(k+1)$ to the BCVTB and it receives the state $x_2(k+1)$ from the BCVTB. The same procedure is done by the simulator 2. The BCVTB synchronizes the data in such a way that it does not matter which of the two simulators is called first.

In terms of numerical methods for ordinary differential equations, this scheme is identical to an explicit Euler integration, which is an integration algorithm for a differential equation

$$\dot{x} = h(x), \tag{3}$$
$$x(0) = x_0, \tag{4}$$

where $h : \mathbb{R}^n \to \mathbb{R}^n$ for some $n \in \mathbb{N}$. On the time interval $t \in [0,\ 1]$, the explicit Euler integration algorithm leads to the following sequence:

Algorithm 4.1

Step 0: Initialize counter $k = 0$ and number of steps $N \in \mathbb{N}$.
Set initial state $x(k) = x_0$ and set time step $\Delta t = 1/N$.

Step 1: Compute new state $x(k+1) = x(k) + h(x(k))\,\Delta t$.
Replace k by $k+1$.

Step 2: If $k = N$ stop, else go to Step 1.

In the situation where equation (3) is integrated over time using co-simulation, the above algorithm becomes:

Algorithm 4.2

Step 0: Initialize counter $k = 0$ and number of steps $N \in \mathbb{N}$.
Set initial states $x_1(k) = x_{1,0}$ and $x_2(k) = x_{2,0}$.
Set time step $\Delta t = 1/N$.

Step 1: Compute new states
$$x_1(k+1) = x_1(k) + f_1(x_1(k),\, x_2(k))\,\Delta t, \text{ and}$$
$$x_2(k+1) = x_2(k) + f_2(x_2(k),\, x_1(k))\,\Delta t.$$
Replace k by $k+1$.

Step 2: If $k = N$ stop, else go to Step 1.

This algorithm is implemented in the BCVTB. It does not require an iteration between the two simulators. However, one needs to be aware of the sequence of data that is being exchanged, as surprising effects can occur if done incorrectly. The following example illustrates such a situation.

Example 4.3 *Let simulator* 1 *be the EnergyPlus building simulation program, and let simulator* 2 *be MATLAB. Suppose we compute in MATLAB a control signal for the slat angle of a window blind* $y_2(k)$*, send this control signal to EnergyPlus, and send from EnergyPlus to MATLAB the current slat angle* $y_1(k)$*. Since the slat angle in EnergyPlus is adjusted instantaneously, EnergyPlus computes*

$$y_1(k+1) = y_2(k). \tag{5}$$

Suppose the slat angle in MATLAB is computed as

$$y_2(k+1) = y_1(k) + \gamma\, e(k), \tag{6}$$

where $\gamma > 0$ *is a gain that corrects for the control error* $e(k)$*. Suppose that* γ *is small enough so that we can use the approximate equations*

$$y_1(k+1) = y_2(k), \tag{7}$$
$$y_2(k+1) = y_1(k). \tag{8}$$

Then, the slat angle will oscillate according to the sequence shown in Tab. 1. To avoid this situation, equation (6) *can be implemented as*

$$y_2(k+1) = y_2(k) + \gamma\, e(k), \tag{9}$$

i.e., the simulator 2 stores the value of $y_2(\cdot)$ *from one time step to the next, and uses this value to compute the new slat angle.*

k	$y_1(k)$	$y_2(k)$
0	$y_{1,0}$	$y_{2,0}$
1	$y_{2,0}$	$y_{1,0}$
2	$y_{1,0}$	$y_{2,0}$
3	$y_{2,0}$	$y_{1,0}$

Table 1: *Sequence of exchanged actuator positions.*

We note that other data synchronizations may be possible. For example, in *strong coupling*, within each time step, simulators exchange data until a convergence criteria is satisfied. This implementation requires the numerical solution of a nonlinear system of equations in which the termination criteria is a function of the state variables of the coupled simulators. However, many building simulation programs contain solvers that compute with relatively coarse precision. This can introduce significant numerical noise which may cause convergence problems for the co-simulation. The computing time of strong coupling vs. loose coupling of EnergyPlus and TRNSYS [Klein et al., 1976] was compared by Trčka et al. [2007]. Although loose coupling required shorter synchronization time steps, the work per time step was smaller (as no iterations were needed) which caused loose coupling to compute faster than strong coupling. An additional implementation benefit of loose coupling is that state variables need not be reset to previous values. Thus, loose coupling is easier to implement, is numerically more robust and it computed faster in the experiments reported by Trčka et al. [2007]. A potentially interesting approach would be the use of an adaptive synchronization time step, but we have not yet explored this experimentally. An implementation would require resetting state variables, which is hard to accomplish in tools like EnergyPlus that contain half a million lines of code and that do not have a data structure that allows storing and resetting all state variables.

4.4 Library for Clients

We implemented a C library with functions that can be used by developers to implement an interface in their client program to connect to and to communicate with the BCVTB. The library provides functions for parsing XML files using search commands in the XPath language, and it can be used to establish the BSD socket connection, to exchange data through the BSD socket and to close the connection. In the next section, we will show an example that explains how these functions can be used to add new clients to the BCVTB.

For developers, the BCVTB contains configuration files that allow compiling the code using Apache Ant [Foundation, 2010], which is a cross platform build tool. The operating systems that are currently supported are Windows, Mac OS X and Linux.

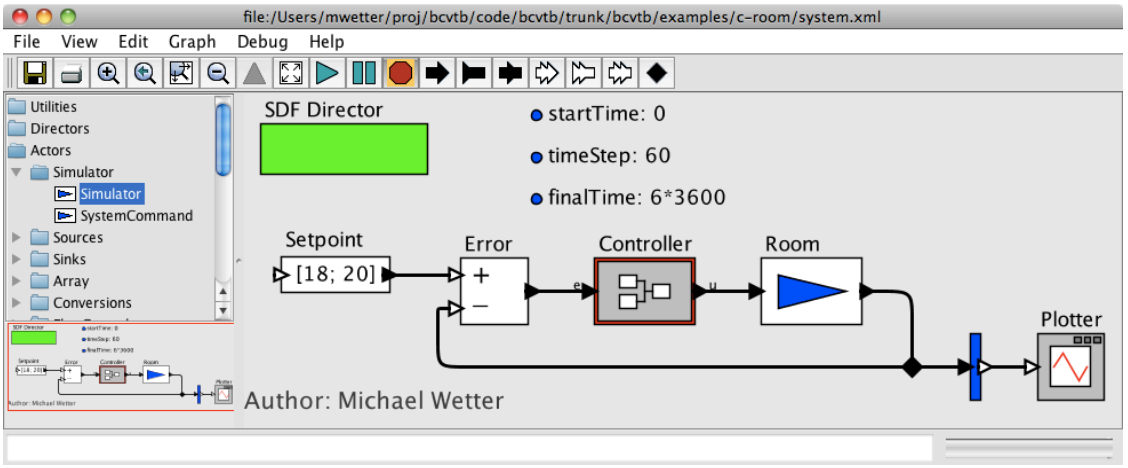


Figure 2: *System diagram that couples the simulation program for the rooms and the controller in Ptolemy II.*

4.5 Example to Illustrate how to Connect a Client to the BCVTB

We will now show an example to illustrate how to connect a client program to the BCVTB. We will consider a system with two rooms, each with a heater that is controlled by a proportional controller. We will implement the simulation program for the two rooms in a C program and the controller in Ptolemy II.

Let $k \in \{0, 1, 2, \ldots\}$ denote equally spaced time steps and let $i \in \{1, 2\}$ denote the number of the room. For the k-th time step and the room number i, let $T^i(k)$ denote the room temperature and let $u^i(k)$ denote the control signal for the heater. The room temperature is governed by

$$T^i(k+1) = T^i(k) + \frac{\Delta t}{C^i} (UA)^i (T_{out} - T^i(k)) + \frac{\Delta t}{C^i} Q_0^i u^i(k), \tag{10}$$

$$T^i(0) = T_0^i, \tag{11}$$

where Δt is the time interval, C^i is the room thermal capacity, $(UA)^i$ is the room heat loss coefficient, T_{out} is the outside temperature, Q_0^i is the nominal capacity of the heater and T_0^i is the initial temperature. In (10) and (11), we assumed that the communication time step is small enough to be used as the integration time step. If this is not the case, we could use a different integration time step and synchronize the integration time step with the communication time step. The governing equation for the control signal is

$$u^i(k+1) = \min\big(1, \max\big(0, \gamma^i (T_{set}^i - T^i(k))\big)\big), \tag{12}$$

where $\gamma^i > 0$ is the proportional gain, T_{set}^i is the set point temperature and the $\min(\cdot,\cdot)$ and $\max(\cdot,\cdot)$ functions limit the control signal between 0 and 1.

Fig. 3 shows a source code snippet of the implemented client. Three functions interface the client with the BCVTB: On line 2, the function `establishclientsocket` establishes

```
1  // Establish the client socket
2  const int sockfd = establishclientsocket("socket.cfg");
3  if (sockfd < 0){
4    fprintf(stderr,"Error: Failed to obtain socket file descriptor.\n");
5    exit((sockfd)+100);  }
6  // Simulation loop
7  while(1){
8    // assign values to be exchanged
9    for(i=0; i < nDblWri; i++)  dblValWri[i]=TRoo[i];
10   // Exchange values
11    retVal = exchangedoubleswithsocket(&sockfd, &flaWri, &flaRea,
12                                       &nDblWri, &nDblRea,
13                                       &simTimWri, dblValWri,
14                                       &simTimRea, dblValRea);
15   ////////////////////////////////////////////////////////////
16   // Check flags
17   if (retVal < 0){
18     sendclientmessage(&sockfd, &cliErrFla);
19     printf("Simulator received value %d from socket.\n", retVal);
20     closeipc(&sockfd);     exit((retVal)+100);  }
21   if (flaRea == 1){
22     printf("Simulator received end of simulation signal.\n");
23     closeipc(&sockfd);     exit(0);  }
24   if (flaRea != 0){
25     printf("Simulator received flag = %d. Exit simulation.\n", flaRea);
26     closeipc(&sockfd);     exit(1);  }
27   ////////////////////////////////////////////////////////////
28   // No flags found that require the simulation to terminate.
29   // Assign exchanged variables
30   for(i=0; i < nRoo; i++)
31     u[i] = dblValRea[i];
32   ////////////////////////////////////////////////////////////
33   // Having obtained u_k, we compute the new state x_k+1 = f(u_k).
34   // This is the actual simulation time step of the client
35   for(i=0; i < nRoo; i++)
36     TRoo[i] = TRoo[i] + delTim/C[i] * ( UA * (TOut-TRoo[i] )
37               + Q0Hea * u[i] );
38   simTimWri += delTim; // advance simulation time
39 } // end of simulation loop
```

Figure 3: *Code snippet that shows the integration of a simple simulator in the BCVTB.*

the socket connection from the client to the middleware. The return value is an integer that references the socket. This descriptor is then used on line 11 as an argument to the function call `exchangedoubleswithsocket`. This function writes data to the socket and reads data from the socket. Its arguments are the socket file descriptor, a flag to send a signal to the middleware (a non-zero value means that the client will stop its simulation) and a flag received from the middleware (a non-zero value indicates that no further values will be written to or read from the socket by the client). The remaining arguments are the array lengths and the array data to be written to and read from the middleware.

After the call to `exchangedoubleswithsocket` follows error handling. The test `retVal < 0` checks for errors during the communication. If there was an error, then a message is sent to the server to indicate that the client will terminate the co-simulation. Finally, the socket connection is closed by calling `closeipc`.

To simulate this example, we implemented the controller directly in the middleware, using actors from the Ptolemy II library. However, the controller could as well be implemented in Modelica, MATLAB, Simulink or in a user written program that communicates through a BSD socket similarly to the C client above. Fig. 2 shows the system diagram with the actor for the controller and the actor that interfaces the simulation program.

4.6 Sequence of Data Exchange

In Fig. 4 we show the sequence of data exchange between the clients and Ptolemy II. In this schematic, we assumed two clients, but more clients are possible if needed. The figure shows the data exchange between the clients and Ptolemy II (dashed arrows), the data exchange inside Ptolemy II (dotted arrows) and the simulation time in the clients and in Ptolemy II. Once a client has initialized its data, it calls the function `exchangedoubleswithsocket` to write its initial values to Ptolemy II, as indicated by the arrow $x_1(0)$ for client 1. Ptolemy II will send these initial values to the other Ptolemy II actors, which may include client 2. This exchange within Ptolemy II is indicated by the dotted arrows. Then, the time integration starts: Ptolemy II sends data that include initial conditions of other clients to the sockets, and the clients receive them. This is illustrated by the arrow labeled $x_2(0)$ where data is sent to client 1 which still waits until its first call to `exchangedoubleswithsocket` returns. Now, client 1 computes $x_1(1) = x_1(0) + \int_{t_0}^{t_1} f_1(x_1(0), x_2(0))\, ds$. Next, client 1 may call its output report routines and then call `exchangedoubleswithsockets` to write $x_1(1)$ to Ptolemy II and to receive $x_2(1)$ from Ptolemy II. After client 1 receives $x_2(1)$, it computes $x_1(2) = x_1(1) + \int_{t_1}^{t_2} f_1(x_1(1), x_2(1))\, ds$ and the procedure is repeated. This exchange scheme continues until Ptolemy II reaches its final time. Then, Ptolemy II writes a flag with a value of 1 to the client to indicate that the last time step has been reached and, hence, that the client won't receive any further data.

4.7 Interfaces to Simulation Programs

This section describes the interfaces that we implemented in various simulation programs to allow exchanging data between these programs and the BCVTB. All interfaces use the routines described in the previous section.

4.7.1 EnergyPlus Interface

To interface EnergyPlus with the BCVTB, we added a module called `ExternalInterface` to EnergyPlus version 5.0. This interface allows overwriting certain EnergyPlus variables during each EnergyPlus zone time step with data received from Ptolemy II and retrieving data from EnergyPlus to be sent to Ptolemy II. In particular, the EnergyPlus interface adds new objects to EnergyPlus that are similar to the existing objects

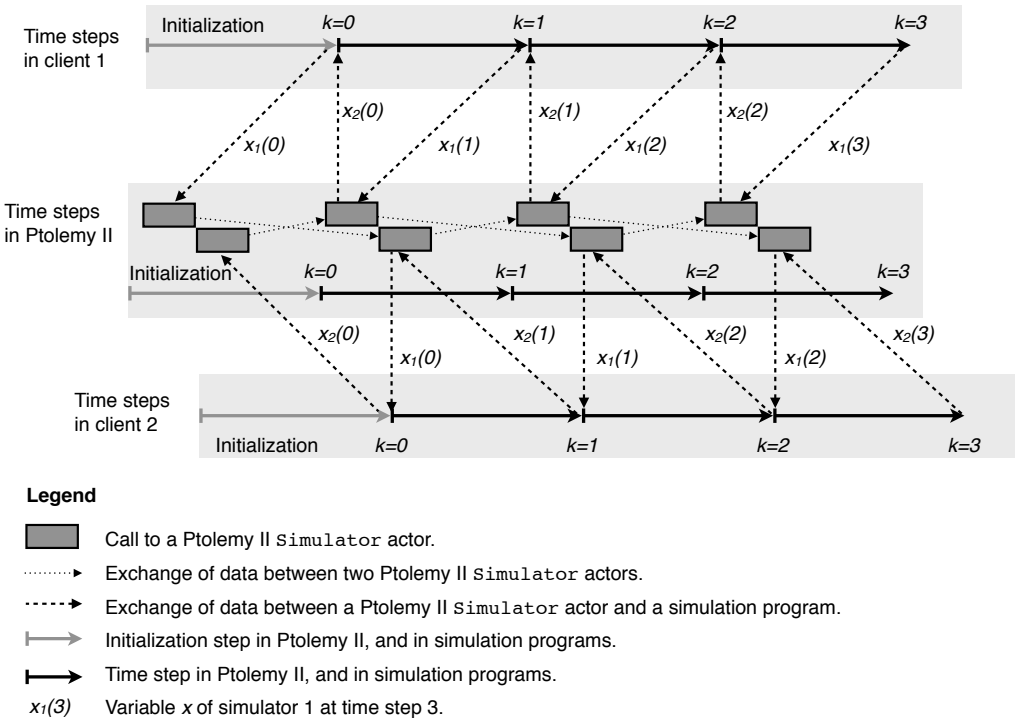


Figure 4: *Data synchronization and function calls between the Ptolemy II middleware and two simulation programs.*

- `Schedule:Compact`,
- `EnergyManagementSystem:Actuator`, and
- `EnergyManagementSystem:GlobalVariable`.

The corresponding BCVTB objects are called

- `ExternalInterface:Schedule`,
- `ExternalInterface:Actuator`, and
- `ExternalInterface:Variable`.

These objects allow writing to EnergyPlus schedules, Energy Management System (EMS) actuator objects and EMS variables at each EnergyPlus zone time step. Schedules can, for example, be used to add sensible or latent heat to rooms or to change

setpoints. EMS actuators allow additional capability to overwrite setpoints and to control equipment. EMS variables can be used within the EnergyPlus Runtime Language to implement custom control functions.

The BCVTB interface also allows to send any variables of type `Output:Variable` and `EnergyManagementSystem:OutputVariable` from EnergyPlus to the BCVTB at the end of each zone time step.

Prior to simulating the run period that is specified in the EnergyPlus input file, EnergyPlus simulates a warm-up period (to compute its initial values) and possibly does a system sizing calculation. During these periods, no data are exchanged between the BCVTB and EnergyPlus. The values of `ExternalInterface:Schedule` and `ExternalInterface:Variable` will be fixed at their initial value, which is a mandatory argument. However, for the object `ExternalInterface:Actuator`, specification of an initial value is optional. If an initial value is specified, it is used during warm-up and system sizing. If it is not specified, then the actuator only overwrites the actuated component after the warm-up and system sizing. Since actuators always overwrite other objects (such as a schedule), all these objects have values that are defined during the warm-up and the system sizing even if no initial value is specified.

Thus, suppose we have an EnergyPlus input data file that contains an instance of `ExternalInterface:Actuator` that overwrites a `Schedule:Compact`. Then, there are the following options:

1. `ExternalInterface:Actuator` specifies an initial value: In this situation, during the warm-up period and the system sizing, the value of `Schedule:Compact` will be set to the initial value specified by the instance of `ExternalInterface:Actuator`.

2. `ExternalInterface:Actuator` does not specify an initial value: In this situation, during the warm-up period and the system sizing, the value of `Schedule:Compact` will be as defined by this schedule, i.e., it can change over time. Once the time stepping starts, the value of `Schedule:Compact` will be overwritten by the numerical value that `ExternalInterface:Actuator` receives from the BCVTB.

4.7.2 Modelica Interface

To interface Modelica with the BCVTB, we implemented in the Modelica Buildings library [Wetter, 2009c,b] a block that exchanges data with the BCVTB. This Modelica block receives from the BCVTB a vector of double values $y(t)$ which become output of the Modelica block. The input of this Modelica block is a vector of double values $u(t)$ that is computed by other Modelica components and that will be sent to the BCVTB. Since the data exchange is conducted at a fixed sampling interval, the output of the block is $y(t) = y(t_k)$ for $t \in [t_k, t_{k+1})$. For the input that is sent to the BCVTB, there are three configurations possible: Let $u^i(t)$ be the i-th element of $u(t)$, let $k \in \{1, 2, \ldots\}$ denote the time step, and let u^i_k be the i-th element of the vector that is sent from Modelica to the BCVTB at time step k. Then, the user can select to send

- the instantaneous value $u^i_k = \lim_{s \uparrow t_k} u^i(s)$,

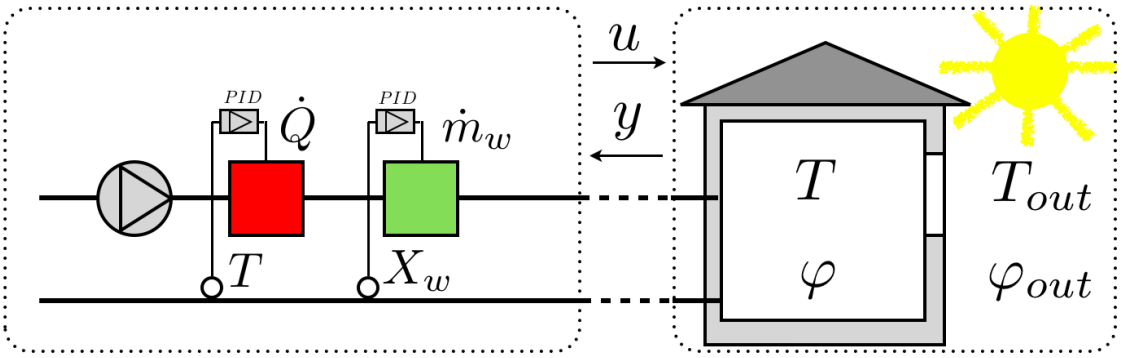


Figure 5: *System decomposition for interfacing an HVAC system that is modeled in Modelica with a building that is modeled in EnergyPlus.*

- the average value over the sampling interval $u_k^i = 1/(t_k - t_{k-1}) \int_{t_{k-1}}^{t_k} u^i(s)\, ds$, or
- the integral over the sampling interval $u_k^i = \int_{t_{k-1}}^{t_k} u^i(s)\, ds$.

For state variables, one may typically want to send the instantaneous value, where as for flow variables, the average or integrated value may be more appropriate. Note that for the instantaneous value, the limit from below is used to avoid an iteration across the BCVTB interface.

We also implemented a Modelica model that allows interfacing an air-conditioning system to the BCVTB. The model takes as input signals the temperature and water vapor concentration and, optionally, a bulk mass flow rate into or out of the system boundary. The state of the air that flows out of this model will be at this temperature and water vapor concentration. The output of this model is the sensible and latent heat exchanged across the system boundary. When used with the BCVTB, a building simulation program such as EnergyPlus may compute the room air temperatures and room air humidity rate, which is then used as an input to this model. The sensible and latent heat flow rates may be sent to EnergyPlus to couple the air-conditioning system to the energy balance of the building model.

We will now illustrate the use of this model. We take a simple HVAC system that consists of a supply fan with constant volume flow rate, a heater and a humidifier that serves a single thermal zone. The heater adds heat to the supply air in the amount $\dot{Q} = y_h \dot{Q}_0$ and the humidifier adds water vapor in the amount $\dot{m}_w = y_w \dot{m}_{w,0}$, where y_h and y_w are the output signals of controllers that track the return air temperature and water vapor concentration (which, in the simulation model, is equal to the room air temperature and water vapor concentration as received from EnergyPlus), and $\dot{Q}_0$ and $\dot{m}_{w,0}$ are the nominal heating and humidification capacities. We partitioned the system model as shown in Fig. 5, where the dotted line indicates the system boundary. On the left is the HVAC model that was implemented in Modelica, and on the right is the thermal zone and the building envelope model that was implemented in EnergyPlus. The

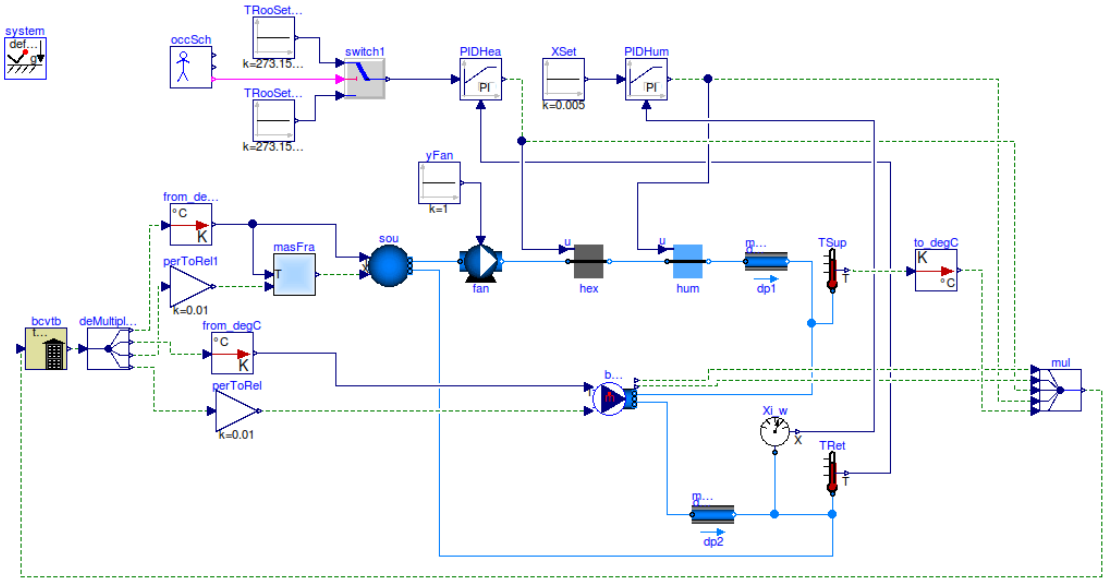


Figure 6: *Modelica implementation of HVAC model with interface to the BCVTB.*

Modelica implementation of the HVAC model is shown in Fig. 6. The dotted lines are signals that are exchanged with the BCVTB.

To couple the two models, we set, in the time interval $t \in [t_k, t_{k+1})$, the boundary condition for the HVAC system model to the room air temperature $T(t) = T_k$, to the room relative humidity $\varphi(t) = \varphi_k$, to the outside air temperature $T_{out}(t) = T_{out,k}$ and to the outside air relative humidity $\varphi_{out}(t) = \varphi_{out,k}$. Thus, the vector sent from EnergyPlus to Modelica is $y_k = (T_k, \varphi_k, T_{out,k}, \varphi_{out,k})$. In the HVAC model, we computed the sensible and latent heat $\dot{Q}_s(t)$ and $\dot{Q}_l(t)$ that is exchanged if the HVAC system serves a room at temperature T_k and relative humidity φ_k. Thus, we sent from Modelica to EnergyPlus the vector $u_0 = (0, 0)^T$ and, for $k > 0$,

$$u_k = \frac{1}{(t_k - t_{k-1})} \int_{t_{k-1}}^{t_k} \begin{pmatrix} \dot{Q}_s(s) \\ \dot{Q}_l(s) \end{pmatrix} ds, \tag{13}$$

which become the boundary conditions for EnergyPlus. In particular, the sensible and latent heat input was added to the room energy balance of EnergyPlus by using an instance of an EnergyPlus object of type `OtherEquipment`, and setting its capacity to a schedule value that was overwritten by the BCVTB using an instance of `ExternalInterface:Schedule`.

4.7.3 MATLAB Interface

We implemented a MATLAB library that provides functions that allow MATLAB to be interfaced to the BCVTB. Fig. 7 shows a code section that illustrates how a MATLAB simulation model can be interfaced with the BCVTB for data exchange at each time step. For brevity, some data initialization and error handling have been omitted. The code

```
1 % Initialize model variables
2 [... omitted for brevity]
3 % Establish the socket connection with the BCVTB
4 sockfd = establishClientSocket('socket.cfg');
5 % Loop for simulation time steps.
6 simulate=true;
7 while (simulate)
8   % Assign values to be exchanged.
9   [retVal, flaRea, simTimRea, u ] = ...
10        exchangeDoublesWithSocket(sockfd, flaWri,  ...
11                                  length(u), simTimWri, TRoo);
12  % Check return flags
13  if (flaRea ~= 0) % Error during communication or final time reached.
14    closeIPC(sockfd);
15    simulate=false;
16  end
17  % Having obtained u_k, we compute the new state x_k+1 = f(u_k)
18  if (simulate)
19    for i=1:2
20      TRoo(i) = TRoo(i) + ...
21          delTim / C(i) * ( UA * (TOut-TRoo(i) ) + Q0Hea * u(i) );
22    end
23    % Advance simulation time
24    simTimWri = simTimWri + delTim;
25  end
26 end
27 exit
```

Figure 7: *Code snippet that illustrates integration of a MATLAB model with the BCVTB.*

is from an implementation of the room model described in (10) and (11). (A complete implementation is provided with the BCVTB installation.) On line 4, a function call is made to establish a BSD socket connection between MATLAB and the BCVTB. On line 9 to 11 is the data exchange with the BCVTB where the current room temperature T_k is sent from MATLAB to the BCVTB, and the control signal u_k is received from the BCVTB. On line 14, the communication is terminated.

4.7.4 Simulink Interface

There is also a Simulink block that allows data to be exchanged with the BCVTB at each Simulink sampling time step. Fig. 8 shows how a Simulink controller model (block labeled "controller" in the figure) is linked with a model that handles the interface with the BCVTB (block labeled `socketIO`). Inputs to the block `socketIO` are a vector containing the control signals, such as setpoints and actuator values, and a trigger signal. If the trigger signal is 0, then the block will not be called during the simulation. This enables a model builder to test and debug the control algorithm in isolation from the model of the plant. Output of the block `socketIO` is a vector of sensor values that has been received from Ptolemy II. The code generation tool of MATLAB, which is used to interface the

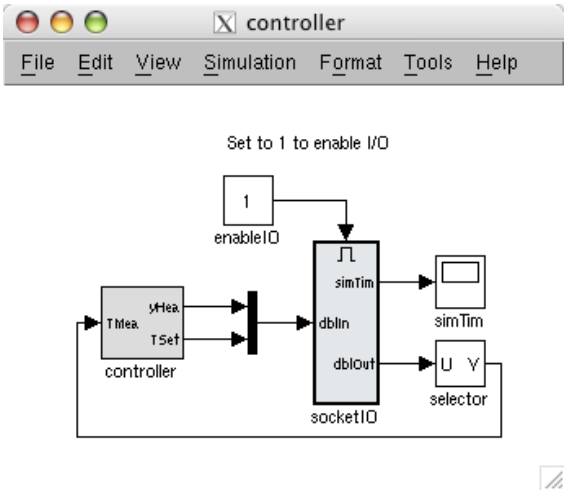


Figure 8: *Simulink model that interfaces a Simulink controller model (left grey box) with a model that interfaces Ptolemy II (right grey box).*

C socket library, requires MATLAB function outputs to have a fixed array size. This causes the output vector of `socketIO` to typically have more elements than required. The block `selector` is used to select the subset of elements needed by the controller. If more elements are needed than are currently allocated in the `socketIO` interface, a user can increase the value in a MATLAB script and recompile the MATLAB/Simulink socket library using a script that is part of the BCVTB.

4.8 Interface to a System Call

In some situations, one may want to call at each time step an external program and retrieve its output after it terminates. A use case is to call, at each time step, a lighting simulation in Radiance to compute the illuminance that would be measured by a photosensor for a given control signal of an active facade. Another use case is to develop and test a controller, that is implemented in the C language, in a simulation environment that couples the controller to a model of the plant that it will control. For such a situation, we added an actor called `SystemCommand`. This actor allows, for example, to call a batch file (on Windows), a shell script (on Mac or Linux), or any other executable program. The input to this program can be done either through program flags, or by writing an input file from Ptolemy II, using actors from Ptolemy II's library.

For illustration, we will show how to call a program that implements a simple controller for a room with closed loop control. The program is implemented in the C language and simulates a proportional controller with output limitation. The command line arguments of the program are the control error `e` and the proportional gain `kP`. The program writes the control signal to a text file, which will then be parsed by Ptolemy II.

Such a program can be called by making an instance of the `SystemCommand` actor. Fig. 9 shows this actor as part of the example that is discussed here. Next, two input

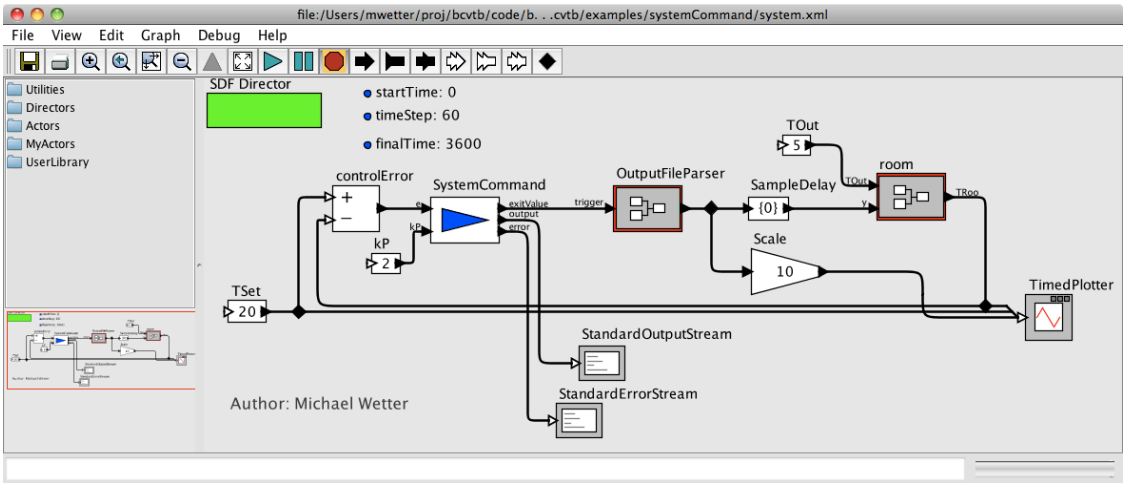


Figure 9: *System model to illustrate the implementation of the* `SystemCommand` *actor that calls an external program that implements the control function.*

ports can be added to this actor. Ptolemy II allows giving a name to each input port, which we called `e` and `kP`. The actor also allows specifying a program name and any number of command line arguments, which we set to `pcontroller` (which is the name of the executable) and to `$e` and `$kP`, respectively. During the simulation, the text strings `$e` and `$kP` in the command line arguments will be replaced by the current value of the input token of port `e` and port `kP`.[1] For example, suppose that at some time step, the input ports have the values $e = 1$ and $kP = 2$. Then the BCVTB will fire the command

```
pcontroller 1 2
```

The `SystemCommand` actor has three output ports: The port `exitValue` outputs the exit value of the program. The port `output` contains the standard output stream of the program, and the port `error` contains the standard error stream of the program.

Upon successful termination, the port `exitValue` will have the token `0`. This token can be used to trigger the actor `OutputFileParser`. If this actor receives `0` on its input port, then it will read the output file that was written by the program `pcontroller` and send the output to a plotter. If it receives a non-zero value, it will stop the BCVTB with an error message. The text streams of the ports `output` and `error` are sent to a display to show to the user the program's standard output and standard error stream.

5 Examples using Whole Building Simulation Programs

In this section, we will present examples that use the EnergyPlus whole building simulation program which is linked to Simulink, to Modelica and to a controller that is implemented in Ptolemy II.

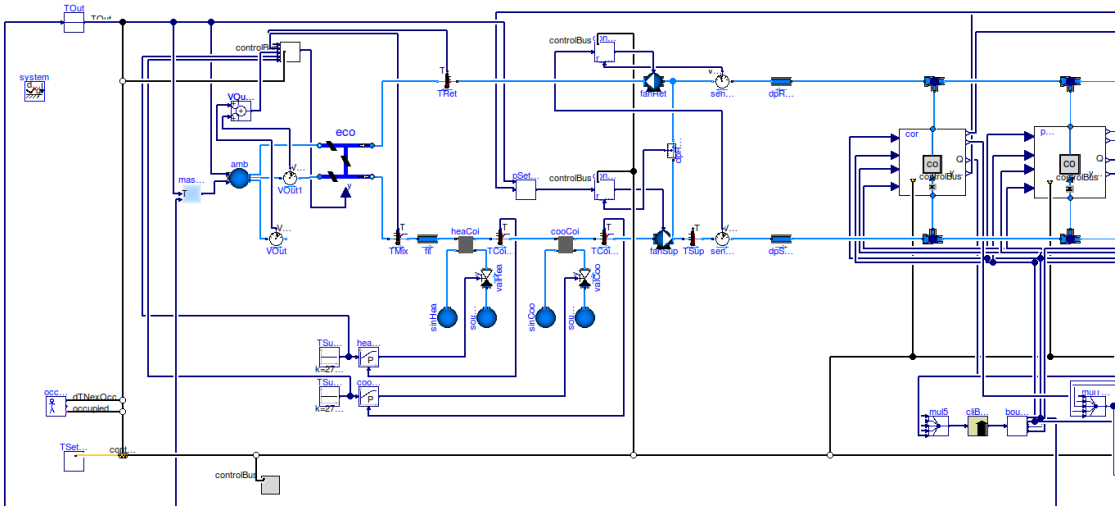
[1]There are also two built-in variables called `$time`, which is the current simulation time, and `$iteration` which is the current iteration step of Ptolemy. For these two variables, no port needs to be defined.

5.1 Use of Modelica and EnergyPlus to Test Control Sequence of a VAV System

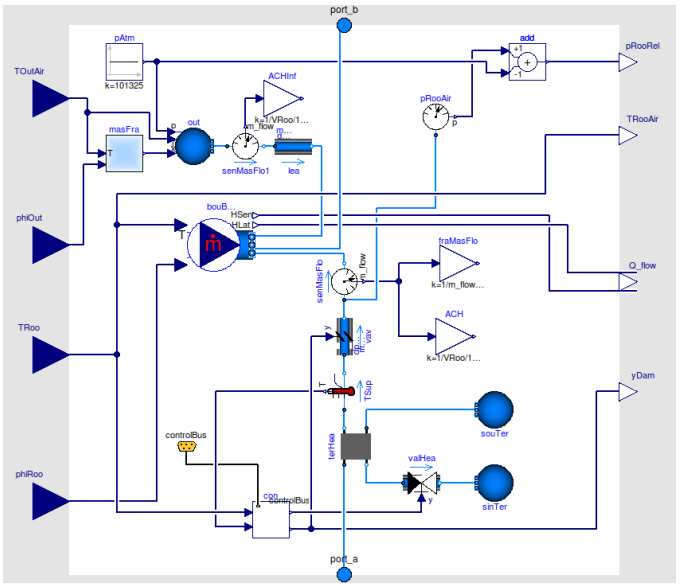
This example illustrates how an HVAC system and its supervisory and local loop control can be modeled in the equation-based object-oriented Modelica language and linked to EnergyPlus through the BCVTB. Modelica is used to implement the HVAC system and the control. The building heat transfer and the temperature and humidity concentration of the thermal zones are implemented in EnergyPlus. The use of co-simulation allowed to use Modelica for its rapid prototyping and flexible modeling capabilities, and for its capability to model and simulate local loop control and supervisory control sequences, pressure-dependent flow distribution in the duct network and the dynamics of HVAC components, which are not possible to model in EnergyPlus. It also allowed to use models from EnergyPlus for the coupled simulation of the building envelope heat transfer, solar gains, detailed fenestration performance and daylight availability, for which Modelica models do not yet exist.

The HVAC system as implemented in Modelica is shown in Fig. 10. It is a variable air volume (VAV) flow system with economizer and a heating and cooling coil in the air handler unit. There is also a reheat coil and an air damper in each of the five zone inlet branches. The control is an implementation of the control sequence *VAV 2A2-21232* of the Sequences of Operation for Common HVAC Systems [ASHRAE, 2006]. In this control sequence, the supply fan speed is regulated based on the duct static pressure. The return fan controller tracks the supply fan air flow rate reduced by a fixed offset. The duct static pressure is adjusted so that at least one VAV damper is 90% open. The economizer dampers are modulated to track the setpoint for the mixed air dry bulb temperature. Priority is given to maintain a minimum outside air volume flow rate. There is also an override that keeps the mixed air temperature above 3°C to prevent freezing of coils. In each zone, the VAV damper is adjusted to meet the room temperature setpoint for cooling, or fully opened during heating. The room temperature setpoint for heating is tracked by varying the water flow rate through the reheat coil. There is also a finite state machine that transitions the mode of operation of the HVAC system between the modes "occupied," "unoccupied off," "unoccupied night set back," "unoccupied warm-up" and "unoccupied pre-cool." In the VAV model, all air flows are computed based on the duct static pressure distribution and the performance curves of the fans. Local loop control is implemented using proportional and proportional-integral controllers, while the supervisory control is implemented using a finite state machine. There is also pressure-driven air leakage through the building envelope. The total system model contained 800 components that led to a differential algebraic equation system with 3,700 scalar equations and 75 state variables.

To model the building envelope, the new construction medium office building for Chicago, IL, was selected from the set of DOE Commercial Building Benchmarks [Deru et al., 2009]. The building is an office building with three floors. Each floor has four perimeter zones and a core zone. The envelope thermal properties meet ASHRAE Standard 90.1-2004. We modified the original input file by removing the HVAC system for the middle floor as this system was modeled in Modelica. We also reduced the time step



(a) Partial view of the model of the central HVAC system that serves five thermal zones.



(b) Model of one thermal zone that is encapsulated in the components shown in the right hand side of subfigure (a).

Figure 10: *Modelica model of HVAC system.*

in EnergyPlus and changed the numerical method that computes heat conduction from conduction transfer functions to finite differences as the conduction transfer function algorithm is not applicable for small time steps.

The coupling between Modelica and EnergyPlus is done through the BCVTB. The inputs to Modelica are the air temperatures and relative humidities for the outside and the five zones of the middle floor. The inputs to EnergyPlus are the sensible and latent

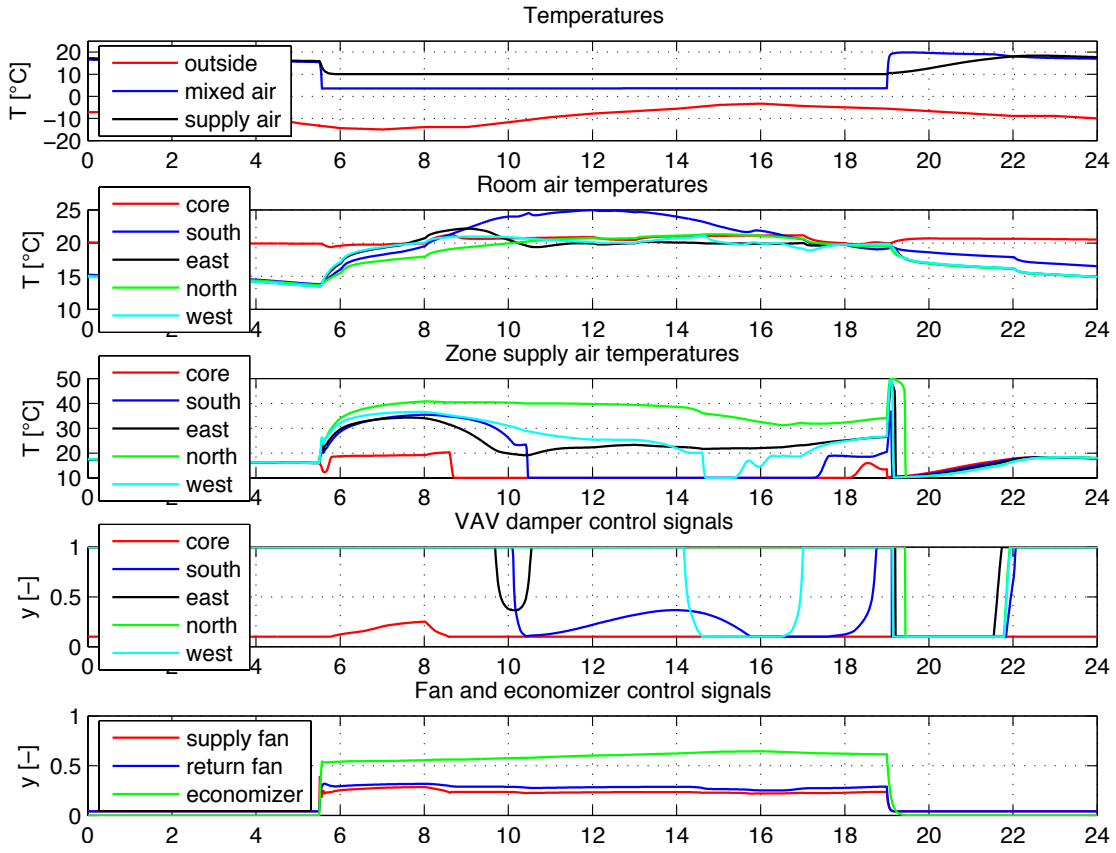


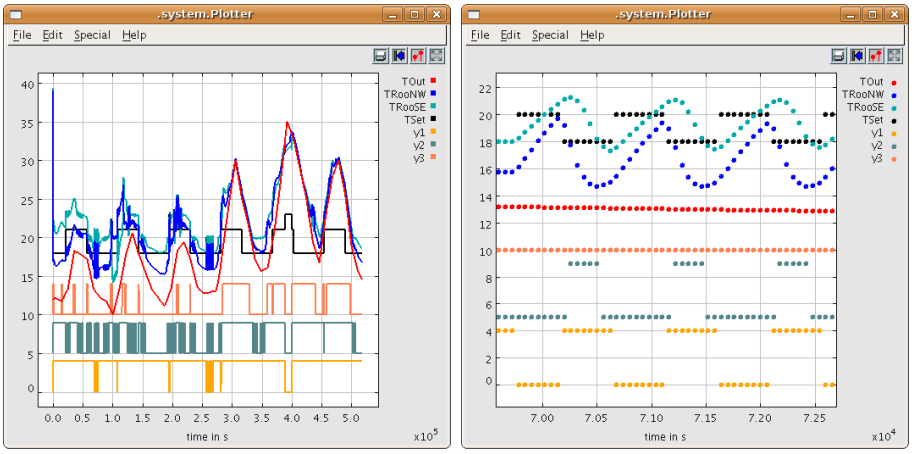
Figure 11: *Temperatures and control signals for a winter day.*

heat exchanged with the room air. Data are exchanged between the two programs every one minute of simulation time.

Fig. 11 shows the air temperatures and control signals for a winter day. After 9 am, the south facing room has the highest temperature, which causes its reheat coil to shut off around 10:30 am. Earlier in the morning, this zone used reheat and opened its VAV damper to increase the air flow rate. Between noon and 4 pm, the VAV damper of the south zone opens up to about 35% because the room temperature approaches the cooling set point temperature and an increase in air flow rate cools the room. The VAV damper and the supply air temperature of the core zone are close to their minimum since this zone has little heating demand. During day-time, there is a slight variation in fan speed to track the duct static pressure setpoint as the VAV dampers change their positions.

5.2 Use of EnergyPlus and Simulink to Test Control of a Naturally Ventilated Building

We will now present an example in which we linked EnergyPlus and MATLAB/Simulink to the BCVTB. EnergyPlus simulates the building including its natural ventilation, and



(a) 5 days plot during summer (b) 45 minutes magnification during night

Figure 12: *Ptolemy II's online plot showing the outside air temperature (red), the room air temperatures (blue and green), the room set point temperature (black) and the window openings (the upper lines indicate open and the lower lines indicate closed windows) during five summer days (subfigure a) and during 45 minutes at night (subfigure b).*

MATLAB/Simulink simulates a controller that determines the window opening positions. The use of co-simulation allowed to use EnergyPlus for modeling the physics, and to use MATLAB/Simulink for its support for control algorithm development.

Both programs exchange data via the BCVTB every 1 minute of simulation time. In EnergyPlus, we use two report variables for the room air temperatures of two rooms through which cross ventilation occurs, and we use a report variable for the outside dry-bulb temperature. These three temperatures are input into the Simulink controller, together with the current clock time. The controller determines the room air setpoint temperature, using different values for day and night and a 2 Kelvin dead-band between opening and closing the windows. The control logic is as follows: For some time step $k \in \mathbb{N}$, let $T_s(k)$ denote the setpoint temperature, let $T_r(k)$ denote the larger of the two room temperatures and let $T_o(k)$ denote the outside dry-bulb temperature. If $T_r(k) > \max(T_s(k), T_o(k))$, then windows are allowed to be open. Otherwise they are closed. There are three opening positions for the windows: For a constant gain $\gamma = 0.5$, the controller computes the control signal $y(k) = \gamma\,(T_r(k) - T_s(k))$. For $0 \leq y(k) \leq 1$, one window group is open, for $1 < y(k) \leq 2$ two window groups are open and for $2 < y(k)$ all windows are open. This control logic is implemented in a Simulink block similar to the block labeled `controller` in Fig. 8.

Fig. 12 shows the temperature trajectories and the window positions. In the first few days, there is significant chattering, with the window opening and closing in a 15 minute long limit cycle (lower graph). This would be unacceptable for building occupants and would wear out the actuators. Hence, a more sophisticated control algorithm should be used as was implemented in the actual building [da Graca et al., 2004]. A better control algorithm may involve modulating the window opening to avoid the limit cycle.

5.3 Use of Ptolemy II and EnergyPlus to Implement a Shading Controller

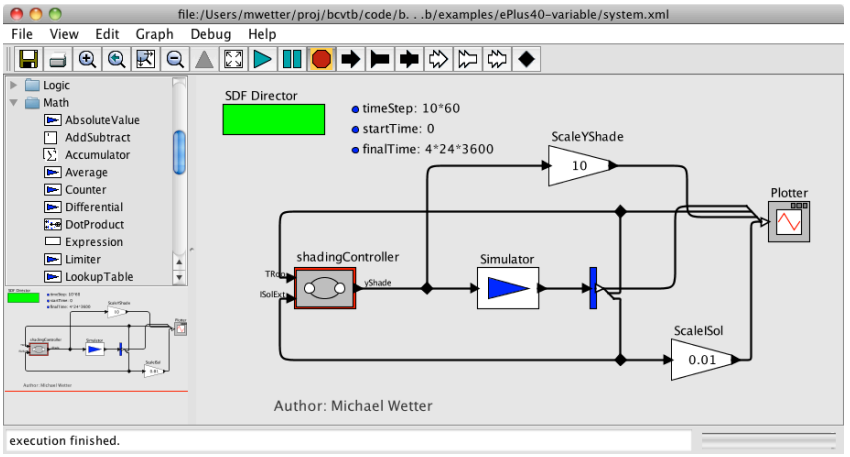
This example illustrates how Ptolemy II can be used to model a controller of a shading device that is linked to an EnergyPlus model of a building. This co-simulation setup allowed to use EnergyPlus to model the physics, and to use the graphical, freely programmable Ptolemy II environment to implement a controller of a shading device using a finite state machine.

The finite state machine switches between the states "night shade deployed," "no shade," and "day shade deployed." The criteria for switching the state of the shading device are the exterior solar irradiation on the window and the room air temperature. These quantities are computed in EnergyPlus. Every 10 minutes of simulation time, EnergyPlus sends the solar irradiation, the room air temperature and the outside air temperature to the BCVTB. In the BCVTB, the finite state machine determines the new desired state of the shading device and sends the control signal to EnergyPlus. Fig. 13(a) shows the implementation of the Ptolemy II model that links an actor for the controller, labeled "shadingController" with the actor labeled "Simulator" that communicates with EnergyPlus. Fig. 13(b) shows the finite state machine that is inside the actor called "shadingController." Each transition of the finite state machine has a guard, which is a test on the solar irradiation and room air temperature in our example. If the guard is true, then the state changes and a new value is assigned to the output signal. Fig. 13(c) shows how the controller state changes with solar irradiation and room air temperature for four days of simulation. A value of $y = 0$ indicates no shade, $y = 1$ indicates day shade deployed and $y = 2$ indicates night shade deployed. The second and fourth day of simulation have large solar radiation and high room air temperatures, which cause the shade to be deployed in the middle of the day.

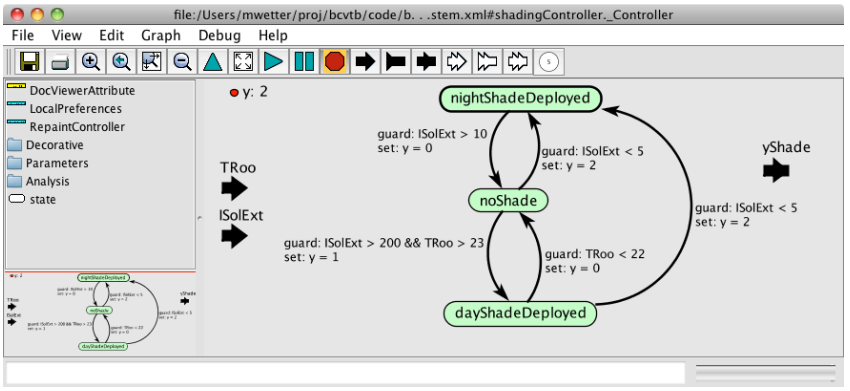
6 Conclusions

Using the Ptolemy II modeling and simulation environment as a middleware for the BCVTB allowed the creation of a modular open-source environment to which different simulation programs or building control systems can be coupled. It also offers users the possibility of implementing system models directly in Ptolemy II, for example, to model physical systems or control systems using different models of computation, such as continuous time, synchronous data flow or finite state machines. Ptolemy II also contains libraries of component models that can be used for data processing; example applications include visualization or transformation of data that are exchanged between different clients.

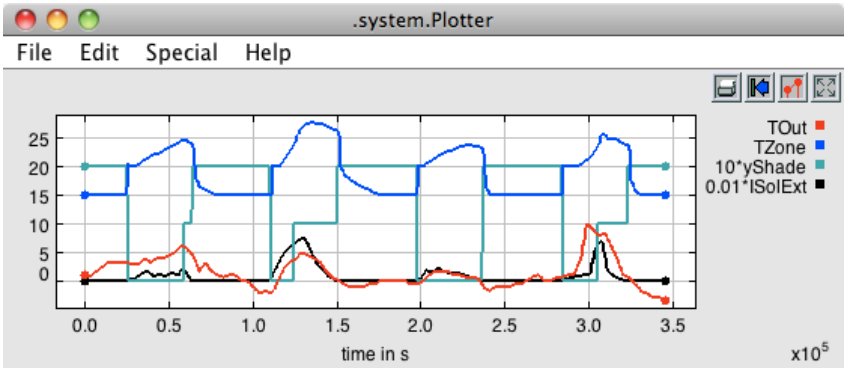
The interface for simulation clients that we added in the form of a Ptolemy II model allowed EnergyPlus, Dymola, MATLAB, Simulink and some custom programs to be coupled to each other. The interface also allows users to add additional simulation programs to the BCVTB. In addition, any executable program can be called from an actor, which allows, for example, Radiance to be coupled to models of facade and light controllers, as well as to models that quantify energy implications of different facades.



(a) Ptolemy II system model that links the controller, EnergyPlus and a plotter.



(b) Implementation of controller for shading device that is encapsulated in the actor "shadingController" in subfigure (a).



(c) Ptolemy II's online plot showing the main simulation results.

Figure 13: *Ptolemy II model that links a shading controller to EnergyPlus and that shows the results in an online plotter as the simulation progresses.*

7 Acknowledgment

This research was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231. We thank Philip Haves from Lawrence Berkeley National Laboratory for his feedback and guidance during the development of this software. Special thanks go to Prof. Edward A. Lee and Christopher Brooks from the University of California at Berkeley for their support in integrating the BCVTB functionality into the Ptolemy II software. We thank Gregor Henze, Charles Corbin, Anthony Florita and Peter May-Ostendorp from the University of Colorado at Boulder for their contributions to the MATLAB interface and the EnergyPlus 3.0 upgrade. We thank Rui Zhang from Carnegie Mellon for her contributions to the Windows configuration and the EnergyPlus 3.1 upgrade.

References

ASHRAE. ANSI/ASHRAE Standard 135-2004, BACnet, a data communication protocol for building automation and control networks, 2004. ISSN 1041-2336.

ASHRAE. *Sequences of Operation for Common HVAC Systems*. ASHRAE, Atlanta, GA, 2006.

Godfried Augenbroe, Pieter de Wilde, Hyeun Jun Moon, and Ali Malkawi. An interoperability workbench for design analysis integration. *Energy and Buildings*, 36(8): 737–748, August 2004. doi: 10.1016/j.enbuild.2004.01.049.

Vladimir Bazjanac. Building energy performance simulation as part of interoperable software environments. *Energy and Buildings*, 39(8):879–883, August 2004. doi: 10.1016/j.buildenv.2004.01.012.

Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, and Haiyang Zheng. Ptolemy II – heterogeneous concurrent modeling and design in Java. Technical Report No. UCB/EECS-2007-7, University of California at Berkeley, Berkeley, CA, January 2007.

Dag Brück, Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson. Dymola for multi-engineering modeling and simulation. In Martin Otter, editor, *Proceedings of the 2nd Modelica conference*, pages 55–1 – 55–8, Oberpfaffenhofen, Germany, March 2002. Modelica Association and Deutsches Zentrum fur Luft- und Raumfahrt. URL `http://www.modelica.org/events/Conference2002`.

Steven T. Bushby, Michael A. Galler, Natascha Milesi Ferretti, and Cheol Park. The virtual cybernetic building testbed – a building emulator. *ASHRAE Transactions*, 116(1):37–44, 2010.

Drury B. Crawley, Linda K. Lawrie, Frederick C. Winkelmann, Walter F. Buhl, Y. Joe Huang, Curtis O. Pedersen, Richard K. Strand, Richard J. Liesen, Daniel E. Fisher, Michael J. Witte, and Jason Glazer. EnergyPlus: creating a new-generation building energy simulation program. *Energy and Buildings*, 33(4):443–457, 2001.

G Carrilho da Graca, P. F. Linden, and P. Haves. Design and testing of a control strategy for a large, naturally ventilated office building. *Building Service Engineering*, 25(3): 223–239, 2004. doi: 10.1191/0143624404bt107oa.

Gaylon M. Decious, Cheol Park, and George E. Kelly. A low cost building/hvac emulator. *HPAC Heating/Piping/AirConditioning*, pages 188–193, January 1997.

M. Deru, K. Field, D. Studer, K. Benne, B. Griffith, P Torcellini, M. Halverson, D. Winiarski, B. Liu, M. Rosenberg, J. Huang, M. Yazdanian, and D. Crawley. Doe commercial building research benchmarks for commercial buildings. Technical report, U.S. Department of Energy, Energy Efficiency and Renewable Energy, Office of Building Technologies, Washington, DC, 2009.

Hilding Elmqvist, Martin Otter, Dan Henriksson, Bernhard Thiele, and Sven Erik Mattsson. Modelica for embedded systems. In Francesco Casella, editor, *Proc. of the 7-th International Modelica Conference*, pages 354–363, Como, Italy, September 2009. Modelica Association. doi: 10.3384/ecp09430096.

Apache Software Foundation. Apache ant 1.8.0 manual. `http://ant.apache.org/`, 2010.

Peter Fritzson and Vadim Engelson. Modelica – A unified object-oriented language for system modeling and simulation. *Lecture Notes in Computer Science*, 1445:67–90, 1998. URL `http://citeseer.ist.psu.edu/fritzson98modelica.html`.

P. Haves, A. L. Dexter, D. R. Jorgensen, K. V. King, and G. Geng. Use of a building emulator to develop techniques for improved commissioning and control of HVAC systems. *ASHRAE Transactions*, 97(1):684–688, 1991.

P. Haves, L. K. Norford, M. DeSimone, and L. Mei. A standard simulation testbed for the evaluation of control algorithms & strategies. Final Report 825-RP, ASHRAE, Atlanta, GA, 1996.

Philip Haves and Peng Xu. The building controls virtual test bed a simulation environment for developing and testing control algorithms, strategies and systems. In Jiang Yi, Zhu Yingxin, Yang Xudong, and Li Xianting, editors, *Proc. of the 10-th IBPSA Conference*, pages 1440–1446. International Building Performance Simulation Association and Tsinghua University, 2007. URL `http://www.ibpsa.org/`.

Philip Haves, Leslie K. Norford, and M. DeSimone. A standard simulation testbed for the evaluation of control algorithms & strategies. *ASHRAE Transactions*, 104(1), 1998.

Jan L. M. Hensen. A comparison of coupled and de-coupled solutions for temperature and air flow in a building. *ASHRAE Transactions*, 105(2):962–969, 1999.

G. E. Kelly, C. Park, and J. P. Barnett. Using emulators/testers for commissioning EMCS software, operator training, algorithm development, and tuning local control loops. *ASHRAE Transactions*, 97(1), 1991.

S. A. Klein, J. A. Duffie, and W. A. Beckman. TRNSYS – A transient simulation program. *ASHRAE Transactions*, 82(1):623–633, 1976.

P. K. Laitila, R. O. Kohonen, K. I. Katajisto, and G. K. Piira. An emulator for testing hvac systems and their control and energy management systems. *ASHRAE Transactions*, 97(1):679–683, 1991.

K. P. Lam, A. Mahdavi, S. Gupta, N. H. Wong, R. Brahme, and Z. Kang. Integrated and distributed computational support for building performance evaluation. *Advances in Engineering Software*, 33(4):199–206, 2002. ISSN 0965-9978. doi: 10.1016/S0965-9978(02)00009-1.

K. P. Lam, N. H. Wong, A. Mahdavi, K. K. Chan, Z. Kang, and S. Gupta. SEMPER-II: an internet-based multi-domain building performance simulation environment for early design support. *Automation in Construction*, 13(5):651–663, September 2004. doi: 10.1016/j.autcon.2003.12.003.

Jean Lebrun. Annex 17. Final report, International Energy Agency, 1992.

Mathworks. Matlab/simulink. `http://www.mathworks.com/`, 2010.

MODELISAR. *Functional Mock-up Interface for Model Exchange*. MODELISAR Consortium, first edition, jan 2010. URL `http://functional-mockup-interface.org`.

John E. Seem, Cheol Park, and John M. House. A new sequencing control strategy for air-handling units. *HVAC&R Research*, 5(1):35–59, January 1999.

M. Trčka, J. L. M. Hensen, and A. J. Th. M. Wijsman. Distributed building performance simulation - a novel approach to overcome legacy code limitations. *ASHRAE HVAC&R*, 12(3a):621–640, 2006.

Marija Trčka, Michael Wetter, and Jan Hensen. Comparison of co-simulation approaches for building and HVAC/R simulation. In Jiang Yi, Zhu Yingxin, Yang Xudong, and Li Xianting, editors, *Proc. of the 10-th IBPSA Conference*, pages 1418–1425. International Building Performance Simulation Association and Tsinghua University, 2007. URL `http://www.ibpsa.org/`.

H. Vaezi-Nejad, E. Hutter, P. Haves, A. L. Dexter, G. Kelly, P. Nusgens, and S. Wang. Use of building emulators to evaluate the performance of building energy and management systems. In J. A. Clarke, J. W. Mitchell, and R. C. Van de Perre, editors, *Proc. of the IBPSA Conference*, pages 209–213, Nice, France, August 1991. URL `http://www.ibpsa.org/conferences.htm`.

S. W. Wang, P. Haves, and P. Nusgens. Design, construction and commissioning of building emulators for emcs applications. *ASHRAE Transactions*, 100(1):1465–1473, 1994.

Gregory J. Ward. The radiance lighting simulation and rendering system. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 459–472, July 1994. URL `http://radsite.lbl.gov/radiance/papers/sg94.1/paper.html`.

Michael Wetter. Modelica-based modeling and simulation to support research and development in building energy and control systems. *Journal of Building Performance Simulation*, 2(2):143–161, June 2009a. doi: 10.1080/19401490902818259.

Michael Wetter. Modelica library for building heating, ventilation and air-conditioning systems. In Francesco Casella, editor, *Proc. of the 7-th International Modelica Conference*, pages 393–402, Como, Italy, September 2009b. Modelica Association. doi: 10.3384/ecp0943.

Michael Wetter. A modelica-based model library for building energy and control systems. In Paul A. Strachan, Nick J. Kelly, and Michäel Kummert, editors, *Proc. of the 11-th IBPSA Conference*, pages 652–659. International Building Performance Simulation Association and University of Strathclyde, 2009c. URL `http://www.ibpsa.org/`.

Christof Wittwer, Werner Hube, Peter Schossig, Andreas Wagner, Christiane Kettner, Max Mertins, and Klaus Rittenhofer. ColSim - a new simulation environment for complex system analysis and controllers. In R. Lamberts, C. O. R. Negrão, and J. Hensen, editors, *Proc. of the 7-th IBPSA Conference*, volume I, pages 237–244, Rio de Janeiro, Brazil, August 2001.

Zhiqiang John Zhai and Qingyan Yan Chen. Performance of coupled building energy and cfd simulations. *Energy and Buildings*, 37(4):333–344, April 2005. doi: 10.1016/j.enbuild.2004.07.001.