Ernest Orlando Lawrence
Berkeley National Laboratory

Building Controls Virtual Test Bed
User Manual
Version 1.0.0

Simulation Research Group
Building Technologies Department
Environmental Energy Technologies Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

http://SimulationResearch.lbl.gov

Michael Wetter
MWetter@lbl.gov

February 28, 2011

Copyright (c) 2010
The Regents of the University of California (through Lawrence Berkeley National Laboratory), subject to receipt of any required approvals from U.S. Department of Energy.
All rights reserved.

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or The Regents of the University of California.

Product and company names mentioned herein may be the trademarks of their respective owners. Any rights not expressly granted herein are reserved.

Contents

1 Introduction **1**

2 Installation and configuration **3**

2.1 Introduction . . . 3

2.2 Installation . . . 3

2.3 Setting system environment variables . . . 3

2.4 Uninstallation . . . 4

3 Running simulations with the BCVTB **5**

3.1 Introduction . . . 5

3.2 Running the BCVTB from an explorer window . . . 5

3.3 Running the BCVTB from a terminal window . . . 5

3.4 Command line arguments . . . 6

3.5 Example files . . . 7

4 Configuring programs for use with the BCVTB **9**

4.1 Introduction . . . 9

4.2 Ptolemy II . . . 9

4.3 EnergyPlus . . . 12

4.3.1 Syntax of the xml file that configures the data mapping between EnergyPlus and the external interface . . . 14

4.3.2 Example 1: Interface using `ExternalInterface:Schedule` . . . 15

4.3.2.1 Create an EnergyPlus idf file . . . 15

4.3.2.2 Create a configuration file . . . 16

4.3.2.3 Create a Ptolemy II model . . . 17

4.3.3 Example 2: Interface using `ExternalInterface:Actuator` . . . 19

4.3.3.1 Create an EnergyPlus idf file . . . 19

4.3.3.2 Create a configuration file . . . 20

4.3.4 Example 3: Interface using `ExternalInterface:Variable` . . . 21

4.3.4.1 Create an EnergyPlus idf file . . . 22
4.3.4.2 Create a configuration file . . . 23
4.4 Dymola . . . 24
4.4.1 Create a Modelica model . . . 24
4.4.2 Create a Modelica script . . . 25
4.4.3 Create a Ptolemy II model . . . 25
4.5 MATLAB . . . 27
4.5.1 Create a MATLAB script . . . 27
4.5.2 Create a Ptolemy II model . . . 27
4.6 Simulink . . . 29
4.6.1 Create a Simulink Block Diagram . . . 29
4.6.2 Create a MATLAB script . . . 32
4.6.3 Create a Ptolemy II model . . . 33
4.7 Custom program using a system command . . . 34
4.7.1 Create a Ptolemy II model . . . 34
4.7.2 Configure the ports of the `SystemCommand` actor . . . 35
4.7.3 Configure the parameters of the `SystemCommand` actor . . . 36
4.8 Radiance . . . 37
4.8.1 Introduction . . . 37
4.8.2 Configuring Radiance . . . 37
4.8.3 Create a Radiance script . . . 37
4.8.4 Create a Ptolemy II model . . . 39
4.9 BACnet . . . 42
4.9.1 Introduction . . . 42
4.9.2 Reading from BACnet . . . 42
4.9.2.1 Specification of data that will be read from BACnet . . . 42
4.9.2.2 Interface to BACnet Stack . . . 44
4.9.3 Writing to BACnet . . . 45
4.9.3.1 Specification of data that will be written to BACnet . . . 45
4.9.3.2 Interface to BACnet Stack . . . 46
4.9.4 Creating a Ptolemy II model . . . 47
4.9.4.1 Configuring the `BACnetReader` . . . 47
4.9.4.2 Configuring the `BACnetWriter` . . . 49
4.9.4.3 Synchronization with real-time . . . 50
4.10 Analog/Digital Interface . . . 50

4.10.1 Introduction . . . 50
4.10.2 Reading from ADInterfaceMCC . . . 50
4.10.2.1 Specification of data that will be read from ADInterfaceMCC . . . 50
4.10.2.2 Interface to adInterfaceMCC-Stack . . . 51
4.10.3 Writing to ADInterfaceMCC . . . 51
4.10.3.1 Specification of data that will be written to ADInterfaceMCC . . . 51
4.10.3.2 Interface to adInterfaceMCC-Stack . . . 52
4.10.4 Creating a Ptolemy II model . . . 52
4.10.4.1 Configuring the `ADInterfaceMCCReader and ADInterfaceMCCWriter` . . . 53
4.10.4.2 Synchronization with real-time . . . 54

5 Mathematics of the Implemented Co-Simulation 55
5.1 Introduction . . . 55
5.2 Description . . . 55

6 Development 57
6.1 Introduction . . . 57
6.2 Functional requirements . . . 57
6.3 Software requirements . . . 57
6.3.1 Linux . . . 58
6.3.2 Mac OS X . . . 58
6.3.3 Windows . . . 58
6.4 Version control . . . 58
6.4.1 Checking out a version . . . 59
6.4.2 Creating a branch . . . 59
6.4.3 Merging . . . 59
6.4.4 Resources . . . 60
6.5 Updating Ptolemy II . . . 60
6.6 Compiling the BCVTB . . . 60
6.6.1 Compiling the BCVTB . . . 60
6.6.2 Custom configuration . . . 61
6.7 Structure of the file system . . . 61
6.8 Running unit tests . . . 61
6.9 Adding actors . . . 63
6.9.1 Adding actors to `lib/ptII/myActors` . . . 63
6.9.2 Adding actors to other directories . . . 63
6.10 Linking a simulation program to the BCVTB . . . 64
6.11 Data exchange between Ptolemy II and programs that are started by the Simulator actor . . . 66

7 Acknowledgements **68**

8 Bibliography **69**

Chapter 1

Introduction

This user manual explains how to install, use and further develop the BCVTB version 1.0.0.

The Building Controls Virtual Test Bed (BCVTB) is a software environment that allows expert users to couple different simulation programs for co-simulation. For example, the BCVTB allows the simulation of a building and HVAC system in EnergyPlus and the control logic in Modelica or in MATLAB/Simulink, while exchanging data between the software as they simulate. A system model for such a coupled simulation is shown in Figure 1.1.

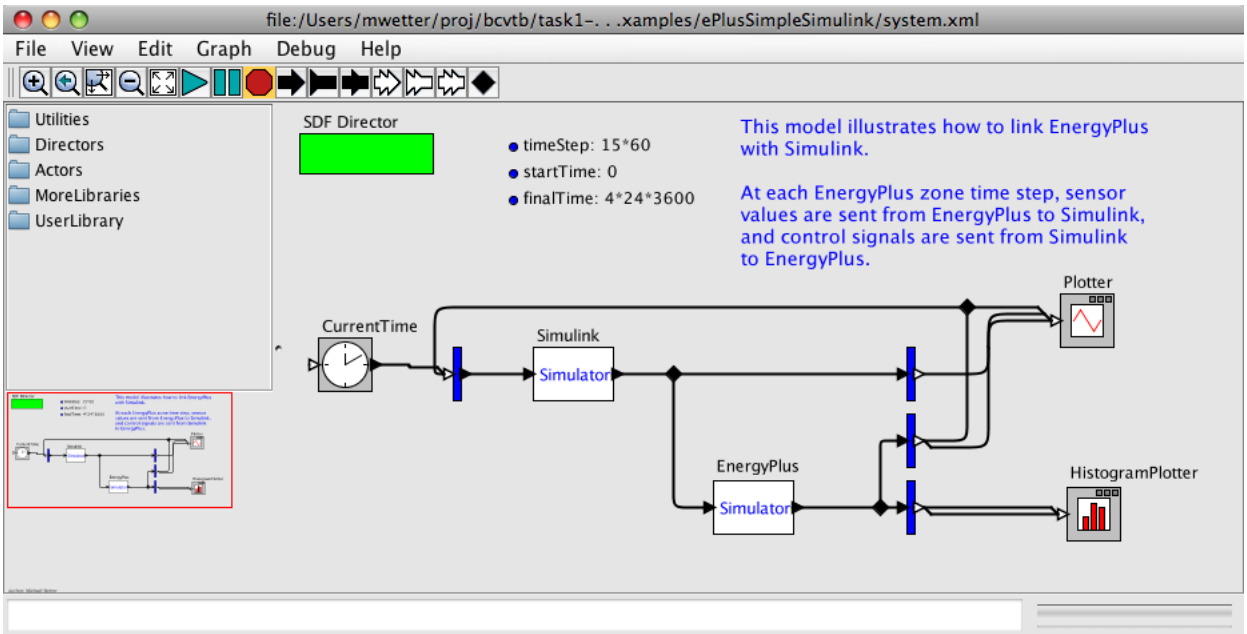

Figure 1.1: System model that links EnergyPlus with Simulink.

The BCVTB is based on the *Ptolemy II* software environment that has been developed by the University of California at Berkeley. The BCVTB is aimed at expert users of simulation that hit limitations of existing simulation programs.

Programs that are currently linked to the BCVTB are

- *EnergyPlus,*
- *Dymola,* which is a *Modelica* modeling and simulation environment,

- *Radiance,*
- *MATLAB,*
- *Simulink,*
- the *BACnet stack,* which is an open-source implementation that allows exchanging data with *BACnet* compatible building automation systems for use of models during operation for fault detection and diagnostics or for model-based operation, and
- an *analog/digital interface* that can be connected to a USB port.

In addition, any executable can be called from the BCVTB at each time step of the BCVTB system model. This allows, for example, the use of an external program with communication through its command line interface or through text files.

In addition to using programs that are coupled to Ptolemy II, Ptolemy II's graphical modeling environment can be used to define models of control systems, models of physical devices, models of communication systems or it can be used for post-processing, real-time visualization and data exchange with databases.

Chapter 2

Installation and configuration

2.1 Introduction

This chapter describes how to install, configure and uninstall the BCVTB. Users who are interested in further developing the BCVTB should also follow the installation described in Section 6.3 *Software requirements*.

2.2 Installation

To install the BCVTB, follow these steps:

1. Download the installation file `bcvtb-install-os-x.y.z.jar` from the download page, where `os` denotes the version of the operating system and `x.y.z` denotes the latest version number.
2. Run the installation program `bcvtb-install-os-x.y.z.jar`.
3. Depending on your installation, you may need to set system environment variables as described in Section 2.3 *Setting system environment variables*.
4. Test the installation by running an example as described in Chapter 3 *Running simulations with the BCVTB*.

Note
This manual assumes that the BCVTB is installed in a directory called `bcvtb`. However, the BCVTB may be installed in any directory. (To run the examples provided with the BCVTB, write permission are required for the directory `bcvtb/examples`.)

2.3 Setting system environment variables

When the BCVTB starts, it reads environment variables from the file `bcvtb/bin/systemVariables-*.properties`, where `*` is the name of the operating system. This file needs to be modified by the user to set the path to the different programs that are used by the BCVTB. The file can be edited with any text editor. It has the following syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<-- This is a comment line -->

<properties>
  <entry key="name">value</entry>
  <entry key="name">value</entry>
</properties>
```

The `properties` section contains the environment variables. The attribute `name` is the name of any environment variable that either exists or that should be set by the BCVTB, and `value` is the new value of this environment variable. There can be any number of environment variables. For example, to set a new environment variable `myProgramBin=C:\myProg` (or `myProgramBin=/usr/local/myProg` on Mac or Linux) and add this variable to the existing %Path% (or $PATH) variable, proceed as follows:

- in Windows, add the lines

  ```
  <entry key="myProgramBin">C:\myProg</entry>
  <entry key="Path">%myProgramBin%;%Path%</entry>
  ```

  to `bcvtb/bin/systemVariables-windows.properties`.

- in Mac OS X, add the lines

  ```
  <entry key="myProgramBin">/usr/local/myProg</entry>
  <entry key="PATH">$myProgramBin:$PATH</entry>
  ```

  to `bcvtb/bin/systemVariables-mac.properties`.

- in Linux, add the same lines as for Mac OS X to `bcvtb/bin/systemVariables-linux.properties`.

Next, restart the BCVTB. To see the new environment variables, type

```
java -jar BCVTB.jar -diagnostics
```

on a console.

2.4 Uninstallation

The BCVTB installation does not write to any file outside the directory `bcvtb`. To uninstall the BCVTB, simply delete the directory `bcvtb`.

Chapter 3

Running simulations with the BCVTB

3.1 Introduction

This chapter describes how to run the BCVTB from a file explorer window or from the command line, using several command line arguments. These command line arguments allow, for example, to overwrite the values of parameters that are defined in a Ptolemy II model or to run Ptolemy II in a console mode that does not open any windows.

Note
At start up, the BCVTB reads system environment variables from the file `bcvtb/bin/systemVariables-*-.properties`, where `*` is either `windows`, `mac` or `linux`. This file may need to be updated by the user to set system environment variables. For a description of this file, see Section 2.3 *Setting system environment variables* .

3.2 Running the BCVTB from an explorer window

To run an example from an explorer window, double-click the file `bcvtb/bin/BCVTB.jar`. This will open a window. From the window, either select any of the examples, or select from the pull-down menu the entry `File -> Open` and open, for example, the file `bcvtb/examples/c-room/system.xml`

To run the example, press the button with the green arrow. You should see an online plot showing the room temperatures and control signals.

Note that some examples have a file `system.xml` and `system-windows.xml`. For these examples, use the file `system-windows.xml` for Windows and `system.xml` for Linux and Mac OS X.

3.3 Running the BCVTB from a terminal window

To run an example from a console (i.e., a dos-shell on Windows or a terminal on Mac OS X or Linux), proceed as follows:

Open a console and change to the directory `bcvtb`.

Type, for example,

```
java -jar bin/BCVTB.jar examples/c-room/system.xml
```

Note that some examples have a file `system.xml` and `system-windows.xml`. For these examples, use the file `system-windows.xml` for Windows and `system.xml` for Linux and Mac OS X.

To run the example, press the button with the green arrow. You should see an online plot showing the room temperatures and control signals.

The file `BCVTB.jar` can be run with several optional flags which are described in Section 3.4 *Command line arguments* .

3.4 Command line arguments

To start the BCVTB from a console, type

```
java -jar ["path_to_bcvtb/bin/"]BCVTB.jar [JVM_options] [BCVTB_options [- ↩
    parameterName value]]
```

where `JVM_options` can be any Java Virtual Machine options (type `java -h` for available options), and `BCVTB_options` include:

`-file fileName`	Open `fileName`, which must be a Ptolemy II file. The flag `-file` is optional if `fileName` is the last parameter.
`-run fileName`	Open `fileName`, which must be a Ptolemy II file, run the program, and terminate.
`-console fileName`	Open `fileName`, which must be a Ptolemy II file, run the program without opening any windows, and terminate.
`-diagnostics`	Print all environment variables to the console window.
`-command program flags`	Runs the executable `program` with the flags `flags`. Any number of flags are allowed. This allows for example to start a new console that has the same environment variables as any other program started by the BCVTB.

The optional argument `-parameterName value` are model parameters and their values, such as `-finalTime 3600`. Note that a hyphen must precede the keyword `parameterName`.

Example 3.1 Examples for command line arguments

1. To run the file `system.xml` and terminate the BCVTB after the simulation, type

   ```
   java -jar BCVTB.jar -run system.xml
   ```

   or, alternatively, type

   ```
   java -jar BCVTB.jar -file system.xml -run
   ```

2. To run `system.xml` as a console application that does not open any windows, type

   ```
   java -jar BCVTB.jar -console system.xml
   ```

3. If the model `system.xml` has a top-level parameter named `finalTime` and an actor with name `Controller`, and `Controller` contains a parameter named `Kp` then

   ```
   java -jar BCVTB.jar -run system.xml -finalTime 86400 -Controller.Kp 10
   ```

   runs the model `system.xml` up to `finalTime=86400`, with the parameter `Kp` of the controller set to `10`.

4. If Java runs out of memory, type

   ```
   java -jar BCVTB.jar -Xmx1024m system.xml
   ```

   to run `system.xml` with increased Java heap size.

5. On Linux, to set environment variables and open a new terminal that can be used to run the Apache Ant build files (see Section 6.6 *Compiling the BCVTB*), type

   ```
   java -jar BCVTB.jar -command xterm
   ```

3.5 Example files

The directory `bcvtb/examples` contains several example files that show how to use the BCVTB. In these examples, the following programs are linked to the BCVTB:

- EnergyPlus,
- MATLAB,
- Simulink,
- Dymola,
- a simulation program implemented in C,
- a simulation program implemented in Fortran 90, and
- a program written in the C language that is called at each time step with different program arguments.

The C and Fortran 90 simulation programs are provided to show developers how to couple a new program to the BCVTB. The program that is called at each time step with different program arguments shows how programs can be

called and how their output files can be parsed. The BCVTB also contains examples that show how models of control systems and of HVAC systems can be implemented directly in Ptolemy II using Ptolemy II's graphical model editor. The examples of control systems include a heterogeneous system consisting of a discrete time controller with a Finite State Machine.

One of the simplest examples can be found in the directory `bcvtb/examples/c-room`. This example illustrates the implementation of a simulation program written in C that communicates with Ptolemy II through BSD sockets. The simulation program computes the temperature change in two rooms with different heat capacities. Input to the simulation program is the vector of control signals u_k . Output of the simulation program are the vector of new room temperatures T_{k+1} . The control action is computed in Ptolemy II. Figure 3.1 shows the Ptolemy II model.

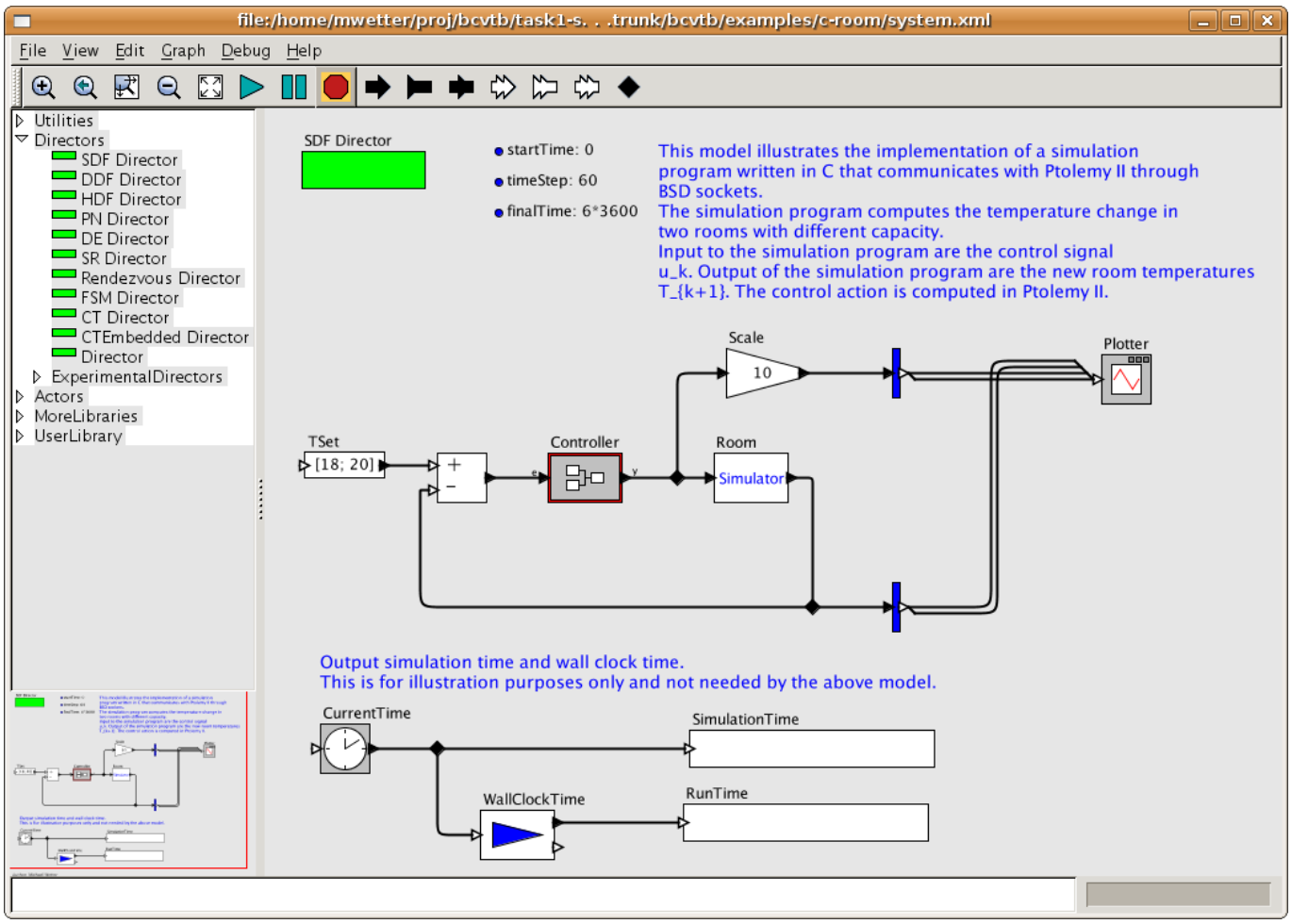

Figure 3.1: System model that links Ptolemy II to a room model that is implemented in the C language.

Chapter 4

Configuring programs for use with the BCVTB

4.1 Introduction

This chapter describes how to configure a simulation model for use with the BCVTB. Before configuring your own simulation, it is recommended to run and modify the examples in the folder `bcvtb/examples` as described in Chapter 3 *Running simulations with the BCVTB*.

Setting up an own simulation is easiest if one of the examples in the folder `bcvtb/examples` is used as a starting point. Configuring a simulation with the BCVTB consists of the following steps, which are described in the next sections:

1. Create a Ptolemy II model. This is described in Section 4.2 *Ptolemy II*.
2. Create and configure a simulation model by following the instructions described in
   a. Section 4.3 *EnergyPlus*,
   b. Section 4.4 *Dymola*,
   c. Section 4.5 *MATLAB*,
   d. Section 4.6 *Simulink*,
   e. Section 4.7 *Custom program using a system command*,
   f. Section 4.8 *Radiance*.
3. Run the Ptolemy II model created in step 1.

4.2 Ptolemy II

For Ptolemy II related information, we recommend to read the Ptolemy II web page and the Ptolemy II tutorial from UC Berkeley.

In Ptolemy II, different *models of computations* can be used to define how the different actors interact with each other. The model of computation is defined by a *director* that needs to be included in the Ptolemy II flow chart diagram. For the BCVTB, we typically use the Synchronous Dataflow director, which is in Ptolemy II called *SDF Director*. This director can be dragged into the model from the left pane shown in Figure 4.1 .

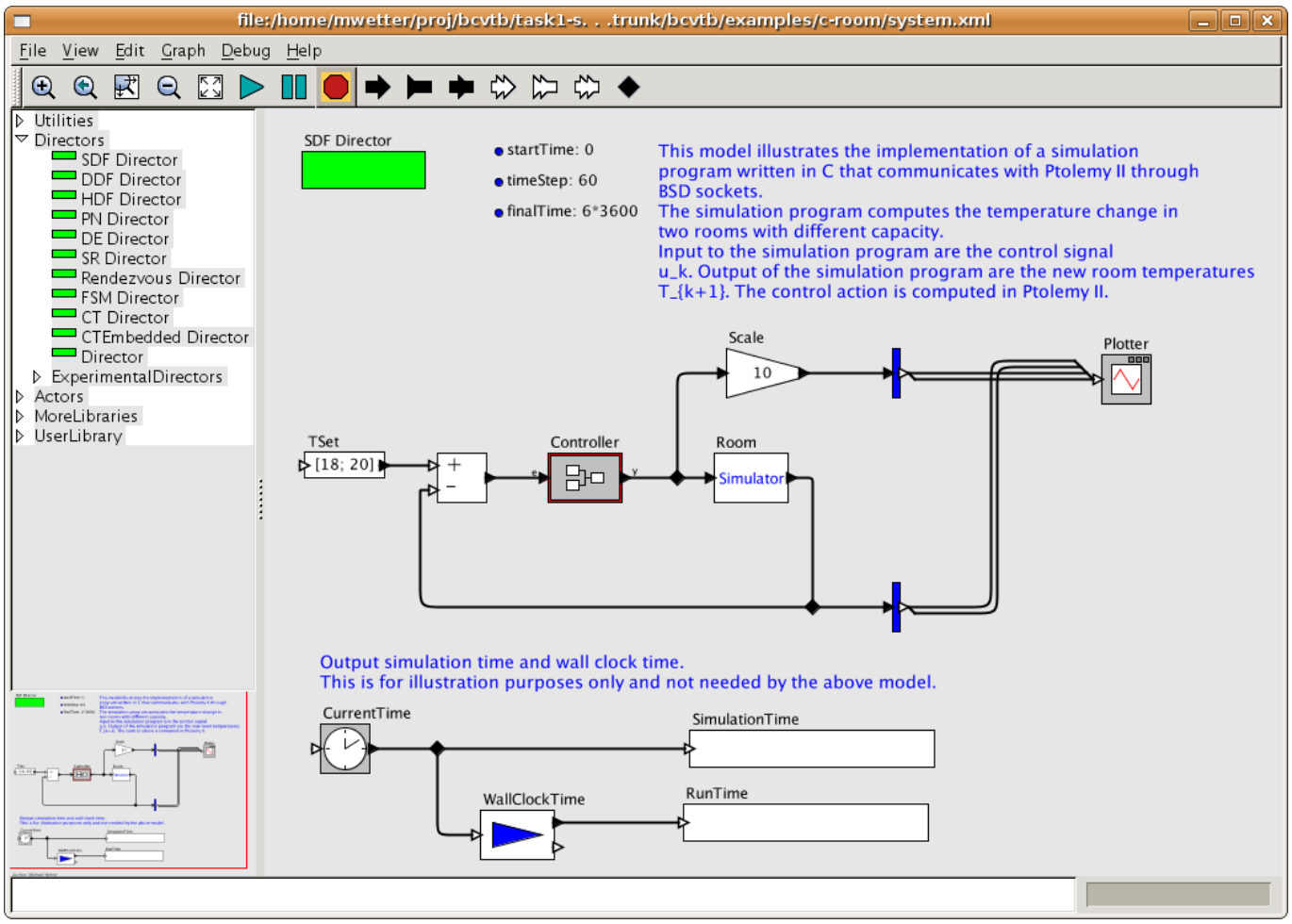

Figure 4.1: Ptolemy II system model that connects a model of a controller and a room.

For convenience, the examples in the BCVTB expose the three parameters `startTime`, `timeStep` and `finalTime`. These three parameters have units of seconds and needs to correspond with the start time, time step and final time that is used in the simulation program. The parameters used to configure the SDF Director are shown in Figure 4.2 .

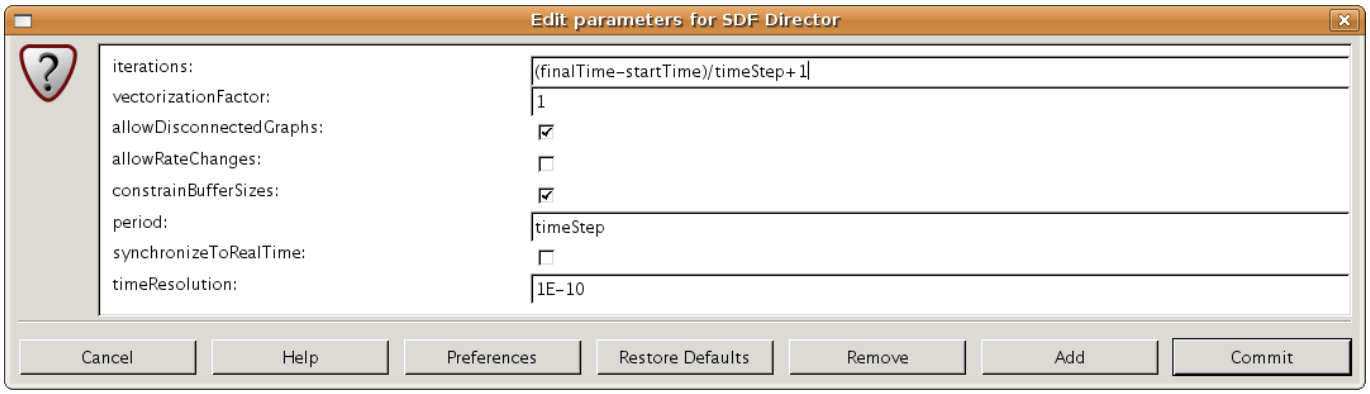

Figure 4.2: Configuration of the SDF director.

Flow charts with the SDF director must not contain algebraic loops. If there is an algebraic loop, then a sample delay actor needs to be inserted. This actor can be found in the Ptolemy II actor library in (`Actors->FlowControl->SequenceControl->SampleDelay`). Figure 4.3 shows the use of a `SampleDelay` actor for delaying the output of a controller by one sampling interval.

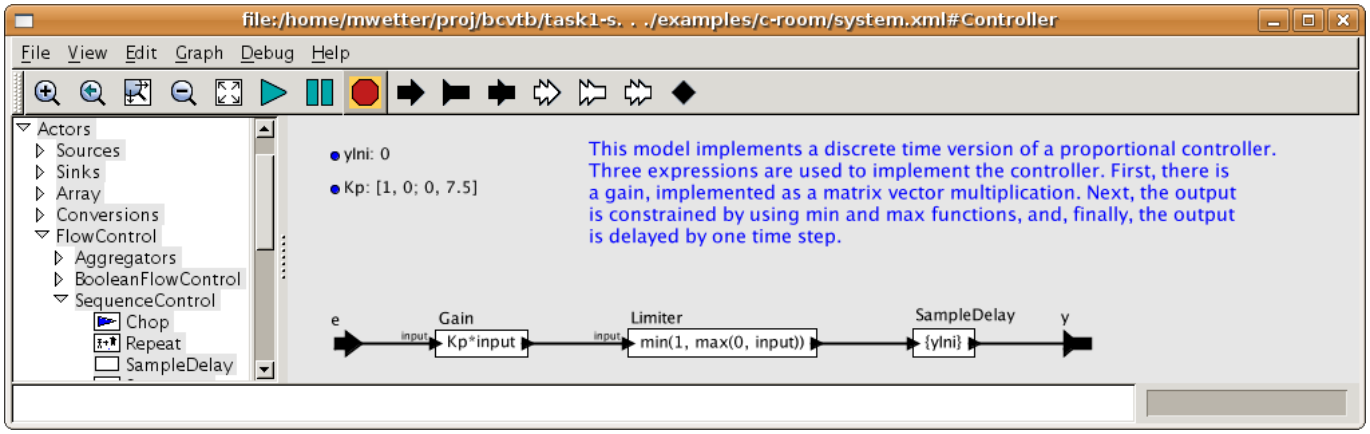

Figure 4.3: Implementation of the controller with the `SampleDelay` actor that delays its output by one sampling interval. This delay eliminates an algebraic loop, which is not allowed with the SDF director.

The `Simulator` actor conducts the data exchange with the simulation program. There can be any number of `Simulator` actors in a model. The parameters of the `Simulator` actor are as shown in Table 4.1 .

Parameter	**Description**
programName	The name of the executable that starts the simulation.
programArguments	Arguments needed by the simulation. Text arguments need to be enclosed in apostrophes.
workingDirectory	Working directory of the program. For the current directory, enter a period.
simulationLogFile	Name of the file to which the BCVTB will write the console output and error stream that it receives from the simulation program. Use a separate file for each simulation program. This file typically shows what may have caused an error.
socketTimeout	Time out in milliseconds for the initial socket connection. At the start of the simulation, the BCVTB waits for the simulation program to connect through a socket connection to the BCVTB. If the simulation program does not connect within the here specified time, the BCVTB will stop with an error.
showConsoleWindow	Check box; if activated, a separate window will be opened that displays the console output of the program.

Table 4.1: Parameters of the `Simulator` actor.

An example that starts EnergyPlus on Linux and Mac is shown in Figure 4.4 .

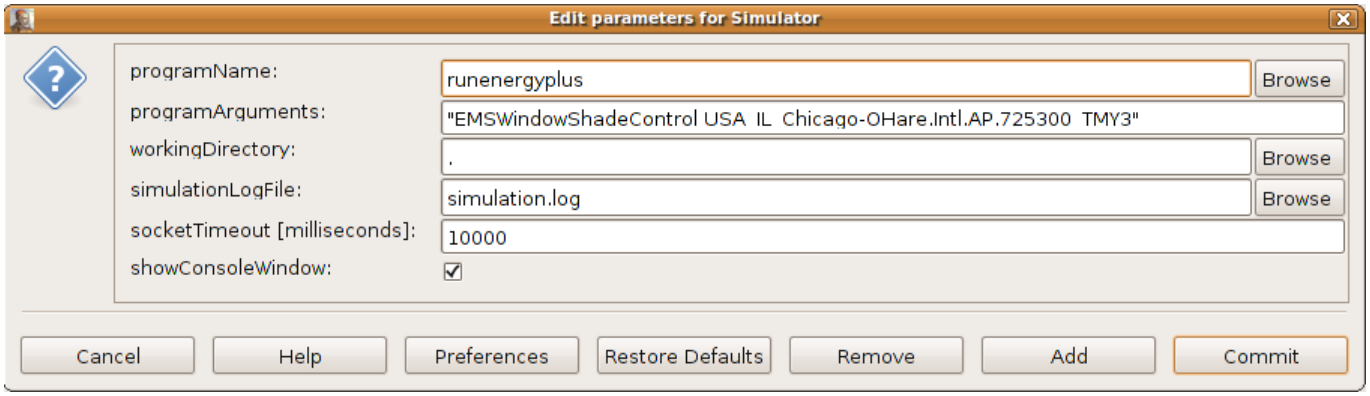

Figure 4.4: Configuration of the `Simulator` actor that starts EnergyPlus on Linux.

4.3 EnergyPlus

Figure 4.5 shows the architecture of the connection between EnergyPlus and Ptolemy II. Ptolemy II connects to the external interface in EnergyPlus. In the external interface, the input/output signals that are exchanged between Ptolemy II and EnergyPlus are mapped to EnergyPlus objects. The subject of this section is to show how to configure this mapping and how to use these objects.

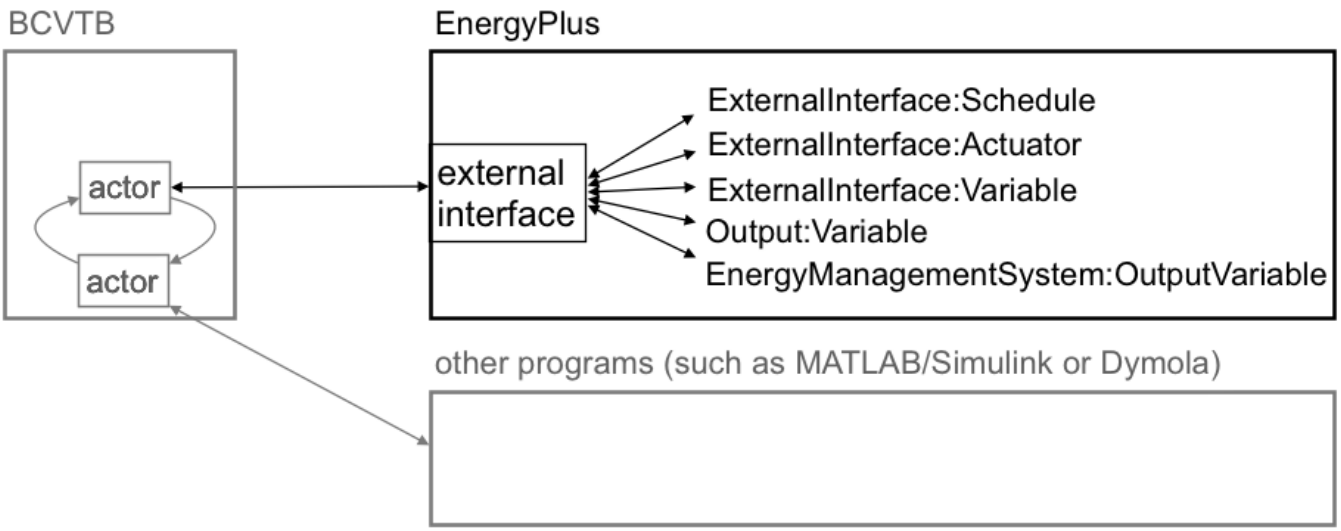

Figure 4.5: Architecture of the BCVTB with the EnergyPlus client (black) and other clients (grey).

The external interface can map to three EnergyPlus input objects called `ExternalInterface:Schedule`, `ExternalInterface:Actuator` and `ExternalInterface:Variable`. The `ExternalInterface:Schedule` can be used to overwrite schedules. The other two objects can be used in place of Energy Management System (EMS) actuators and EMS variables. The objects have similar functionality as the objects `Schedule:Compact`, `EnergyManagementSystem:Actuator` and `EnergyManagementSystem:GlobalVariable`, except that their numerical value is obtained from the external interface at the beginning of each EnergyPlus zone time step, and will remain constant during this zone time step.

Compared to `EnergyManagementSystem:Actuator`, the object `ExternalInterface:Actuator` has an optional field called "initial value." If a value is specified for this field, then this value will be used during the warm-up period and the system sizing. If unspecified, then the numerical value for this object will only be used during the time stepping. 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. For the objects `ExternalInterface:Schedule` and `ExternalInterface:Variable`, the field "initial value" is required, and its value will be used during the warm-up period and the system-sizing.

`ExternalInterface:Variable` is a global variable from the point of view of the EMS language. Thus, it can be used within any `EnergyManagementSystem:Program` in the same way as an `EnergyManagementSystem:GlobalVariable` or an `EnergyManagementSystem:Sensor` can be used.

Although variables of type `ExternalInterface:Variable` can be assigned to `EnergyManagementSystem:Actuator` objects, for convenience, there is also an object called `ExternalInterface:Actuator`. This object behaves identically to `EnergyManagementSystem:Actuator`, with the following exceptions:

- Its value is assigned by the external interface.
- Its value is fixed during the zone time step because this is the synchronization time step for the external interface.

The external interface can also map to the EnergyPlus objects `Output:Variable` and `EnergyManagementSystem:OutputVariable`. These objects can be used to send data from EnergyPlus to Ptolemy II at each zone time step.

We will now present examples that use all of these objects. Table 4.2 shows which EnergyPlus features are used in the examples.

	Example 1	Example 2	Example 3
`ExternalInterface:Schedule`	x		
`ExternalInterface:Actuator`		x	
`ExternalInterface:Variable`			x
`Output:Variable`	x	x	x
`EnergyManagementSystem:OutputVariable`			x

Table 4.2: Overview of the EnergyPlus objects used in the examples.

To configure the data exchange, the following three steps are required from the user:

1. Create an EnergyPlus idf file.
2. Create an xml file that defines the mapping between EnergyPlus and BCVTB variables.
3. Create a Ptolemy II model.

These steps are described in the examples below. Prior to discussing the examples, we will explain the syntax of the xml configuration file that defines how data are mapped between the external interface and EnergyPlus

4.3.1 Syntax of the xml file that configures the data mapping between EnergyPlus and the external interface

The data mapping between EnergyPlus and the external interface is defined in an xml file called `variables.cfg`. This file needs to be in the same directory as the EnergyPlus idf file. The file has the following header:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE BCVTB-variables SYSTEM "variables.dtd">
```

Following the header is an element of the form

```
<BCVTB-variables>

</BCVTB-variables>
```

This element will contain child elements that define the variable mapping. In between the element tags, a user needs to specify how the exchanged data is mapped to EnergyPlus objects. Hence, the order of these elements matter, and it need to be the same as the order of the elements in the input and output signal vector of Ptolemy II actor that calls EnergyPlus. The exchanged variables are declared in elements that are called "variable" and have an attribute "source." As described above, the external interface can send data to `ExternalInterface:Schedule`, `ExternalInterface:Actuator`, and `ExternalInterface:Variable`. For these objects, the `source` attribute needs to be set to `Ptolemy`, because they are sent by Ptolemy II. The xml elements for these objects are defined as follows:

- For `ExternalInterface:Schedule`, use

  ```
  <variable source="Ptolemy">
      <EnergyPlus schedule="NAME"/>
  </variable>
  ```

  where `NAME` needs to be the EnergyPlus schedule name. See Section 4.3.2 *Example 1: Interface using ExternalInterface:Schedule* for an example.

- For `ExternalInterface:Actuator`, use

  ```
  <variable source="Ptolemy">
      <EnergyPlus actuator="NAME" />
  </variable>
  ```

  where `NAME` needs to be the EnergyPlus actuator name. See Section 4.3.3 *Example 2: Interface using ExternalInterface:Actuator* for an example.

- For `ExternalInterface:Variable`, use

  ```
  <variable source="Ptolemy">
      <EnergyPlus variable="NAME"/>
  </variable>
  ```

  where `NAME` needs to be the EnergyPlus Energy Runtime Language (Erl) variable name. See Section 4.3.4 *Example 3: Interface using ExternalInterface:Variable* for an example.

The external interface can also read data from any `Output:Variable` and `EnergyManagementSystem:OutputVariable`. For these objects, set the "source" attribute to "EnergyPlus," because they are computed by EnergyPlus.

- The read an `Output:Variable`, use

```
<variable source="EnergyPlus">
  <EnergyPlus name="NAME" type="TYPE"/>
</variable>
```

  where `NAME` needs to be the EnergyPlus key value (which is typically the name of the EnergyPlus object instance, such as `WEST ZONE`) and `TYPE` needs to be the EnergyPlus variable (such as ZONE/SYS AIR TEMP). See the following sections for an example.

- To read an `EnergyManagementSystem:OutputVariable`, use

```
<variable source="EnergyPlus">
    <EnergyPlus name="EMS" type="TYPE"/>
</variable>
```

  i.e., the attribute `name` must be EMS, and the attribute `type` must be set to the EMS variable name. See Section 4.3.4 *Example 3: Interface using ExternalInterface:Variable* for an example.

The following sections present examples of this xml file.

4.3.2 Example 1: Interface using `ExternalInterface:Schedule`

In this example, a controller that is implemented in Ptolemy II computes the room temperature set points for cooling and heating. The example can be found in the BCVTB distribution in the folder `bcvtb/examples/ePlusX--schedule`, where `X` stands for the EnergyPlus version number. Suppose we need to send a schedule value from Ptolemy II to EnergyPlus, and an output variable from EnergyPlus to Ptolemy II at each zone time step. This can be accomplished by using an object of type `ExternalInterface:Schedule` and an object of type `Output:Variable`. To interface EnergyPlus using the EMS feature, the following three items are needed:

- An object that instructs EnergyPlus to activate the external interface.
- EnergyPlus objects that write data from the external interface to the EMS.
- A configuration file to configure the data exchange.

The following sections explain how to declare these items.

4.3.2.1 Create an EnergyPlus idf file

The EnergyPlus idf file contains the following objects to activate and use the external interface:

- An object that instructs EnergyPlus to activate the external interface.
- An object of type `ExternalInterface:Schedule`. The external interface will write its values to these objects at each zone time-step.
- Objects of type `Output:Variable` that store the data that will be read by the external interface. The value of any EnergyPlus `Output:Variable` can be read by the external interface.

The code below shows how to declare these objects. To activate the external interface, we use:

```
ExternalInterface,           !- Object to activate the external interface
 PtolemyServer;              !- Name of external interface
```

To enter schedules to which the external interface writes, we use:

```
! Cooling schedule. This schedule is set directly by the external interface.
! During warm-up and system-sizing, it is fixed at 24 degC.
  ExternalInterface:Schedule,
    TSetCoo,                 !- Name
    Temperature,             !- ScheduleType
    24;                      !- Initial Value, used during warm-up

! Heating schedule. This schedule is set directly by the external interface.
! During warm-up and system-sizing, it is fixed at 20 degC.
  ExternalInterface:Schedule,
    TSetHea,                 !- Name
    Temperature,             !- ScheduleType
    20;                      !- Initial Value, used during warm-up
```

These schedules can be used in the same way as other EnergyPlus schedules. In this example, they are used to change a thermostat setpoint:

```
ThermostatSetpoint:DualSetpoint,
    DualSetPoint,            !- Name
    TSetHea,                 !- Heating Setpoint Temperature Schedule Name
    TSetCoo;                 !- Cooling Setpoint Temperature Schedule Name
```

We also want to read output variables from EnergyPlus, which we declare as

```
Output:Variable,
    TSetHea,        !- Key Value
    Schedule Value, !- Variable Name
    TimeStep;       !- Reporting Frequency

Output:Variable,
    TSetCoo,        !- Key Value
    Schedule Value, !- Variable Name
    TimeStep;       !- Reporting Frequency
```

To specify that data should be exchanged every 15 minutes of simulation time, we enter in the `idf` file the section

```
Timestep,
    4;          !- Number of Timesteps per Hour
```

4.3.2.2 Create a configuration file

Note that we have not yet specified the order of the elements in the signal vector that is exchanged between EnergyPlus and Ptolemy II. This information is specified in the file `variables.cfg`. The file `variables.cfg` needs to be in the same directory as the EnergyPlus `idf` file. For the objects used in the section above, the file looks like

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE BCVTB-variables SYSTEM "variables.dtd">
<BCVTB-variables>
  <!-- The next two elements send the set points to E+ -->
  <variable source="Ptolemy">
```

```
    <EnergyPlus schedule="TSetHea"/>
  </variable>
  <variable source="Ptolemy">
    <EnergyPlus schedule="TSetCoo"/>
  </variable>
  <!-- The next two elements receive the outdoor and
       the zone air temperature from E+ -->
  <variable source="EnergyPlus">
   <EnergyPlus name="ENVIRONMENT" type="OUTDOOR DRY BULB"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="ZSF1" type="ZONE/SYS AIR TEMPERATURE"/>
  </variable>
  <!-- The next two elements receive the schedule value as an output from E+ -->
  <variable source="EnergyPlus">
    <EnergyPlus name="TSetHea" type="Schedule Value"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="TSetCoo" type="Schedule Value"/>
  </variable>
</BCVTB-variables>
```

This file specifies that the actor that calls EnergyPlus has an input vector with two elements that are computed by Ptolemy II and sent to EnergyPlus, and that it has an output vector with four elements that are computed by EnergyPlus and sent to Ptolemy II. The order of the elements in each vector is determined by the order in the above XML file. Hence, the input vector that contains the signals sent to EnergyPlus has elements

```
  TSetHea
  TSetCoo
```

and the output vector that contains values computed by EnergyPlus has elements

```
  Environment (Outdoor drybulb temperature)
  ZSF1 (ZONE/SYS AIR TEMPERATURE)
  TSetHea (Schedule Value)
  TSetCoo (Schedule Value)
```

4.3.2.3 Create a Ptolemy II model

To start EnergyPlus from the BCVTB, you will need to create a Ptolemy II model.

The model `bcvtb/examples/ePlus*-schedule/system-windows.xml`, which is part of the BCVTB installation and is shown in Figure 4.6 , may be used as a starting point. (For Mac and Linux, use the file `system.-xml`.) In this example, the time step is 15 minutes and the simulation period is four days.

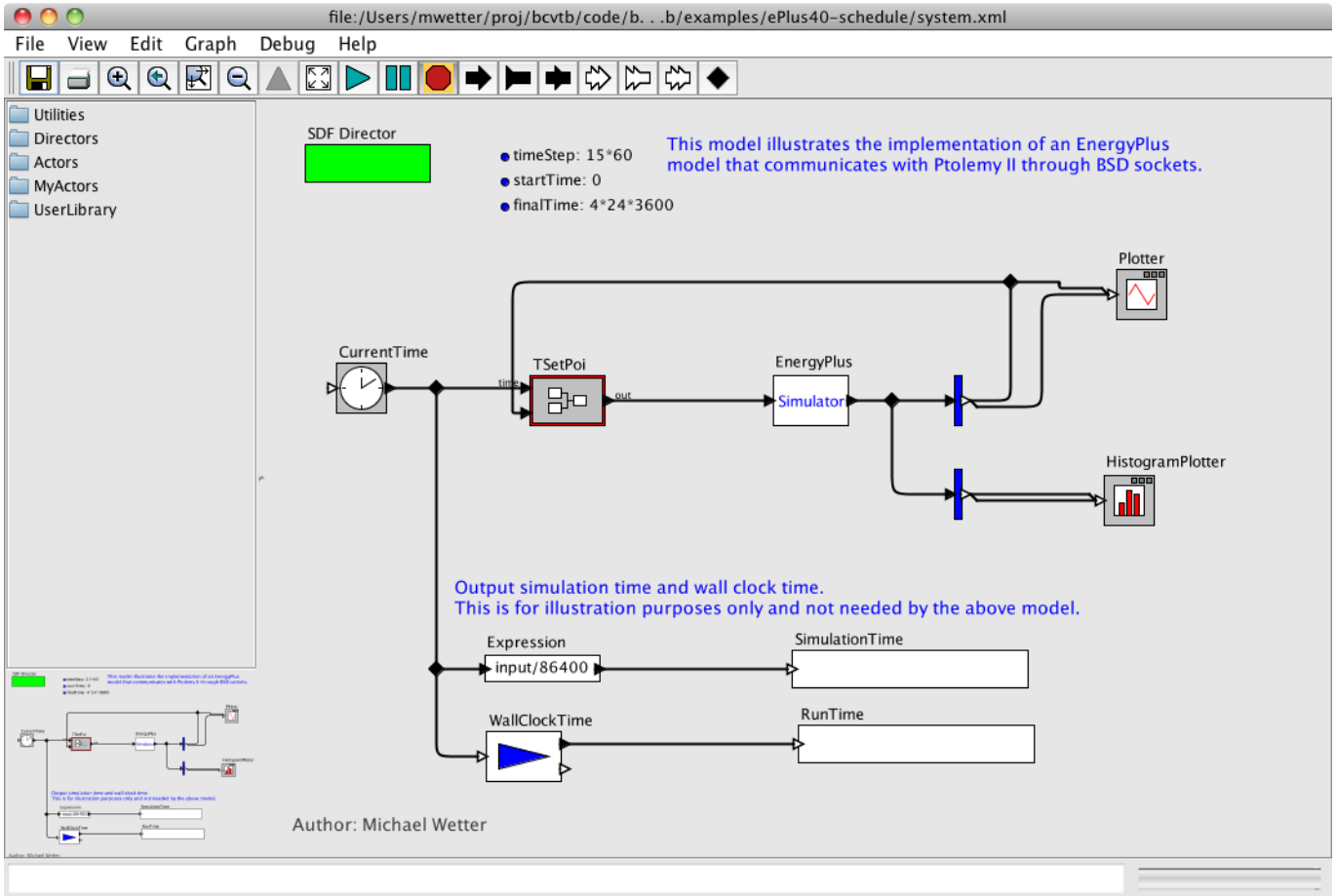

Figure 4.6: Ptolemy II system model that links an actor that computes the room temperature setpoint with the `Simulator` actor that communicates with EnergyPlus.

In this model, the Simulator actor that calls EnergyPlus is configured for Windows as shown in Figure 4.7 .

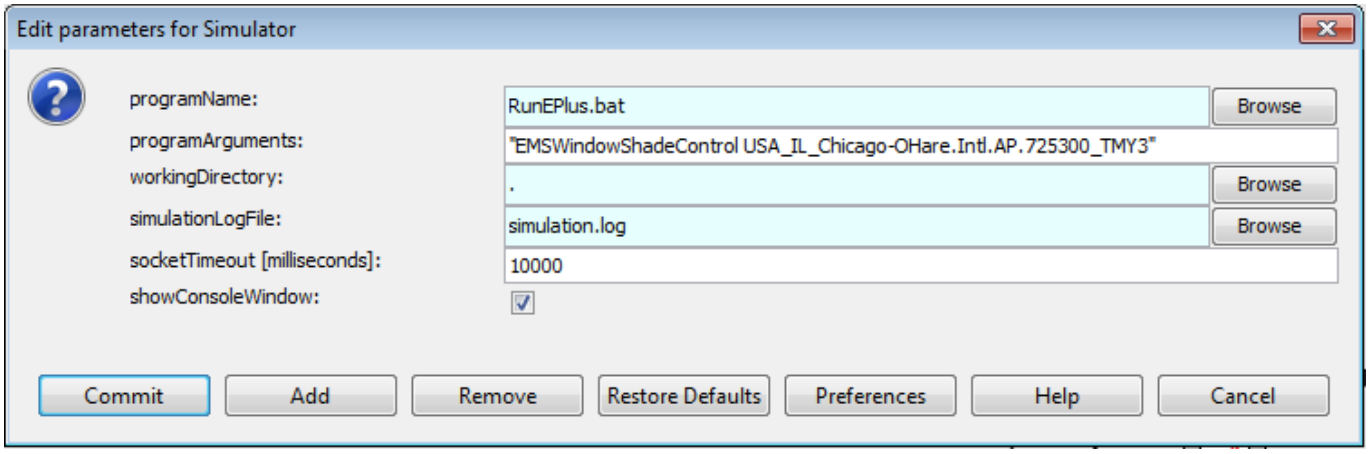

Figure 4.7: Configuration of the `Simulator` actor that calls EnergyPlus on Windows.

The actor calls the file `RunEPlus.bat`, with arguments `EMSWindowShadeControl USA_IL_Chicago-O-`

`Hare.Intl.AP.725300_TMY3`. The working directory is the current directory and the console output is written to the file `simulation.log`. If EnergyPlus does not communicate with Ptolemy II within 10 seconds, Ptolemy II will terminate the connection. For Mac OS X and Linux, the configuration is similar:

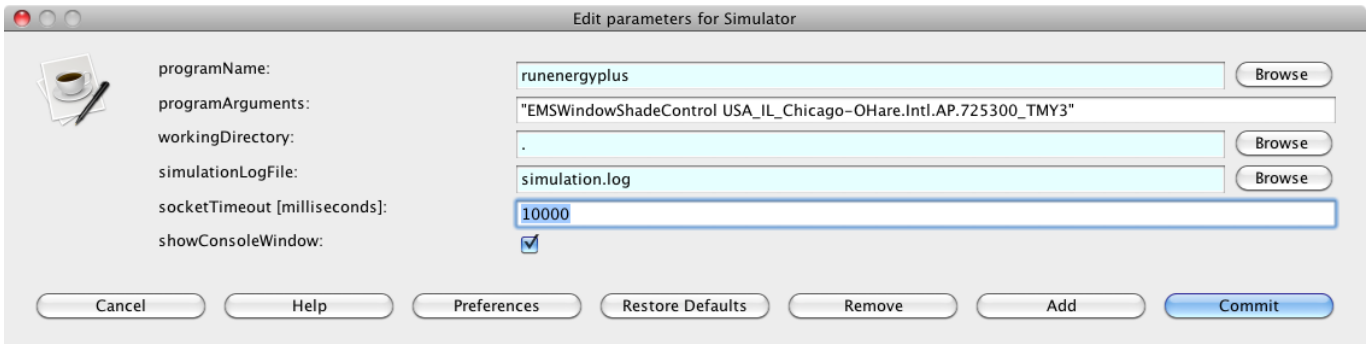

Figure 4.8: Configuration of the `Simulator` actor that calls EnergyPlus on Mac OS X.

This completes the configuration.

4.3.3 Example 2: Interface using `ExternalInterface:Actuator`

In this example, a shading controller with a finite state machine is implemented in Ptolemy II. Inputs to the controller are the outside temperature and the solar radiation that is incident on the window. The output of the controller is the shading actuation signal. This example describes how to set up EnergyPlus to exchange data between Ptolemy II and EnergyPlus, using an Energy Management System (EMS) actuator. The example can be found in the BCVTB distribution in the folder `bcvtb/examples/ePlus*-actuator`, where `*` stands for the EnergyPlus version number. The object of type `ExternalInterface:Actuator` behaves identically to `EnergyManagementSystem:Actuator`, with the following exceptions:

1. Its value is assigned by the external interface.
2. Its value is fixed during the EnergyPlus zone time step because this is the synchronization time step for the external interface.

To interface EnergyPlus using the EMS feature, the following three items are needed:

- An object that instructs EnergyPlus to activate the external interface.
- EnergyPlus objects that write data from the external interface to the EMS.
- A configuration file to configure the data exchange.

The following sections explain how to declare these items.

4.3.3.1 Create an EnergyPlus idf file

The code below shows how to set up an EnergyPlus file that uses `EnergyManagmentSystem:Actuator`. To activate the external interface, we use:

```
ExternalInterface,            !- Object to activate the external interface
    PtolemyServer;            !- Name of external interface
```

To declare an actuator that changes the control status of the window with name `Zn001:Wall001:Win001`, we use:

```
ExternalInterface:Actuator,
    Zn001_Wall001_Win001_Shading_Deploy_Status,  !- Name
    Zn001:Wall001:Win001,     !- Actuated Component Unique Name
    Window Shading Control,   !- Actuated Component Type
    Control Status,           !- Actuated Component Control Type
     ;                        !- Optional Initial Value
```

Thus, the entry is identical with `EnergyManagementSystem:Actuator`, except for the additional optional field that specifies the initial value. If unspecified, then the actuator will only be used during the time stepping, but not during the warm-up and the 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.

We also want to read the outdoor temperature, the zone air temperature, the solar radiation that is incident on the window, and the fraction of time that the shading is on from EnergyPlus. Thus, we declare the output variables

```
Output:Variable,
   Environment,        !- Key Value
   Outdoor Dry Bulb,   !- Variable Name
   timestep;           !- Reporting Frequency

Output:Variable,
   *,                           !- Key Value
   Zone Mean Air Temperature,   !- Variable Name
   timestep;                    !- Reporting Frequency

Output:Variable,
   Zn001:Wall001:Win001,        !- Key Value
   Surface Ext Solar Incident,  !- Variable Name
   timestep;                    !- Reporting Frequency

Output:Variable,
   *,                                      !- Key Value
   Fraction of Time Shading Device Is On,  !- Variable Name
   timestep;                               !- Reporting Frequency
```

To specify that data should be exchanged every 10 minutes of simulation time, we enter in the idf file the section

```
  Timestep,
    6;           !- Number of Timesteps per Hour
```

4.3.3.2 Create a configuration file

Note that we have not yet specified the order of the elements in the signal vector that is exchanged between EnergyPlus and Ptolemy II. This information is specified in the file `variables.cfg`. The file `variables.cfg` needs to be in the same directory as the EnergyPlus idf file. For the objects used in the section above, the file looks like

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE BCVTB-variables SYSTEM "variables.dtd">
```

```
<BCVTB-variables>
  <variable source="EnergyPlus">
    <EnergyPlus name="ENVIRONMENT"
                type="OUTDOOR DRY BULB"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="WEST ZONE"
                type="Zone Mean Air Temperature"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="Zn001:Wall001:Win001"
                type="Surface Ext Solar Incident"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="Zn001:Wall001:Win001"
                type="Fraction of Time Shading Device Is On"/>
  </variable>
  <variable source="Ptolemy">
    <EnergyPlus actuator="Zn001_Wall001_Win001_Shading_Deploy_Status"/>
  </variable>
</BCVTB-variables>
```

This file specifies that the simulator actor that calls EnergyPlus has an input vector with one element that will be written to the actuator, and that it has an output vector with four elements that are computed by EnergyPlus and sent to Ptolemy II. The order of the elements in each vector is determined by the order in the above XML file. Hence, the output vector that contains the signals computed by EnergyPlus has elements

```
  ENVIRONMENT (OUTDOOR DRY BULB)
  WEST ZONE (Zone Mean Air Temperature)
  Zn001:Wall001:Win001 (Surface Ext Solar Incident)
  Zn001:Wall001:Win001 (Fraction of Time Shading Device Is On)
```

The configuration of the Ptolemy II model is identical to the configuration in Example 1., which is described in Section 4.3.2.3 *Create a Ptolemy II model* .

4.3.4 Example 3: Interface using `ExternalInterface:Variable`

This example implements the same controller as the Example 2. However, the interface with EnergyPlus is done using an external interface variable instead of an external interface actuator. In addition, to set up data that will be read by the external interface, the example uses an `EnergyManagementSystem:OutputVariable`.

Similarly to `EnergyManagementSystem:GlobalVariable`, an `ExternalInterface:Variable` can be used in any `EnergyManagementSystem:Program`. The subject of this example is to illustrate how an `ExternalInterface:Variable` can be set up for use in an `EnergyManagementSystem:Program`. The example can be found in the BCVTB distribution in the folder `bcvtb/examples/ePlus*-variable`, where `*` stands for the EnergyPlus version number.

To interface EnergyPlus using an external interface variable, the following items are needed:

- An object that instructs EnergyPlus to activate the external interface.
- EnergyPlus objects that write data from the external interface to the EMS.
- A configuration file to configure the data exchange.

4.3.4.1 Create an EnergyPlus idf file

The following sections explain how to declare these items.

To write data from the external interface to an EnergyPlus EMS variable, the following EnergyPlus objects may be declared in the `idf` file:

```
ExternalInterface,         !- Object to activate the external interface
   PtolemyServer;          !- Name of external interface

  ExternalInterface:Variable,
    yShade,           !- Name of Erl variable
    1;                !- Initial Value
```

The above `idf` section activates the external interface and declares a variable with name `yShade` that can be used in an Erl program. During the warm-up period and the system-sizing, the variable `yShade` will be set to its initial value. Afterwards, the value will be assigned from the external interface at each beginning of a zone time step and kept constant during the zone time step. From the point of view of the EMS language, `ExternalInterface:Variable` can be used like any global variable. Thus, it can be used within any `EnergyManagementSystem:Program` in the same way as an `EnergyManagementSystem:GlobalVariable` or an `EnergyManagementSystem:Sensor`. The following `idf` section uses `yShade` to actuate the shading control of the window `Zn001:Wall001:Win001`:

```
! EMS program. The first assignments sets the shading status
!              and converts it into the
!              EnergyPlus signal (i.e., replace 1 by 6).
!              The second assignment sets yShade to
!              an EnergyManagementSystem:OutputVariable
!              which will be read by the external interface.
  EnergyManagementSystem:Program,
    Set_Shade_Control_State,          !- Name
    Set Shade_Signal = 6*yShade,      !- Program Line 1
    Set Shade_Signal_01 = yShade+0.1; !- Program Line 2

! Declare an actuator to which the EnergyManagementSystem:Program will write
  EnergyManagementSystem:Actuator,
    Shade_Signal,  !- Name
    Zn001:Wall001:Win001,    !- Actuated Component Unique Name
    Window Shading Control,  !- Actuated Component Type
    Control Status;          !- Actuated Component Control Type

! Declare a global variable to which the EnergyManagementSystem:Program will write
  EnergyManagementSystem:GlobalVariable,
    Shade_Signal_01;           !- Name of Erl variable
```

Next, suppose we want to read the outdoor temperature, the zone air temperature and the solar radiation that is incident on the window. In addition, we want to read the variable `Erl Shading Control Status`. This can be done with the following declaration:

```
Output:Variable,
   Environment,       !- Key Value
   Outdoor Dry Bulb,  !- Variable Name
   timestep;          !- Reporting Frequency

  Output:Variable,
  *,                          !- Key Value
```

```
  Zone Mean Air Temperature,  !- Variable Name
  timestep;                   !- Reporting Frequency

  Output:Variable,
  Zn001:Wall001:Win001,       !- Key Value
  Surface Ext Solar Incident, !- Variable Name
  timestep;                   !- Reporting Frequency

! Declare an output variable. This variable is equal to the shading
! signal + 0.1.
! It will be read by the external interface to demonstrate how
! to receive variables.
  EnergyManagementSystem:OutputVariable,
    Erl Shading Control Status,  !- Name
    Shade_Signal_01,             !- EMS Variable Name
    Averaged,                    !- Type of Data in Variable
    ZoneTimeStep;                !- Update Frequency
```

To specify that data should be exchanged every 10 minutes of simulation time, we enter in the `idf` file the section

```
  Timestep,
    6;           !- Number of Timesteps per Hour
```

4.3.4.2 Create a configuration file

Note that we have not yet specified the order of the elements in the signal vector that is exchanged between EnergyPlus and Ptolemy II. This information is specified in the file `variables.cfg`. The file `variables.cfg` needs to be in the same directory as the EnergyPlus idf file. For the objects used in the section above, the file looks like

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE BCVTB-variables SYSTEM "variables.dtd">
<BCVTB-variables>
  <variable source="Ptolemy">
    <EnergyPlus variable="yShade"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="ENVIRONMENT" type="OUTDOOR DRY BULB"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="WEST ZONE" type="Zone Mean Air Temperature"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="Zn001:Wall001:Win001" type="Surface Ext Solar Incident"/>
  </variable>
  <variable source="EnergyPlus">
    <EnergyPlus name="EMS" type="Erl Shading Control Status"/>
  </variable>
</BCVTB-variables>
```

This file specifies that the simulator actor that calls EnergyPlus has an input vector with one element that will be written to the actuator, and that it has an output vector with four elements that are computed by EnergyPlus and sent to Ptolemy II. The order of the elements in each vector is determined by the order in the above XML file. Note that the fourth element has the name `EMS` because it is an `EnergyManagementSystem:OutputVariable`. Hence, the output vector that contains the signals computed by EnergyPlus has elements

```
ENVIRONMENT (OUTDOOR DRY BULB)
WEST ZONE (Zone Mean Air Temperature)
Zn001:Wall001:Win001 (Surface Ext Solar Incident)
EMS (Erl Shading Control Status)
```

The configuration of the Ptolemy II model is identical to the configuration of Example 1, which is described in Section 4.3.2.3 *Create a Ptolemy II model* .

4.4 Dymola

To configure a Modelica model that will be simulated by Dymola, you may modify the files in the directory `BCVTB/examples/dymola-room`, or you may create a new Modelica model. This section describes how to create a new Modelica model using the Dymola modeling and simulation environment. The configuration consists of creating a Modelica model, a Modelica script and a Ptolemy II model.

4.4.1 Create a Modelica model

To create a new Modelica model, proceed as follows: First, open Dymola and the Buildings library, which may be downloaded from http://simulationresearch.lbl.gov/modelica . From the `Buildings` library, add the block Buildings.Utilities.IO.BCVTB.BCVTB to your model. Next, connect the `bcvtb` block to your other Modelica models to create a system model that takes signals from the `bcvtb` block and writes signals to the `bcvtb` block. This may yield a system model as shown in Figure 4.9 , which is the model in the file `bcvtb/examples/dymola-room/TwoRoomsTotal.mo`.

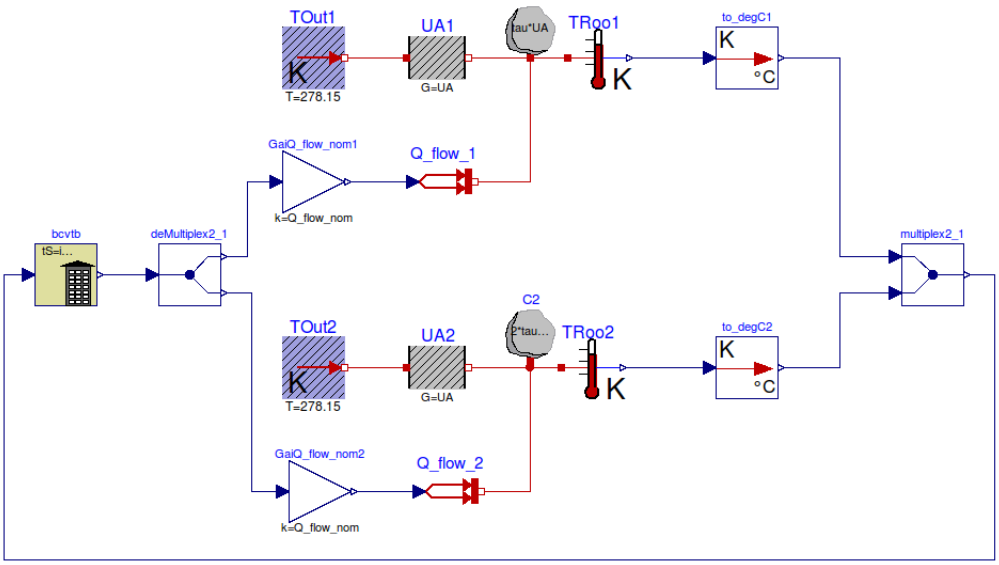

Figure 4.9: Graphical view of the Modelica model that computes the change in temperature for two simple room models.

To configure the `bcvtb` interface, double-click on the `bcvtb` block that is shown in the left of the figure. This will open the input form shown in Figure 4.10 .

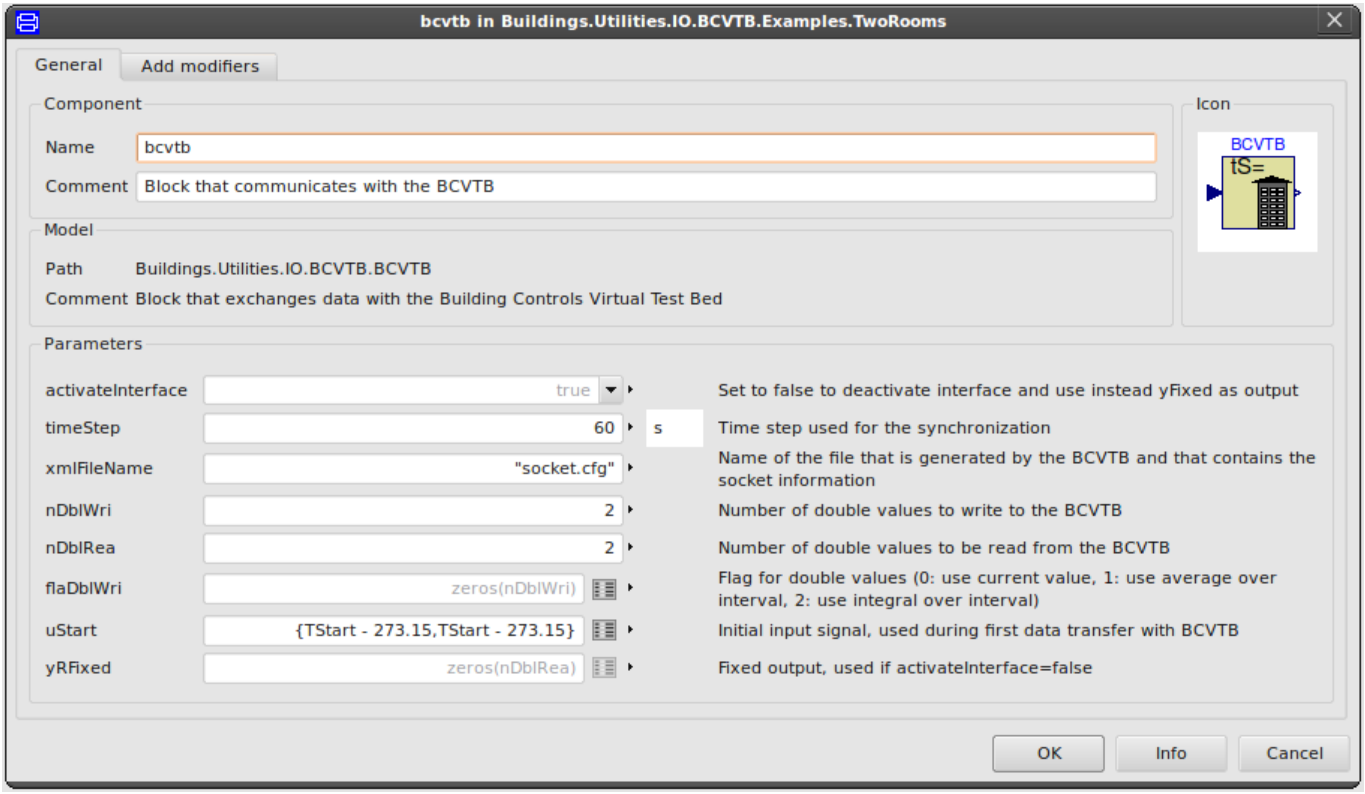

Figure 4.10: Configuration of the `bcvtb` block in the Modelica Buildings library

In this example, a vector with two double values are obtained from the BCVTB and written to the BCVTB every 60 seconds of simulation time. Additional information about this block can be obtained by pressing the Info button.

4.4.2 Create a Modelica script

To perform a simulation, the BCVTB will call a batch file (on Windows) or a shell script (on Linux), which in turn calls Dymola to execute a Modelica script that opens and simulates the model. The batch file or shell script is stored in the directory `bcvtb/bin` and need not be changed by the user. To create the Modelica script, adjust the following three lines as needed and save them in a file called `simulateAndExit.mos`:

```
openModel("TwoRoomsTotal.mo");
simulateModel("Buildings_Utilities_IO_BCVTB_Examples_TwoRooms", stopTime=21600);
exit();
```

4.4.3 Create a Ptolemy II model

To start Dymola from Ptolemy II, a Ptolemy II model will need to be created. The model `BCVTB/examples/dymola-room/system-windows.xml` shown in Figure 4.11 may be used as a starting point.

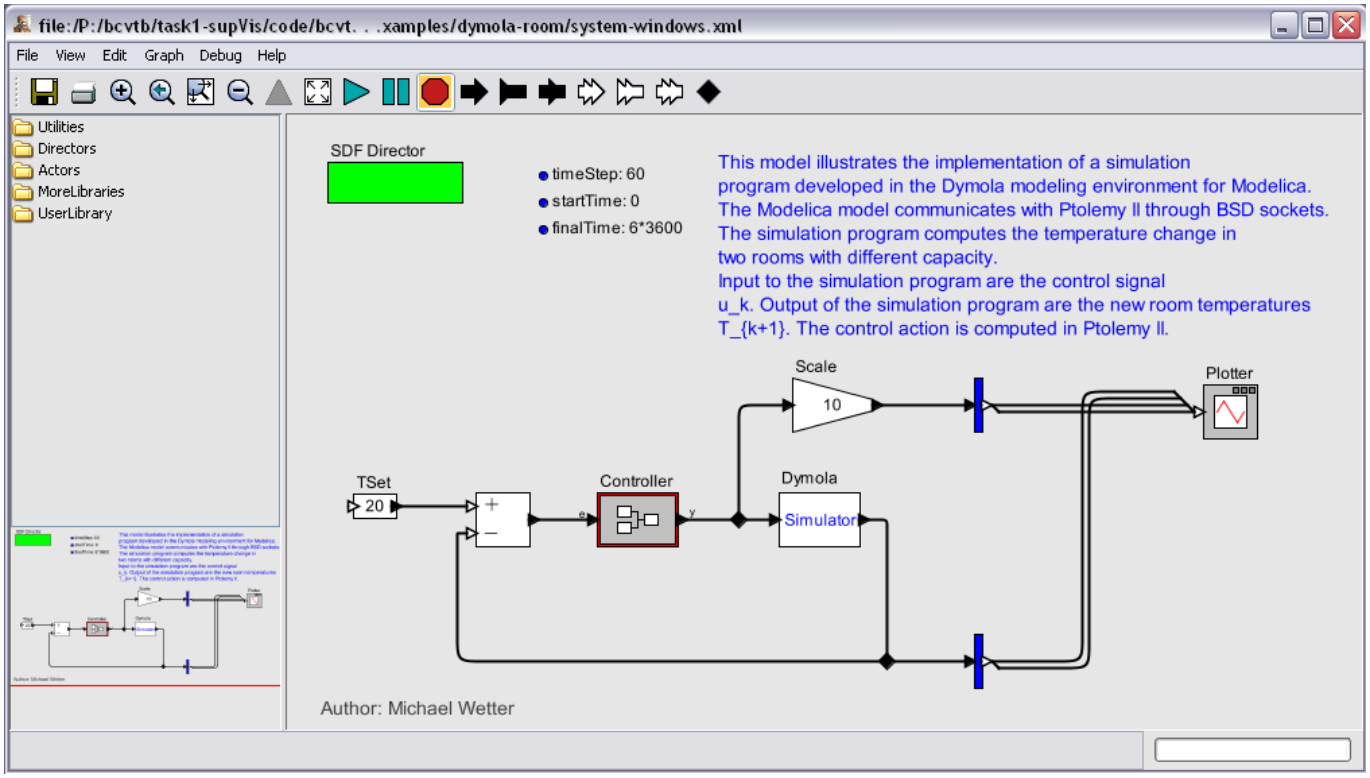

Figure 4.11: Ptolemy II system model that links a model of a controller with the `Simulator` actor that communicates with the Modelica modeling and simulation environment Dymola.

In this model, the `Simulator` actor that calls Dymola is configured as shown in Figure 4.12 .

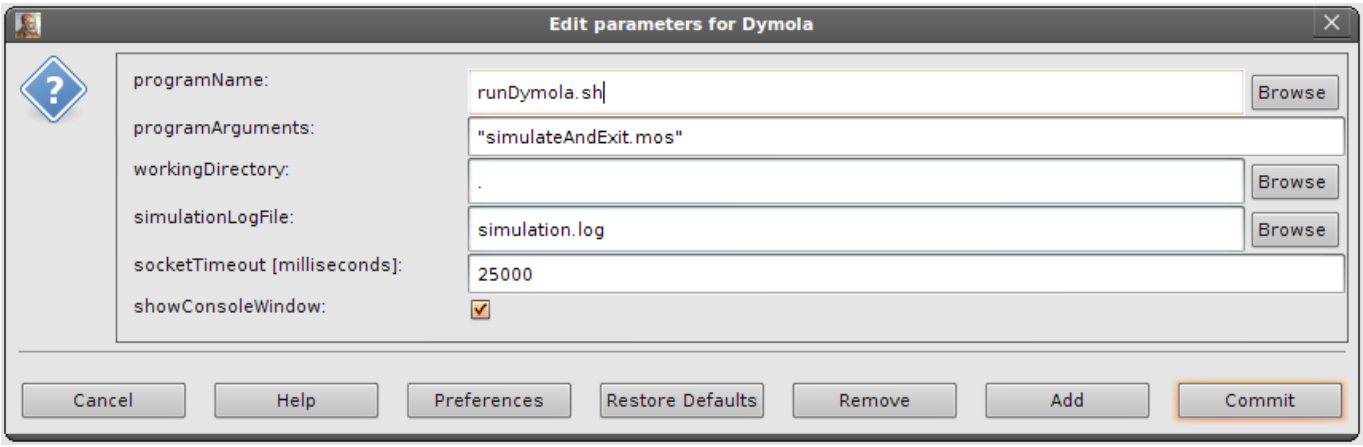

Figure 4.12: Configuration of the `Simulator` actor that calls Dymola on Linux.

The batch file `runDymola.bat` (on Windows) or the shell script `runDymola.sh` (on Linux) will copy the binary and header files that are required by Dymola. If either `dymosim.exe` (or `dymosim` on Linux) or `dsin.txt` do not exist in the current directory, then the batch file starts Dymola, translates and simulates the model. Otherwise, the batch file will call `dymosim -s` to simulate the model.

4.5 MATLAB

To configure MATLAB, you may modify an example such as the one in the directory `BCVTB/examples/matlab-room`, or you may create new input files. This section describes the latter approach, which consists of creating a MATLAB script and a Ptolemy II model.

4.5.1 Create a MATLAB script

A MATLAB script that exchanges data with the BCVTB has the following structure:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize variables
% ... (not shown)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Add path to BCVTB matlab libraries
addpath( strcat(getenv('BCVTB_HOME'), '/lib/matlab'));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Establish the socket connection
sockfd = establishClientSocket('socket.cfg');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exchange data (call this at each time step)
% ... (loop over each time step)
[retVal, flaRea, simTimRea, dblValRea ] = ...
  exchangeDoublesWithSocket(sockfd, flaWri, length(u), simTimWri, ...
                            dblValWri);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Close socket at the end of the simulation
closeIPC(sockfd);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Exit MATLAB
exit
```

For a complete MATLAB script that also includes error handling, see the file `simulateAndExit.m` in the directory `bcvtb/examples/matlab-room`.

4.5.2 Create a Ptolemy II model

To start MATLAB from Ptolemy II, you will need to create a Ptolemy II model. The model `BCVTB/examples/-matlab-room/system.xml` shown in Figure 4.13 may be used as a starting point. In this example, MATLAB computes the temperature change in two rooms for a given control input. The controller is implemented in Ptolemy II.

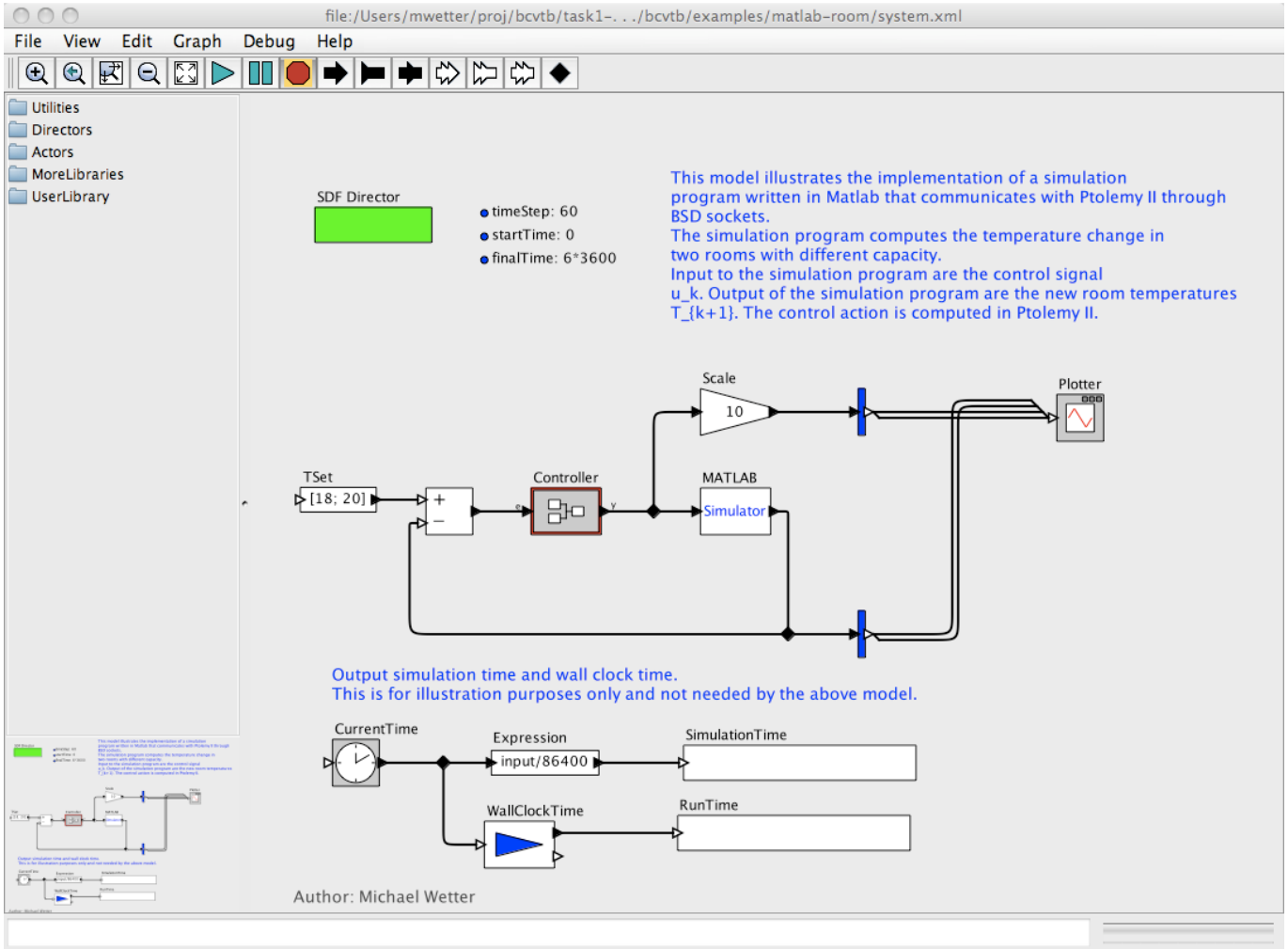

Figure 4.13: Ptolemy II system model that links an actor that computes a control signal with the `Simulator` actor that communicates with MATLAB.

In this model, the `Simulator` actor that calls MATLAB is configured as follows:

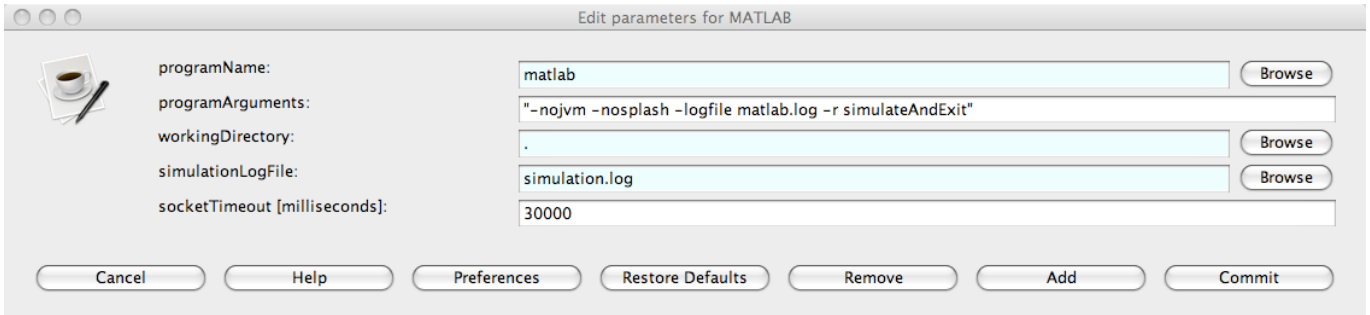

Figure 4.14: Configuration of the `Simulator` actor that calls MATLAB on Mac OS X.

This completes the configuration.

4.6 Simulink

To configure Simulink, you may modify an example such as the one in the directory `BCVTB/examples/simulink-room`, or you may create a new Simulink model. This section describes the latter, which consists of creating a Simulink block diagram, a MATLAB script and a Ptolemy II model.

4.6.1 Create a Simulink Block Diagram

To create a new Simulink block diagram, proceed as follows:

First, set the path to the Simulink library for the BCVTB: On the MATLAB prompt, type

```
addpath([getenv('BCVTB_HOME'), '/lib/matlab']);
```

Open Simulink and select `File -> New`. Then, in the Simulink Library Browser, select the BCVTB library.

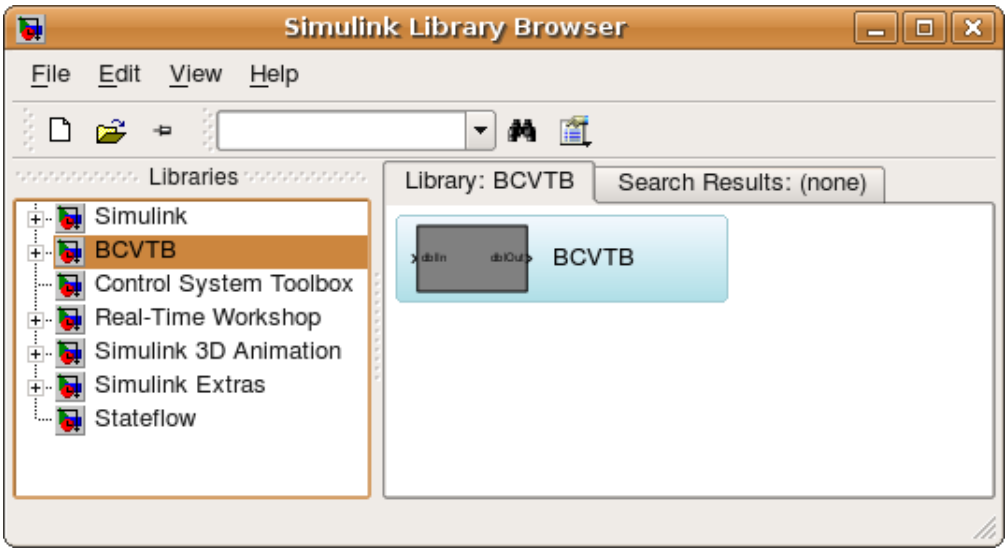

Figure 4.15: Simulink library with the block that connects to the BCVTB.

Drag and drop the BCVTB block into your Simulink flow chart. In the Simulink flow chart, open the BCVTB block which should show this model:

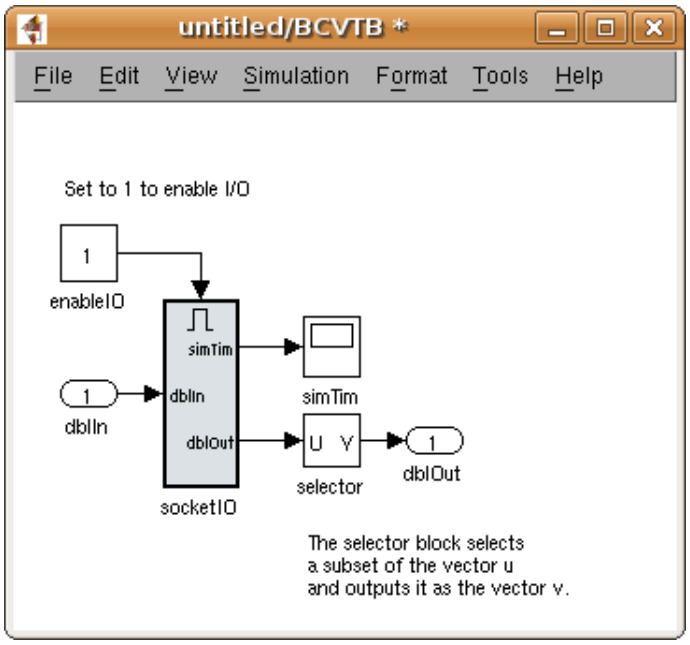

Figure 4.16: Model that is encapsulated in the `BCVTB` Simulink block.

In this model, the block `socketIO` implements the communication with the BCVTB. It typically need not be modified. However, you will need to open the block `selector` to adjust the field called `Index` in the input form shown in Figure 4.17 . This field specifies which elements of the input vector should be selected and used as an output of this block. For example, if we were to require three values, then the field `Index` needs to be `[1 2 3]` to select the first, second and third element of the input vector. Entering each element of the vector is inconvenient if a large number of elements needs to be received. In this situation, one can enter, for example, `linspace(1,50,50)` to retrieve a vector with 50 elements.

In Figure 4.17 , the field `Input port size` denotes the size of the input vector. It typically need not be changed unless you changed the file `bcvtb/lib/defines.h`.

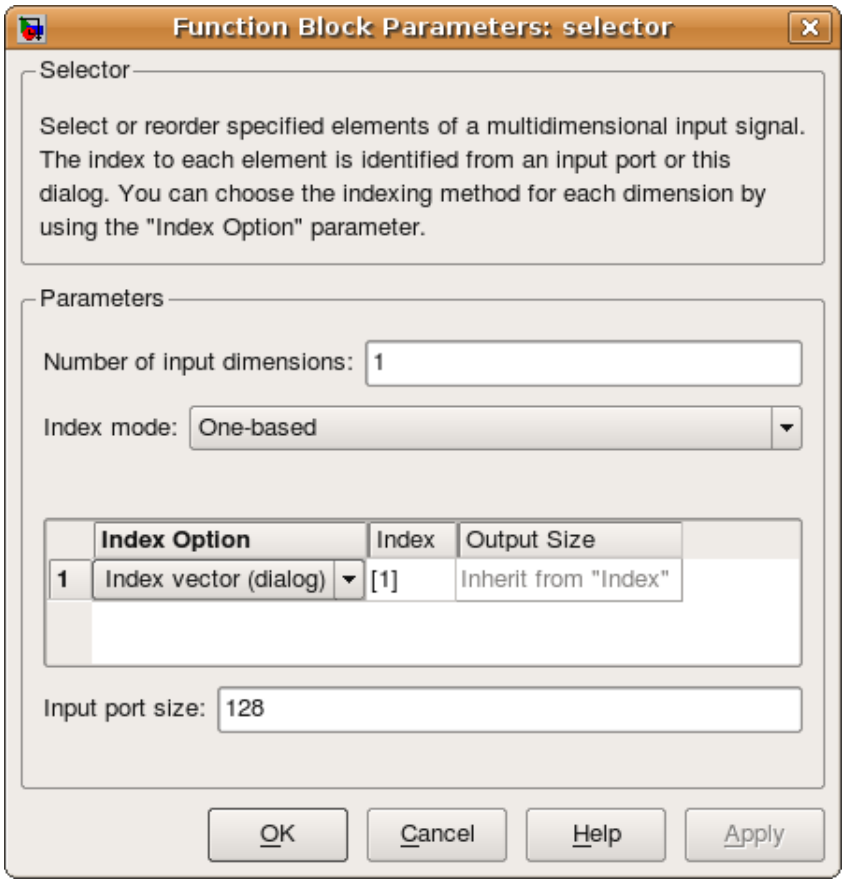

Figure 4.17: Configuration of the `selector` block that is shown in Figure 4.16.

Next, the sampling time step needs to be set. In this example, we assume that the Simulink simulation needs to be run with a fixed time step of 120 seconds. To implement this configuration, select `Simulation -> Configuration Parameters...` and configure the input form as shown in Figure 4.18 . Note that we set the stop time to `inf` since Simulink will receive from the BCVTB interface a signal when the final time is reached. We also set the step size to `120`, which is equal to the time step in seconds that will, in this example, be used in the Ptolemy II model.

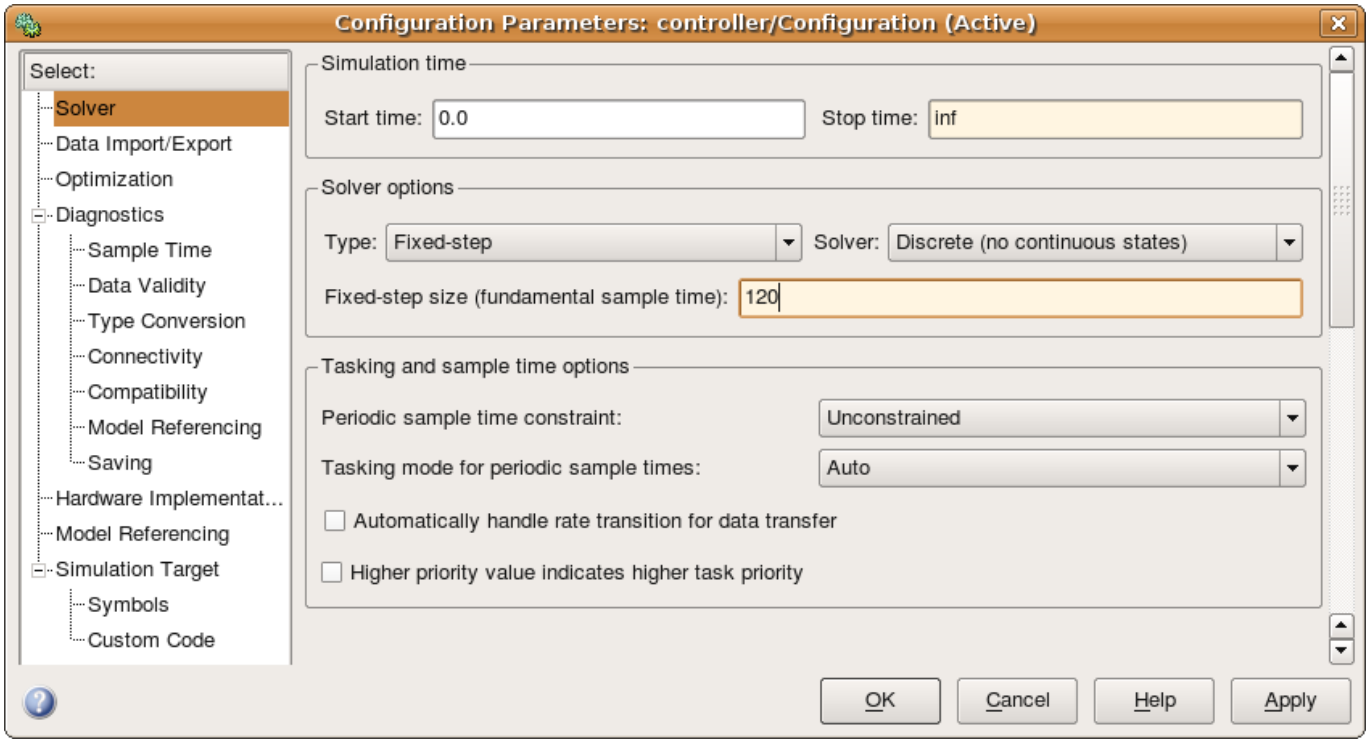

Figure 4.18: Configuration of the Simulink solver.

The BCVTB block can now be connected to a model that processes the output from the `BCVTB` block and produces new input for the `BCVTB` block. Such an implementation can be found in the model `BCVTB/examples/simulink-room/controller.mdl`, which is shown in Figure 4.19 .

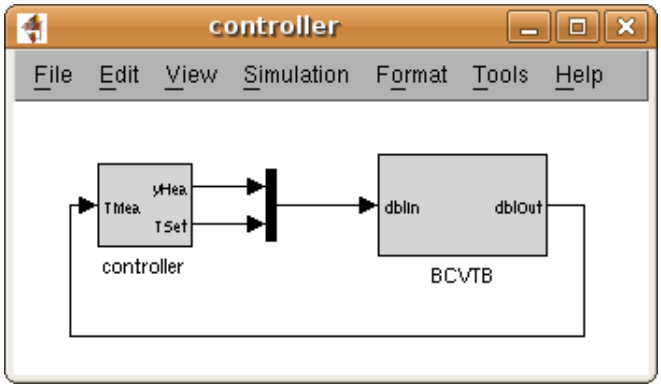

Figure 4.19: Simulink block diagram that links the controller with the block that communicates with Ptolemy II.

4.6.2 Create a MATLAB script

To perform a simulation, the BCVTB will call a MATLAB script that adds the path of the BCVTB library to the MATLAB path and then simulates the above model. To create the MATLAB script, save the following three lines in a file called `simulateAndExit.m`:

```
addpath([getenv('BCVTB_HOME'), '/lib/matlab']);
```

```
sim('controller');
quit;
```

4.6.3 Create a Ptolemy II model

To start Simulink from Ptolemy II, a Ptolemy II model needs to be created. The model `BCVTB/examples/simulink-room/system.xml` that is shown in Figure 4.20 may be used as a starting point.

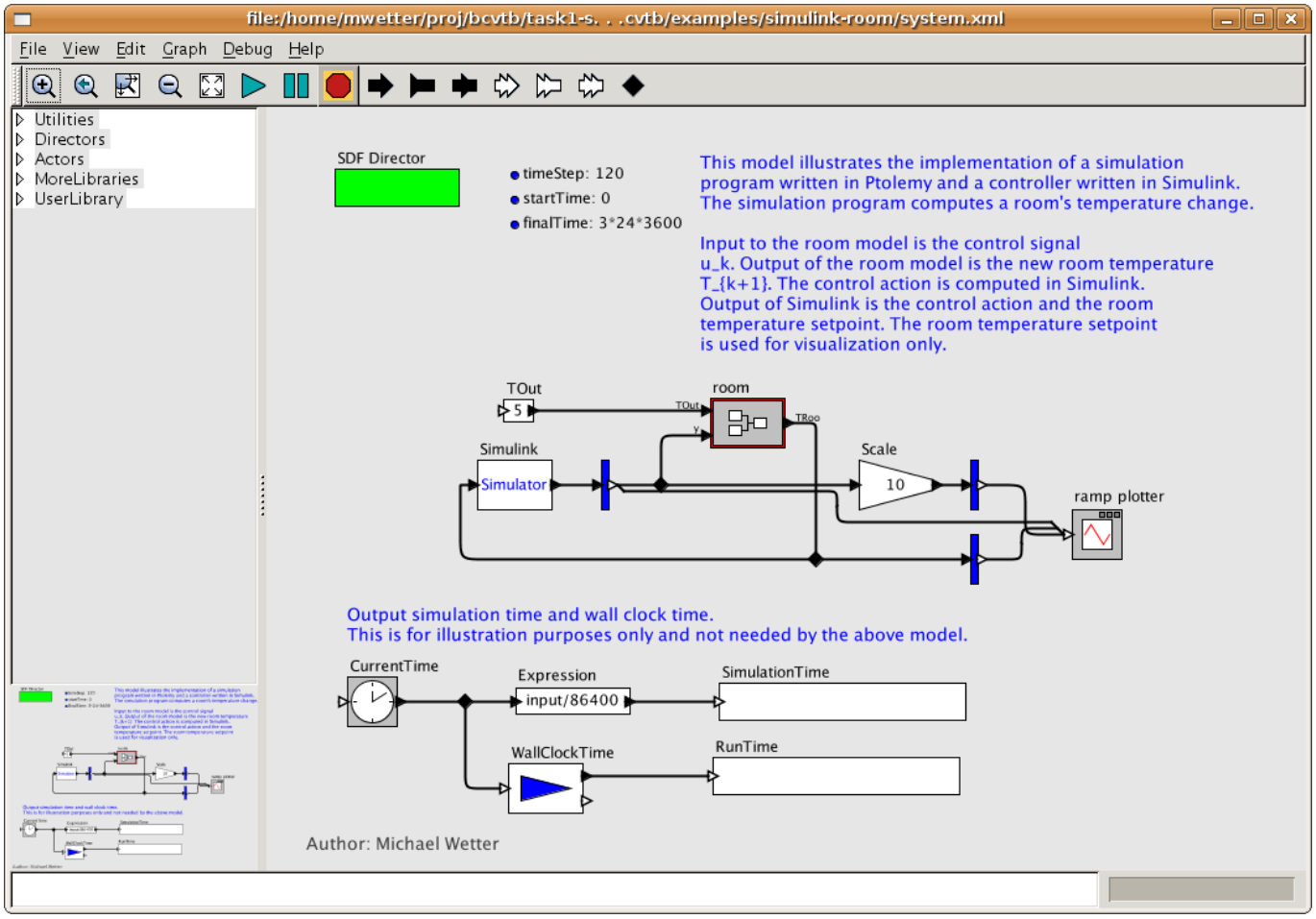

Figure 4.20: Ptolemy II system model that links the `Simulator` actor that communicates with MATLAB with an actor that computes the room temperature and with an actor that plots the results as the simulation progresses.

In this model, the `Simulator` actor that calls Simulink is configured as shown in Figure 4.21 .

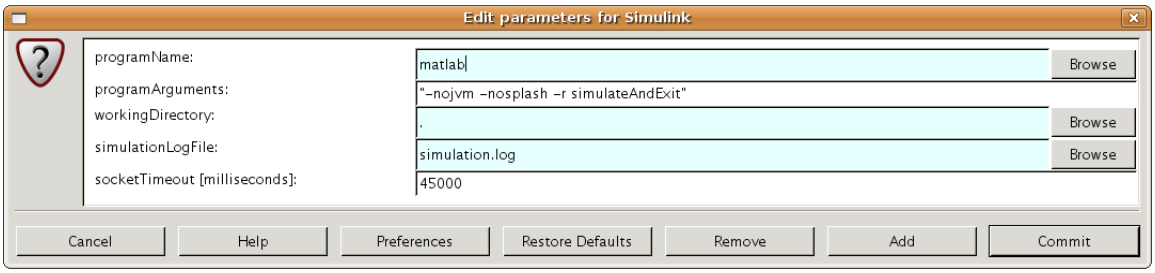

Figure 4.21: Configuration of the `Simulator` actor that calls MATLAB on Linux.

This completes the configuration.

4.7 Custom program using a system command

This page explains how to call a custom program from the BCVTB at each time step by using the `SystemCommand` actor. This 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 programs can be done either through program flags, or by writing an input file from Ptolemy II, using actors from the library `Actors->IO`.

To explain how to use this actor, we will show how to call a program that implements a proportional controller with output limitation for a room with closed loop control. The program is implemented in the C language. (Note that such a controller could be directly implemented in Ptolemy II. However, for illustration, we implemented this controller in a C program.) The program writes the control signal to a text file, which will then be parsed by Ptolemy II. We assume that the program needs to be called with two arguments, i.e., the numerical values of the control error `e` and the proportional gain `kP`, as

```
pcontroller e kP
```

where the numerical values `e` and `kP` may change at each call. We assume that the program writes the output file `output.txt` that needs to be read by Ptolemy II to receive the control signal. We also assume that the program returns the exit value `0` if no error occurred, or non-zero otherwise.

The next sections explain how to build such a system.

4.7.1 Create a Ptolemy II model

First, build a Ptolemy II block diagram that includes the `SystemCommand` actor from the library `Actors->Simulator`. Such a system model is implemented in the file `bcvtb/examples/systemCommand/system.xml` that is shown in Figure 4.22 .

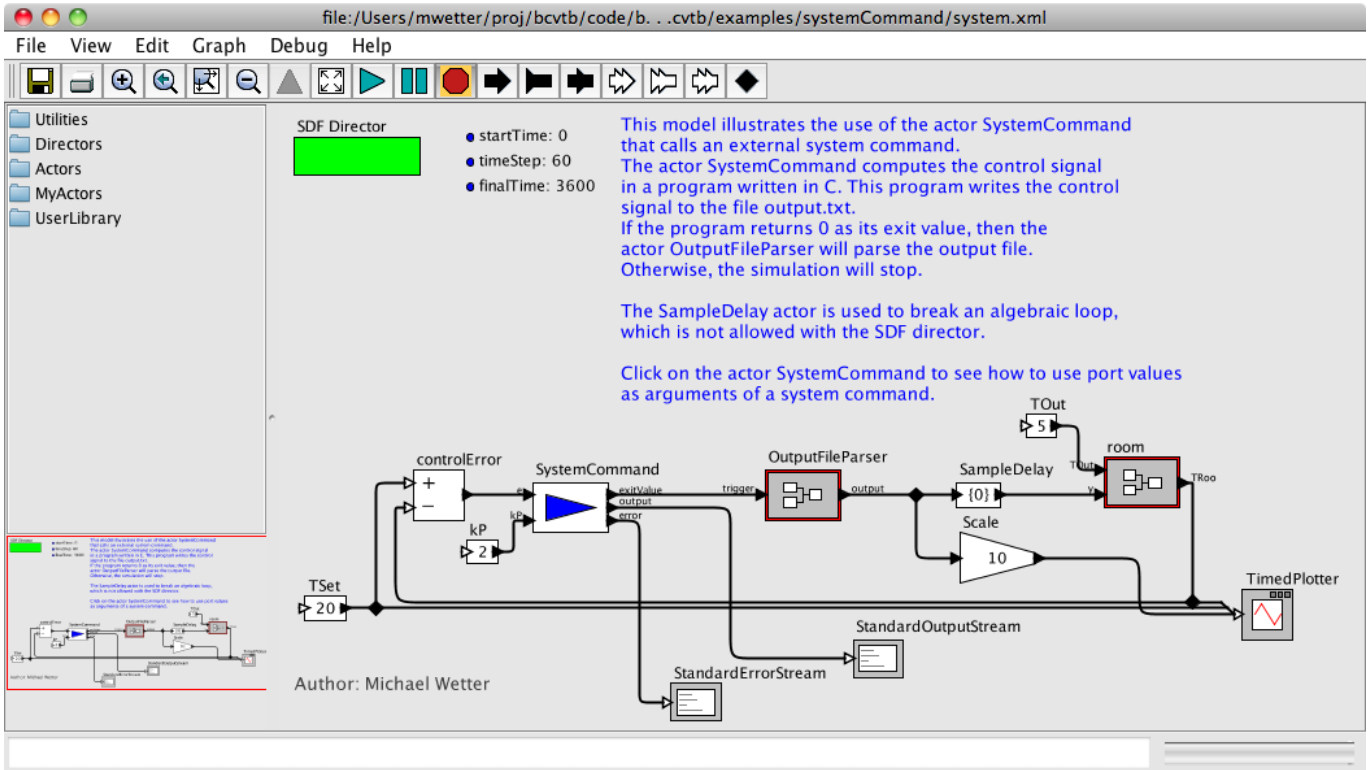

Figure 4.22: Ptolemy II system model that links the `SystemCommand` actor, which calls a C program to compute the new room temperature, with actors that parse output files and compute the room temperature.

4.7.2 Configure the ports of the `SystemCommand` actor

The `SystemCommand` actor has three predefined 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.

Next, we will configure the input ports of the `SystemCommand` actor by right-clicking on the actor, and selecting `Customize->Ports`. This will show the following window:

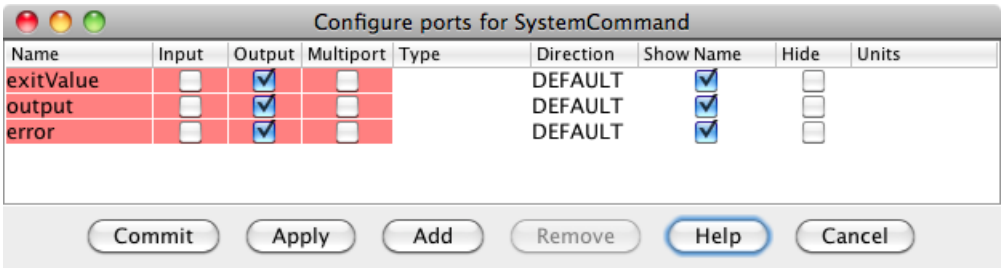

Figure 4.23: Input form that is used to add new ports to the actor.

Next, click the Add button and enter the input ports `e` and `kP`. The port names can be selected arbitrarily by the user, and there can be as many input ports as needed. After adding the ports, the window should look as follows:

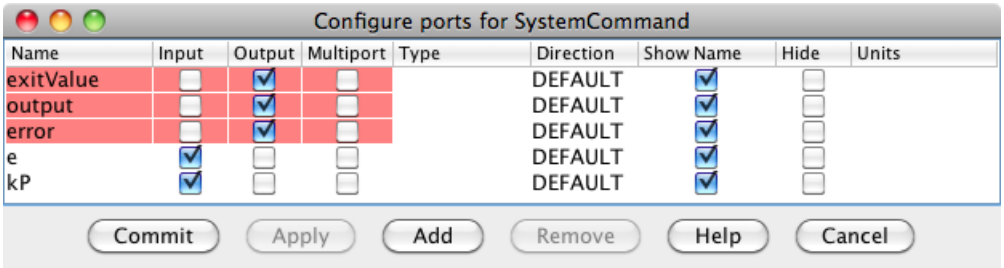

Figure 4.24: Input form after new ports have been added to the actor.

4.7.3 Configure the parameters of the `SystemCommand` actor

Finally, configure the parameters of the `SystemCommand` actor by double-clicking on its icon. This will show an input form where various parameters of the actor can be configured. The parameters of the `SystemCommand` actor are as desribed in Table 4.3 .

Parameter	Description
programName	The name of the executable that starts the simulation.
programArguments	Arguments needed by the simulation. Text arguments need to be enclosed in apostrophes.
workingDirectory	Working directory of the program. For the current directory, enter a period.
simulationLogFile	Name of the file to which the BCVTB will write the console output and error stream that it receives from the simulation program. Use a separate file for each simulation program. This file typically shows what may have caused an error.
showConsoleWindow	Check box; if activated, a separate window will be opened that displays the console output of the program.

Table 4.3: Parameters of the `SystemCommand` actor.

To pass the current value of port variables to the program as its argument, configure the actor as follows:

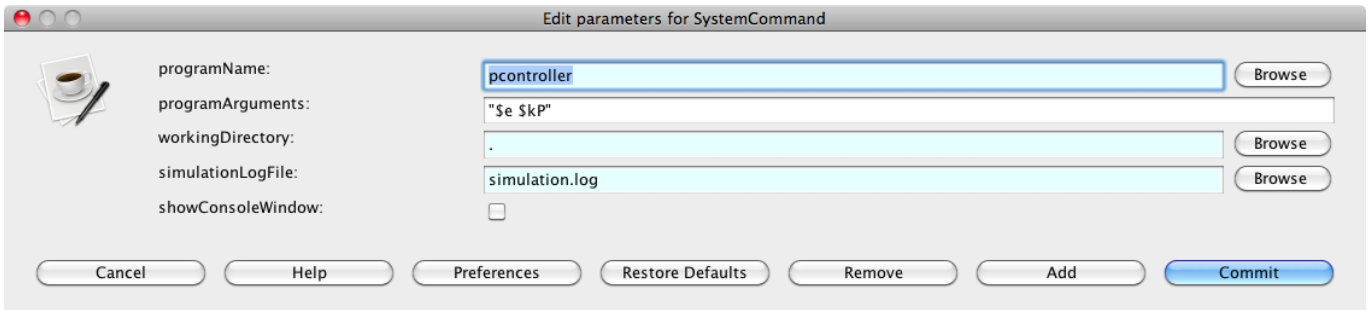

Figure 4.25: Configuration of the parameters of the `SystemCommand` actor.

This configuration will cause Ptolemy II to call the command `pcontroller $e $kP` at each time step, with the values `$e` and `$kP` being replaced by the current value of the input token. There are also two built-in variables called `$time`, which is the current simulation time, and `$iteraton` which is the current iteration step of Ptolemy II. No port needs to be defined for these two variables.

For example, for the configuration above, if at some time step the input ports are `e=1` and `kP=2`, then the BCVTB will fire the command

```
pcontroller 1 2
```

and wait until the program `pcontroller` terminates. Upon successful termination, the port `exitValue` will have the token `0`.

For an explanation of all parameters of the `SystemCommand` actor, right-click on the actor and select `Documentation->Get Documentation`.

4.8 Radiance

Note
Using Radiance in BCVTB requires Radiance simulation experience. A BCVTB user that is new to Radiance should first learn to use Radiance before attempting to use Radiance in BCVTB.

Note
The example has only been tested on Mac OS X and Linux, but not on Windows.

4.8.1 Introduction

Radiance is a collection of command line programs that are executed in various orders to perform simulations. Radiance commands with arguments are often stored in a script file for repetitive execution. The system command actor provides a means to perform Radiance simulations by executing this script and collecting the output. The BCVTB distribution includes two BCVTB Radiance examples.

4.8.2 Configuring Radiance

Radiance should be downloaded and installed on the computer as normal (the installation process varies based on the operating system). Do not forget to set the environment variables `PATH` and `RAYPATH` as described in the `README` file that is provided by the Radiance installation program. See also Section 2.3 *Setting system environment variables* for how to set environment variables for the BCVTB.

The following example uses Radiance to calculate average illuminance at a point in a model. This example generates a sky file based on weather file input, compiles an octree model and calculates illuminance at the point.

4.8.3 Create a Radiance script

We first create a Radiance script that computes the illuminance. The script takes as input arguments the month, day, and hour, the direct normal and diffuse horizontal irradiation, as well as the latitude, longitude and meridien. The output of the csh script is the illuminance, which will be written to the console. The script is as follows:

```
#!/bin/csh
##############################################################
# Script to run radiance.
##############################################################
set month = $argv[1]
```

```
set day = $argv[2]
set hour = `ev $argv[3]-.5`
set dirnorm = $argv[4]
set difhoriz = $argv[5]
set lat = $argv[6]
set long = $argv[7]
set mer = $argv[8]

set alt = `gensky $month $day $hour -a $lat -o $long -m $mer | awk '{if(NR==3)if( ↩
    $6>0) print 1; else print 0}'`

if ($alt == 1) then
### Generate perez sky
gendaylit $month $day $hour -a $lat -o $long -m $mer -W $dirnorm $difhoriz -g .1 > ↩
     rads/sky.rad

cat >> rads/sky.rad <<EOF

skyfunc glow skyglow
0
0
4 1 1 1 0

skyglow source sky
0
0
4 0 0 1 180

skyglow source ground
0
0
4 0 0 -1 180

EOF

### Compile octree model
oconv rads/sky.rad rads/approx.mat rads/room_basic.rad rads/top_panels.rad rads/ ↩
    desks.rad \
  rads/PC.rad rads/window_pane.rad rads/glass.rad > octs/model_sky.oct

### Create file of test points
echo 22 60 32 0 0 1 > data/test.pts

### Perform rtrace simulation
rtrace -h- -w- -n 2 -I -ab 2 -ad 2000 -as 1000 < data/test.pts octs/model_sky.oct  ↩
    | \
  rcalc -e '$1=179*($1*0.265+$2*0.670+$3*0.065)'

else
  echo 0.0
endif
```

4.8.4 Create a Ptolemy II model

Next, we create a Ptolemy II model that prepares the input data for the Radiance script, parses the output of the Radiance script, and displays the illuminance in a plotter.

Figure 4.26 shows the Ptolemy II system model that performs the following five steps for each time step iteration:

1. Read a line from the a weather data file that is in the EnergyPlus `epw` format.
2. Parse the weather record for the necessary data (month, day, hour, direct normal irradiance, and diffuse horizontal irradiance).
3. Run the Radiance script to simulate daylight using information from the weather data. This step will generate a Perez sky, compile an octree model, and simulate illuminance at a point.
4. Convert the script output from string to double precision format.
5. Plot illuminance vs. time.

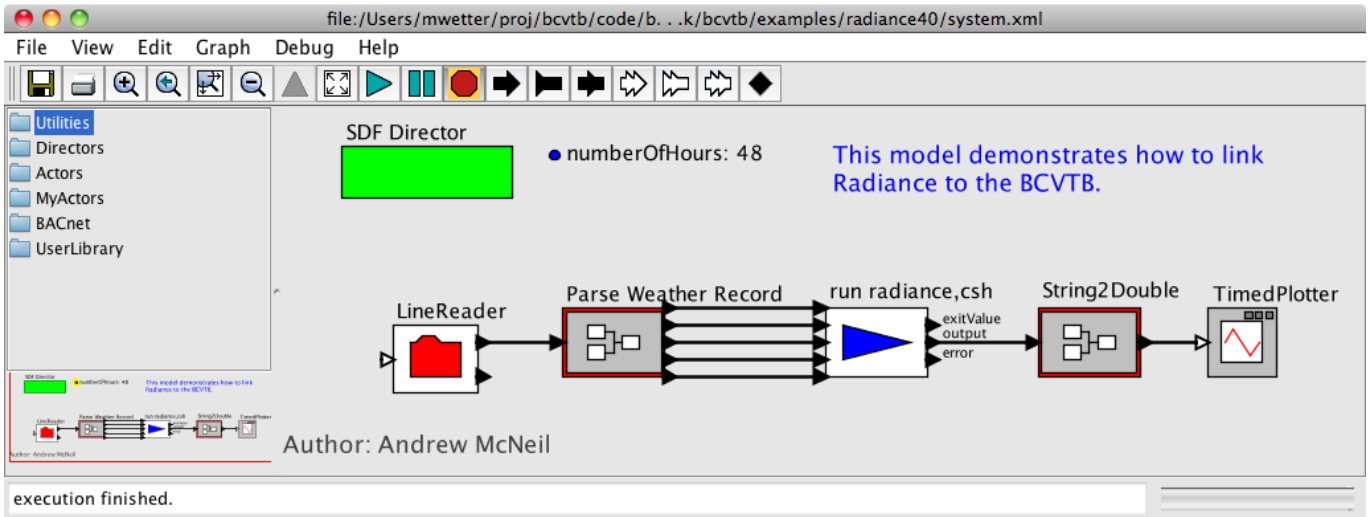

Figure 4.26: Ptolemy II system model that uses the `SystemCommand` actor to run Radiance.

The line reader actor reads the `epw` weather data file. The header (first 8 lines) is skipped by entering `8` into the `numberOfLinesToSkip` field as shown in Figure 4.27.

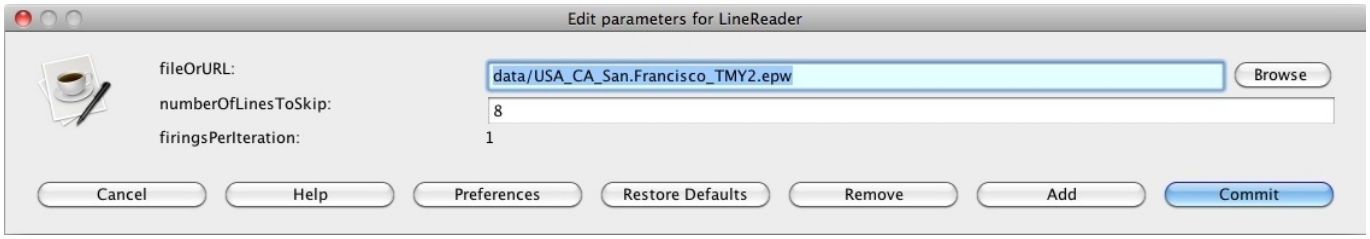

Figure 4.27: Parameters for the `LineReader` actor.

The parsing of the file is done by a composite actor. Looking inside the composite actor reveals an expression actor that splits the string at the commas into an array and five array element actors that select an element from the array as shown in Figure 4.28.

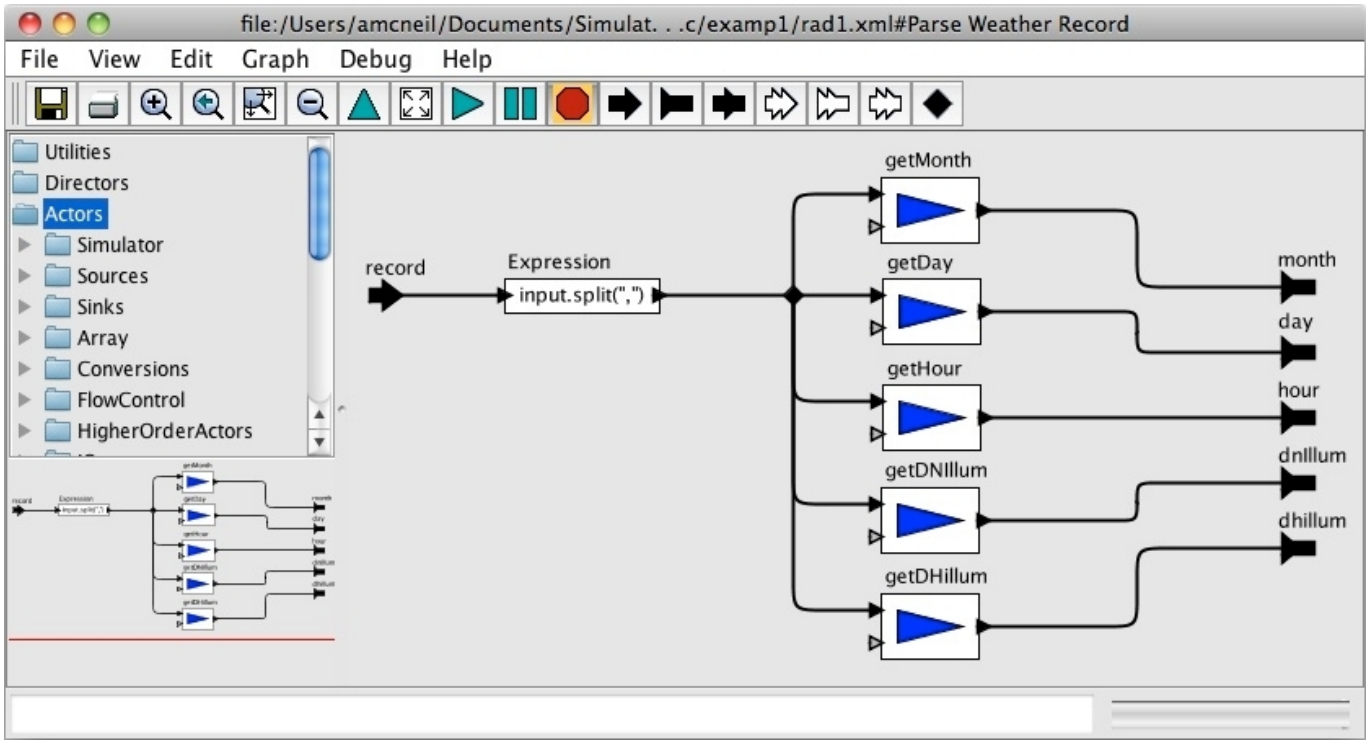

Figure 4.28: View inside the `ParseWeatherRecord` composite actor.

The system command actor runs a C shell script containing Radiance commands. The values read from the weather data file are passed as arguments to this C shell script as shown in Figure 4.29.

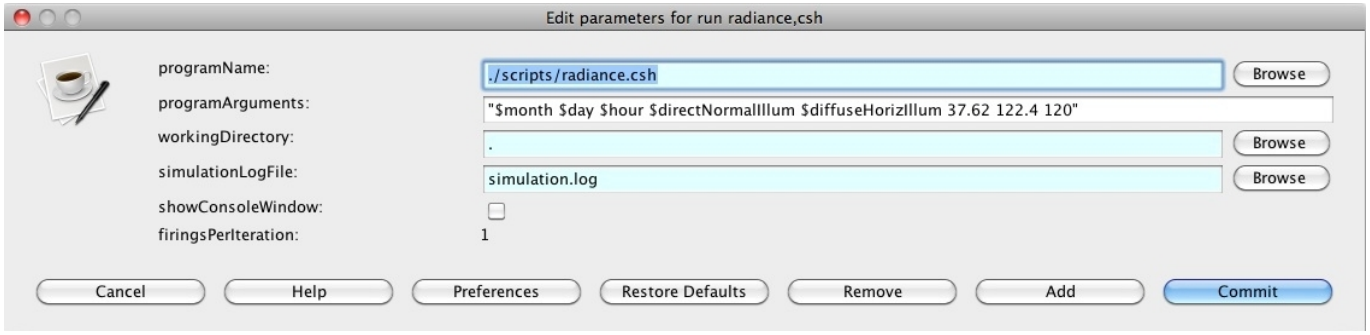

Figure 4.29: Parameters for the `SystemCommand` actor.

The output of the csh script that is called by the `SystemCommand` actor is the illuminance, but of type string and not double precision. The composite actor `String2Double` contains actors that convert the string to double as shown in Figure 4.30. The `trim` actor is required to strip the newline character from the end of the string.

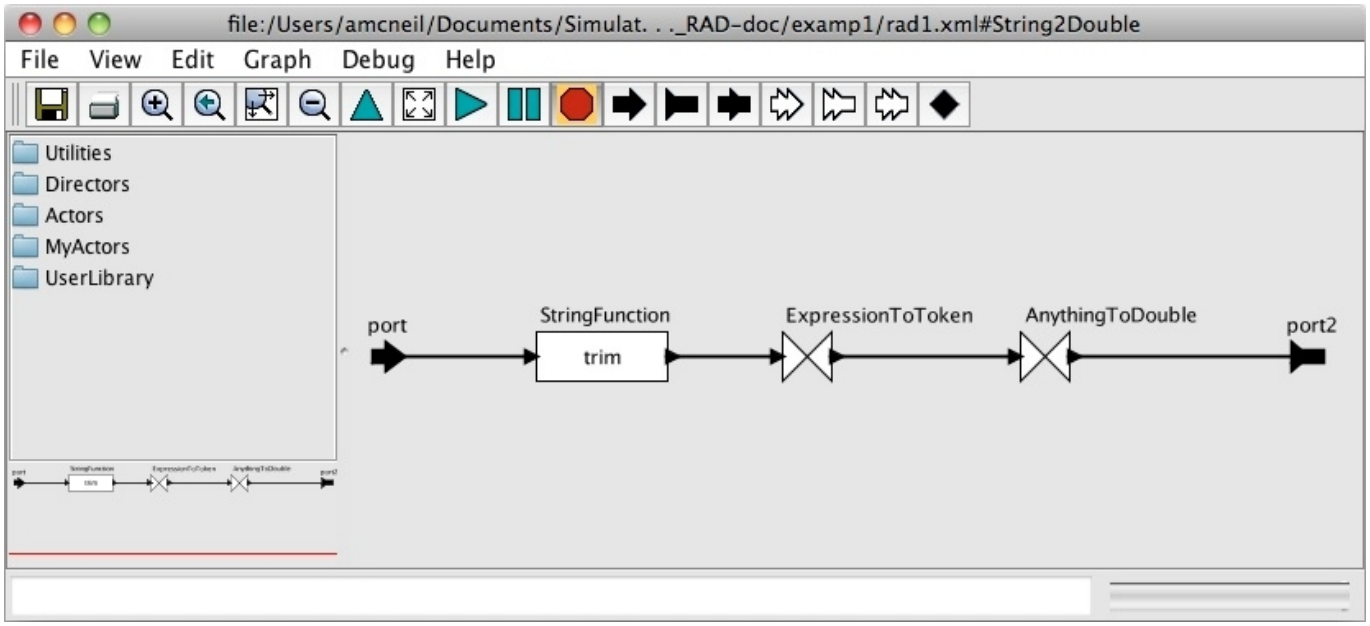

Figure 4.30: Inside the `String2Double` composite actor.

The `TimedPlotter` actor plots the illuminance vs. time, measured in hours from the beginning of the weather data file, as shown in Figure 4.31.

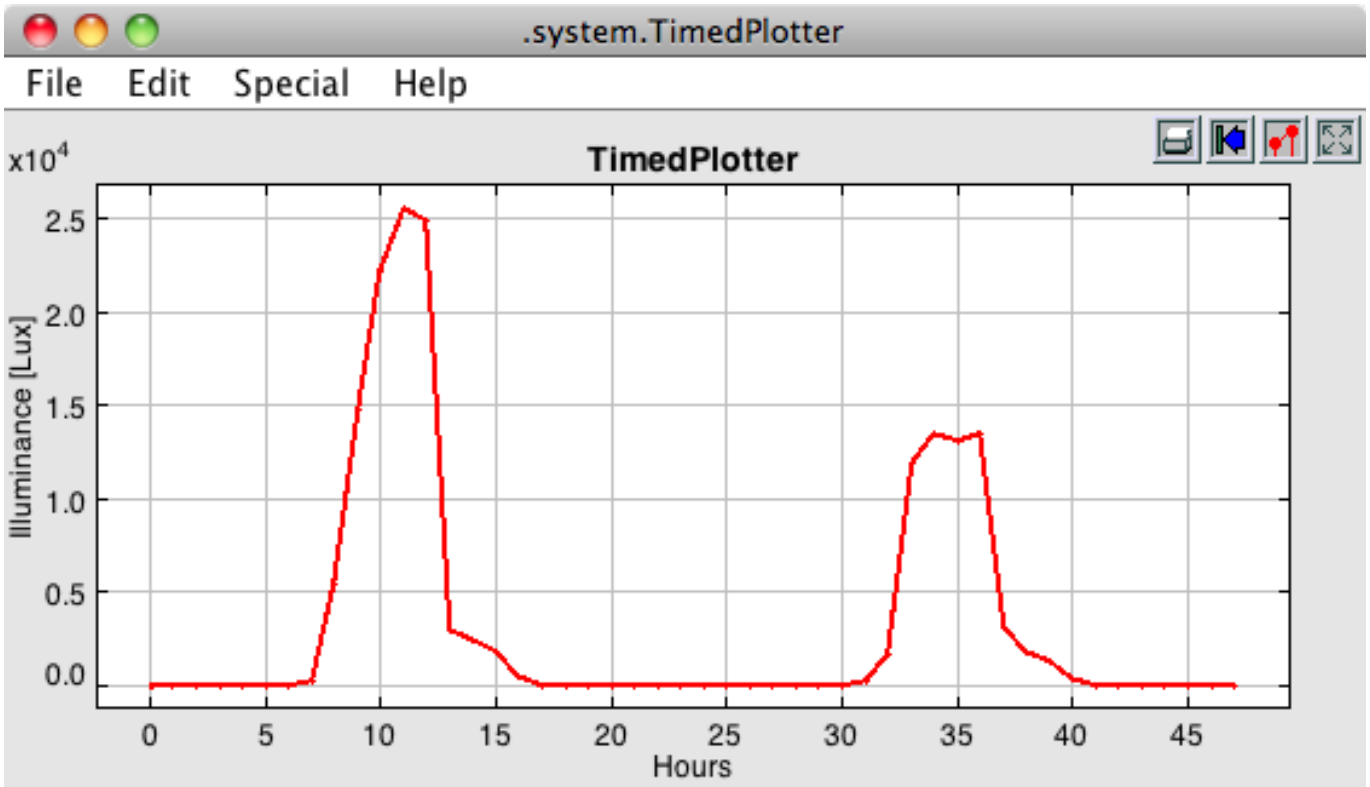

Figure 4.31: Output of the `TimedPlotter` actor.

4.9 BACnet

Note
The BACnet interface is only supported on Windows and on Linux. It has not been ported to Mac OS X.

4.9.1 Introduction

The BCVTB contains an actor, called `BACnetReader`, that can read from BACnet devices and an actor, called `BACnetWriter`, that can write to BACnet devices. These actors use the open source BACnet protocol stack , which is shipped with the BCVTB installation and that has been developed by Steve Karg. Both actors use a configuration file that specifies the BACnet devices, the object types and the property identifiers with which data is to be exchanged. The next sections describe how to configure these configuration files, and how to set up a model that reads from and writes to BACnet devices.

Note
BACnet systems typically allow a user to export a list of BACnet *object types* with their instance numbers. Such a list needs to be obtained for the particular control system in order to configure the data exchange that is described in Section 4.9.2 *Reading from BACnet* and Section 4.9.3 *Writing to BACnet* .

4.9.2 Reading from BACnet

4.9.2.1 Specification of data that will be read from BACnet

The `BACnetReader` actor reads an xml configuration file to determine what data it needs to read from BACnet devices. This configuration file specifies the BACnet *object types* and their child elements, [1] which can be other BACnet *object types* or BACnet *property identifiers* . The xml configuration file has the following syntax: It starts and ends with

```
<?xml version="1.0" encoding="utf-8"?>
<BACnet>
   <!-- Child elements are not shown. -->
</BACnet>
```

The above element `BACnet` requires at least one child element of the form

```
<Object Type="Device" Instance="123">
   <!-- Child elements are not shown. -->
</Object>
```

i.e., the element name is `Object`, the attribute `Type` needs to be `Device` and the attribute `Instance` needs to be set to its instance number, which is a unique number that is assigned at the discretion of the control provider. Any `Object` element can contain other `Object` elements and other `PropertyIdentifier` elements.

The `Object` elements can have any of the following values for the attribute `Type` (the meaning of these types is explained in Chapter 12 of the BACnet Standard [ASHRAE 2004]): `Analog Input`, `Analog Output`, `Analog Value`, `Binary Input`, `Binary Output`, `Binary Value`, `Calendar`, `Command`, `Device`, `Event Enrollment`, `File`, `Group`, `Loop`, `Multi State Input`, `Multi State Output`, `Notification`

[1] In xml, an element `B` is called a child element of an element `A` if `B` is contained exactly one level below element `A`.

`Class`, `Program`, `Schedule`, `Averaging`, `Multi State Value`, `Trend Log`, `Life Safety Point`, `Life Safety Zone`, `Accumulator`, `Pulse Converter`, `Event Log`, `Global Group`, `Trend Log Multiple`, `Load Control`, `Structured View`, `Access Door`, `Lighting Output`, `Access Credential`, `Access Point`, `Access Rights`, `Access User`, `Access Zone`, `Authentication Factor Input`, `Max ASHRAE`, `Load Control`, `Structured View`, `Access Door`, `Lighting Output`, `Access Credential`, `Access Point`, `Access Rights`, `Access User`, `Access Zone`, `Authentication Factor Input`, `Max ASHRAE`.

Each of these object types has its own set of properties that can be read or written to. These properties are declared in the element `PropertyIdentifier` which has one attribute called `Name`. For example, for the object with type `Analog Output`, the BACnet standard lists in Table 12-3 the properties shown in Table 4.4 .

Property Identifier	**Property Datatype**	**Conformance Code**
Object_Identifier	BACnetObjectIdentifier	R
Object_Name	CharacterString	R
Object_Type	BACnetObjectType	R
Present_Value	REAL	W
Description	CharacterString	O
(further entries are omitted)		

Table 4.4: Properties of the Analog Output Object Type according to BACnet Standard, Table 12-3 (not all properties are shown).

Thus, we can set, for example,

```
<?xml version="1.0" encoding="utf-8"?>
<BACnet>
  <Object Type="Device" Instance="123">
    <Object Type="Analog Output" Instance="1">
      <PropertyIdentifier Name="Present_Value"/>
    </Object>
  </Object>
</BACnet>
```

which would cause the `BACnetReader` to read the present value of the BACnet Analog Output object type with instance number 1, which is part of a BACnet Device Object with instance number 123.

The following code listing shows an example of a larger configuration file that is used to read data from a BACnet system.

```
<?xml version="1.0" encoding="utf-8"?>
<BACnet>
  <Object Type="Device" Instance="637501">❶

    <PropertyIdentifier Name="Local_Date"/>❷

    <Object Type="Analog Input" Instance="1">❸
      <PropertyIdentifier Name="Object_Identifier"/>❹
      <PropertyIdentifier Name="Units"/>❺
      <PropertyIdentifier Name="Present_Value"/>❻
    </Object>

    <Object Type="Analog Output" Instance="2">❼
      <PropertyIdentifier Name="Present_Value"
                          Index="2"/>❽
```

```
    </Object>

  </Object>

  <Object Type="Device" Instance="637502">❾

    <Object Type="Analog Input" Instance="1">
      <PropertyIdentifier Name="Present_Value"/>
    </Object>

    <Object Name="Analog Output" Instance="3">
      <PropertyIdentifier Name="Present_Value"/>
    </Object>

  </Object>
</BACnet>
```

The numbered items have the following functionalities:

❶, ❾ The BACnet devices are declared at the top-level of the control system. The only valid elements are

```
<Object Type="Device" Instance="123">
   <!-- Child elements are not shown. -->
</Object>
```

which all need to have a unique, system-dependent instance number.

❷ This line declares a BACnet property identifier of the device with instance number 637501. This statement will cause the BACnet reader to read the local date from the device.

❸, ❼ These lines declare BACnet object types that are children of the device object type with instance number 637501. The first instance has the instance number 1, and the second instance has the instance number 2. Note that instance numbers are assigned by the controls provider and need not start at 1.

❹, ❺, ❻ These entries declare BACnet property identifiers of the device with instance number 1. These statements will cause the BACnet reader to read its object identifier, its units and its present value.

❽ The optional attribute `Index="2"` declares that the present value will only be obtained for the second element of this Analog Output object. If the `Index` would not be specified and the Analog Output object has an array of values, then all elements of the array would be read.

4.9.2.2 Interface to BACnet Stack

To read data from BACnet devices, the `BACnetReader` actor calls an executable program that is provided by the BACnet stack. This section describes how the entries in the xml file relate to this executable. The example shows the low-level implementation and may be skipped by users who are not interested in the implementation.

To read from BACnet, the BACnet stack provides the following function:

```
bacrp device-instance object-type object-instance property [index]
```

(For an explanation of the arguments, type `./bacrp --help` on a console.) The above xml file would cause the following commands to be executed:

```
bacrp 637501 8 637501 56
bacrp 637501 0 1 75
bacrp 637501 0 1 117
bacrp 637501 0 1 191
bacrp 637501 1 2 117 2
bacrp 637502 0 1 191
bacrp 637502 1 3 191
```

In the first command, the second argument is 8 as this is the enumeration for the BACnet Object Device, and the fourth argument is 56 as this is the enumeration for the Local Date Property. The following lines are constructed similarly, using the enumerations that are defined in the file `bacenum.h` that is part of the BACnet stack.

4.9.3 Writing to BACnet

4.9.3.1 Specification of data that will be written to BACnet

The BCVTB contains an actor called `BACnetWriter` that can write to BACnet devices. The BACnet standard [ASHRAE 2004] defines the conformance codes shown in Table 4.5 . The `BACnetReader` can write to any BACnet properties with the conformance code `W`.

O	Indicates that the property is optional.
R	Indicates that the property is required to be present and readable using BACnet services.
W	Indicates that the property is required to be present, readable, and writeable using BACnet services.

Table 4.5: BACnet Conformance Codes.

The `BACnetWriter` provides the *WriteProperty Service* that is specified in Section 15.9 in the BACnet Standard [ASHRAE 2004] . The configuration file that is used by the `BACnetWriter` is identical to the one used for the `BACnetReader` explained in Section 4.9.2 *Reading from BACnet* , except that the xml elements of type `PropertyIdentifier` have the additional attributes `ApplicationTag`, `Priority`, and `Index`. These attributes are explained in Table 4.6 . The following program listing shows an example configuration file.

```
<?xml version="1.0" encoding="utf-8"?>
<BACnet>
  <!-- Top level BACnet device -->
  <Object Type="Device" Instance="637501">

    <!-- BACnet object for analog input -->
    <Object Type="Analog Input" Instance="1">
      <PropertyIdentifier Name="Present_Value" ApplicationTag="Real"
        Priority="15" Index="-1"/>
    </Object>
  </Object>
  <!-- Top level BACnet device -->
  <Object Type="Device" Instance="637502">

    <Object Type="Analog Input" Instance="1">
      <PropertyIdentifier Name="Present_Value" ApplicationTag="Real"
        Priority="15" Index="-1"/>
    </Object>
    <!-- BACnet object for analog input -->
    <Object Type="Analog Input" Instance="2">
```

```
      <PropertyIdentifier Name="Present_Value" ApplicationTag="Real"
        Priority="15" Index="-1"/>
    </Object>
  </Object>
</BACnet>
```

For the `BACnetReader`, the xml element `PropertyIdentifier` has the attributes shown in Table 4.6 .

Attribute name	Required	Description
Name	yes	The name of the property identifier.
ApplicationTag	yes	This attribute specifies the data format that is used to send the value to the BACnet device. Possible entries are `NULL`, `BOOLEAN`, `UNSIGNED_INT`, `SIGNED_INT`, `REAL`, `DOUBLE`, `OCTET_STRING`, `CHARACTER_STRING`, `BIT_STRING`, `ENUMERATED`, `DATE`, `TIME`, `OBJECT_ID`, `MAX_BACNET_APPLICATION_TAG`. The value of this attribute will be converted to upper-case, and then sent to the BACnet interface.
Priority	no	This parameter sets the priority of the write operation. Allowed entries are any integers from 0 to 16. If Priority 0 is given, no priority is sent, which defaults according to the BACnet standard to the lowest priority. The highest priority is 1 and the lowest priority is 16. If the value is not specified, then it is set to 15.
Index	no	This integer parameter is the index number of an array. If the property is an array, individual elements can be written to if supported by the BACnet device. If this parameter is -1, the index is ignored and hence the entire array is referenced. If the value is not specified, then it is set to -1.

Table 4.6: Attributes of the `PropertyIdentifier` xml element if used to write to a BACnet device.

4.9.3.2 Interface to BACnet Stack

To write data to BACnet devices, the `BACnetWriter` actor calls an executable program that is provided by the BACnet stack. This section describes how the entries in the xml file relate to this executable. The example shows the low-level implementation and may be skipped by users who are not interested in the implementation.

To write to BACnet, the BACnet stack provides the following function:

```
bacwp device-instance object-type object-instance property priority index tag  ↩
    value [tag value...]
```

(For an explanation of the arguments, type `./bacwp --help` on a console.) The above xml file would cause the following commands to be executed:

```
bacwp 637501 0 1 85 15 -1 4 "value[1]"
bacwp 637502 0 1 85 15 -1 4 "value[2]"
bacwp 637502 0 2 85 15 -1 4 "value[3]"
```

Note that our implementation only supports one pair of `tag value`. However, multiple pairs can be constructed by declaring a separate `PropertyIdentifier` element for each pair.

In the first command, the second argument is zero as this is the enumeration for analog input objects in the BACnet stack; the fourth argument is 85 which is the enumeration for the present value property; the second last element is 4 as this is the enumeration for the application tag; and "value[1]" will be replaced with the actual value of the first element of the vector that is received at the input port of the actor.

In the second command, "value[2]" will be replaced with the actual value of the second element of the vector that is received at the input port of the actor. For a list of the enumerations that are used in the above commands, see the file `bacenum.h` that is part of the BACnet stack.

4.9.4 Creating a Ptolemy II model

The `BACnetReader` and the `BACnetWriter` actor can be used in the same Ptolemy II model. In this section, however, we will explain how to configure separate Ptolemy II models that write to and read from BACnet devices. These files can be found in the directories `bcvtb/examples/BACnetReaderALC` and `bcvtb/examples/-BACnetWriterALC`. Note that these examples have been developed for a particular hardware setup. To run these examples for other hardware, their configuration files need to be modified as described in Section 4.9.2 *Reading from BACnet* and Section 4.9.3 *Writing to BACnet* .

4.9.4.1 Configuring the `BACnetReader`

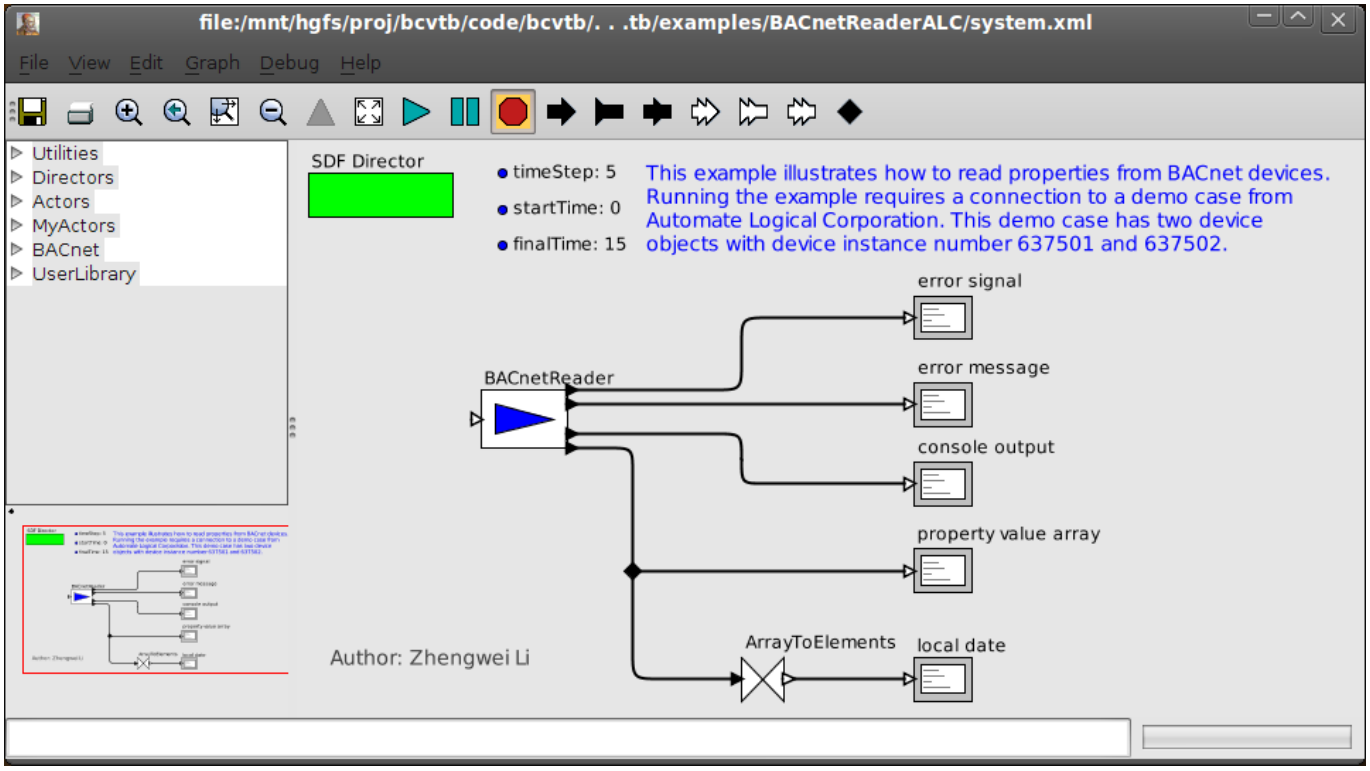

Figure 4.32: Ptolemy II system model that uses the `BACnetReader` actor.

Figure 4.32 shows a Ptolemy II system model that uses the `BACnetReader` actor. To configure the `BACnetR-eader`, double-click on its icon and add the name of its configuration file that has been developed as described in Section 4.9.2 *Reading from BACnet* . There is also a check-box called `continueWhenError`. If activated and an error occurs, then Ptolemy II will continue the simulation and the actor will output at its ports the last known value

and the error message, unless the error occurs in the first step, in which case the simulation stops. If deactivated and an error occurs, then the simulation will stop, the error message will be displayed on the screen and the user is required to confirm the error message by clicking on its OK button. *Thus, select this box if the BCVTB should continue its operation when a run-time error, such as a network timeout, occurs.*

The `BACnetReader` has the following ports: There is one input port, which is a trigger port. If the SDF Director is used in the Ptolemy II system model, then this port need not be connected. The `BACnetReader` has the following output ports:

Port	**Description**
errorSignal	If there were no errors in the previous data exchange, then this port outputs zero. Otherwise, the output is a non-zero integer.
errorMessage	If there was an error in the previous data exchange, then this port outputs the error message that was generated by the `BACnetReader` actor. (The error messages that were generated by the BACnet stack are output of the `consoleOutput` port.)
consoleOutput	This port outputs the standard output stream and the standard error stream of the executable that communicates with BACnet.
propertyValueArray	This port outputs the values obtained at the last successful communication with the BACnet devices. If there was an error in the last communication, then the values from the previous time step will be output of this port. The output data type is an array whose elements are string representations of the BACnet properties that are read according to the configuration file. Elements can be extracted from this array using actors from Ptolemy II's `Actors->Array` library.

4.9.4.2 Configuring the `BACnetWriter`

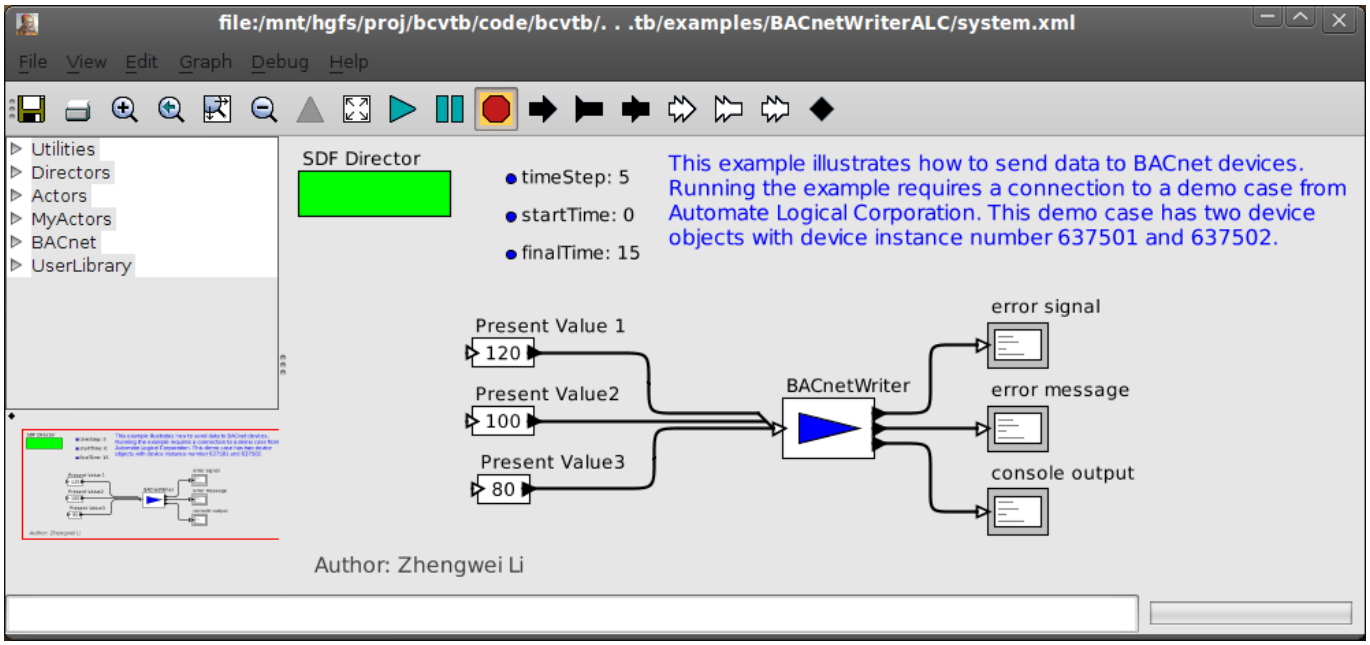

Figure 4.33: Ptolemy II system model that uses the `BACnetWriter` actor.

The configuration of the `BACnetWriter` actor is similar to the configuration of `BACnetReader`. Figure 4.33 shows a Ptolemy II system model that uses the `BACnetWriter`. To configure the `BACnetWriter`, double-click on its icon and add the name of its configuration file that has been developed as described in Section 4.9.3 *Writing to BACnet* . There is also a check-box called `continueWhenError`. If activated and an error occurs, then Ptolemy II will continue the simulation and the actor will output the error message at its ports, unless the error occurs in the first step, in which case the simulation stops. If deactivated and an error occurs, then the simulation will stop, the error message will be displayed on the screen and the user is required to confirm the error message by clicking on its OK button. *Thus, select this box if the BCVTB should continue its operation when a run-time error, such as a network timeout, occurs.*

Input into the actor is an array of values that will be written to the BACnet devices according to the order specified in the xml configuration file. In Ptolemy II, such an array can be composed from scalar inputs by using the actor `Actors->Array->ElementsToArray`.

The `BACnetWriter` has one input port. This port is used to collect the data that need to be sent to the BACnet devices. The `BACnetReader` has the following output ports:

Port	Description
errorSignal	If there were no errors in the previous data exchange, then this port outputs zero. Otherwise, the output is a non-zero integer.
errorMessage	If there was an error in the previous data exchange, then this port outputs the error message that was generated by the `BACnetReader` actor. (The error messages that were generated by the BACnet stack are output of the `consoleOutput` port.)
consoleOutput	This port outputs the standard output stream and the standard error stream of the executable that communicates with BACnet.

4.9.4.3 Synchronization with real-time

In most cases, the BCVTB should be synchronized to real time. This can be done in Ptolemy II by double-clicking the director icon, and activate the check-box `synchronizeToRealtime`.

4.10 Analog/Digital Interface

Note
The ADInterfaceMCC (Analog/Digital Interface) is an interface between the BCVTB and USB-data acquisition devices. This interface has been tested for one device (USB-1208LS) of the Measurement Computing Corporation and is only supported on Windows. This interface will work properly only if the drivers of the Data Acquisition device have been installed correctly and the device has been configured. Read the user guide available at http://www.mccdaq.com/PDFmanuals/USB-1208LS.pdf and the Data Acquisition software manual http://www.mccdaq.com/PDFs/Manuals/DAQ-Software-Quick-Start.pdf to learn how to install and to configure the USB-1208LS.

4.10.1 Introduction

The BCVTB contains an actor, called `ADInterfaceMCCReader`, that can read from ADInterfaceMCC devices and an actor, called `ADInterfaceMCCWriter`, that can write to ADInterfaceMCC devices. Both actors use a configuration file that specifies the ADInterfaceMCC devices. The next sections describe how to write these configuration files, and how to set up a model that reads from and writes to ADInterfaceMCC devices.

4.10.2 Reading from ADInterfaceMCC

4.10.2.1 Specification of data that will be read from ADInterfaceMCC

The `ADInterfaceMCCReader` actor reads an xml configuration file to determine what data it needs to read from ADInterfaceMCC devices. The xml configuration file has the following syntax: It starts and ends with

```
<?xml version="1.0" encoding="utf-8"?>
     <ADInterfaceMCC>
     <!-- Child elements are not shown. -->
     </ADInterfaceMCC>
```

The above element `ADInterfaceMCC` has at least one child element of the form

```
<Object BoardNumber = "0" ChannelNumber = "0" ChannelGain = "2" ChannelOptions = " ↩
    0"  ApplicationTag = "read"/>
```

The element name needs to be set to `Object`.

The attribute `BoardNumber` is the board number associated with the board used to collect the data when it was installed with InstalCal.

The attribute `ChannelNumber` is the A/D (Analog/Digital) channel number.

The attribute `ChannelGain` is the A/D range code. If the board has a programmable gain, it will be set according to this argument value.

The attribute `ChannelOptions` is reserved for future use by the hardware manufacturer and should be set to zero.

The attribute `ApplicationTag` is a value that can be set to `read` or `write` depending on whether the interface should be used in the `READ` or `WRITE` mode. Table 4.7 gives a short overview about the values that the attributes could have in the configuration file.

Attribute	**Value**
BoardNumber	Integer from 0 to 99
ChannelNumber	Integer from 0 to 7
ChannelGain	1: channel with 10 volts unipolar 2: channel with 10 volts bipolar 3: channel with 5 volts unipolar 4: channel with 5 volts bipolar
ChannelOptions	0
ApplicationTag	read

Table 4.7: Attribute's values for the USB-1208LS Data Acquisition Device (`READ` mode).

4.10.2.2 Interface to adInterfaceMCC-Stack

To read data from ADInterfaceMCC devices, the `ADInterfaceMCCReader` actor calls an executable program that is in the directory `bcvtb/lib/adInterfaceMCC-stack`. This section describes how the entries in the xml file relate to this executable. The example shows the low-level implementation.

To read from ADInterfaceMCC, the adInterfaceMCC-stack provides the following function:

```
java -jar adInterfaceReader.jar BoardNumber ChannelNumber ChannelGain ↩
   ChannelOptions
```

4.10.3 Writing to ADInterfaceMCC

4.10.3.1 Specification of data that will be written to ADInterfaceMCC

The `ADInterfaceMCCWriter` actor reads an xml configuration file to determine what data it needs to write to ADInterfaceMCC devices. The xml configuration file has the following syntax: It starts and ends with

```
<?xml version="1.0" encoding="utf-8"?>
     <ADInterfaceMCC>
     <!-- Child elements are not shown. -->
     </ADInterfaceMCC>
```

The above element `ADInterfaceMCC` has at least one child element of the form

```
<Object BoardNumber = "0" ChannelNumber = "0" ChannelGain = "3" ChannelOptions = " ↩
   0"  ApplicationTag = "write"/>
```

The element name needs to be set to `Object`.

The attribute `BoardNumber` is the board number associated with the board used to collect the data when it was installed with InstalCal.

The attribute `ChannelNumber` is the D/A (Digital/Analog) channel number. The maximum allowable channel depends on which type of D/A board is being used.

The attribute `ChannelGain` is the D/A range code. If the device has a programmable gain, it will be set according to this argument value. If the specified range is not supported, the function will return a BADRANGE error. If the gain is fixed or manually selectable, you must make sure this argument matches the gain configured for the device.

The attribute `ChannelOptions` is reserved for future use by the hardware manufacturer and should be set to zero.

The attribute `ApplicationTag` is a value that can be set to `read` or `write` depending on whether the interface should be used in the `READ` or `WRITE` mode. Table 4.8 gives a short overview about the values that the attributes could have in the configuration file.

Attribute	**Value**
BoardNumber	Integer from 0 to 99
ChannelNumber	0 or 1
ChannelGain	1: channel with 10 volts unipolar 2: channel with 10 volts bipolar 3: channel with 5 volts unipolar 4: channel with 5 volts bipolar
ChannelOptions	0
ApplicationTag	write

Table 4.8: Attribute's values for the USB-1208LS Data Acquisition Device (`WRITE` mode).

4.10.3.2 Interface to adInterfaceMCC-Stack

To write data to ADInterfaceMCC devices, the `ADInterfaceMCCWriter` actor calls an executable program that is provided by the adInterfaceMCC-stack. This section describes how the entries in the xml file relate to this executable. The example shows the low-level implementation.

To write to ADInterfaceMCC, the adInterfaceMCC-stack provides the following function:

```
java -jar adInterfaceWriter.jar BoardNumber ChannelNumber ChannelGain  ↩
   ValueToBeWritten ChannelOptions
```

4.10.4 Creating a Ptolemy II model

The `ADInterfaceMCCReader` and the `ADInterfaceMCCWriter` actor can be used in the same Ptolemy II model. In this section, we will explain how to configure a Ptolemy II model that writes to and reads from ADInterfaceMCCdevices. This example can be found in the directory `bcvtb/examples/adInterfaceMCC-roomControl`. A similar example can also be found in the directory `bcvtb/examples/adInterfaceMCC-room`.

4.10.4.1 Configuring the `ADInterfaceMCCReader` and `ADInterfaceMCCWriter`

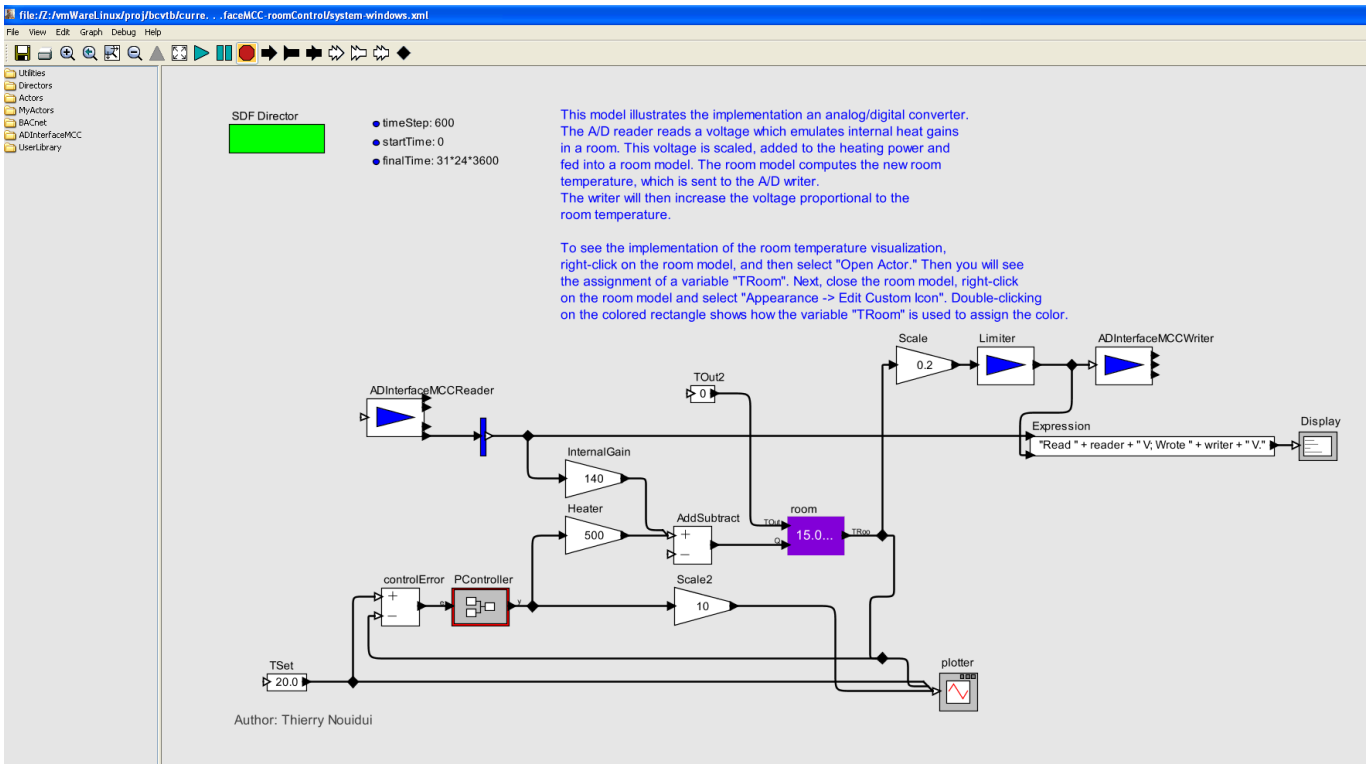

Figure 4.34: Ptolemy II system model that uses the `ADInterfaceMCCReader` and the `ADInterfaceMCCWriter` actors.

Figure 4.34 shows a Ptolemy II system model that uses the `ADInterfaceMCCReader` and the `ADInterfaceMCCWriter` actors. To configure the `ADInterfaceMCCReader`, double-click on its icon and add the name of its configuration file that has been developed as described in Section 4.10.2 *Reading from ADInterfaceMCC* . There is also a check-box called `continueWhenError`. If activated and an error occurs, then Ptolemy II will continue the simulation and the actor will output at its ports the last known value and the error message, unless the error occurs in the first step, in which case the simulation stops. If deactivated and an error occurs, then the simulation will stop, the error message will be displayed on the screen and the user is required to confirm the error message by clicking on its OK button. *Thus, select this box if the BCVTB should continue its operation when a run-time error occurs.*

The `ADInterfaceMCCReader` has the following ports: There is one input port, which is a trigger port. If the SDF Director is used in the Ptolemy II system model, then this port need not be connected. The `ADInterfaceMCCReader` has the following output ports:

Port	Description
errorSignal	If there were no errors in the previous data exchange, then this port outputs zero. Otherwise, the output is a non-zero integer.
errorMessage	If there was an error in the previous data exchange, then this port outputs the error message that was generated by the `ADInterfaceMCCReader` actor. (The error messages that were generated by the binary of the ADInterfaceMCC are output of the `consoleOutput` port.)
consoleOutput	This port outputs the standard output stream and the standard error stream of the binary that communicates with ADInterfaceMCC.

Port	Description
propertyValue	This port outputs the values obtained at the last successful communication with the ADInterfaceMCC device. The output data type is a vector whose elements are string representations of the ADInterfaceMCC properties that are read according to the configuration file. Elements can be extracted from this vector using actors from Ptolemy II's `Actors->VectorDisassembler` library.

The configuration of the `ADInterfaceMCCWriter` actor is similar to the configuration of the `ADInterfaceMCCReader`. To configure the `ADInterfaceMCCWriter`, double-click on its icon and add the name of its configuration file that has been developed as described in Section 4.10.3 *Writing to ADInterfaceMCC* . There is also a check-box called `continueWhenError`. If activated and an error occurs, then Ptolemy II will continue the simulation and the actor will output the error message at its ports, unless the error occurs in the first step, in which case the simulation stops. If deactivated and an error occurs, then the simulation will stop, the error message will be displayed on the screen and the user is required to confirm the error message by clicking on its OK button. *Thus, select this box if the BCVTB should continue its operation when a run-time error occurs.*

Input into the actor are values that will be written to the ADInterfaceMCC device according to the order specified in the xml configuration file.

The `ADInterfaceMCCWriter` has one multiport input. This port is used to collect the data that need to be sent to the ADInterfaceMCCdevices. The `ADInterfaceMCCWriter` has the following output ports:

Port	Description
errorSignal	If there were no errors in the previous data exchange, then this port outputs zero. Otherwise, the output is a non-zero integer.
errorMessage	If there was an error in the previous data exchange, then this port outputs the error message that was generated by the `ADInterfaceMCCWriter` actor. (The error messages that were generated by the binary of the ADInterfaceMCC are output of the `consoleOutput` port.)
consoleOutput	This port outputs the standard output stream and the standard error stream of the binary that communicates with ADInterfaceMCC.

4.10.4.2 Synchronization with real-time

In most cases, the BCVTB should be synchronized to real time. This can be done in Ptolemy II by double-clicking the director icon, and activate the check-box `synchronizeToRealtime`.

Chapter 5

Mathematics of the Implemented Co-Simulation

5.1 Introduction

This section describes the mathematical model that is used to implement the co-simulation. The section helps understanding when variables that are computed during the time integration are updated.

5.2 Description

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*. See [Hensen (1999)] and [Zhai and Chen (2005)] for details. 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 $f_1 : \Re^{n_1} \times \Re^{n_2} \rightarrow \Re^{n_1}$ and $f_2 : \Re^{n_2} \times \Re^{n_1} \rightarrow \Re^{n_2}$ 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

$$x_1(k+1) = f_1(x_1(k), x_2(k)),$$

and, similarly, the simulator 2 computes the sequence

$$x_2(k+1) = f_2(x_2(k), x_1(k)),$$

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 $\mathrm{k} = 0$, both simulators exchange their initial value $x_{1,0}$ and $x_{2,0}$. To advance from time k to $\mathrm{k}+1$, each simulator uses its own time integration algorithm. At the end of the time step, the simulator 1 sends the new state $x_1(\mathrm{k}+1)$ to the BCVTB and it receives the state $x_2(\mathrm{k}+1)$ 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

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

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

Step 0:	Initialize counter $\mathrm{k} = 0$ and number of steps $n \in \mathbb{N}$.
	Set initial state $\mathrm{x(k)} = \mathrm{x}_0$ and set time step $\Delta\mathrm{t} = 1/\mathrm{N}$.
Step 1:	Compute new state $\mathrm{x(k+1)} = \mathrm{x(k)} + h(\mathrm{x(k)})\,\Delta\mathrm{t}$.
	Replace k by $\mathrm{k}+1$.
Step 2:	If $\mathrm{k} = \mathrm{N}$ stop, else go to Step 1.

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

Step 0:	Initialize counter $\mathrm{k} = 0$ and number of steps $n \in \mathbb{N}$
	Set initial states $x_1(\mathrm{k}) = x_{1,0}$ and $x_2(\mathrm{k}) = x_{2,0}$.
	Set time step $\Delta\mathrm{t} = 1/\mathrm{N}$.
Step 1:	Compute new states
	$x_1(k+1) = x_1(k) + f_1(x_1(k), x_2(k))\,\Delta t$, and
	$x_2(k+1) = x_2(k) + f_2(x_2(k), x_1(k))\,\Delta t$.
	Replace k by $\mathrm{k}+1$.
Step 2:	If $\mathrm{k} = \mathrm{N}$ stop, else go to Step 1.

This algorithm is implemented in the BCVTB. It does not require an iteration between the two simulators.

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 was compared by [Trcka 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 [Trcka et al. (2007)].

Chapter 6

Development

6.1 Introduction

This chapter contains information that is of interest to developers who compile or extend the BCVTB to provide new functionalities, or who link additional simulation programs to the BCVTB.

6.2 Functional requirements

The high level functional requirements for the BCVTB are:

- The BCVTB should be modular and simulation tool independent so that different clients can be coupled to it. Examples of clients are EnergyPlus, a BACnet compatible Building Automation System, MATLAB/Simulink, simulation environments for Modelica and visualization tools for the online plot of variables.
- For BACnet operation, the coupling should be fault tolerant in the sense that clients can proceed with their operation even if no updated values are available from BACnet. This situation can occur if communication problems prevent BACnet from sending updated values.
- The BCVTB should allow users to couple different simulation programs or Building Automation Systems without having to modify source code of the BCVTB.
- 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.
- The BCVTB should allow communication using BSD sockets or BACnet, and allow users to add other communication mechanism as needed.
- The BCVTB should run on Windows, Linux and Mac OS X.

6.3 Software requirements

The BCVTB has been compiled on Linux Ubuntu 10.04, Mac OS X 10.5 and Windows XP Professional.

To install the tools required for developing and compiling the BCVTB, proceed first as described in Section 2.2 *Installation* to install the BCVTB. Then, install the software described below and compile the BCVTB.

For development, the following additional software need to be installed.

6.3.1 Linux

Linux requires the Java Development Kit 5.0, Update 14 or higher, which may be obtained from http://java.sun.com/javase/downloads/index.jsp

Also required are the GNOME xml library, the expat library, and the Doxygen source code documentation generator tool. These programs can be installed by typing in a shell

```
sudo apt-get install libxml2-dev libexpat-dev doxygen graphviz
```

To generate the documentation, the following packages are required

```
sudo apt-get install docbook docbook-xsl libsaxon-java libxalan2-java docbook-xsl- ↩
    saxon dblatex pdftk
```

6.3.2 Mac OS X

The XCode development environment is needed, which provides all required libraries as well as the required Java Development Kit.

To generate source code documentation, Doxygen is required.

6.3.3 Windows

The Java Development Kit is required. This may be obtained from http://java.sun.com/javase/downloads/index.jsp . Tested versions are version 5.0, update 14 and version 6, update 7. In the Windows Environment Variables setting, you may want to set the `Path` variable to `C:\Program Files\Java\jdk1.6.0_06\bin-;%Path%`

To generate source code documentation, Doxygen is required.

To compile C code or MATLAB/Simulink programs, Microsoft Visual C++ Express 2008 is required.

The Intel Fortran compiler can be used to compile and link Fortran programs to the BCVTB libraries.

MATLAB/Simulink provides its own compiler. However, due to compatibility problems with the BCVTB libraries, we only support the Microsoft compiler. MATLAB/Simulink can be configured to use the Microsoft compiler by typing at the MATLAB prompt

```
mex -setup
```

This command will provide a list of available compilers, from which the Microsoft compiler should be selected. MATLAB will write the selection to its configuration file; and hence the selection needs to be done only once.

6.4 Version control

The BCVTB source code can be accessed using the Subversion version control system under https://corbu.lbl.gov/svn/bcvtb/ , and it is distributed with the installation program.

To obtain an account, email MWetter@lbl.gov .

Please always keep the trunk of the repository in a working condition and work on your own branch for development and testing. Prior to committing changes to the trunk, make sure that all unit tests work without an error. How to run unit tests is described in Section 6.6 *Compiling the BCVTB* .

6.4.1 Checking out a version

To check out a release, type

```
svn checkout https://corbu.lbl.gov/svn/bcvtb/tags/releases/0.1.0/bcvtb
```

To check out the trunk, type

```
svn checkout https://corbu.lbl.gov/svn/bcvtb/trunk/bcvtb
```

6.4.2 Creating a branch

For own development and testing, create a branch using

```
cd branches/[your_login]
svn mkdir ../[your_login]/work
svn copy https://corbu.lbl.gov/svn/bcvtb/trunk/bcvtb ../[your_login]/work
svn co   ../[your_login] -m "Checked in working branch"
```

Prior to committing changes to the trunk, make sure that all unit tests work without an error. How to run unit tests is described in Section 6.6 *Compiling the BCVTB* . If all unit tests work without an error, proceed as follows:

6.4.3 Merging

To merge changes from your working branch to the trunk, proceed as follows:

1. Do a dry run to see what happens:

   ```
   mwetter@localhost:trunk$ svn merge --dry-run https://corbu.lbl.gov/svn/bcvtb/ ↩
      trunk https://corbu.lbl.gov/svn/bcvtb/branches/[your_login]/work ../trunk
   D    bcvtb/bin/file1.txt
   A    bcvtb/bin/file2.txt
   ```

   Here, `file1.txt` will be deleted and `file2.txt` will be added.

2. Merge the files:

   ```
   mwetter@localhost:trunk$ svn merge https://corbu.lbl.gov/svn/bcvtb/trunk  ↩
      https://corbu.lbl.gov/svn/bcvtb/branches/mwetter/work ../trunk
   D    bcvtb/bin/file1.txt
   A    bcvtb/bin/file2.txt
   ```

   This updates the *local* copy of the repository.

3. Run the unit test by running from the `bcvtb/example` directory the command

   ```
   ant unitTest
   ```

4. If successful, commit the changes in your local copy of the trunk to the repository:

   ```
   mwetter@localhost:trunk$ svn commit -m "merged changes from mwetter/work  ↩
      branch to trunk"
   ```

5. If there are problems and you need to revert to the latest copy of your local repository, type

   ```
   svn revert -R ../trunk
   ```

6.4.4 Resources

For SVN instructions, see the online book *Version Control with Subversion* : http://svnbook.red-bean.com/nightly/en/index.html

For SVN clients, see http://subversion.tigris.org/project_packages.html

6.5 Updating Ptolemy II

The BCVTB is a combination of a subset of the Ptolemy II software package and code developed by LBNL. This section explains how to update the subset of Ptolemy II that is used by the BCVTB. The process is the same for Linux and Mac OS X, and the files produced by this process will run on Linux, Mac OS X and Windows.

To update the subset of Ptolemy II that is used by the BCVTB, proceed as follows:

1. Download the Ptolemy II source code from http://ptolemy.berkeley.edu/ptolemyII/ptII8.0/index.htm to a directory, say to `~/ptII-dev`.

2. Compile Ptolemy II by typing

   ```
   cd ~/ptII-dev
   export PTII=`pwd`
   rm -f config.*
   ./configure
   make fast install
   ```

3. Go to the directory where the BCVTB is installed, and type

   ```
   export BCVTB_PTIISrc=$PTII
   ant updatePtolemyFiles
   export PTII=""
   ```

   This will copy the subset of Ptolemy II that is used by the BCVTB to the directory `bcvtb/lib/ptII-/ptolemy`. The statement `export PTII=""` avoids that the Ptolemy II distribution in `~/ptII-dev` is used.

4. Optionally, delete the directory `~/ptII-dev`.

6.6 Compiling the BCVTB

To compile the BCVTB and to run unit tests, the Apache Ant build tool is used.

6.6.1 Compiling the BCVTB

To compile the BCVTB, change to the BCVTB root directory and proceed as follows:

1. On Windows, double-click the file `bcvtb/bin/setDevelopmentEnvironment.bat`, or Linux and Mac, type `source bin/setDevelopmentEnvironment.sh`. This will detect your system configuration, set some environment variables, write the file `bcvtb/build.properties` and open a console.

2. To see a list with available targets, type

```
ant -p
```

3. To delete old binary files and recompile the BCVTB, run

```
ant clean all
```

   Note that this command can be run from any directory in `bcvtb/lib` or in `bcvtb/examples`. This allows a recursive compilation of an individual directory and any of its subdirectories.

4. To run unit tests, run

```
ant unitTest
```

If there are problems, more output can be obtained by typing

```
ant diagnostics
```

and by adding the flag `-v` to any ant command.

6.6.2 Custom configuration

Ant reads two configuration files: `build.properties` which is generated by `bcvtb/bin/setDevelopmentEnvironment.bat` (on Windows) and `bcvtb/bin/setDevelopmentEnvironment.sh` (on Mac OS X and Linux), and `user.properties` which is not changed by any program. Any settings in `user.properties` will overwrite the settings in `build.properties`.

For example, when executing `bcvtb/bin/setDevelopmentEnvironment.bat`, the following line may be added to `build.properties` if MATLAB is installed:

```
haveMatlab=true
```

To overwrite this setting, specify in `user.properties` a line of the form

```
haveMatlab=false
```

This will tell the Ant build system that MATLAB is not installed on this computer.

6.7 Structure of the file system

Table 6.1 shows the structure of the file system. Each directory contains an Apache Ant build file called `build.xml` that can be used to compile code and run unit tests. These files recursively run targets in all their subdirectories. See Section 6.6 *Compiling the BCVTB* for details.

6.8 Running unit tests

After making changes to the BCVTB source files, we recommend to run all examples to ensure that no errors have been introduced. This can be done by changing to the `bcvtb` directory and typing

```
ant clean all unitTest
```

If all examples work without errors, the console will show the message `BUILD SUCCESSFUL`.

Directory	**Contents**
bin	Scripts to set environment variables, the jar file that starts the BCVTB, and scripts to start the BCVTB or to start simulation programs.
doc	Documentation.
doc/code	Auto-generated source code documentation.
doc/manual	Source files, pdf and html files for manual.
examples	Example problems that are used to illustrate the use of the BCVTB and to conduct unit tests.
install	Files to build the installer.
lib	Library files that are used by various programs.
lib/apache-ant	Apache Ant build system that is used to compile the BCVTB.
lib/bacnet-stack	Source code and executables for the BACnet interface.
lib/config	Code for detecting the configuration on Windows systems.
lib/launcher	Code for building the jar file that launches the BCVTB.
lib/linux	Files that are used on Linux only. This directory contains, for example, the expat parser.
lib/matlab	MATLAB and Simulink source code and libraries that are needed to connect MATLAB and Simulink to Ptolemy II.
lib/modelica	C source code that is called by Modelica to link to Ptolemy II. The Modelica source code is distributed with the Buildings library.
lib/pt	Binaries of a subset of Ptolemy II that is used for the BCVTB.
lib/util	Code that implements the socket connection for the clients and the xml file parsing.
lib/windows	Files that are used on Windows only. This directory contains the C runtime library files that are needed by users who did not install the Microsoft Developer Studio. It also contains the expat xml parser.
lib/xml	Code to validate the xml file `variables.cfg`.

Table 6.1: Structure of the file system.

6.9 Adding actors

The easiest way to add new actors is to add them to the directory `lib/ptII/myActors` which is described in Section 6.9.1 *Adding actors to lib/ptII/myActors*. Section 6.9.2 *Adding actors to other directories* describes how to add actors to another directory.

6.9.1 Adding actors to `lib/ptII/myActors`

Users can add new actors in the form of a Java class to the BCVTB. To add an actor, proceed as follows:

1. Create a Ptolemy II actor in the directory `bcvtb/lib/ptII/myActors`. This may be easiest by copying and modifying an existing actor, such as done in the example `bcvtb/lib/ptII/myActors/MyRamp.java`. For instructions about creating actors, see http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.pdf .

2. Edit the file `bcvtb/lib/ptII/myActors/myActor.xml` to include the new actor. This can be done by making a copy of the existing line

   ```
   <entity name="MyRamp" class="myActors.MyRamp"/>
   ```

   Edit the value of the `name` attribute (which is the name of the Java file without extension) and the `class` attribute (which is the Java package name).

3. To compile the actor, type on a command shell

   ```
   cd bcvtb/lib/ptII/myActors
   ant all
   ```

   After the compilation, the message `BUILD SUCCESSFUL` should be displayed.

When the BCVTB is restarted, the new actor should be displayed in the actor menu.

6.9.2 Adding actors to other directories

Note
Note that adding actors to directories other than `bcvtb/lib/ptII/myActors` requires having Ptolemy II installed and compiled as described in Section 6.5 *Updating Ptolemy II*.

If the actor is in a different directory than `bcvtb/lib/ptII/myActors`, then the following additional steps are required:

1. Create an entity file (similar to `lib/ptII/myActors/myActor.xml`) that points to the new actor.

2. Open the file `bcvtb/lib/ptII/build.xml`.

3. Make a copy of the section

```
<entity name="MyActors" class="ptolemy.moml.EntityLibrary">
  <configure>
    <?moml
      <group>
<input source="myActors/myActor.xml"/>
      </group>
    ?>
  </configure>
</entity>
```

Replace `MyActors` with the name that you want to see in the class browser of the Ptolemy II graphical user interface, and replace the value `myActors/myActor.xml` with the path and name of the entity file of the new actor.

4. Run

```
cd bcvtb/lib/ptII
ant updatePtolemyFiles
```

This will copy the entity section to the Ptolemy II configuration file.

6.10 Linking a simulation program to the BCVTB

This section describes an example that illustrates how to link a simulation program to the BCVTB in such a way that they exchange data at a fixed time step through a BSD socket connection. We will consider a system with two rooms. Each room has a heater that is controlled by a proportional controller. We will implement the simulation program for the two rooms in a C program, and we will link it to a controller that is implemented in Ptolemy II.

Let $k \in \{1, 2, ...\}$ 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

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

with initial conditions $T^i(0) = T_0^i$, where Δ t is the time interval, C^i is the room thermal capacity, $(\mathrm{UA})^i$ 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 these equations, 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 $u^i(k+1) = \min(1, \max(0, \gamma^i(T_{set}^i - T^i(k))))$, 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.

Figure 6.1 shows a source code snippet of the implemented client. This source code can be found in the directory `bcvtb/examples/c-room`. A similar implementation in Fortran 90 can be found in the directory `bcvtb/examples/f90-room`.

```c
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 6.1: Source code for a model of two rooms that is implemented in the C language.

There are three functions that interface the client with the BCVTB: The function call `establishclientsocket` establishes 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 compile the source code, type on a command shell

```
cd bcvtb/examples/c-room
ant all
```

This will invoke the ant build system, which calls the file `bcvtb/examples/c-room/build.xml` that contains the compiler and linker commands.

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. Figure 6.2 shows the system diagram with the actor for the controller and the actor that interfaces the simulation program.

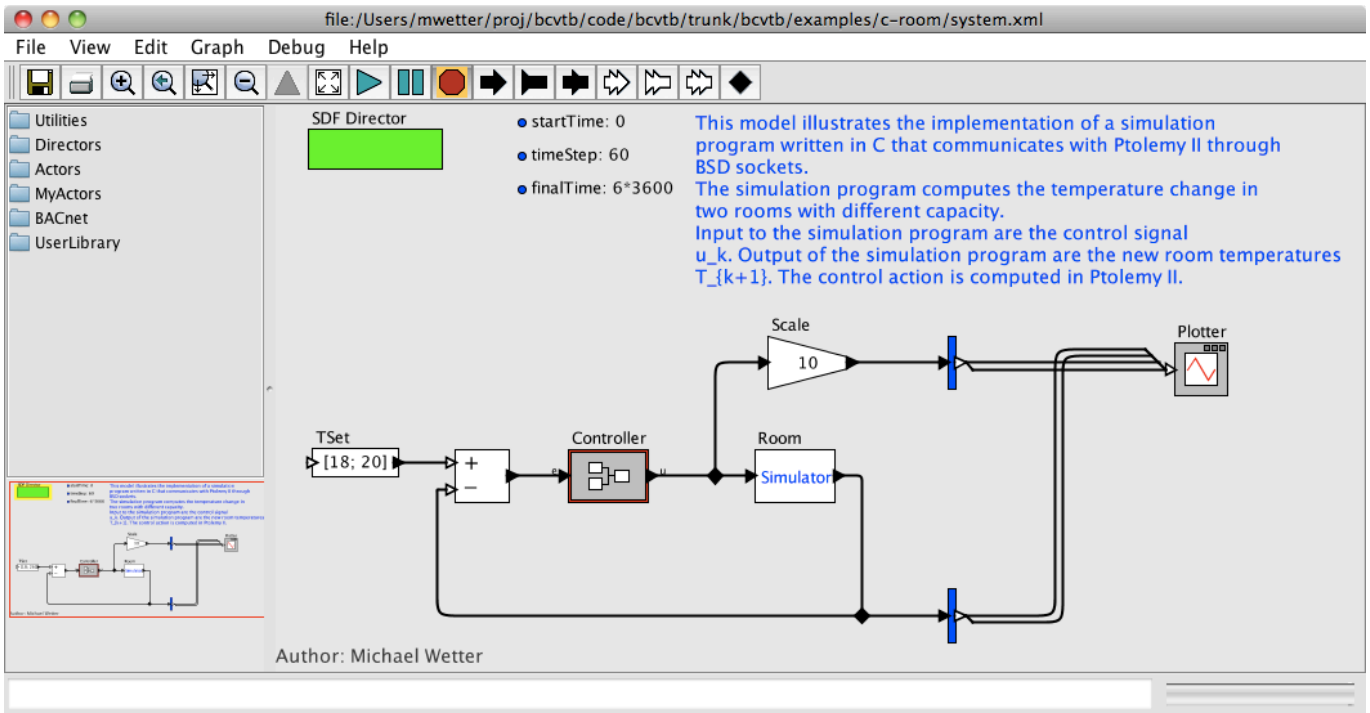

Figure 6.2: Ptolemy II system model that connects a model of a controller and a room.

6.11 Data exchange between Ptolemy II and programs that are started by the Simulator actor

Simulation programs that are started by the `Simulator` actor exchange data with Ptolemy II through a BSD socket connection. Each simulation program has its own socket connection. The exchange data is parsed into a text string, and this text string is sent from the simulation program to the `Simulator` actor, and from the `Simulator` actor to the simulation program.

The text string has the following format:

```
a b c d e f g_1 g_2
```

where `a` is the version number that is defined by the constant `MAINVERSION` in `lib/defines.h` and `b` is a flag that is defined in Table 6.2. What follows are the number of variables that are exchanged. In particular, `c` is the number of doubles, `d` is the number of integers and `e` is the number of booleans that will be exchanged. Currently, `d`

and `e` need to be set to `0`. Next, `f` is the current simulation time in seconds. The remaining entries `g_1`, `g_2` up to `g_c` are the double values. The string is terminated by the character `\n`.

The flag `b` is defined as follows:

Flag	**Description**
+1	Simulation reached end time.
0	Normal operation.
-1	Simulation terminated due to an unspecified error.
-10	Simulation terminated due to an error during the initialization.
-20	Simulation terminated due to an error during the time integration.

Table 6.2: Definition of flag of BSD Socket message

An example where 2 values are sent at time equals 60 looks like

```
2 0 2 0 0 6.000000000000000e+01 9.958333333333334e+00 9.979166666666666e+00
```

To stop a simulation program because the final time has been reached, send the following string:

```
2 1 0 0 0
```

Chapter 7

Acknowledgements

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.

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.
- Rui Zhang from Carnegie Mellon for her contributions to the Windows configuration and the EnergyPlus 3.1 upgrade.
- Zhengwei Li from the Georgia Institute of Technology for the implementation of the BACnet interface.
- We thank Andrew McNeill from LBNL for providing the Radiance example.
- We thank Thierry S. Nouidui from LBNL for the implementation of the analog/digital interface.

Chapter 8

Bibliography

[ASHRAE 2004] , *ANSI/ASHRAE Standard 135-2004*, BACnet - A Data Communication Protocol for Building Automation and Control Networks, 2004, 1041-2336.

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

[Trcka et al. (2007)] Marija Trcka, Michael Wetter, and Jan L.M. Hensen, *Comparison of co-simulation approaches for building and HVAC/R Simulation*, , 2007.

[Zhai and Chen (2005)] Z.J. ZhaiQ.Y. Chen, *Performance of coupled building energy and CFD simulations*, , 4, 2005, 333-344.