OpenBuildingControl

Control Description Language

Feb 06, 2019

Contents

Chapter 1

Preamble

1.1 Purpose of the Document

This document describes the process workflow, use cases, requirements and specification of the Control Description Language (CDL). It also describes a case study that illustrates the use of CDL for performance comparison of a control sequence during design.

The document is a working document that is used as a discussion basis and will evolve as the development progresses. The proposed design should not be considered finalized.

Chapter 2

Conventions

1. We write a requirement *shall* be met if it must be fulfilled. If the feature that implements a shall requirement is not in the final system, then the system does not meet this requirement. We write a requirement *should* be met if it is not critical to the system working, but is still desirable.
2. Text in bracket such as "[. . .]" denotes informative text that is not part of the specification.
3. Courier font names such as `input` denote variables or statements used in computer code.

Chapter 3

Process Workflow

Fig. 3.1 shows the process of selecting, deploying and verifying a control sequence that we follow in OpenBuildingControl. First, given regulations and efficiency targets, labeled as (1) in Fig. 3.1, a design engineer selects, configures, tests and evaluates the performance of a control sequence using building energy simulation (2), starting from a control sequence library that contains ASHRAE GPC 36 sequences, as well as user-added sequences (3), linked to a model of the mechanical system and the building (4). If the sequences meet closed-loop performance requirements, the designer exports a control specification, including the sequences and functional verification tests expressed in the Controls Description Language CDL (5). Optionally, for reuse in similar projects, the sequences can be added to a user-library (6). This specification is used by the control vendor to bid on the project (7) and to implement the sequence (8) in product-specific code. Prior to operation, a commissioning provider verifies the correct functionality of these implemented sequences by running functional tests against the electronic, executable specification in the Commissioning and Functional Verification Tool (9). If the verification tests fail, the implementation needs to be corrected.

For closed-loop performance assessment, Modelica models of the HVAC systems and controls will be linked to a Modelica envelope model or to an EnergyPlus envelope model. This can currently be done through the External Interface, and a more direct coupling is in development through the Spawn of EnergyPlus project.

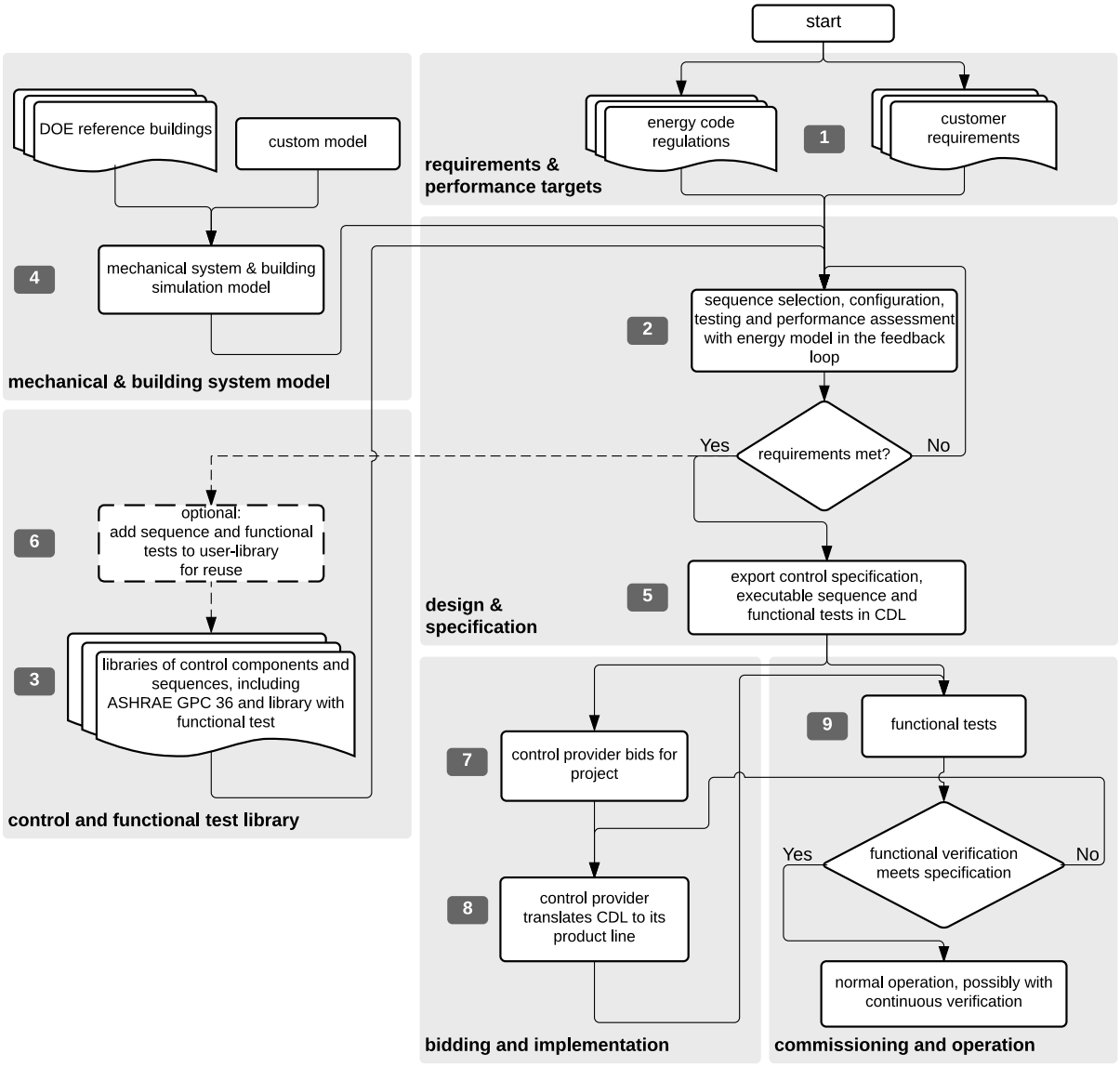


Fig. 3.1: Process workflow for controls design, specification and functional verification.

Chapter 4

Use Cases

This section describes use cases for end-user interaction, including the following:

- use the controls design tool to design a control sequence and export it as a CDL-compliant specification,
- use the CDL to bid on a project and, when selected for the project, implement the control sequence in a building automation system,
- use the control design tool to create control block diagrams in addition to control sequences and automatically produce a points list with a standard naming convention and/or tagging convention, a plain language sequence of operation, and verification that the control diagram includes all instrumentation required to complete the control sequence,
- use the commissioning and functional verification tool during commissioning

4.1 Controls Design

4.1.1 Loading a standard sequence from a library

This use case describes how to load, edit and store a control sequence from a library. For illustration, we use here a sequence from the Guideline 36 library.

Use case name	**Loading a standard sequence from Guideline 36**
Related Requirements	User able to change the pre-set elements within the standard sequence, with automatic download of associated CDL block diagram.
Goal in Context	Enable fast adaptation of Guideline 36
Preconditions	All Guideline 36 sequences need to be pre-programmed into visual block diagrams using CDL. CDL and block diagrams need to be modular so that they can be easily updated when key elements are changed/deleted/added.

Continued on next page

Table 4.1 – continued from previous page

Use case name	**Loading a standard sequence from Guideline 36**
Successful End Condition	User is able to download the CDL/block diagrams using a specific reference to Guideline 36 sequences. User is able to change/delete/add key elements using CDL.
Failed End Condition	Missing Guideline 36 sequence in library. When a user changes/deletes/adds elements to CDL/visual block diagram, no associated CDL/visual block diagram appears/disappears.
Primary Actors	Mechanical Designer/Consultant
Secondary Actors	Controls contractor
Trigger	Designing control system using Guideline 36 as default sequence or a starting point, then needs to change key elements because the system is different to Guideline 36 presumed system configuration.
Main Flow	**Action**
1	User opens Guideline 36 library and sees a contents menu of the standard sequences for selection.
2	User selects a sequence
3	The corresponding CDL and visual block diagram appears in the controls design tool. Key mechanical elements (e.g. fan, cooling coil valve, control damper) controlled by the standard sequence are also displayed.
Extensions	
1	User saves copy of the imported sequence prior to editing
2	User deletes/adds elementary blocks or composite blocks.
2	User saves the modified sequence.

4.1.2 Customizing a control sequence for an HVAC system

This use case describes how to connect a control sequence to a system model and then customize the control sequence, using a VAV system as an example.

Use case name	**Customizing a control sequence for a VAV system**
Related Requirements	n/a
Goal in Context	A mechanical engineer wants to customize a control sequence, starting with a template.
Preconditions	System model of the HVAC and building, with sensor output signals and actuator input signals exposed. Preconfigured control sequence, stored in the OpenBuildingControls library. A set of performance requirements.

Continued on next page

Table 4.2 – continued from previous page

Use case name	**Customizing a control sequence for a VAV system**
Successful End Condition	Implemented VAV sequence with customized control, ready for performance assessment (Use case *Performance assessment of a control sequence*) and ready for export in CDL for subsequent implementation.
Failed End Condition	n/a
Primary Actors	A mechanical engineer.
Secondary Actors	The controls design tool with template control sequences and a package with elementary CDL blocks. The HVAC plant and control sequence library.
Trigger	n/a
Main Flow	**Action**
1	The user opens the controls design tool in OpenStudio
2	The user drags and drops a preconfigured VAV control sequence from the Buildings library.
3	The user clicks on the pre-configured VAV control sequence and selects in the tool a function that will store the sequence in the project library to allow further editing.
4	The controls design tool saves the sequence in the project library.
5	The user connects sensors and actuators of the *plant* model to control inputs and outputs of the *controller* model.
6	The user opens the system model that is composed of controls, HVAC system model and building envelope in the controls design tool.
7	The user opens in the project library the composite sequence saved in step 4.
8	The user adds and connects additional control blocks from the elementary CDL-block library.
9	The user selects “Check model” to verify that the implemented sequence complies with the CDL specification.

Fig. 4.1 shows the sequence diagram for this use case.

4.1.3 Customizing and configuring a control sequence for a single-zone VAV system

This use case describes how to customize and configure a control sequence for a single zone VAV system.

Use case name	**Customizing a control sequence for a single-zone VAV system**
Related Requirements	n/a

Continued on next page

Table 4.3 – continued from previous page

Use case name	**Customizing a control sequence for a single-zone VAV system**
Goal in Context	A mechanical engineer wants to customize a control sequence, starting with a template.
Preconditions	A model of the *plant* (consisting of HVAC and building model). Preconfigured control sequence, stored in an OpenBuildingControls-compatible library. A set of performance requirements.
Successful End Condition	Implemented single zone VAV sequence with customized control, ready for performance assessment (Use case *Performance assessment of a control sequence*) and ready for export in CDL.
Failed End Condition	n/a
Primary Actors	A mechanical engineer.
Secondary Actors	The controls design tool with template control sequences and a package with elementary CDL blocks. The HVAC and controls library.
Trigger	n/a
Main Flow	**Action**
1	The user opens the controls design tool in OpenStudio.
2	The user opens the HVAC model and building model in the controls design tool.
3	The user drags and drops a single-zone VAV control sequence from the Buildings library into the tool.
4	The user clicks on the pre-defined single-zone VAV control sequence and selects a function that will store a copy of the sequence in the project library to allow further editing.
5	The controls design tool stores a copy of the sequence in the project library.
6	The user loads a copy of the sequence into the sequence editor.
7	The user specifies the mapping of the control points to HVAC system sensors and actuators, e.g. AHU
8	The user initiates the saving of the composite HVAC+building+control model, for use as a reference model against which to compare alternative control sequences
9	If necessary, the user executes the reference model and inspects the resulting performance to identify potential modifications
10	The user makes a copy of the sequence prior to replication and loads it into the sequence editor.

Continued on next page

Table 4.3 – continued from previous page

Use case name	**Customizing a control sequence for a single-zone VAV system**
11	The user edits the sequence by deleting and/or moving elementary and composite blocks and/or adding control blocks from the elementary CDL-block library
12	The user selects “Check model” to verify whether the implemented sequence complies with the CDL specification, editing and re-checking as necessary.
13	The user connects the modified sequence to the HVAC system and building models, using Step 7, and saves the resulting composite model
15	The user assesses the relative performance of the modified and unmodified sequences using the procedure defined in the ‘Performance assessment of a control sequence’ use case below.

4.1.4 Customizing and configuring a control sequence for a multizone VAV system

This use case describes how to customize and configure a control sequence for a multizone VAV system.

Use case name	**Customizing a control sequence for a multi-zone VAV system**
Related Requirements	n/a
Goal in Context	A mechanical engineer wants to customize a control sequence, starting with a template.
Preconditions	HVAC system model connected to building model. The repeated elements in the HVAC system model (i.e. the terminal boxes) must be tagged and numbered. Preconfigured control sequence, stored in an OpenBuildingControls-compatible library. The terminal boxes control blocks must be tagged to indicate that they can be replicated by a predefined function in the editor. A set of performance requirements.
Successful End Condition	Implemented multi-zone VAV sequence with customized control, ready for performance assessment (Use case *Performance assessment of a control sequence*) and ready for export in CDL.
Failed End Condition	n/a
Primary Actors	A mechanical engineer.
Secondary Actors	The controls design tool with template control sequences and a package with elementary CDL blocks. The HVAC and controls library.

Continued on next page

Table 4.4 – continued from previous page

Use case name	**Customizing a control sequence for a multi-zone VAV system**
Trigger	n/a
Main Flow	**Action**
1	The user opens the controls design tool in OpenStudio
2	The user opens the HVAC model and building model in the controls design tool.
3	The user drags and drops a multi-zone VAV control sequence from the Buildings library into the tool
5	The user clicks on the pre-defined VAV control sequence and selects a function that will store a copy of the sequence in the project library to allow further editing.
6	The controls design tool stores a copy of the sequence in the project library.
7	The user loads a copy of the sequence into the sequence editor.
8	The user specifies the number of zones (NZi) with each type of terminal box and selects a function that will replicate and instantiate sets of NZi terminal box control blocks for each type of terminal box
9	The tool replicates and instantiates NZi terminal box control blocks of each type
10	The user initiates a tool function that maps zones with specific types of terminal box to the corresponding terminal box control blocks and then applies a user-defined mapping of zone-level control points to terminal box sensors and actuators and zone temperature and occupancy sensors
11	The tool executes the actions described in Step 10
12	The user specifies the mapping of the remaining control points to HVAC system sensors and actuators, e.g. AHU
13	The user initiates the saving of the composite HVAC+building+control model, for use as a reference model against which to compare alternative control sequences
14	If necessary, the user executes the reference model and inspects the resulting performance to identify potential modifications
15	The user makes a copy of the reference/library sequence prior to replication and loads it into the sequence editor.
16	The user edits the sequence by deleting and/or moving elementary and composite blocks and/or adding control blocks from the elementary CDL-block library

Continued on next page

Table 4.4 – continued from previous page

Use case name	Customizing a control sequence for a multi-zone VAV system
17	The user selects “Check model” to verify whether the implemented sequence complies with the CDL specification, editing and re-checking as necessary.
18	The user connects the modified sequence to the HVAC system and building models, using Steps 8-12, and saves the resulting composite model
19	The user assesses the relative performance of the modified and unmodified sequences using the procedure defined in the ‘Performance assessment of a control sequence’ use case below.

4.1.5 Performance assessment of a control sequence

This use case describes how to assess the performance of a control sequence using the controls design tool.

Separate sequences are given below for the cases where local loop control is to be included in, or excluded from, the evaluation.

Use case name	Performance assessment of a control sequence
Related Requirements	n/a
Goal in Context	Evaluate the performance of a specific control sequence in the context of a particular design project.
Preconditions	Either a) whole building or system model for the particular design project, or b) sufficient information about the current state of the design, to enable the configuration of a model template based on a generic design for the appropriate building type. The model must be complete down to the required sensors and actuation points, which may be actual actuators, if the sequence includes local loop control, or set-points for local loop control, if the sequence only performs supervisory control. Control sequence to be assessed must match, or be capable of being configured to match, the building/system model in terms of sensing and actuation points and modes of operation. Relevant statutory requirements and design performance targets. Performance metrics derived from these requirements and targets.

Continued on next page

Table 4.5 – continued from previous page

Use case name	**Performance assessment of a control sequence**
Successful End Condition	User is able to (i) compare the performance of different control sequences in terms of selected pre-defined criteria, and (ii) evaluate the ability of a selected control sequence to enable the building/system to meet or exceed externally-defined performance criteria.
Failed End Condition	Building/system model or configuration information for generic model template is incomplete. Performance requirements or targets are incomplete or inconsistent wrt the specific control sequence Simulation fails to run to completion or fails convergence tests.
Primary Actors	A mechanical engineer.
Secondary Actors	
Trigger	Need to select or improve a control sequence for a building or system.
Main Flow	**Action**
1	User loads the building/system model for the project or uses design information to configure a model template.
2	User selects and loads weather data and operation schedules.
3	User configures control sequence with project-specific information, e.g. number of terminal units on an air loop, and connects to building/system model.
3a	If the sequence contains feedback loops that are to be included in the evaluation, these loops must be tuned, either automatically or manually.
4	User selects short periods for initial testing and performs predefined tests to verify basic functionality, similar to commissioning.
5	User initiates simulation of building/system controlled performance over full reference year or statistically-selected short reference year that reports output variables required to evaluate performance according to pre-defined metrics.
6	User compares metric values to requirements and/or targets and determines whether the sequence is acceptable as is, needs modification or appears fundamentally flawed.

4.1.6 Defining integration with non-HVAC systems such as lighting, façade and presence detection

This use case describe the connection of a facade control with the HVAC control in the control design tool.

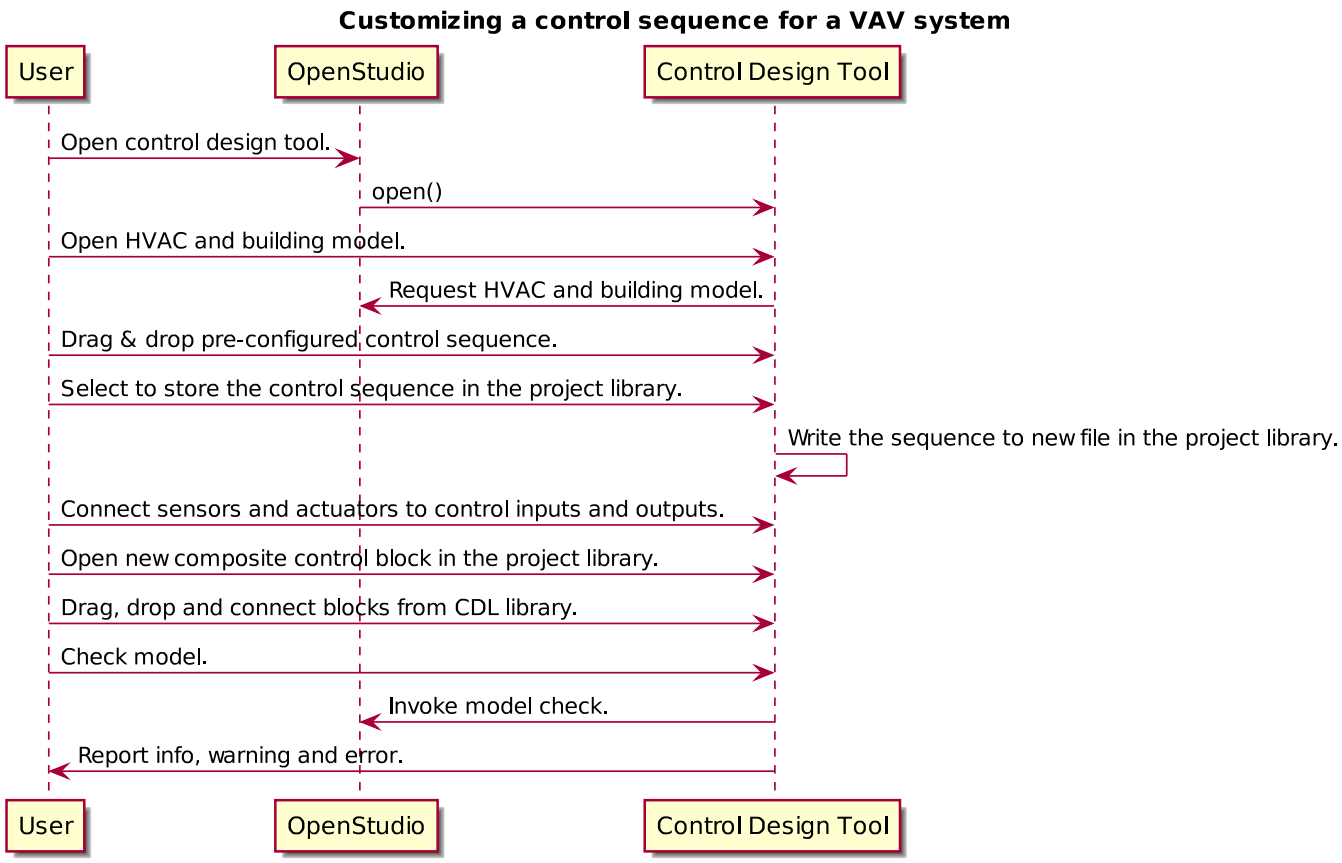


Fig. 4.1: Customizing a control sequence for a VAV system.

Use case name	**Defining integration with non-HVAC systems such as lighting, façade and presence detection**
Related Requirements	The model represents the non-HVAC systems and the associated control blocks are represented using CDL.
Goal in Context	Integration actions between HVAC and non-HVAC systems can be defined using CDL. Optional goal - Tool to also configures and verifies HVAC to non-HVAC integration.
Preconditions	Examples of HVAC and non-HVAC integrations available for adaptation using CDL, non-HVAC systems can be façade louvre control, lighting on/off or presence detection status.
Successful End Condition	User able to use CDL to define common HVAC and non-HVAC integrations
Failed End Condition	Failure to include HVAC and façade/lighting/presence detection interactions in CDL.
Primary Actors	Mechanical Designer/Consultant
Secondary Actors	
Trigger	
Main Flow	**Action**
1	User opens a menu of the non-HVAC systems for selection.
2	User selects the non-HVAC object and the visual block diagram and associated CDL elements appear.
3	User clicks on a non-HVAC object and a menu of status and actions pops up.
4	User selects the integration status or actions of the non-HVAC system, and links it to HVAC system status or action block

4.2 Bidding and BAS Implementation

4.2.1 Generate control point schedule from sequences

This use case describes how to generate control points from a sequence specification.

Use case name	**Generate control points schedule from sequences**
Goal in Context	The same control specification can be used to generate controls points schedule
Preconditions	Each control points needs to be defined using AI/AO/DI/-DO/Network interface types and consistent tagging/naming

Continued on next page

Table 4.7 – continued from previous page

Use case name	**Generate control points schedule from sequences**
Successful End Condition	Control points schedule can be automatically produced by extracting from the sequences, including tagging (AHU/T-DX/1), point name, point type and comments (such as differential pressure to be installed at 2/3 down index leg)
Failed End Condition	Control points schedule is inaccurate or doesn't contain sufficient information.
Primary Actors	Mechanical Designer/Consultant
Secondary Actors	Controls contractor
Trigger	
Main Flow	**Action**
1	When a user adds a control point in the controls design tool, the tool provides default values and allows the user to change the values for tagging/point name/point type/-comments
2	User clicks on a button to generate Points Schedule, an Excel file is then generated listing all the points and their details, and also counts the total number of different type of points.
3	User clicks on a button to generate a tag list of unique control devices within the project in Excel, so that the associated specification section can be extracted and populated within third party software.

4.3 Commissioning, Operation, and Maintenance

4.3.1 Conducting verification test of a VAV Cooling-Only Terminal Unit

This use case describes the verification of an installed control sequence relative to the design intent.

Use case name	**Conducting verification test of a VAV Cooling-Only Terminal Unit**
Related Requirements	
Goal in Context	A commissioning agent wants to verify on site that the controller operates in accordance with the sequence of operation
Preconditions	CDL-conformant control sequence and verification tests are imported into verification tool. Field instrumentation is per spec. Installation of field equipment is correct. Point-to-point testing from point in field through to graphic is correct.

Continued on next page

Table 4.8 – continued from previous page

Use case name	**Conducting verification test of a VAV Cooling-Only Terminal Unit**
Successful End Condition	Control devices carry out the right sequence of actions, and the verification tool verifies compliance with the design intent. Control devices carry out wrong sequence of actions, and the verification tool shows non-compliance with the design intent.
Failed End Condition	The verification tool fails to recognize verification success/failure.
Primary Actors	Commissioning agent
Secondary Actors	BMS engineer (optional) Vendor software which replicates uploaded CDL code
Trigger	The verification tool is connected to the BMS and receives the following signals from the VAV box controller: • occupied mode, unoccupied mode • Vmin, Vcool-max etc. • setpoints and timers The control parameters of the VAV box are configured and the results are compared to the output of the CDL code in the tool.
Main Flow 1	**Automatic Control Functionality Checks**
1	Set VAV box to unoccupied.
2	Set VAV box to occupied.
3	Continue through sequence, commissioning agent will get a report of control actions and whether they were compliant with the design intent.
Main Flow 2	**Commissioning Override Checks**
1	Force zone airflow setpoint to zero.
2	Force zone airflow setpoint to minimum flow.
3	Force damper full closed/open.
4	Reset request-hours accumulator point to zero (provide one point for each reset type).

4.3.2 As-Built Sequence Generator

This use case will confirm that the installed control sequence is similar to the intended sequence.

Use case name	**As-Built Sequence Generator**
Related Requirements	Tool can translate sequence logic to controls programming logic. Below would do this in reverse.

Continued on next page

Table 4.9 – continued from previous page

Use case name	**As-Built Sequence Generator**
Goal in Context	An owner's facilities engineer wishes to confirm the actual installed controls sequences in an existing building. This could be done as a Q/C step for new construction or to periodically document as-operating conditions.
Preconditions	Installed control system must be capable of communication with the tool. Translation protocol must be established.
Successful End Condition	
Failed End Condition	
Primary Actors	Owners facilities engineers
Secondary Actors	Owners HVAC technicians, new construction project managers
Trigger	Need for investigation of building performance. Or, periodic snap-shot documentation of as-installed controls sequences.
Main Flow	**Action**
1	User opens tool interface.
2	User configures tool to connect with desired control system.
3	User initiates translation of installed control logic to sequence documentation.

Chapter 5

Requirements

This section describes the functional, mathematical and software requirements. The requirements are currently in discussion and revision with the team.

In these discussion, by *plant*, we mean the controlled system, which may be a chiller plant, an HVAC system, an active facade, a model of the building etc.

5.1 Controls Design Tool

1. The controls design tool shall contain a library of predefined control sequences for HVAC primary systems, HVAC secondary systems and active facades in a way that allows users to customize these sequences.
2. The controls design tool shall contain a library with functional and performance requirement tests that can be tested during design and during commissioning.
3. The controls design tool shall allow users to add libraries of custom control sequences.
4. The controls design tool shall allow users to add libraries of custom functional and performance requirement tests.
5. The controls design tool shall allow testing energy, peak demand, energy cost, and comfort (for each instant of the simulation) of control sequences when connected to a building system model.
6. The controls design tool shall allow users to test control sequences coupled to the equipment that constitutes their HVAC system.
7. When the control sequences are coupled to plant models, the controls design tool shall allow users to tag the thermofluid dependencies between different pieces of equipment in the object model. [For example, for any VAV box, the user can define which AHU provides the airflow, which boiler (or system) provides the hot water for heating, etc.]
8. The control design tool shall include templates for common objects.
9. A design engineer should be able to easily modify the library of predefined control sequences by adding or removing blocks.
10. The controls design tool shall prompt the user to provide necessary information when instantiating objects. For example, the object representing an air handler should include fan, filter, and optional coil and damper elements (each of which is itself an object). When setting up an AHU instance, the user should be prompted to define which of these objects exist.
11. To the extent feasible, the control design tool shall prevent mutually exclusive options in the description of the

physical equipment. [For example, an air handler can have a dedicated minimum outside air intake, or it can have a combined economizer/minimum OA intake, but it cannot have both.]

12. The controls design tool shall hide the complexity of the object model from the end user.
13. The controls design tool shall integrate with OpenStudio.
14. The controls design tool shall work on Windows, Linux Ubuntu and Mac OS X.
15. The controls design tool shall either run as a webtool (i.e. in a browser) or via a standalone executable that can be installed including all its dependencies.

5.2 CDL

1. The CDL shall be declarative.
2. CDL shall be able to express control sequences and their linkage to an object model which represents the plant.
3. CDL shall represent control sequences as a set of blocks (see Section 7.3) with inputs and outputs through which blocks can be connnected.
4. It shall be possible to compose blocks hierarchically to form new blocks.
5. The elementary building blocks [such as an gain] are defined through their input, outputs, parameters, and their response to given outputs. The actual implementation is not part of the standard [as this is language dependent].
6. Each block shall have tags that provide information about its general function/application [e.g. this is an AHU control block] and its specific application [e.g. this particular block controls AHU 2].
7. It shall be possible to identify whether a block represents a physical sensor/actuator, or a logical signal source/sink. [As this is used for pricing.]
8. Blocks and their inputs and outputs shall be allowed to contain metadata. The metadata shall identify expected characteristics, including but not limited to the following. For inputs and outputs:
    1. units,
    2. a quantity [such as “return air temperature” or “heating requests” or “cooling requests”],
    3. analog or digital input or output, and
    4. for physical sensors or data input, the application (e.g. return air temperature, supply air temperature).

    For blocks:
    1. an equipment tag [e.g., air handler control],
    2. a location [e.g., 1st-floor-office-south], and
    3. if they represent a sensor or actuator, whether they are a physical device or a software point. [For physical sensors, the signal is read by a sensor element, which converts the physical signal into a software point.]
9. It shall be possible to translate control sequences that are expressed in the CDL to implementation of major control vendors.
10. It shall be possible to render CDL-compliant control sequences in a visual editor and in a textual editor.
11. CDL shall be a proper subset of Modelica 3.3 [Mod12]. [Section *Control Description Language* specifies what subset shall be supported. This will allow visualizing, editing and simulating CDL with Modelica tools rather than requiring a separate tool. It will also simplify the integration of CDL with the design and verification tools, since they use Modelica.]
12. It shall be possible to simulate CDL-compliant control sequences in an open-source, freely available Modelica environment.
13. It shall be possible to simulate CDL-compliant control sequences in the Spawn of EnergyPlus.
14. The object model must be rigorous, extensible and flexible.
15. The object model must be relational, inherently defining connections between different objects.
16. The system must support many-to-many relationships [For example, two parallel chilled water pumps can serve

three parallel chillers (see also Brick's "isPartOf" and "feeds").]

17. Each distinct piece of equipment [e.g. return air temperature sensor] shall be represented by a unique instance.

5.3 Commissioning and Functional Verification Tool

1. The CDL tool shall import verification tests expressed in CDL, and a list of control points that are used for monitoring and active functional testing.
2. The commissioning and functional verification tool shall be able to read data from, and send data to, BACnet, possibly using a middleware such as VOLTTRON or the BCVTB, or read archived data.
3. It shall be possible to run the tool in batch mode as part of a real-time application that continuously monitors the functional verification tests.
4. The commissioning and functional verification tool shall work on Windows, Linux Ubuntu and Mac OS X.

Chapter 6

Software Architecture

This section describes the software architecture of the controls design tool and the functional verification tool. In the text below, we mean by *plant* the HVAC and building system, and by *control* the controls other than product integrated controllers (PIC). Thus, the HVAC or building system model may, and likely will, contain product integrated controllers, which will be out of scope for CDL apart from reading measured values from PICs and sending setpoints to PICs.

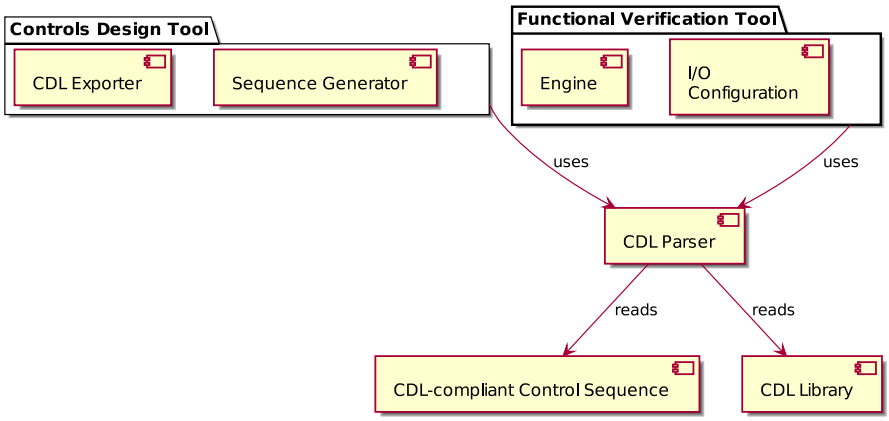


Fig. 6.1: Overall software architecture.

Fig. 6.1 shows the overall system with the *Controls Design Tool* and the *Functional Verification Tool*. Both use a *CDL Parser* which parses the CDL language. This parser is currently in development at https://github.com/lbl-srg/modelica-json.[1] The CDL parser reads a *CDL-compliant Control Sequence*, which may be provided by the user or taken from http://simulationresearch.lbl.gov/modelica/releases/v5.0.1/help/Buildings_Controls_OBC_ASHRAE.html and the *CDL Library*, see http://simulationresearch.lbl.gov/modelica/releases/v5.0.1/help/Buildings_Controls_OBC_CDL.html All these components will be made available through OpenStudio. This allows using the OpenStudio model authoring and simulation capability that is being developed for the Spawn of EnergyPlus (SOEP). See also https://www.energy.gov/eere/buildings/downloads/spawn-energyplus-soep and its development site https://lbl-srg.github.io/soep/softwareArchitecture.html.

[1] Using a parser that only requires Java has the advantage that it can be used in other applications that may not have access to a JModelica installation.

6.1 Controls Design Tool

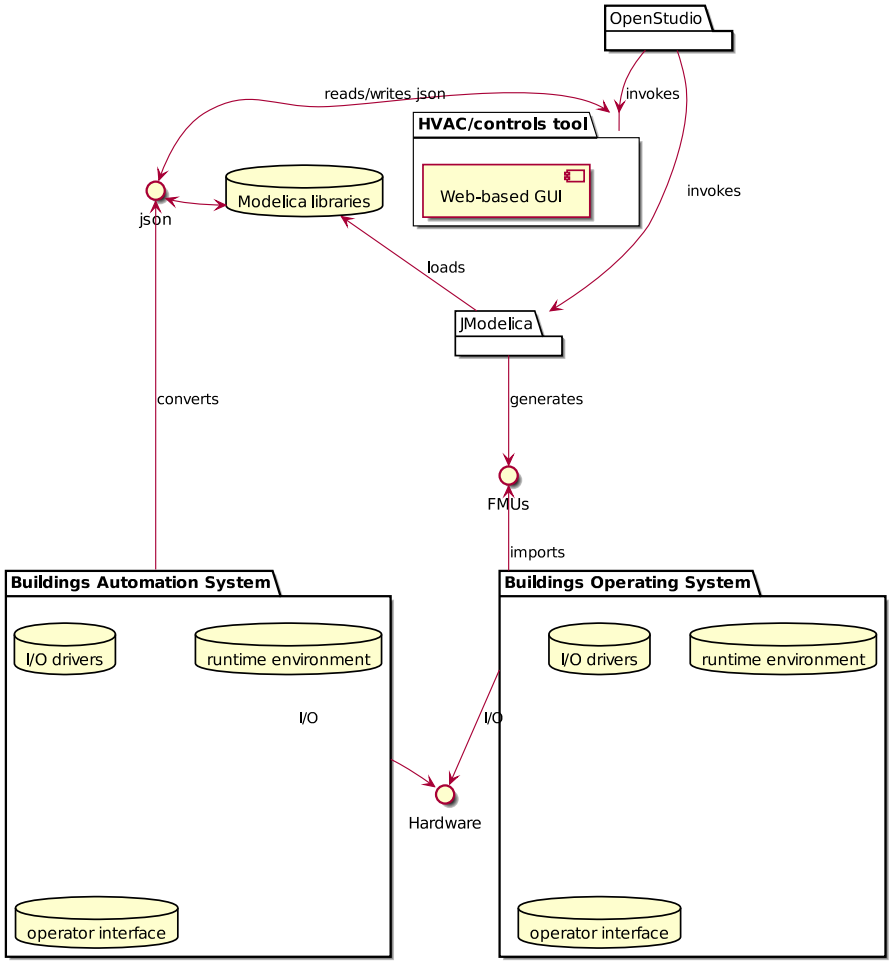


Fig. 6.2: Overall software architecture of the Controls Design Tool.

Fig. 6.2 shows the overall software architecture of the controls design tool. The *OpenStudio* invokes a Modelica to json parser which parses the Modelica libraries to *json*, and it invokes the *HVAC/controls tool*. The *HVAC/controls tool* reads the json representation of the Modelica libraries that are used. The *HVAC/controls tool* updates the json reprensentation of the model, and these changes will be merged into the Modelica model or Modelica package that has been edited. For exporting the sequence for simulation or for operation, *OpenStudio* invokes *JModelica* which generates an FMU of the sequence, or multiple FMUs if the sequence is to be distributed to different field devices. The *Building Operating System* then imports these FMUs.

If a *Building Automation System* prefers not to run FMUs to compute the control signals, then it could convert the json format to a native implementation of the control sequence.

Optionally, to aid the user in customizing sequences, a *Sequence Generator* could be generated. This is currently not shown in Fig. 6.2. The *Sequence Generator* will guide the user through a series of questions about the plant and control, and then generates a *Control Model* that contains the open-loop control sequence. This *Control Model* uses the CDL language, and can be stored in the *Custom or Manufacturer Modelica Library*. Using the *HVAC/controls tool*, the user will

then connect it to a plant model (which consist of the HVAC and building model with exposed control inputs and sensor outputs). This connection will allow testing and modification of the *Control Model* as needed. Hence, using the *Schematic editor*, the user can manipulate the sequence to adapt it to the actual project.

How sequences can be exported to control systems is described in Section 8.

6.2 Functional Verification Tool

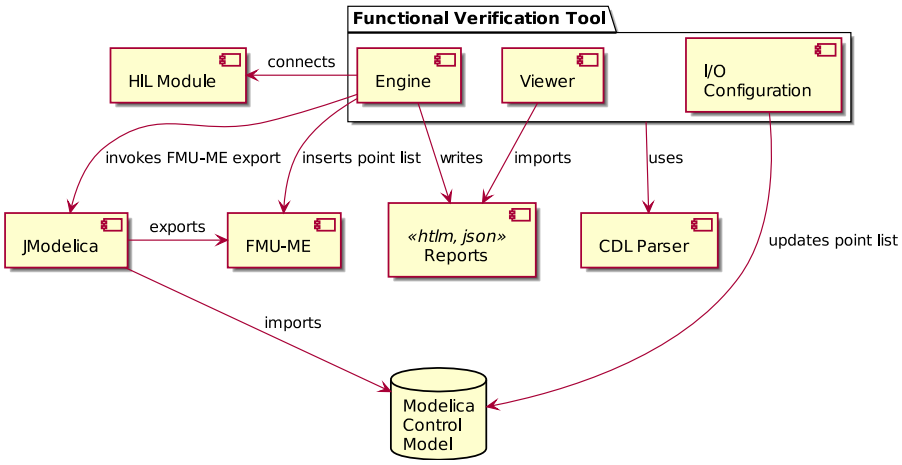


Fig. 6.3: Overall software architecture of the Functional Verification Tool.

The *Functional Verification Tool* consists of three modules:

- An *I/O Configuration* module that adds I/O information to the point list,
- a *Engine* that is used to conduct the actual verification, and
- a *Viewer* that displays the results of the verification.

The *Functional Verification Tool* uses that same *CDL Parser* as is used for the *Controls Design Tool*. The *I/O Configuration* module will allow users (such as a commissioning agent) to update the point list. This is needed as not all point mappings may be known during the design phase. The *Engine* invokes *JModelica* to export an FMU-ME of the control blocks. As *JModelica* does not parse CDL information that is stored in vendor annotations (such as the point mapping), the *Engine* will insert point lists into the `Resources` directory of the *FMU-ME*. To conduct the verification, the *Engine* will connect to a *HIL Module*, such as Volttron or the BCVTB, and set up a closed loop model, using the point list from the FMU's `Resources` directory. During the verification, the *Engine* will write reports that are displayed by the *Viewer*.

Chapter 7

Control Description Language

This section specifies the Control Description Language (CDL).

The CDL consists of the following elements:

- A list of elementary control blocks, such as a block that adds two signals and outputs the sum, or a block that represents a PID controller.
- Connectors through which these blocks receive values and output values.
- Permissible data types.
- Syntax to specify
  - how to instantiate these blocks and assign values of parameters, such as a proportional gain.
  - how to connect inputs of blocks to outputs of other blocks.
  - how to document blocks.
  - how to add annotations such as for graphical rendering of blocks and their connections.
  - how to specify composite blocks.
- A model of computation that describes when blocks are executed and when outputs are assigned to inputs.

The next sections explain the elements of CDL.

7.1 Syntax

In order to use CDL with building energy simulation programs, and to not invent yet another language with new syntax, we use a subset of the Modelica 3.3 specification for the implementation of CDL [Mod12]. The selected subset is needed to instantiate classes, assign parameters, connect objects and document classes. This subset is fully compatible with Modelica, e.g., no other information that violates the Modelica Standard has been added, thereby allowing users to view, modify and simulate CDL-conformant control sequences with any Modelica-compliant simulation environment.

To simplify the support of CDL for tools and control systems, the following Modelica keywords are not supported in CDL:

1. `extends`
2. `redeclare`
3. `constrainedby`
4. `inner` and `outer`

Also, the following Modelica language features are not supported in CDL:

1. Clocks [which are used in Modelica for hybrid system modeling].
2. `algorithm` sections. [As the elementary building blocks are black-box models as far as CDL is concerned and thus CDL compliant tools need not parse the `algorithm` section.]
3. `initial equation` and `initial algorithm` sections.

7.2 Permissible Data Types

The basic data types are, in addition to the elementary building blocks, parameters of type `Real`, `Integer`, `Boolean`, `String`, and `enumeration`. [Parameters do not change their value as time progress.] See also the Modelica 3.3 specification, Chapter 3. All specifications in CDL shall be declaration of blocks, instances of blocks, or declarations of type `parameter`, `constant`, or `enumeration`. Variables are not allowed. [Variables are used in the elementary building blocks, but these can only be used as inputs to other blocks if they are declared as an output.]

The declaration of such types is identical to the declaration in Modelica. [The keyword `parameter` is used before the type declaration, and typically a graphical user interface allows users to change the value of a parameter when the simulation or control sequence is not running. For example, to declare a real-valued parameter, use `parameter Real k = 1 "A parameter with value 1";`. In contrast, a `constant` cannot be changed after the software is compiled, and is typically not shown in a graphical user interface menu. For example, a `constant` is used to define latent heat of evaporation if used in a controller.]

Each of these data types, including the elementary building blocks, can be a single instance or one-dimensional array. Array indices shall be of type `Integer` only. The first element of an array has index `1`. An array of size `0` is an empty array. See the Modelica 3.3 specification Chapter 10 for array notation.

[`enumeration` or `Boolean` data types are not permitted as array indices.]

Note: We still need to define the allowed values for `quantity`, for example `ThermodynamicTemperature` rather than `Temp`.

7.3 Encapsulation of Functionality

All computations are encapsulated in a `block`. Blocks expose parameters (used to configure the block, such as a control gain), and they expose inputs and outputs using *connectors*.

Blocks are either *elementary building blocks* (Section 7.4) or *composite blocks* (Section 7.10).

7.4 Elementary Building Blocks

The CDL library contains elementary building blocks that are used to compose control sequences. The functionality of elementary building blocks, but not their implementation, is part of the CDL specification. Thus, in the most general form,

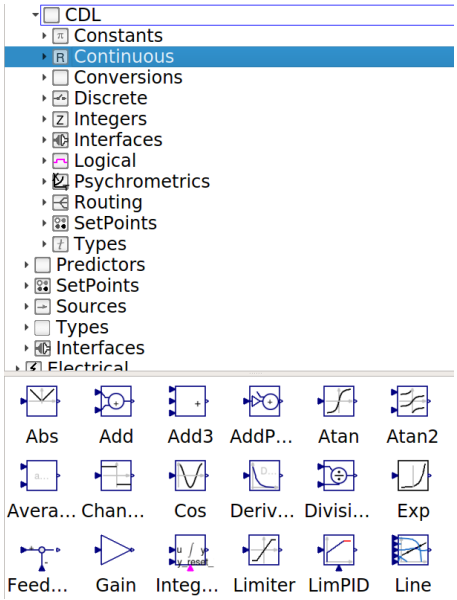


Fig. 7.1: Screenshot of CDL library.

elementary building blocks can be considered as functions that for given parameters p, time t and internal state $x(t)$, map inputs $u(t)$ to new values for the outputs $y(t)$ and states $x'(t)$, e.g.,

$$(p, t, u(t), x(t)) \mapsto (y(t), x'(t)).$$

Control providers who support CDL need to be able to implement the same functionality as is provided by the elementary CDL blocks.

[CDL implementations are allowed to use a different implementation of the elementary building blocks, because the implementation is language specific. However, implementations shall have the same inputs, outputs and parameters, and they shall compute the same response for the same value of inputs and state variables.]

Users are not allowed to add new elementary building blocks. Rather, users can use them to implement composite blocks (Section 7.10).

Note: The elementary building blocks can be browsed in any of these ways:

- Open a web browser at http://simulationresearch.lbl.gov/modelica/releases/latest/help/Buildings_Controls_OBC_CDL.html.
- Download https://github.com/lbl-srg/modelica-buildings/archive/master.zip, unzip the file, and open `Buildings/package.mo` in the graphical model editor of Dymola or OpenModelica. All models in the *Examples* and *Validation* packages can be simulated with these tools. They can also be simulated with JModelica.

An actual implementation of an elementary building block looks as follows, where we omitted the annotations that are used for graphical rendering:

```
block AddParameter "Output the sum of an input plus a parameter"

  parameter Real p "Value to be added";

  parameter Real k "Gain of input";

  Interfaces.RealInput u "Connector of Real input signal";

  Interfaces.RealOutput y "Connector of Real output signal";

equation
  y = k*u + p;

  annotation(Documentation(info("
  <html>
  <p>
  Block that outputs <code>y = k u + p</code>,
  where <code>k</code> and <code>p</code> are
  parameters and <code>u</code> is an input.
  </p>
  </html>"));
end AddParameter;
```

For the complete implementation, see the github repository.

7.5 Instantiation

Instantiation is identical to Modelica.

[For example, to instantiate a gain, one would write

```
Continuous.Gain myGain(k=-1) "Constant gain of -1" annotation(...);
```

where the documentation string is optional. The annotations is typically used for the graphical positioning of the instance in a block-diagram.]

In the assignment of `parameters`, calculations are allowed.

[For example, a hysteresis block could be configured as follows

```
parameter Real pRel(unit="Pa") = 50 "Pressure difference across damper";

CDL.Logical.Hysteresis hys(
  uLow  = pRel-25,
  uHigh = pRel+25) "Hysteresis for fan control";
```

]

Instances can conditionally be removed by using an `if` clause.

[This allows to have one implementation for an economizer control sequence that can be configured to take into account enthalpy rather than temperature. An example code snippet is

```
parameter Boolean use_enthalpy = true
  "Set to true to evaluate outdoor air enthalpy in addition to temperature";

CDL.Interfaces.RealInput hOut if use_enthalpy
   "Outdoor air enthalpy";
```

By the Modelica language definition, all connections (Section 7.8) to `hOut` will be removed if `use_enthalpy = false`.]

7.6 Connectors

Blocks expose their inputs and outputs through input and output connectors.

The permissible connectors are implemented in the package `CDL.Interfaces`, and are `BooleanInput`, `BooleanOutput`, `DayTypeInput`, `DayTypeOutput`, `IntegerInput`, `IntegerOutput`, `RealInput` and `RealOutput`. `DayType` is an `enumeration` for working day, non-working day and holiday.

Connectors can only carry scalar variables. For arrays, the connectors need to be explicitly declared as an array.

[For example, to declare an array of `nin` input signals, use

```
parameter Integer nin(min=1) "Number of inputs";

Interfaces.RealInput u[nin] "Connector for 2 Real input signals";
```

Hence, unlike in Modelica 3.3, we do not allow for automatic vectorization of input signals.]

7.7 Equations

After the instantiations (Section 7.5), a keyword `equation` must be present to introduce the equation section. The equation section can only contain connections (Section 7.8) and annotations (Section 7.9).

Unlike in Modelica, an `equation` section shall not contain equations such as `y=2*u;` or commands such as `for`, `if`, `while` and `when`.

Furthermore, unlike in Modelica, there shall not be an `initial equation`, `initial algorithm` or `algorithm` section. (They can however be part of a elementary building block.)

7.8 Connections

Connections connect input to output connector (Section 7.6). For scalar connectors, each input connector of a block needs to be connected to exactly one output connector of a block. For vectorized connectors, each (element of an) input connector needs to be connected to exactly one (element of an) output connector. Vectorized input connectors can be

connected to vectorized output connectors using one connection statement provided that they have the same number of elements.

Connections are listed after the instantiation of the blocks in an `equation` section. The syntax is

```
connect(port_a, port_b) annotation(...);
```

where `annotation(...)` is used to declare the graphical rendering of the connection (see Section 7.9). The order of the connections and the order of the arguments in the `connect` statement does not matter.

[For example, to connect an input `u` of an instance `gain` to the output `y` of an instance `maxValue`, one would declare

```
Continuous.Max maxValue "Output maximum value";
Continous.Gain gain(k=60) "Gain";

equation
  connect(gain.u, maxValue.y);
```

]

Signals shall be connected using a `connect` statement; assigning the value of a signal in the instantiation of the output connnector is not allowed.

[This ensures that all control sequences are expressed as block diagrams. For example, the following model is valid

```
block MyAdderValid
  Interfaces.RealInput u1;
  RealInput u2;
  Interfaces.RealOutput y;
  Continuous.Add add;
equation
  connect(add.u1, u1);
  connect(add.u2, u2);
  connect(add.y,  y);
end MyAdderValid;
```

whereas the following implementation is not valid in CDL, although it is valid in Modelica

```
block MyAdderInvalid
  Interfaces.RealInput u1;
  Interfaces.RealInput u2;
  Interfaces.RealOutput y = u1 + u2; // not allowed
end MyAdderInvalid;
```

]

7.9 Annotations

Annotations follow the same rules as described in the following Modelica 3.3 Specifications

- 18.2 Annotations for Documentation

- 18.6 Annotations for Graphical Objects, with the exception of
  - 18.6.7 User input
- 18.8 Annotations for Version Handling

[For CDL, annotations are primarily used to graphically visualize block layouts, graphically visualize input and output signal connections, and to declare vendor annotations (Sec. 18.1 in Modelica 3.3 Specification).]

7.10 Composite Blocks

CDL allows building composite blocks such as shown in Fig. 7.2.

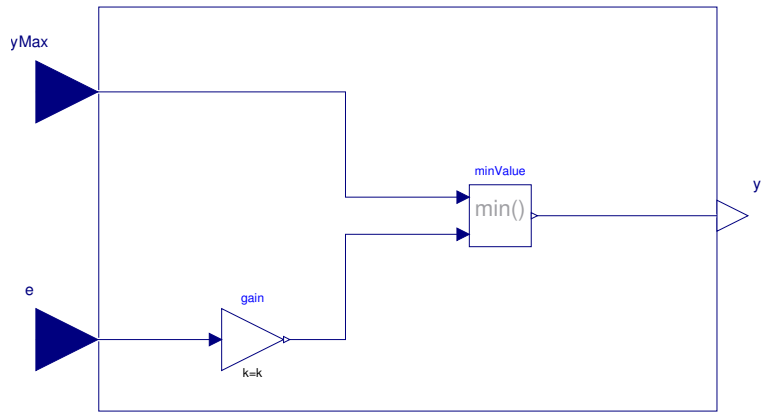


Fig. 7.2: Example of a composite control block that outputs $y = \min(k\, e, y_{max})$ where k is a parameter.

Composite blocks can contain other composite blocks.

Each composite block shall be stored on the file system under the name of the composite block with the file extension `.mo`, and with each package name being a directory. The name shall be an allowed Modelica class name.

[For example, if a user specifies a new composite block `MyController.MyAdder`, then it shall be stored in the file `MyController/MyAdder.mo` on Linux or OS X, or `MyController\\MyAdder.mo` on Windows.]

[The following statement, when saved as `CustomPWithLimiter.mo`, is the declaration of the composite block shown in Fig. 7.2

```
block CustomPWithLimiter
  "Custom implementation of a P controller with variable output limiter"

  parameter Real k "Constant gain";

  CDL.Interfaces.RealInput yMax "Maximum value of output signal"
    annotation (Placement(transformation(extent={{-140,20},{-100,60}})));

  CDL.Interfaces.RealInput e "Control error"
    annotation (Placement(transformation(extent={{-140,-60},{-100,-20}})));

```

(continues on next page)

(continued from previous page)

```
  CDL.Interfaces.RealOutput y "Control signal"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));

  CDL.Continuous.Gain gain(final k=k) "Constant gain"
    annotation (Placement(transformation(extent={{-60,-50},{-40,-30}})));

  CDL.Continuous.Min minValue "Outputs the minimum of its inputs"
    annotation (Placement(transformation(extent={{20,-10},{40,10}})));
equation
  connect(yMax, minValue.u1) annotation (
    Line(points={{-120,40},{-120,40},{-20,40},{-20, 6},{18,6}},
         color={0,0,127}));
  connect(e, gain.u) annotation (
    Line(points={{-120,-40},{-92,-40},{-62,-40}},
         color={0,0,127}));
  connect(gain.y, minValue.u2) annotation (
    Line(points={{-39,-40},{-20,-40},{-20,-6}, {18,-6}},
         color={0,0,127}));
  connect(minValue.y, y) annotation (
     Line(points={{41,0},{110,0}},
          color={0,0,127}));

  annotation (Documentation(info="<html>
<p>
Block that outputs <code>y = min(yMax, k*e)</code>,
where
<code>yMax</code> and <code>e</code> are real-valued input signals and
<code>k</code> is a parameter.
</p>
</html>"));
end CustomPWithLimiter;
```

Composite blocks are needed to preserve grouping of control blocks and their connections, and are needed for hierarchical composition of control sequences.]

7.11 Model of Computation

CDL uses the synchronous data flow principle and the single assignment rule, which are defined below. [The definition is adopted from and consistent with the Modelica 3.3 Specification, Section 8.4.]

1. All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant.
2. Computation and communication at an event instant does not take time. [If computation or communication time has to be simulated, this property has to be explicitly modeled.]
3. Every input connector shall be connected to exactly one output connector.

In addition, the dependency graph from inputs to outputs that directly depend on inputs shall be directed and acyclic. I.e., connections that form an algebraic loop are not allowed. [To break an algebraic loop, one could place a delay block or an

integrator in the loop, because the outputs of a delay or integrator does *not* depend directly on the input.]

7.12 Tags

CDL has sufficient information for tools that process CDL to generate for example point lists that list all analog temperature sensors, or to verify that a pressure control signal is not connected to a temperature input of a controller. Some, but not all, of this information can be inferred from the CDL language described above. We will use tags, implemented through Modelica vendor annotations, to provide this additional information. In Section 7.12.1, we will explain the properties that can be inferred, and in Section 7.12.2, we will explain how to use tagging schemes in CDL.

Note: None of this information affects the computation of a control signal. Rather, it can be used for example to facilitate the implementation of cost estimation tools, or to detect incorrect connections between outputs and inputs.

7.12.1 Inferred Properties

To avoid that signals with physically incompatible quantities are connected, tools that parse CDL can infer the physical quantities from the `unit` and `quantity` attributes.

[For example, a differential pressure input signal with name `u` can be declared as

```
Interfaces.RealInput u(
  quantity="PressureDifference",
  unit="Pa") "Differential pressure signal" annotation (...);
```

Hence, tools can verify that the `PressureDifference` is not connected to `AbsolutePressure`, and they can infer that the input has units of Pascal.

Therefore, tools that process CDL can infer the following information:

- Numerical value: *Binary value* (which in CDL is represented by a `Boolean` data type), *analog value*, (which in CDL is represented by a `Real` data type) *mode* (which in CDL is presented by an `Integer` data type or an enumeration, which allow for example encoding of the ASHRAE Guideline 36 Freeze Protection which has 4 stages).
- Source: Hardware point or software point.
- Quantity: such as Temperature, Pressure, Humidity or Speed.
- Unit: Unit and preferred display unit. (The display unit can be overwritten by a tool. This allows for example a control vendor to use the same sequences in North America displaying IP units, and in the rest of the world displaying SI units.)

]

7.12.2 Tagged Properties

The buildings industry uses different tagging schemes such as Brick (http://brickschema.org/) and Haystack (http://project-haystack.org/). CDL allows, but does not require, use of the Brick or Haystack tagging scheme.

CDL allows to add tags to declarations that instantiate

- elementary building blocks (Section 7.4), and
- composite blocks (Section 7.10).

[We currently do not see a use case that would require adding a tag to a `parameter` declaration.]

To implement such tags, CDL blocks can contain vendor annotations with the following syntax:

```
annotation :
  annotation "(" [annotations ","]
     __cdl "(" [ __cdl_annotation ] ")" ["," annotations] ")"
```

where `__cdl__annotation` is the annotation for CDL.

For Brick, the `__cdl_annotation` is

```
brick_annotation:
   brick "(" RDF ")"
```

where `RDF` is the RDF 1.1 Turtle (https://www.w3.org/TR/turtle/) specification of the Brick object.

[Note that, for example for a controller with two output signals y_1 and y_2, Brick seems to have no means to specify that y_1 `controls` a fan speed and y_2 `controls` a heating valve, where `controls` is the Brick relationship. Therefore, we allow the `brick_annotation` to only be at the block level, but not at the level of instances of input or output connectors.

For example, the Brick specification

```
soda_hall:flow_sensor_SODA1F1_VAV_AV a brick:Supply_Air_Flow_Sensor ;
   bf:hasTag brick:Average ;
   bf:isLocatedIn soda_hall:floor_1 .
```

can be declared in CDL as

```
annotation(__cdl(brick="soda_hall:flow_sensor_SODA1F1_VAV_AV a brick:Supply_Air_Flow_Sensor ␣
↪;
  bf:hasTag brick:Average ;
  bf:isLocatedIn soda_hall:floor_1 ."));
```

]

For Haystack, the `__cdl_annotation` is

```
haystack_annotation:
   haystack "(" JSON ")"
```

where `JSON` is the JSON encoding of the Haystack object.

[For example, the AHU discharge air temperature setpoint of the example in http://project-haystack.org/tag/sensor, which is in Haystack declared as

```
id: @whitehouse.ahu3.dat
dis: "White House AHU-3 DischargeAirTemp"
point
```

(continues on next page)

(continued from previous page)

```
siteRef: @whitehouse
equipRef: @whitehouse.ahu3
discharge
air
temp
sensor
kind: "Number"
unit: "degF"
```

can be declared in CDL as

```
annotation(__cdl( haystack=
  "{\"id\"   : \"@whitehouse.ahu3.dat\",
    \"dis\" : \"White House AHU-3 DischargeAirTemp\",
    \"point\" : \"m:\",
    \"siteRef\" : \"@whitehouse\",
    \"equipRef\" : \"@whitehouse.ahu3\",
    \"discharge\" : \"m:\",
    \"air\"       : \"m:\",
    \"temp\"      : \"m:\",
    \"sensor\"    : \"m:\",
    \"kind\"       : \"Number\"
    \"unit\"       : \"degF\"}"));
```

Tools that process CDL can interpret the `brick` or `haystack` annotation, but for control purposes CDL will ignore it. [This avoids potential conflict for entities that are declared differently in Brick (or Haystack) and CDL, and may be conflicting. For example, the above sensor input declares in Haystack that it belongs to an ahu3. CDL, however, has a different syntax to declare such dependencies: In CDL, through the `connect(whitehouse.ahu3.TSup, ...)` statement, a tool can infer what upstream component sends the input signal.]

Chapter 8

Code Generation

This section describes the development of a proof-of-concept translator from CDL to a building automation system. Translating the *CDL library* to a building automation system needs to be done only when the CDL library is updated, and hence only developers need to perform this step. However, translation of a *CDL-conforming control sequence*, as well as translation of verification tests, will need to be done for each building project.

8.1 Challenges and Implications for Translation of Control Sequences

This section discusses challenges and implications for translating CDL-conforming control sequences to executable code on a building automation system.

First, we note that the translation will for most, if not all, systems only be possible from CDL to a building automation system, but not vice versa. This is due to specific constructs that may exist in building automation systems but not in CDL. For example, if Sedona were the target platform, then translating from Sedona to CDL will not be possible because Sedona allows boolean variables to take on the values `true`, `false` and `null`, but CDL has no `null` value.

Second, we note that most building automation product lines are based on old computing technologies. One may argue that to meet future process requirements and user-friendliness, these may be updated in the near future. Relatively recent or new product lines include

- Tridium Niagara, or its open version Sedona (http://www.sedonadev.org/),
- Distech control (http://www.distech-controls.com/en/us/), and
- Schneider Electric's EcoStruxture (https://www.schneider-electric.us/en/work/campaign/innovation/overview.jsp).

While Sedona has been designed for 3rd party developers to add new functionality, the others seem rather closed. For example, detailed developer documentation that describes the following is difficult to find, or may not exist:

- the language specification for implementation of block diagrams,
- the model of computation, and
- how to simulate open loop control responses and implement regression testing,

Sedona "is designed to make it easy to build smart, networked embedded devices" and Sedona attempts to create an "Open Source Ecosystem" (http://www.sedonadev.org/). Block diagrams can be developed with the free Sedona Application Editor (https://www.ccontrols.com/basautomation/sae.htm).

8.2 Use of Control Sequences or Verification Tests in Realtime Applications

Use of control sequences or verification tests in realtime applications, such as in a building automation system or in a verification test module, consists of the following steps:

1. Implementation of the control sequence or verification test as a Modelica model.
2. Export of the Modelica model as a *Functional Mockup Unit* for Model Exchange (FMU-ME) or as a JSON specification.
3. Import of the FMU-ME in the runtime environment, or translation of the JSON specification to the language used by the building automation system.

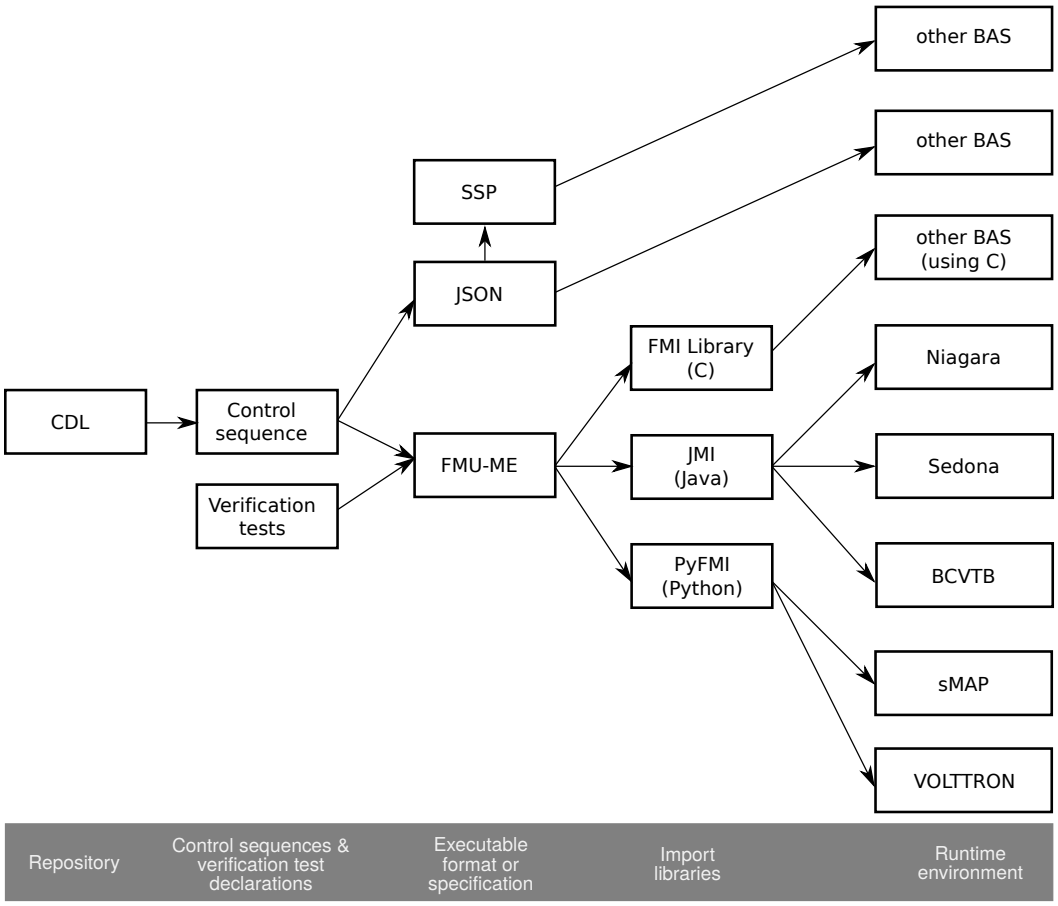


Fig. 8.1: Overview of the code export and import of control sequences and verification tests.

Fig. 8.1 shows the process of exporting and importing control sequences or verification tests.

We will now describe three different approaches that can be used by control vendors to translate CDL to their product line:

1. Export of the whole CDL-compliant sequence to one FMU (Section 8.2.1),
2. Translation of the CDL-compliant sequence to a JSON intermediate format, which can be translated to the format used by the control platform (Section 8.2.2), and
3. Translation of the CDL-compliant sequence to an xml-based standard called System Structure and Parameterization (SSP), which is then used to parameterize, link and execute pre-compiled elementary CDL blocks (Section 8.2.3).

The best approach will depend on the control platform.

8.2.1 Export of a Control Sequence or a Verification Test using the FMI Standard

This section describes how to export a control sequence, or a verification test, using the *FMI standard*. In this workflow, the intermediate format that is used is FMI for model exchange, as it is an open standard, and because FMI can easily be integrated into tools for controls or verification using a variety of languages.

Note: Also possible, but outside of the scope of this project, is the translation of the control sequences to JavaScript, which could then be executed in a building automation system. For a Modelica to JavasScript converter, see https://github.com/tshort/openmodelica-javascript.

To implement control sequences, blocks from the CDL library (Section 7.4) can be used to compose sequences that conform to the CDL language specification described in Section 7. For verification tests, any Modelica block can be used. Next, to export the Modelica model, a Modelica tool such as JModelica, OpenModelica or Dymola can be used. For example, with JModelica a control sequence can be exported using the Python commands

```
from pymodelica import compile_fmu
compile_fmu("Buildings.Controls.OBC.ASHRAE.G36_PR1.AHUs.SingleZone.Economizers.Controller")
```

This will generate an FMU-ME. Finally, to import the FMU-ME in a runtime environment, various tools can be used, including:

- Tools based on Python, which could be used to interface with sMAP (https://pythonhosted.org/Smap/en/2.0/index.html) or Volttron (https://energy.gov/eere/buildings/volttron):
  - PyFMI (https://pypi.python.org/pypi/PyFMI)
- Tools based on Java:
  - Building Controls Virtual Test Bed (http://simulationresearch.lbl.gov/bcvtb)
  - JFMI (https://ptolemy.eecs.berkeley.edu/java/jfmi/)
  - JavaFMI (https://bitbucket.org/siani/javafmi/wiki/Home)
- Tools based on C:
  - FMI Library (http://www.jmodelica.org/FMILibrary)
- Modelica tools, of which many if not all provide functionality for real-time simulation:
  - JModelica (http://www.jmodelica.org)
  - OpenModelica (https://openmodelica.org/)
  - Dymola (https://www.3ds.com/products-services/catia/products/dymola/)
  - MapleSim (https://www.maplesoft.com/products/maplesim/)
  - SimulationX (https://www.simulationx.com/)
  - SystemModeler (http://www.wolfram.com/system-modeler/index.html)

See also http://fmi-standard.org/tools/ for other tools.

Note that directly compiling Modelica models to building automation systems also allows leveraging the ongoing EMPHYSIS project (2017-20, Euro 14M) that develops technologies for running dynamic models on electronic control units (ECU), micro controllers or other embedded systems. This may be attractive for FDD and some advanced control sequences.

8.2.2 Translation of a Control Sequence using a JSON Intermediate Format

Control companies that choose to not use C-code generation or the FMI standard to execute CDL-compliant control sequences can develop translators from CDL to their native language. To aid in this process, a CDL to JSON translator can be used. Such a translator is currently being developed at https://github.com/lbl-srg/modelica-json. This translator parses CDL-compliant control sequences to a JSON format. The parser generates the following output formats:

1. A JSON representation of the control sequence,
2. a simplified version of this JSON representation, and
3. an html-formated documentation of the control sequence.

To translate CDL-compliant control sequences to the language that is used by the respective building automation system, the simplified JSON representation is most suited.

As an illustrative example, consider the composite control block shown in Fig. 8.2.

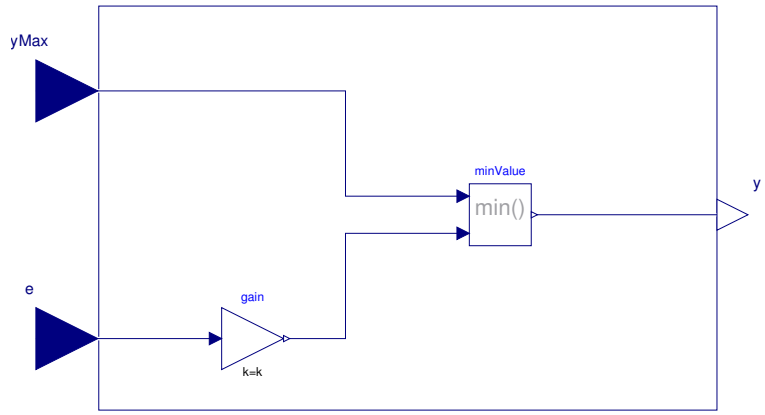


Fig. 8.2: Example of a composite control block that outputs $y = \max(k\,e, y_{max})$ where k is a parameter.

In CDL, this would be specified as

```
block CustomPWithLimiter
  "Custom implementation of a P controller with variable output limiter"
  parameter Real k "Constant gain";
  CDL.Interfaces.RealInput yMax "Maximum value of output signal"
    annotation (Placement(transformation(extent={{-140,20},{-100,60}})));
  CDL.Interfaces.RealInput e "Control error"
    annotation (Placement(transformation(extent={{-140,-60},{-100,-20}})));
  CDL.Interfaces.RealOutput y "Control signal"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
  CDL.Continuous.Gain gain(final k=k) "Constant gain"
    annotation (Placement(transformation(extent={{-60,-50},{-40,-30}})));
  CDL.Continuous.Min minValue "Outputs the minimum of its inputs"
    annotation (Placement(transformation(extent={{20,-10},{40,10}})));
equation
  connect(yMax, minValue.u1) annotation (
    Line(points={{-120,40},{-120,40},{-20,40},{-20, 6},{18,6}}, color={0,0,127}));
  connect(e, gain.u) annotation (
```

(continues on next page)

(continued from previous page)

```
    Line(points={{-120,-40},{-92,-40},{-62,-40}}, color={0,0,127}));
  connect(gain.y, minValue.u2) annotation (
    Line(points={{-39,-40},{-20,-40},{-20,-6}, {18,-6}}, color={0,0,127}));
  connect(minValue.y, y) annotation (
     Line(points={{41,0},{110,0}}, color={0,0,127}));
  annotation (Documentation(info="<html>
<p>
Block that outputs <code>y = min(yMax, k*e)</code>,
where
<code>yMax</code> and <code>e</code> are real-valued input signals and
<code>k</code> is a parameter.
</p>
</html>"));
end CustomPWithLimiter;
```

This specification can be converted to JSON using the program modelica-json. Executing the command

```
node modelica-json/app.js -f CustomPWithLimiter.mo -o json-simplified
```

will produce a file called `CustomPWithLimiter-simplified.json` that looks as follows:

```
[
  {
    "modelicaFile": "CustomPWithLimiter.mo",
    "topClassName": "CustomPWithLimiter",
    "comment": "Custom implementation of a P controller with variable output limiter",
    "public": {
      "parameters": [
        {
          "className": "Real",
          "name": "k",
          "comment": "Constant gain",
          "annotation": {
            "dialog": {
              "tab": "General",
              "group": "Parameters"
            }
          }
        }
      ],
      "models": [
        {
          "className": "CDL.Interfaces.RealInput",
          "name": "yMax",
          "comment": "Maximum value of output signal"
        },
        {
          "className": "CDL.Interfaces.RealInput",
          "name": "e",
```

(continues on next page)

(continued from previous page)

```
          "comment": "Control error"
        },
        {
          "className": "CDL.Interfaces.RealOutput",
          "name": "y",
          "comment": "Control signal"
        },
        {
          "className": "CDL.Continuous.Gain",
          "name": "gain",
          "comment": "Constant gain",
          "modifications": [
            {
              "name": "k",
              "value": "k",
              "isFinal": true
            }
          ]
        },
        {
          "className": "CDL.Continuous.Min",
          "name": "minValue",
          "comment": "Outputs the minimum of its inputs"
        }
      ]
    },
    "info": "<html>\n<p>\nBlock that outputs <code>y = min(yMax, k*e)</code>,\nwhere\n<code>
↪yMax</code> and <code>e</code> are real-valued input signals and\n<code>k</code> is a␣
↪parameter.\n</p>\n</html>",
    "connections": [
      [
        {
          "instance": "yMax"
        },
        {
          "instance": "minValue",
          "connector": "u1"
        }
      ],
      [
        {
          "instance": "e"
        },
        {
          "instance": "gain",
          "connector": "u"
        }
      ],
      [
```

(continues on next page)

(continued from previous page)

```
          {
            "instance": "gain",
            "connector": "y"
          },
          {
            "instance": "minValue",
            "connector": "u2"
          }
        ],
        [
          {
            "instance": "minValue",
            "connector": "y"
          },
          {
            "instance": "y"
          }
        ]
      ]
    }
]
```

Note that the graphical annotations are not shown. The JSON representation can then be parsed and converted to another block-diagram language. Note that `CDL.Continuous.Gain` is an elementary CDL block (see Section 7.4). If it were a composite CDL block (see Section 7.10), it would be parsed recursively until only elementary CDL blocks are present in the JSON file. Various examples of CDL converted to JSON can be found at https://github.com/lbl-srg/modelica-json/tree/master/test/FromModelica.

Todo: Add a JSON-schema definition for the simplified JSON representation, see http://json-schema.org.

8.2.3 Modular Export of a Control Sequence using the FMI Standard for Control Blocks and using the SSP Standard for the Run-time Environment

In early 2018, a new standard called System Structure and Parameterization (SSP) will be released. The standard provides an xml scheme for the specification of FMU parameter values, their input and output connections, and their graphical layout. The SSP standard allows for transporting complex networks of FMUs between different platforms for simulation, hardware-in-the-loop and model-in-the-loop [KohlerHM+16]. Various tools that can simulate systems specified using the SSP standard are in development, with FMI composer (http://www.modelon.com/products/modelon-deployment-suite/fmi-composer/) from Modelon being commercially available.

CDL-compliant control sequences could be exported to the SSP standard as shown in Fig. 8.3.

In such a workflow, a control vendor would translate the elementary CDL blocks (Section 7.4) to a repository of FMU-ME blocks. These blocks will then be used during operation. For each project, its CDL-compliant control sequence could be translated to the simplified JSON format, as described in Section 8.2.2. Using a template engine (similar as is used by `modelica-json` to translate the simplified JSON to html), the simplified JSON representation could then be converted

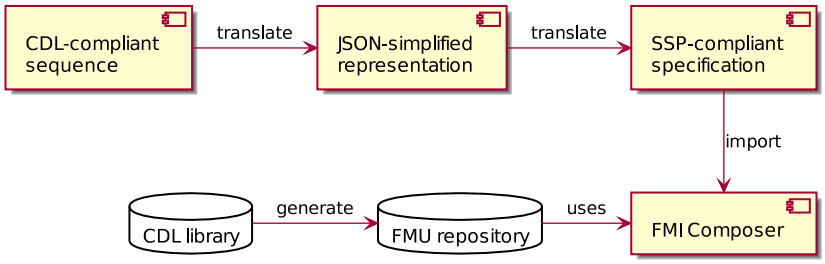


Fig. 8.3: Translation of CDL to SSP.

to the xml syntax specified in the SSP standard. Finally, a tool such as the FMI Composer could import the SSP-compliant specification, and execute the control sequence using the elementary CDL block FMUs from the FMU repository.

Note: In this workflow, all key representations are based on standards: The CDL-specification uses a subset of the Modelica standard, the elementary CDL blocks are converted to the FMI standard, and finally the runtime environment uses the SSP standard.

8.3 Replacement of Elementary CDL Blocks during Translation

When translating CDL to a control product lines, a translator may want to conduct certain substitutions. Some of these substitutions can change the control response, which can cause the verification that checks whether the actual implementation conforms to the specification to fail.

This section therefore explains how certain substitutions can be performed in a way that allows formal verification to pass. (How verification tests will be conducted will be specified later in 2018, but essentially we will require that the control response from the actual control implementation is within a certain tolerance of the control response computed by the CDL specification, provided that both sequences receive the same input signals and use the same parameter values.)

8.3.1 Substitutions that Give Identical Control Response

Consider the gain `CDL.Continuous.Gain` used above. If a product line uses different names for the inputs, outputs and parameters, then they can be replaced.

Moreover, certain transformations that do not change the response of the block are permissible: For example, consider the PID controller in the CDL library. The implementation has a parameter for the time constant of the integrator block. If a control vendor requires the specification of an integrator gain rather than the integrator time constant, then such a parameter transformation can be done during the translation, as both implementations yield an identical response.

8.3.2 Substitutions that Change the Control Response

If a control vendor likes to use for example a different implementation of the anti-windup in a PID controller, then such a substitution will cause the verification to fail if the control responses differ between the CDL-compliant specification and

the vendor's implementation.

Therefore, if a customer requires the implemented control sequence to comply with the specification, then the workflow shall be such that the control provider provides an executable implementation of its controller, and the control provider shall ask the customer to replace in the control specification the PID controller from the CDL library with the PID controller provided by the control provider. Afterwards, verification can be conducted as usual.

Note: Such an executable implementation of a vendor's PID controller can be made available by publishing the controller or by contributing the controller to the Modelica Buildings Library. The implementation of the control logic can be done either using other `CDL` blocks, which is the preferred approach, using the C language, or by providing a compiled library. See the Modelica Specification [Mod12] for implementation details if C code or compiled libraries are provided. If a compiled library is provided, then binaries shall be provided for Windows 32/64 bit, Linux 32/64 bit, and OS X 64 bit.

8.3.3 Adding Blocks that are not in the CDL Library

If a control vendor likes to use a block that is not in the CDL library, such as a block that uses machine learning to schedule optimal warm-up, then such an addition must be approved by the customer. If the customer requires the part of the control sequence that contains this block to be verified, then the block shall be made available as described in Section 8.3.2.

Chapter 9

Verification

This section describes how to formally verify whether the control sequence is implemented according to specification. This process would be done as part of the commissioning, as indicated in step 9 in the process diagram Fig. 3.1. For the requirements, see Section 5.3.

For clarity, we remind that *verification* tests whether the implementation of the control sequence conforms with its specification. In contrast, *validation* would test whether the control sequence, together with the building system, is such that it meets the building owner's need. Hence, validation would be done in step 2 in Fig. 3.1.

As this step only verifies that the control logic is implemented correctly, it should be conducted in addition to other functional tests, such as tests that verify that sensor and actuators are connected to the correct inputs and outputs, that sensors are installed properly and that the installed mechanical system meets the specification.

9.1 Scope of the verification

For OpenBuildingControl, we currently only verify the implementation of the control sequence. Outside the scope of our verification are tests that verify whether the I/O points are connected properly, whether the mechanical equipment is installed and functions correctly, and whether the building envelop is meeting its specification. Therefore, with our tests, we aim to verify that the control provider implemented the sequence as specified, and that it executes correctly.

9.2 Methodology

A typical usage would be as follows: A commissioning agent exports trended control inputs and outputs and stores them in a CSV file. The commissioning agent then executes the CDL specification for the trended inputs, and compares the following:

1. Whether the trended outputs and the outputs computed by the CDL specification are close to each other.
2. Whether the trended inputs and outputs lead to the right sequence diagrams, for example, whether an airhandler's economizer outdoor air damper is fully open when the system is in free cooling mode.

Technically, step 2 is not needed if step 1 succeeds. However, feedback from mechanical designers indicate the desire to confirm during commissioning that the sequence diagrams are indeed correct (and hence the original control specification is correct for the given system).

Fig. 9.1 shows the flow diagram for the verification. Rather than using real-time data through BACnet or other protocols, set points, inputs and outputs of the actual controller are stored in an archive, here a CSV file. This allows to reproduce the verification tests, and it does not require the verification tool to have access to the actual building control system. During the verification, the archived data are read into a Modelica model that conducts the verification. The verification will use three blocks. The block labeled *input file reader* reads the archived data, which may typically be in CSV format. As this data may be directly written by a building automation system, its units will differ from the units used in CDL. Therefore, the block called *unit conversion* converts the data to the units used in the CDL control specification. Next, the block labeled *control specification* is the control sequence specification in CDL format. This is the specification that was exported during design and sent to the control provider. Given the set points and measurement signals, it outputs the control signals according to the specification. The block labeled *time series verification* compares this output with trended control signals, and indicates where the signals differ by more than a prescribed tolerance in time and in signal value. The block labeled *sequence chart* creates x-y or scatter plots. These can be used to verify for example that an economizer outdoor air damper has the expected position as a function of the outside air temperature.

Below, we will further describe the blocks in the box labeled *verification*.

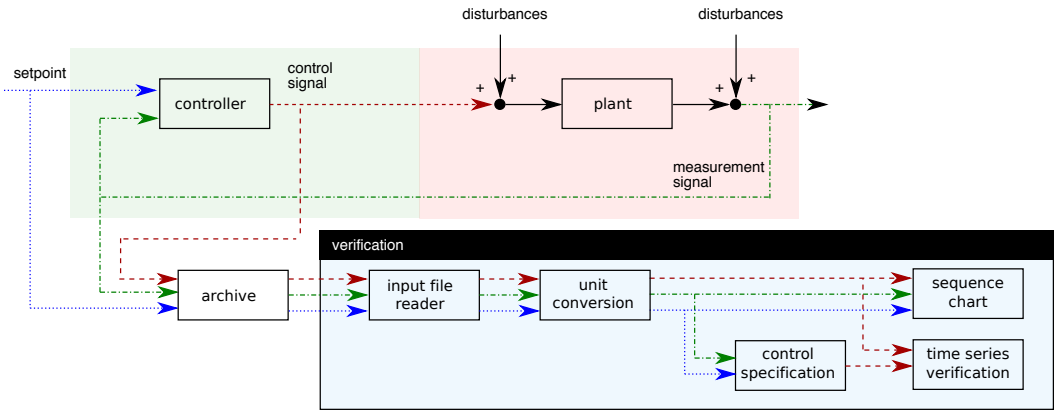


Fig. 9.1: *Overview of the verification that tests whether the installed control sequence meets the specification.*

Note: We also considered testing for criteria such as “whether room temperatures are satisfactory” or “a damper control signal is not oscillating”. However, discussions with design engineers and commissioning providers showed that there is currently no accepted method for turning such questions into hard requirements. We implemented software that tests criteria such as “Room air temperature shall be within the setpoint ± 0.5 Kelvin for at least 45 min within each 60 minute window.” and “Damper signal shall not oscillate more than 4 times per hour between a change of ± 0.025 (for a 2 minute sample period)”. Software implementations of such tests are available on the Modelica Buildings Library github repository, commit 454cc75.

Besides these tests, we also considered automatic fault detection and diagnostics methods that were proposed for inclusion in ASHRAE RP-1455 and Guideline 36, and we considered using methods such as in [Ver13] that automatically detect faulty regulation, including excessively oscillatory behavior. However, as it is not yet clear how sensitive these methods are to site-specific tuning, and because field tests are ongoing in a NIST project, we did not implement them.

9.3 Modules of the verification test

To conduct the verification, the following models and tools are used.

9.3.1 CSV file reader

To read CSV files, the data reader `Modelica.Blocks.Sources.CombiTimeTable` from the Modelica Standard Library can be used. It requires the CSV file to have the following structure:

```
#1
# comment line
double tab1(6,2)
# time in seconds, column 1
  0   0
  1   0
  1   1
  2   4
  3   9
  4  16
```

Note, that the first two characters in the file need to be `#1` (a line comment defining the version number of the file format). Afterwards, the corresponding matrix has to be declared with type `double`, name and dimensions. Finally, in successive rows of the file, the elements of the matrix have to be given. The elements have to be provided as a sequence of numbers in row-wise order (therefore a matrix row can span several lines in the file and need not start at the beginning of a line). Numbers have to be given according to C syntax (such as 2.3, -2, +2.e4). Number separators are spaces, tab, comma, or semicolon. Line comments start with the hash symbol (#) and can appear everywhere.

9.3.2 Unit conversion

Building automation systems store physical quantities in various units. To convert them to the units used by Modelica and hence also by CDL, we developed the package `Buildings.Controls.OBC.UnitConversions`. This package provides blocks that convert between SI units and units that are commonly used in the HVAC industry.

9.3.3 Comparison of time series data

We have been developing a tool called *funnel* to conduct time series comparison. The tool imports two CSV files, one containing the reference data set and the other the test data set. Both CSV files contain time series that need to be compared against each other. The comparison is conducted by computing a funnel around the reference curve. For this funnel, users can specify the tolerances with respect to time and with respect to the trended quantity. The tool then checks whether the time series of the test data set is within the funnel and computes the corresponding exceeding error curve.

The tool is available from https://github.com/lbl-srg/funnel.

It is primarily intended to be used by means of a Python binding:

- either by importing the module `pyfunnel` and using `compareAndReport` and `plot_funnel` functions: see Fig. 9.2 for typical plot output;

- or by running directly the Python script from terminal: see usage information by running `python {path to pyfunnel.py} --help`. This produces the following:

```
usage: pyfunnel.py [-h] --reference REFERENCE --test TEST [--output OUTPUT]
                   [--atolx ATOLX] [--atoly ATOLY] [--rtolx RTOLX]
                   [--rtoly RTOLY]

Run funnel binary from terminal.

Output `errors.csv`, `lowerBound.csv`, `upperBound.csv`, `reference.csv`, `test.csv` into␣
↪the output directory (`./results` by default).

optional arguments:
  -h, --help            show this help message and exit
  --output OUTPUT       Path of directory to store output data
  --atolx ATOLX         Absolute tolerance along x axis
  --atoly ATOLY         Absolute tolerance along y axis
  --rtolx RTOLX         Relative tolerance along x axis
  --rtoly RTOLY         Relative tolerance along y axis

required named arguments:
  --reference REFERENCE
                        Path of CSV file with reference data
  --test TEST           Path of CSV file with test data

Note: At least one of the two possible tolerance parameters (atol or rtol) must be defined␣
↪for each axis.
Relative tolerance is relative to the range of x or y values.

Typical use from terminal:
$ python {path to pyfunnel.py} --reference trended.csv --test simulated.csv --atolx 0.002 --
↪atoly 0.002 --output results

Full documentation at https://github.com/lbl-srg/funnel
```

9.3.4 Verification of sequence diagrams

To verify sequence diagrams we developed the Modelica package `Buildings.Utilities.IO.Plotters`. Fig. 9.3 shows an example in which this block is used to produce the sequence diagram shown in Fig. 9.4. While in this example, we used the control output of the CDL implementation, during commissioning, one would use the control signal from the building automation system. The model is available from the Modelica Buildings Library, see the model `Buildings.Utilities.Plotters.Examples.SingleZoneVAVSupply_u`.

Simulating the model shown in Fig. 9.3 generates an html file that contains the scatter plots shown in Fig. 9.5.

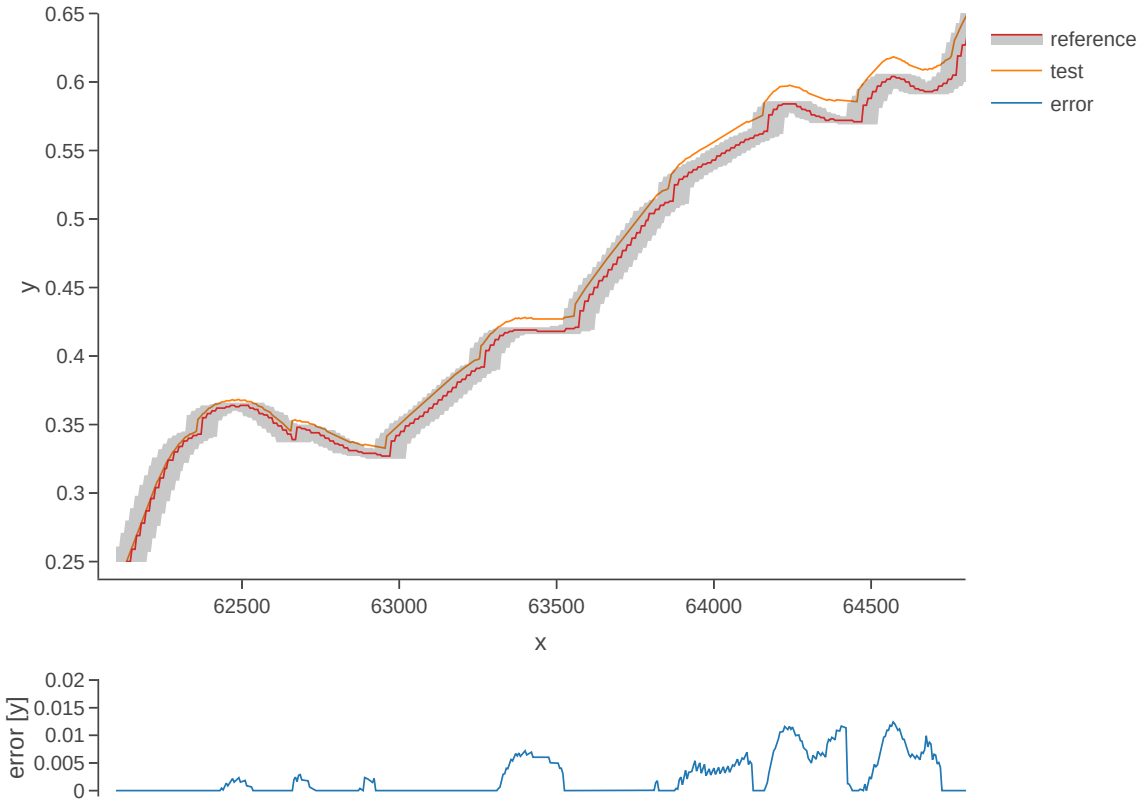


Fig. 9.2: *Typical plot generated by* `pyfunnel.plot_funnel` *for comparing test and reference time series.*

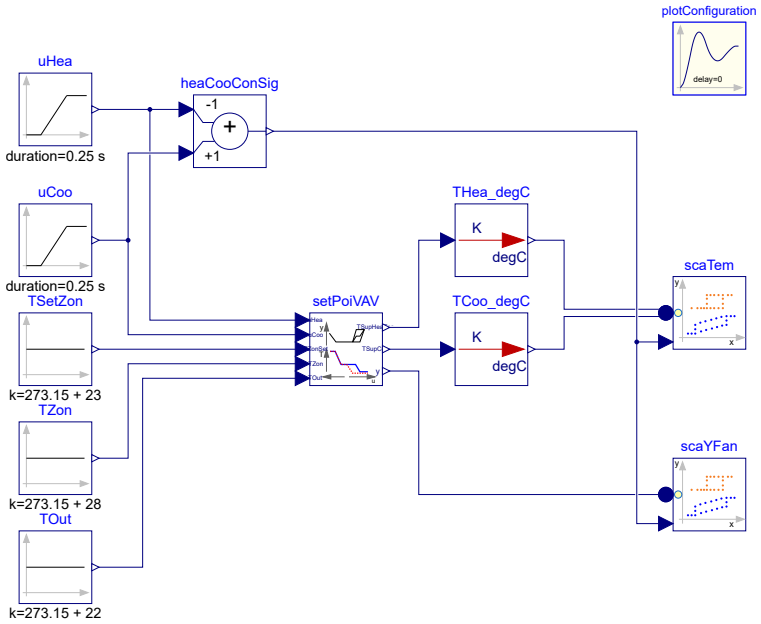


Fig. 9.3: *Modelica model that verifies the sequence diagram. On the left are the blocks that generate the control input. In a real verification, these would be replaced with a file reader that reads data that have been archived by the building automation system. In the center is the control sequence implementation. Some of its output is converted to degree Celsius, and then fed to the plotters on the right that generate a scatter plot for the temperatures and a scatter plot for the fan control signal. The block labeled* `plotConfiguration` *configures the file name for the plots and the sampling interval.*

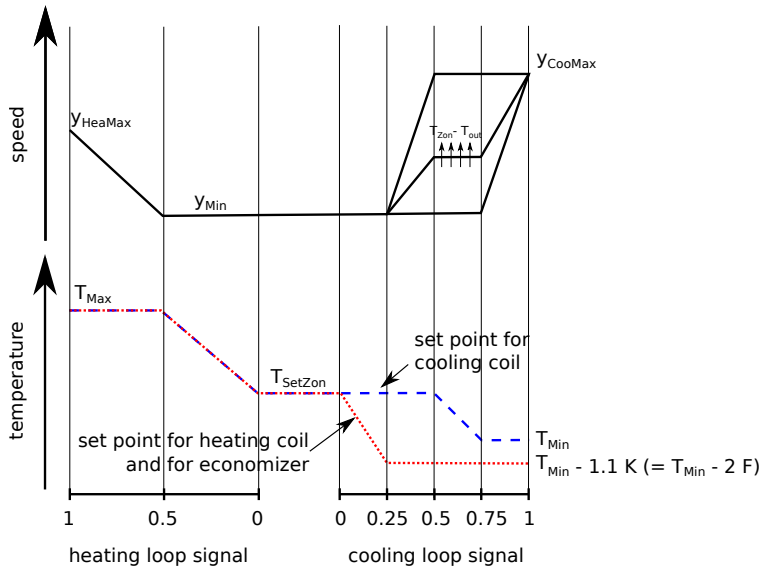


Fig. 9.4: *Control sequence diagram for the VAV single zone control sequence from ASHRAE Guideline 36.*

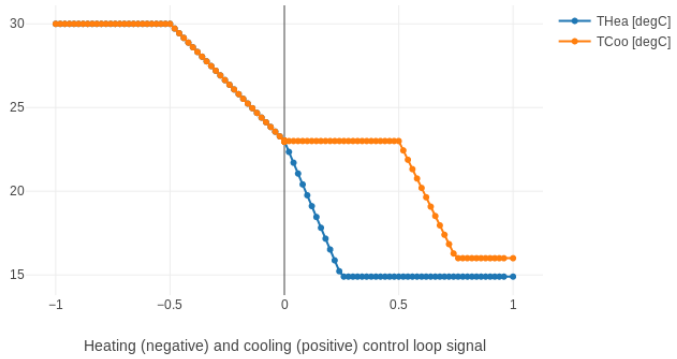


Fig. 9.5: *Scatter plots that show the control sequence diagram generated from the simulated sequence.*

9.4 Example

In this example we validated a trended output of a control sequence that defines the cooling coil valve position. The cooling coil valve sequence is a part of the ALC EIKON control logic implemented in building 33 on the main LBNL campus in Berkeley, CA. The subsequence is shown in Fig. 9.6. It comprises a PI controller that tracks the supply air temperature, an upstream subsequence that enables the controller and a downstream output limiter that is active in case of low supply air temperatures.

We created a CDL specification of the same cooling coil valve position control sequence, see Fig. 9.7, to validate the recorded output. We recorded trend data in 5 second intervals for

- Supply air temperature in [F]
- Supply air temperature setpoint in [F]
- Outdoor air temperature in [F]
- VFD fan enable status in [0/1]
- VFD fan feedback in [%]
- Cooling coil valve position, which is the output of the controller, in [%].

The input and output trends were processed with a script that converts them to the format required by the data readers. The data used in the example begins at midnight on June 7 2018. In addition to the input and output trends, we recorded all parameters, such as the hysteresis offset (see Fig. 9.8) and the controller gains (see Fig. 9.9), to use them in the CDL implementation.

We configured the CDL PID controller parameters such that they correspond to the parameters of the ALC PI controller. The ALC PID controller implementation is described in the ALC EIKON software help section, while the CDL PID controller is described in the info section of the model Buildings.Controls.OBC.CDL.Continuous.LimPID. The ALC controller tracks the temperature in degree Fahrenheit, while CDL uses SI units. An additional implementation difference is that for cooling applications, the ALC controller uses direct control action, whereas the CDL controller needs to be configured to use reverse control action, which can be done by setting its parameter *reverseAction=true*. Furthermore, the ALC controller outputs the control action in percentages, while the CDL controller outputs a signal between 0 and 1. To reconcile the differences, the ALC controller gains were converted for CDL as follows: The proportional gain $k_{p,cdl}$ was set to

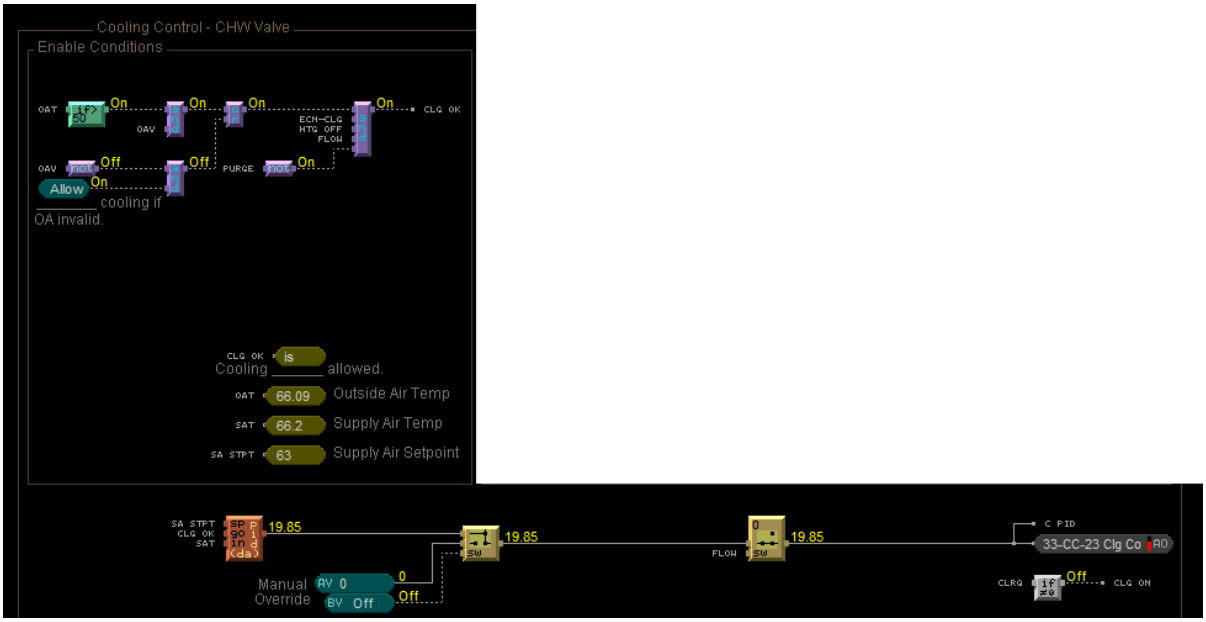


Fig. 9.6: *ALC EIKON specification of the cooling coil valve position control sequence.*

$k_{p,cdl} = u\, k_{p,alc}$, where $u = 9/5$ is a ratio of one degree Celsius (or Kelvin) to one degree Fahrenheit of temperature difference. The integrator time constant was converted as $T_{i,cdl} = k_{p,cdl}\, I_{alc}/(u\, k_{i,alc})$. Both controllers were enabled throughout the whole validation time.

Fig. 9.10 shows the Modelica model that was used to conduct the verification. On the left hand side are the data readers that read the input and output trends from files. Next are unit converters, and a conversion for the fan status between a real value and a boolean value. These data are fed into the instance labeled `cooValSta`, which contains the control sequence as shown in Fig. 9.7. The plotters on the right hand side then compare the simulated cooling coil valve position with the recorded data.

Fig. 9.11, which was produced by the Modelica model using blocks from the `Buildings.Utilities.Plotters` package, shows the trended input temperatures for the control sequence, the trended and simulated cooling valve control signal for the same time period, which are practically on top of each other, and a correlation error between the trended and simulated cooling valve control signal.

The difference in modeled vs. trended results is due to the following factors:

- ALC EIKON uses a discrete time step for the time integration with a user-defined time step length, whereas CDL uses a continuous time integrator that adjusts the time step based on the integration error.
- ALC EIKON uses a proprietary algorithm for the anti-windup, which differs from the one used in the CDL implementation.

Despite these differences, the computed and the simulated control signals show good agreement, which is also demonstrated by verifying the time series with the funnel software, whose output is shown in Fig. 9.12.

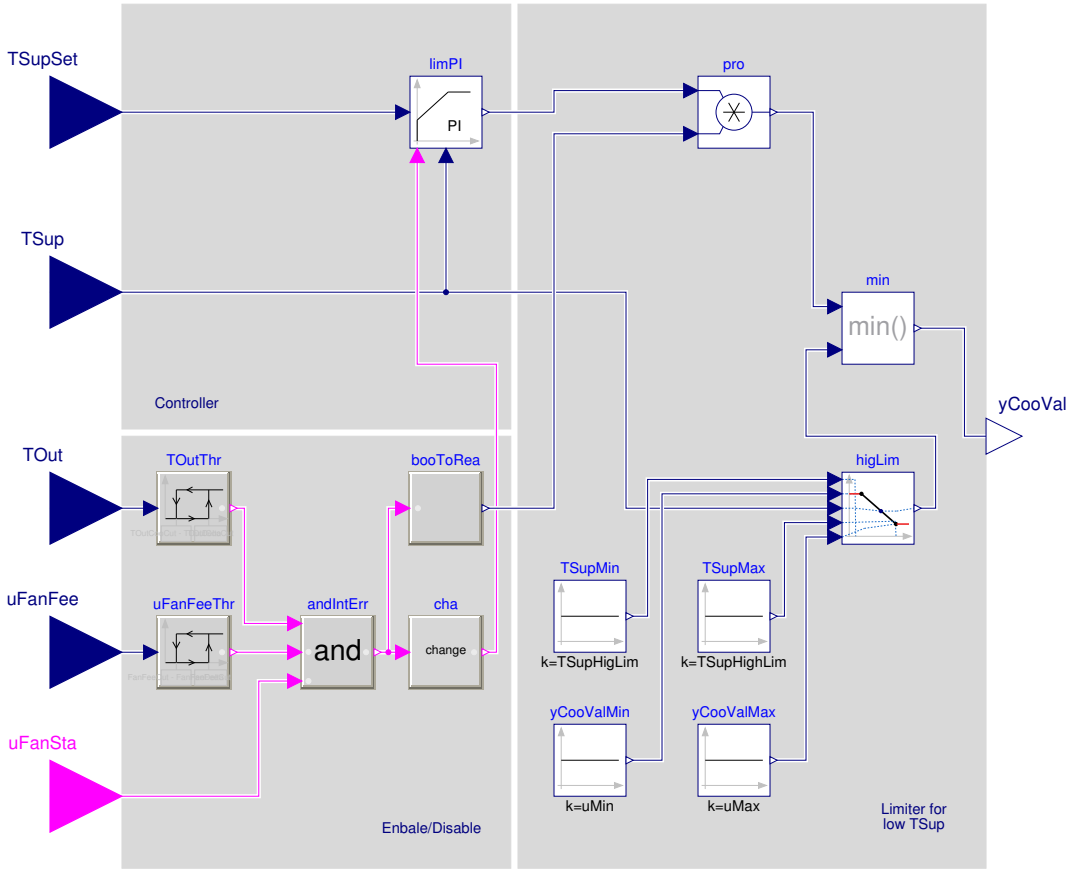


Fig. 9.7: CDL specification of the cooling coil valve position control sequence.

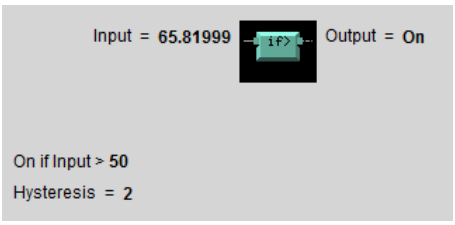


Fig. 9.8: ALC EIKON outdoor air temperature hysteresis to enable/disable the controller

Fig. 9.9: ALC EIKON PI controller parameters

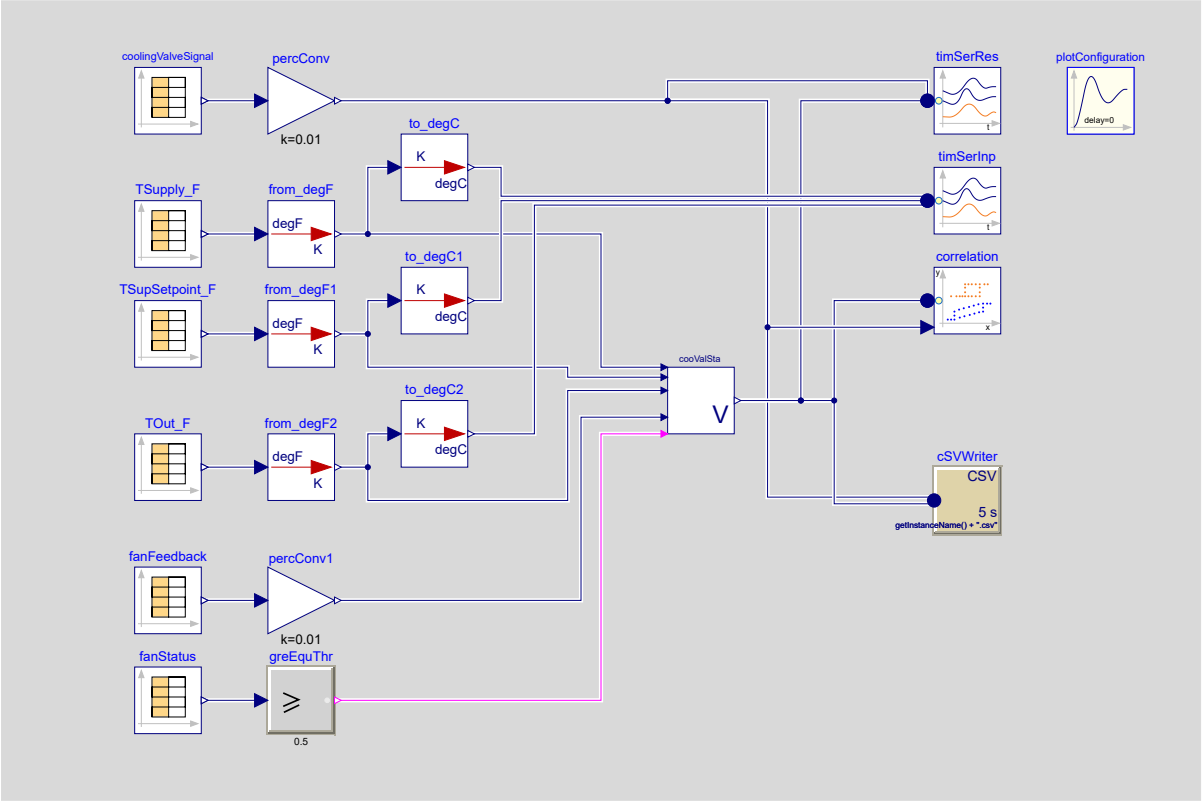
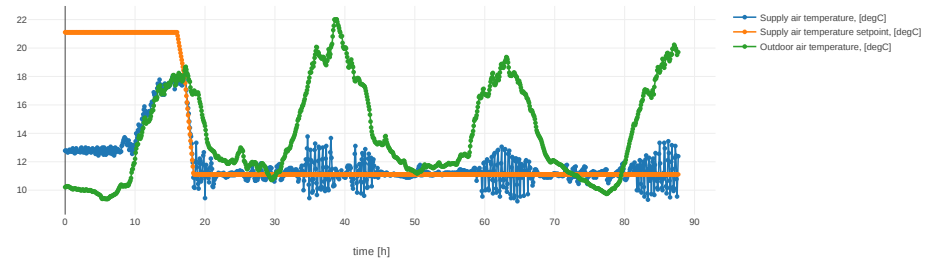
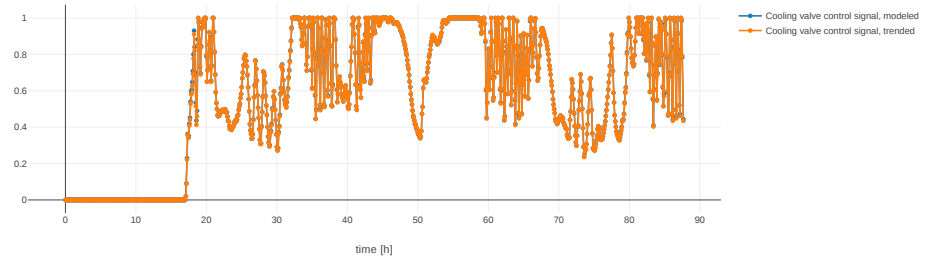


Fig. 9.10: Modelica model that conducts the verification.

Trended input signals



Cooling valve control signal: reference trend vs. modeled result



Modeled result/recorded trend correlation

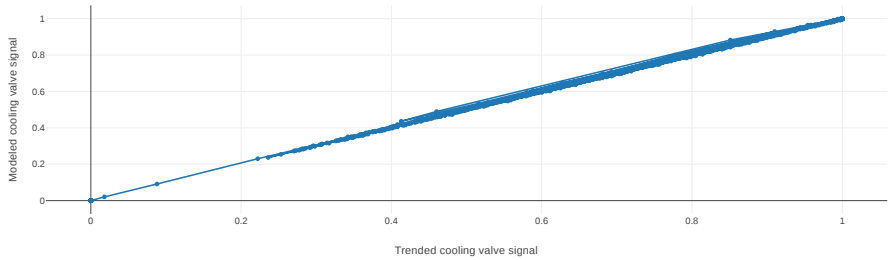


Fig. 9.11: Verification of the cooling valve control signal between ALC EIKON computed signal and simulated signal.

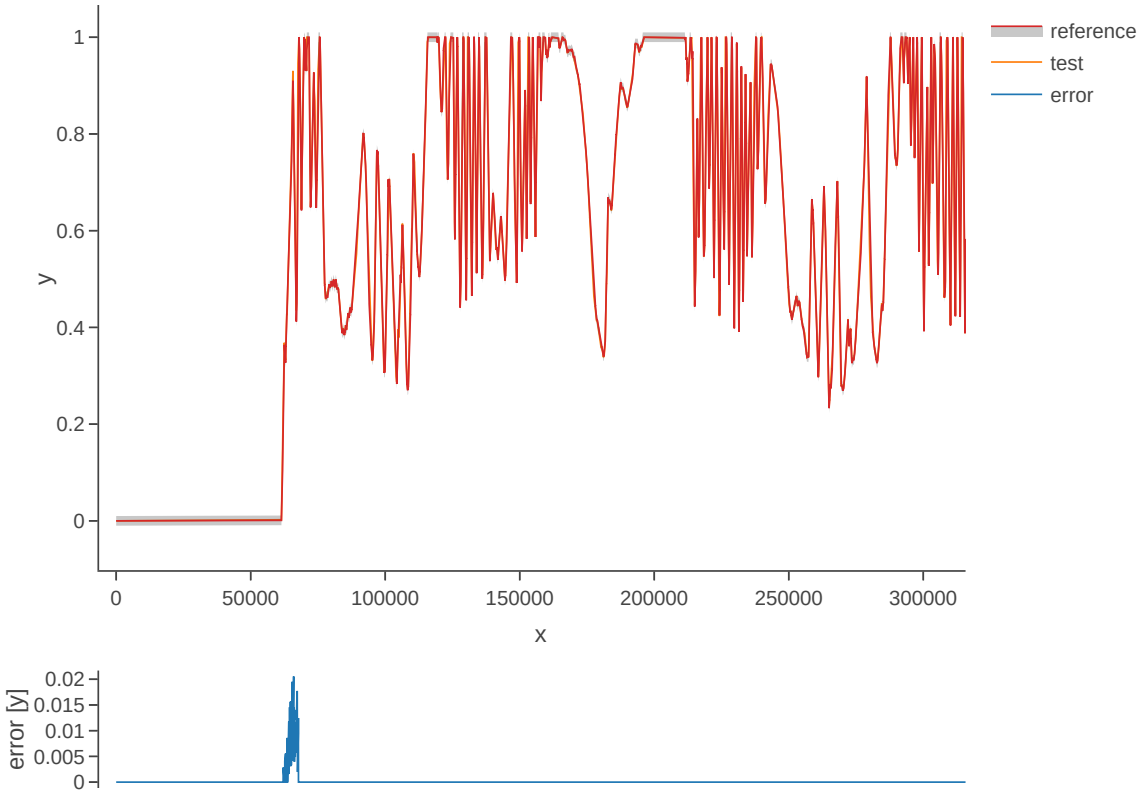


Fig. 9.12: Verification of the cooling valve control signal with the funnel software (error computed with an absolute tolerance in time of 1 s and a relative tolerance in y *of 1%).*

Chapter 10

Example Application

In this section, we compare the performance of two different control sequences, applied to a floor of a prototypical office building. The objectives are to demonstrate the setup for closed loop performance assessment, to demonstrate how to compare the control performance, and to assess the difference in annual energy consumption. For the basecase, we implemented a control sequence published in ASHRAE's Sequences of Operation for Common HVAC Systems [ASH06]. For the other case, we implemented the control sequence published in ASHRAE Guideline 36 [ASHRAE16]. The main conceptual differences between the two control sequences, which are described in more detail in Section 10.1.5, are as follows:

- The base case uses constant supply air temperature setpoints for heating and cooling during occupied hours, whereas the guideline 36 uses one supply air temperature setpoint, which is reset based on outdoor air temperature and zone cooling requests, as obtained from the VAV terminal unit controllers. The reset is dynamic using the trim and respond logic.
- The base case resets the supply fan static pressure setpoint based on the VAV damper position, which is only modulated during cooling, whereas the guideline 36 resets the fan static pressure setpoint based on zone pressure requests from the VAV terminal controllers. The reset is dynamic using the trim and respond logic.
- The base case controls the economizer to track a mixed air temperature setpoint, whereas guideline 36 controls the economizer based on supply air temperature control loop signal.
- The base case controls the VAV dampers based on the zone's cooling temperature setpoint, whereas guideline 36 uses the heating and cooling loop signal to control the VAV dampers.

The next sections are as follows: In Section 10.1 we describe the methodology, the models and the performance metrics, in Section 10.2 we compare the performance, in Section 10.3 we recommend improvements to the guideline 36 and in Section 10.4 we discuss the main findings and present concluding remarks.

10.1 Methodology

All models are implemented in Modelica, using models from the Buildings library [WZNP14].

The models are available from https://github.com/lbl-srg/modelica-buildings/releases/tag/v5.0.0

As a test case, we used a simulation model that consists of five thermal zones that are representative of one floor of the new construction medium office building for Chicago, IL, as described in the set of DOE Commercial Building Benchmarks

[DFS+11]. There are four perimeter zones and one core zone. The envelope thermal properties meet ASHRAE Standard 90.1-2004. The system model consist of an HVAC system, a building envelope model and a model for air flow through building leakage and through open doors based on wind pressure and flow imbalance of the HVAC system.

10.1.1 HVAC model

The HVAC system is a variable air volume (VAV) flow system with economizer and a heating and cooling coil in the air handler unit. There is also a reheat coil and an air damper in each of the five zone inlet branches.

Fig. 10.1 shows the schematic diagram of the HVAC system.

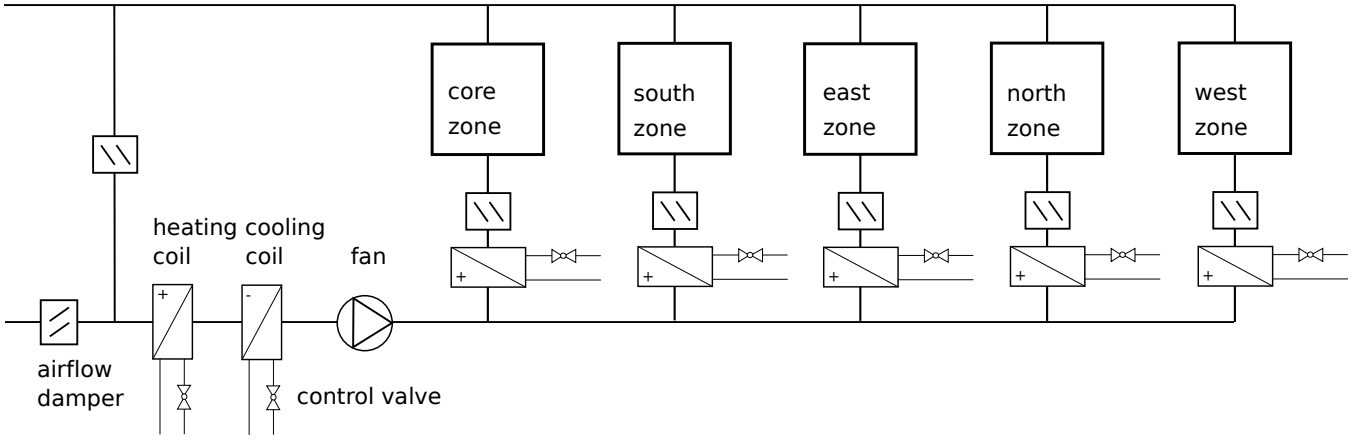


Fig. 10.1: Schematic diagram of the HVAC system.

In the VAV model, all air flows are computed based on the duct static pressure distribution and the performance curves of the fans. The fans are modeled as described in [Wet13].

10.1.2 Envelope heat transfer

The thermal room model computes transient heat conduction through walls, floors and ceilings and long-wave radiative heat exchange between surfaces. The convective heat transfer coefficient is computed based on the temperature difference between the surface and the room air. There is also a layer-by-layer short-wave radiation, long-wave radiation, convection and conduction heat transfer model for the windows. The model is similar to the Window 5 model. The physics implemented in the building model is further described in [WZN11].

There is no moisture buffering in the envelope, but the room volume has a dynamic equation for the moisture content.

10.1.3 Internal loads

We use an internal load schedule as shown in Fig. 10.2, of which 20% is radiant, 40% is convective sensible and 40% is latent. Each zone has the same internal load per floor area.

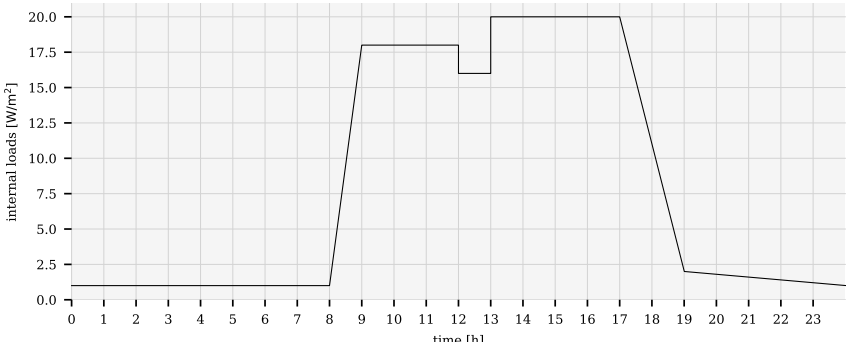


Fig. 10.2: *Internal load schedule.*

10.1.4 Multi-zone air exchange

Each thermal zone has air flow from the HVAC system, through leakages of the building envelope (except for the core zone) and through bi-directional air exchange through open doors that connect adjacent zones. The bi-directional air exchange is modeled based on the differences in static pressure between adjacent rooms at a reference height plus the difference in static pressure across the door height as a function of the difference in air density. Air infiltration is a function of the flow imbalance of the HVAC system. The multizone airflow models are further described in [Wet06].

10.1.5 Control sequences

For the above models, we implemented two different control sequences, which are described below. The control sequences are the only difference between the two cases.

For the base case, we implemented the control sequence *VAV 2A2-21232* of the Sequences of Operation for Common HVAC Systems [ASH06]. In this control sequence, the supply fan speed is regulated based on the duct static pressure. The duct static pressure is adjusted so that at least one VAV damper is 90% open. The economizer dampers are modulated to track the setpoint for the mixed air dry bulb temperature. The supply air temperature setpoints for heating and cooling are constant during occupied hours, which may not comply with some energy codes. Priority is given to maintain a minimum outside air volume flow rate. In each zone, the VAV damper is adjusted to meet the room temperature setpoint for cooling, or fully opened during heating. The room temperature setpoint for heating is tracked by varying the water flow rate through the reheat coil. There is also a finite state machine that transitions the mode of operation of the HVAC system between the modes *occupied*, *unoccupied off*, *unoccupied night set back*, *unoccupied warm-up* and *unoccupied pre-cool*. Local loop control is implemented using proportional and proportional-integral controllers, while the supervisory control is implemented using a finite state machine.

For the detailed implementation of the control logic, see the model Buildings.Examples.VAVReheat.ASHRAE2006, which is also shown in Fig. 10.6.

Our implementation differs from VAV 2A2-21232 in the following points:

- We removed the return air fan as the building static pressure is sufficiently large. With the return fan, building static pressure was not adequate.

- In order to have the identical mechanical system as for guideline 36, we do not have a minimum outdoor air damper, but rather controlled the outdoor air damper to allow sufficient outdoor air if the mixed air temperature control loop would yield too little outdoor air.

For the guideline 36 case, we implemented the multi-zone VAV control sequence based on [ASHRAE16]. Fig. 10.3 shows the sequence diagram, and the detailed implementation is available in the model Buildings.Examples.VAVReheat.Guideline36.

In the guideline 36 sequence, the duct static pressure is reset using trim and respond logic based on zone pressure reset requests, which are issued from the terminal box controller based on whether the measured flow rate tracks the set point. The implementation of the controller that issues these system requests is shown in Fig. 10.4. The economizer dampers are modulated based on a control signal for the supply air temperature set point, which is also used to control the heating and cooling coil valve in the air handler unit. Priority is given to maintain a minimum outside air volume flow rate. The supply air temperature setpoints for heating and cooling at the air handler unit are reset based on outdoor air temperature, zone temperature reset requests from the terminal boxes and operation mode.

In each zone, the VAV damper and the reheat coil is controlled using the sequence shown in Fig. 10.5, where `THeaSet` is the set point temperature for heating, `dTDisMax` is the maximum temperature difference for the discharge temperature above `THeaSet`, `TSup` is the supply air temperature, `VAct*` are the active airflow rates for heating (`Hea`) and cooling (`Coo`), with their minimum and maximum values denoted by `Min` and `Max`.

Our implementation differs from guideline 36 in the following points:

- Guideline 36 prescribes “To avoid abrupt changes in equipment operation, the output of every control loop shall be capable of being limited by a user adjustable maximum rate of change, with a default of 25% per minute.”
  We did not implement this limitation of the output as it leads to delays which can make control loop tuning more difficult if the output limitation is slower than the dynamics of the controlled process. We did however add a first order hold at the trim and response logic that outputs the duct static pressure setpoint for the fan speed.
- Not all alarms are included.
- Where guideline 36 prescribes that equipment is enabled if a controlled quantity is above or below a setpoint, we added a hysteresis. In real systems, this avoids short-cycling due to measurement noise, in simulation, this is needed to guard against numerical noise that may be introduced by a solver.

10.1.6 Site electricity use

To convert cooling and heating energy as transferred by the coil to site electricity use, we apply the conversion factors from EnergyStar [Ene13]. Therefore, for an electric chiller, we assume an average coefficient of performance (COP) of 3.2 and for a geothermal heat pump, we assume a COP of 4.0.

10.1.7 Simulations

Fig. 10.6 shows the top-level of the system model of the base case, and Fig. 10.7 shows the same view for the guideline 36 model.

The complexity of the control implementation is visible in Fig. 10.4 which computes the temperature and pressure requests of each terminal box that is sent to the air handler unit control.

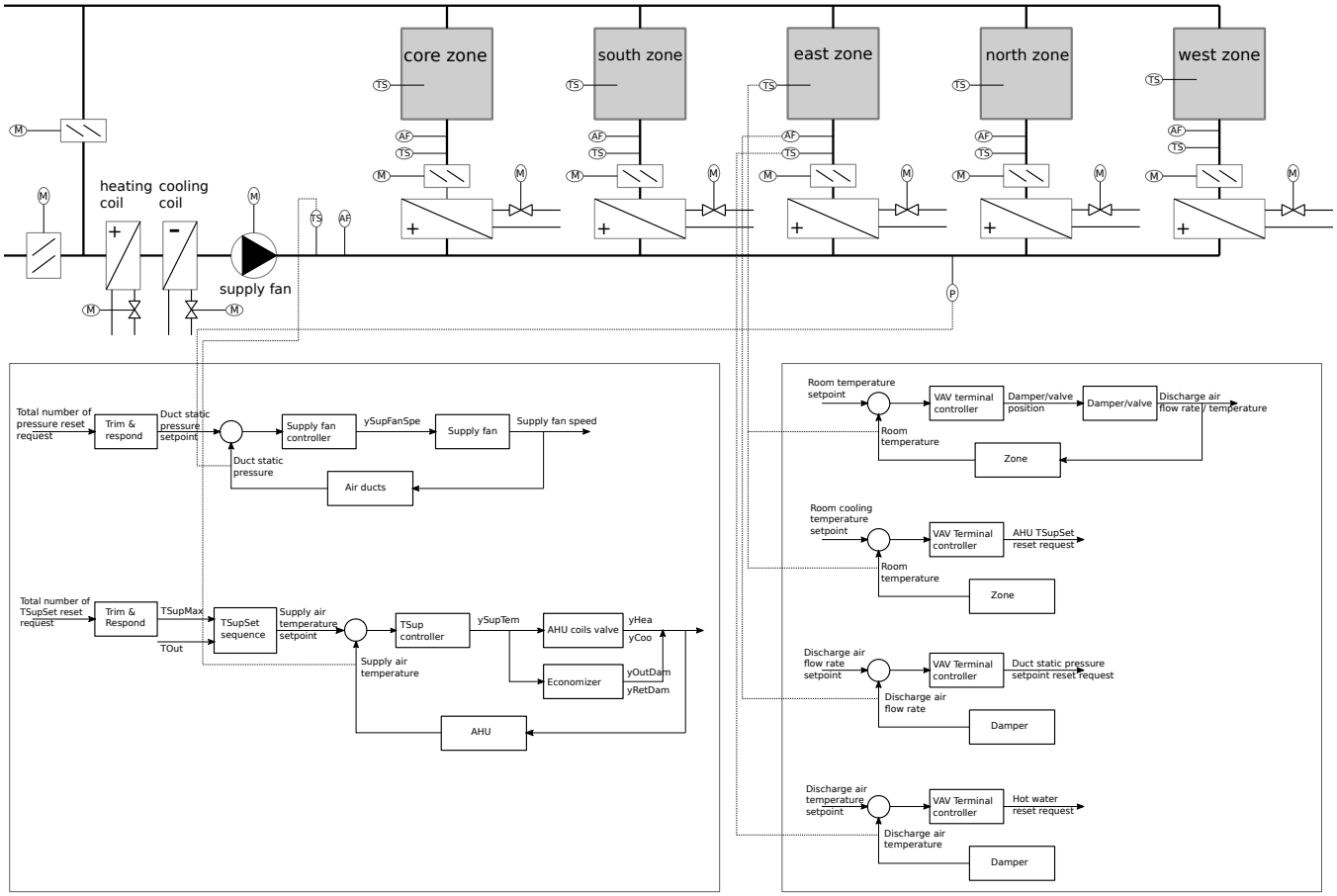


Fig. 10.3: *Control schematics of guideline 36 case.*

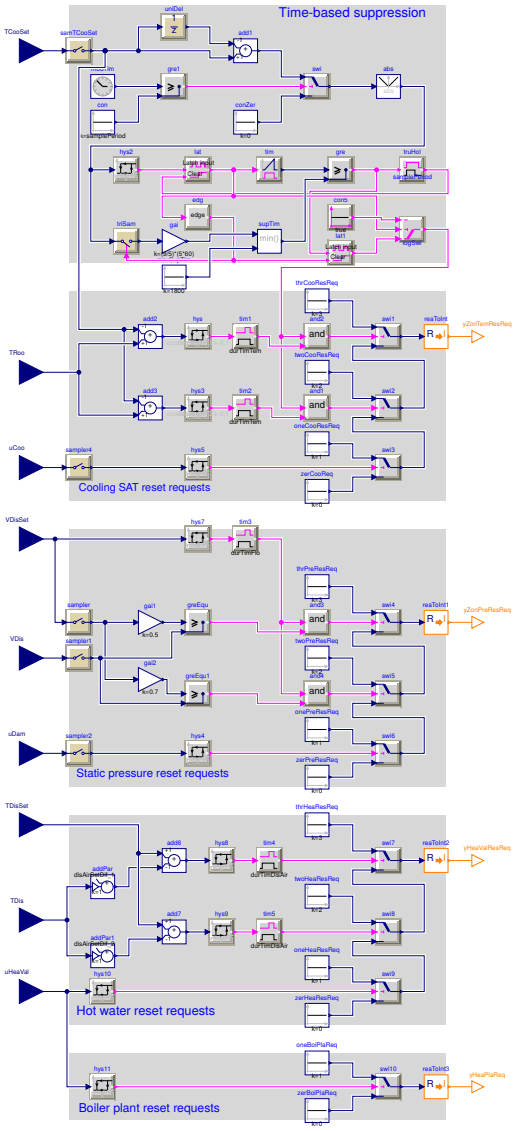


Fig. 10.4: Composite block that implements the sequence for the VAV terminal units that output the system requests. (Browsable version.)

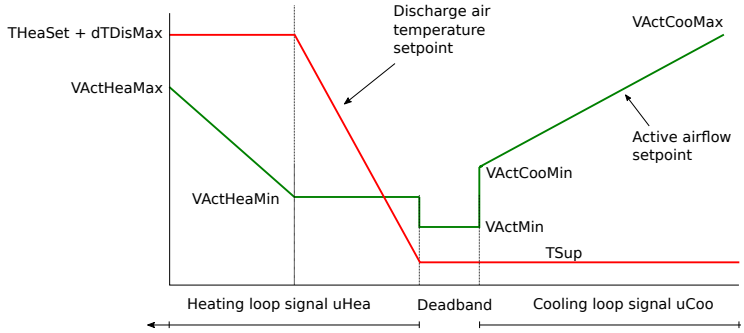


Fig. 10.5: Control sequence for VAV terminal unit.

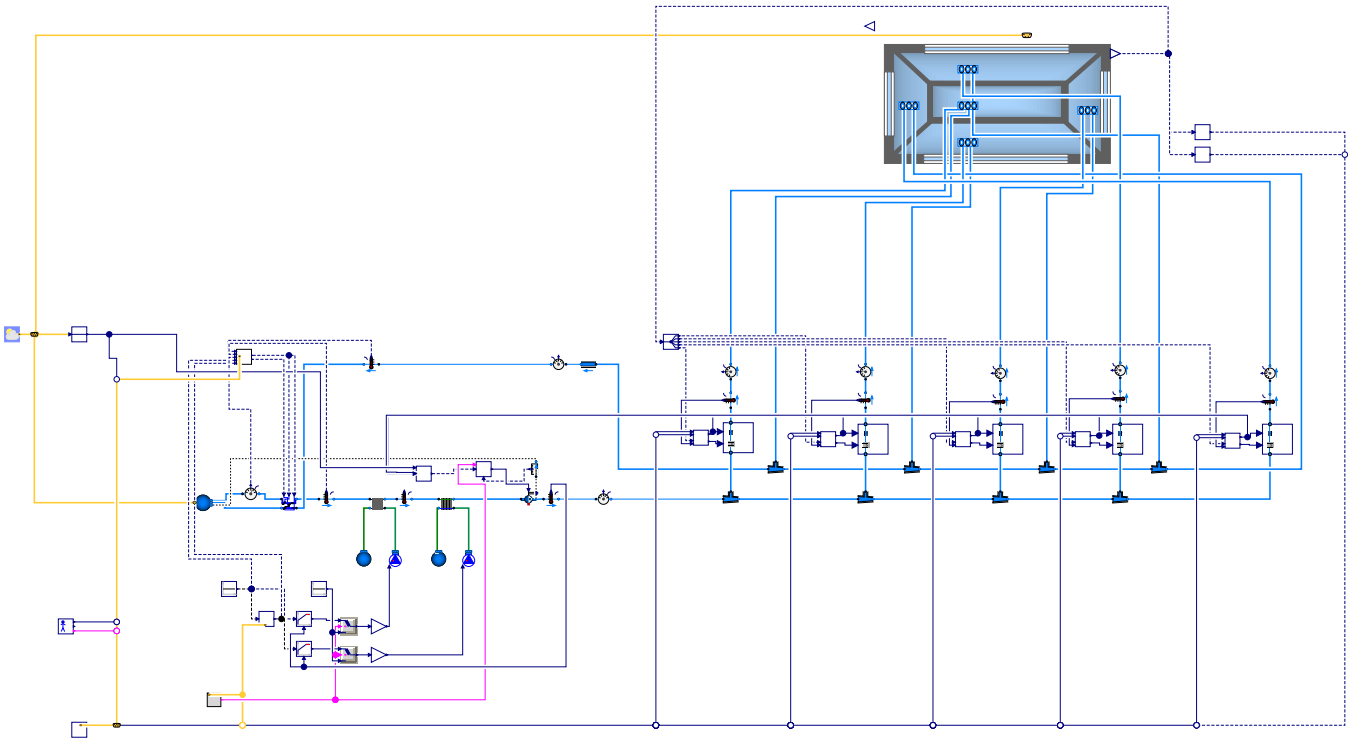


Fig. 10.6: Top level view of Modelica model for the base case.

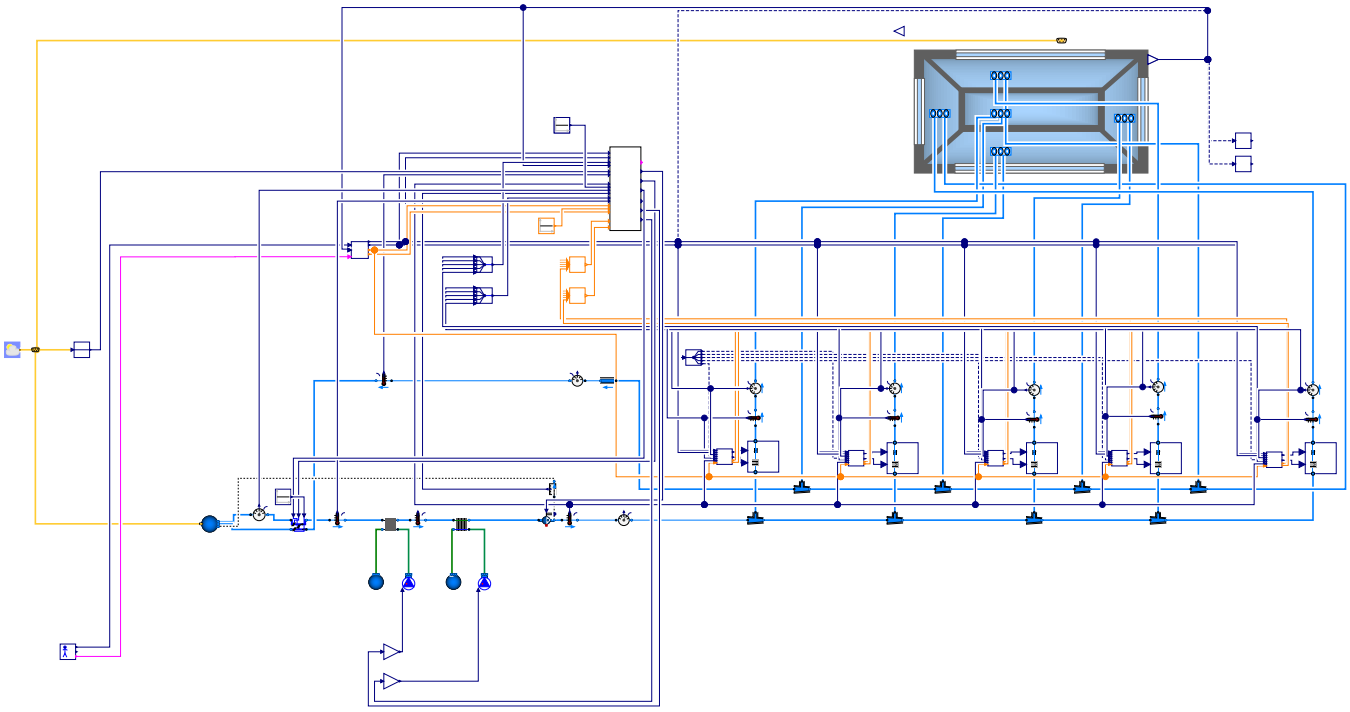


Fig. 10.7: Top level view of Modelica model for the guideline 36 case.

All simulations were done with Dymola 2018 FD01 beta3 using Ubuntu 16.04 64 bit. We used the Radau solver with a tolerance of 10^{-6}. This solver adaptively changes the time step to control the integration error. Also, the time step is adapted to properly simulate *time events* and *state events*.

The base case and the guideline 36 case use the same HVAC and building model, which is implemented in the base class `Buildings.Examples.VAVReheat.BaseClasses.PartialOpenLoop`. The two cases differ in their implementation of the control sequence only, which is implemented in the models `Buildings.Examples.VAVReheat.BaseClasses.ASHRAE2006` and `Buildings.Examples.VAVReheat.BaseClasses.Guideline36`.

Table 10.1 shows an overview of the model and simulation statistics. The differences in the number of variables and in the number of time varying variables reflect that the guideline 36 control is significantly more detailed than what may otherwise be used for simulation of what the authors believe represents a realistic implementation of a feedback control sequence. The entry approximate number of control I/O connections counts the number of input and output connections among the control blocks of the two implementations. For example, If a P controller receives one set point, one measured quantity and sends it signal to a limiter and the limiter output is connected to a valve, then this would count as four connections. Any connections inside the PI controller would not be counted, as the PI controller is an elementary building block (see Section 7.4) of CDL.

Table 10.1: Model and simulation statistics.

Quantity	Base case	Guideline 36
Number of components	2826	4400
Number of variables (prior to translation)	33,700	40,400
Number of continuous states	178	190
Number of time-varying variables	3400	4800
Time for annual simulation in minutes	70	100

10.2 Performance comparison

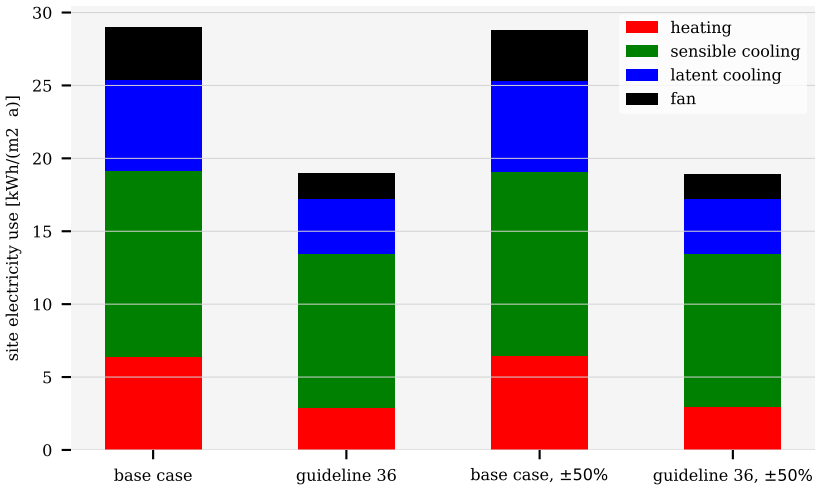


Fig. 10.8: Comparison of energy use. For the cases labeled ±50%, the internal gains have been increased and decreased as described in Section 10.1.3.

Table 10.2: Heating, cooling, fan and total site energy, and savings of guideline 36 case versus base case.

E_h $[kWh/(m^2\,a)]$	E_c $[kWh/(m^2\,a)]$	E_f $[kWh/(m^2\,a)]$	E_{tot} $[kWh/(m^2\,a)]$	[%]
6.419	18.98	3.572	28.97	
2.912	14.29	1.74	18.94	34.6

Fig. 10.8 and Table 10.2 compare the annual site electricity use between the annual simulations with the base case control and the guideline 36 control. The bars labeled ±50% were obtained with simulations in which we changed the diversity of the internal loads. Specifically, we reduced the internal loads for the north zone by 50% and increased them for the south zone by the same amount.

The guideline 36 control saves around 30% site electrical energy. These are significant savings that can be achieved through software only, without the need for additional hardware or equipment. Our experience, however, was that it is

rather challenging to program guideline 36 sequence due to their complex logic that contains various mode changes, interlocks and timers. Various programming errors and misinterpretations or ambiguities of the guideline were only discovered in closed loop simulations. We therefore believe it is important to provide robust, validated implementations of guideline 36 that encapsulates the complexity for the energy modeler and the control provider.

Fig. 10.9 shows the outside air temperature temperature T_{out} and the global horizontal irradiation $H_{glo,hor}$ for a period in winter, spring and summer. These days will be used to compare the trajectories of various quantities of the baseline and of guideline 36.

Fig. 10.10 compares the time trajectories of the room air temperatures. The figures show that the room air temperatures are controlled within the setpoints for both cases. Small set point violations have been observed due to the dynamic nature of the control sequence and the controlled process.

Fig. 10.11 shows the control signals of the reheat coils y_{hea} and the VAV damper y_{vav} for the north and south zones.

Fig. 10.12 shows the temperatures of the air handler unit. The figure shows the supply air temperature after the fan T_{sup}, its control error relative to its set point $T_{set,sup}$, the mixed air temperature after the economizer T_{mix} and the return air temperature from the building T_{ret}. A notable difference is that guideline 36 resets the supply air temperature, whereas the base case is controlled for a supply air temperature of 10°C for heating and 12°C for cooling.

Fig. 10.13 show reasonable fan speeds and economizer operation. Note that during the winter days 5, 6 and 7, the outdoor air damper opens. However, this is only to track the setpoint for the minimum outside air flow rate as the fan speed is at its minimum.

Fig. 10.14 shows the volume flow rate of the fan $\dot{V}_{fan,sup}/V_{bui}$, where V_{bui} is the volume of the building, and of the outside air intake of the economizer $\dot{V}_{eco,out}/V_{bui}$, expressed in air changes per hour. Note that guideline 36 has smaller outside air flow rates in cold winter and hot summer days. The system has relatively low air changes per hour. As fan energy is low for this building, it may be more efficient to increase flow rates and use higher cooling and lower heating temperatures, in particular if heating and cooling is provided by a heat pump and chiller. We have however not further analyzed this trade-off.

Fig. 10.15 compares the room air temperatures for the north and south zone for the standard internal loads, and the case where we reduced the internal loads in the north zone by 50% and increased it by the same amount in the south zone. The trajectories with subscript $\pm$50% are the simulations with the internal heat gains reduced or increased by 50%. The room air temperature trajectories are practically on top of each other for winter and spring, but the guideline 36 sequence shows somewhat better setpoint tracking during summer. Both control sequences are comparable in terms of compensating for this diversity, and as we saw in Fig. 10.8, their energy consumption is not noticeably affected.

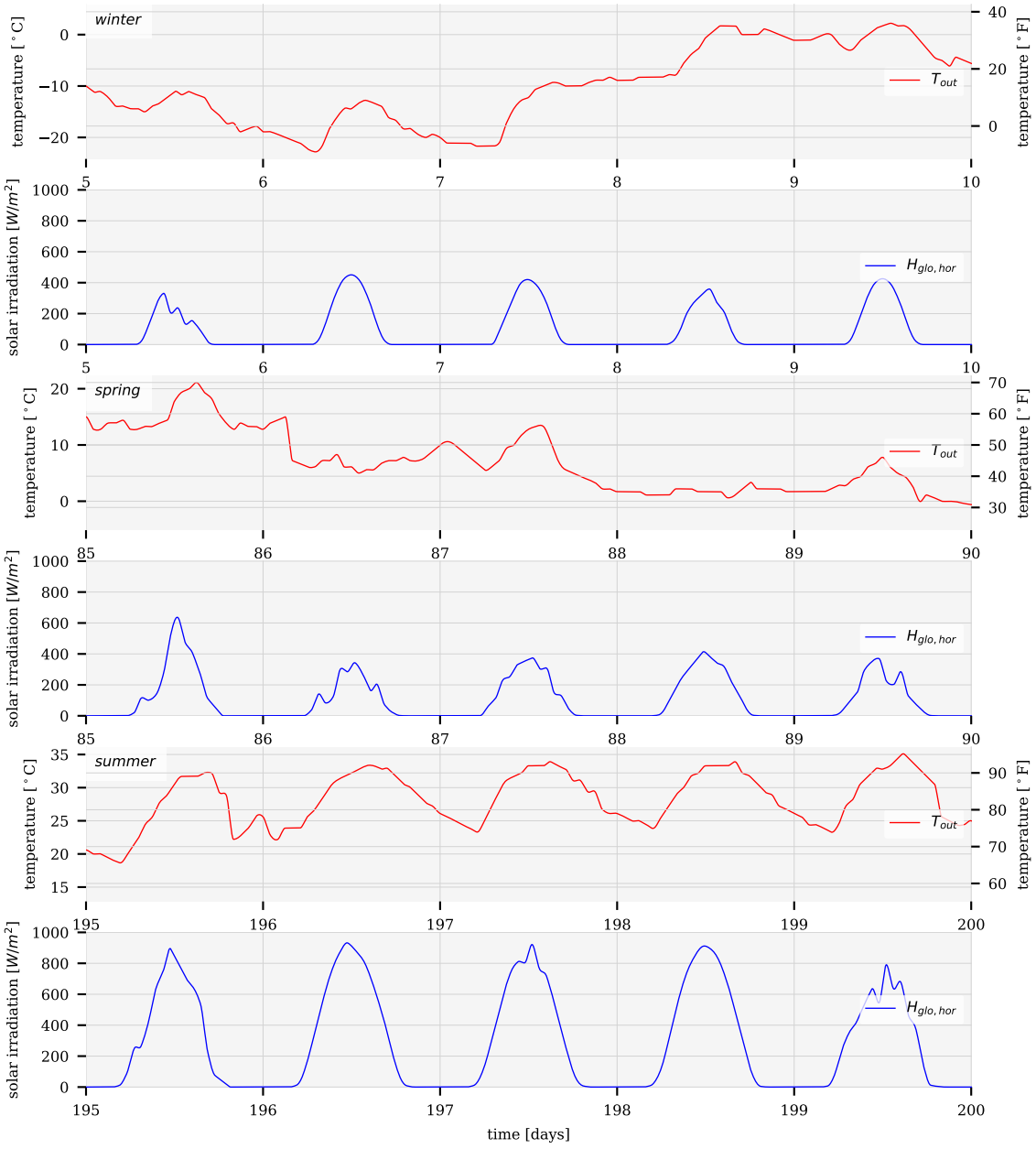


Fig. 10.9: Outside air temperature and global horizontal irradiation for the three periods that will be further used in the analysis.

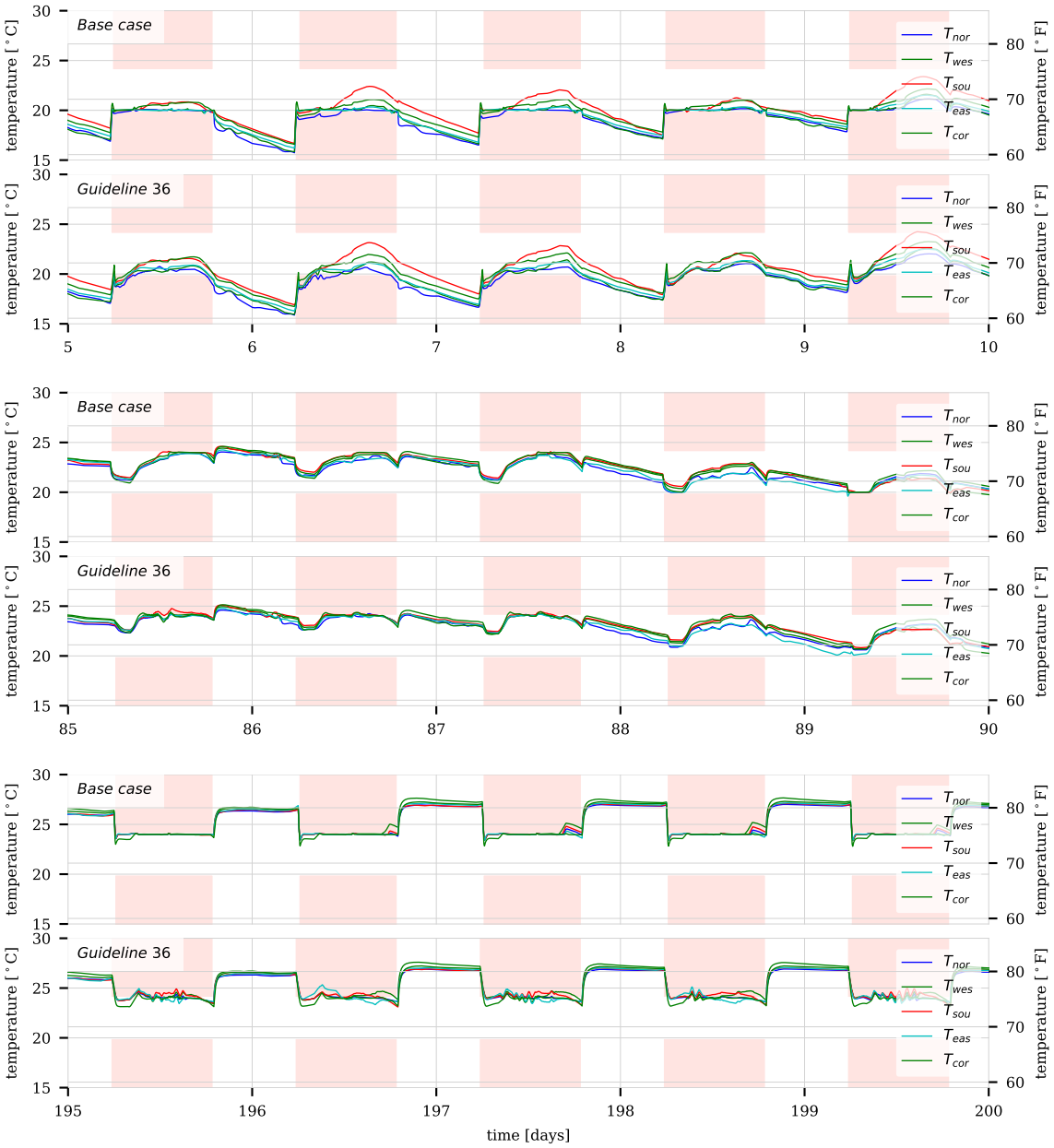


Fig. 10.10: Room air temperatures. The white area indicates the region between the heating and cooling setpoint temperatures.

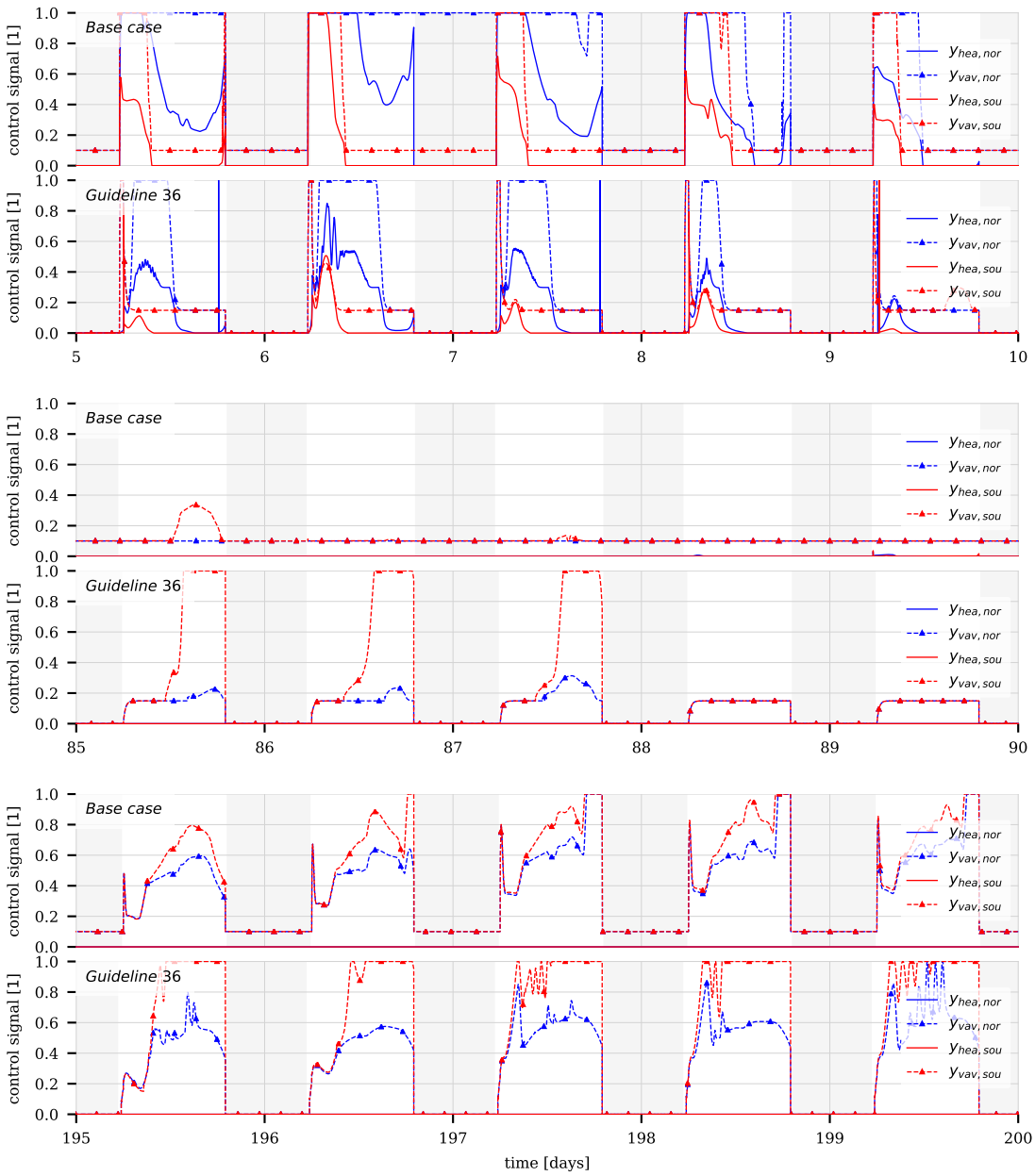


Fig. 10.11: VAV control signals for the north and south zones. The white areas indicate the day-time operation.

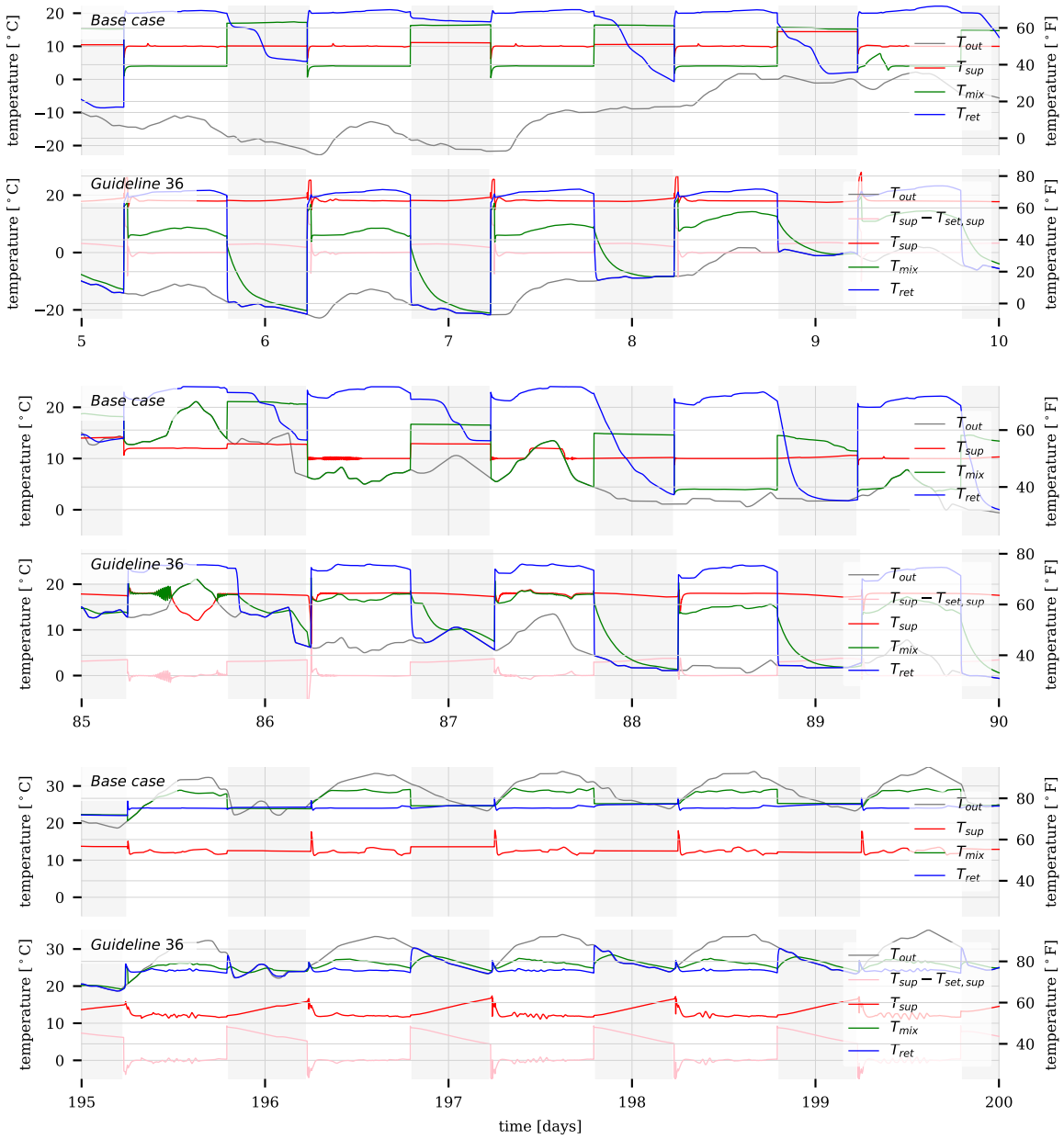


Fig. 10.12: AHU temperatures.

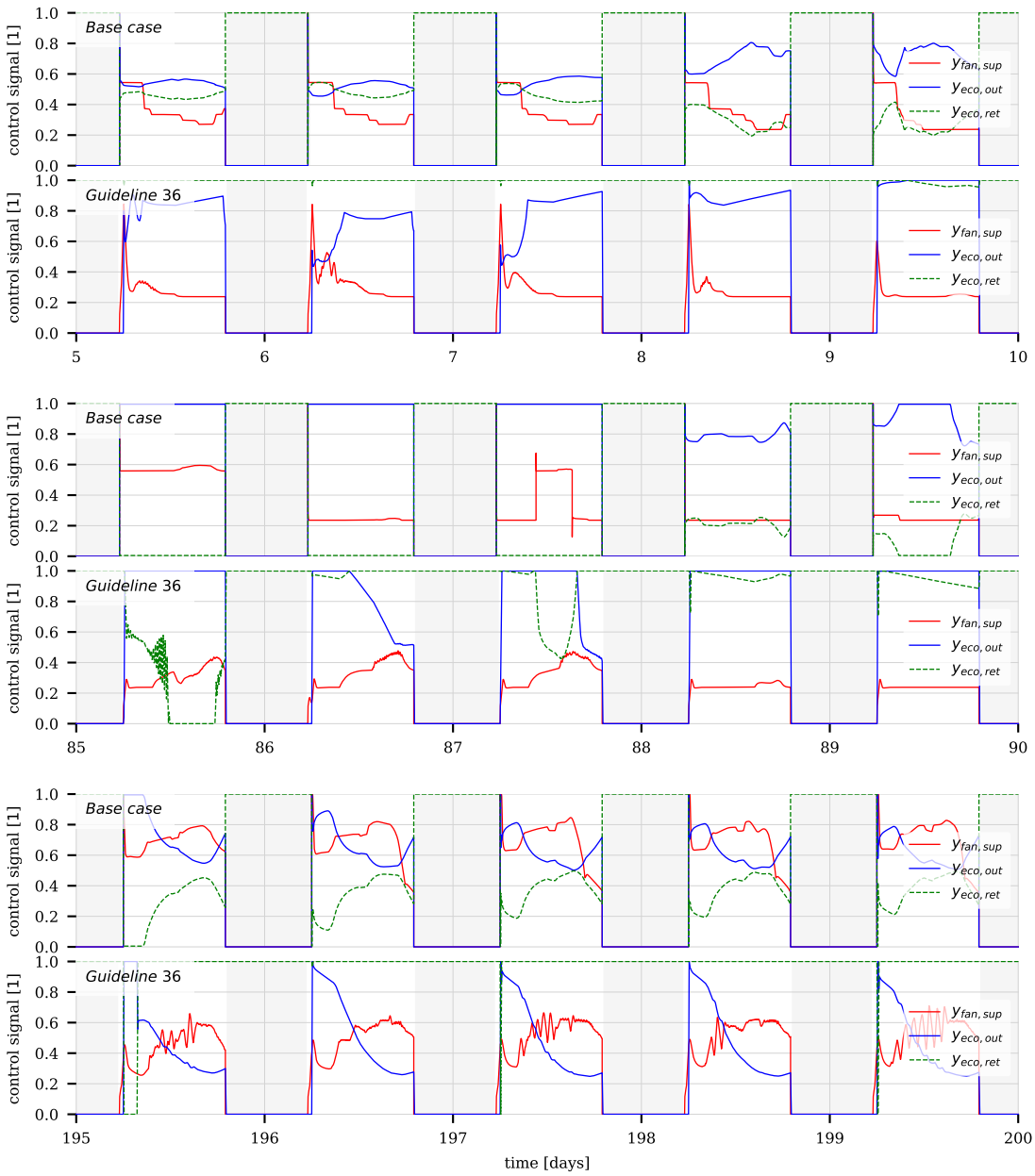


Fig. 10.13: Control signals for the supply fan, outside air damper and return air damper.

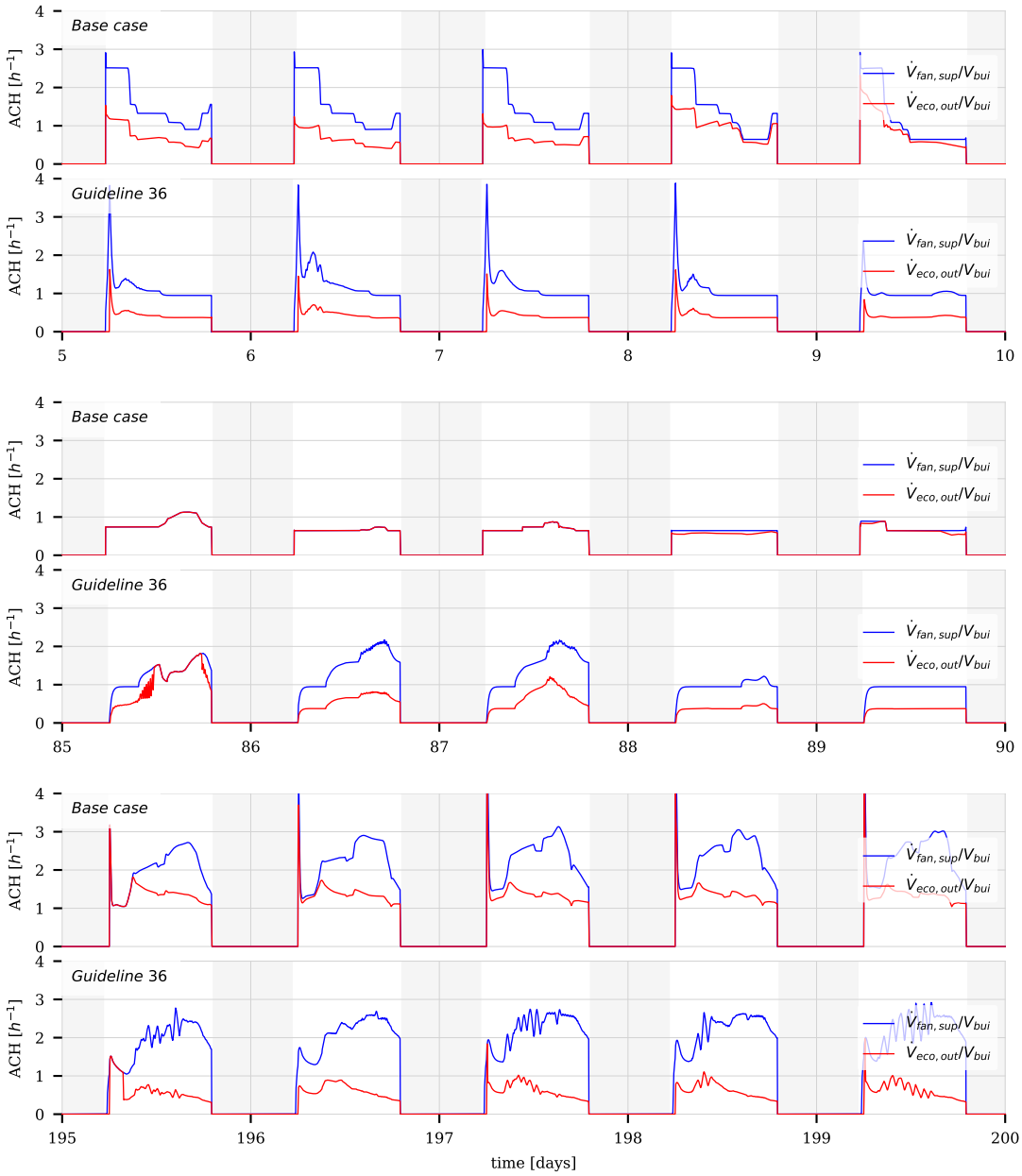


Fig. 10.14: Fan and outside air volume flow rates, normalized by the room air volume.

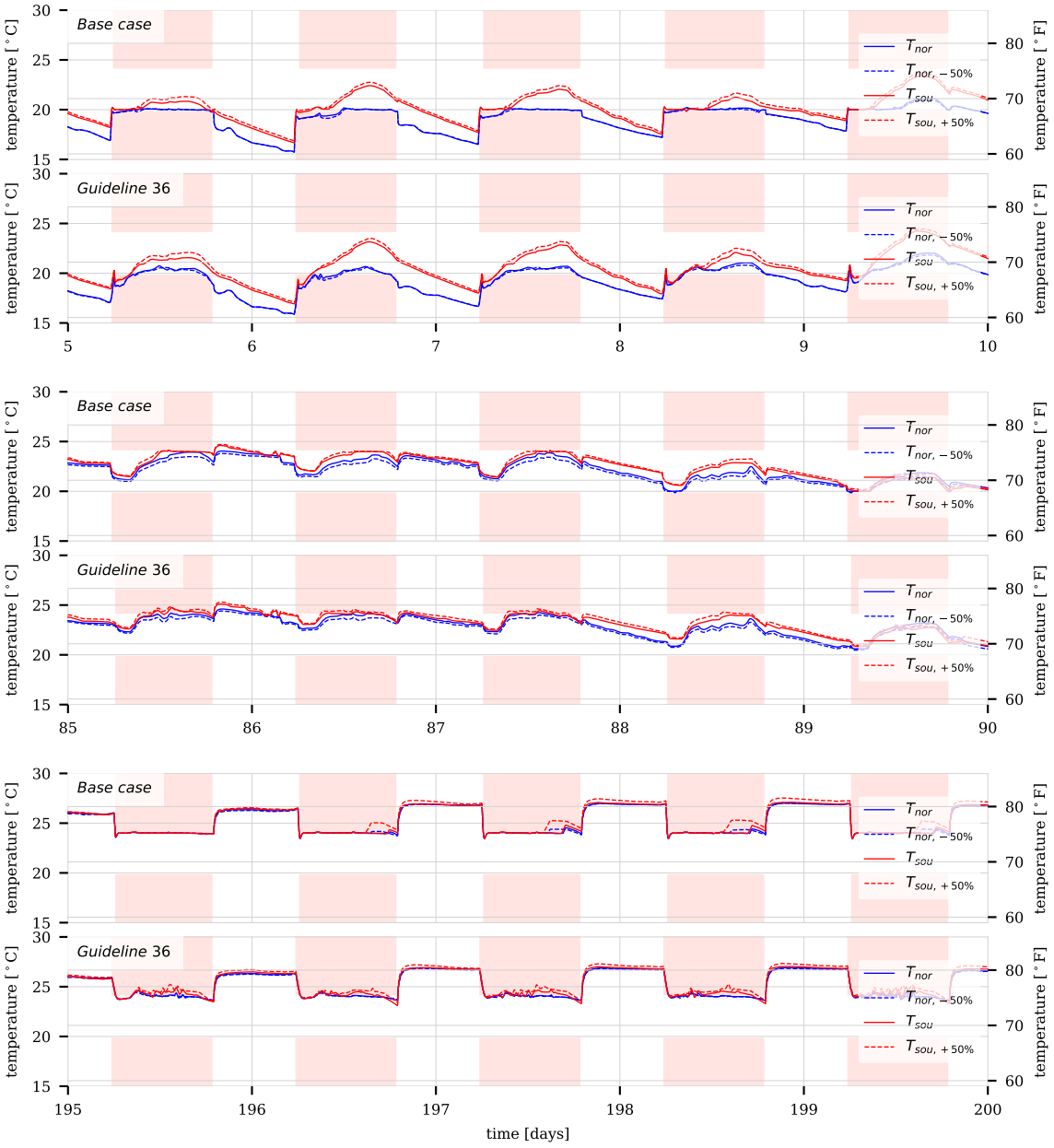


Fig. 10.15: Outdoor air and room air temperatures for the north and south zone with equal internal loads, and with diversity added to the internal loads. The white area indicates the region between the heating and cooling setpoint temperatures.

10.3 Improvement to guideline 36 specification

This section describes improvements that we recommend for the guideline 36 specification, based on the first public review draft [ASHRAE16].

10.3.1 Freeze protection for mixed air temperature

The sequences have no freeze protection for the mixed air temperature.

The guideline states (emphasis added):

> If the supply air temperature drops below 4.4°C (40°F) for 5 minutes, send two (or more, as required to ensure that heating plant is active) Boiler Plant Requests, override the outdoor air damper to the minimum position, and *modulate the heating coil to maintain a supply air temperature* of at least 5.6° C (42°F). Disable this function when supply air temperature rises above 7.2°C (45°F) for 5 minutes.

Depending on the outdoor air requirements, the mixed air temperature T_{mix} may be below freezing, which could freeze the heating coil if it has low water flow rate. Note that guideline 36 controls based on the supply air temperature and not the mixed air temperature. Hence, this control would not have been active.

Fig. 10.16 shows the mixed air temperature and the economizer control signal for cold climate. The trajectories whose subscripts end in *no* are without freeze protection control based on the mixed air temperture, as is the case for guideline 36, whereas for the trajectories that end in *with*, we added freeze protection that adjusts the economizer to limit the mixed air temperature. For these simulations, we reduced the outdoor air temperture by 10 Kelvin (18 Fahrenheit) below the values obtained from the TMY3 weather data. This caused in day 6 and 7 in Fig. 10.16 sub-freezing mixed air temperatures during day-time, as the outdoor air damper was open to provide sufficient fresh air. We also observed that outside air is infiltrated through the AHU when the fan is switched off. This is because the wind pressure on the building causes the building to be slightly below the ambient pressure, thereby infiltrating air through the economizers closed air dampers (that have some leakage). This causes a mixed air temperatures below freezing at night when the fan is off. Note that these temperatures are qualitative rather than quantitative results as the model is quite simplified at these small flow rates, which are about 0.01% of the air flow rate that the model has when the fan is operating.

We therefore recommend adding either of the following wording to guideline 36, which is translated from [Bun86]: Use a capillary sensor installed after the heating coil. If the temperature after the heating coil is below $4°C$,

1. enable frost protection by opening the heating coil valve,
2. send frost alarm,
3. switch on pump of the heating coil.

The frost alarm requires manual confirmation.

If the temperature at the capillary sensor exceeds $6°C$, close the valve but keep the pump running until the frost alarm is manually reset. (Closing the valve avoids overheating).

Recknagel [RSS05] adds further:

1. Add bypass at valve to ensure 5% water flow.
2. In winter, keep bypass always open, possibly with thermostatically actuated valve.
3. If the HVAC system is off, keep the heating coil valve open to allow water flow if there is a risk of frost in the AHU room.

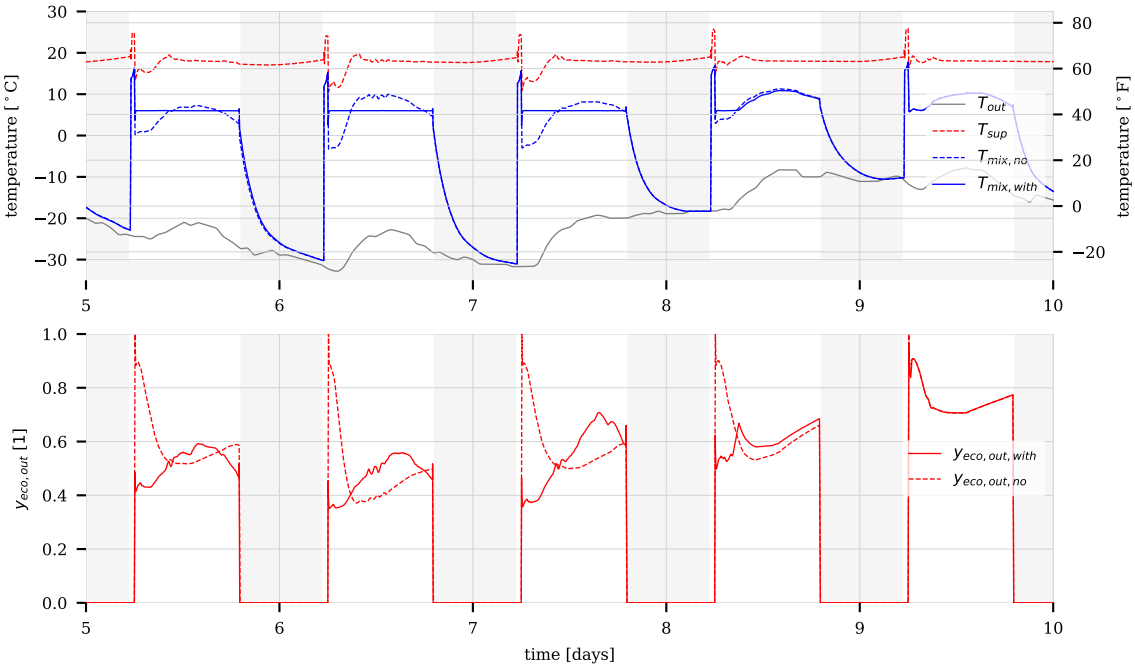


Fig. 10.16: Mixed air temperature and economizer control signal for guideline 36 case with and without freeze protection.

4. If the heating coil is closed, open the outdoor air damper with a time delay when fan switches on to allow heating coil valve to open.
5. For pre-heat coil, install a circulation pump to ensure full water flow rate through the coil.

10.3.2 Deadbands for hard switches

There are various sequences in which the set point changes as a step function of the control signal, such as shown in Fig. 10.5. In our simulations of the VAV terminal boxes, the switch in air flow rate caused chattering. We circumvented the problem by checking if the heating control signal remains bigger than 0.05 for 5 minutes. If it falls below 0.01, the timer was switched off. This avoids chattering. We therefore recommend to be more explicit for where to add hysteresis or time delays.

10.3.3 Averaging air flow measurements

The guideline 36 does not seem to prescribe that outdoor airflow rate measurements need to be time averaged. As such measurements can fluctuate due to turbulence, we recommend to consider averaging this measurement. In the simulations, we averaged the outdoor airflow measurement over a 5 second moving window (in the simulation, this was done to avoid an algebraic system of equations, but in practice, this would filter measurement noise).

10.3.4 Minor editorial revision

The guideline states:

> When a control loop is enabled or re-enabled, it and all its constituents (such as the proportional and integral terms) shall be set initially to a Neutral value.

This should be changed to "... such as the integral terms..." because the proportional term cannot be reset.

10.3.5 Cross-referencing and modularization

For citing individual sections or blocks of the guideline, it would be helpful if the guideline where available at a permanent web site as html, with a unique url and anchor to each section. This would allow cross-referencing the guideline from a particular implementation in a way that allows the user to quickly see the original specification.

As part of such a restructuring, it would be helpful for the reader to clearly state what are the input signals, what are configurable parameters, such as the control gain, and what are the output signals. This in turn would structure the guideline into distinct modules, for which one could also provide a reference implementation in software.

10.3.6 Lessons learned regarding the simulations

This is probably the most detailed building control simulation that has been done with the Modelica Buildings library. A few lessons have been learned and are reported here. Note that related best-practices are also available at http://simulationresearch.lbl.gov/modelica/userGuide/bestPractice.html

- *Fan with prescribed mass flow rate:* In earlier implementations, we converted the control signal for the fan to a mass flow rate, and used a fan model whose mass flow rate is equal to its control input, regardless of the pressure head. During start of the system, this caused a unrealistic large fan head of 4000 Pa (16 inch of water) because VAV dampers opened slowly. The large pressure drop also lead to large power consumption and hence unrealistic temperature increase across the fan.
- *Fan with prescribed presssure head:* We also tried to use a fan with prescribed pressure head. Similarly as above, the fan maintains the presssure head as obtained from its control signal, regardless of the volume flow rate. This caused unrealistic large flow rates in the return duct which has very small pressure drops. (Subsequently, we removed the return fan as it is not needed for this system.)
- *Time sampling certain physical calculations:* Dymola 2018FD01 uses the same time step for all continuous-time equations. Depending on the dynamics, this can be smaller than a second. Since some of the control samples every 2 minutes, it has shown to be favorable to also time sample the long-wave radiation network, whose solution is time consuming. We expect further speed up can be achieved by time sampling other slow physical processes.
- *Non-convergence:* In earlier simulations, sometimes the solver failed to converge. This was due to errors in the control implementation that caused event iterations for discrete equations that seemed to have no solution. In another case, division by zero in the control implementation caused a problem. The solver found a way to work around this division by zero (using heuristics) but then failed to converge. Since we corrected these issues, the simulations are stable.
- *Too fast dynamics of coil:* The cooling coil is implemented using a finite volume model. Heat conduction across the water and air flow used to be neglected as the mode of heat transfer is dominated by forced convection if the fan and pump are operating. However, during night when the system is off, the small infiltration due to wind pressure caused in earlier simulations the water in the coil to freeze. Adding diffusion circumvented this problem, and the coil model in the library includes now by default a small diffusive term.

10.4 Discussion and conclusions

The guideline 36 sequence reduced annual site energy by 30% compared to the baseline implementation, by comparable thermal comfort. Such savings are significant, and have been achieved by software only that can relatively easy be deployed to buildings.

Implementing the sequence was however rather challenging due to its complexity caused by the various mode changes, interlocks, timers and cascading control loops. These mode changes, interlocks and dynamic dependencies made verification of the correctness through inspection of the control signals difficult. As a consequence, various programming errors and misinterpretations or ambiguities of the guideline were only discovered in closed loop simulations, despite of having implemented open-loop test cases for each block of the sequence. We therefore believe it is important to provide robust, validated implementations of the sequences published in guideline 36. Such implementations would encapsulate the complexity and provide assurances that energy modeler and control providers have correct implementations. With the implementation in the Modelica package *Buildings.Controls.OBC.ASHRAE.G36_PR1*, we made a start for such an implementation and laid out the structure and conventions, but have not covered all of the guideline yet. Furthermore, conducting field validations would be useful too.

A key short-coming from an implementer point of view was that the sequence was only available in English language, and as an implementation in ALC EIKON of sequences that are “close to the currently used version of the guideline”. Neither allowed a validation of the CDL implementation because the English language version leaves room for interpretation (and cannot be executed) and because EIKON has quite limited simulation support that is cumbersome to use for testing the dynamic response of control sequences for different input trajectories. Therefore, a benefit of the Modelica implementation

is that such reference trajectories can now easily be generated to validate alternate implementations.

A benefit of the simulation based assessment was that it allowed detecting potential issues such as a mixed air temperature below the freezing point (Section 10.3.1) and chattering due to hard switches (Section 10.3.2). Having a simulation model of the controlled process also allowed verification of work-arounds for these issues.

One can, correctly, argue that the magnitude of the energy savings are higher the worse the baseline control is. However, the baseline control was carefully implemented, following the author's interpretation of ASHRAE's Sequences of Operation for Common HVAC Systems [ASH06]. While higher efficiency of the baseline may be achieved through supply air temperature reset or different economizer control, such potential improvements were only recognized after seeing the results of the guideline 36 sequence. Thus, regardless of whether a building is using guideline 36, having a baseline control against which alternative implementations can be compared and benchmarked is an immensely valuable feature enabled by a library of standardized control sequences. Without a benchmark, one can easily claim to have a good control, while not recognizing what potential savings one may miss.

Chapter 11

Glossary

This section provides definitions for abbreviations and terms introduced in the Open Building Controls project.

Analog Value In CDL, we say a value is analog if it represents a continuous number. The value may be presented by an analog signal such as voltage, or by a digital signal.

Binary Value In CDL, we say a value is binary if it can take on the values 0 and 1. The value may however be presented by an analog signal that can take on two values (within some tolerance) in order to communicate the binary value.

Building Model Digital model of the physical behavior of a given building over time, which accounts for any elements of the building envelope and includes a representation of internal gains and occupancy. Building model has connectors to be coupled with an environment model and any HVAC and non-HVAC system models pertaining to the building.

CDL See *Controls Description Language*.

Controls Description Language The Control Description Language (CDL) is the language that is used to express control sequences and requirements. It is a declarative language based on a subset of the Modelica language and specified in Section 7.

Controls Design Tool The Controls Design Tool is a software that can be used to

- design control sequences,
- declare formal, executable requirements,
- test the control sequences and the requirements with a model of the HVAC system and the building in the loop, and
- export the control sequence and the verification test in the *Controls Description Language*.

Control Sequence Requirement A requirement is a condition that is tested and either passes, fails, or is untested. For example, a requirement would be that the actual actuation signal is within 2% of the signal computed using the CDL representation of a sequence, provided that they both receive the same input data.

Control System Any software and hardware required to perform the control function for a plant.

Controller A controller is a device that computes control signals for a plant.

Co-simulation Co-simulation refers to a simulation in which different simulation programs exchange run-time data at certain synchronization time points. A master algorithm sets the current time, input and states, and request the simulator to advance time, after which the master will retrieve the new values for the state. Each simulator is responsible for integrating in time its differential equation. See also *model-exchange*.

Events An event is either a *time event* if time triggers the change, or a *state event* if a test on the state triggers the change.

Functional Mockup Interface The Functional Mockup Interface (FMI) standard defines an open interface to be implemented by an executable called *Functional Mockup Unit* (FMU). The FMI functions are called by a simulator to

create one or more instances of the FMU, called models, and to run these models, typically together with other models. An FMU may either be self-integrating (*co-simulation*) or require the simulator to perform the numerical integration (*model-exchange*). The first are sometimes called FMU-CS, while the second are called FMU-ME. See further http://fmi-standard.org/.

Functional Mockup Unit Compiled code or source code that can be executed using the application programming interface defined in the *Functional Mockup Interface* standard.

Functional Verification Tool The Functional Verification Tool is a software that takes as an input the control sequence in CDL, requirements expressed in CDL, a list of I/O connections, and a configuration file, and then tests whether the measured control signals satisfy the requirements, violate them, or whether some requirements remain untested.

G36 Sequence A control sequence specified by ASHRAE Guideline 36. See also control sequence.

HVAC System Any HVAC plant coupled with the control system.

HVAC System Model Consists of all components and connections used to model the behavior of an HVAC System.

Open Building Controls Open Building Controls (OBC) is the name of project that develops open source software for building control sequences and for testing of requirements.

OBC See *Open Building Controls*.

Mode In CDL, by mode we mean a signal that can take on multiple distinct values, such as `On`, `Off`, `PreCool`.

Model-exchange Model-exchange refers to a simulation in which different simulation programs exchange run-time data. A master algorithm sets time, inputs and states, and requests from the simulator the time derivative. The master algorithm integrates the differential equations in time. See also *co-simulation*.

Non-HVAC System Any non-HVAC plant coupled with the control system.

Plant A plant is the physical system that is being controlled by a *controller*. In our context, plant is not only used for example a chiller plant, but also for an HVAC system or an actuated shade.

Standard control sequence A control sequence defined in the CDL control sequence library based on a standard or any other document which contains a full English language description of the implemented sequence.

State event We say that a simulation has a state event if its model changes based on a test that depends on a state variable. For example, for some initial condition $x(0) = x_0$,

$$\frac{dx}{dt} = \begin{cases} 1, & \text{if } x < 1, \\ 0, & \text{otherwise,} \end{cases}$$

has a state event when $x = 1$.

Time event We say that a simulation has a time event if its model changes based on a test that only depends on time. For example,

$$y = \begin{cases} 0, & \text{if } t < 1, \\ 1, & \text{otherwise,} \end{cases}$$

has a time event at $t = 1$.

Chapter 12

Acknowledgments

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, and
- the California Energy Commission's Electric Program Investment Charge (EPIC) Program.

Chapter 13

References

[Bun86] *Steuern und Regeln in der Heizungs- und Lüftungstechnik*. Bundesamt für Konjunkturfragen, Bern, Switzerland, 1986.

[Mod12] *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification, Version 3.3*. Modelica Association, May 2012. URL: https://www.modelica.org/documents/ModelicaSpec33.pdf.

[Ene13] *Energy Star Portfolio Manager – Technical Reference Source Energy*. Energy Star, US Government, July 2013. URL: https://portfoliomanager.energystar.gov/pdf/reference/SourceEnergy.pdf.

[ASHRAE16] *ASHRAE Guideline 36P, High Performance Sequences of Operation for HVAC systems, First Public Review Draft*. ASHRAE, June 2016. URL: http://gpc36.savemyenergy.com/public-files.

[ASH06] ASHRAE. *Sequences of Operation for Common HVAC Systems*. ASHRAE, Atlanta, GA, 2006.

[DFS+11] Michael Deru, Kristin Field, Daniel Studer, Kyle Benne, Brent Griffith, Paul Torcellini, Bing Liu, Mark Halverson, Dave Winiarski, Michael Rosenberg, Mehry Yazdanian, Joe Huang, and Drury Crawley. U.S. Department of Energy commercial reference building models of the national building stock. Technical Report NREL/TP-5500-46861, National Renewables Energy Laboratory, 1617 Cole Boulevard, Golden, Colorado 80401, February 2011.

[KohlerHM+16] Jochen Köhler, Hans-Martin Heinkel, Pierre Mai, Jürgen Krasser, Markus Deppe, and Mikio Nagasawa. Modelica-Association - Project "System Structure and Parameterization" - Early Insights. In *Proc. of the 1st Japanese Modelica Conference*, 35–42. Tokyo, Japan, May 2016. Modelica Association. URL: http://dx.doi.org/10.3384/ecp1612435, doi:DOI:10.3384/ecp1612435.

[RSS05] Hermann Recknagel, Eberhard Sprenger, and Ernst-Rudolf Schramek. *Taschenbuch für Heizung und Klimatechnik*. Number 72. Oldenbourg Industrieverlag, München, 2005. ISBN 3-486-26560-1.

[Ver13] Daniel A. Veronica. Automatically detecting faulty regulation in hvac controls. *HVAC&R Research*, 19(4):412–422, 2013. URL: https://www.tandfonline.com/doi/abs/10.1080/10789669.2013.789369, doi:10.1080/10789669.2013.789369.

[Wet06] Michael Wetter. Multizone airflow model in Modelica. In Christian Kral and Anton Haumer, editors, *Proc. of the 5-th International Modelica Conference*, volume 2, 431–440. Vienna, Austria, September 2006. Modelica Association and Arsenal Research. URL: https://www.modelica.org/events/modelica2006/Proceedings/sessions/Session413.pdf.

[Wet13] Michael Wetter. Fan and pump model that has a unique solution for any pressure boundary condition and control signal. In Jean Jacques Roux and Monika Woloszyn, editors, *Proc. of the 13-th IBPSA Conference*, 3505–3512. 2013. URL: http://simulationresearch.lbl.gov/wetter/download/2013-IBPSA-Wetter.pdf.

[WZNP14] Michael Wetter, Wangda Zuo, Thierry S. Nouidui, and Xiufeng Pang. Modelica Buildings library. *Journal of Building Performance Simulation*, 7(4):253–270, 2014. doi:DOI:10.1080/19401493.2013.765506.

[WZN11] Michael Wetter, Wangda Zuo, and Thierry Stephane Nouidui. Modeling of heat transfer in rooms in the Modelica "Buildings" library. In *Proc. of the 12-th IBPSA Conference*, 1096–1103. International Building Performance Simulation Association, November 2011. URL: http://www.ibpsa.org/.

Index