ERNEST ORLANDO LAWRENCE
BERKELEY NATIONAL LABORATORY

Working Report

https://github.com/lbl-srg/obc

January 14, 2025

Contents

1 Preamble 1
1.1 Purpose of the Document 1

2 Conventions 2

3 Process Workflow 3

4 Use Cases 5
4.1 Controls Design 5
4.1.1 Loading a Standard Sequence from a Library 5
4.1.2 Customizing a Control Sequence for an HVAC System 6
4.1.3 Customizing and Configuring a Control Sequence for a Single-Zone VAV System 7
4.1.4 Customizing and Configuring a Control Sequence for a Multizone VAV System 9
4.1.5 Performance Assessment of a Control Sequence 11
4.1.6 Defining Integration with non-HVAC Systems such as Lighting, Façade and Presence Detection 12
4.2 Bidding and BAS Implementation 14
4.2.1 Generate Control Point Schedule from Sequences 14
4.3 Commissioning, Operation, and Maintenance 15
4.3.1 Conducting Verification Test of a VAV Cooling-Only Terminal Unit 15
4.3.2 As-Built Sequence Generator 16

5 Requirements 17
5.1 Controls Design Tool 17
5.2 CDL 18
5.3 Commissioning and Functional Verification Tool 19

6 Software Architecture 20
6.1 Controls Design Tool 21
6.2 Functional Verification Tool 22

7 Control Description Language 23
7.1 Introduction 23
7.2 Overview of CDL and Terminology 24
7.3 Syntax 27
7.4 Permissible Data Types 27
7.4.1 Data Types 27
7.4.2 Parameters 30

7.4.3 Constants 30
7.4.4 Arrays 31
7.5 Encapsulation of Functionality 31
7.6 Elementary Blocks 32
7.7 Instantiation 33
7.7.1 Parameter Declaration and Assigning of Values to Parameters 33
7.7.2 Functions 34
7.7.3 Evaluation of Assignment of Values to Parameters 35
7.7.4 Conditionally Removing Instances 37
7.7.5 Point list 38
7.8 Connectors 43
7.9 Equations 44
7.10 Connections 44
7.11 Annotations 46
7.12 Composite Blocks 47
7.13 Extension Blocks 48
7.14 Replaceable Blocks 50
7.15 Extension of a Composite Block 51
7.16 Model of Computation 54
7.17 Metadata 55
7.17.1 Inferred Properties 55
7.17.2 Semantic Information 56

8 Control eXchange Format (CXF) 62
8.1 Introduction 62
8.2 Classes and Properties 63
8.3 Generating CXF from an instance of a CDL class 64
8.4 Source of CXF translation 65
8.5 Representing Instances in CXF 65
8.6 Handling Arrays and Expressions 65
8.7 ExtensionBlocks 66

9 Documentation of Control Sequences 67
9.1 Introduction 67
9.2 Editing a Sequence that is Specified in a Word Document 67
9.3 Exporting the Control Logic from a CDL Model 67

10 Controls Library 71
10.1 Introduction 71
10.2 CDL Library 71
10.3 Library of Control Sequences 71

11 Code Generation 74
11.1 Introduction 74
11.2 Challenges and Implications for Translation of Control Sequences from and to Building Control Product Lines 76
11.3 Translation of a Control Sequence using a JSON Intermediate Format 76
11.4 Export of a Control Sequence or a Verification Test using the FMI Standard 80

11.5 Modular Export of a Control Sequence using the FMI Standard for Control Blocks and using the SSP Standard for the Run-time Environment 81
11.6 Replacement of Elementary CDL Blocks during Translation 82
11.6.1 Substitutions that Give Identical Control Response 82
11.6.2 Substitutions that Change the Control Response 82
11.6.3 Adding Blocks that are not in the CDL Library 83

12 Verification 84
12.1 Introduction 84
12.2 Terminology 84
12.3 Scope of the Verification 85
12.4 Methodology 85
12.5 Modules of the Verification Test 86
12.5.1 CSV File Reader 86
12.5.2 Unit Conversion 87
12.5.3 Comparison of Time Series Data 87
12.5.4 Verification of Sequence Diagrams 87
12.6 Example 87
12.7 Specification for Automating the Verification 91
12.7.1 Use Cases 96
12.7.2 Scenario 1: Control Input Obtained by Simulating a CDL Model 96
12.7.3 Scenario 2: Control Input Obtained by Trending a Real Controller 100

13 Generating a Modelica Model from Semantic Model 103
13.1 Workflow 103
13.2 Example 103

14 Example Application 106
14.1 Introduction 106
14.2 Methodology 107
14.2.1 HVAC Model 107
14.2.2 Envelope Heat Transfer 107
14.2.3 Internal Loads 108
14.2.4 Multi-Zone Air Exchange 108
14.2.5 Control Sequences 108
14.2.6 Site Electricity Use 109
14.2.7 Simulations 112
14.3 Performance Comparison 114
14.4 Improvement to Guideline 36 Specification 123
14.4.1 Freeze Protection for Mixed Air Temperature 123
14.4.2 Deadbands for Hard Switches 125
14.4.3 Averaging Air Flow Measurements 125
14.4.4 Cross-Referencing and Modularization 125
14.4.5 Lessons Learned Regarding the Simulations 125
14.5 Discussion and Conclusions 126

15 Glossary 128

16 Acknowledgments 130

17 References 131

Bibliography 132

Chapter 1

Preamble

1.1 Purpose of the Document

This document describes the development of the Control Description Language (CDL) that is being developed within the OpenBuildingControl project. It also describes the process workflow, use cases and requirements, as well as 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.

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 Guideline 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 Control 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). For current control product lines, step (8) involves a translation of CDL to their programming languages, whereas in the future, control providers could build systems that directly use CDL. 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 can be linked to a Modelica envelope model [WZN11] or to an EnergyPlus envelope model. The latter can be done through Spawn of EnergyPlus [WBG+20], which is being developed in a related project at https://lbl-srg.github.io/soep/.

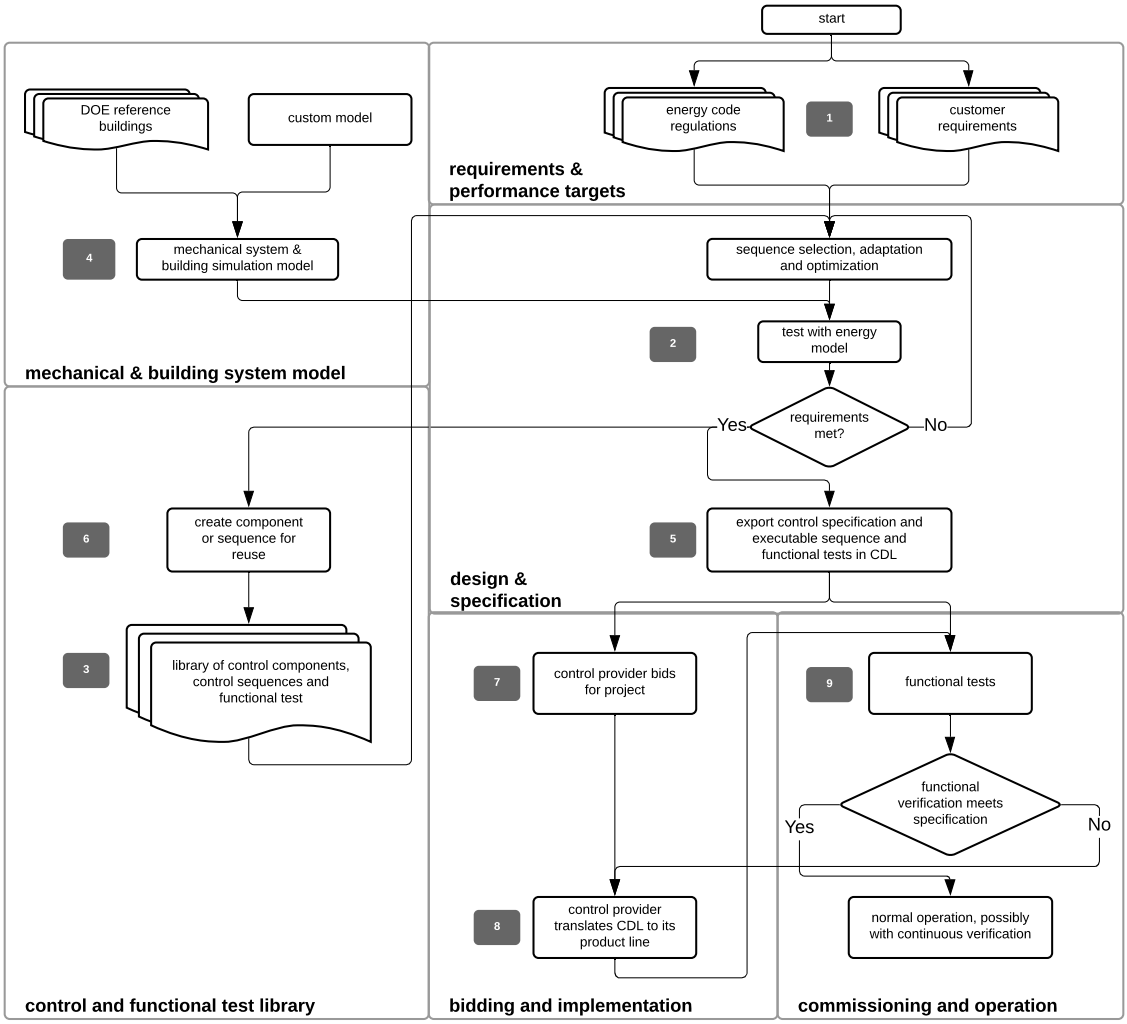


Fig. 3.1: *Process workflow for controls design, specification and functional verification.*

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.

continues 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.

continues 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

continues 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.

continues 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.

continues 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

continues 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.

continues 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.

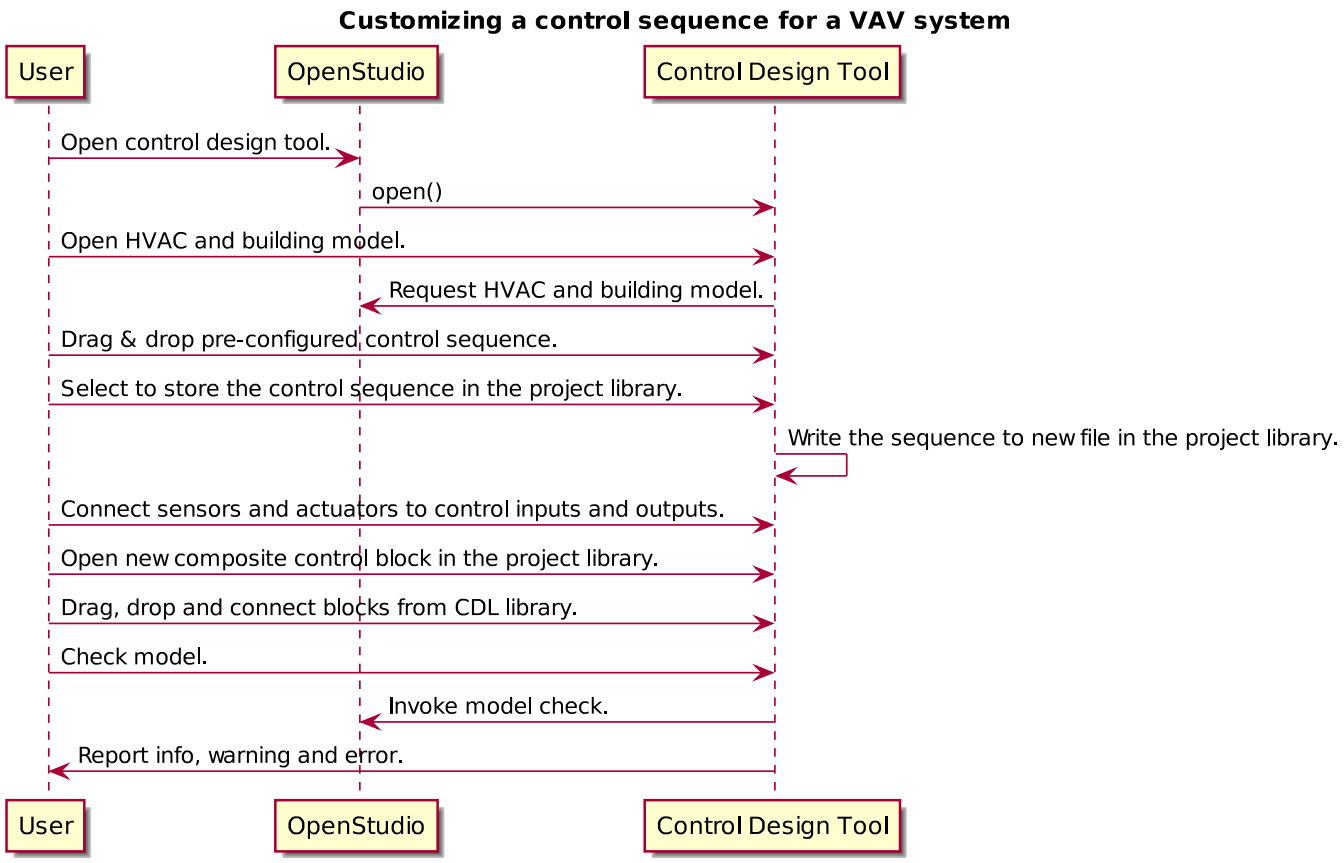


Fig. 4.1: Customizing a control sequence for a VAV system.

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 tagg
Successful End Condition	Control points schedule can be automatically produced by extracting from the sequences, including tagg
Failed End Condition	Control points schedule is inaccurate or doesn't contain sufficient information.
Primary Actors	Mechanical Designer/Consultant
Secondary Actors	Controls contractor

Table 4.7 – continued fro

Use case name	**Generate control points schedule from sequences**
Trigger	
Main Flow	**Action**
1	When a user adds a control point in the controls design tool, the tool provides default values and allows
2	User clicks on a button to generate Points Schedule, an Excel file is then generated listing all the points
3	User clicks on a button to generate a tag list of unique control devices within the project in Excel, so tha

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.
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

continues on next page

Table 4.8 – continued from previous page

Use case name	**Conducting verification test of a VAV Cooling-Only Terminal Unit**
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.
Goal in Context	An owner's facilities engineer wishes to confirm the actual installed controls sequences in an existing bu
Preconditions	Installed control system must be capable of communication with the tool. Translation protocol must be e
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 cor
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 7.5) with inputs and outputs through which blocks can be connected.
4. It shall be possible to compose blocks hierarchically to form new blocks.
5. The elementary building blocks [such as a 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 3.6 [Mod23]. [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. Each distinct piece of equipment [e.g. VAV terminal box controller] 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.

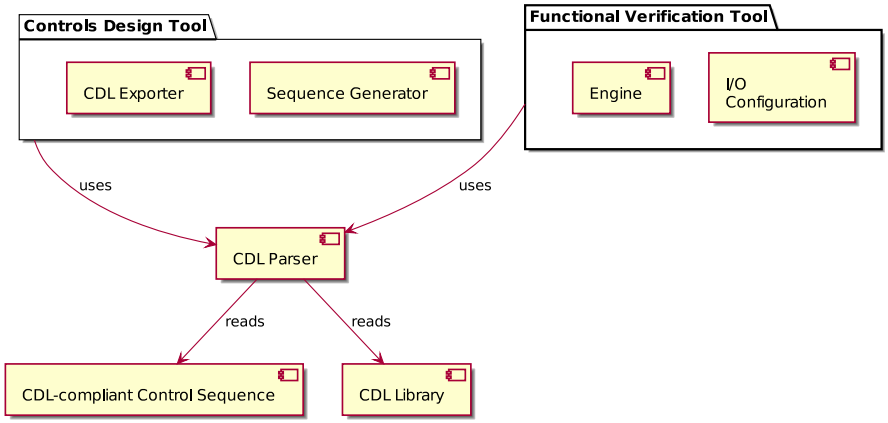


Fig. 6.1: Overall software architecture.

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 https://simulationresearch.lbl.gov/modelica/releases/v10.0.0/help/Buildings_Controls_OBC_ASHRAE.html and the *CDL Library*, see https://simulationresearch.lbl.gov/modelica/releases/v10.0.0/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/articles/spawn-energyplus-spawn 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 OPTIMICA installation.

6.1 Controls Design Tool

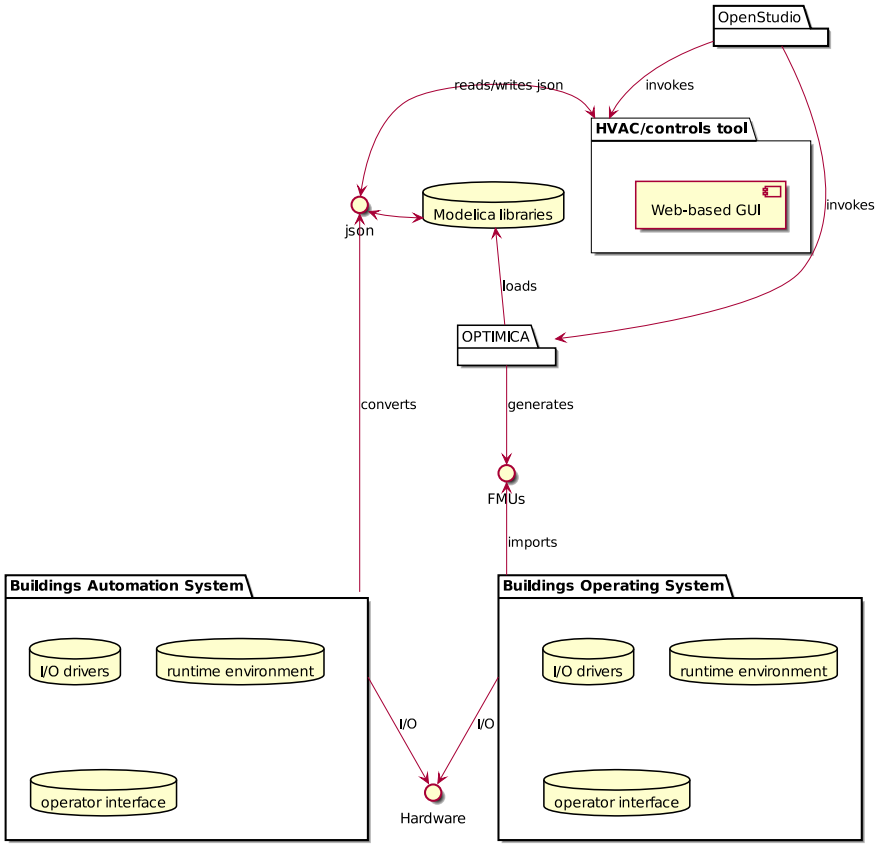


Fig. 6.2: Overall software architecture of the Controls Design Tool.

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 *OPTIMICA* 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 11.

6.2 Functional Verification Tool

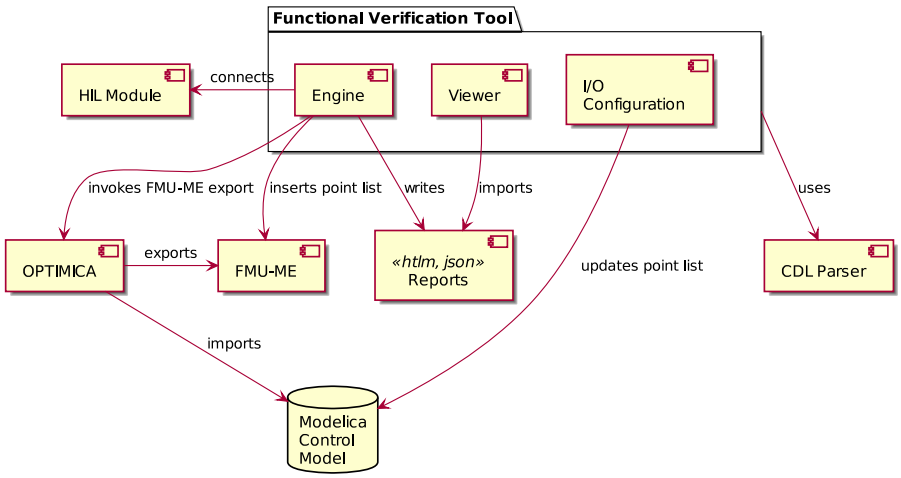


Fig. 6.3: Overall software architecture of the Functional Verification Tool.

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 *OPTIMICA* to export an FMU-ME of the control blocks. As *OPTIMICA* 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

7.1 Introduction

This section specifies the Control Description Language (CDL), a declarative language that can be used to express control sequences using block-diagrams. It is designed in such a way that it can be used to conveniently specify building control sequences in a vendor-independent format, use them within whole building energy simulation, and translate them for use in building control systems.

A key technical challenge encountered when developing CDL was that existing control product lines are heterogeneous. They differ in their functionality for expressing control sequences, in their semantics of how control output gets updated, and in their syntax which ranges from graphical languages to textual languages. Code generation for a variety of products is common in the Electronic Design Automation industry. However, in the Electronic Design Automation industry, engineers write models and controllers are built to conform to the models. If this were to be applied to the buildings industry, then control providers would need to update their product line in order to be able to faithfully comply with the model. We think such costly product line reconfigurations are not reasonable to expect in the next decade. Therefore, for the immediate future, we will need to build digital models of control sequences that can conform to their implementation on target control product lines; while ensuring that as new product lines are being developed, the manufacturers can invert the paradigm and build controllers that conform to the models. We therefore selected the path of designing CDL in such a way that it provide a minimum set of capabilities that can be expected to be supported by current control product lines, while allowing future control product lines to directly use CDL for the implementation of the control sequences. As we have demonstrated with one commercial product, the barrier to translate CDL to the programming language of a current control product line is low.

To put CDL in context, and to introduce terminology, Fig. 7.1 shows the translation of CDL to a control product line or to English language documentation. Input into the translation is CDL. An open-source tool called `modelica-json` translator (see also Section 11.3 and https://github.com/lbl-srg/modelica-json) translates CDL to an intermediate format that we call *CDL-JSON*. From CDL-JSON, further translations can be done to a control product line, or to generate point lists, English language documentation or a semantic model of the control sequences. We anticipate that future control product lines use directly CDL as shown in the right of Fig. 7.1. Such a translation can then be done using various existing Modelica tools to generate code for real-time simulation.

The next sections give an overview and definition of the CDL language. A collection of control sequences that are composed using the CDL language is described in Section 10. These sequences can be simulated with Modelica simulation

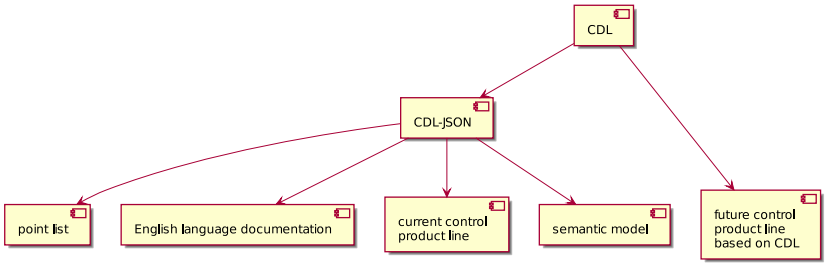


Fig. 7.1: Translation of CDL to the CDL-JSON intermediate format and to a product line, a semantic model or English language documentation.

environments. The translation of such sequences to control product lines using `modelica-json`, or other means of translation, is described in Section 11.

7.2 Overview of CDL and Terminology

CDL is a declarative, modular language for expressing block diagrams that was introduced in [WGH18]. CDL allows hierarchical modeling to encapsulate and reuse, through object instantiation, preconfigured control sequences. CDL also defines syntax for connecting inputs and outputs of blocks and for propagating the values of parameters. CDL allows users to declare new blocks, store them in a library, and instantiate them for use in a control sequence. CDL also has annotations that declare how to graphically render the block diagrams, and to document control sequences.

CDL uses a small subset of the Modelica language that is needed for declaration of block diagrams. We selected Modelica as it is an open standard, as it provides various open-source and commercial modeling and simulation environments, as it allows to generate highly efficient code for simulation, and because it is increasingly used to simulate building energy and control systems.

As the model of computation, CDL uses the synchronous data flow principle and single assignment rule, which is consistent with the Modelica Language Specification [Mod23]. Therefore, all variables keep their value until the value is explicitly changed, values are always present (and hence can be accessed at any time instant), computation and communication at an event instant do not take time, and every input connector must be connected to exactly one output connector.

CDL consists of three types of blocks:

- *Elementary blocks*: These are built-in blocks that cannot be changed by users. All implementations of CDL need to provide the functionality of these blocks. An example is a block that outputs the sum of two inputs.
- *Composite blocks*: These are blocks that are composed hierarchically using elementary blocks or other composite blocks. Composite blocks can be used to declare control sequences, they can be stored in a library for reuse, and they can be instantiated and configured for a particular energy system.
- *Extension blocks*: Extension blocks allow users to implement new blocks that may be difficult or impossible to implement using the rules of composite blocks. For example, an extension block could be used to call a web service, or to implement a finite state machine that rotates chillers in a chiller plant.

The functionality of elementary blocks, but not their implementation, is part of the CDL specification. Thus, in the most general form, elementary 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., $(p, t, u(t), x(t)) \mapsto (y(t), x'(t))$. By the composition rules of composite blocks, composite blocks are also such functions. This abstraction is important as it allows to execute

CDL sequences that are composed of composite blocks using a variety of programming languages, it guarantees that the elementary and composite blocks have a well-defined scope and it guarantees that the calculations of a block have no side effects on other blocks. This however is not necessarily true for extension blocks (for example, two extension blocks could exchange data through a web service, thereby causing one block to have side effects on the behavior of the other block). Thus, use of composite blocks is preferred. To execute extension blocks, extension blocks need to be compiled and implemented using the Functional Mockup Interface Standard to provide run-time interoperability.

The CDL language consists of the following elements:

- A list of elementary 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.
  - how to add new blocks that go beyond the capabilities of composite blocks.
- A model of computation that describes when blocks are executed and when outputs are assigned to inputs.

Table 7.1 gives an overview of the terminology used to describe CDL.

Table 7.1: Main terminology used in CDL. For a more detailed definition, follow the corresponding links.

Term	Description

definition and instantiation	We call the implementation of an object (such as a block or parameter) an *object definition*. To use an object, one declares an *instance* of it.
	For example, the statement
	`block myBlock`
	`CDL.Reals.Sources.Constant c(k=1);`
	`end myBlock;`
	is a block *definition* for *myBlock*, and the second line declares an *instance* of the block *CDL.Reals.Sources.Constant*.

block	A block is an object that has any number of constants, parameters, input connectors, output connectors and instances of other blocks. Blocks typically encapsulate calculations. We distinguish between *elementary blocks*, *composite blocks* and *extension blocks*.
elementary block	An elementary block (Section 7.6) is a block that is part of the CDL library. Elementary blocks are the basic language blocks and are not to be changed by users.
composite block	A composite block (Section 7.12) is a block (and thus can have any number of constants, parameters, input connectors and output connectors) that instantiates any number of other elementary blocks or composite blocks, and declares connections between inputs and outputs. Composite blocks are used to implement control sequences.
extension block	An extension block (Section 7.13) is a block that conforms in CDL to the Modelica definition of a block (and thus can have textual equations, call C functions or functions in a dynamically linked library). In CDL-json, the json specification declares its constants, parameters, input connectors and output connectors, and it declares the file name of a Functional Mockup Unit for Model Exchange that can be used to compute its outputs.
parameter	A parameter (Section 7.4.2) is an instance of a native data type (such as a *Real* or *Integer*) whose value is time invariant, and hence its value cannot be changed based on an input signal. To change its value when simulating a control logic, one would need to stop the simulation and change the value. In an actual controller, one may change the value through a graphical user interface.
constant	A constant (Section 7.4.3) is an instance of a native data type (such as a *Real* or *Integer*) whose value cannot be changed after compilation.
input (output) connector	An input (output) connector (Section 7.8) is an object to which a connection can be made to transfer a signal value into (out of) a block.
connection	A connection (Section 7.10) is used to connect an input connector to an output connector, thereby indicating that the value at the input connector is equal to the value at the output connector.
function	A function (Section 7.7.2) is an object that can have any fixed number of arguments and returns a scalar- or array-valued object, such as a Real number or an Integer array. Functions can be used to assign values to constants and parameters, and to assign values to attributes of constants, parameters, inputs and outputs.
annotation	An annotation (Section 7.11) is a declaration that is used to store information about blocks, input connectors, output connectors and parameters that does not affect the computations. Annotations are used for example to store documentation, to provide a means to group related parameters of a block so they can be shown next to each other in a graphical user interface, or to store semantic information.

7.3 Syntax

In order to use CDL with building energy simulation programs, and to not invent yet another language with new syntax, the CDL syntax conforms to a subset of the Modelica Language Specification [Mod23]. 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 construct 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 (except inside the extension blocks, Section 7.13):

1. `inner` and `outer` for instance hierarchy lookup
2. `break` for component deselection.

Also, the following Modelica language features are not supported in CDL, except inside extension blocks:

1. Clocks for clocked state machines [which are used in Modelica for hybrid system modeling].
2. `algorithm` sections for expressing sequences of statements [because the elementary blocks are black-box models as far as CDL is concerned and thus CDL compliant tools do not parse the `algorithm` section.]
3. `initial equation` and `initial algorithm` sections for system initialization.

7.4 Permissible Data Types

7.4.1 Data Types

This section defines the basic data types. The definition is a subset of Modelica in which we left out attributes that are not needed for CDL.

The attributes that are present in Modelica but not in CDL are marked with `//--`.

[Note the following: The `start` attribute is not needed in CDL because the start value of states is declared through a *parameter*. The `equation` section has been removed because how to deal with variables that are out of limit should be left to the implementation of the control system.]

7.4.1.1 Real Type

The following is the predefined `Real` type:

```
type Real // Note: Defined with Modelica syntax although predefined
  RealType value; // Accessed without dot-notation
  parameter StringType quantity    = "";
  parameter StringType unit        = "" "Unit used in equations";
  parameter StringType displayUnit = "" "Default display unit";
  parameter RealType min=-Inf, max=+Inf; // Inf denotes a large value
  //-- parameter RealType start    = 0; // Initial value
  //-- parameter BooleanType fixed = true,   // default for parameter/constant;
  //--                            = false;  // default for other variables
```

(continues on next page)

(continued from previous page)

```
  parameter RealType nominal = 1;            // Nominal value
  parameter BooleanType unbounded = false;   // For error control
  //-- parameter StateSelect stateSelect = StateSelect.default;
  //-- equation
  //--   assert(value >= min and value <= max, "Variable value out of limit");
end Real;
```

`Real Type/double` matches the IEC 60559:1989 (ANSI/IEEE 754-1985) double format.

The `quantity` attribute is optional, can take on the following values:

- `""`, which is the default, is considered as no quantity being specified.
- `Angle` for area (such as used for sun position).
- `Area` for area.
- `Energy` for energy.
- `Frequency` for frequency.
- `Illuminance` for illuminance.
- `Irradiance` for solar irradiance.
- `MassFlowRate` for mass flow rate.
- `MassFraction` for mass fraction.
- `Power` for power.
- `PowerFactor` for power factor.
- `Pressure` for absolute pressure.
- `PressureDifference` for pressure difference.
- `SpecificEnergy` for specific energy.
- `TemperatureDifference` for temperature difference.
- `Time` for time.
- `ThermodynamicTemperature` for absolute temperature.
- `Velocity` for velocity.
- `VolumeFlowRate` for volume flow rate.

[These quantities are compatible with the quantities used in the Modelica Standard Library, to allow connecting CDL models to Modelica models, see also Section 7.10.]

[The *quantity* attribute could be used for example to declare in a sequence that a real signal is a `AbsolutePressure`. This could be used to aid connecting signals or filtering data. Quantities serve a different purpose than tagged properties (Section 7.17.2).]

The value of `displayUnit` is used as a recommendation for how to display units to the user. [For example, tools that implement CDL may convert the `value` from `unit` to `displayUnit` before showing it in a GUI or a log file. Moreover, tools may have a global list where users can specify, for example, to display `degC` and `K` in `degF`.]

The nominal attribute is meant to be used for scaling purposes and to define tolerances, such as for integrators, in relative terms.

7.4.1.2 Integer Type

The following is the predefined `Integer` type:

```
type Integer // Note: Defined with Modelica syntax although predefined
  IntegerType value; // Accessed without dot-notation
  //-- parameter StringType quantity = "";
  parameter IntegerType min=-Inf, max=+Inf;
  //-- parameter IntegerType start = 0; // Initial value
  //-- parameter BooleanType fixed = true,  // default for parameter/constant;
  //--                            = false; // default for other variables
//-- equation
//--   assert(value >= min and value <= max, "Variable value out of limit");
end Integer;
```

The minimal recommended number range for IntegerType is from −2147483648 to +2147483647, corresponding to a two's-complement 32-bit integer implementation.

[The `quantity` attribute could be used for example to declare in a sequence that a integer signal is a `NumberOfHeatingRequest`. This could be used to aid connecting signals or filtering data.]

7.4.1.3 Boolean Type

The following is the predefined `Boolean` type:

```
type Boolean // Note: Defined with Modelica syntax although predefined
  BooleanType value; // Accessed without dot-notation
//--  parameter StringType quantity = "";
//--  parameter BooleanType start = false; // Initial value
//--  parameter BooleanType fixed = true,  // default for parameter/constant;
//--                             = false, // default for other variables
end Boolean;
```

[The `quantity` attribute could be used for example to declare in a sequence that a boolean signal is a `ChillerOn` command.]

7.4.1.4 String Type

The following is the predefined `String` type:

```
type String // Note: Defined with Modelica syntax although predefined
  StringType value; // Accessed without dot-notation
//--  parameter StringType quantity = "";
//--  parameter StringType start = "";     // Initial value
//--  parameter BooleanType fixed = true,  // default for parameter/constant;
//--                             = false, // default for other variables
end String;
```

7.4.1.5 Enumeration Types

A declaration of the form

```
type E = enumeration([enumList]);
```

defines an enumeration type `E` and the associated enumeration literals of the `enumList`. The enumeration literals shall be distinct within the enumeration type. The names of the enumeration literals are defined inside the scope of `E`. Each enumeration literal in the `enumList` has type `E`.

[Example:

```
type SimpleController = enumeration(P, PI, PD, PID);

parameter SimpleController = SimpleController.P;
```

]

An optional comment string can be specified with each enumeration literal.

[Example:

```
type SimpleController = enumeration(
    P "P controller",
    PI "PI controller",
    PD "PD controller",
    PID "PID controller")
  "Enumeration defining P, PI, PD, or PID simple controller type";
```

]

[Enumerations can for example be used to declare a list of mode of operations, such as `on`, `off`, `startUp`, `coolDown`.]

7.4.2 Parameters

A `parameter` is a value that does not change as time progresses, except through stopping the executation of the control sequence, setting a new value through a user interaction or an API, and restarting the execution. In other words, the value of a `parameter` cannot be changed through an input connector (Section 7.8). Parameters are declared with the `parameter` prefix.

[For example, to declare a proportional gain, use

```
parameter Real k(min=0) = 1 "Proportional gain of controller";
```

]

7.4.3 Constants

A `constant` is a value that is fixed at compilation time. Constants are declared with the `constant` prefix.

[For example,

```
constant Real pi = 3.14159;
```

]

7.4.4 Arrays

Each of these data types, including the elementary blocks, composite blocks, extension blocks and connectors, can be a single instance, one-dimensional array or two-dimensional array (matrix). 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.

Values of arrays may be declared using

- the notation `{x1, x2, ...}`, for example `parameter Integer k[3,2] = {{1, 2}, {3, 4}, {5, 6}}`,
- one or several iterators, for example `parameter Real k[2,3] = {i*0.5+j for i in 1:3, j in 1:2}`,
- a `fill` or `cat` function, see Section 7.7.1.

[For example, to following declarations all assign the array `{1, 2, 3}` to parameters:

```
parameter Real k1[3] = {1, 2, 3};
parameter Real k2[3] = {i for i in 1:3};
parameter Real k3[3] = k1;
parameter Real k4[3] = fill(1, 3) + {0, 1, 2};
parameter Real k5[3] = cat(1, {1}, {2}, {3});
```

The following declaration instantiates two blocks, and sets the value of the parameter `k` to `2` and `3`:

```
MultiplyByParameter mul[2](k={2, 3});
```

]

The size of arrays will be fixed at translation. It cannot be changed during run-time.

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

See the Modelica 3.6 specification Chapter 10 for array notation and these functions.

7.5 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 blocks* (Section 7.6) or *composite blocks* (Section 7.12).

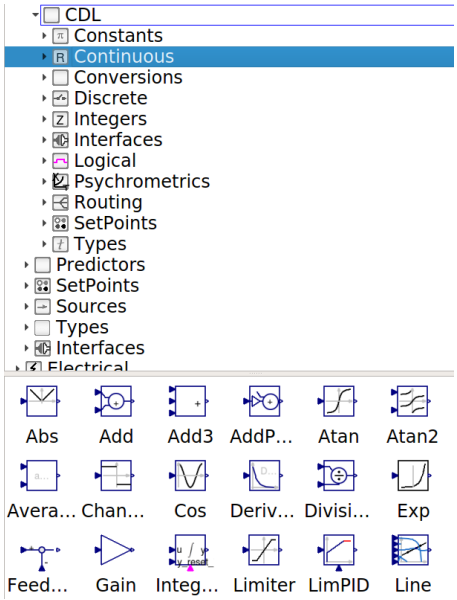


Fig. 7.2: Screenshot of CDL library.

7.6 Elementary Blocks

The CDL library contains elementary blocks that are used to compose control sequences. The functionality of elementary blocks, but not their implementation, is part of the CDL specification. Thus, in the most general form, elementary blocks can be considered as functions that for given parameters p, time t and internal states $x(t)$, map inputs $u(t)$ to new outputs $y(t)$, e.g.,

$$(p, t, u(t), x(t)) \mapsto y(t).$$

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 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 blocks. Rather, users can use the existing elementary blocks to implement composite blocks (Section 7.12).

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

- Open a web browser at https://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 OpenModelica, Impact, or Dymola. All models in the *Examples* and *Validation* packages can be simulated with these tools.

An actual implementation of an elementary 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";

  Interfaces.RealInput u "Connector of Real input signal";
  Interfaces.RealOutput y "Connector of Real output signal";

equation
  y = u + p;

  annotation(Documentation(info("
  <html>
  <p>
  Block that outputs <code>y = u + p</code>,
  where <code>p</code> is parameter and <code>u</code> is an input.
  </p>
  </html>"));

end AddParameter;
```

For the complete implementation, see the github repository.

7.7 Instantiation

7.7.1 Parameter Declaration and Assigning of Values to Parameters

Parameters are values that do not depend on time. The values of parameters can be changed during run-time through a user interaction with the control program (such as to change a control gain), unless a parameter is a *structural parameter*.

The declaration of parameters and their values is identical to Modelica, but we limit the type of expressions that are allowed in such assignments. In particular, for `Boolean` parameters, we allow expressions involving `and`, `or` and `not` and the function `fill(..)` in Table 7.2. For `Real` and `Integer`, expressions are allowed that involve

- the basic arithmetic functions +, -, *, -,
- the relations >, >=, <, <=, ==, <>,
- calls to the functions listed in Table 7.2.

[For example, to instantiate a block that multiplies its input by a parameter, one would write

```
CDL.Reals.MultiplyByParameter gai(k=-1) "Constant gain of -1" annotation(...);
```

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

7.7.2 Functions

CDL provide built-in functions that can be used when assigning values of parameters and attributes of constants, parameters, inputs, and outputs. Table 7.2 lists the supported functions.

Table 7.2: Functions that are allowed in parameter assignments. The functions are consistent with Modelica 3.6.

Function	Description
`abs(v)`	Absolute value of `v`.
`sign(v)`	Returns `if v>0 then 1 else if v<0 then -1 else 0`.
`sqrt(v)`	Returns the square root of `v` if `v >=0`, or an error otherwise.
`div(x, y)`	Returns the algebraic quotient `x/y` with any fractional part discarded (also known as truncation toward zero). [Note: this is defined for `/` in C99; in C89 the result for negative numbers is implementation-defined, so the standard function `div()` must be used.]. Result and arguments shall have type `Real` or `Integer`. If either of the arguments is `Real` the result is `Real` otherwise it is `Integer`.
`mod(x, y)`	Returns the integer modulus of `x/y` , i.e. `mod(x,y)=x-floor(x/y)*y`. Result and arguments shall have type `Real` or `Integer`. If either of the arguments is `Real` the result is `Real` otherwise it is `Integer`. [Examples are `mod(3,1.4)=0.2`, `mod(-3,1.4)=1.2` and `mod(3,-1.4)=-1.2`.]
`rem(x,y)`	Returns the integer remainder of `x/y`, such that `div(x,y)*y + rem(x, y) = x`. Result and arguments shall have type `Real` or `Integer`. If either of the arguments is `Real` the result is `Real` otherwise it is `Integer`. [Examples are `rem(3,1.4)=0.2` and `rem(-3,1.4)=-0.2`.]
`ceil(x)`	Returns the smallest integer not less than `x`. Result and argument shall have type `Real`.
`floor(x)`	Returns the largest integer not greater than `x`. Result and argument shall have type `Real`.
`integer(x)`	Returns the largest integer not greater than `x`. The argument shall have type Real. The result has type Integer.
`min(A)`	Returns the least element of array expression `A`.
`min(x, y)`	Returns the least element of the scalars `x` and `y`.
`max(A)`	Returns the greatest element of array expression `A`.
`max(x, y)`	Returns the greatest element of the scalars `x` and `y`.
`sum(...)`	The expression `sum( e(i, ..., j) for i in u, ..., j in v)` returns the sum of the expression `e(i, ..., j)` evaluated for all combinations of `i` in `u, ..., j in v: e(u[1], ... , v[1]) + e(u[2], ... ,v[1])+... +e(u[end],... ,v[1])+...+e(u[end],... ,v[end])` The type of `sum(e(i, ..., j) for i in u, ..., j in v)` is the same as the type of `e(i,...j)`.
`fill(s, n1, n2, ...)`	Returns the $n_1 \times n_2 \times n_3 \times$ ... array with all elements equal to scalar or array expression `s` ($n_i \geq 0$). The returned array has the same type as `s`. Recursive definition: `fill(s, n1, n2, n3, ...) = fill( fill(s, n2, n3, ...), n1);`, `fill(s,n)={s, s, ..., s}` The function needs two or more arguments; that is `fill(s)` is not legal.
`size(...)`	Returns dimensions of an array. For $1 \leq i \leq n$, where n is the number of dimensions in `A`, the expression `size(A,i)` returns the size of dimension i of array expression `A`. The expression `size(A)` returns a vector of length n containing the dimension sizes of `A`. [Examples are `size([1, 2, 3; 3, -4, 5], 1)=2` and `size([1, 2, 3; 3, -4, 5])={2,3}`.]

[For example, if a controller has a parameter for the set point for the outdoor air flow rate of ten (equally sized) zones that needs to be set to 0.1 m^3/s, a declaration may look like:

```
parameter Real VSet_flow[10](
 final unit=fill("m3/s", 10)) = fill(0.1, 10);
```

]

7.7.3 Evaluation of Assignment of Values to Parameters

Using expressions in parameter assignments, and propagating values of parameters in a hierarchical formulation of a control sequence, are convenient language constructs to express relations between parameters. However, most of today's building control product lines do not support propagation of parameter values and evaluation of expressions in parameter assignments. For CDL to be compatible with this limitation, the `modelica-json` translator has optional flags, described below, that trigger the evaluation of propagated parameters, and that evaluate expressions that involve parameters.

CDL also has a keyword called `final` that prevents a declaration from being changed by the user. This can be used in a hierarchical controller to ensure that parameter values are propagated to lower level controller in such a way that users can only change their value at the top-level location. It can also be used in CDL to enforce that different instances of blocks have the same parameter value. For example, if a controller samples two signals, then `final` could be used to ensure that they sample at the same rate. However, most of today's building control product lines do not support such a language construct. Therefore, while the CDL translator preserves the `final` keyword in the `CDL-JSON` format, a translator from `CDL-JSON` to a control product line is allowed to ignore this declaration.

Note: People who implement control sequences that require that values of parameters are identical among multiple instances of blocks must use blocks that take these values as an input, rather than rely on the `final` keyword. This could be done as explained in these two examples:

Example 1: If a controller has two samplers called `sam1` and `sam2` and their parameter `samplePeriod` must satisfy `sam1.samplePeriod = sam2.samplePeriod` for the logic to work correctly, then the controller can be implemented using CDL.Logical.Sources.SampleTrigger and connect its output to two instances of CDL.Discrete.TriggeredSampler that sample the corresponding signals.

Example 2: If a controller normalized two input signals by dividing it by a gain `k1`, then rather than using two instances of CDL.Reals.MultiplyByParameter with parameter `k = 1/k1`, one could use a constant source CDL.Reals.Sources.Constant with parameter `k=k1` and two instances of CDL.Reals.Divide, and then connect the output of the constant source with the inputs of the division blocks.

We will now describe how assignments of values to parameters can optionally be evaluated by the CDL translator. While such an evaluation is not preferred, it is allowed in CDL to accommodate the situation that most building control product lines, in contrast to modeling tools such as Modelica, Simulink or LabVIEW, do not support the propagation of parameters, nor do they support the use of expressions in parameter assignments.

Consider the statement

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

(continues on next page)

(continued from previous page)

```
CDL.Reals.Sources.Constant con(
  k = pRel) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = pRel-25,
  uHigh = pRel+25) "Hysteresis for fan control";
```

Some building control product lines will need to evaluate this at translation because they cannot propagate parameters and/or cannot evaluate expressions.

To lower the barrier for the development of a CDL translator to a control product line, the `modelica-json` translator has two flags. One flag, called `evaluatePropagatedParameters` will cause the translator to evaluate the propagated parameter, leading to a CDL-JSON declaration that is equivalent to the declaration

```
CDL.Reals.Sources.Constant con(
  k(unit="Pa") = 50) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = 50-25,
  uHigh = 50+25) "Hysteresis for fan control";
```

Note

1. the `parameter Real pRel(unit="Pa") = 50` has been removed as it is no longer used anywhere.
2. the parameter `con.k` has now the `unit` attribute set as this information would otherwise be lost.
3. the parameter `hys.uLow` has the unit *not* set because the assignment involves an expression. As expressions can be used to convert a value to a different unit, the unit will not be propagated if the assignment involves an expression.

Another flag called `evaluateExpressions` will cause all mathematical expressions to be evaluated, leading to a CDL-JSON declaration that is equivalent to the CDL declaration

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

CDL.Reals.Sources.Constant con(
  k = pRel) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = 25,
  uHigh = 75) "Hysteresis for fan control";
```

If both `evaluatePropagatedParameters` and `evaluateExpressions` are set, the result would be equivalent of the declaration

```
CDL.Reals.Sources.Constant con(
  k(unit="Pa") = 50) "Block producing constant output";
CDL.Logical.Hysteresis hys(
  uLow  = 25,
  uHigh = 75) "Hysteresis for fan control";
```

Clearly, use of these flags is not preferred, but they have been introduced to accomodate the capabilities that are present in most of today's building control product lines.

Note: A commonly used construct in control sequences is to declare a `parameter` and then use the parameter once to assign the value of a block in this sequences. In CDL, this construct looks like

```
parameter Real pRel(unit="Pa") = 50 "Pressure difference across damper";
CDL.Reals.Sources.Constant con(k = pRel) "Block producing constant output";
```

Note that the English language sequence description would typically refer to the parameter `pRel`. If this is evaluated during translation due to the `evaluatePropagatedParameters` flag, then `pRel` would be removed as it is no longer used. Hence, such a translation should then rename the block `con` to `pRel`, e.g., it should produce a sequence that is equivalent to the CDL declaration

```
CDL.Reals.Sources.Constant pRel(k = 50) "Block producing constant output";
```

In this way, references in the English language sequence to `pRel` are still valid.

7.7.4 Conditionally Removing Instances

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

This allows, for example, to have an implementation of a controller that optionally takes as an input the number of occupants in a zone.

An example code snippet is

```
parameter Boolean have_occSen=false
  "Set to true if zones have occupancy sensor";

CDL.Interfaces.IntegerInput nOcc if have_occSen
  "Number of occupants"
    annotation (__cdl(default = 0));

CDL.Reals.MultiplyByParameter gai(
  k = VOutPerPer_flow) if have_occSen
    "Outdoor air per person";
equation
connect(nOcc, gai.u);
```

By the Modelica language definition, all connections (Section 7.10) to `nOcc` will be removed if `have_occSen = false`.

Some building automation systems do not allow to conditionally removing instances of blocks, inputs and outputs, and their connections. Rather, these instances are always present, and a value for the input must be present. To accomodate this case, every input connector that can be conditionally removed can declare a default value of the form `__cdl(default = value)`, where `value` is the default value that will be used if the building automation system does not support conditionally removing instances. The type of `value` must be the same as the type of the connector. For `Boolean` connectors, the allowed values are `true` and `false`.

If the `__cdl(default = value)` annotation is absent, then the following values are assumed as default:

- For RealInput, the default values are:
  - If `unit=K`: If `quantity="TemperatureDifference"`, the default is 0 K, otherwise it is 293.15 K.
  - If `unit=Pa`: If `quantity="PressureDifference"`, the default is 0 Pa, otherwise it is 101325 Pa.
  - For all other units, the default value is 0.
- For `IntegerInput`, the default value is 0.
- For `BooleanInput`, the default value is `false`.

Note that output connectors must not have a specification of a default value, because if a building automation system cannot conditionally remove instances, then the block (or input connector) upstream of the output will always be present (or will have a default value).

7.7.5 Point list

From CDL-conforming sequences, point lists can be generated. [This could be accomplished using the `modelica-json` tool, see Fig. 7.1.]

For point lists,

- the connectors `RealInput` and `IntegerInput` are analog inputs.
- the connectors `RealOutput` and `IntegerOutput` are analog outputs.
- the connectors `BooleanInput` and `BooleanOutput` are digital inputs and outputs.

7.7.5.1 Annotations that Cause Point Lists to be Generated

The vendor annotation `__cdl(generatePointlist=Boolean, controlledDevice=String)` at the class level specifies that a point list of the sequence is generated. If not specified, it is assumed that `__cdl(generatePointlist=false)`. The key `controlledDevice` is optional. It can be used to list the device that is being controlled. Its value will be written to the point list, but not used otherwise, see Table 7.3 for an example.

When instantiating a block, the `__cdl(generatePointlist=Boolean)` annotation can also be added to the instantiation clause, and it will override the class level declaration.

[For example,

```
block A
  MyController con1;
  MyController con2 annotation(__cdl(generatePointlist=false));
  annotation(__cdl(generatePointlist=true));
end A;
```

generates a point list for `A.con1` only, while

```
block A
  MyController con1;
  MyController con2 annotation(__cdl(generatePointlist=true));
  annotation(__cdl(generatePointlist=false));
end A;
```

generates a point list for `A.con2` only.]

The `generatePointlist` annotation can be propagated down in a composite block (see Section 7.12) by specifying in the instantiation clause the annotation

```
__cdl(propagate(instance="subCon1", generatePointlist=true))
```

Controllers deeper in the hierarchy are referred to using the dot notation, such as in `instance="subCon1.subSubCon1"` where `subSubCon1` is an instance of an elementary or composite block in `subCon1`.

The value of `instance=` must be an elementary block (see Section 7.6) or a composite block (see Section 7.12). It must declared, but it can be conditionally removed (see Section 7.7.4), in which case the declaration can safely be ignored.

Higher-level declarations override lower-level declarations.

[For example, assume `con1` has a block called `subCon1`. Then, the declaration

```
MyController con1 annotation(__cdl(propagate(instance="subCon1", generatePointlist=true)));
```

sets `generatePointlist=true` in the instance `con1.subCon1`.]

There can be any number of `propagate(...)` annotations for a controller. [Specifying multiple `propagate(...)` annotations is useful for composite controllers. For example,

```
MyController con1 annotation(
  __cdl(
    propagate(instance="subCon1",             generatePointlist=true),
    propagate(instance="subCon1.subSubCon1", generatePointlist=true),
    propagate(instance="subCon1.subSubCon2", generatePointlist=false)
  )
);
```

allows a fine grained propagation to individual blocks of a composite block.]

7.7.5.2 Annotations for Connectors

Connectors (see Section 7.8) can have a vendor annotation of the form

```
__cdl(connection(hardwired=Boolean))
```

The field `hardwired` specifies whether the connection should be hardwired or not, the default value is `false`.

Connectors can also have a vendor annotation of the form

```
__cdl(trend(interval=Real, enable=Boolean))
```

The field `interval` must be specified and its value is the trending interval in seconds. The field `enable` is optional, with default value of `true`, and it can be used to overwrite the value used in the sequence declaration.

Similar to `generatePointlist`, the `connection` and `trend` annotations can be propagated. If a composite block contains a block `con1`, which in turn contains a block `subCon1` that has an input `u`, the declaration

```
MyController con1 annotation(
  __cdl(propagate(instance="subCon1.u", connection(hardwired=Boolean)));
```

can be used to set the type of connection of input (or output) `con1.subCon1.u`. The value assigned to `instance` must be the instance name of a connector.

Similarly, the declaration

```
MyController con1 annotation(
  __cdl(propagate(instance="subCon1.u", trend(interval=Real, enable=Boolean)));
```

can be used to set how to trend that input (or output).

These statements can also be combined into

```
MyController con1 annotation(
  __cdl(propagate(instance="subCon1.u", connection(hardwired=Boolean),
                                        trend(interval=Real, enable=Boolean)));
```

As in Section 7.7.5.1,

- the value assigned to `instance` must be the name of an instance that exist, (but it can be conditionally removed in which case the annotation can be ignored),
- higher-level declarations override lower-level declarations, and
- any number of `propagate(...)` annotations can be present.

[For example, consider the pseudo-code

```
block Controller

   Interfaces.RealInput u1
     annotation(__cdl(connection(hardwired=true), trend(interval=60, enable=true)));
   Interfaces.RealInput u2
     annotation(__cdl(connection(hardwired=false),
                      trend(interval=120, enable=true),
                      propagate(instance="con1.u1",
                        connection(hardwired=false),
                        trend(interval=120, enable=true))));

   MyController con1 annotation(__cdl(generatePointlist=true));
   MyController con2 annotation(__cdl(generatePointlist=false,
                                      propagate(instance="subCon1", generatePointlist=true),
                                      propagate(instance="subCon2", generatePointlist=true)));

equation
   connect(u1, con1.u1);
   connect(u2, con1.u2);
   connect(u1, con2.u1);
   connect(u2, con2.u2);
```

(continues on next page)

(continued from previous page)

```
   annotation(__cdl(generatePointlist=true));
end Controller;

...

block MyController
   Interfaces.RealInput u1
     annotation(__cdl(connection(hardwired=false), trend(interval=120, enable=true)));
   Interfaces.RealInput u2
     annotation(__cdl(connection(hardwired=true),  trend(interval=60,  enable=true)));
   ...
   SubController1 subCon1;
   SubController2 subCon2;
   ...
   annotation(__cdl(generatePointlist=true));
end MyController;
```

The translator will generate an annotation propagation list as shown below. There will be point list for `Controller`, `Controller.con1`, `Controller.con2.subCon1` and `Controller.con2.subCon1`. Also, the annotation `connection(hardwired=true), trend(interval=60, enable=true)` of `con1.u2` will be overridden as `connection(hardwired=false), trend(interval=120, enable=true)`.

```
[
  {
    "className": "Controller",
    "points": [
      {
        "name": "u1",
        "hardwired": true,
        "trend": {
          "enable": true,
          "interval": 60
        }
      },
      {
        "name": "u2",
        "hardwired": false,
        "trend": {
          "enable": true,
          "interval": 120
        }
      }
    ]
  },
```

(continues on next page)

(continued from previous page)

```
  {
    "className": "Controller.con1",
    "points": [
      {
        "name": "u1",
        "hardwired": false,
        "trend": {
          "enable": true,
          "interval": 120
        }
      },
      {
        "name": "u2",
        "hardwired": false,
        "trend": {
          "enable": true,
          "interval": 120
        }
      }
    ]
  },
  {
    "className": "Controller.con2.subCon1",
    "points": [
      ...
    ]
  },
  {
    "className": "Controller.con2.subCon2",
    "points": [
      ...
    ]
  }
]
```

]

[For an example of a point list generation, consider the pseudo-code shown below.

```
within Buildings.Controls.OBC.ASHRAE.G36G36.TerminalUnits.Reheat
block Controller "Controller for room VAV box with reheat"
   ...;
   CDL.Interfaces.BooleanInput uWin "Windows status"
      annotation (__cdl(connection(hardwired=true),
                       trend(interval=60, enable=true)));
   CDL.Interfaces.RealOutput yVal "Signal for heating coil valve"
```

(continues on next page)

(continued from previous page)

```
      annotation (__cdl(connection(hardwired=false),
                        trend(interval=60, enable=true)));
   ...
annotation (__cdl(generatePointlist=true, controlledDevice="Terminal unit"));
```

It specifies that a point list should be generated for the sequence that controls the system or equipment specified by `controlledDevice`, that `uWin` is a digital input point that is hardwired, and that `yVal` is an analog output point that is not hardwired. Both of them can be trended with a time interval of 1 minute. The point list table will look as shown in Table 7.3.

Table 7.3: Sample point list table generated by the `modelica-json` tool.

System/Equipment	Name	Type	Hardwired?	Trend [s]	Description
Terminal unit	`uWin`	DI	Yes	60	Windows status
Terminal unit	`yVal`	AO	No	60	Signal for heating coil valve
...	...	...	...	...	...

]

7.8 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`, `IntegerInput`, `IntegerOutput`, `RealInput` and `RealOutput`.

Connectors must be in a `public` section.

Connectors can carry scalar variables, vectors or arrays of values (each having the same data type). 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";
```

]

Note: In general, today's building control product lines only support scalar variables on graphical connections. This leads to the situation that different control sequences need to be implemented for any combination of equipment. For example, if only scalars are allowed in connections, then a chiller plant with two chillers needs a different sequence than a chiller plant with three chillers. With vectors, however, one sequence can be implemented for chiller plants with any number of chillers. This is currently done when implementing sequences from ASHRAE RP-1711 in CDL.

If control product lines do not support vectors on connections, then during translation from CDL to the control product line, the vectors (or arrays) can be flattened. For example, blocks of the form

```
parameter Integer n = 2 "Number of blocks";
CDL.Reals.Sources.Constant con[n](k={1, 2});
CDL.Reals.MultiSum mulSum(nin=n); // multiSum that contains an input connector u[nin]
equation
connect(con.y, mulSum.u);
```

could be translated to the equivalent of

```
CDL.Reals.Sources.Constant con_1(k=1);
CDL.Reals.Sources.Constant con_2(k=1);
CDL.Reals.MultiSum mulSum(nin=2);
equation
connect(con_1.y, mulSum.u_1);
connect(con_2.y, mulSum.u_2);
```

E.g., two instances of `CDL.Reals.Sources.Constant` are used, the vectorized input `mulSum.u[2]` is flattened to two inputs, and two separate connections are instantiated. This will preserve the control logic, but the components will need to be graphically rearranged after translation.

7.9 Equations

After the instantiations (Section 7.7), a keyword `equation` must be present to introduce the equation section. The equation section can only contain connections (Section 7.10) and annotations (Section 7.11).

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 block.)

7.10 Connections

Connections connect input to output connector (Section 7.8). For scalar connectors, each input connector of a block needs to be connected to exactly one output connector of a block. For vectorized connectors, or vectorized instances with scalar connectors, each (element of an) input connector needs to be connected to exactly one (element of an) output connector.

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 7.11). 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

```
CDL.Reals.Max maxValue "Output maximum value";
CDL.Reals.MultiplyByParameter gain(k=60) "Gain";

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

]

Only connectors that carry the same data type (Section 7.4.1) can be connected.

Attributes of the variables that are connected are handled as follows:

- If the `quantity`, `unit`, `min` or `max` attributes are set to a non-default value for both connector variables, then they must be equal.
- If only one of the two connector variables declares the `quantity`, `unit`, `min` or `max` attribute, then this value is applied to both connector variables.
- If two connectors have different values for the `displayUnit` attribute, then either can be used. [It is a quality of the implementation that a warning is issued if declarations are inconsistent. However, because `displayUnit` does not affect the computations in the sequence, the connection is still valid.]

[For example,

```
Reals.Max maxValue(y(unit="m/s")) "Output maximum value";
Reals.MultiplyByParameter gain(                    k=60) "Gain";
Reals.MultiplyByParameter gainOK(   u(unit="m/s" ), k=60) "Gain";
Reals.MultiplyByParameter gainWrong(u(unit="kg/s"), k=60) "Gain";

equation
  connect(gain.u,      maxValue.y); // This sets gain.u(unit="m/s")
                                    // as gain.u does not declare its unit
  connect(gainOK.u,    maxValue.y); // Correct, because unit attributes are consistent
  connect(gainWrong.u, maxValue.y); // Not allowed, because of inconsistent unit attributes
```

]

Signals shall be connected using a `connect` statement; assigning the value of a signal in the instantiation of the output connector 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;
  Reals.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;
```

]

7.11 Annotations

Annotations follow the same rules as described in the following Modelica 3.6 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 3.6 Specification), such as to specify default value of connector as below.]

CDL also uses annotations to declare default values for conditionally removable input connectors, see Section 7.7.4.

For CDL implementations of sources such as ASHRAE Guideline 36, any instance, such as a parameter, input or output, that is not provided in the original documentation shall be annotated. For instances, the annotation is `__cdl(InstanceInReference=false)` while for parameter values, the annotation is `__cdl(ValueInReference=false)`. For both, if not specified the default value is `true`.

[A specification may look like

```
parameter Real anyOutOfScoMult(
  final unit = "1",
  final min = 0,
  final max = 1)=0.8
  "Outside of G36 recommended staging order chiller type SPLR multiplier"
  annotation(Evaluate=true, __cdl(ValueInReference=false));
```

]

Note: This annotation is not provided for parameters that are in general not specified in the ASHRAE Guideline 36, such as hysteresis deadband, default gains for a controller, or any reformulations of ASHRAE parameters that are needed for sequence generalization, for instance a matrix variable used to indicate which chillers are used in each stage.

7.12 Composite Blocks

A composite block is a block that is composed of any number of instances of

- constants,
- parameters,
- input connectors,
- output connectors,
- elementary blocks, and
- other composite blocks.

Composite blocks also contain an `equation` section in which connections are instantiated to connect inputs connectors and output connectors of the composite block and its elementary and composite blocks. These rules allow the definition of composite blocks in a library, and the instantiation and possible configuration of these instances to implement a particular control sequence.

A simple example of a composite block that multiplies one of its inputs, adds it to the other input and produces at its output connector the sum is shown in Fig. 7.3.

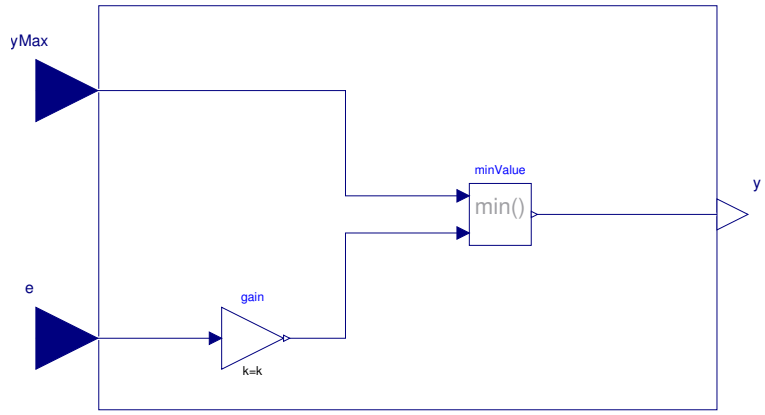


Fig. 7.3: Example of a composite control block that outputs $y = \min(k\, e, y_{max})$ where k is a parameter.

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. 7.3

```
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"
```

(continues on next page)

(continued from previous page)

```
    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.Reals.MultiplyByParameter gain(final k=k) "Constant gain"
    annotation (Placement(transformation(extent={{-60,-50},{-40,-30}})));

  CDL.Reals.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.]

7.13 Extension Blocks

To support functionalities that cannot, or may be hard to, implement with a composite block, *extension blocks* are introduced.

Note: Extension blocks are introduced to allow implementation of blocks that contain statistical functions such as for regression, fault detection and diagnostics methods, or state machines for operation mode switches, as well as proprietary code.

Extension blocks are also suited to propose new elementary blocks for later inclusion in ASHRAE Standard 231P. In fact, elementary blocks are implemented using extension blocks, except that the annotation `__cdl(extensionBlock=true)` (see below) is not present because tools can recognize them because they are stored in the `CDL` package.

In CDL, extension blocks must have the annotations

```
annotation(__cdl(extensionBlock=true))
```

This annotation allows translators to recognize them as extension blocks. Extension blocks are equivalent to the class `block` in Modelica. Thus, extension blocks can contain any declarations that are allowed in a Modelica `block`.

Note: The fact that extension blocks allow any declaration that is allow in a Modelica `block` implies that extension blocks can have any number of parameters, inputs and outputs, identical to composite blocks. It also implies that extension blocks can be used to

- call code, for example in C or from a compiled library,
- import a Functional Mockup Unit that may contain a process model or a fault detection and diagnostics method, and
- implement state machines.

For example, the demand response client Buildings.Controls.DemandResponse.Client would be an extension block if it were to contain the annotation `__cdl(extensionBlock=true)`, as would the Kalman filter that is used in the Example Buildings.Utilities.IO.Python_3_8.Examples.KalmanFilter.

Translation of an extension block to json must reproduce the following:

- All public parameters, inputs and outputs.
- A Functional Mockup Unit for Model Exchange or for Co-simulation, version 2.0, with the file name being the full class name and the extension being `.fmu`.

Note: With OpenModelica 1.20.0, a Functional Mockup Unit for Model Exchange 2.0 of an extension block can be generated with the commands:

```
echo "loadFile(\"Buildings/package.mo\");" > translate.mos
echo "translateModelFMU(Buildings.Controls.OBC.CDL.Reals.PID);" >> translate.mos
omc translate.mos
```

This will generate the fmu `Buildings.Controls.OBC.CDL.Reals.PID.fmu`.

7.14 Replaceable Blocks

CDL allows the use of the Modelica `replaceable`, `constrainedby` and `redeclare` keywords.

The `replaceable` keyword allows to replace a block by another block when translating a composite block.

To declare a block as `replaceable`, the syntax is

```
replaceable ClassName instanceName comment annotation;
```

where `ClassName` is the name of the class, `instanceName` is the name of the instance, and `comment` and `annotation` are optional comments or annotations.

Optionally, the `constrainedby` keyword can be added after `instanceName` to constrain what blocks can be used when redeclaring the replaceable block. The declaration is then

```
replaceable ClassName instanceName constrainedby NameOfConstrainingClass parameterBindings␣
↪comment annotation;
```

where `NameOfConstrainingClass` is the name of the constraining class, and `parameterBindings` is optional and can be used to assign parameters, with or without the `final` keyword.

[For example, consider a composite block that has a PID controller. Suppose the developer of the composite block uses its custom PID controller called `MyPID`, and the developer wants to allow a user of the composite block to replace the PID controller with any custom PID controller, as long as it provides the inputs, outputs, and parameters of the elementary block of the PID controller `CDL.Reals.PID`.

Then, the composite block can be implemented as

```
block SomeCompositeBlock "A composite block in a library"
...
parameter Real k = 2 "Proportional gain";
replaceable MyPID con constrainedby CDL.Reals.PID(
 k=k)
 "PID controller";
...
end SomeCompositeBlock;
```

Because of the `constrainedby` clause, a user of the composite block can replace the controller `MyPID` with any other PID controller that also provides the inputs, outputs, and parameters that are present in `CDL.Reals.PID`. Moreover, the assignment `k=k` will also be applied when the controller is redeclared. Such a redeclaration in which a block `MyPreferredPID` is used for the instance `con` can be done using

```
block SomeCompositeBlock "A composite block in a library"

   parameter Real k = 2 "Proportional gain";

   replaceable Buildings.Controls.OBC.CDL.Reals.PID conPID
     constrainedby Buildings.Controls.OBC.CDL.Reals.PID(k=k)
     "PID controller"
```

(continues on next page)

(continued from previous page)

```
     annotation(
    Placement(transformation(extent = {{-10, -10}, {10, 10}})));

   annotation(
    uses(Buildings(version = "12.0.0")));

end SomeCompositeBlock;
```

In a `redeclare` statement, any parameters can be assigned, for example by writing `redeclare MyPreferredPID conPID(Ti=60)`, which sets the parameter `Ti` to `60`.

The `constrainedby` keyword can also be used to allow use of a block that has other parameters or inputs. A simple example is

```
package ReplaceableExample
  block ReplaceableBlock
    replaceable Buildings.Controls.OBC.CDL.Reals.Sources.Constant con(k=1)
      constrainedby Buildings.Controls.OBC.CDL.Reals.Sources.CivilTime
     "Replaceable block, constrained by a block that imposes as a requirement
     that the redeclaration provides a block with output y (but no parameter k is needed)";
  end ReplaceableBlock;

  block MyNewBlock "Composite block, with sou replaced by a Pulse with period=0.1"
    ReplaceableBlock repBlo(
      redeclare Buildings.Controls.OBC.CDL.Reals.Sources.Pulse con(period=0.1));
  end MyNewBlock;
  annotation (
    uses(Buildings(version = "11.0.0")));
end ReplaceableExample;
```

In the above code, the `constrainedby` keyword specifies the block `CivilTime`. As `CivilTime` has only a `RealOutput` called `y`, but no parameters or inputs, the `Constant` block can be replaced by a `Pulse` block, although `Pulse` has no parameter `k`. Without the `constrainedby CDL.Reals.Sources.CivilTime` clause, `Pulse` could not have been used as it has no parameter `k`.

]

When translating CDL to CXF, the keywords `replaceable`, `constrainedby` and `redeclare` need to be evaluated and removed. E.g., they are not present in `CXF`.

7.15 Extension of a Composite Block

A composite block can have a single `extends` statement. The `extends` statement must reference another Composite Block, but it cannot extend an Elementary Block or an Extension Block. The `extends` statement can have any number of declarations that assign a parameter value or parameter attributes.

Note: There are three restrictions compared to the Modelica Language Specification [Mod23]:

1. Only a single `extends` statement is allowed. This is for simplicity because two `extends` statements could require having to reconcile two different hierarchy trees that ultimately extend from the same base block, but may assign different values to a parameter that is inherited from the common base block. Such a case would be for example

```
package MultipleExtends

  block A0
    extends Buildings.Controls.OBC.CDL.Reals.Sources.Constant(k=0);
  end A0;

  block A1
    extends Buildings.Controls.OBC.CDL.Reals.Sources.Constant(k=1);
  end A1;

  block NotValid "Block that is not valid"
    extends A0;
    extends A1;
  end NotValid;

  annotation(
    uses(Buildings(version = "12.0.0")),
    Documentation(
      info = "<p>
Package with a block that is not valid CDL due to multiple extends statements.
</p>"));

end MultipleExtends;
```

   Note that in Modelica, multiple `extends` are allowed, but the block `NotValid` is not valid and tools will issue an error message.
2. The `break` keyword for component deselection is not allowed.
3. Modelica allows to assign a value to a variable declared as an `input`. This is not allowed in CDL. This restriction avoids that input connectors can no longer be graphically connected (as they then would have two bindings to a value, causing the block to be overdetermined).

[A simple illustrative example of an `extends` statement would be to extends the block `OBC.Utilities.PIDWithInputGains`, restricts its output to be always between `0` and `1`, and adding an output connector that can be used to access the control error.

This could be accomplished as

```
block MyPID
  extends Buildings.Controls.OBC.Utilities.PIDWithInputGains(
    final yMin = 0,
    final yMax = 1);
```

(continues on next page)

(continued from previous page)

```
  Buildings.Controls.OBC.CDL.Interfaces.RealOutput error
    "Control error"
    annotation(
      Placement(
        transformation(
          origin = {240, -120},
          extent = {{-20, -20}, {20, 20}}),
        iconTransformation(
          origin = {120, -60},
          extent = {{-20, -20}, {20, 20}})));
equation
  connect(controlError.y, error) annotation(
    Line(
      points = {{-178, -6}, {-160, -6},
                {-160, -120}, {240, -120}},
      color = {0, 0, 127}));
  annotation(
    uses(Buildings(version = "12.0.0")),
    Documentation(
      info = "<p>
PID controller that extends the PID controller
with input gains, and that limits the output
between 0 and 1, and
adds an output connector that reports
the control error.
</p>"));

end MyPID;
```

]

The `extends` statement can also have any number of `redeclare` statements (Section 7.14).

[For example, in the block below, the controller with name `conPID` is replaced with the block `OBC.CDL.Reals.PIDWithReset`.

```
model MyBlockWithRedeclare
  extends SomeCompositeBlock(
    redeclare Buildings.Controls.OBC.CDL.Reals.PIDWithReset conPID);

end MyBlockWithRedeclare;
```

]

In a replaceable declaration, the optional constraining-clause defines a constraining type. Any modifications following the constraining type name are applied both for the purpose of defining the actual constraining type, and they are automatically applied in the declaration and in any subsequent redeclaration. The precedence order is that declaration modifiers override

constraining type modifiers.

If the constraining-clause is not present in the original declaration (i.e., the non-redeclared declaration), then the following applies:

- The type of the declaration is also used as a constraining type.
- The modifiers for subsequent redeclarations and constraining type are the modifiers on the component or short-class-definition if that is used in the original declaration, otherwise empty.

[Consider the following example, and see also Section 7.3.2 in [Mod23]:

```
model Constraints
  record BaseRecord
    parameter Integer param=0;
  end BaseRecord;

  record Record
    extends BaseRecord(final param=1);
  end Record;

  model Model
    parameter Integer param=2;
    // OpenModelica errors because param is declared final
    replaceable parameter BaseRecord rec constrainedby BaseRecord(param=param);
  end Model;

  // Overriding by param=param from the constraining clause:
  // OpenModelica and Dymola errors,
  // Modelon OPTIMICA evaluates to 2.
  Model component1(redeclare Record rec);

   // Precedence of declaration over constraining clause from
   // https://specification.modelica.org/maint/3.5/inheritance-modification-and-redeclaration.
↪html#constraining-type:
   // Dymola and Modelon OPTIMICA return 1.
  Model component2(rec(final param=1));

  // Precedence of declaration over constraining clause: Dymola and OCT return 1.
  Model component3(rec=Record());
end Constraints;
```

]

7.16 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 Language Specification [Mod23].]

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.]

7.17 Metadata

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 metadata, implemented through Modelica vendor annotations, to provide this additional information. In Section 7.17.1, we will explain the properties that can be inferred, and in Section 7.17.2, we will explain how to use semantic models 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.

7.17.1 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.)

]

7.17.2 Semantic Information

The buildings industry has started to integrate different metadata languages such as Brick and Project Haystack into their control software and technology. ASHRAE Standard 223p is another upcoming metadata language that will describe the equipment topology in buildings and also the flow of different media. This section specifies the syntax to support these metadata languages and include the semantic information represented using these languages in a CDL class.

Semantic information shall be included within the `annotation` keyword, using the `__Buildings` or `__cdl` vendor annotation. `__cdl` shall be used when the semantic information is part of a control sequence and `__Buildings` shall be used for every other instance such as equipment or a zone. The following instances shall optionally have annotations containing semantic information:

- input and output connectors (Section 7.8),
- parameters (Section 7.4.2),
- constants (Section 7.4.3),
- connections (Section 7.10),
- elementary blocks (Section 7.6),
- composite blocks (Section 7.12),
- extension blocks (Section 7.13), and
- packages.

All semantic information shall be included under the `semantic` section within the `__Buildings` or `__cdl` annotations, using the syntax shown here:

```
annotation (__cdl(      semantic(<semantic information>)));
annotation (__Buildings(semantic(<semantic information>)));
```

where `<semantic information>` is a place holder for the semantic information.

The `semantic` annotation declared in the class definition shall optionally contain the `metadataLanguageDefinition` or the `naturalLanguageDefinition` for each of the languages used. The `metadataLanguageDefinition` and `naturalLanguageDefinition` are used to provide additional information about the different metadata languages and natural languages that are used throughout the class. The language definitions contain information such as a short description of the language or the URL to the webpage of the language.

The optional `metadataLanguageDefinition` shall have the following syntax:

```
annotation (__cdl(semantic(
  metadataLanguageDefinition="<metadataLanguageName> <version> <format>" ["informative text
↪"])));
annotation (__Buildings(semantic(
  metadataLanguageDefinition="<metadataLanguageName> <version> <format>" ["informative text
↪"])));
```

where `<metadataLanguageName>` shall be replaced with the name of the metadata language, `<version>` is the mandatory entry for the version, `<format>` is the mandatory field for format of the language, such as `text/turtle`, and `["informative text"]` is an optional description of the language, such as the URL to the language. The `version`

represents the version of the <metadataLanguageName> used in a particular class. The format represents the format that the semantic information is expressed in. The format shall be expressed using MIME types.

The optional naturalLanguageDefinition shall have the following syntax:

```
annotation (__cdl(semantic(
  naturalLanguageDefinition="<naturalLanguageName>" ["informative text"])));
annotation (__Buildings(
  semantic(naturalLanguageDefinition="<naturalLanguageName>" ["informative text"])));
```

where <naturalLanguageName> shall be replaced with the indicator of the natural language, represented using the ISO-639 language codes and ["informative text"] is an optional description of the language. All <naturalLanguageName> metadata will be in the format text/plain.

[Examples of the <metadataLanguageName> include web ontology languages (OWL) such as Brick or ASHRAE S223p, and examples of <naturalLanguageName> include en or es. Below is an example of how to define multiple metadataLanguageDefinition and naturalLanguageDefinition in a class definition annotation.

Example:

```
annotation (__cdl(semantic(
  metadataLanguageDefinition="Brick 1.3 text/turtle" "https://brickschema.org/ontology/1.3",
  metadataLanguageDefinition="Project-Haystack 3.9.12 application/ld+json" "https://project-
↪haystack.org/",
  naturalLanguageDefinition="en" "Text in English language"
)));
]
```

The semantic information shall be included as a metadataLanguage/metadata or a naturalLanguage/metadata pair within the semantic section in the __cdl or __Buildings annotation using the following syntax:

```
annotation (__cdl(semantic(metadataLanguage="<metadataLanguageName> <version> <format>" "
↪<metadata>")));
annotation (__Buildings(semantic(metadataLanguage="<metadataLanguageName> <version> <format>" "
↪<metadata>")));

annotation (__cdl(semantic(naturalLanguage="<naturalLanguageName>" "<metadata>")));
annotation (__Buildings(semantic(naturalLanguage="<naturalLanguageName>" "<metadata>")));
```

where <metadataLanguageName> shall be replaced with the name of the metadata language, <version> is an entry for the version of the metadataLanguage, <format> is the format of the metadataLanguage, such as text/turtle, <naturalLanguageName> shall be replaced with the ISO-639 indicator of the natural language and <metadata> is the metadata for that instance as specified in <metadataLanguageName> or <naturalLanguageName> language.

Note: Depending on the metadataLanguage ("<metadataLanguageName> <version> <format>"), the metadata can be represented in multiple formats. For example, text/turtle and application/ld+json are a couple of formats to represent the metadata of web ontology languages such as Brick and ASHRAE S223p. Project-Haystack metadata can also be represented in multiple formats such as text/zinc, text/turtle and application/ld+json.

Semantic information in the class definition annotations shall optionally be used to define class level information about the metadata languages. These include, but are not restricted to, namespace definitions (namespaces in ontologies provide a means to unambiguously interpret identifiers and make the rest of the ontology presentation more readable) and prefixes (prefixes are shortcut abbreviations and help make the semantic information more readable).

[In the example below, for the `metadataLanguage` "`Brick 1.3 text/turtle`", the class definition annotation has been used to define the namespace prefixes and for "`Project-Haystack 3.9.12 application/ld+json`", it has been used to define namespaces, prefixes and contexts.

Example:

```
annotation (__cdl(semantic(
  metadataLanguage="Brick 1.3 text/turtle" "@prefix Brick: <https://brickschema.org/schema/Brick
↪#> .
                                            @prefix bldg: <urn:bldg/> . ",
  metadataLanguage="Project-Haystack 3.9.12 application/ld+json"
                                           "{\"@context\": { \"ph\": \"https://project-haystack.
↪org/def/ph/3.9.12#\",
                                           \"phScience\": \"https://project-haystack.org/def/
↪phScience/3.9.12#\",
                                           \"phIoT\": \"https://project-haystack.org/def/phIoT/
↪3.9.12#\",
                                           \"rdf\": \"http://www.w3.org/1999/02/22-rdf-syntax-ns
↪#\",
                                           \"rdfs\": \"http://www.w3.org/2000/01/rdf-schema#\"}}
↪")));
```

]

If an instance declaration contains semantic information, it overrides the semantic information of its class definition. If an instance declaration does not contain semantic information, it inherits the semantic information of its class definition.

Additionally, if there already exists a semantic model for a particular class or for an instance, it shall be referred to in the annotation using the syntax defined below:

```
annotation (__cdl(semantic(metadataLanguage="<metadataLanguageName> <version> <format>" "url=
↪<path>")));
annotation (__Buildings(semantic(metadataLanguage="<metadataLanguageName> <version> <format>"
↪"url=<path>")));

annotation (__cdl(semantic(naturalLanguage="<naturalLanguageName>" "url=<path>")));
annotation (__Buildings(semantic(naturalLanguage="<naturalLanguageName>" "url=<path>")));
```

where <path> shall be either a URL for a model that is on the network or a model that is present on the file system. If the `url=` is included in the `metadata`, the semantic model will be exported from <path>. If `url=` is not included in the `metadata`, <path> shall be the the `metadata`.

If the metadata model is present on the file system as separate file, the following syntax shall be followed:

```
annotation (__cdl(semantic(metadataLanguage="<metadataLanguageName> <version> <format>"
↪"url=file:///<path/to/file>")));
annotation (__Buildings(semantic(metadataLanguage="<metadataLanguageName> <version> <format>"
↪"url=file:///<path/to/file>")));

annotation (__cdl(semantic(naturalLanguage="<naturalLanguageName>" "url=file:///<path/to/file>
↪")));
annotation (__Buildings(semantic(naturalLanguage="<naturalLanguageName>" "url=file:///<path/to/
↪file>")));
```

[Below are examples of how to refer to an existing “Brick 1.3 text/turtle” semantic model existing on the file system at “/home/user/soda_hall/soda_brick.ttl” and a “Project-Haystack 3.9.12 application/ld+json” semantic model on the network at the URL “https://project-haystack.org/example/download/alpha.jsonld”.

Example:

```
annotation (__cdl(
  semantic(
    metadataLanguage="Brick 1.3 text/turtle" "url=file:///home/user/soda_hall/soda_brick.ttl
↪")));

annotation (__cdl(
  semantic(
    metadataLanguage="Project-Haystack 3.9.12 application/ld+json" "url=https://project-
↪haystack.org/example/download/alpha.jsonld")));
```

]

<instanceName>: The text <instanceName> (including the < and > characters) within the metadata of an annotation containing semantic information shall be replaced with the fully qualified name of the instance that contains the semantic annotation. A fully qualified name to an instance refers to the complete hierarchical path that specifies the instance’s location within an object structure. This qualified name shall include all parent instances leading up to the current instance, with each level of instantiation separated by an underscore (“_”). If an instance is nested within multiple levels of instance definitions, the text that replaces <instanceName> shall reflect the entire chain of instantiation. This avoids the user having to repeat the name of the instance and makes it less prone to errors and inconsistencies.

[An example of CDL semantic information for an instance in a class with multiple metadataLanguage/metadata pair is shown below. We can see that <instanceName> has been used in the metadata and Brick metadata will be inferred as bldg:THeaCoiSup_in a Brick:Hot_Water_Supply_Temperature_Sensor . and the Project Haystack identifier as {"@id": "THeaCoiSup_in"}.

Example:

```
Modelica.Blocks.Interfaces.RealInput THeaCoiSup_in
    "Heating coil water supply temperature measurement"
    annotation (Placement(transformation(extent={{-140,-180},{-100,-140}})),
                __cdl(semantic(
                metadataLanguage="Brick 1.3 text/turtle"
```

(continues on next page)

(continued from previous page)

```
                "bldg:<instanceName> a Brick:Hot_Water_Supply_Temperature_Sensor .",
              metadataLanguage=" Project-Haystack 3.9.12 application/ld+json"
                "{
                   \"@id\": \"_:<instanceName>\",
                   \"ph:hasTag\": [
                       {\"@id\": \"phIoT:cur\"},
                       {\"@id\": \"phIoT:hot\"},
                       {\"@id\": \"phIoT:leaving\"},
                       {\"@id\": \"phIoT:point\"},
                       {\"@id\": \"phIoT:sensor\"},
                       {\"@id\": \"phScience:temp\"},
                       {\"@id\": \"phScience:water\"}
                   ],
                   \"rdfs:label\": \" Heating Hot Water Supply Temperature\"
                }",
              metadataLanguage="en"
                "<instanceName> is a temperature reading input that should be hardwired to␣
↪heating coil temperature sensor")));
```

]

<parameter>: This syntax allows for a value of a parameter to be used within an annotation containing semantic information where the parameter shall refer to the name of a parameter instance within the class. The text <parameter> (including the < and > characters) shall be replaced by the value of the parameter. The class must have an instance of a parameter with the name specified by <parameter>, otherwise the specification is not valid.

[In the below example, if the fully qualified name of reaFloSup is reaFloSup, the <instanceName> will be replaced by reaFloSup. The location of the sensor, represented by the brick:hasLocation relationship, after replacing <instanceName> will be bldg:<zon>. <zon> refers to the value of the zon parameter within the instantiated reaFloSup, which is east. Hence, the completely evaluated semantic information becomes:

```
bldg:reaFloSup a brick:Supply_Air_Flow_Sensor;
         brick:hasLocation bldg:east .
```

Example:

```
MyCompositeBlock.MyFlowSensor reaFloSup (zon="east") "Supply Air Flow Rate"
    annotation ( __cdl(semantic(
    metadataLanguage="Brick 1.3 text/turtle"
        "bldg:<instanceName> a brick:Supply_Air_Flow_Sensor;
               brick:hasLocation bldg:<zon> .")));
```

]

The semantic information of an instance shall be able to refer to the semantic information of other instances declared in the class (or in inherited classes). If the instance does not exist, the semantic model is invalid.

[In the below example, the semantic information of heating coil heaCoi is referring to the semantic information of the hot water supply temperature sensor THeaCoiSup_in.

```
 Modelica.Blocks.Interfaces.RealInput THeaCoiSup_in
     "Heating coil water supply temperature measurement"
     annotation (Placement(transformation(extent={{-140,-180},{-100,-140}})),
                 __cdl(semantic(
                 metadataLanguage="Brick 1.3 text/turtle"
                   "bldg:<instanceName> a Brick:Hot_Water_Supply_Temperature_Sensor ."
                )));

Buildings.Fluid.HeatExchangers.DryCoilEffectivenessNTU heaCoi(
  show_T=true,
  dp1_nominal=3000,
  dp2_nominal=0
  ) "Heating coil"
  annotation (Placement(transformation(extent={{118,-36},{98,-56}})),
              __Buildings(semantic(
                 metadataLanguage="Brick 1.3 text/turtle"
                   "bldg:<instanceName> a Brick:Heating_Coil ;
                               brick:hasPoint bldg:THeaCoiSup_in ."
                ));
```

]

If a class inherits one or more classes (CDL only allows for inheriting one class), all the semantic information in the classes is inherited. However, if the classes being inherited and the class inheriting it contains different `metadataLanguage` or `naturalLanguage` due to differences in any of `<metadataLanguageName>` or `<version>` or `<format>` or `<naturalLanguageName>` parts of the syntax, they shall be treated as different languages.

If an inherited `replaceable` instance has been replaced using the `redeclare` keyword, the semantic information of the instance that replaced the original instance shall be used, and the semantic information of the replaced class shall be ignored. If there is no semantic information in the redeclared instance annotation, any semantic information of the constraining clause (using the `constrainedby` Modelica keyword) of the original `replaceable` instance shall be used. Any semantic information in the original `replaceable` instance shall not be used if it has been redeclared irrespective of the presence or absence of semantic information in the constraining clause of the redeclared instance.

Chapter 8

Control eXchange Format (CXF)

8.1 Introduction

CXF is a representation of CDL in a format that is intended to be readily imported and exported into building automation systems. For example, a commercial control provider might utilize it to import control logic from a design tool and deploy it their commercial building automation system for a particular project. Structurally the content of a logic in CDL and CXF are identical, in that both utilize the same ElementaryBlocks, CompositeBlocks, and ExtensionBlocks as well as Constants, Parameters, InputConnectors and OutputConnectors. While CDL has language constructs that are used to build library of sequences, CXF was designed to only represent a specifically configured logic. The logic described in a CDL implementation is identical to the logic described in its CXF representation. But there are several key differences between CDL and CXF:

- CXF is defined utilizing the linked data format JSON-LD, while CDL utilizes the modeling language Modelica. JSON-LD is a syntax to serialize linked data in JSON (ECMA-404).
- There is a translation process required to convert a control logic from CDL to CXF.
- For ElementaryBlocks (Section 7.6), their CXF representation does not include the implementation (`equation` section).
- Like many scientific modeling languages, Modelica requires tight casting of data types.
  [For example, in Modelica, a data type needs to be declared as type `Real` or `Integer`. `Real` data are not allowed to be tested for equality since computations are prone to rounding errors.]

  Note: When importing a CXF representation of a CDL logic into a commercial control system that does not support `Real` or `Integer` data types, the commercial entity's "CDL import" software tool must determine on how to handle the `Real` and `Integer` InputConnectors, OutputConnectors, Parameters and Constants. For example, the tool could change it to `Analog`. Similarly, while exporting a CXF representation of a control logic implemented in a commercial control system, the commercial entity's "CDL export" software tool must decide how to translate unsupported data types such as `Analog` into `Real` or `Integer` InputConnectors, OutputConnectors, Parameters and Constants.

- Control logic which utilize arrays (both one- and multi -dimensional) in CDL shall have the option to be modified (or "flattened") in CXF (more details provided in a later section).

8.2 Classes and Properties

A valid CXF file contains Blocks (ElementaryBlocks, CompositeBlocks, ExtensionBlocks or a combination of these) and each instance of a Block uses the set of InputConnectors, OutputConnectors, Parameters, and Constants as defined within definition of the Block. To support the translation of a CDL control logic to its CXF representation, a Resource Description Framework graph representation of the standard has been provided in a CXF-Core.jsonld file using the MIME type `application/ld+json`. CXF-Core.jsonld can be found here. The key classes and properties present in CXF-Core.jsonld that can be used to created CXF classes are shown in Table Table 8.1 and Table Table 8.2 respectively.

Table 8.1: Key classes within `CXF-Core.jsonld`

Class	Description
Package	A Package is a specialized class used to group multiple Blocks.
Blocks	A Block is the abstract interface of a control logic.
Elementary-Block	An ElementaryBlock defined by ASHRAE S231 (`subClassOf Block`) (Section 7.6).
Composite-Block	A CompositeBlock is a collection of ElementaryBlocks or other CompositeBlocks (`subClassOf Block`) and the connections through their inputs and outputs (Section 7.12).
ExtensionBlock	An ExtensionBlock supports functionalities that cannot, or are hard to, implement with a CompositeBlock (`subClassOf Block`) (Section 7.13).
InputConnector	An InputConnector provides an input to a Block.
OutputConnector	An OutputConnector provides an output from a Block.
Parameter	A Parameter is a value that is time-invariant and cannot be changed based on an input signal.
Constant	A Constant is a value that is fixed at compilation time.
DataType	A data type description for InputConnectors, OutputConnectors, Parameters and Constants.
BooleanInput	An InputConnector of the Boolean data type.
BooleanOutput	An OutputConnector of the Boolean data type.
IntegerInput	An InputConnector of the Integer data type.
IntegerOutput	An OutputConnector of the Integer data type.
RealInput	An InputConnector of the Real data type.
RealOutput	An OutputConnector of the Real data type.
EnumerationType	An Integer enumeration starting with the value 1, each element is mapped to a unique String.
String	A data type to represent text.

Table 8.2: Key properties within `CXF-Core.jsonld`

Property	Domain	Range	Description
hasInput	Block	InputConnector	A property that relates an instance of an InputConnector with a Block.
hasOutput	Block	OutputConnector	A property that relates an instance of an OutputConnector with a Block.
hasParameter	Block	Parameter	A property that relates an instance of a Parameter with a Block.
hasConstant	Block	Constant	A property that relates an instance of a Constant with a Block.
hasInstance	Block	Block, InputConnector, OutputConnector, Parameter, Constant	A property that associates an instance of an InputConnector, OutputConnector, Parameter, Constant or a Block within a Block with the instance of the Block itself.
hasFmuPath	ExtensionBlock	String	A property that specifies the (local or on the network) path to a Functional Mockup Unit implementation of an ExtensionBlock.
isOfDataType	InputConnector, OutputConnector, Parameter, Constant	DataType	A property that specifies the data type for instances of InputConnectors, OutputConnectors, Parameters and Constants.
containsBlock	Block	Block	A property that specifies that an instance of a Block is composed in part with an instance of another Block.
connectedTo	OutputConnector, InputConnector	InputConnector, OutputConnector	A property that relates the output of one Block to the input of another Block (and vice-versa). Only InputConnectors and OutputConnectors that carry the same data type can be connected.
translationSoftware	Package, Block	String	A property that specifies the name of the software used to generate the CXF representation of the control logic.
translationSoftwareVersion	Package, Block	String	A property that specifies the version of the software used to generate CXF representation of the control logic.

All the ElementaryBlocks within the standard have been defined and included in `CXF-Core.jsonld`. However, CXF representation of elementary blocks does not contain the implementation details of the blocks.

8.3 Generating CXF from an instance of a CDL class

If the instantiation of a CDL block (within a Modelica or another CDL class) contains the annotation `__cdl(export=true)`, the CDL class of the instantiated block shall be translated to CXF. Specifying the `export` annotation is optional and if unspecified, `export=false` is assumed.

8.4 Source of CXF translation

The CXF representation of a control logic shall optionally include the name and the version of the software that generated it using the properties `translationSoftware` and `translationSoftwareVersion` respectively.

8.5 Representing Instances in CXF

In the CXF representation of a CDL control logic, the instances of the CDL class shall contain the entire package path of the CDL class, the octothorpe character (#), followed by the name of the instance. An ("child") instance of an ("parent") instance shall be referenced in CXF by the parent instance's CXF representation, followed by a period character (`.`) and then the child instance's name. Additionally, the CXF representation of the parent instance shall contain a `hasInstance` property associating it to the child instance.

[Example of a CDL instance representation in CXF

CDL:

```
within ExamplePackage;
block ExampleSeq
CDL.Reals.MultiplyByParameter gain(k = 100000)
    "My gain";
end ExampleSeq;
```

CXF reference to `gain` instance: `ExamplePackage.ExampleSeq#gain`

CXF reference to `gain.k` instance: `ExamplePackage.ExampleSeq#gain.k`

CXF property linking `gain` and `gain.k`: `ExamplePackage.ExampleSeq#gain S231:hasInstance ExamplePackage.ExampleSeq#gain.k .`]

8.6 Handling Arrays and Expressions

Arrays and expressions in a CDL class shall be represented in CXF as specified below:

- Arrays (both one-dimensional (vectors) and multi-dimensional): In the CXF translation, array references shall either be preserved or flattened. If the array references are to be flattened, the indices appearing within square brackets (`[` and `]`) in CDL shall be appended with the underscore (_) character and each index shall be concatenated with the underscore character (_).
  [For example, if there array references are preserved, `A[1]` in CDL shall be referenced as `A[1]` in CXF. If they are flattened, `A[1]` shall be represented as `A_1` and `B[1 ,2]` shall be represented as `B_1_2`.]
  Array references in CDL shall be flattened in the row-major approach in CXF. Flattened array references shall be generated row-wise, starting from the left-most element of the first row to the right-most element of the first row, before advancing to the next row, until the right-most element of the last row.
  If there already exists an instance in the CDL logic with the same name as a flattened array reference, then the translation process shall raise an error.

[For example, if in a CDL class, there exists a parameter instance `A_1` and a vector with 3 elements `A[3]`, upon flattening, references to the first element of the vector (`A[1]`) would become `A_1`. As this instance already exists, the CXF translator tool shall raise an error.]

- Expressions: The CXF translation of a CDL control logic shall either preserve or evaluate all the expressions present in the CDL logic, such as those within assignment operations, conditional assignments and arithmetic operations. By default, the expressions shall be preserved in the CXF representation. If the expressions have to be evaluated and the expressions contain references to a parameter, the value of the parameter shall be used in the evaluating the expression. If the expressions have to be evaluated and expressions contain references to parameter(s) that does not have a value binding, then the translation process shall raise an error.

Note: The determination of whether arrays should be flattened or expressions should be evaluated shall be made by the software tool that generates CXF representations of the CDL control logic.

8.7 ExtensionBlocks

Instances of ExtensionBlocks within a CDL classs shall contain the annotation `__cdl(extenstion=true)`. The location of the Functional Mockup Unit implementation of the ExtensionBlock shall be included in the CXF representation using the property `hasFmuPath`.

Chapter 9

Documentation of Control Sequences

9.1 Introduction

This section describes how to generate a control sequence description based on a CDL specification.

There are two distinct situations:

1. The control sequence could be from a publication such as ASHRAE Guideline 36 for which a Microsoft Word version exists, or
2. The control sequence could be for a sequence that only exists in CDL.

The approach for 1. is currently being developed. Approach 2 is described in Section 9.3.

9.2 Editing a Sequence that is Specified in a Word Document

This is currently being specified and will be added later.

9.3 Exporting the Control Logic from a CDL Model

This section describes how a English language description of a sequence could be exported from the CDL implementation. This will allow developers and users to build libraries of control sequences for which an English language specification can be exported without having to have a template Word document (which generally does not exist for this use case).

Two different representations will be supported:

1. *Specifications for sequences of operations.* These specifications expresses the intent of the designer for the sequence. They contain text in the form of requirements, such as “The room temperature shall be maintained between . . . ”. Such requirements leave room for different interpretations and resulting implementations of the control inputs and outputs, and the control logic, thereby making verification as in Section 12.7 impractical. It also risk that the sequences do not satisfy the designer’s intent. However, if encoded in a library that has been tested, the control sequence can be specified more precise.

2. *Documentation of the as-implemented sequences.* These typically serve the operator, and may contain text such as "The controller tracks the room temperature set point by ...". This type of formulation is also what is typically used to document the implementation of sequences in the Modelica Buildings Library.

Control sequences of the form 1) typically contain additional requirements that are not part of the sequence description, such as what energy code to follow. Such information can however be included in a section that precedes or follows the actual sequence implementation. Thus, the here described export will document only the sequences, which can then be combined with these other documentation.

To export sequence specifications of the form 1), we introduce a new optional annotation `annotation(__CDL(SequenceSpecification(info=STRING)))` where `STRING` is an html formatted string that contains the sequence specification. E.g., the annotation is in the same format as the CDL annotation `annotation(Documentation(info=STRING)`. The new optional annotation is introduced solely for the purpose that in the buildings industry, control specifications use a different form than what is usually used in Modelica, i.e., to address the differences between 1) and 2) above. I.e., Modelica documentation describe what a sequence does, whereas for sequence specifications, the sequence description must follow the structure dictated by the Construction Specification Institute (CSI) and the American Institute of Architects (AIA) because they become legal documents.

How to generate the sequence description that can be inserted into these construction documents is described using a small example. Consider the model Buildings.ThermalZones.EnergyPlus_9_6_0.Examples.SingleFamilyHouse.RadiantHeatingCooling_TSurface. This model has two sequences, one for the radiant heating and one for the radiant cooling. These two sequences are described in Buildings.Controls.OBC.RadiantSystems.Heating.HighMassSupplyTemperature_TRoom and in Buildings.Controls.OBC.RadiantSystems.Cooling.HighMassSupplyTemperature_TRoomRelHum using html format.

To export sequences from these models, `modelica-json` will need to generate a Microsoft Word document using the following procedure.

1. Read the top-level Modelica file and extract each block that is in the package `Buildings.Controls.OBC`. Put the names of these blocks in a list.
2. Remove from this list all blocks that are in `Buildings.Controls.OBC.CDL`. (These are are elementary blocks that need not be documented.)
3. Read the top-level Modelica file and extract all blocks that contain in their class definition the annotation `__cdl(document=true)`. Add these blocks to the list. (This will allow users to add composite control blocks that will be documented.)
4. For each block in the list.
   a. If the block contains a section `annotation(__CDL(SequenceSpecification(info=STRING)))`, use the value of this section as the sequence documentation of this block. Goto step d).
   b. If the block contains a section `annotation(Documentation(info=STRING))`, write a warning that this block will be documented with as-implemented description rather than a sequence specification as no control sequence specification has been found, and use the value of this section as the sequence documentation of this block. Goto step d).
   c. Issue a warning that this block contains no control sequence description and proceed to the next block.
   d. In the sequence description of this block, for each parameter that is in the description, add the value and units. For example, an entry such as `... between <code>TSupSetMin</code> and <code>TSupSetMax</code> based on ...` becomes `... between <code>TSupSetMin</code> (=20&deg; adjustable) and <code>TSupSetMax</code> (=40&deg; adjustable) based on ...`. Note that the word "adjustable" must not be added if the parameter value is declared as `final`. Proceed to the next block.
5. Collect the descriptions of each block and output it in a Word document. Configure the Word document to have

automatic section numbering.

As an example, consider the following snippet of a composite control block.

```
HighMassSupplyTemperature_TRoom con(TSubSet_max=303.15, final TSubSet_min=293.15);

block HighMassSupplyTemperature_TRoom
  "Room temperature controller for radiant heating with constant mass flow and variable supply␣
↪temperature"

  parameter Real TSupSet_max(
    final unit="K",
    displayUnit="degC") "Maximum heating supply water temperature";
  parameter Real TSupSet_min(
    final unit="K",
    displayUnit="degC") = 293.15 "Minimum heating supply water temperature";

  parameter Controls.OBC.CDL.Types.SimpleController
    controllerType = Buildings.Controls.OBC.CDL.Types.SimpleController.P
    "Type of controller" annotation (Dialog(group="Control gains"));

  ... [omitted]

  annotation(
    Documentation(
      info="<html>
        <p>
        Controller for a radiant heating system.
        </p>
        <p>
        The controller tracks the room temperature set point <code>TRooSet</code> by
        adjusting the supply water temperature set point <code>TSupSet</code> linearly between
        <code>TSupSetMin</code> and <code>TSupSetMax</code>

        PI-controller likely saturate due to the slow system response.
        </p>
        </html>"
      ),
      __cdl(
        SequenceSpecification(
          info="<html>
            <p>
            Controller for a radiant heating system.
            </p>
            <p>
            The controller shall track the room temperature set point by
            adjusting the supply water temperature set point <code>TSupSet</code> linearly␣
↪between
```

(continues on next page)

(continued from previous page)

```
            <code>TSupSetMin</code> and <code>TSupSetMax</code>
            based on the output signal of the proportional controller.
            The pump shall be either off or be operating at full speed, in which case <code>
↪yPum = 1</code>.
            The pump control shall be based on a hysteresis that switches the pump on when the␣
↪output of the
            proportional controller <code>y</code> exceeds <i>0.2</i>, and the pump shall be␣
↪commanded off when the output falls
            below <i>0.1</i>. See figure below for the control charts.
            </p>
            <p align="center">
            <img alt="Image of control output"
            src="modelica://Buildings/Resources/Images/Controls/OBC/RadiantSystems/Heating/
↪HighMassSupplyTemperature_TRoom.png"/>
            </p>
            <p>
            <-- cdl(visible=(not (controllerType is final))) or controllerType <> CDL.Types.
↪SimpleController.P -->
            <b>Note:</b>
            For systems with high thermal mass, this controller should be left configured
            as a P-controller, which is the default setting.
            PI-controller likely saturate due to the slow system response.
            </p>
            <-- end cdl -->
          </html>"
        )
    )
  );
end HighMassSupplyTemperature_TRoom;
```

For this control block, `modelica-json` will produce content for the Word description that looks like

> "The controller shall track the room temperature set point by adjusting the supply water temperature set point `TSupSet` linearly between `TSupSetMin` (= 20°) and `TSupSetMax` (= 30° adjustable) based on the output signal of the proportional controller..."

`modelica-json` will remove the notice at the end of the sequence description if the `controllerType` is declared as `final` (because then, no other choice can be made). Through this mechanism, sections and images can be removed or enabled in the generated sequence description.

To use IP units, `modelica-json` will have a configuration that specifies what units should be used. The documentation will also include the figure as declared in the CDL specification.

The Control Sequence Selection and Configuration tool could make the section `annotation(__CDL(SequenceSpecification(info=STRING)))` editable, thereby allowing users to customize the description of the sequence and add any other desired documentation.

Chapter 10

Controls Library

10.1 Introduction

To implement control sequences that conform to the CDL specification of Section 7, we implemented a library of elementary control blocks, and a library of control sequences that are composed of these elementary blocks, using composition rules that are specified in the CDL specification. The next two sections give a brief overview of these library. To see their implementation, browse the online documentation at https://simulationresearch.lbl.gov/modelica/releases/latest/help/Buildings_Controls_OBC.html.

10.2 CDL Library

To implement control sequences in CDL, we created the CDL library. This library contains all compositional elements of the CDL language, such as connectors for input and output signals of various types (real, integer etc.), type definitions such as for the day-of-week, and the elementary blocks that are described in Section 7.6. This library consist of about 130 elementary blocks, such as a block that adds two real-valued input signals and produces its sum as the output, a block that implements a proportional-integral-derivative controller with anti-windup, and blocks that perform basic operations on boolean signals. Thus, the CDL library defines the necessary and sufficient set of models that need to be supported by control product lines to which control sequences that are expressed in CDL can be translated to, using the process described in Section 11.3.

These elementary blocks are used to compose control sequences for mechanical systems, lighting systems and active facades as described in the next section.

10.3 Library of Control Sequences

To make ready-to-use control sequences available to building designers, researchers and control providers, we implemented control sequences for secondary HVAC systems based on ASHRAE Guideline 36, for lighting systems and for active facades.

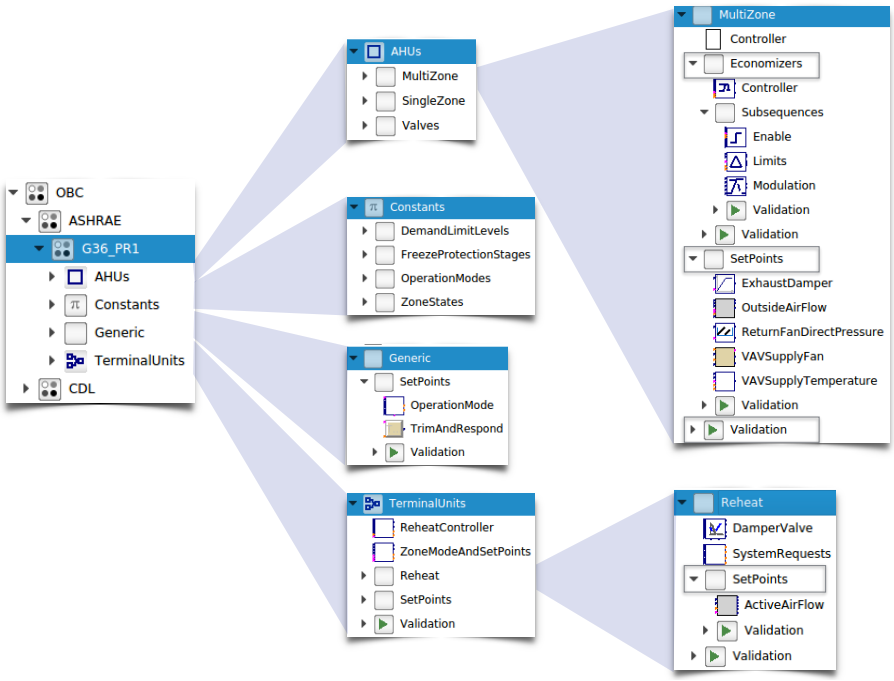


Fig. 10.1: Overview of package that includes control sequences from ASHRAE Guideline 36.

For example, Fig. 10.1 shows an overview of the control sequences that have been implemented based on ASHRAE Guideline 36. The implementation is structured hierarchically into packages for air handler units, into constants that indicate the mode of operation, into generic sequences such as for a trim and respond logic, and into sequences for terminal units. Around 30 smaller sequences are used to hierarchically compose controllers for single-zone and multi-zone VAV systems.

Every sequence contains an English language description, an implementation using block diagram modeling, and one or more examples that illustrate the use of the sequence. These examples are available in the `Validation` package in which the sequences are used, typically with open-loop tests. For top-level sequences, there are also closed loop tests available. For example Fig. 10.2 shows the schematic view of the model that evaluates the performance of the single zone VAV controller based on ASHRAE Guideline 36 [ZBG+20]. In this model, the controller output is connected to an HVAC system model, which in turn is connected to a model of the building. Sensor data from the HVAC system and the room air temperature are fed back to the controller to form the closed loop test. The model is available in the Modelica Buildings Library as the model `Buildings.Air.Systems.SingleZone.VAV.Examples.Guideline36`.

As of Fall 2020, additional sequences are being implemented for chilled water plants and for boiler plants, following the ASHRAE Research Project Report 1711, and for optimal start-up (for heating) and cool down (for cooling).

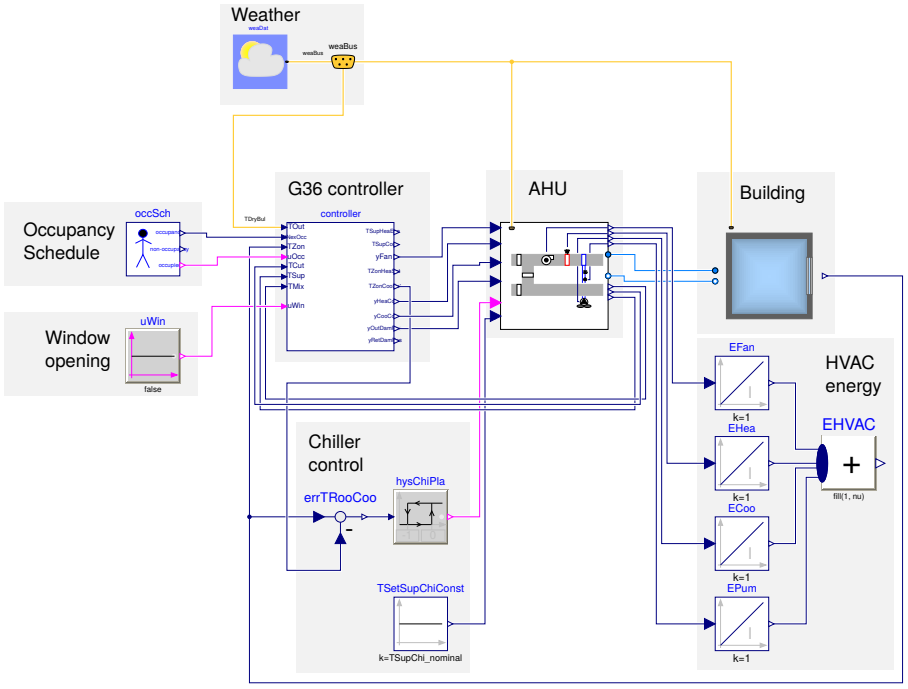


Fig. 10.2: Schematic view of model that uses the CDL implementation of the single zone VAV controller based on ASHRAE Guideline 36.

Chapter 11

Code Generation

11.1 Introduction

This section describes the translation of control sequences expressed in CDL to a building automation system.

Translating the *CDL library* to a building automation system to make it available as part of a product line 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* that has been developed for a specific building will need to be done for each building project.

While translation from CDL to C code or to a *Functional Mockup Unit* is support by Modelica simulation environments, translation to legacy building automation product lines is more difficult as they typically do not allow executing custom C code. Moreover, a building operator typically needs a graphical operator interface, which would not be supported if one were to simply upload compiled C code to a building automation system.

Use of CDL control sequences for building operation, or use of such sequences in a verification test module, consists of the following steps:

1. Implementation of the control sequence using CDL.
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. 11.1 shows the process of exporting and importing control sequences.

The next section describes three different approaches that can be used by control vendors to translate CDL to their product line:

1. Translation of the CDL-compliant sequence to a JSON intermediate format, which can be translated to the format used by the control platform (Section 11.3).
2. Export of the whole CDL-compliant sequence using the *FMI standard* (Section 11.4), a standard for exchanging simulation models that can be simulated using a variety of open-source tools.
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 11.5).

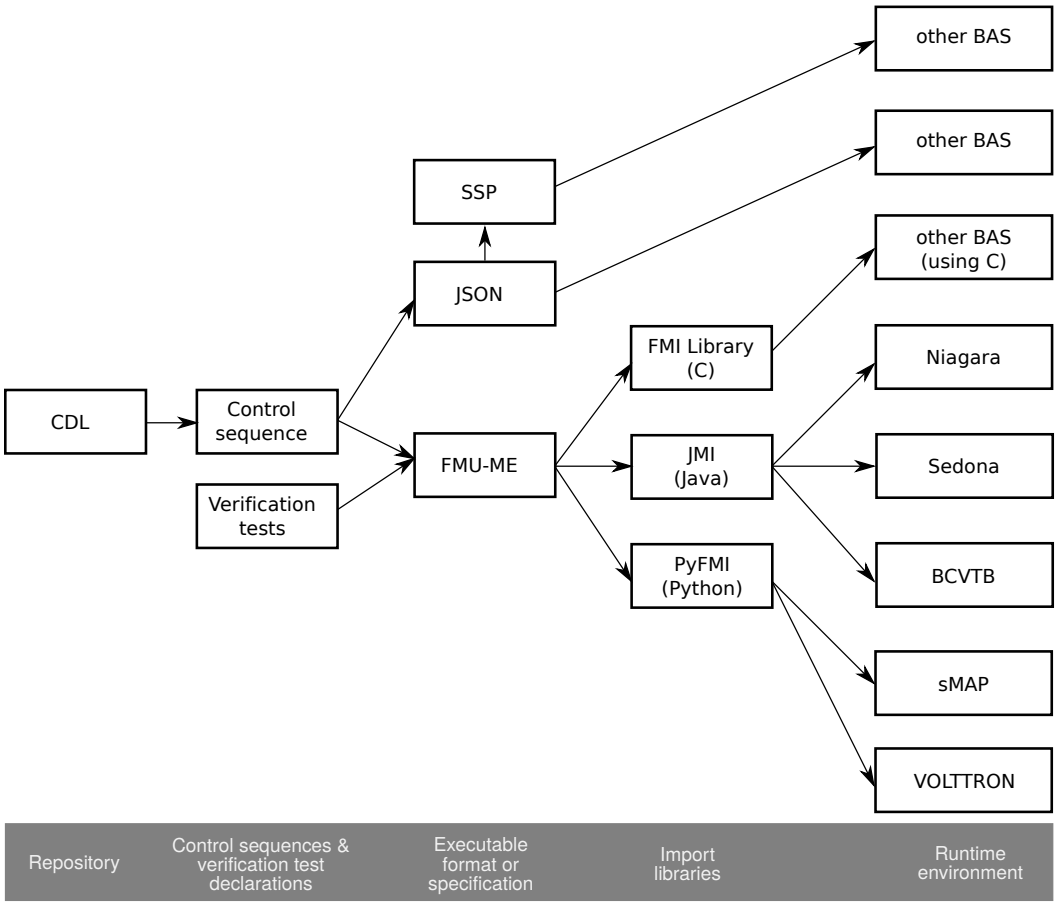


Fig. 11.1: *Overview of the code export and import of control sequences and verification tests.*

The best approach will depend on the control platform. While in the short-term, option 1) is likely preferred as it allows reusing existing control product lines, the long term vision is that control product lines would directly compile CDL using option 2) or 3). Before explaining these three approaches, we first discuss challenges of translation of CDL sequences to building automation systems, as well as their implications.

11.2 Challenges and Implications for Translation of Control Sequences from and to Building Control Product Lines

This section discusses challenges and implications for translating CDL-conforming control sequences to the programming languages used by building automation system.

First, we note that simply generating C code is not viable for such applications because building automation systems generally do not allow users to upload C code. Moreover, they also need to provide an interface for the building operator that allows editing the control parameters and control sequences.

Second, 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 (https://www.sedona-alliance.org/) 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.

11.3 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 target building automation system, the simplified JSON representation is most suited.

As an illustrative example, consider the composite control block shown in Fig. 7.3 and reproduced in Fig. 11.2.

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}})));
```

(continues on next page)

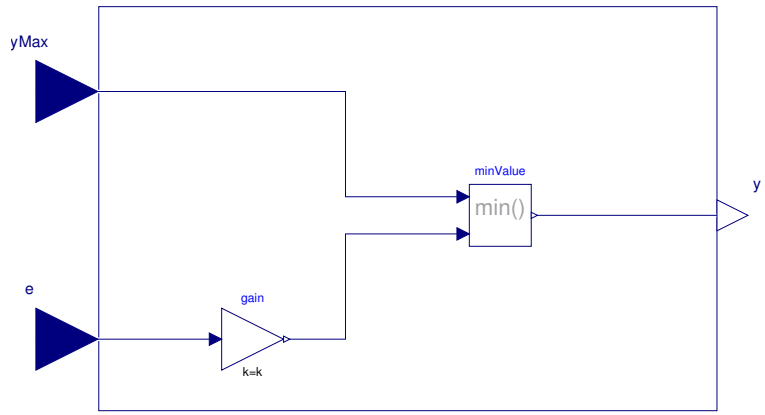


Fig. 11.2: Example of a composite control block that outputs $y = \max(k\,e, y_{max})$ where k is a parameter.

(continued from previous page)

```
  CDL.Interfaces.RealOutput y "Control signal"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
  CDL.Reals.MultiplyByParameter gain(final k=k) "Constant gain"
    annotation (Placement(transformation(extent={{-60,-50},{-40,-30}})));
  CDL.Reals.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;
```

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",
          "comment": "Control error"
        },
        {
          "className": "CDL.Interfaces.RealOutput",
          "name": "y",
          "comment": "Control signal"
        },
        {
          "className": "CDL.Reals.MultiplyByParameter",
          "name": "gain",
          "comment": "Constant gain",
          "modifications": [
            {
              "name": "k",
              "value": "k",
              "isFinal": true
            }
```


(continues on next page)

(continued from previous page)

```
          ]
        },
        {
          "className": "CDL.Reals.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"
        }
      ],
      [
        {
          "instance": "gain",
          "connector": "y"
        },
        {
          "instance": "minValue",
          "connector": "u2"
        }
      ],
      [
        {
          "instance": "minValue",
          "connector": "y"
```

(continues on next page)

(continued from previous page)

```
        },
        {
          "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.Reals.MultiplyByParameter` is an elementary CDL block (see Section 7.6). If it were a composite CDL block (see Section 7.12), 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.

The simplified JSON representation of a CDL sequence must be compliant with the corresponding JSON Schema. A JSON Schema describes the data format and file structure, lists the required or optional properties, and sets limitations on values such as patterns for strings or extrema for numbers.

The CDL Schema can be found at https://github.com/lbl-srg/modelica-json/blob/master/schema-cdl.json .

The program modelica-json automatically tests the JSON representation parsed from a CDL file against the schema right after it is generated.

The validation of an existing JSON representation of a CDL file against the schema can be done executing the command

```
node modelica-json/validation.js -f filename.json
```

Control providers can use the JSON Schema as a specification to develop a translator to a control product line. If JSON files are the starting point, then they should first validate the JSON files against the JSON Schema, as this ensures that the input files to the translator are valid.

11.4 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 JavaScript converter, see https://github.com/tshort/openmodelica-javascript.

To implement control sequences, blocks from the CDL library (Section 7.6) 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 OpenModelica, Impact, OPTIMICA or Dymola can be used. For example, with OPTIMICA a control sequence can be exported using the Python commands

```
from pymodelica import compile_fmu
compile_fmu("Buildings.Controls.OBC.ASHRAE.G36.AHUs.SingleZone.VAV.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://www.energy.gov/eere/buildings/volttron):
  - PyFMI (https://pypi.org/pypi/PyFMI)
- Tools based on Java:
  - Building Controls Virtual Test Bed (https://simulationresearch.lbl.gov/bcvtb)
  - JFMI (https://ptolemy.berkeley.edu/java/jfmi/)
  - JavaFMI (https://bitbucket.org/siani/javafmi/wiki/Home)
- Tools based on C:
  - FMI Library (https://github.com/modelon-community/fmi-library)
- Modelica tools, of which many if not all provide functionality for real-time simulation:
  - OpenModelica (https://openmodelica.org/)
  - Impact (https://www.modelon.com/modelon-impact/)
  - Dymola (https://www.3ds.com/products-services/catia/products/dymola/)
  - MapleSim (https://www.maplesoft.com/products/maplesim/)
  - SimulationX (https://www.esi-group.com/products/system-simulation)
  - SystemModeler (https://www.wolfram.com/system-modeler/index.html)

See also https://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.

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

In 2019, a new standard called System Structure and Parameterization (SSP) was released (https://ssp-standard.org/). 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 available, see https://ssp-standard.org/tools/.

CDL-compliant control sequences could be exported to the SSP standard as shown in Fig. 11.3.

In such a workflow, a control vendor would translate the elementary CDL blocks (Section 7.6) 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 11.3. 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 to

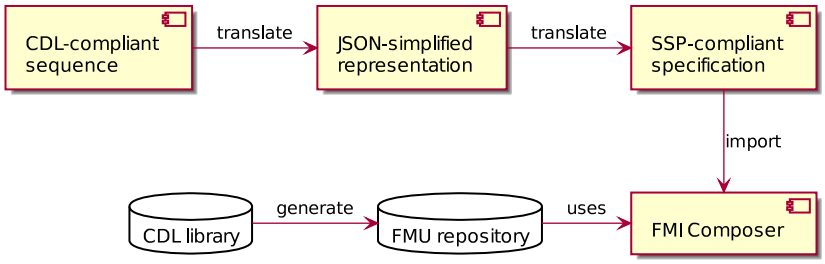


Fig. 11.3: Translation of CDL to SSP.

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.

11.6 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.)

11.6.1 Substitutions that Give Identical Control Response

Consider the gain `CDL.Reals.MultiplyByParameter` 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.

11.6.2 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 [Mod23] 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.

11.6.3 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 11.6.2.

Chapter 12

Verification

12.1 Introduction

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 note 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.

12.2 Terminology

We will use the following terminology, see also Section 7 for more details.

By a *real controller*, we mean a control device implemented in a building automation system.

By a *controller*, we mean a Modelica block that conforms to the CDL specification and that contains a control sequence.

By *input* and *output*, we mean the input connectors (or ports) and output connector (or ports) of a (real) controller.

By *input value* or *output value*, we mean the value that is present at an input or output connector at a given time instant.

By *time series*, we mean a series of values at successive times. The time stamps of the series need not be equidistant, but they need to be non-decreasing, e.g., we allow for time series with two equal time stamps to indicate when a values switches.

By *signal*, we mean a function that maps time to a value.

By *parameter*, we mean a configuration value of a controller that is constant, unless it is changed by an operator or by the user who runs the simulation. Typical parameters are sample times, dead bands or proportional gains.

12.3 Scope of the Verification

For OpenBuildingControl, we currently only verify the implementation of the control sequence. The verification is done by comparing output time series between a real controller and a simulated controller for the same input time series and the same control parameters. The comparison checks whether the difference between these time series are within a user-specified tolerance. Therefore, with our tests, we aim to verify that the control provider implemented the sequence as specified, and that it executes correctly.

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 envelope is meeting its specification.

12.4 Methodology

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

1. Whether the trended output time series and the output time series computed by the CDL specification are close to each other.
2. Whether the trended input and output time series 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. 12.1 shows the flow diagram for the verification. Rather than using real-time data through BACnet or other protocols, set points, input time series and output time series 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 time series 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 time series, 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 time series, it outputs the control time series according to the specification. The block labeled *time series verification* compares these time series with trended control time series, and indicates where the time series differ by more than a prescribed tolerance in time and in control variable 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*.

Note: We also considered testing 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

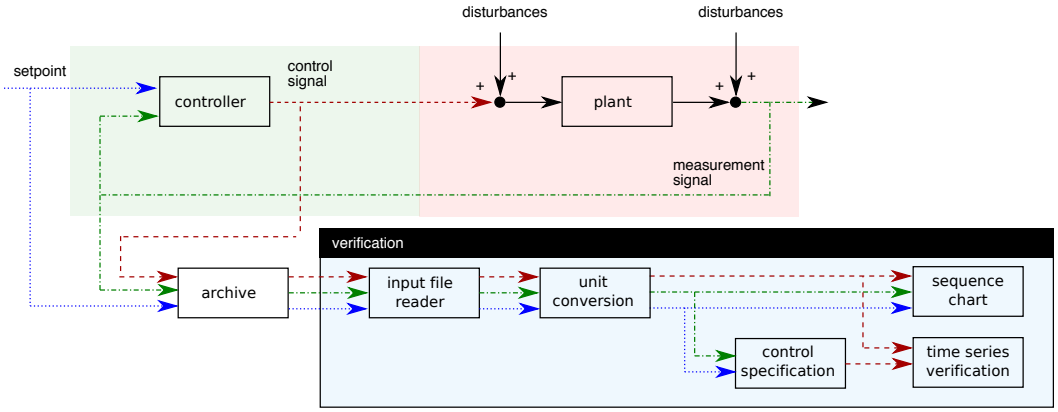


Fig. 12.1: Overview of the verification that tests whether the installed control sequence meets the specification.

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.

12.5 Modules of the Verification Test

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

12.5.1 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.

12.5.2 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.

12.5.3 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. This can be done in two ways:

- Import the module `pyfunnel` and use the `compareAndReport` and `plot_funnel` functions. Fig. 12.2 shows a typical plot generated by use of these functions.
- Run directly the Python script from terminal. For usage information, run `python pyfunnel.py --help`.

For the full documentation of the funnel software, visit https://github.com/lbl-srg/funnel

12.5.4 Verification of Sequence Diagrams

To verify sequence diagrams we developed the Modelica package `Buildings.Utilities.IO.Plotters`. Fig. 12.3 shows an example in which this block is used to produce the sequence diagram shown in Fig. 12.4. While in this example, we used the control output time series of the CDL implementation, during commissioning, one would use the controller output time series 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. 12.3 generates an html file that contains the scatter plots shown in Fig. 12.5.

12.6 Example

In this example we validated a trended output time series 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

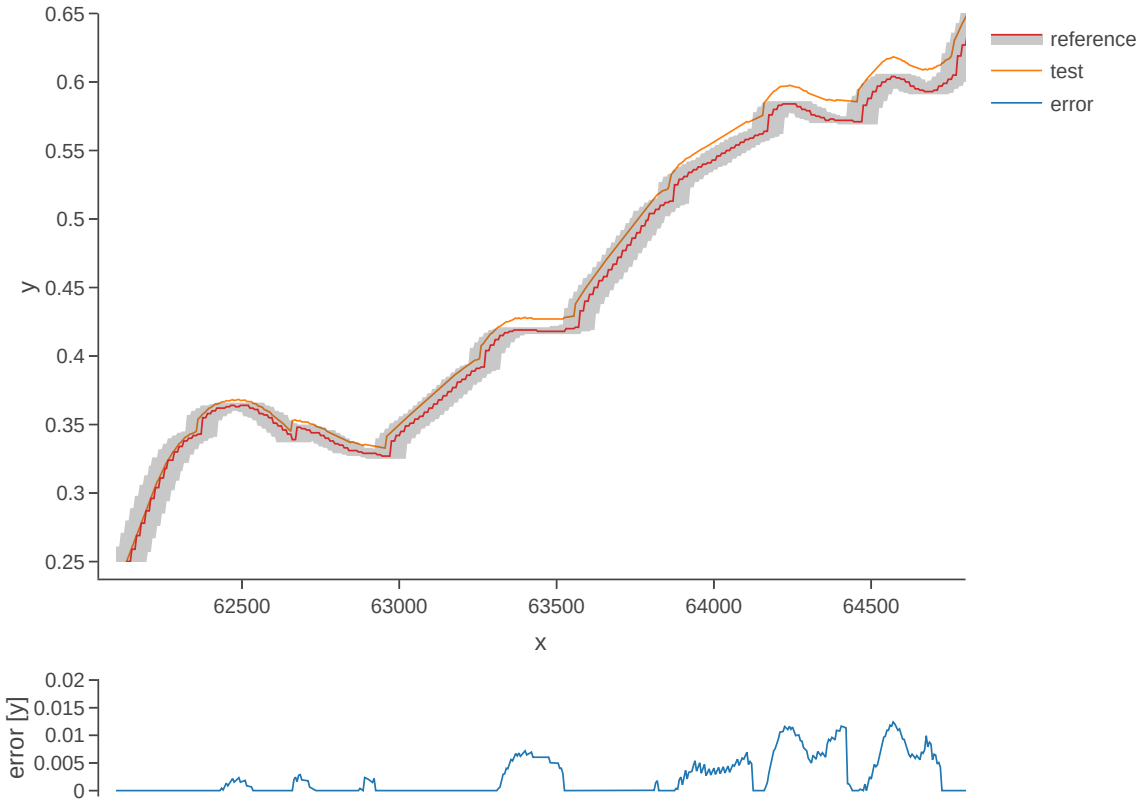


Fig. 12.2: Typical plot generated by `pyfunnel.plot_funnel` *for comparing test and reference time series.*

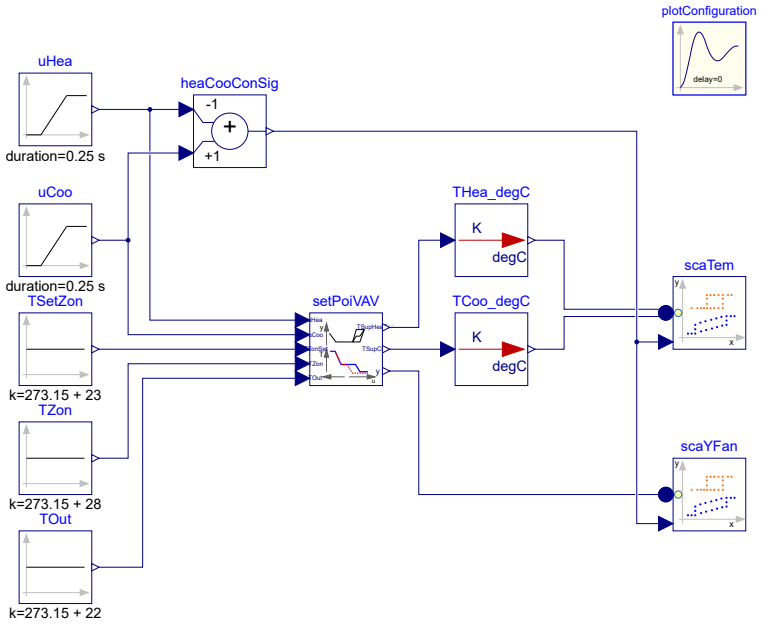


Fig. 12.3: *Modelica model that verifies the sequence diagram. On the left are the blocks that generate the control input time series. In a real verification, these would be replaced with a file reader that reads data that have been archived by the building automation system. In the center is the control sequence implementation. Some of its output values are 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.*

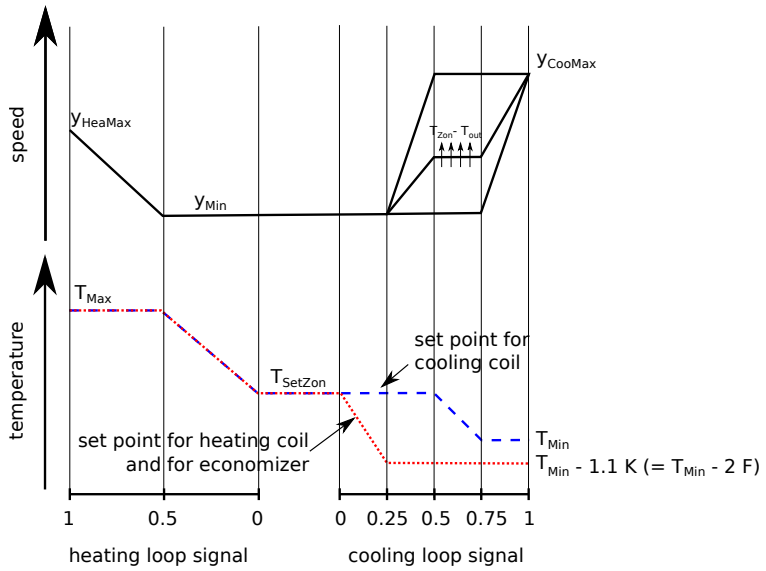


Fig. 12.4: *Control sequence diagram for the VAV single zone control sequence from ASHRAE Guideline 36.*

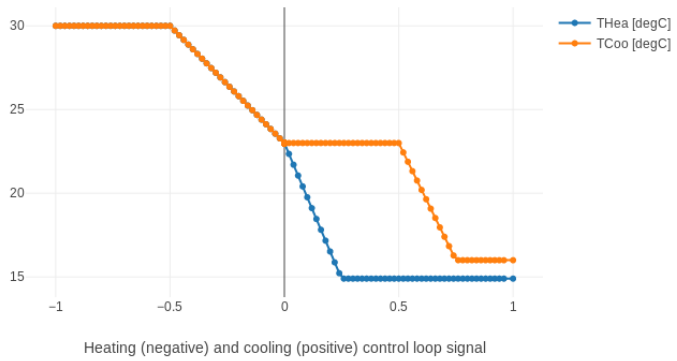


Fig. 12.5: *Scatter plots that show the control sequence diagram generated from the simulated sequence.*

campus in Berkeley, CA. The subsequence is shown in Fig. 12.6. 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. 12.7, to validate the trended output time series. We trended controller inputs and outputs 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 trended input and output time series 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 trended input and output time series, we recorded all parameters, such as the hysteresis offset (see Fig. 12.8) and the controller gains (see Fig. 12.9), to use them in the CDL controller.

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.Reals.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 $k_{p,cdl} = u\, k_{p,alc}$, 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. 12.10 shows the Modelica model that was used to conduct the verification. On the left hand side are the data readers

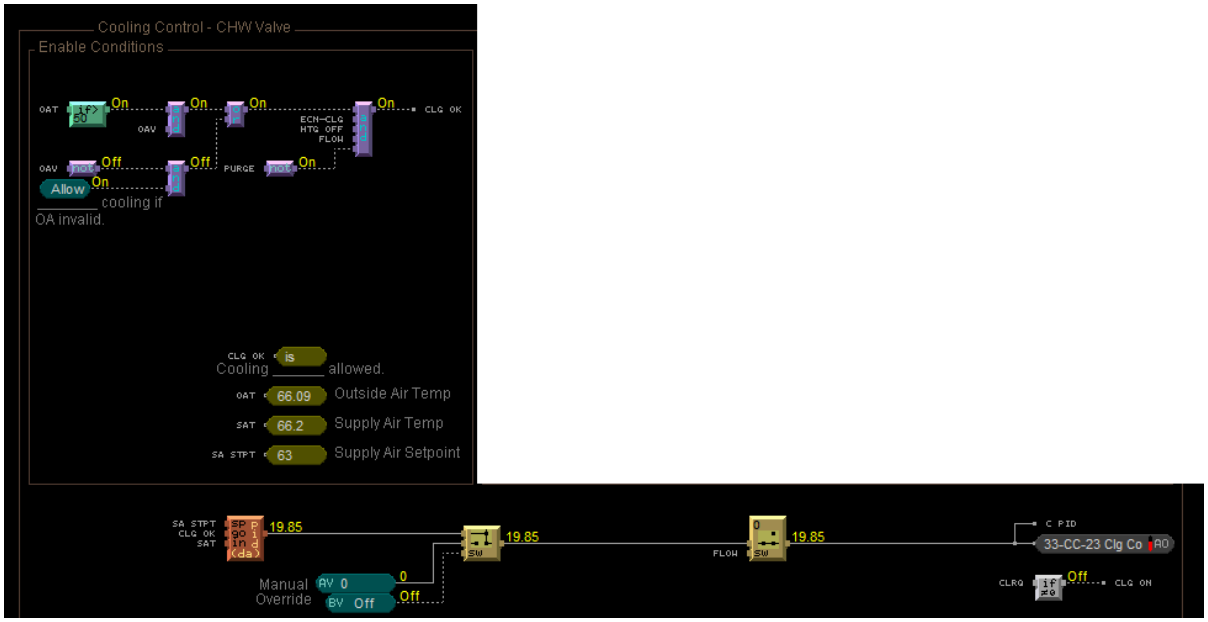


Fig. 12.6: ALC EIKON specification of the cooling coil valve position control sequence.

that read the trended input and output time series 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. 12.7. The plotters on the right hand side then compare the simulated cooling coil valve position with the trended time series.

Fig. 12.11, 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. 12.12.

12.7 Specification for Automating the Verification

The example Section 12.6 describes a manual process of composing the verification model and executing the verification process. In this section, we provide specifications for how this process can be automated. The automated workflow uses the same modules as in Section 12.6, except that the unit conversion will need to be done by the tool that reads the CSV files and sends data to the Building Automation System, and that reads data from the Building Automation System and

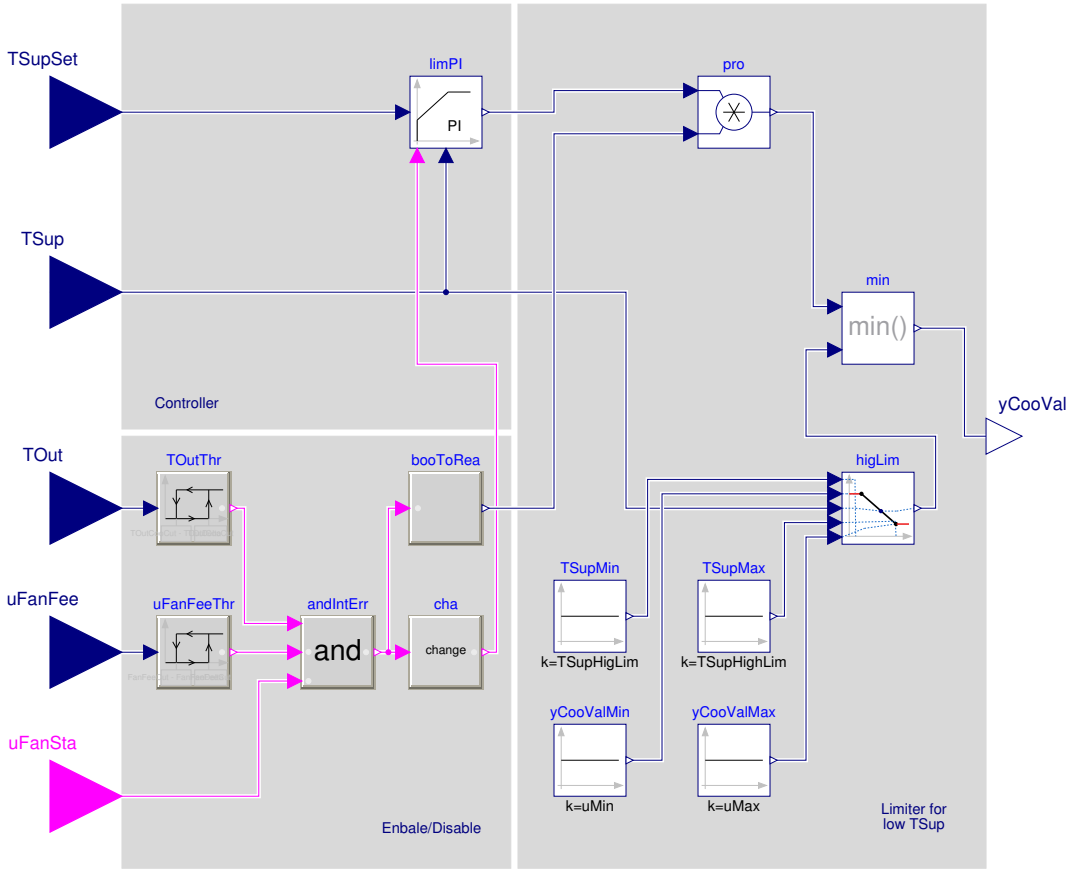


Fig. 12.7: CDL specification of the cooling coil valve position control sequence.

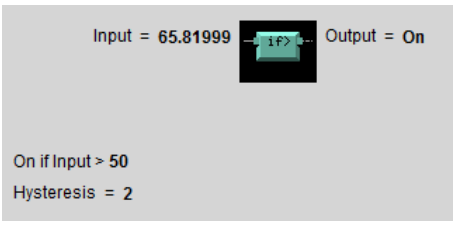


Fig. 12.8: ALC EIKON outdoor air temperature hysteresis to enable/disable the controller.

Fig. 12.9: ALC EIKON PI controller parameters.

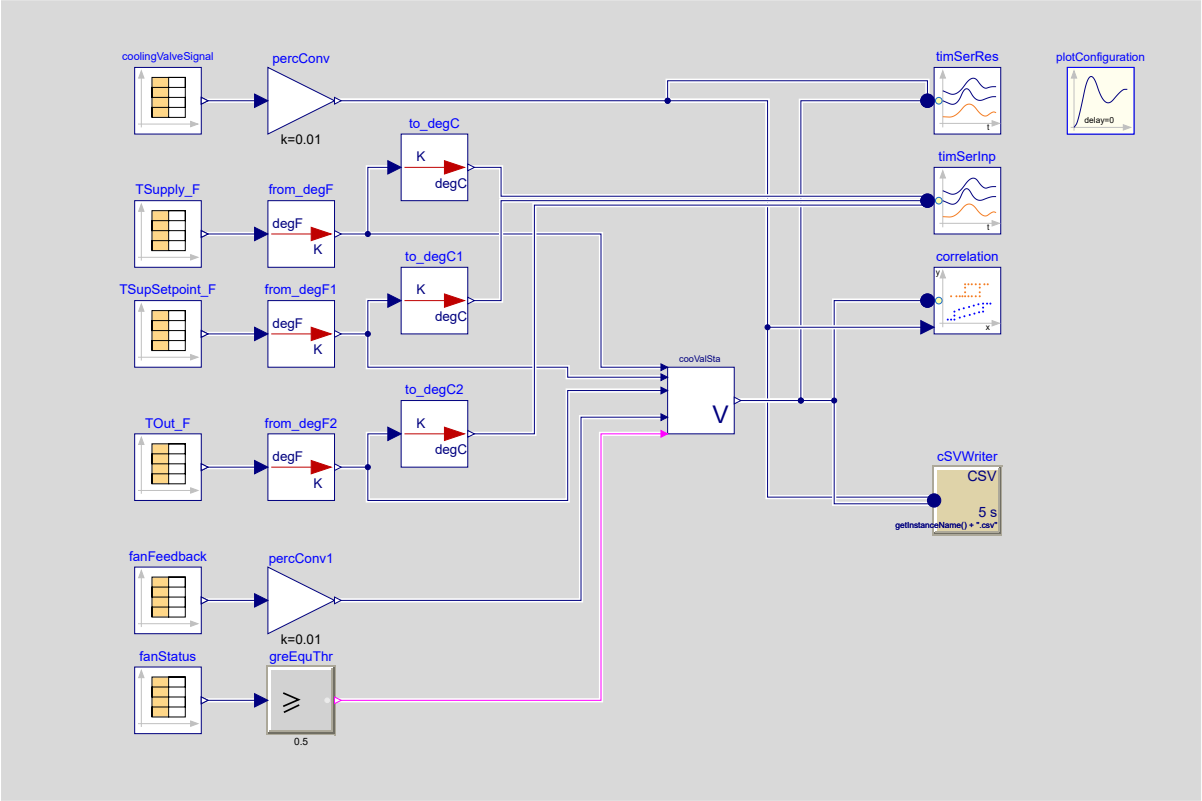
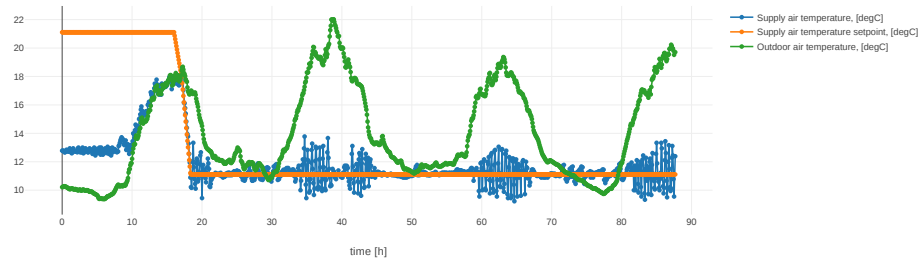
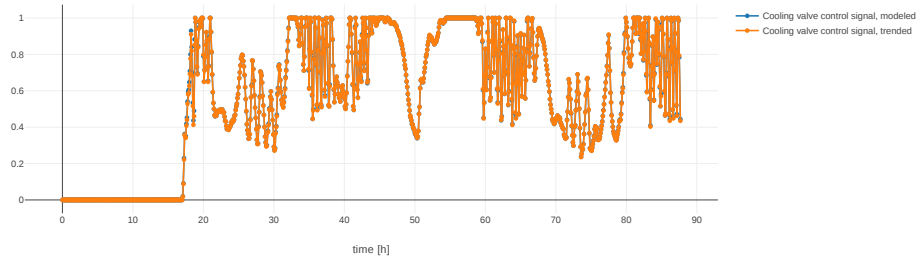


Fig. 12.10: Modelica model that conducts the verification.

Trended input signals



Cooling valve control signal: reference trend vs. modeled result



Modeled result/recorded trend correlation

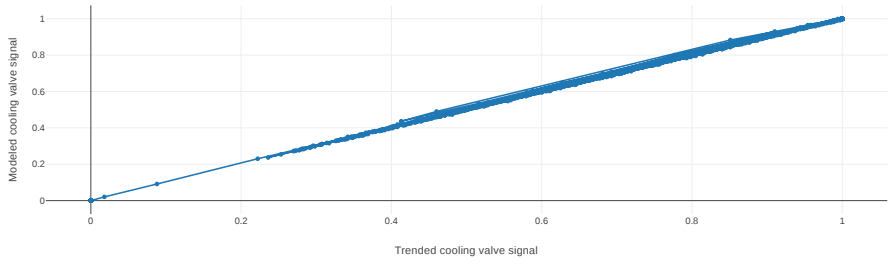


Fig. 12.11: Verification of the cooling valve control signal between ALC EIKON computed signal and simulated signal.

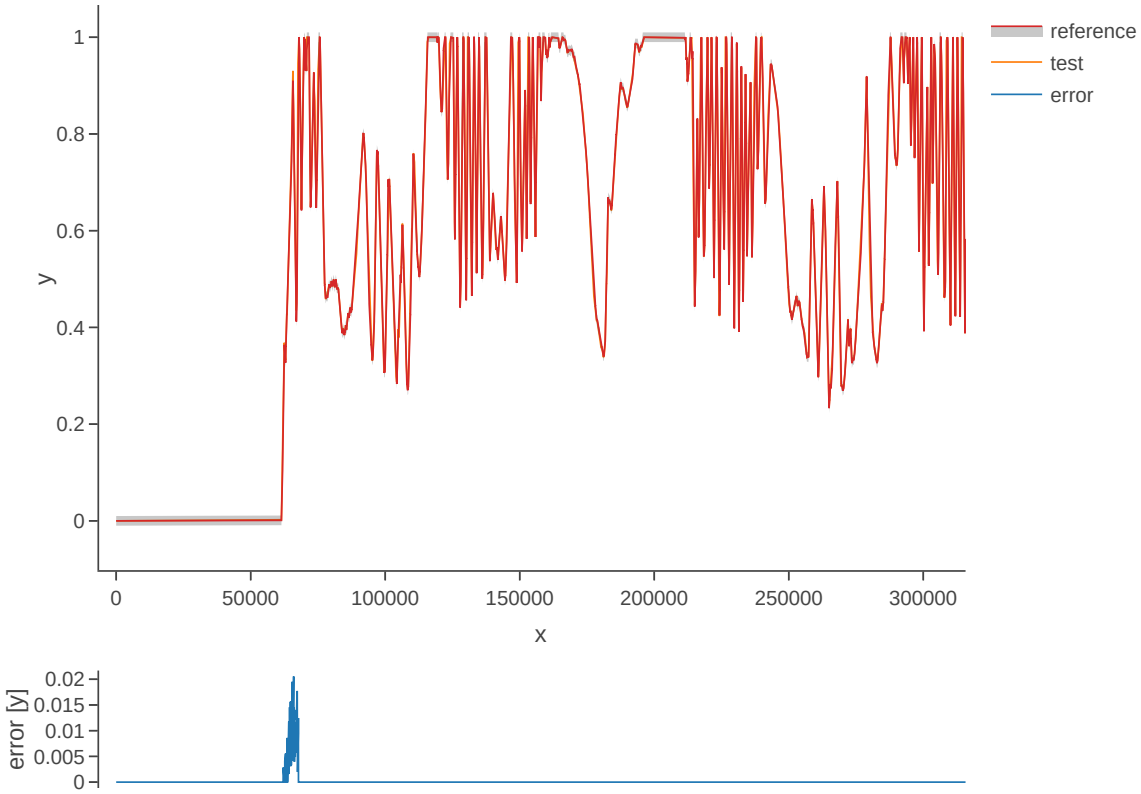


Fig. 12.12: Verification of the cooling valve control signal with the funnel software (error computed with an absolute tolerance in time of 1 s and a relative tolerance in y *of 1%).*

writes them to the CSV files. This design decision has been done because CDL provides all required unit information, but this is not the case in general for a building automation system. Moreover, in the process described in this section, the CSV files will be read directly by the Modelica simulation environment rather than using the CSV file reader described in Section 12.5.1.

12.7.1 Use Cases

We address two use cases. Both uses cases verify conformance of the time series generated by a control control sequence specified in CDL against the time series of an implementation of a real controller. For both use cases, the precondition is that one control sequence, or several control sequences, are available in CDL. The output will be a report that describes whether the real implementation conforms to the CDL implementation within a user-specified error tolerance. The difference between the two uses cases is as follows: In scenario 1, the CDL model contains the controller that is connected to upstream blocks that generate the control input time series. The time series from this CDL model will be used to test the real controller. In scenario 2, data trended from a real controller will be used to verify the controller against the output time series of its CDL specification, using as inputs and parameters of the CDL specification the trended time series and parameters of the real controller.

To conduct the verification, the following three steps will be conducted:

1. Specify the test setup,
2. generate data from the real controller, and
3. produce the test results.

Next, we will describe the specifications for the two scenarios. The specifications focus on the CDL side. In addition, for Scenario 1, steps 5 & 6, and for Scenario 2, steps 3 & 4, a data collection tool need to be developed that utilizes the JSON and CSV files described below and does the following to generate the data from the real controller:

1. Identifies which objects in the building automation system match with the desired collection.
2. Shows the user a list of all objects that don't match and a list of objects from the building automation system and allows for the user to manually match them.
3. Sets up the data collection.
4. Starts collecting data at the desired intervals.
5. Store the data.
6. Export the desired data in the format specified below.

Note: In support of this step, work is ongoing in exporting semantic information from the CDL implementation.

12.7.2 Scenario 1: Control Input Obtained by Simulating a CDL Model

For this scenario, we verify whether a real controller outputs time series that are similar to the time series of a controller that is implemented in a CDL model. The inputs of the real controller will be connected to the time series that were exported when simulating a controller that is connected to upstream blocks that generate the control input time series.

An application of this use case is to test whether a controller complies with the sequences specified in CDL for a given input time series and control parameters, either as part of verifying correct implementation during control development, or verifying correct implementation in a Building Automation System that allows overwriting control input time series.

We have also developed a verification tool for verifying the control sequences implemented in a controller using CDL.

For this scenario, we are given the following data:

i. A list of CDL models to be tested.
ii. Relative and absolute tolerances, either for all output variables, or optionally for individual output variables of the sequence.
iii. Optionally, a boolean variable in the model that we call an indicator variable. An indicator variable allows to indicate when to pause a test, such as during a fast transient, and when to resume the test, for example when the controls is expected to have reached steady-state. If its value is `true`, then the output should be tested at that time instant, and if it is `false`, the output must not be tested at that time instant.

For example, consider the validation test OBC.ASHRAE.G36.AHUs.SingleZone.VAV.SetPoints.Validation.Supply_u. and suppose we want to verify the sequences of its instances `setPoiVAV` and `setPoiVAV1`. To do so, we first write a specification as shown in Listing 12.1.

Listing 12.1: Configuration of test setup.

```
{
  "references": [
    {
      "model": "Buildings.Controls.OBC.ASHRAE.G36.AHUs.SingleZone.VAV.SetPoints.Validation.
↪Supply_u",
      "generateJson": false,
      "sequence": "setPoiVAV",
      "pointNameMapping": "realControllerPointMapping.json",
      "runController": false,
      "controllerOutput": "test/real_outputs.csv"
    },
    {
      "model": "Buildings.Controls.OBC.ASHRAE.G36.AHUs.SingleZone.VAV.SetPoints.Validation.
↪Supply_u",
      "generateJson": true,
      "sequence": "setPoiVAV1",
      "pointNameMapping": "realControllerPointMapping.json",
      "runController": true,
      "controllerOutput": "test/real_outputs.csv",
      "outputs": {
        "setPoiVAV1.TSup*": { "atoly": 0.5 }
      },
      "indicators": {
        "setPoiVAV1.TSup*": [ "fanSta.y" ]
      },
      "sampling": 60
    }
  ],
  "modelJsonDirectory": "test",
  "tolerances": { "rtolx": 0.002, "rtoly": 0.002, "atolx": 10, "atoly": 0 },
```

(continues on next page)

(continued from previous page)

```
  "sampling": 120,
  "controller": {
    "networkAddress": "192.168.0.115/24",
    "deviceAddress": "192.168.0.227",
    "deviceId": 240001
  }
}
```

This specifies two tests, one for the controller `setPoiVAV` and one for `setPoiVAV1`. In this example, `setPoiVAV` and `setPoiVAV1` happen to be the same sequence, but their input time series and/or parameters are different, and therefore their output time series will be different. The `generateJson` flag will determine if the json translation for the specified model under test must be generated during the test using the `modelica-json` tool. If it is set to `false`, the software assumes that the json translation is already present in `modelJsonDirectory`. The test for `setPoiVAV` will use the globally specified tolerances, and use a sampling rate of 120 seconds. The mapping of the variables to the I/O points of the real controller is provided in the file `realControllerPointMapping.json`, shown in Listing 12.2. The test `setPoiVAV` will not run the controller during the test because of the specification `runController = false`. Rather, it will use the saved results `test/real_outputs.csv` from a previous run. The test for `setPoiVAV1` will use different tolerances on each output variable that matches the regular expression `setPoiVAV1.TSup*`. Moreover, for each variable that matches the regular expression `setPoiVAV1.TSup*`, the verification will be suspended whenever `fanSta.y = false`. The sampling rate is 60 seconds. This test will also use `realControllerPointMapping.json` to map the variables to points of the real controller. Because `runController = true`, this test will run the controller in real-time and save the time-series of the output variables in the file specified by `controllerOutput`. The real controller's network configuration can be found under the `controller` section of the configuration. The `networkAddress` is the controller's BACnet subnet, the `deviceAddress` is the controller's IP address and the `deviceId` is the controller's BACnet device identifier. The tolerances `rtolx` and `atolx` are relative and absolute tolerances in the independent variable, e.g., in time, and `rtoly` and `atoly` are relative and absolute tolerances in the control output variable.

Listing 12.2: Example `pointNameMapping` file.

```
[
   {
       "cdl": {   "name": "TZonCooSetOcc",                "unit": "K",    "type": "float"},
       "device": {"name": "Occupied Cooling Setpoint_1",   "unit": "degF", "type": "float"}
   },
   {
       "cdl": {   "name": "TZonHeaSetOcc",                "unit": "K",    "type": "float"},
       "device": {"name": "Occupied Heating Setpoint_1",   "unit": "degF", "type": "float"}
   },
   {
       "cdl": {   "name": "TZonCooSetUno",                "unit": "K",    "type": "float"},
       "device": {"name": "Unoccupied Cooling Setpoint_1", "unit": "degF", "type": "float"}
   },
   {
       "cdl": {   "name": "TZonHeaSetUno",                "unit": "K",    "type": "float"},
       "device": {"name": "Unoccupied Heating Setpoint_1", "unit": "degF", "type": "float"}
   },
```

(continues on next page)

(continued from previous page)

```
  {
      "cdl": {   "name": "setAdj",                          "unit": "K",    "type": "float"},
      "device": {"name": "setpt_adj_1",                     "unit": "degF", "type": "float"}
  },
  {
      "cdl": {   "name": "heaSetAdj",                       "unit": "K",    "type": "float"},
      "device": {"name": "Heating Adjustment_1",            "unit": "degF", "type": "float"}
  },
  {
      "cdl": {   "name": "uOccSen",                                          "type": "int"},
      "device": {"name": "occ_sensor_bni_1",                                 "type": "bool"}
  },
  {
      "cdl": {   "name": "uWinSta",                                          "type": "int"},
      "device": {"name": "window_sw_1",                                      "type": "bool"}
  },
  {
      "cdl": {   "name": "TZonCooSet",                     "unit": "K",    "type": "float"},
      "device": {"name": "Effective Cooling Setpoint_1", "unit": "degF", "type": "float"}
  },
  {
      "cdl": {   "name": "TZonHeaSet",                     "unit": "K",    "type": "float"},
      "device": {"name": "Effective Heating Setpoint_1", "unit": "degF", "type": "float"}
  }
]
```

Listing 12.2 is an example of the `pointNameMapping` file. It is a list of dictionaries, with each dictionaries having two parts: The `cdl` part specifies the `name`, the `unit` and the `type` of the point in the CDL sequence. Similarly, the `device` part specifies this information for the corresponding point in the real controller. The `type` refers to the data type of the variable in the specific context, i.e., in CDL or in the actual controller. It should also be noted that some points may not have a unit, but only have a type. For example, the input `uOccSen` is a CDL point that is 1 if there is occupancy and 0 otherwise.

To create test input and output time series, we generate CSV files. This needs to be done for each controller (or control sequence) under test, and we will explain it only for the controller `setPoiVAV`. For brevity, we call `OBC.ASHRAE.G36.AHUs.SingleZone.VAV.SetPoints.Validation.Supply_u` simply `Supply_u`.

Once we have the configuration and the `pointNameMapping` file set up, the sequence verification (handled by the verification tool) goes through the following steps:

1. Generate a json translation of the modelica code. Currently the verification tool does invoke the `modelica-json` tool from within itself, depending on the `generateJson` flag in the configuration (and stores the output in the directory mentioned under `modelJsonDirectory`). The user can themselves invoke the `modelica-json` tool using:

   ```
   node app.js -f Buildings/Controls/OBC/ASHRAE/G36/AHUs/SingleZone/VAV/SetPoints/Validation/
   ↪Supply_u.mo -o json -d test
   ```

   This will produce `Supply_u.json` (file name is abbreviated) in the output directory `test`. See https://github.com/

lbl-srg/modelica-json for the json schema.
2. From `Supply_u.json`, extract all input and output variable declarations of the instance `setPoiVAV` and generate an I/O list . The tool also extracts public parameters of the instance `setPoiVAV` and stores them. For this sequence, the public parameters are `TSupSetMax`, `TSupSetMin`, `yHeaMax`, `yMin` and `yCooMax`.
3. Obtain reference time series by simulating `Supply_u.mo` with time series of all input, output and indicator time series. The verification tool accomplishes this by using the free open-source tool OpenModelica. The verification tool will create Modelica scripts to translate the model, followed by one to simulate the model. This will produce a `Supply_u_res.mat` file, from which the tool will extract the timeseries of the inputs and the outputs and store it as `Supply_u_res.csv`.
   More information about the python script used to run the OpenModelica simulation can be found at software/verification/openmodelica_sim.py.
4. Using the input and output variables extracted for the sequence `setPoiVAV`, the verification tool then separates the input and the output timeseries (reference outputs).
5. Steps 6 and 7 are applied only if `runController` flag in the test configuration file is set to `True`. Else, the tool will use the real outputs previously generated by the controller and saved in the file mentioned under `controllerOutput`. Proceed to step 8.
6. If the `runController` flag is `True`, the verification applies the parameters that have been extracted to the real controller, and runs the real controller for the input time series extracted in the step above. Using the `pointNameMapping` file, the tool will also handle the unit conversions and the type conversions on the time series as needed for the controller under test.
7. As the controller is being set different input values, the output variables are trended and saved to `setPoiVAV_real_outputs.csv`. The point names, units and the types of the output time series will also be converted to match the CDL input timeseries as specified in the `pointNameMapping` file.
8. Produce the test results by running the funnel software (https://github.com/lbl-srg/funnel) for each time series of the output variables generated by the controller (`setPoiVAV_output.csv` or file in `controllerOutput`) against the corresponding output variables generated by the CDL simulation. Before sending the time series to the funnel software, set the value of the reference and the controller output to zero whenever the indicator function is zero for that time stamp. This will exclude the value from the verification. This will give, for each time series, output files that show where the error exceeds the specified tolerance.

The sequence above can be run for each test case, and the results from step 8 are to be used to generate a test report for all tested sequences.

An example of a sequence under test, along with real inputs from a controller have been included in the verification tool software. Please see software/verification for how to automate this process.

12.7.3 Scenario 2: Control Input Obtained by Trending a Real Controller

For this scenario, we verify whether a real controller produces time series that are similar to the time series of a controller that is implemented in a CDL model. As control input time series, the time series trended from the real controller are used.

An applications of this use case is to test if a controller complies with the sequences specified in CDL for already trended data.

For this scenario, we are given the following data:

i. The CDL class name of the control sequence to be tested, in our example `Buildings.Controls.OBC.ASHRAE.G36.AHUs.SingleZone.VAV.SetPoints.Supply`.

ii. Relative and absolute tolerances, either for all output variables, or optionally for individual output variables of the sequence.

Therefore, a test specification looks as shown in Listing 12.3, which is identical to Listing 12.1, except that the elements *indicator* and *sampling* are removed because a sequence cannot have an indicator function, and because CDL simulators control the accuracy and hence a sampling time step is not needed. However, a time series for an indicator function can be provided, see step 4 below.

Listing 12.3: Specification of test setup.

```
references : [
  { "model": "Buildings.Controls.OBC.ASHRAE.G36.AHUs.SingleZone.VAV.SetPoints.Supply" },
    "tolerances": {"atoly": 0.5, "variable": "TSup*" },
  }
],
"tolerances": {"rtolx": 0.002, "rtoly": 0.002, "atolx": 10, "atoly": 0},
```

Note that we allow for multiple entries in `references` to allow testing more than one sequence.

To create test input and output time series, we generate again CSV files. This needs to be done for each control sequence. Here, we only explain it for the one sequence shown in Listing 12.3.

The procedure is as follows:

1. Produce the json file `Supply.json` (name abbreviated) by running `modelica-json` as

   ```
   node app.js -f Buildings/Controls/OBC/ASHRAE/G36/AHUs/SingleZone/VAV/SetPoints/Supply.mo -
   ↪o json -d test1
   ```

2. Generate the list of input and output variable declarations `reference_io.json` and the parameter list `reference_parameters.json` as in Step 2 in Section 12.7.2.
3. Trend the input and output time series specified in `reference_io.json` from the real controller, trending as input time series whatever the controller receives from the actual building automation system. (However, make sure there is reasonable excitation of the control input.)
4. Convert the trended input time series of the real controller to the units specified in `reference_io.json`, and write the converted input time series to a new file `reference_input.csv`, using the format

   ```
   time, uHea, uCoo, TZonSet,    TZon,    TOut, uFan
      0,    1,    0,  293.15,  292.15,  283.15,    1
     60,  0.5,    0,  293.15,  292.15,  283.15,    1
    120,    0,  0.5,  293.15,  292.15,  283.15,    1
    180,    0,    1,  293.15,  292.15,  283.15,    1
   3600,    0,    1,  293.15,  292.15,  283.15,    1
   ```

   where the first column is time in seconds.
   Do the same for the trended output time series of the real controller and store them in the new file `controller_output.csv` that has the same format as `reference_input.csv`
   Optionally, also store one or several indicator time series in `indicator.csv`, with the header of each time series being the name of the control output variable whose verification should be suspended whenever the indicator time series is `0` at that time instant. For example, to suspend the verification of an output called `TSupCoo` between t = 120 and t = 600 seconds, the file `indicator.csv` looks like

```
time, TSupCoo
0, 1
120, 0
600, 1
```

5. Convert the parameter values for `TSupSetMax`, `TSupSetMin`, `yHeaMax`, `yMin` and `yCooMax` as used in the real controller to the units specified in `reference_parameters.json` and store them in a text file `reference_parameters.txt`. For our example, suppose this file is

```
TSupSetMax=303.15
TSupSetMin=289.15
yHeaMax=0.7
yMin=0.3
yCooMax=1
```

6. Simulate the sequence specified in the class definition `Supply.mo`, using the parameter values from `reference_parameters.txt` and the input time series from `reference_input.csv`.
   This can be accomplished with the free open-source tool OpenModelica by running

```
#~/bin/bash
set -e
export OPENMODELICALIBRARY=`pwd`:/usr/lib/omlibrary
python3 -i simulateCDL.py
rm -f Buildings.* 2&> /dev/null
```

   with the file `simulateCDL.py` being

```
import shutil
import os
from OMPython import OMCSessionZMQ

model="Buildings.Controls.OBC.ASHRAE.G36.AHUs.SingleZone.VAV.SetPoints.Supply"
parameters="(TSupSetMax=303.15, TSupSetMin=289.15, yHeaMax=0.7, yMin=0.3, yCooMax=1)"
omc = OMCSessionZMQ()
omc.sendExpression("loadModel(Buildings)")
omc.sendExpression("simulate({}, startTime=0, stopTime=3600, simflags=\"-csvInput␣
↪reference_input.csv\", outputFormat=\"csv\")".format(model))
shutil.move("{}_res.csv".format(model), "reference.csv")
```

   This will produce the CSV file `reference.csv` that contains all control input and output time series.
7. Produce the test results as in Step 7 in Section 12.7.2.

The sequence above can be run for each test case, and the results from step 7 are to be used to generate a test report for all tested sequences.

Chapter 13

Generating a Modelica Model from Semantic Model

In this section, we will specify how to generate a Modelica model from a semantic model of a building or a Heating, Ventilation and Air Conditioing (HVAC) system. This Modelica model then serves as the format from which control sequences (CDL and CXF), point lists, etc. can be exported.

Building semantic models provide information about the equipment topology and how they are connected. Semantic models enable assigning machine-readable metadata to control points and enable the development of portable analytics and control applications.

13.1 Workflow

The current workflow for generating a Modelica model (along with the control sequences) leverages the Templates package of in the Modelica Buildings Library (https://simulationresearch.lbl.gov/modelica/releases/v11.0.0/help/Buildings_Templates.html). As of Buildings version 11, there are three system templates - one for an Air Handling Unit (AHU), one for a Variable Air Volume (VAV) terminal and one for an air source heat pump plant. As the templates are used to generate the Modelica models, only these types of systems are covered. The process is described in Fig. 13.1

The software that implements this workflow is available at https://github.com/lbl-srg/obc/tree/master/software/s223ToMo.

13.2 Example

From a ASHRAE Standard 223P semantic model that describes two VAV boxes, Fig. 13.2 describes the workflow to query the necessary sensors and instantiate the corresponding Modelica models using the VAVBox template from the Modelica Buildings Library.

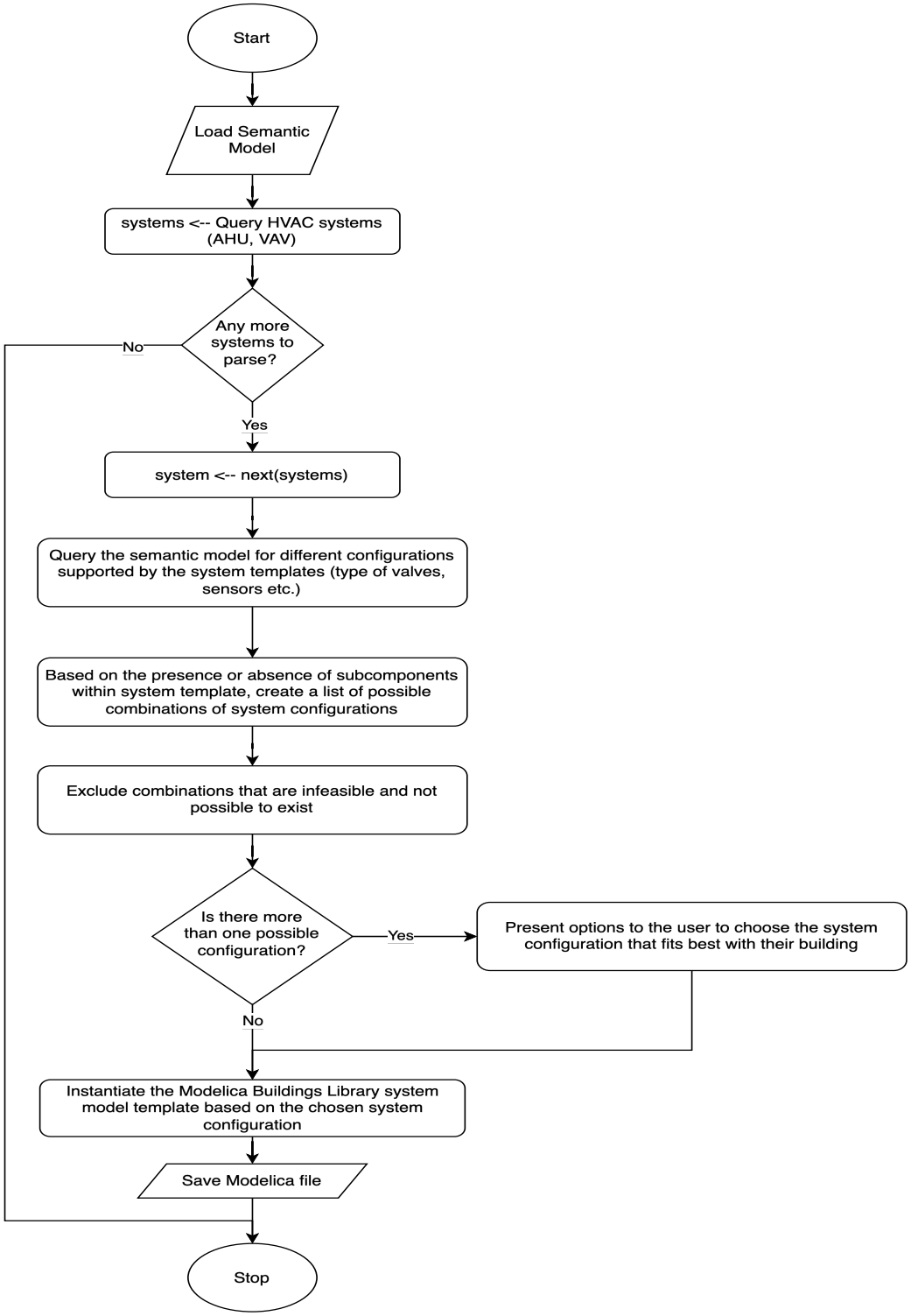


Fig. 13.1: Flowchart describing the workflow of generating a Modelica file from a semantic model of a HVAC system.

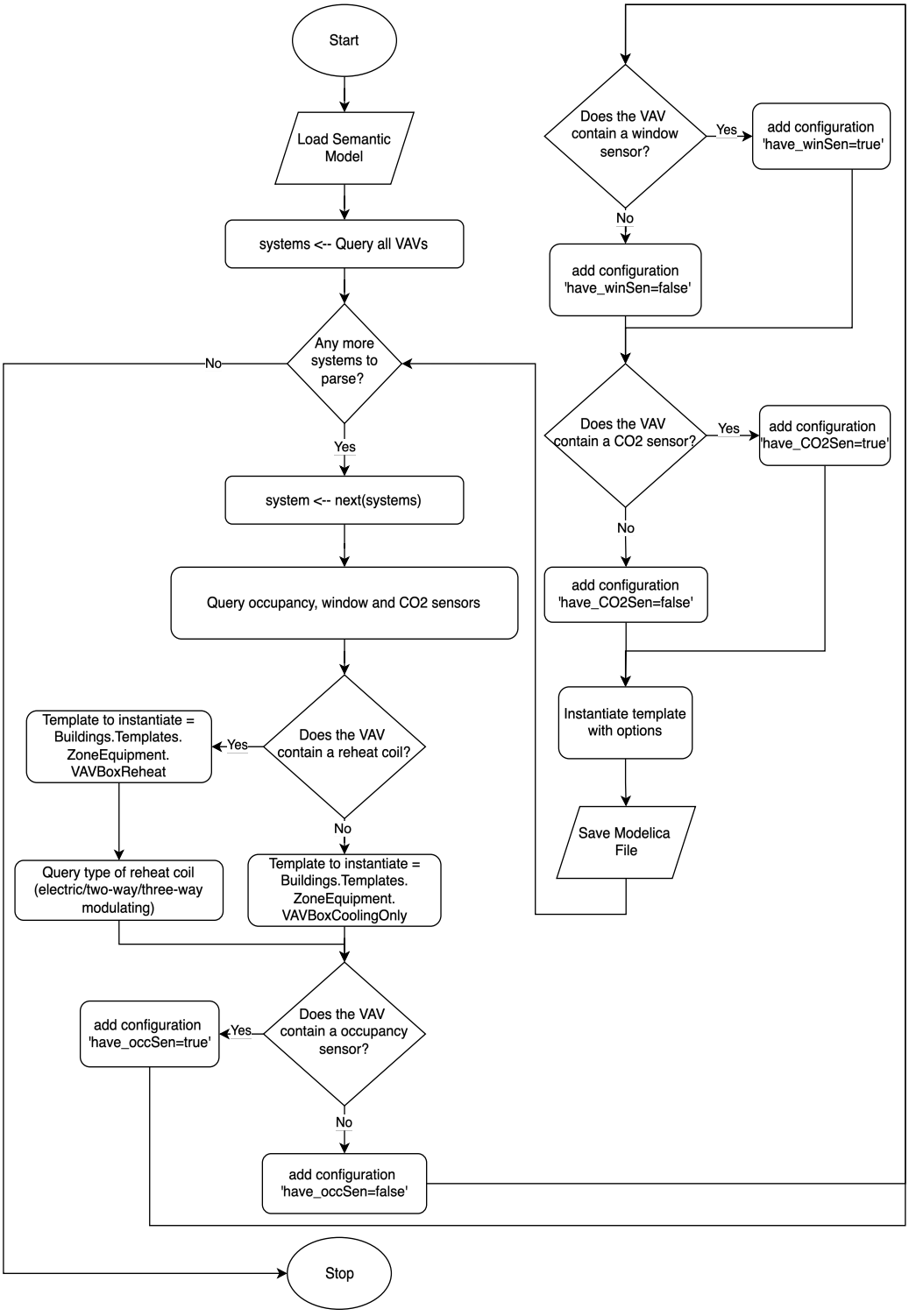


Fig. 13.2: *Flowchart describing the workflow of generating a Modelica model of two VAVBoxes described using the proposed ASHRAE Standard 223P.*

Chapter 14

Example Application

14.1 Introduction

In this section, we compare the performance of two different control sequences. The objectives are to demonstrate the setup for closed loop performance assessment, to demonstrate how to compare control performance, and to assess the difference in annual energy consumption.

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 [WZN11] and a model for air flow through building leakage and through open doors based on wind pressure and flow imbalance of the HVAC system [Wet06]. Thus, at every simulation step, a full pressure drop calculation is done to compute the air flow distribution based on damper positions, fan control signal and fan curve.

For the base case, 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 14.2.5, are as follows:

- The base case uses two different but constant supply air temperature setpoints for heating and cooling during occupied hours, whereas Guideline 36 case resets the supply air temperature setpoint based on outdoor air temperature and zone cooling requests, as obtained from the VAV terminal unit controllers. The reset is using the trim and respond logic.
- The base case resets the supply fan static pressure setpoint based on the VAV damper positions, whereas the Guideline 36 case resets the fan static pressure setpoint based on zone pressure requests from the VAV terminal controllers. The reset is 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 14.2 we describe the methodology, the models and the performance metrics, in Section 14.3 we compare the performance, in Section 14.4 we recommend improvements to the Guideline 36 and in

Section 14.5 we discuss the main findings and present concluding remarks.

14.2 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

14.2.1 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. 14.1 shows the schematic diagram of the HVAC system.

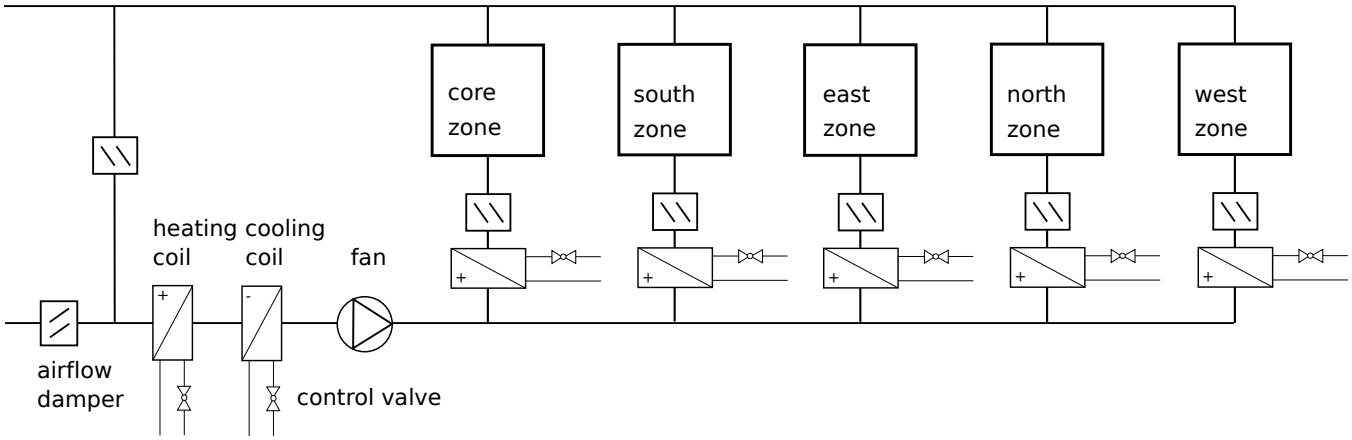


Fig. 14.1: *Schematic diagram of the HVAC system.*

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].

14.2.2 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.

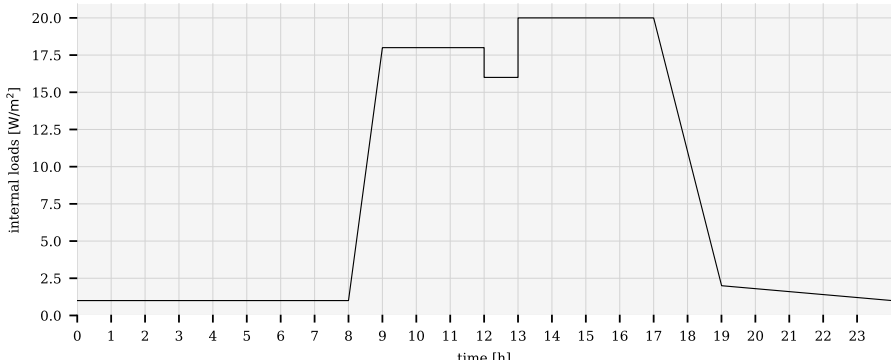


Fig. 14.2: *Internal load schedule.*

14.2.3 Internal Loads

We use an internal load schedule as shown in Fig. 14.2, of which 20% is radiant, 40% is convective sensible and 40% is latent. Each zone has the same internal load per floor area.

14.2.4 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].

14.2.5 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 modulated to maintain a duct static pressure setpoint. The duct static pressure setpoint 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 controlled 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. 14.6.

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 the Guideline 36 case, 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. 14.3 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. 14.4. 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. 14.5, 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.

14.2.6 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.

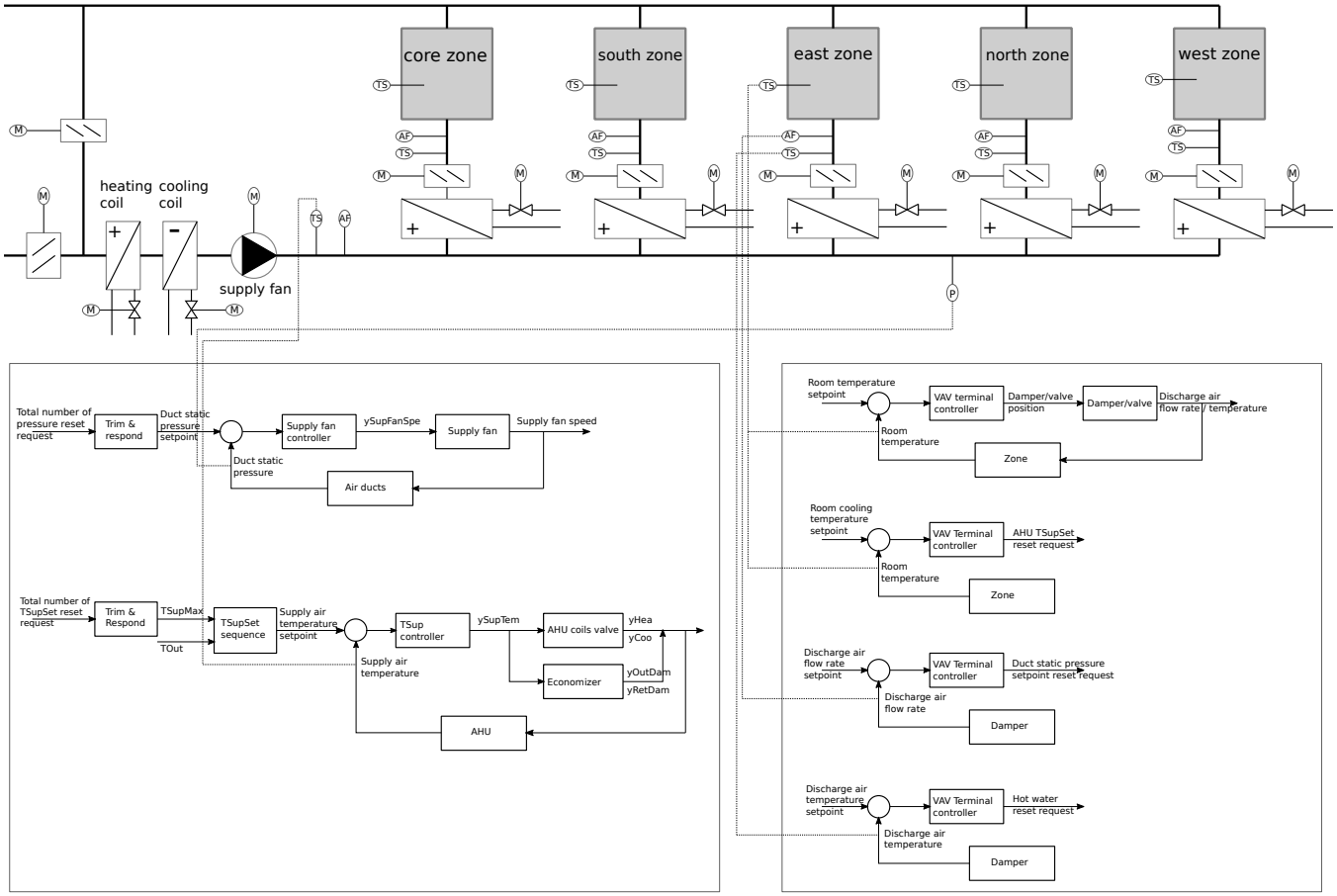


Fig. 14.3: *Control schematics of Guideline 36 case.*

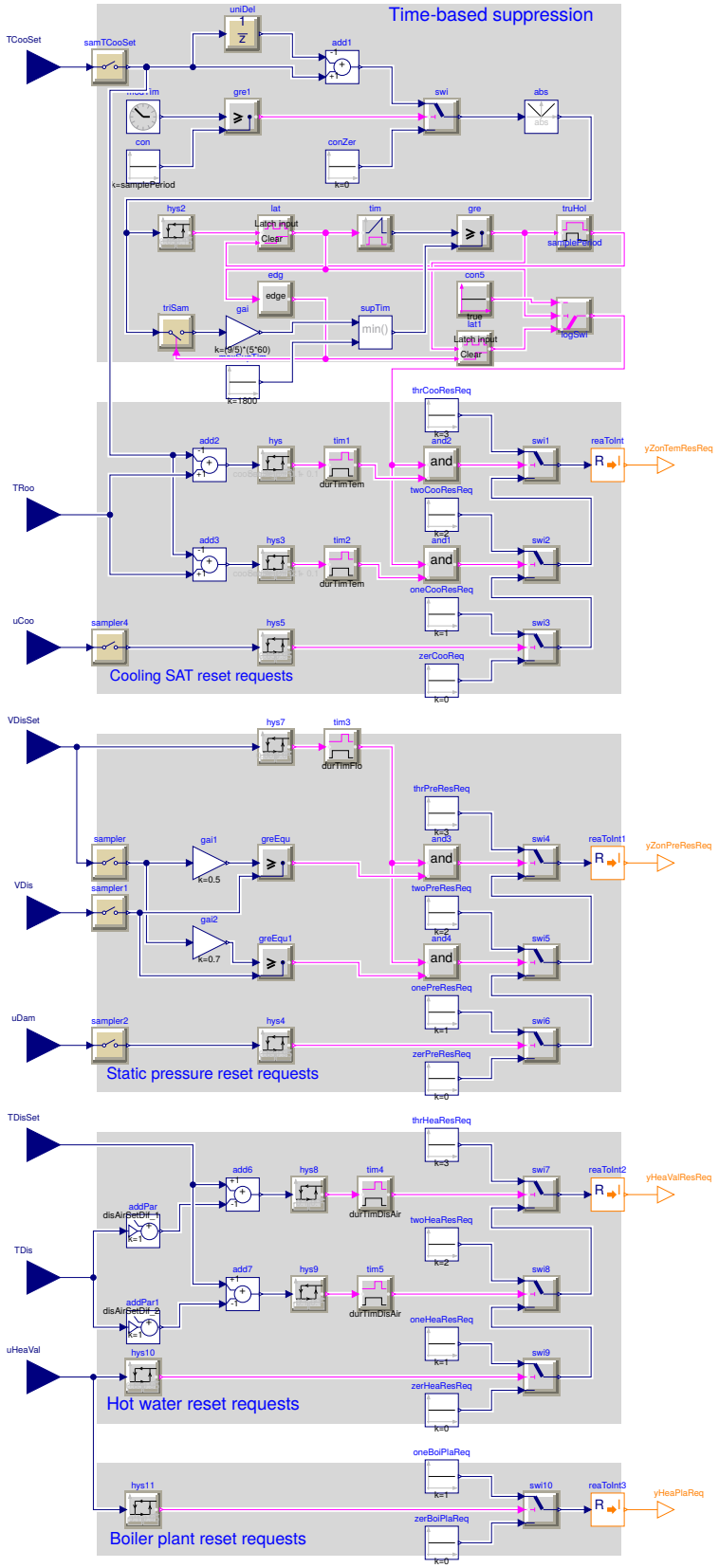


Fig. 14.4: Composite block that implements the sequence for the VAV terminal units that output the system requests. (Browsable version.)

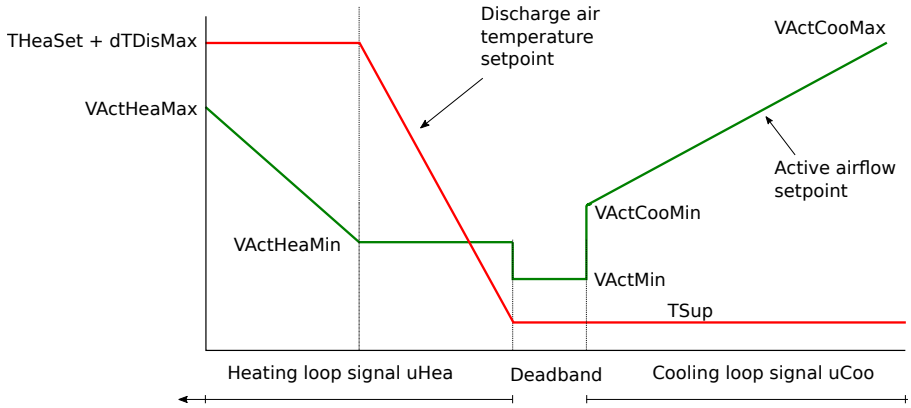


Fig. 14.5: Control sequence for VAV terminal unit.

14.2.7 Simulations

Fig. 14.6 shows the top-level of the system model of the base case, and Fig. 14.7 shows the same view for the Guideline 36 model.

The complexity of the control implementation is visible in Fig. 14.4 which computes the temperature and pressure requests of each terminal box that is sent to the air handler unit control.

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 14.1 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 7.6) of CDL.

Table 14.1: 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

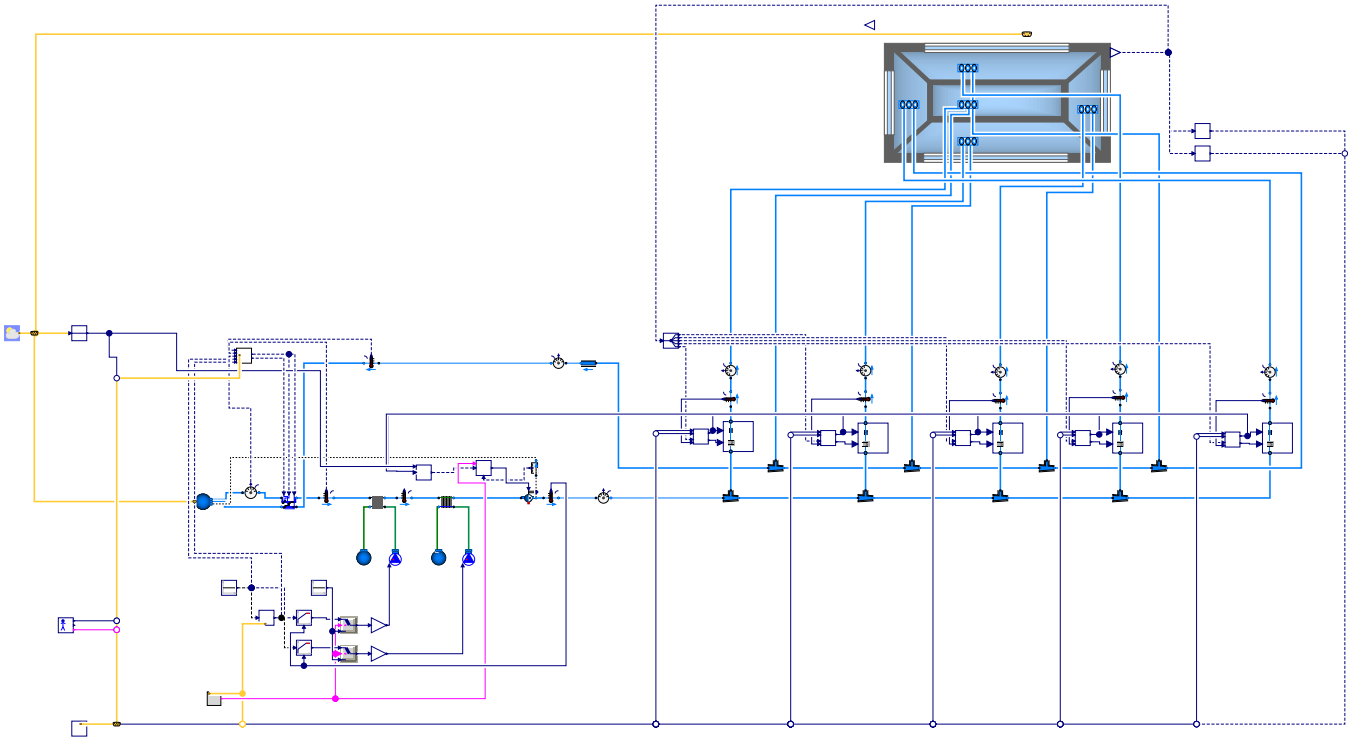


Fig. 14.6: Top level view of Modelica model for the base case.

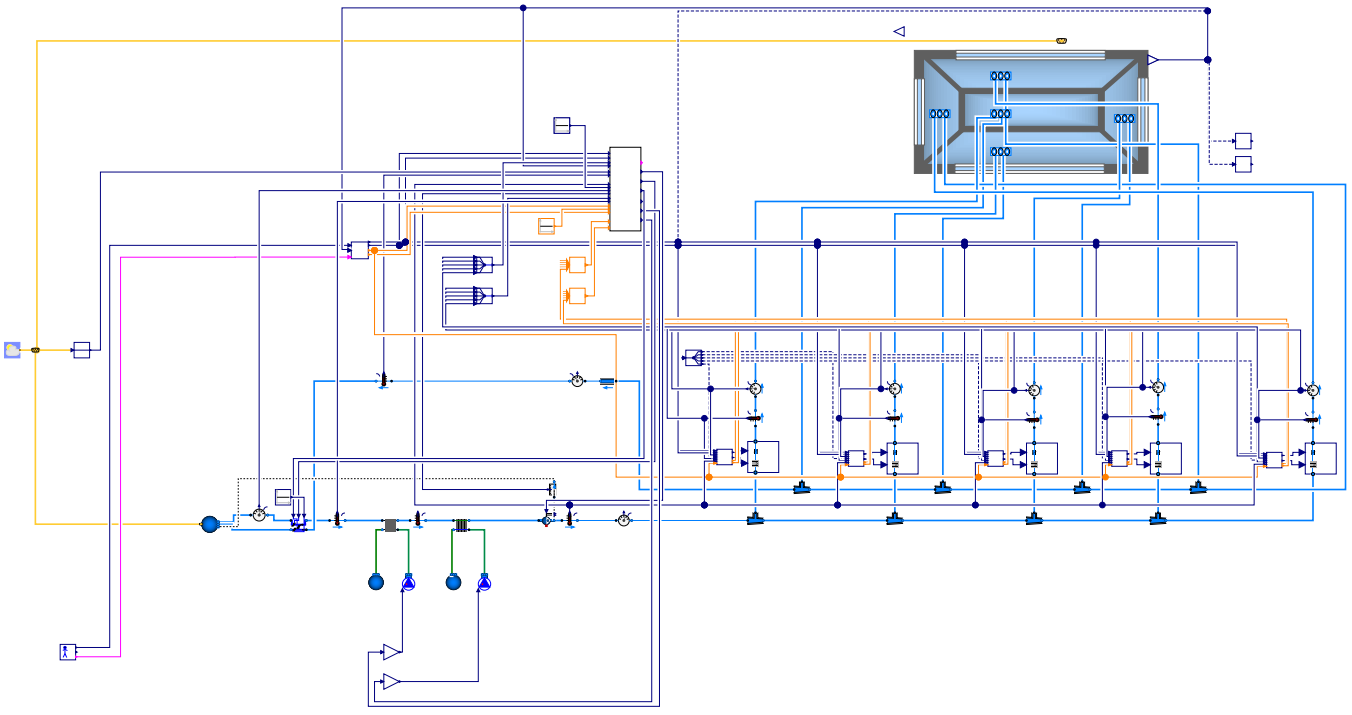


Fig. 14.7: Top level view of Modelica model for the Guideline 36 case.

14.3 Performance Comparison

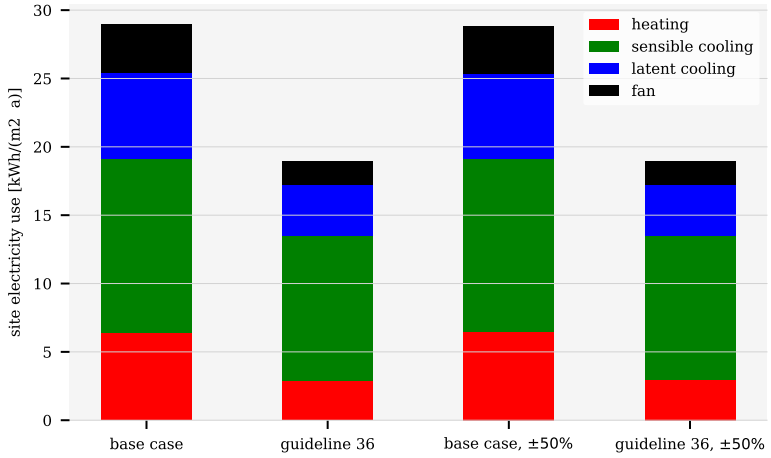


Fig. 14.8: *Comparison of energy use. For the cases labeled ±50%, the internal gains have been increased and decreased as described in Section 14.2.3.*

Table 14.2: *Heating, cooling, fan and total site HVAC 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. 14.8 and Table 14.2 compare the annual site HVAC 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.

In this case study, the Guideline 36 control saves around 30% site HVAC 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 the Guideline 36 sequence due to their complex logic that contains various mode changes, interlocks and timers. Various programming errors and misinterpretations or ambiguities of Guideline 36 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. 14.9 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 base case and the Guideline 36 case.

Fig. 14.10 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.

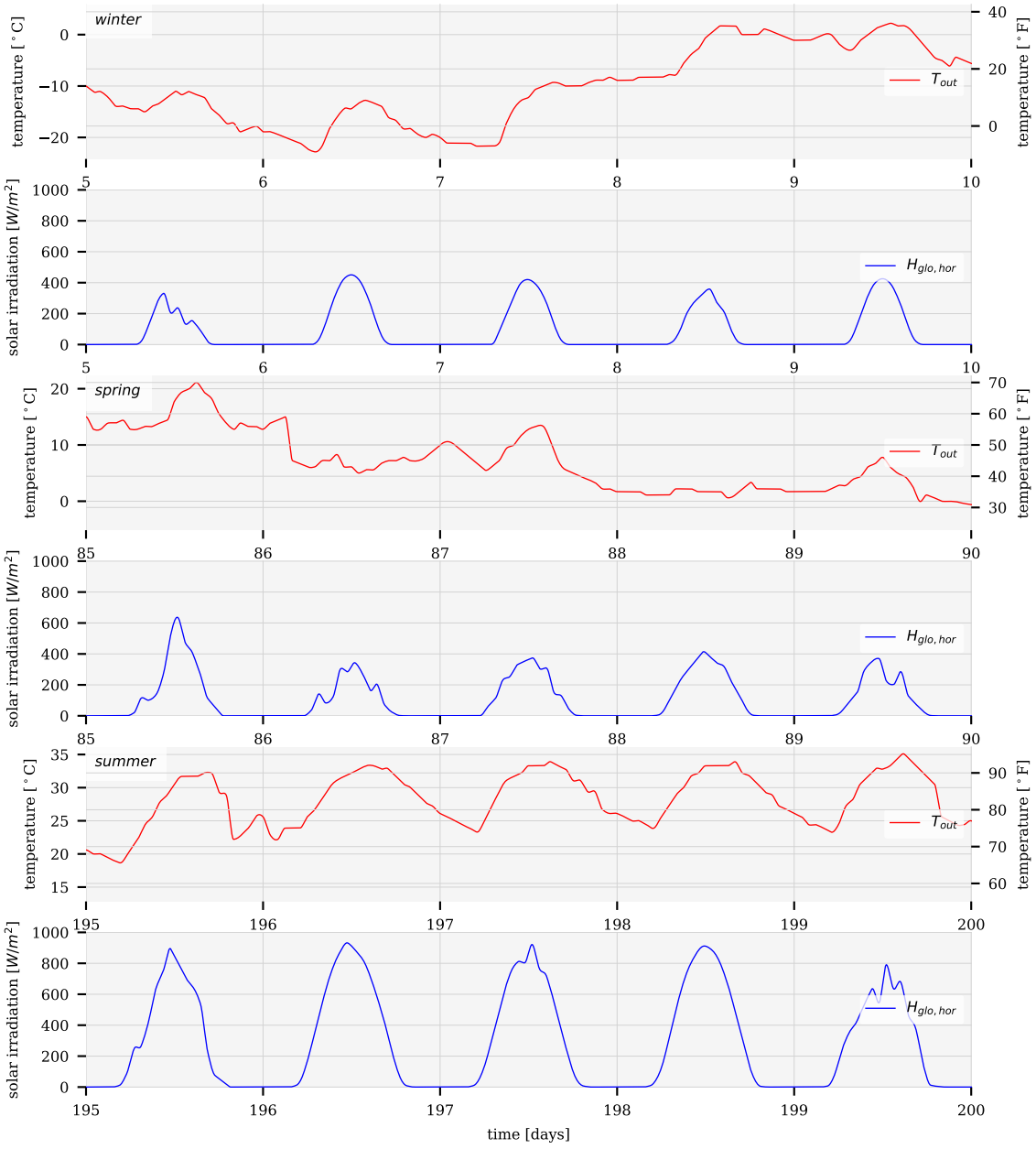


Fig. 14.9: Outside air temperature and global horizontal irradiation for the three periods that will be further used in the analysis.

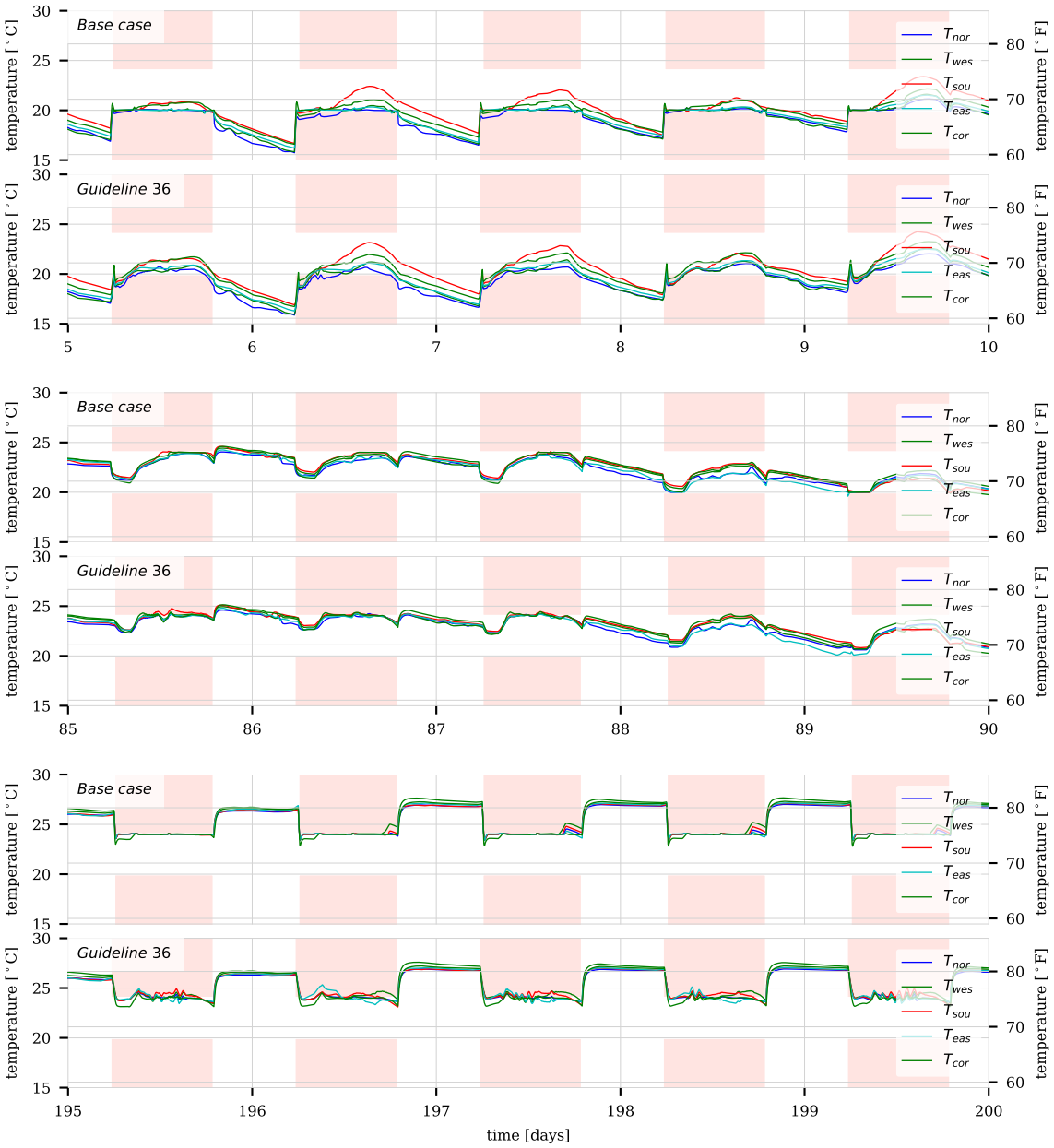


Fig. 14.10: Room air temperatures. The white area indicates the region between the heating and cooling setpoint temperatures.

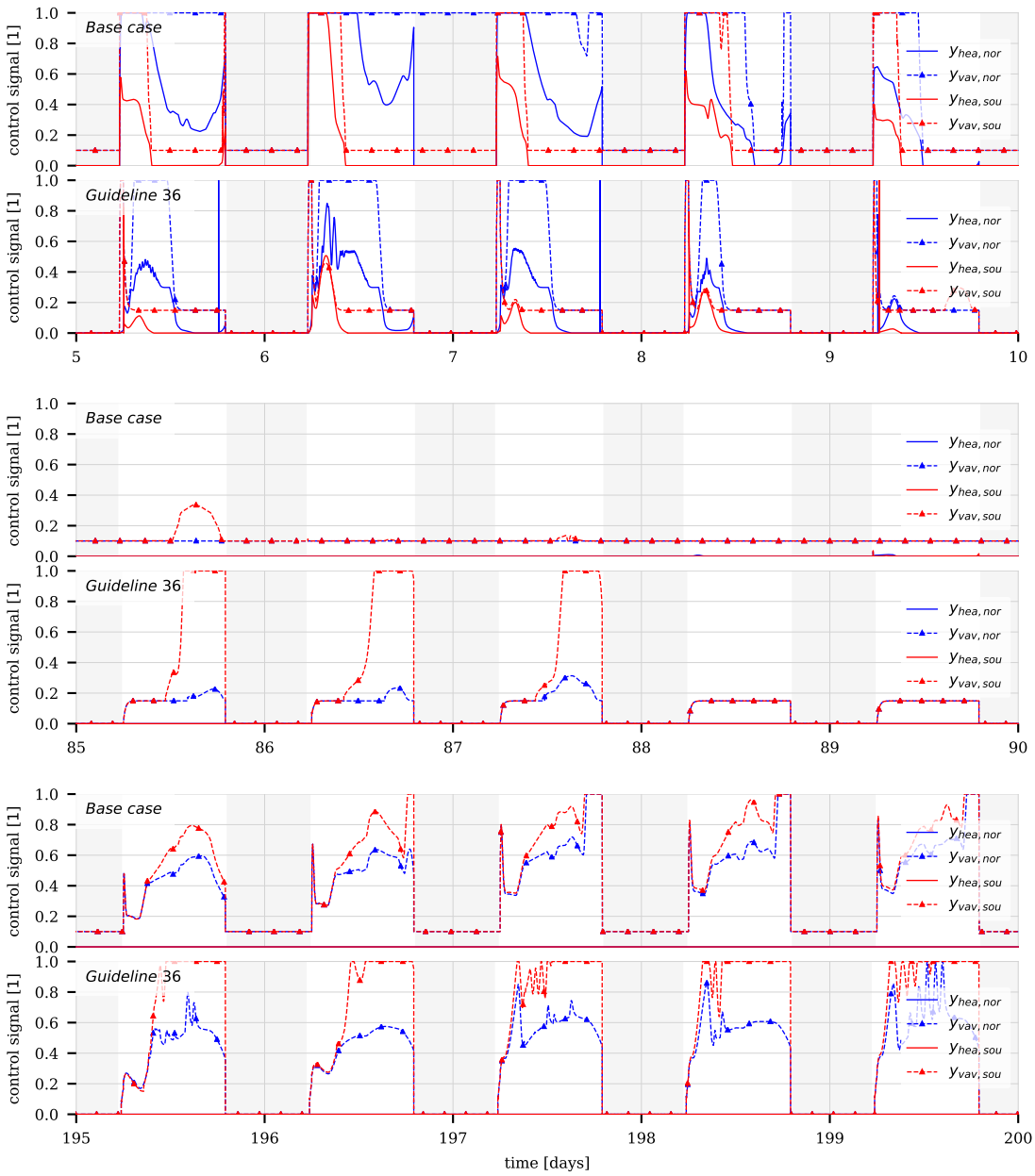


Fig. 14.11: VAV control signals for the north and south zones. The white areas indicate the day-time operation.

Fig. 14.11 shows the control signals of the reheat coils y_{hea} and the VAV damper y_{vav} for the north and south zones.

Fig. 14.12 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 the Guideline 36 sequence 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. 14.13 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. 14.14 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. 14.15 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. 14.8, their energy consumption is not noticeably affected.

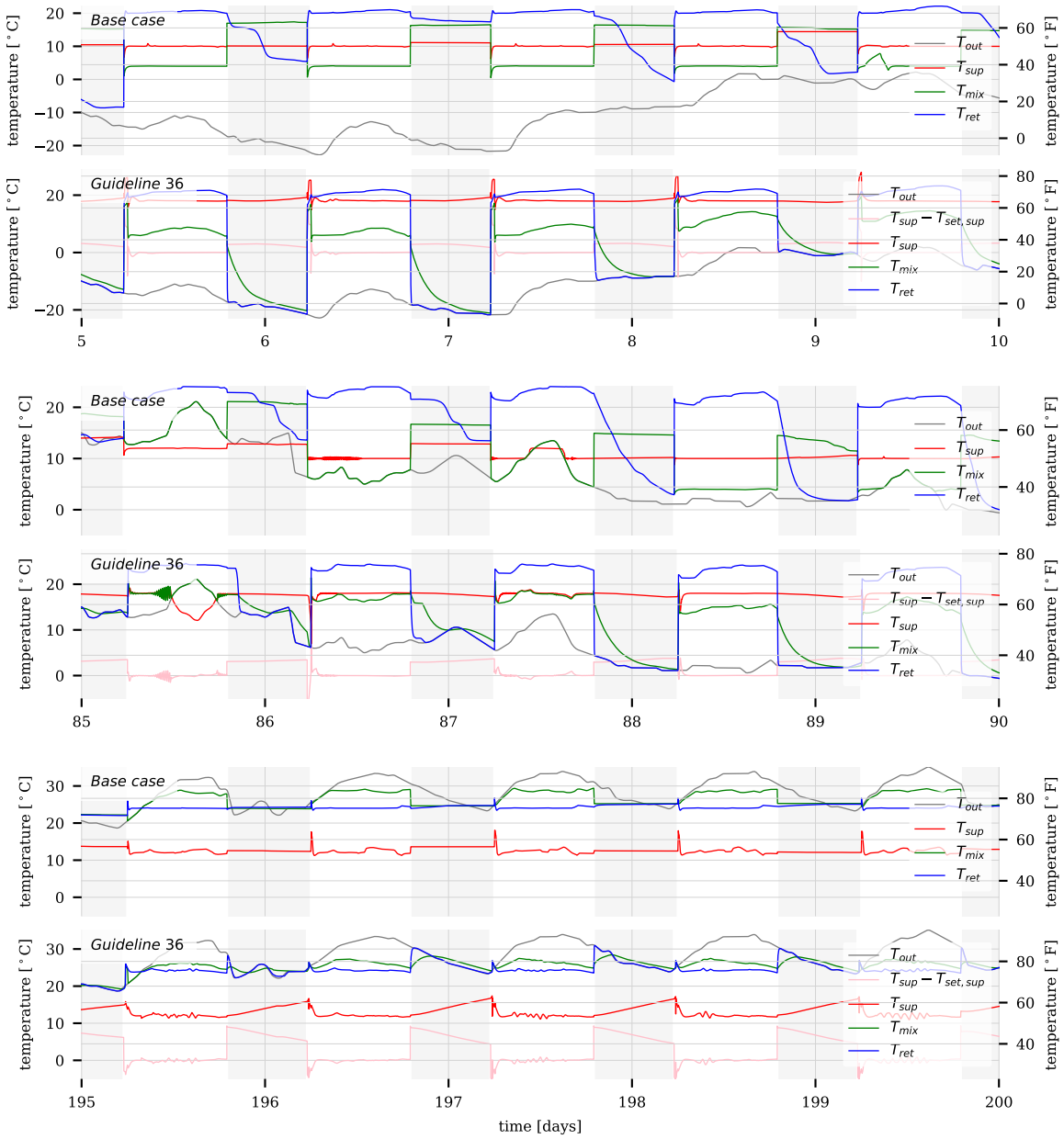


Fig. 14.12: AHU temperatures.

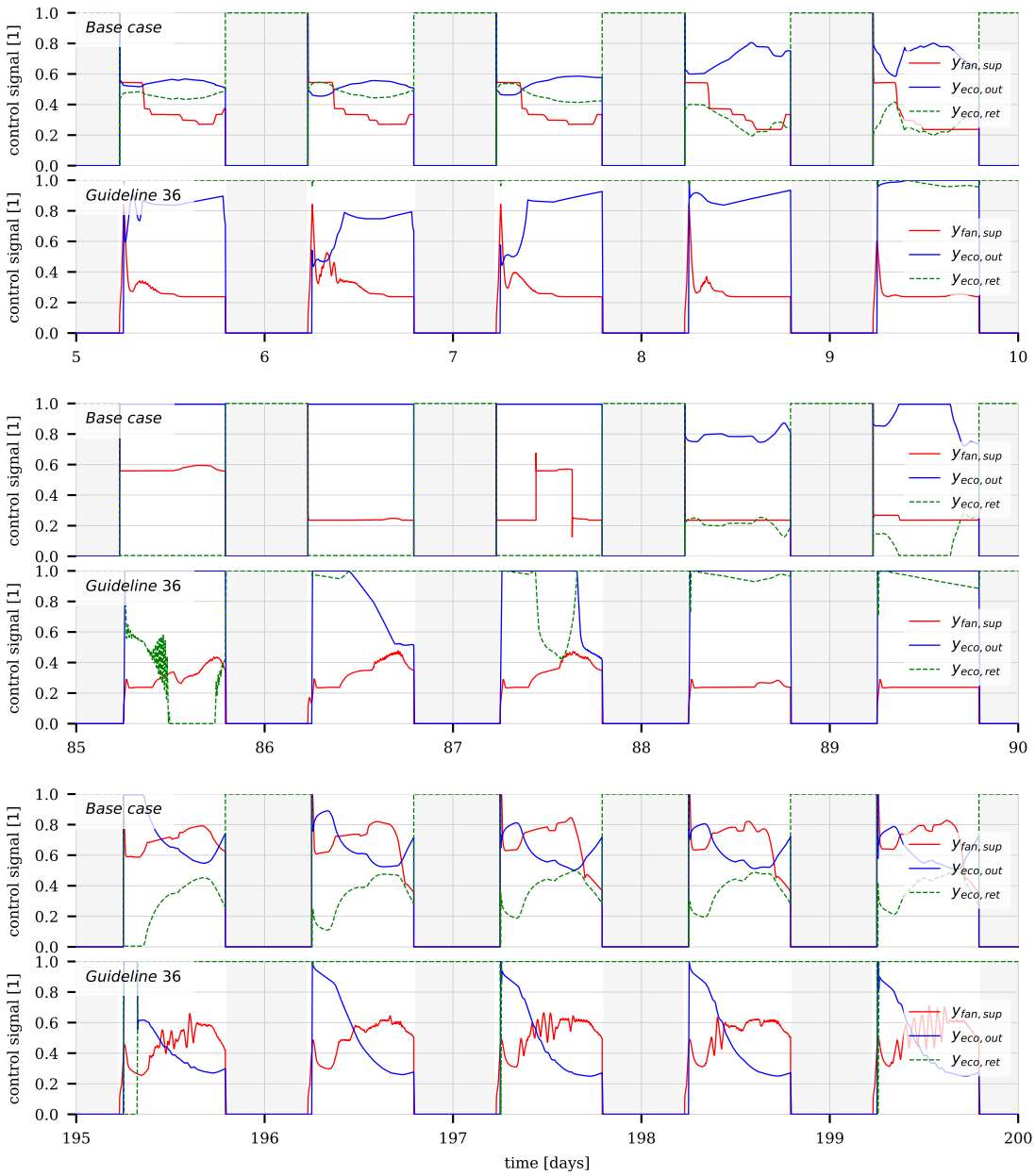


Fig. 14.13: Control signals for the supply fan, outside air damper and return air damper.

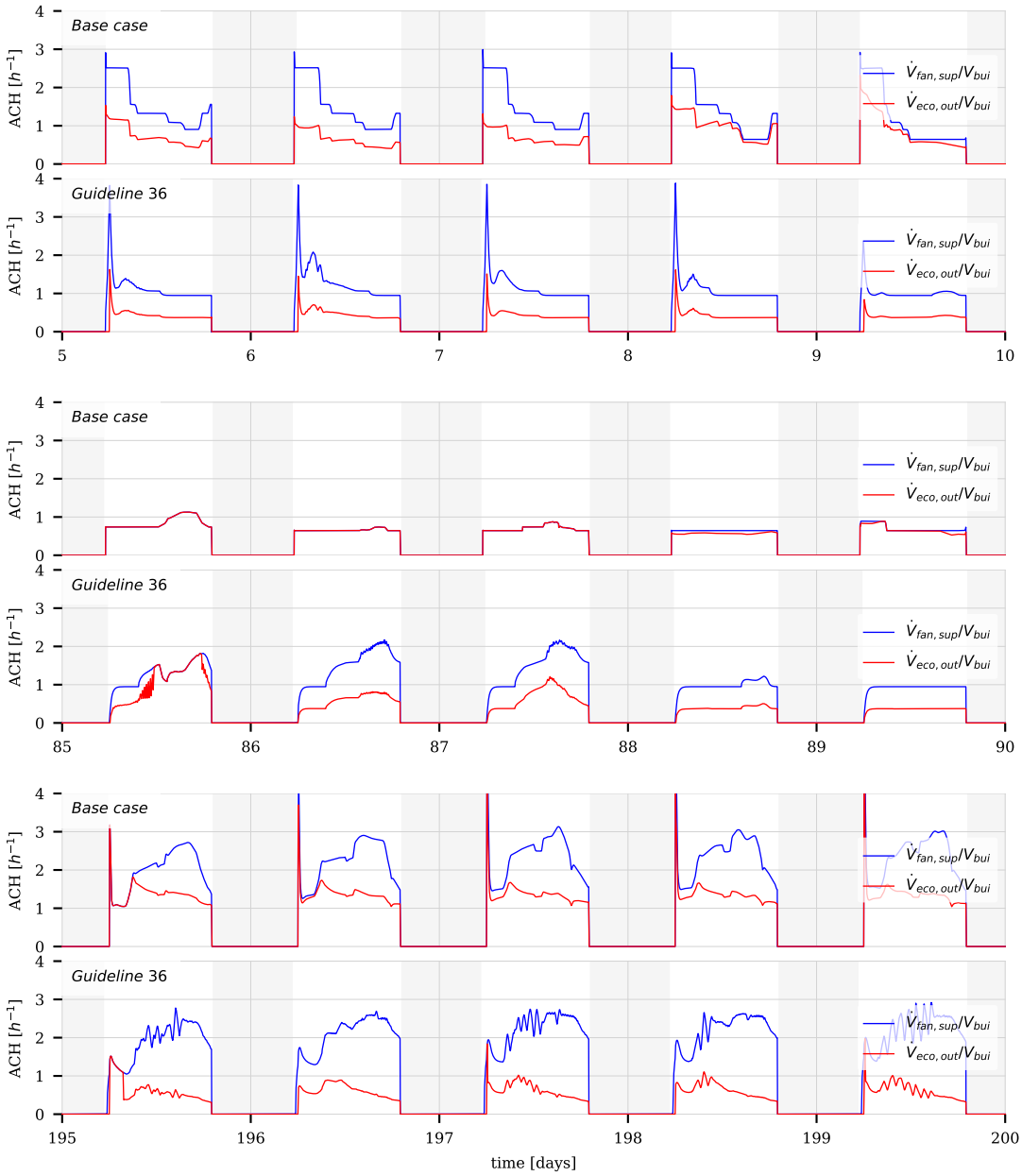


Fig. 14.14: Fan and outside air volume flow rates, normalized by the room air volume.

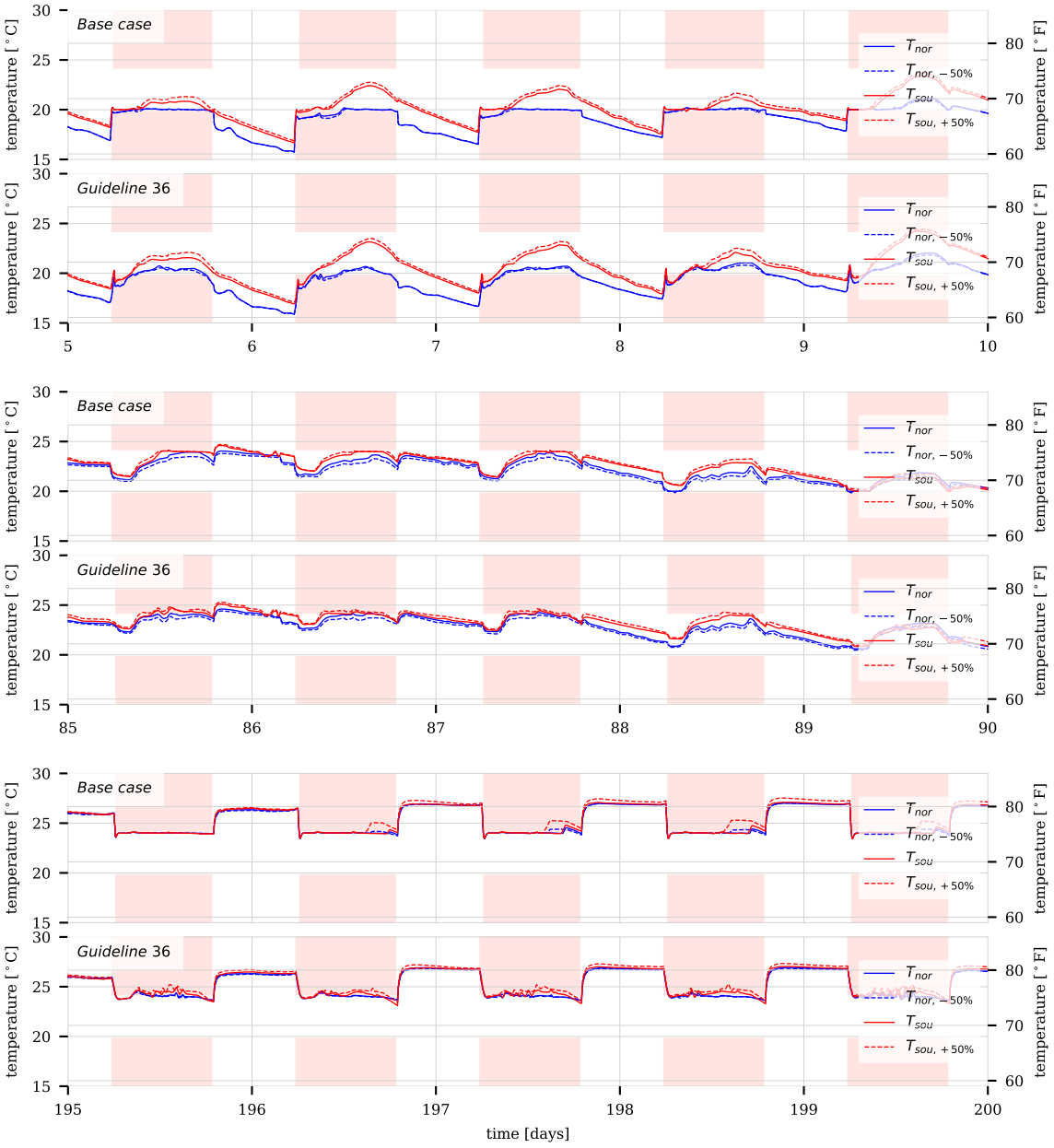


Fig. 14.15: Outdoor air and room air temperatures for the north and south zone with equal internal loads, and with diversity added to the internal loads. The white area indicates the region between the heating and cooling setpoint temperatures.

14.4 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].

14.4.1 Freeze Protection for Mixed Air Temperature

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

Guideline 36 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 the Guideline 36 sequence controls based on the supply air temperature and not the mixed air temperature. Hence, this control would not have been active.

Fig. 14.16 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 temperature, 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 temperature by 10 Kelvin (18 Fahrenheit) below the values obtained from the TMY3 weather data. This caused in day 6 and 7 in Fig. 14.16 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 the following wordings 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.

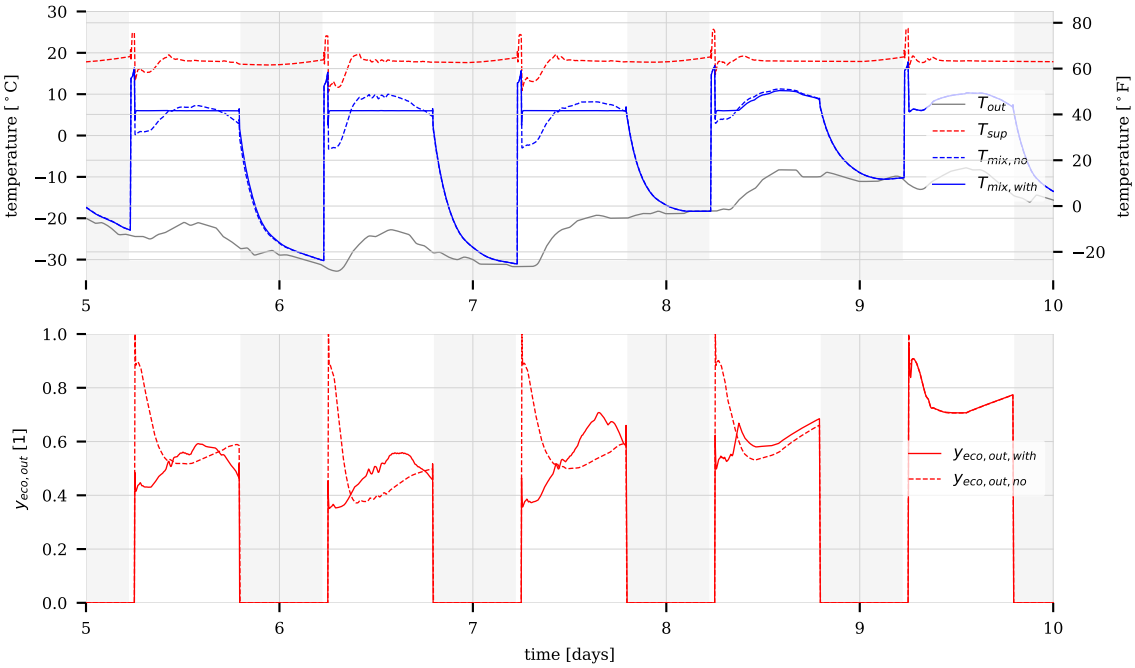


Fig. 14.16: Mixed air temperature and economizer control signal for Guideline 36 case with and without freeze protection.

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.

14.4.2 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. 14.5. 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.

14.4.3 Averaging Air Flow Measurements

The Guideline 36 sequence 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).

14.4.4 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 to 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.

14.4.5 Lessons Learned Regarding the Simulations

A few lessons regarding simulating such systems have been learned and are reported here. Note that related best practices are also available at https://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 the fan increased its mass flow rate faster than the VAV dampers opened. The large pressure drop also lead to large power consumption and hence unrealistic temperature increase across the fan.
- *Fan with prescribed pressure head:* We also tried to use a fan with prescribed pressure head. Similarly as above, the fan maintains the pressure 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 computationally expensive simulation of the long-wave radiation network in the rooms. Because surface temperatures change slowly, computing it every 2 minutes suffices. 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 diffusion among the control volumes of the water and among the control volumes of air used to be neglected as the dominant mode of heat transfer is 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.

14.5 Discussion and Conclusions

In this case study, the Guideline 36 sequence reduced annual site HVAC energy use by 30% compared to the baseline implementation with comparable thermal comfort. Such savings are significant, and have been achieved by changes in controls programming only which can relatively easy be deployed in buildings.

Implementing the Guideline 36 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, we made such an implementation and laid out the structure and conventions.

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 14.4.1) and chattering due to hard switches (Section 14.4.2). 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 our 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 15

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 *Control Description Language*.

CDL-JSON The JSON representation of the *Control Description Language*, which can be generated with the `modelica-json` translator that is available at https://github.com/lbl-srg/modelica-json.

Control Description Language The Control Description Language (CDL) is a declarative object-oriented language that can be used to express control sequences. CDL is a subset of the Modelica Language standard 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 *Control Description Language*.

Control eXchange Format The Control eXchange Format (CXF) is a JSON-LD representation of a CDL logic that is intended to be readily imported and exported to and from building automation systems. CXF specifications are in Section 8.

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 https://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 $x = 1$.

Structural parameter We say that a parameter is a *structural parameter* if changing its value can change the system of equations that is being evaluated in the control logic. For example, a parameter that changes a controller from a P to a PI controller is a structural parameter because an integrator is being added. A parameter that enables an input or that changes the size of an array is a structural parameter.

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 16

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 17

References

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

[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.

[Mod23] *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification, Version 3.6*. Modelica Association, March 2023. URL: https://specification.modelica.org/maint/3.6/MLS.pdf.

[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://doi.org/10.1201/9781003151906, doi:10.1201/9781003151906.

[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://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: https://simulationresearch.lbl.gov/wetter/download/2013-IBPSA-Wetter.pdf.

[WBG+20] Michael Wetter, Kyle Benne, Antoine Gautier, Thierry S. Nouidui, Agnes Ramle, Amir Roth, Hubertus Tummescheit, Stuart Mentzer, and Christian Winther. Lifting the garage door on spawn, an open-source bem-controls engine. In *Proc. of Building Performance Modeling Conference and SimBuild*, 518–525. Chicago, IL, USA, 2020.

[WGH18] Michael Wetter, Milica Grahovac, and Jianjun Hu. Control description language. In *1st American Modelica Conference*. Cambridge, MA, USA, August 2018. URL: https://doi.org/10.3384/ecp1815417, doi:10.3384/ecp1815417.

[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/.

[ZBG+20] Kun Zhang, David H. Blum, Milica Grahovac, Jianjun Hu, Jessica Granderson, and Michael Wetter. Development and verification of control sequences for single-zone variable air volume system based on ashrae guideline 36. In *2nd American Modelica Conference*, 81–90. Boulder, CO, USA, 2020. URL: https://doi.org/10.3384/ecp2016981, doi:10.3384/ecp2016981.