Title

Open Building Control

Authors

Wetter, Michael
Ehrlich, Paul
Gautier, Antoine
et al.

Publication Date

2021-03-12

Peer reviewed

CALIFORNIA
ENERGY COMMISSION

Energy Research and Development Division

FINAL PROJECT REPORT

Open Building Control

Gavin Newsom, Governor
February 2021 |CEC-500-2021-012

PREPARED BY:

Primary Author:
Michael Wetter[1]
Paul Ehrlich[2]
Antoine Gautier[1]
Milica Grahovac[1]
Philip Haves[1]
Jianjun Hu[1]
Kun Zhang[1]

[1]Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA 94720
(510)486-2000
https://www.lbl.gov

[2]Building Intelligence Group
1751 SW Prospect Drive
Portland, OR 97201
(651)-204-0105
https://www.buildingintelligencegroup.com

Contract Number: EPC-16-056

PREPARED FOR:
California Energy Commission

Karen Perrin
Project Manager

Virginia Lew
Office Manager
ENERGY EFFICIENCY RESEARCH OFFICE

Laurie ten Hope
Deputy Director
ENERGY RESEARCH AND DEVELOPMENT DIVISION

Drew Bohan
Executive Director

ACKNOWLEDGEMENTS

This research was supported by:

- The Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231
- The California Energy Commission's Electric Program Investment Charge (EPIC) Program.

The authors would like to thank the following people and organizations for their contributions to the project:

- Troy R. Maeder - Automated Logic
- Caleb Clough, David Pritchard, Amy Shen, Paul Switenki - ARUP Dave Robin - BSC Softworks
- Brent Eubanks - Carbon Lighthouse Brian Turner - ControlCo
- Rick Stehmeyer - Cx Associates
- Jay Santos, Jamie Nickels - Facility Dynamics Mark Hydeman - Google
- John Bruschi, Dave Guerrant, Andrea Traber, John Nelson, Fiona Woods - Integral Group Jonathan Schoenfeld - Kodaro
- Jim Kelsey - kW Engineering
- David H. Blum, Janie Page, Mary Ann Piette, Anand Prakash, Marco Pritoni, Lisa Rivalin - Lawrence Berkeley National Laboratory
- Francisco Ruiz, Rich Rockwood - Oracle
- Yan Chen, Karthikeya Devaprasad - Pacific Northwest National Laboratory Gerry Hamilton, Stanford Facilities Energy Management
- Hwakong Cheng, Brandon Gill, Reece Kiriu, Steven T. Taylor - Taylor Engineering

PREFACE

The California Energy Commission's (CEC) Energy Research and Development Division supports energy research and development programs to spur innovation in energy efficiency, renewable energy and advanced clean generation, energy-related environmental protection, energy transmission and distribution and transportation.

In 2012, the Electric Program Investment Charge (EPIC) was established by the California Public Utilities Commission to fund public investments in research to create and advance new energy solutions, foster regional innovation and bring ideas from the lab to the marketplace. The CEC and the state's three largest investor-owned utilities—Pacific Gas and Electric Company, San Diego Gas & Electric Company and Southern California Edison Company—were selected to administer the EPIC funds and advance novel technologies, tools, and strategies that provide benefits to their electric ratepayers.

The CEC is committed to ensuring public participation in its research and development programs that promote greater reliability, lower costs, and increase safety for the California electric ratepayer and include:

- Providing societal benefits.
- Reducing greenhouse gas emission in the electricity sector at the lowest possible cost.
- Supporting California's loading order to meet energy needs first with energy efficiency and demand response, next with renewable energy (distributed generation and utility scale), and finally with clean, conventional electricity supply.
- Supporting low-emission vehicles and transportation.
- Providing economic development.
- Using ratepayer funds efficiently.

Open Building Control is the final report for the OpenBuilding Control Project (Contract Number: EPC-16-056) conducted by Lawrence Berkeley National Laboratory. The information from this project contributes to the Energy Research and Development Division's EPIC Program.

For more information about the Energy Research and Development Division, please visit the CEC's research website (www.energy.ca.gov/research/) or contact the CEC at 916-327-1551.

ABSTRACT

Best practice control sequences are often not implemented correctly, or are not implemented at all, in large commercial buildings. This typically leads to 10-30 percent energy waste, along with reduced occupant productivity and unnecessary equipment wear. The current process of designing and implementing such control sequences is a manual process that starts with designers who often don't have adequate training, then requires controls programmers to interpret and program a verbose written sequence. This process has been shown to fail to deliver high performance control sequences at scale. The Open Building Control project digitizes the current control delivery process. The project is developing tools for system designers to select control sequences, assess their energy performance and load flexibility potential using whole building simulation, specify the sequence for implementation using machine-to-machine translation by a control provider and formally testing the as-installed sequences by a commissioning agent. The project developed tools for each stage of this delivery process. The key innovation of the project is the development of the Control Description Language, a language that allows such a digitized control delivery process with end-to-end verification.

Libraries of control sequences have been implemented using the Control Description Language, and their performance has been demonstrated using whole building energy simulation. An automated translation of such sequences to a commercial control product line has been conducted using a prototype translator. Tools for formal verification of as-installed control sequences relative to their specification have been developed and demonstrated. The American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE) started the process of forming a committee to make this language an ASHRAE/ANSI Standard. This new standard will complement existing and emerging ASHRAE standards for building communication and semantic modeling by providing a standard for expressing the control logic - the actual brain of the building. We expect this language and the process it enables to be an important contribution to the deployment of high performance building control sequences at scale because it allows taming the complexity of the control delivery process, which is continually increasing due to the need for higher performance and increased load flexibility to meet goals for net zero energy and increased renewable integration.

Keywords: OpenBuildingControl, commercial buildings, energy efficiency, automation, high-performance controls, description language, sequences of operation, BACnet, ASHRAE

Please use the following citation for this report:

Wetter, Michael. 2021. *Open Building Control*. California Energy Commission. Publication Number: CEC 500-2021-012.

Contents

Chapter 1

Executive Summary

1.1 Introduction

In the United States, commercial buildings account for just under 20% of all energy use. Energy consumption is driven by systems that include HVAC and lighting. Proper control of these systems, based on factors such as building occu- pancy and weather conditions, can reduce building energy use by 10 to 30%. However, few commercial buildings have optimized control systems. Many existing buildings predate current energy codes, standards and guidelines which require optimized sequences. New construction projects that are designed to implement such strategies frequently struggle due to the inherently complicated process of traditional design development, documentation, interpretation, implementation and owner operation. This results in buildings that are inefficient and often uncomfortable, which results in wasted energy and lost occupant productivity.

OpenBuildingControl is a project whose aim is to improve the process and tools necessary for the design, cost-effective implementation, and validation of the control sequences used in commercial buildings. The first phase of the project, reported here, has been co-funded by the California Energy Commission and the United States Department of Energy (DOE). A second phase of the project is being funded by the DOE.

The Phase 1 work reported here has focused on providing the capability to avoid major problems with the current process for the design and implementation of controls in commercial buildings. Current practice involves the HVAC designer writing a sequence, which depending on the skill level of the HVAC designer can be ambiguous and error-prone. The sequence is a verbose description of the control system operation, which a project technician has to interpret to write the necessary code for deployment of the sequence in a proprietary control system. This is followed by a manual process to validate and confirm the operation.

The OpenBuildingControl project built the foundation to enable the digitization of the current paper-based delivery process. The project has built tools for system designers to select, model the performance of, and then specify sequences for implementation, using a digitized workflow with end-to-end verification, including formal testing of the installed control sequences. The designer will be able to express the desired sequence in an electronic format that can be readily implemented or translated to programming code without the need for manual interpretation. The project will also provide tools to automatically document the sequences of operation implemented in a building and compare them to the original design intent. Used together, this set of tools will have the potential to substantially reduce energy use in both new commercial buildings and in existing buildings with controls retrofits. However, to be effective, these tools need to be widely adopted and used by industry, including system engineers, designers, controls manufacturers, controls subcontractors, owners,

and to be required or incentivized by other interested parties, including state energy agencies and utilities.

The OpenBuildingControl project complements work by the Standing Guideline Project Committee 36 of the American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE), which collects, develops and publishes control sequences considered to be industry best-in-class for improving system stability, energy performance, indoor air quality and comfort. Current versions of energy standards and codes, such as ASHRAE 90.1 and the California Energy Code, Title 24, require specific algorithms documented in Guideline 36, and are anticipated to adopt or reference Guideline 36 sequences as awareness of the Guideline grows.

1.2 Project Purpose

The purpose of the OpenBuildingControl project is to substantially reduce commercial building energy use by optimizing the design, implementation, and validation of building controls. The project team has documented existing buildings controls practices and developed processes and tools to remove impediments to effective design and correct implementation.

A key paradigm shift is the development of a process and supporting software that paves the way for digitization of the controls delivery process. The current process starts with the need for a design engineer to write a “controls sequence” using a verbose format to describe each part of the operation of a system. There are several challenges with this process. The first is that many design engineers are not well trained in controls and have difficulty writing sequences that are appropriate and will result in efficient operation. The second challenge is that a controls technician has to interpret what was written and then express it in a proprietary controls programming language. The project team has developed a formal end-to-end process that starts with a library of optimized sequences expressed both in English and in a unambiguous digital format. The system designer can select the sequences that will work best with the project’s mechanical system, using tools developed in this project. The digital sequence specification allows the performance of building control sequences, including annual energy, load flexibility, peak demand and comfort, to be assessed using whole building simulation. The control sequences can then be used directly or be translated for use in commercial building control product lines using machine-to-machine translation. Finally, new tools will assist in verifying proper implementation of the sequences. Such a process will allow error-free deployment of control sequences, thereby addressing the situation that the current paper-based process fails to implement high-performance control sequences at scale.

The main audience for the technology developed in this project consists of:

- Building owners and operators, who are responsible for operating commercial buildings so that they are safe, productive, and efficient.
- Researchers and control companies who develop new HVAC systems and control sequences for building energy systems.
- Professional organizations such as ASHRAE who are developing guidelines for high performance building control sequences.
- Analysts who assess the performance of control sequences when updating energy codes such as California’s Building Energy Efficiency Standard Title 24 or ANSI/ASHRAE/IES Standard 90.1, Energy Standard for Buildings Except Low-Rise Residential Buildings.
- Mechanical designers who specify control sequences for a particular building.
- Control companies and system integrators who implement control sequences in new construction or retrofit projects.
- Commissioning agents who verify whether the as-installed control sequences comply with the specification from the mechanical designer.

1.3 Project Approach

The US Department of Energy's Lawrence Berkeley National Laboratory is leading this project, with regular reviews from the US Department of Energy and the California Energy Commission program management. The process started with the establishment of a project team consisting of Lawrence Berkeley National Laboratory staff and industry experts in the design and implementation of control systems, along with an advisory panel that includes design engineers, general contractors, mechanical subcontractors, controls subcontractors, controls manufacturers, commissioning agents, and building owners and operators.

The advisory panel provided industry input and feedback while the core team was responsible for defining the new process and coding, testing, validating, and documenting the associated tools. Several presentations at ASHRAE also led to feedback that affected the direction of the research and development, as did presentation to the advisory panel, to selected companies and to the scientific community at various conferences.

A key technical challenge encountered by the project was that, due to a lack of standards, 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 programming syntax, which ranges from graphical languages to textual languages. Code generation for a variety of products is common in the Electronic Design Automation industry, which develops software tools for designing electronic systems such as integrated circuits and printed circuit boards. However, in the Electronic Design Automation industry, engineers write models of the physical device and the controls, using graphical and textual languages, and actual controllers are then built to conform to the models. If this process were to be applied to the buildings industry, then control providers would need to update their product lines. The project team believes that once CDL becomes a standard, that suppliers consider adding it to new or existing products. That process may take 5 to 10 years to complete. Therefore, for the immediate future, the OpenBuildingControl process will need to involve the building of models of control sequences that can conform to their implementation on existing control product lines, while ensuring that, as new product lines are being developed, they can invert the paradigm and build controllers that conform to the models. The project team has, therefore, selected the path of designing the Control Description Language in such a way that it provides a minimum set of capabilities that can be expected to be supported by current control products. As we have demonstrated with one commercial product, the barrier to supporting this language is low, and we therefore expect that suppliers may elect to develop and support translators. We are also working with industry to establish the Control Description Language as an ASHRAE/ANSI Standard and, eventually, an ISO Standard. Getting industry support to make the Control Description Language a standard would allow for products to be developed that use the Control Description Language without the need for translation.

1.4 Project Results

The project achievements to date have been very positively received by industry and by members of the ASHRAE Standing Guideline Project Committee 36 which develops high performance control sequences. The following items resulted from this project phase:

- Definition of use cases and processes related to controls design and implementation.
- Definition and documentation of the semantics and syntax of the Control Description Language and of its JSON export format.
- A library of control sequences for building energy systems expressed in the Control Description Language.
- Modeling tools that can simulate sequences expressed in the Control Description Language coupled to heating, ventilation, and air-conditioning models from the Modelica Buildings library and linked to Spawn of EnergyPlus

envelope models.
- Tools that verify that the control response from a Control Description Language–specified sequence and trended control outputs are within user-specified tolerances.
- Tools to translate the Control Description Language into open formats such as JSON and HTML, as well as to Microsoft Word.
- Demonstration of sequences expressed in the Control Description Language being translated to a proprietary language and uploaded into a working control system.
- Case studies that demonstrate the use of the tools and the energy savings obtained through the use of high performance control sequence.
- A commercialization and market transformation plan.
- The specification to develop a system design tool that will allow an engineer to specify the type of system to control and to select control options. The tool will then select and generate the proper control sequence using the Control Description Language. This tool will include a library of capabilities from sources such as ASHRAE Guideline 36 and the engineers' current library and will make use of the Spawn of EnergyPlus simulation tool to compute the performance of the selected option using whole building energy simulation.
- The formation of an ASHRAE Standard Project Committee for making the Control Description Language an ASHRAE/ANSI Standard and, ultimately, an ISO Standard.

There is also a set of items that were not completed by the end of Phase 1 of this project; partial follow-on funding to further develop these items has been secured. These items include:

- The implementation of the systems design tool.
- An expanded library of control sequences, expressed in the Control Description Language, that can be used as input for the above system design tool.
- Tools and documentation that can be used by control systems suppliers to develop translators from the JSON representation of the Control Description Language to their proprietary control system.
- Provisions to add tagging to the Control Description Language so that it can be used with Brick, Project Haystack and other similar semantic tagging and data modeling standardization efforts.
- Programs for implementing market transformation.
- Tools for evaluating a current control system and developing documentation for installed sequences.

1.5 Technology/Knowledge Transfer/Market Adoption

To build market adoption, the project team worked with key committees of ASHRAE to align the developed technology with the needs of the industry. Furthermore, all technology has been developed in such a way that it directly integrates with the roadmap of the US Department of Energy's Building Technologies Office for energy simulation and for sensors and controls.

To align the developed technologies with industry needs, the project team developed a detailed commercialization and market transformation plan. This plan outlines the current state of the process of control specification, delivery, commissioning and building operation. It discusses the tools and workflow developed by the project team. It lists benefits for mechanical designers, control providers, building operators and building owners. Lastly, it describes a path to establish a digitized control delivery process.

Foundational work for this deployment started during this project: A key part of the technology transfer is the work that has started on making the Control Description Language an ASHRAE/ANSI standard, thereby ensuring the industry that there is a robust foundation on which industry can make further investments. The tools developed in this project havebecome

a key part of the tool development sponsored by the US Department of Energy. Specifically, Spawn of EnergyPlus is, in part, being developed to support the design, deployment and operation of advanced energy and controls for buildings, for district heating and cooling systems and for geothermal systems through its Building Technologies Office, Advanced Manufacturing Office and Geothermal Office, respectively.

To support the update of energy codes, such as California's Building Energy Efficiency Standard Title 24 and ANSI/ASHRAE/IES Standard 90.1, we anticipate that analysts will use the Control Description Language together with Spawn of EnergyPlus. This will allow analyzing the energy impacts of measures related to building control across a portfolio of buildings in different climate zones. Moreover, prescriptive codes may state which control sequences need to be used and they could then provide the specification of these control sequences in the Control Description Language for use in project specifications and for implementation on the building's control system.

1.6 Benefits to California

This project will benefit both the State of California and the rest of the US — and, ideally, the world. The key benefits are as follows:

- *Reduced cost to design and implement advanced controls*. This project will make the use of these advanced controls sequences more cost effective for new construction and, even more importantly, for retrofit, where costs and complexity are often impediments to implementation.
- *Improved energy efficiency*. The project team has documented the potential to reduce heating, ventilation, and air-conditioning system energy use by 30% through the use of advanced controls sequences for airside HVAC systems. The team is confident that this approach can be extended to other building systems, including primary systems, lighting systems, and active façade systems. The ability to reduce building energy use is a significant benefit for the state and is essential to achieving California's 2030 goal of having all new commercial buildings, and 50% of commercial buildings being retrofitted, to be net zero energy.

The adoption of OpenBuildingControl will result in improved design and implementation of commercial building controls without requiring major retraining or process changes to how controls are designed or delivered. The energy savings from widespread adoption of the processes and tools can be estimated as follows. A major barrier to achieving the state's statutory energy goals is the failure of most commercial buildings to perform close to the technical potential of the design and its associated equipment. An LBNL meta-study identified 16% median actual savings from retro-commissioning and a study of 481 operational issues identified in existing commercial buildings found that control problems accounted for more than 75% of the potential energy savings. Therefore, we assume that around 75% of the 16% energy savings, equal to a total of 12% of energy savings, associated with commissioning are related to controls. Assuming that the technologies to be developed in the project can save 12% in the 50% of commercial building floor area higher than 50,000 sf, we will assume our technology can reduce energy consumption on average across all commercial buildings by 6%.

The California savings are estimated as follows: The annual energy consumption of California commercial buildings is about 67.1 TWh of electricity, equivalent to 0.64 quads (188 TWh) of source energy, and 1278.6 Mtherms (0.13 quads, 37.4 TWh) of natural gas. The estimated 6% savings correspond to 4.03 TWh of electricity, equivalent to 0.038 quads (11.28 TWh) of source energy, and 0.00764 quads (2.24 TWh) of natural gas. Assuming a price of 0.17 $/kWh for electricity and 8 $/(1000 ft3) for natural gas (corresponding to 0.027 $/kWh), the cost savings would be $0.69B in electricity and $0.064B in natural gas.

The US national savings are estimated as follows: The annual energy consumption of US commercial buildings is about 1240 TWh of electricity, equivalent to 11.9 quads (3472 TWh) of source energy, and about 22,500 MTherms (2.25 quads, 659 TWh) of natural gas. The estimated 6% savings correspond to 74.4 TWh of electricity, equivalent to 0.71 quads (208

TWh) of source energy, and 0.135 quads (39.5 TWh) of natural gas. Assuming a price of 0.11 $/kWh for electricity and 8 $/(1000 ft3) for natural gas, the electricity cost savings would be $8.2B and the natural gas savings $1.07B.

These electricity savings correspond to 25 Rosenfelds in the US and 1.5 Rosenfelds in California.

If we assume 75% adoption of OpenBuildingControl over the next ten years, a controls retrofit rate of 10% per year and a new building construction rate of 1.5% per year, then, after 10 years, the fraction of the building stock, weighted by floor area, that is impacted by OpenBuildingControl is 21%. Assuming the potential benefit of $0.69B savings in electricity, the estimated benefits are $146M/yr savings for California ratepayers.

Chapter 2

Introduction

2.1 Background

More than 1 quad/yr of energy is wasted in the US because, for most commercial building projects, control sequences are poorly specified and implemented. The process to specify, implement and verify controls sequences is often only partially successful, with efficiency being the most difficult part to accomplish. This is particulary the case for built-up HVAC systems, which require custom-control solutions and which are common in large buildings. For such systems, the current state is that, at best, the mechanical designer specifies the building control sequences in an English language specification. However, such a specification cannot be tested formally for correctness. It is also ambiguous, leaving room for different implementations, including variants that were not intended by the designer or may not work correctly. The implementation of the sequences is often done by a controls contractor who either attempts to implement the sequence as specified, or uses a sequence from a similar project that appears to have the same control intent. During commissioning, the lack of an executable specification of the control sequence against which the implementation can be tested makes commissioning of the control sequences expensive and limited in terms of code coverage [GF17]. Not surprisingly, programming errors are the dominating issue among control-related problems that impact energy use in buildings [BHK+02].

However, formal controls design and verification in other industries has led to significant labor cost savings and performance improvements.

2.2 Project Goals

The overall goal of this project is to significantly improve building energy efficiency through a robust workflow that allows deploying high performance building control sequence at scale. In support of this, the project developed a process, together with an integrated set of tools, to enable design engineers to unambiguously specify energy-efficient control sequences for commercial buildings and then verify their correct implementation, providing end-to-end quality control.

2.3 Approach

Our approach is to digitize the delivery of control. Rather than paper-based English language specification, our process is fully digitized, allowing performance assessment in design, electronic specification in a format that was designed to allow machine-to-machine translation to control product lines, and formal verification of the control response relative to its electronic specification. Specifically, the project developed a process and a set of tools that enable digital control specification, performance assessment using whole building energy simulation, and delivery and implementation on existing building automation product lines.

The technical environment for such a digital control delivery starts to fall in place: For communication of control signals, standards such as BACnet and LonWorks are widely used. For semantic modeling, Haystack and Brick Schema are increasingly used, and the proposed ASHRAE Standard 223P attempts to standardize semantic modeling, building on these previous efforts.

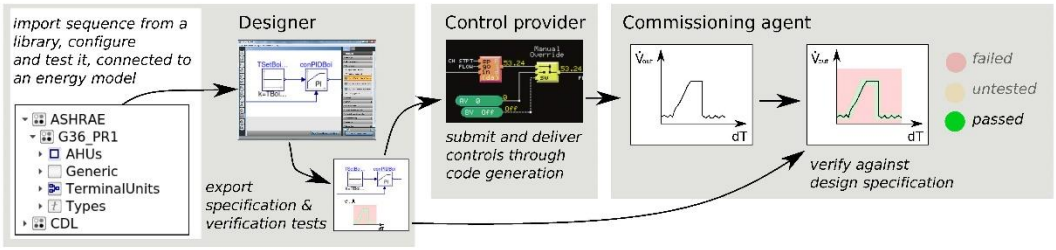


Fig. 2.1: *Overview of process for control sequence design, export of a specification, implementation on a control platform and verification against the specification.*

However, what is missing is a means to express control sequences in a way that can be simulated during design, exported for control specification and documentation, used directly or translated to commercial product lines and reused for formal verification of the correct implementation of the control sequences. This gap is what the OpenBuildingControl project attempts to close. A key element of the OpenBuildingControl is the Control Description Language (CDL) that has been developed in the project. CDL is a declarative language for expressing control sequences through block diagram modeling. To enable simulation of closed loop control as part of annual energy modeling during building design or control research, we designed CDL to be a proper subset of the Modelica language, an open standard for an eqution-based object-oriented modeling language [ME97][Mod12]. As CDL is a declarative language, the control specification can be exported in a vendor-independent json format that serves as an intermediate format to produce English language documentation including point lists, and that can serve as input to a code translator to a particular control product line. The control specification can also be exported for use in a formal process that verifies that a control signal generated by the actual implementation is within a user-selected tolerance of the simulated control signal. This provides, therefore, a workflow with an end-to-end verification as shown in Fig. 2.1. Therefore, CDL complements communication (ASHRAE 135 - BACnet) and semantic modeling (ASHRAE 223P - Designation and Classification of Semantic Tags for Building Data) by expressing the control logic, with the goal of standardizing this missing part of the control representation.

We believe that the timing of such an effort is ideal due to the convergence of various technologies related to the digitization of the building design and operation, and related to emerging needs of building energy systems. Regarding digitization of the building design and operation, declarative modeling (Modelica) progressed substantially over the last years, getting to the point where fully coupled closed loop control simulation is possible within annual energy simulation. Furthermore, advances in code generation eases machine-to-machine translation of declarative languages and semantic modeling (BRICK or ASHRAE 223p), putting in place the foundation to generate a semantic model from a declarative Modelica model. This combination promises to allow the semi-automatic connection of an actual building system to a digital twin

of the control and related algorithms that support building analytics (Mortar [FPA+19]). Regarding emerging needs of building energy systems, there is a shift towards all electric buildings in various US states and various countries. In these systems, heating and cooling often includes the use of heat sources and heat storage that are close to ambient temperatures. To increase the 2nd law efficiency of such systems, systems operate with low temperature lifts, rather than the large temperature lifts that are customary in fossil-fuel based heating systems and conventional cooling systems. Moreover, building systems also have the added recent requirement to provide flexibility to the electrical grid. All of these lead to more complex energy and control systems. The OpenBuildingControl process has been developed to support the transitions towards such high performance systems.

2.4 Project Results

The project resulted in a process and a set of software tools, documented at obc.lbl.gov, that pave the way to a digitized control delivery process. They enable the performance evaluation and improvement of building control sequences using whole building energy simulation. Typical performance indices are annual energy use, greenhouse gas emissions, peak demand and thermal comfort. Such performance assessment can be done by researchers and control companies as part of developing and evaluating new control sequences, or by mechanical designers as part of the building design process. These control sequences can then be exported to a digital format, for which we showed, as a proof-of-concept, that it can be translated to a commercial building control platform, thereby running the control sequence that was used in simulation natively on a commercial building control platform. This intermediate format also provides control providers data needed to build digital cost estimation tools, further streamlining the control procurement process.

As part of the project, we demonstrated each step of such a digitized control design, delivery and verification process. We also started forming an ASHRAE Standard Project Committee whose purpose is to turn the Control Description Language that has been developed in this project into an ASHRAE/ANSI standard. Such a standard will then complement existing standards for building control *communication* (ASHRAE 135 - BACnet), emerging standards for *semantic* data (ASHRAE 223P - Designation and Classification of Semantic Tags for Building Data) with a standard that allows expressing the control *logic* in a way that is independent of a particular control product line.

The potential energy savings of this project, if adopted widely, are estimated to be in California, 4.03 TWh of electricity, equivalent to 0.038 quads (11.28 TWh) of source energy, and 0.00764 quads (2.24 TWh) of natural gas. In the US, the potenial savings are 74.4 TWh of electricity, equivalent to 0.71 quads (208 TWh) of source energy, and 0.135 quads (39.5 TWh) of natural gas.

If we assume 75% adoption of OpenBuildingControl over the next ten years, a controls retrofit rate of 10% per year and a new building construction rate of 1.5% per year, then, after 10 years, the fraction of the building stock, weighted by floor area, that is impacted by OpenBuildingControl is 21%. This would result in estimated benefits of $146M/yr for California electricity ratepayers.

The next sections provide more details about the results of this project. They are structured as follows:

Section 3 describes the overall process from control design to performance assessment, export of control specification, cost-estimation, implementation by a control vendor and formal verification of the implemented control sequences relative to the design specifications.

Section 4 describes the Control Description Language (CDL), which is the key technology developed in this project. This language is used to express control sequences digitally and in English language, in a format that is then translated for simulation, for cost estimation, and for implementation in control product lines. This section is rather technical, and is mainly of interest to developers who implement tools that use CDL. Less technical readers may skip this section.

Section 5 describes how the CDL language has been used to implement libraries of ready-to-use control sequence that can be used within the process described in Section 3.

Section 6 describes various paths of how CDL can be translated for use in building control systems, respecting the need for reusing existing control product lines, but also showing how established and emerging standards could be used to streamline this process if a control provider develops a new control product line.

Section 7 describes how to formally verify that a control sequence that is implemented on a real control hardware conforms to the CDL specification. It presents an actual example that illustrates the verification, and closes with specifications for how to automate such a verification.

Section 8 presents an example in which we compared the annual energy peformance of two different control sequences applied to the same building and HVAC system. In this example, simply changing the control sequence led to about 30% annual savings in HVAC site electricity use.

Section 9 describes the benefits to the California rate payers.

Section 10 explains technical terms used throughout the report.

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/.

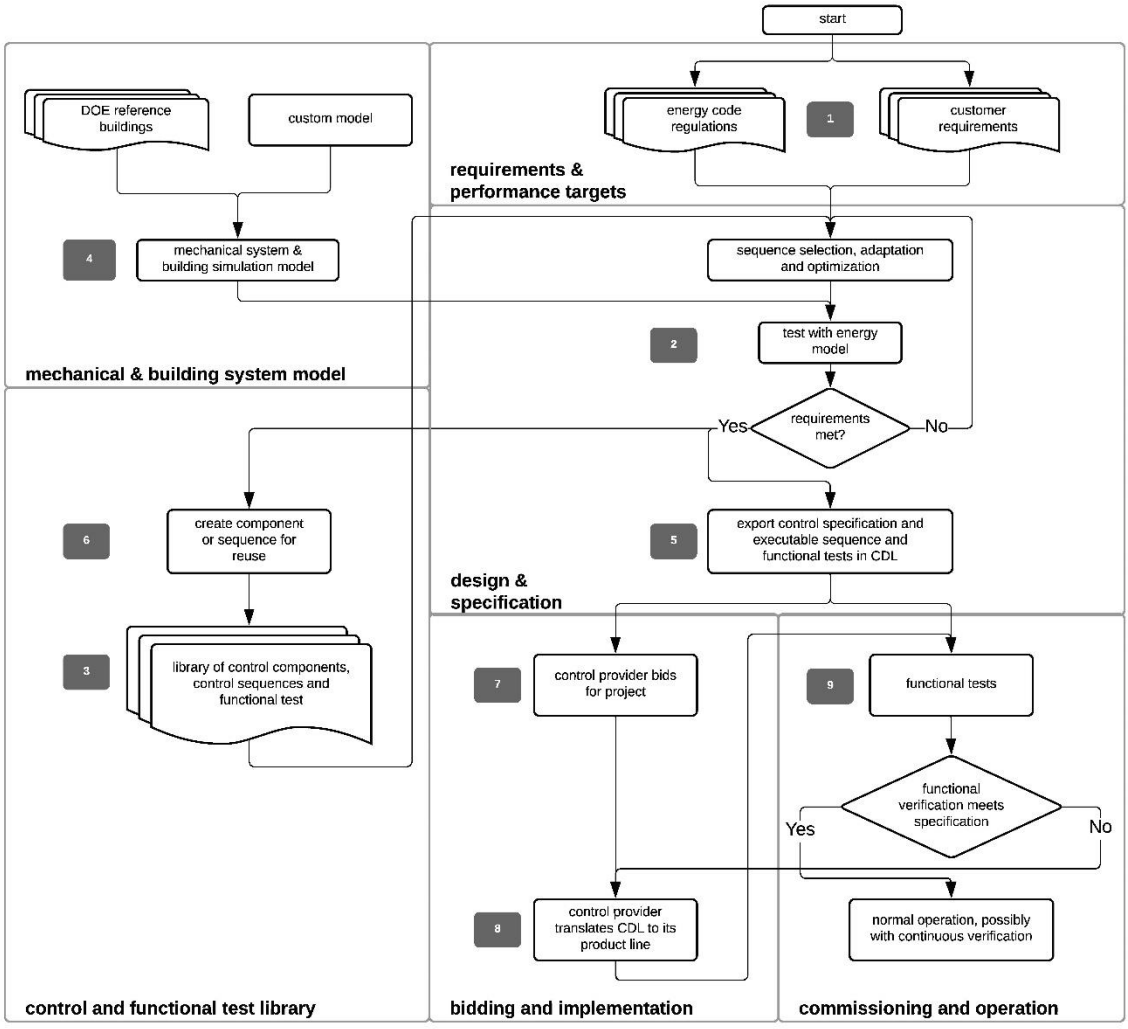


Fig. 3.1: *Process workflow for controls design, specification and functional verification.*

Chapter 4

Control Description Language

4.1 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. 4.1 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 6.3 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 or English language documentation of the control sequences. We anticipate that future control product lines use directly CDL as shown in the right of Fig. 4.1. Such a translation can then be done using various existing Modelica tools to generate code for real-time simulation.

The next sections define the CDL language. A collection of control sequences that are composed using the CDL language is described in Section 5. These sequences can be simulated with Modelica simulation environments. The translation of

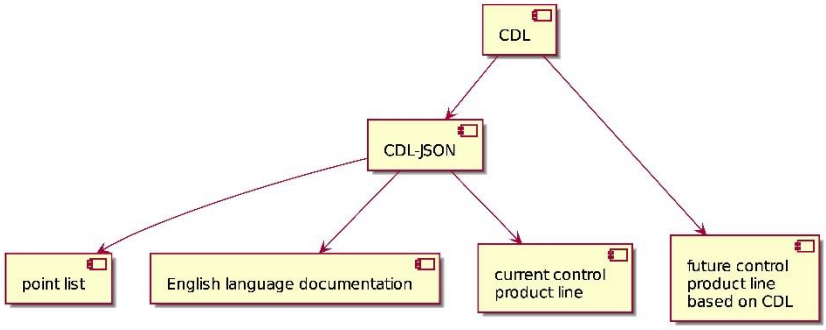


Fig. 4.1: *Translation of CDL to the CDL-JSON intermediate format and to a product line or English language documentation.*

such sequences to control product lines using `modelica-json`, or other means of translation, is described in Section 6.

4.2 Basic Elements of CDL

The CDL consists of the following elements:

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

4.3 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 3.3 specification [Mod12]. The selected subset is needed to instantiate classes, assign parameters, connect objects and document classes. This subset is fully compatible with Modelica, e.g., no 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:

1. `extends`
2. `redeclare`
3. `constrainedby`

4. `inner` and `outer`

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

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

4.4 Permissible Data Types

4.4.1 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.]

4.4.1.1 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
  parameter RealType nominal;            // 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 4.10.]

[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 4.14.2).]

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.

4.4.1.2 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.]

4.4.1.3 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.]

4.4.1.4 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;
```

4.4.1.5 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`.]

4.4.2 Parameter and constant declarations

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 4.8). 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";
```

]

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;
```

]

4.4.3 Arrays

Each of these data types, including the elementary building blocks, 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.

Arrays may be constructed using the notation `{x1,x2,...}`, for example `parameter Integer k[3,2] = {{1, 2},{3,4},{5,6}}`, or using one or several iterators, for example `parameter Real k[2,3] = {i*0.5+j for i in 1:3, j in 1:2};`. They can also be constructed using a `fill` or `cat` function, see Section 4.7.1.

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 3.3 specification Chapter 10 for array notation and these functions.

4.5 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 building blocks* (Section 4.6) or *composite blocks* (Section 4.12).

4.6 Elementary Building Blocks

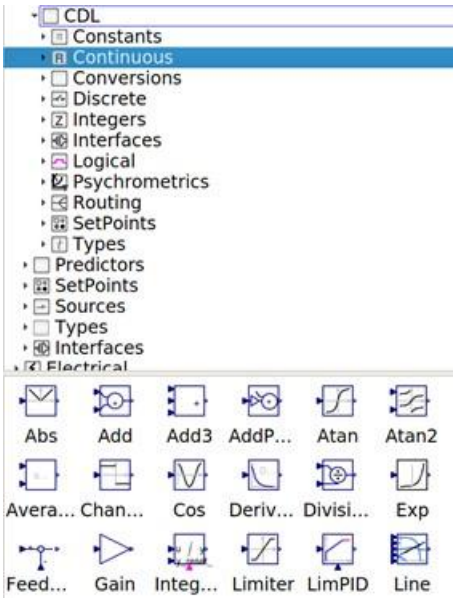


Fig. 4.2: *Screenshot of CDL library.*

The CDL library contains elementary building blocks that are used to compose control sequences. The functionality of elementary building blocks, but not their implementation, is part of the CDL specification. Thus, in the most general form, elementary building 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 building blocks, because the implementation is language specific. However, implementations shall have the same inputs, outputs and parameters, and they shall compute the same response for the same value of inputs and state variables.]

Users are not allowed to add new elementary building blocks. Rather, users can use the existing elementary blocks to implement composite blocks (Section 4.12).

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

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

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

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

  parameter Real p "Value to be added";
  parameter Real k "Gain of input";

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

equation
  y = k*u + p;

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

For the complete implementation, see the github repository.

4.7 Instantiation

4.7.1 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 4.1. For `Real` and `Integer`, expressions are allowed that involve

- the basic arithmetic functions +, −, *, −,
- the relations >, >=, <, <=, ==, <>,
- calls to the functions listed in Table 4.1.

Table 4.1: Functions that are allowed in parameter assignments. The functions are consistent with Modelica 3.3.

Function	Descrition
`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.

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

```
Continuous.Gain 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.]

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 con- troller 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.Continuous.Gain with parameter `k = 1/k1`, one could use a constant source CDL.Continuous.Sources.Constant with parameter `k=k1` and two instances of CDL.Continuous.Division, and then connect the output of the constant source with the inputs of the division blocks.

4.7.2 Evaluation of Assignment of Values to Parameters

We will now describe how assignments of values to parameters can optionally be evaluated by the CDL translator. While such an evaluation is not prefered, it is allowed in CDL to accomodate the situation that most building control product lines, in constrast 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";

CDL.Continuous.Sources.Constant con(
  k = pRel) "Block producing constant output";
CDL.Logical.Hysteresis hys(
```

(continues on next page)

(continued from previous page)

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

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

4.7.3 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.Continuous.Gain gai(
  k = VOutPerPer_flow) if have_occSen
    "Outdoor air per person";
equation
connect(nOcc, gai.u);
```

By the Modelica language definition, all connections (Section 4.10) 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 K, 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`.
- For `DayTypeInput`, the default value is `WorkingDay`.

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).

4.8 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`, `DayTypeInput`, `DayTypeOutput`, `IntegerInput`, `IntegerOutput`, `RealInput` and `RealOutput`. `DayType` is an `enumeration` for working day, non-working day and holiday.

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.Continuous.Sources.Constant con[n](k={1, 2});
CDL.Continuous.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.Continuous.Sources.Constant con_1(k=1);
CDL.Continuous.Sources.Constant con_2(k=1);
CDL.Continuous.MultiSum mulSum(nin=2);
```

(continues on next page)

(continued from previous page)

```
equation
connect(con_1.y, mulSum.u_1);
connect(con_2.y, mulSum.u_2);
```

E.g., two instances of `CDL.Continuous.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.

4.9 Equations

After the instantiations (Section 4.7), a keyword `equation` must be present to introduce the equation section. The equation section can only contain connections (Section 4.10) and annotations (Section 4.11).

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 building block.)

4.10 Connections

Connections connect input to output connector (Section 4.8). 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 4.11). 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

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

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

]

Only connectors that carry the same data type (Section 4.4.1) 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. Otherwise an error should be issued.
- 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,

```
Continuous.Max maxValue(y(unit="m/s")) "Output maximum value";
Continous.Gain gain(                     k=60) "Gain";
Continous.Gain gainOK(   u(unit="m/s" ), k=60) "Gain";
Continous.Gain 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 connnector is not allowed.

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

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

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

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

4.11 Annotations

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

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

[For CDL, annotations are primarily used to graphically visualize block layouts, graphically visualize input and output signal connections, and to declare vendor annotations, (Sec. 18.1 in Modelica 3.3 Specification), 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 4.7.3.

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.

4.12 Composite Blocks

CDL allows building composite blocks such as shown in Fig. 4.3.

Composite blocks can contain other composite blocks.

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

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

[The following statement, when saved as `CustomPWithLimiter.mo`, is the declaration of the composite block shown in Fig. 4.3

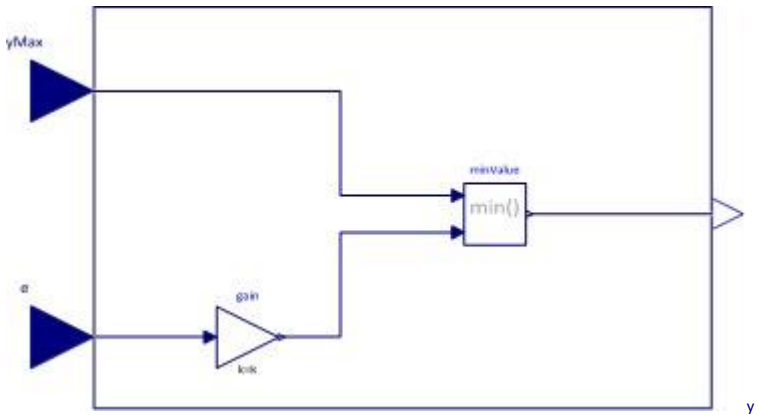


Fig. 4.3: *Example of a composite control block that outputs* $y = \min(k\,e, y_{max})$ *where* k *is a parameter.*

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

  parameter Real k "Constant gain";

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
  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.]

4.13 Model of Computation

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

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

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

4.14 Tags

CDL has sufficient information for tools that process CDL to generate for example point lists that list all analog temperature sensors, or to verify that a pressure control signal is not connected to a temperature input of a controller. Some, but not all, of this information can be inferred from the CDL language described above. We will use tags, implemented through Modelica vendor annotations, to provide this additional information. In Section 4.14.1, we will explain the properties that can be inferred, and in Section 4.14.2, we will explain how to use tagging schemes in CDL.

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

4.14.1 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.)

]

4.14.2 Tagged Properties

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

CDL allows to add tags to declarations that instantiate

- elementary building blocks (Section 4.6), and
- composite blocks (Section 4.12).

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

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

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

where `_cdl_annotation` is the annotation for CDL.

For Brick, the `_cdl_annotation` is

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

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

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

For example, the Brick specification

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

can be declared in CDL as

```
annotation(_cdl(brick="soda_hall:flow_sensor_SODA1F1_VAV_AV a brick:Supply_Air_Flow_Sensor
→;
  bf:hasTag brick:Average ;
  bf:isLocatedIn soda_hall:floor_1 ."));
```

]

For Haystack, the `cdl_annotation` is

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

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

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

```
id: @whitehouse.ahu3.dat
dis: "White House AHU-3 DischargeAirTemp"
point
siteRef: @whitehouse
equipRef: @whitehouse.ahu3
discharge
air
temp
sensor
kind: "Number"
unit: "degF"
```

can be declared in CDL as

```
annotation(_cdl( haystack=
  "{\"id\"   : \"@whitehouse.ahu3.dat\",
    \"dis\" : \"White House AHU-3 DischargeAirTemp\",
    \"point\" : \"m:\",
    \"siteRef\" : \"@whitehouse\",
    \"equipRef\" : \"@whitehouse.ahu3\",
    \"discharge\" : \"m:\",
```

(continues on next page)

(continued from previous page)

```
    \"air\"       : \"m:\",
    \"temp\"      : \"m:\",
    \"sensor\"    : \"m:\",
    \"kind\"       : \"Number\"
    \"unit\"       : \"degF\"}"));
```

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

Chapter 5

Controls Library

5.1 Introduction

To implement control sequences that conform to the CDL specification of Section 4, we implemented a library of elementary control blocks, and a library of control sequences that are composed of these elementary control 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.

5.2 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 control blocks that are described in Section 4.6. This library consist of about 130 elementary control 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 6.3.

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

5.3 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.

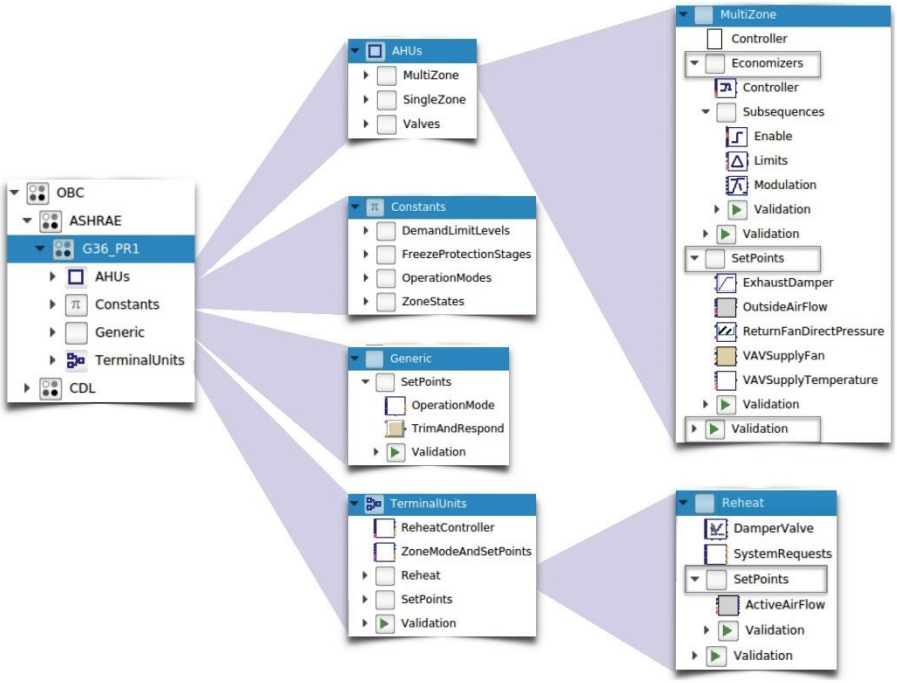


Fig. 5.1: *Overview of package that includes control sequences from ASHRAE Guideline 36.*

For example, Fig. 5.1 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. 5.2 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).

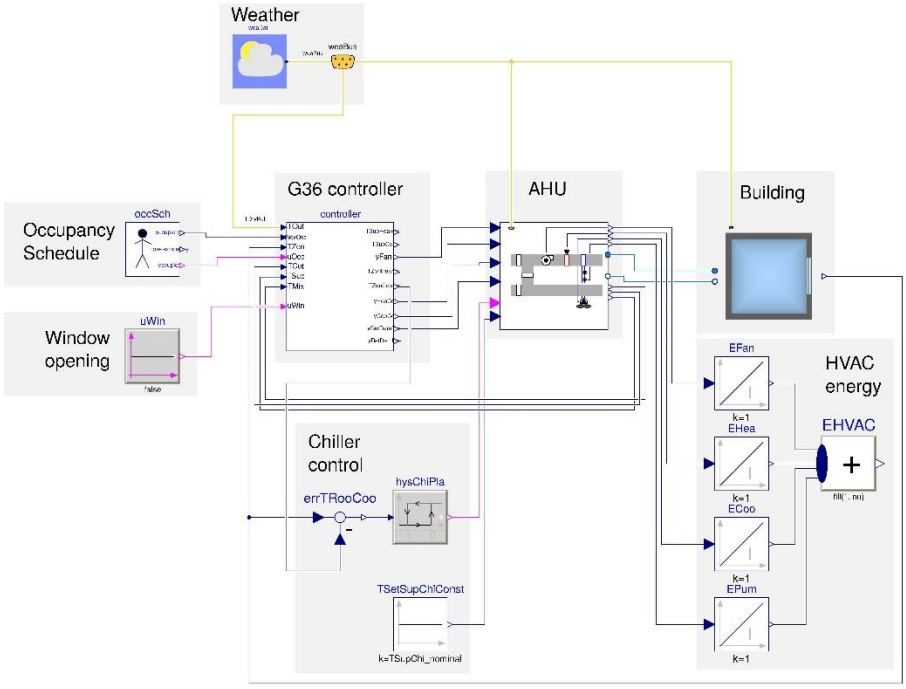


Fig. 5.2: *Schematic view of model that uses the CDL implementation of the single zone VAV controller based on ASHRAE Guideline 36.*

Chapter 6

Code Generation

6.1 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. 6.1 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 6.3).
2. Export of the whole CDL-compliant sequence using the *FMI standard* (Section 6.4), 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 6.5).

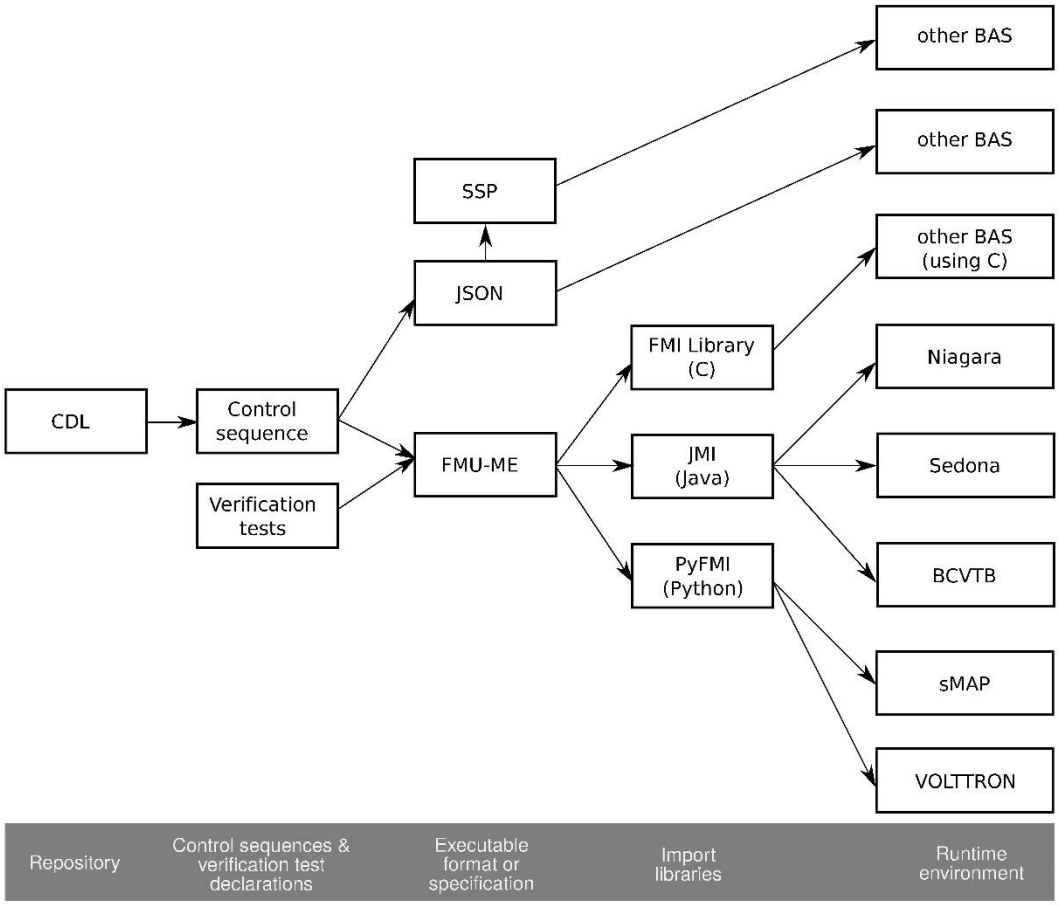


Fig. 6.1: *Overview of the code export and import of control sequences and verification tests.*

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.

6.2 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.

6.3 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. 4.3 and reproduced in Fig. 6.2.

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)

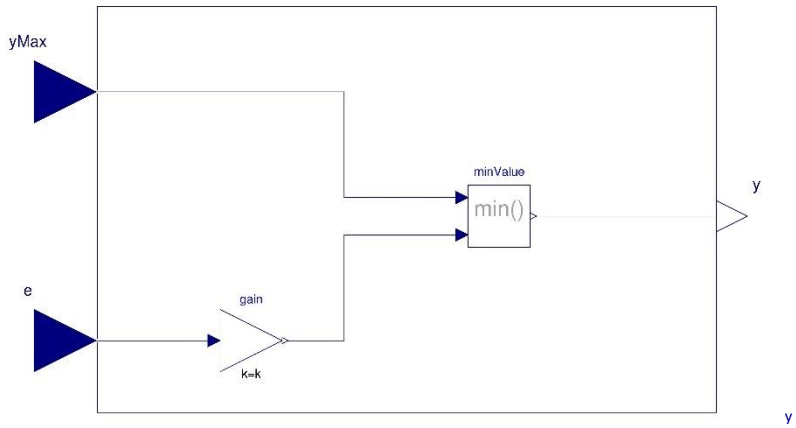


Fig. 6.2: *Example of a composite control block that outputs* $y = \max(k\,e, y_{max})$ *where* k *is a parameter.*

(continued from previous page)

```
  CDL.Interfaces.RealOutput y "Control signal"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
  CDL.Continuous.Gain gain(final k=k) "Constant gain"
    annotation (Placement(transformation(extent={{-60,-50},{-40,-30}})));
  CDL.Continuous.Min minValue "Outputs the minimum of its inputs"
    annotation (Placement(transformation(extent={{20,-10},{40,10}})));
equation
  connect(yMax, minValue.u1) annotation (
    Line(points={{-120,40},{-120,40},{-20,40},{-20, 6},{18,6}}, color={0,0,127}));
  connect(e, gain.u) annotation (
    Line(points={{-120,-40},{-92,-40},{-62,-40}}, color={0,0,127}));
  connect(gain.y, minValue.u2) annotation (
    Line(points={{-39,-40},{-20,-40},{-20,-6}, {18,-6}}, color={0,0,127}));
  connect(minValue.y, y) annotation (
     Line(points={{41,0},{110,0}}, color={0,0,127}));
  annotation (Documentation(info="<html>
<p>
Block that outputs <code>y = min(yMax, k*e)</code>,
where
<code>yMax</code> and <code>e</code> are real-valued input signals and
<code>k</code> is a parameter.
</p>
</html>"));
end CustomPWithLimiter;
```

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:

(continues on next page)

(continued from previous page)

```
{
  "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.Continuous.Gain",
        "name": "gain",
        "comment": "Constant gain",
        "modifications": [
          {
            "name": "k",
            "value": "k",
            "isFinal": true
          }
        ]
      },
      {
        "className": "CDL.Continuous.Min",
        "name": "minValue",
```

(continues on next page)

(continued from previous page)

```
            "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"
        },
        {
          "instance": "y"
        }
      ]
    ]
  }
]
```

Note that the graphical annotations are not shown. The JSON representation can then be parsed and converted to another

block-diagram language. Note that `CDL.Continuous.Gain` is an elementary CDL block (see Section 4.6). If it were a composite CDL block (see Section 4.12), 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://raw.githubusercontent.com/lbl-srg/modelica-json/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.

6.4 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 4.6) can be used to compose sequences that conform to the CDL language specification described in Section 4. For verification tests, any Modelica block can be used. Next, to export the Modelica model, a Modelica tool such as OpenModelica, JModelica, 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_PR1.AHUs.SingleZone.Economizers.Controller")
```

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

- Tools based on Python, which could be used to interface with sMAP (https://pythonhosted.org/Smap/en/2.0/index.html) or Volttron (https://energy.gov/eere/buildings/volttron):
  - PyFMI (https://pypi.python.org/pypi/PyFMI)

- Tools based on Java:
  - Building Controls Virtual Test Bed (http://simulationresearch.lbl.gov/bcvtb)
  - JFMI (https://ptolemy.eecs.berkeley.edu/java/jfmi/)
  - JavaFMI (https://bitbucket.org/siani/javafmi/wiki/Home)
- Tools based on C:
  - FMI Library (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/)
  - JModelica (https://www.jmodelica.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.simulationx.com/)
  - SystemModeler (http://www.wolfram.com/system-modeler/index.html)

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

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

6.5 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 connec- tions, and their graphical layout. The SSP standard allows for transporting complex networks of FMUs between different platforms for simulation, hardware-in-the-loop and model-in-the-loop [KohlerHM+16]. Various tools that can simulate systems specified using the SSP standard are in development, with FMI composer (http://www.modelon.com/products/modelon-deployment-suite/fmi-composer/) from Modelon being commercially available.

CDL-compliant control sequences could be exported to the SSP standard as shown in Fig. 6.3.

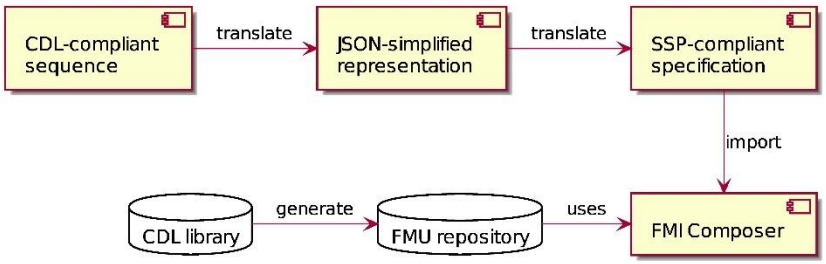


Fig. 6.3: Translation of CDL to SSP.

In such a workflow, a control vendor would translate the elementary CDL blocks (Section 4.6) to a repository ofFMU-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 6.3. 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 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.

6.6 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.)

6.6.1 Substitutions that Give Identical Control Response

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

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

6.6.2 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 [Mod12] for implementation details if C code or compiled libraries are provided. If a compiled library is provided, then binaries shall be provided for Windows 32/64 bit, Linux 32/64 bit, and OS X 64 bit.

6.6.3 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 6.6.2.

Chapter 7

Verification

7.1 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 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.

7.2 Terminology

We will use the following terminology, see also Section 4 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.

7.3 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.

7.4 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. 7.1 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

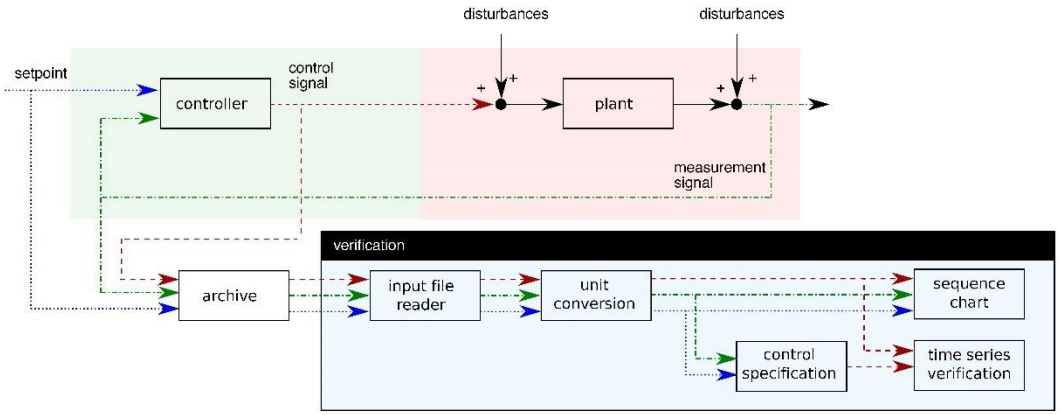


Fig. 7.1: *Overview of the verification that tests whether the installed control sequence meets the specification.*

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.

7.5 Modules of the Verification Test

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

7.5.1 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.

7.5.2 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.

7.5.3 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. 7.2 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

7.5.4 Verification of Sequence Diagrams

To verify sequence diagrams we developed the Modelica package `Buildings.Utilities.IO.Plotters`. Fig. 7.3 shows an example in which this block is used to produce the sequence diagram shown in Fig. 7.4. 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. 7.3 generates an html file that contains the scatter plots shown in Fig. 7.5.

7.6 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

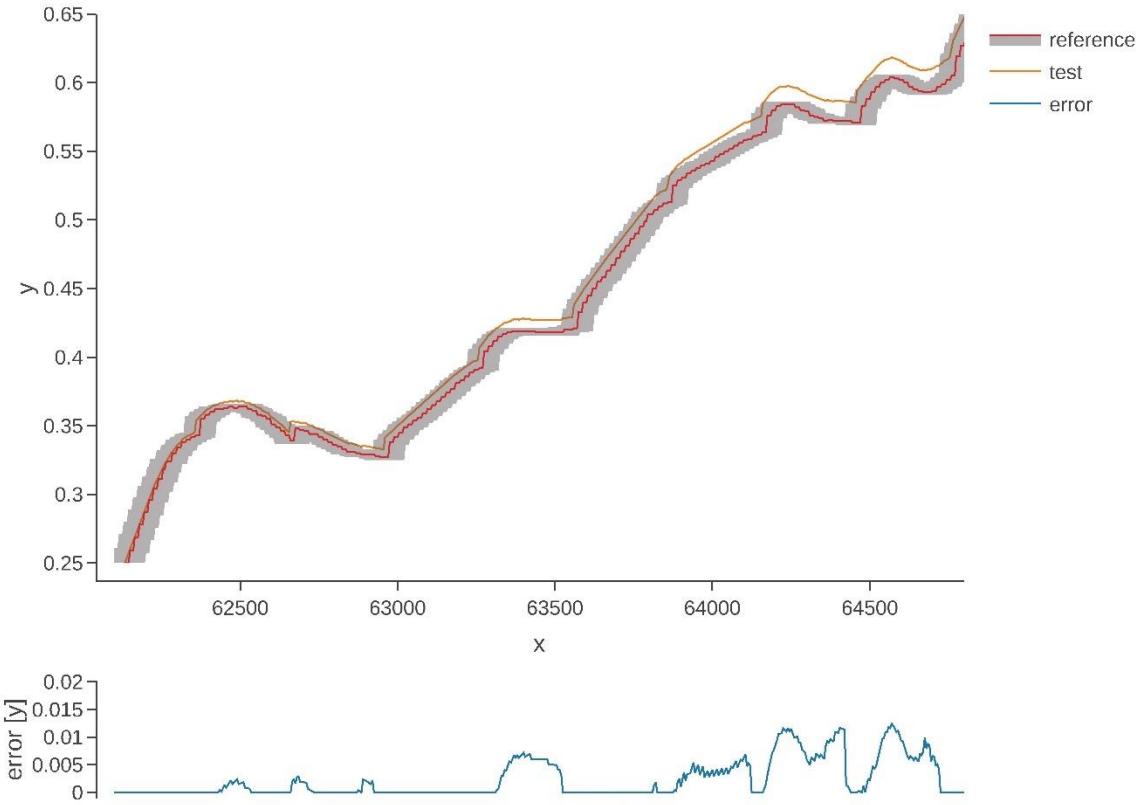


Fig. 7.2: *Typical plot generated by* `pyfunnel.plot_funnel for`*r comparing test and reference time series.*

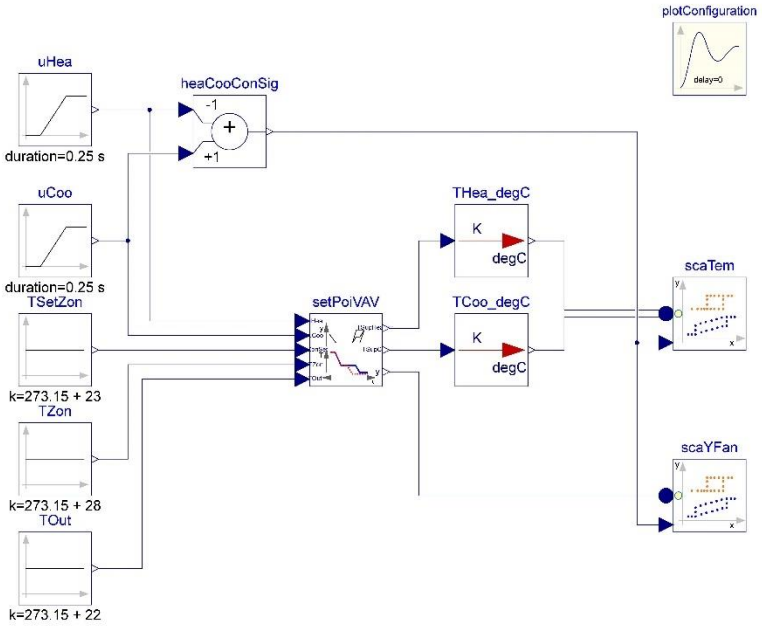


Fig. 7.3: *Modelica model that verifies the sequence diagram. On the left are the blocks that generate the control input time series. In a real verification, these would be replaced with a file reader that reads data that have been archived by the building automation system. In the center is the control 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.*

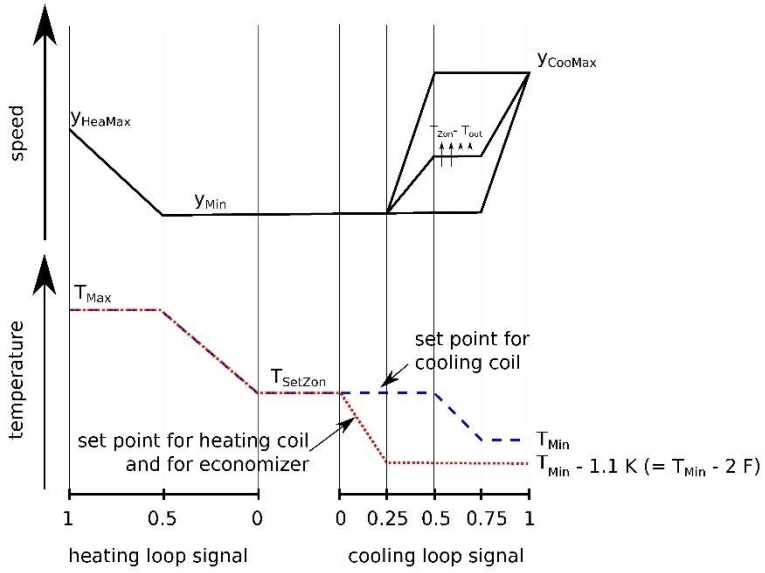


Fig. 7.4: *Control sequence diagram for the VAV single zone control sequence from ASHRAE Guideline 36.*

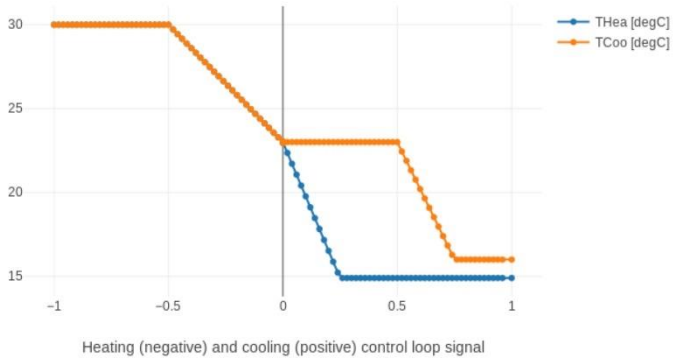


Fig. 7.5: *Scatter plots that show the control sequence diagram generated from the simulated sequence.*

campus in Berkeley, CA. The subsequence is shown in Fig. 7.6. 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. 7.7, 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. 7.8) and the controller gains (see Fig. 7.9), 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.Continuous.LimPID. The ALC controller tracks the temperature in degree Fahrenheit, while CDL uses SI units. An additional implementation difference is that for cooling applications, the ALC controller uses direct control action, whereas the CDL controller needs to be configured to use reverse control action, which can be done by setting its parameter *reverseAction=true*. Furthermore, the ALC controller outputs the control action in percentages, while the CDL controller outputs a signal between 0 and 1. To reconcile the differences, the ALC controller gains were converted for CDL as follows: The proportional gain $k_{p,cdl}$ was set to $k_{p,cdl} = u\, k_{p,alc}$, 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. 7.10 shows the Modelica model that was used to conduct the verification. On the left hand side are the data readers

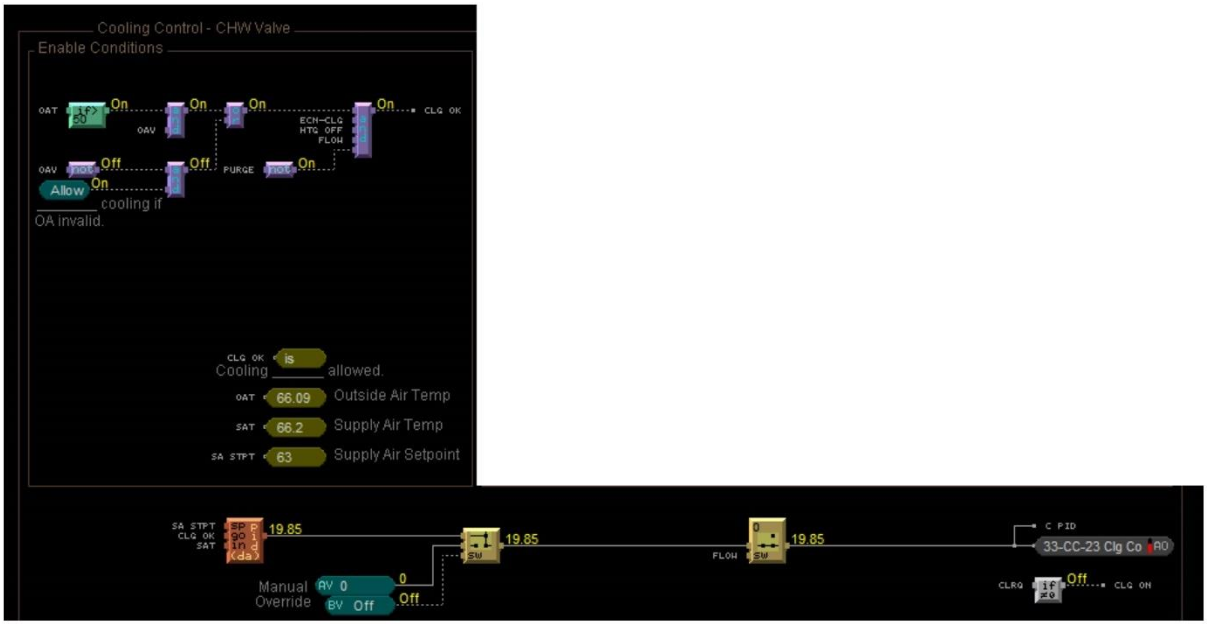


Fig. 7.6: *ALC EIKON specification of the cooling coil valve position control sequence.*

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. 7.7. The plotters on the right hand side then compare the simulated cooling coil valve position with the trended time series.

Fig. 7.11, 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. 7.12.

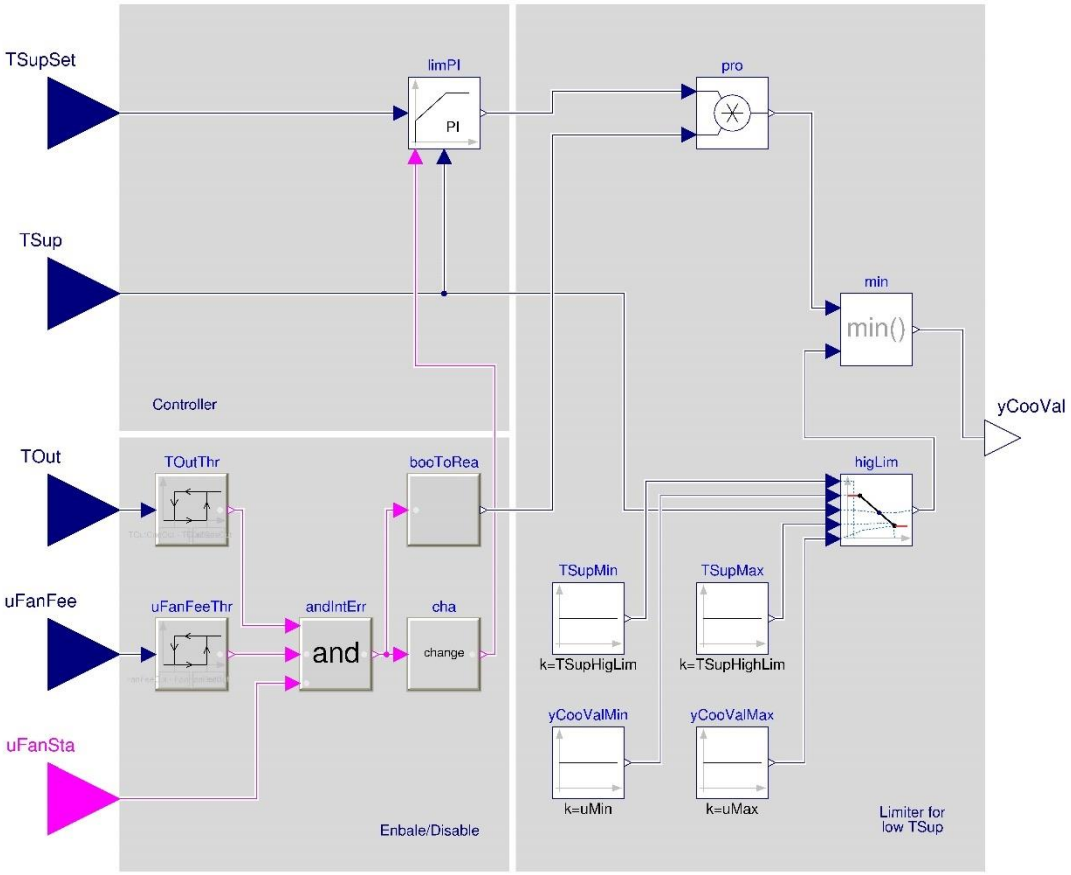


Fig. 7.7 : *CDL specification of the cooling coil valve position control sequence.*

Fig. 7.8: *ALC EIKON outdoor air temperature hysteresis to enable/disable the controller*

Fig. 7.9: *ALC EIKON PI controller parameters*

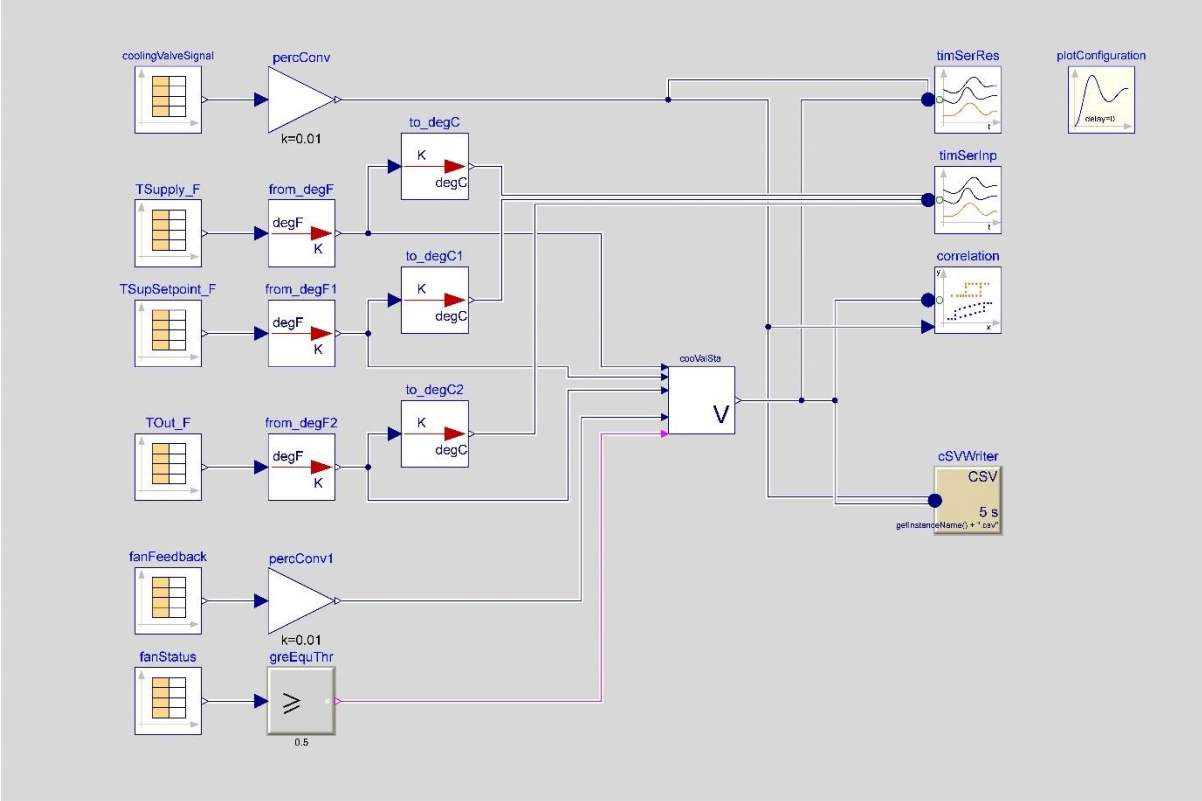
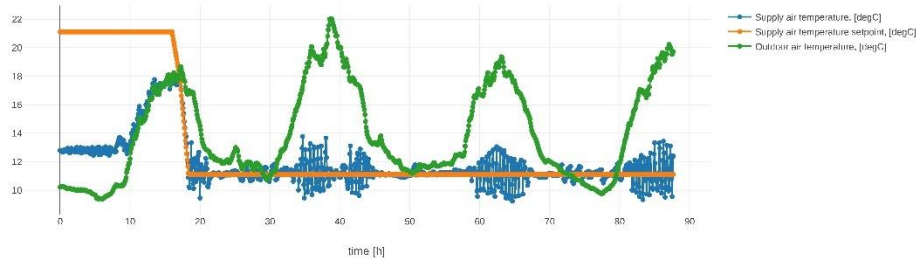
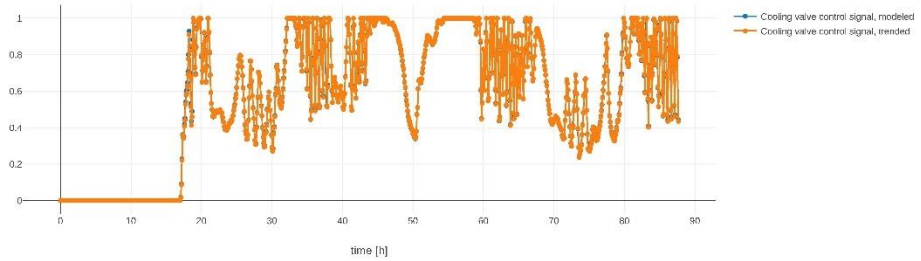


Fig. 7.10: *Modelica model that conducts the verification.*

Trended input signals



Cooling valve control signal: reference trend vs. modeled result



Modeled result/recorded trend correlation

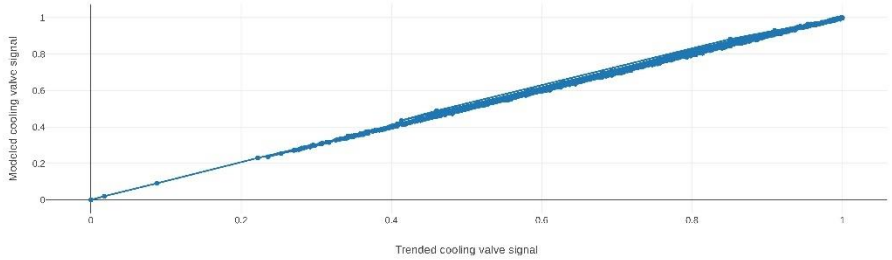


Fig. 7.11: *Verification of the cooling valve control signal between ALC EIKON computed signal and simulated signal.*

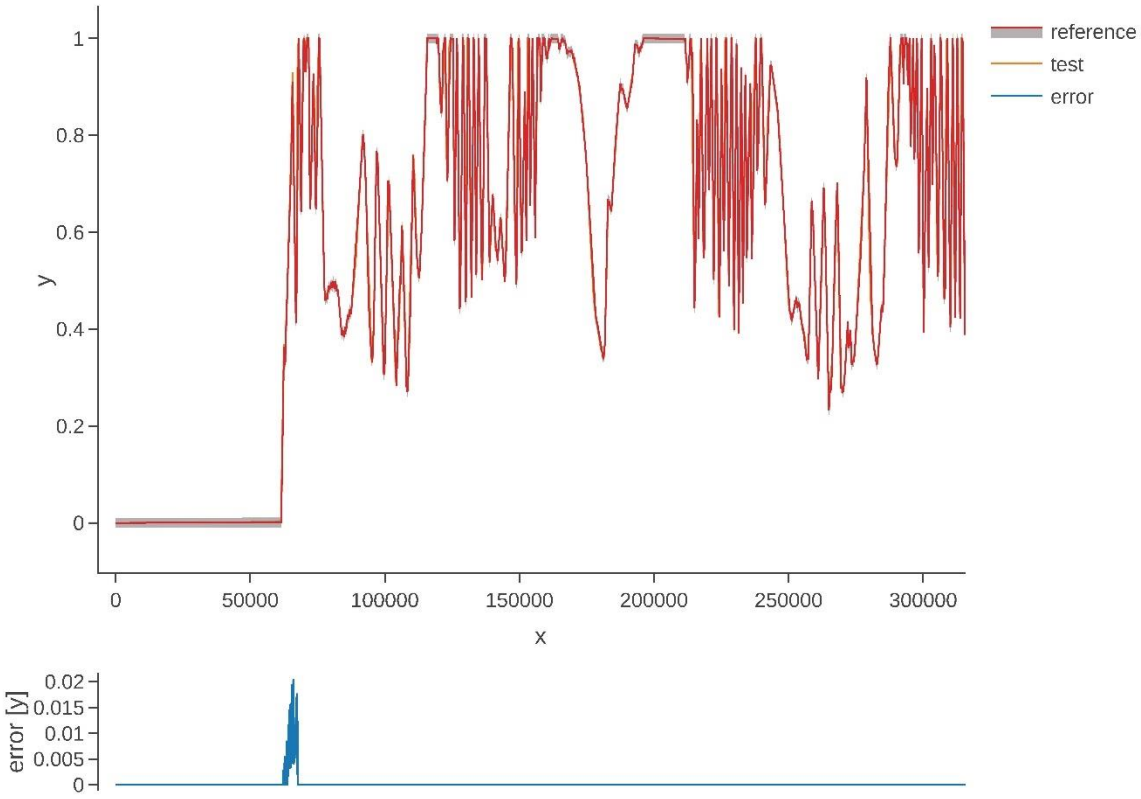


Fig. 7.12: *Verification of the cooling valve control signal with the funnel software (error computed with an absolute tolerance in time of 1 s and a relative tolerance in* y *of 1%).*

7.7 Specification for Automating the Verification

The example Section 7.6 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 7.6, 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 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 7.5.1.

7.7.1 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.

7.7.2 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.

For this scenario, we are given the following data:

i. A list of CDL models, and for each model, the instance name of one control sequence 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_PR1.AHUs.SingleZone.VAV.SetPoints.Validation.Supply_u. To verify the sequences of its instances `setPoiVAV` and `setPoiVAV1`, a specification may be

```
{
  "references": [
    {
      "model": "Buildings.Controls.OBC.ASHRAE.G36_PR1.AHUs.SingleZone.VAV.SetPoints.
↪Validation.Supply_u",
      "sequence": "setPoiVAV",
      "pointNameMapping": "realControllerPointMapping.json"
    },
    {
      "model": "Buildings.Controls.OBC.ASHRAE.G36_PR1.AHUs.SingleZone.VAV.SetPoints.
↪Validation.Supply_u",
      "sequence": "setPoiVAV1",
      "pointNameMapping": "realControllerPointMapping.json",
      "outputs": [
        { "atoly": 0.5, "variable": "setPoiVAV1.TSup*" }
      ],
      "indicators": [
        { "setPoiVAV1.TSup*": [ "fanSta.y" ] }
      ],
      "sampling": 60
    }
  ],
  "tolerances": { "rtolx": 0.002, "rtoly": 0.002, "atolx": 10, "atoly": 0 },
  "sampling": 120
}
```

Listing 7.1: Configuration of test setup.

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 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`. 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`, and the sampling rate is 60 seconds. This test will also use `realControllerPointMapping.json` to map the variables to points of the real controller. 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.

To create test input and output time series, we generate CSV files. This needs to be done for each controller, and we will explain it only for the controller `setPoiVAV`. For brevity , we call `OBC.ASHRAE.G36_PR1.AHUs.SingleZone.VAV.SetPoints.Validation.Supply_u` simply `Supply_u`

The procedure is as follows:

1. Parse the model to json by running `modelica-json` as

   ```
   node app.js -f Buildings/Controls/OBC/ASHRAE/G36_PR1/AHUs/SingleZone/VAV/SetPoints/
   ↪Validation/Supply_u.mo -o json -d test1
   ```

   This will produce `Supply_u.json` (file name is abbreviated) in the output directory `test1`. 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 that we will call `reference_io.json`. Also, extract all public parameters of the instance `setPoiVAV` and store them in a file that we will call `reference_parameters.json`. For this sequence, the public parameters are `TSupSetMax`, `TSupSetMin`, `yHeaMax`, `yMin` and `yCooMax`.
3. Obtain reference time series by simulating `Supply_u.mo` to produce a CSV file `reference.csv` with time series of all input, output and indicator time series. This can be accomplished with the free open-source tool OpenModelica by running

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

   with the file `simulateReference.py` being

   ```
   import shutil
   from OMPython import OMCSessionZMQ
   model="Buildings.Controls.OBC.ASHRAE.G36_PR1.AHUs.SingleZone.VAV.SetPoints.Validation.
   ↪Supply_u"
   # Load and simulate the model
   omc = OMCSessionZMQ()
   omc.sendExpression("loadModel(Buildings)")
   omc.sendExpression("simulate({}, outputFormat=\"csv\")".format(model))
   # Copy output file
   shutil.move("{}_res.csv".format(model), "reference.csv")
   ```
4. To make a CSV file that only contains the control input time series, read `reference_io.json` to extract the names of the input variables of the sequence `setPoiVAV` and write the corresponing time series from `reference.csv` to a new file `reference_input.csv`.
5. Apply the parameters from `reference_parameters.json` to the real controller, and run the real controller for the input time series in `reference_input.csv`. Convert the units of the parameters and the time series as needed for the tested controller. Note that `reference_io.json` will contain the unit declarations.
6. Convert the output time series of the real controller to the units specified in `reference_io.json`, and write the

time series to a new file `controller_output.csv`. Use the CDL output variable names in the header of the CSV file.

7. Produce the test results by running the funnel software (https://github.com/lbl-srg/funnel) for each time series in `controller_output.csv` and in `reference.csv`. 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 7 are to be used to generate a test report for all tested sequences.

7.7.3 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_PR1.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 7.2, which is identical to Listing 7.1, 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 7.2: Specification of test setup.

```
references : [
  { "model": "Buildings.Controls.OBC.ASHRAE.G36_PR1.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 7.2.

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_PR1/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 7.7.2.
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
```

(continues on next page)

(continued from previous page)

```
import os
from OMPython import OMCSessionZMQ

model="Buildings.Controls.OBC.ASHRAE.G36_PR1.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 7.7.2.

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 8

Example Application

8.1 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 8.2.5, 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 8.2 we describe the methodology, the models and the performance metrics, in Section 8.3 we compare the performance, in Section 8.4 we recommend improvements to the Guideline 36 and in Section

8.5 we discuss the main findings and present concluding remarks.

8.2 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

8.2.1 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. 8.1 shows the schematic diagram of the HVAC system.

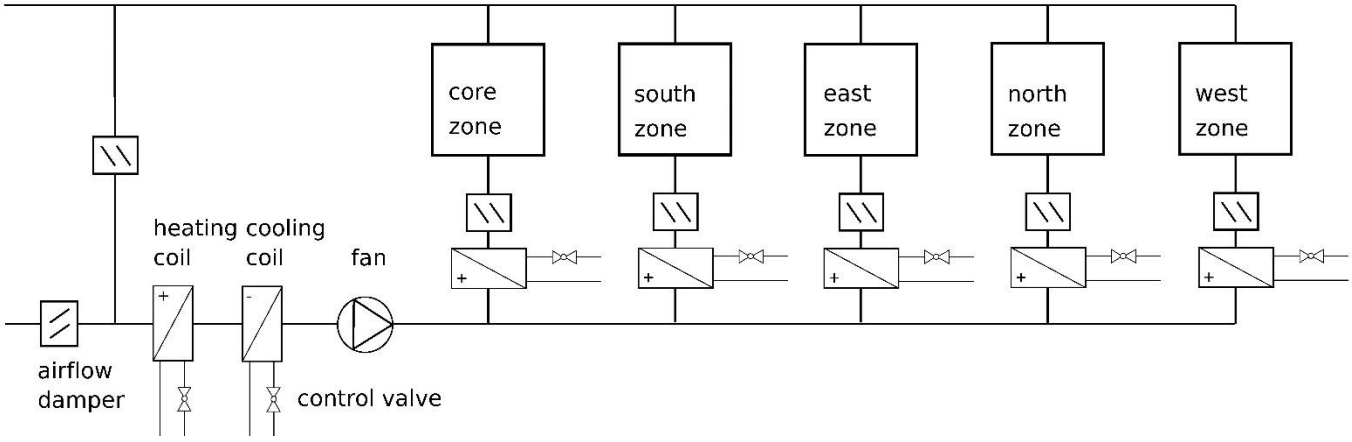


Fig. 8.1: Schematic diagram of the HVAC system.

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].

8.2.2 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.

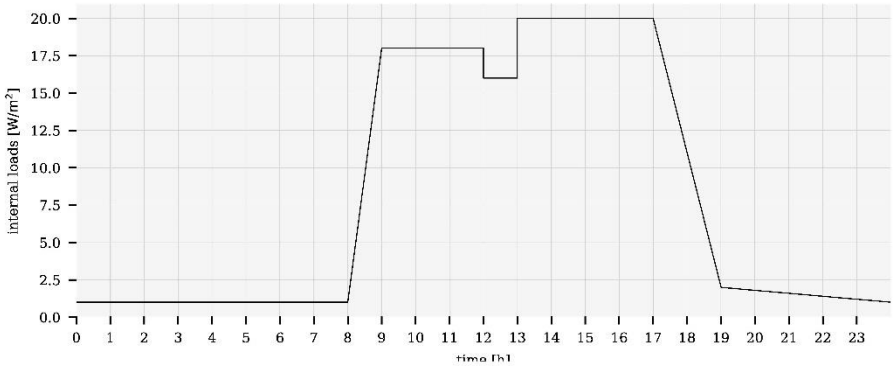


Fig. 8.2: Internal load schedule.

8.2.3 Internal Loads

We use an internal load schedule as shown in Fig. 8.2, of which 20% is radiant, 40% is convective sensible and 40% is latent. Each zone has the same internal load per floor area.

8.2.4 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].

8.2.5 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. 8.6.

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. 8.3 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. 8.4. 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. 8.5, 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.

8.2.6 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.

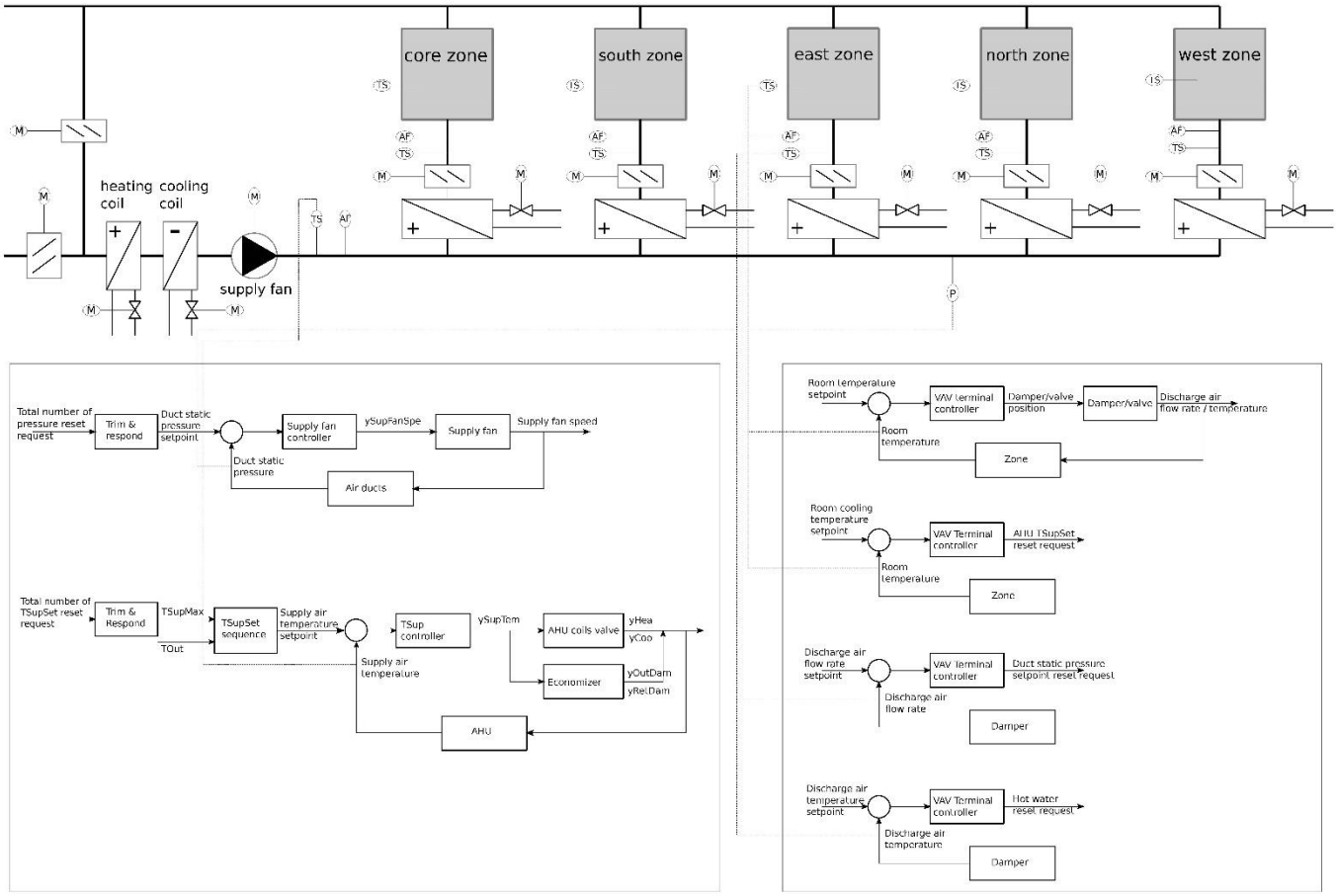


Fig. 8.3: *Control schematics of Guideline 36 case.*

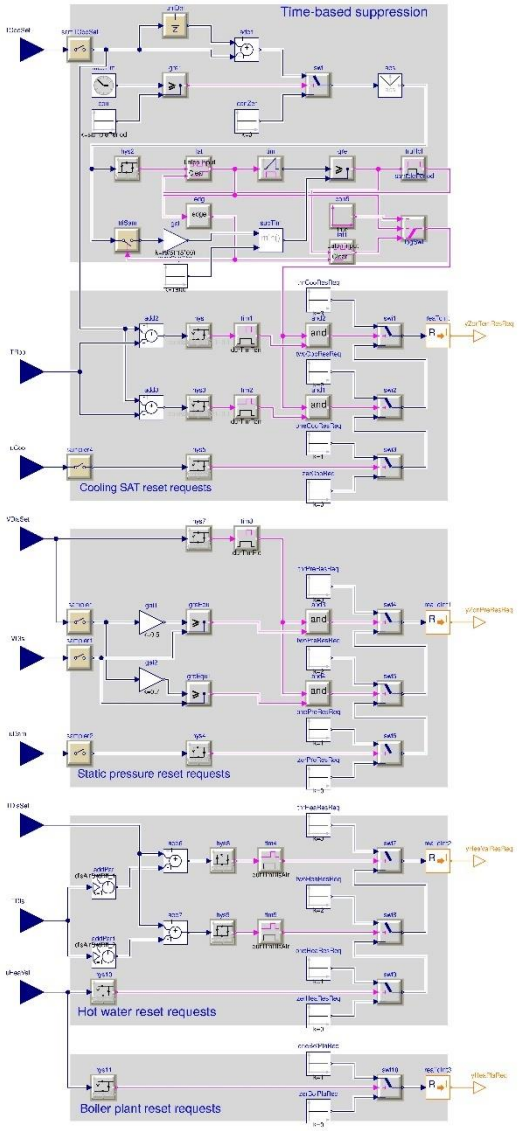


Fig. 8.4: *Composite block that implements the sequence for the VAV terminal units that output the system requests. (Browsable version.)*

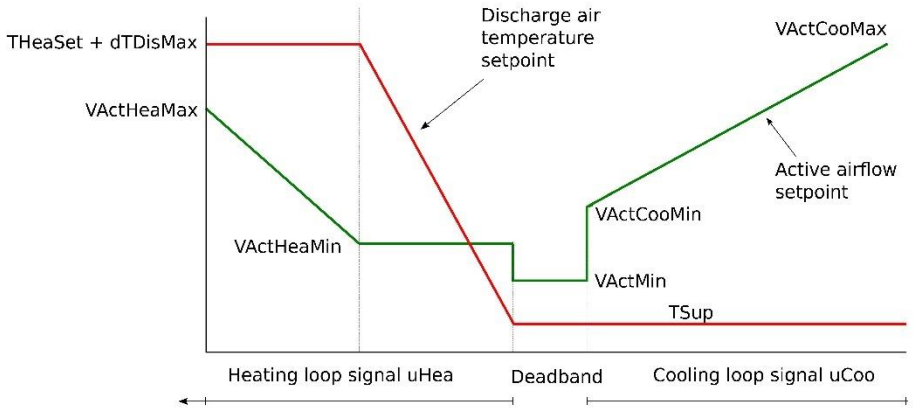


Fig. 8.5: Control sequence for VAV terminal unit.

8.2.7 Simulations

Fig. 8.6 shows the top-level of the system model of the base case, and Fig. 8.7 shows the same view for the Guideline 36 model.

The complexity of the control implementation is visible in Fig. 8.4 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 8.1 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 4.6) of CDL.

Table 8.1: 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

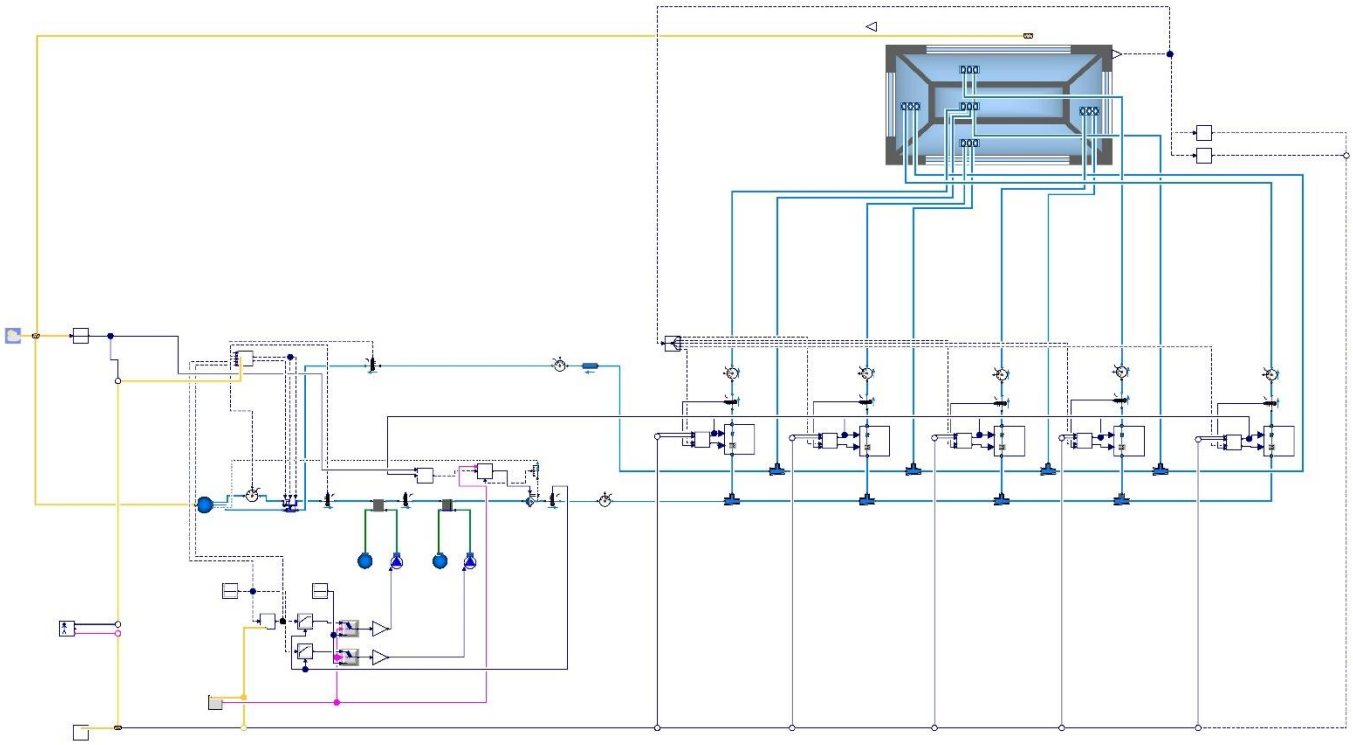


Fig. 8.6: *Top level view of Modelica model for the base case.*

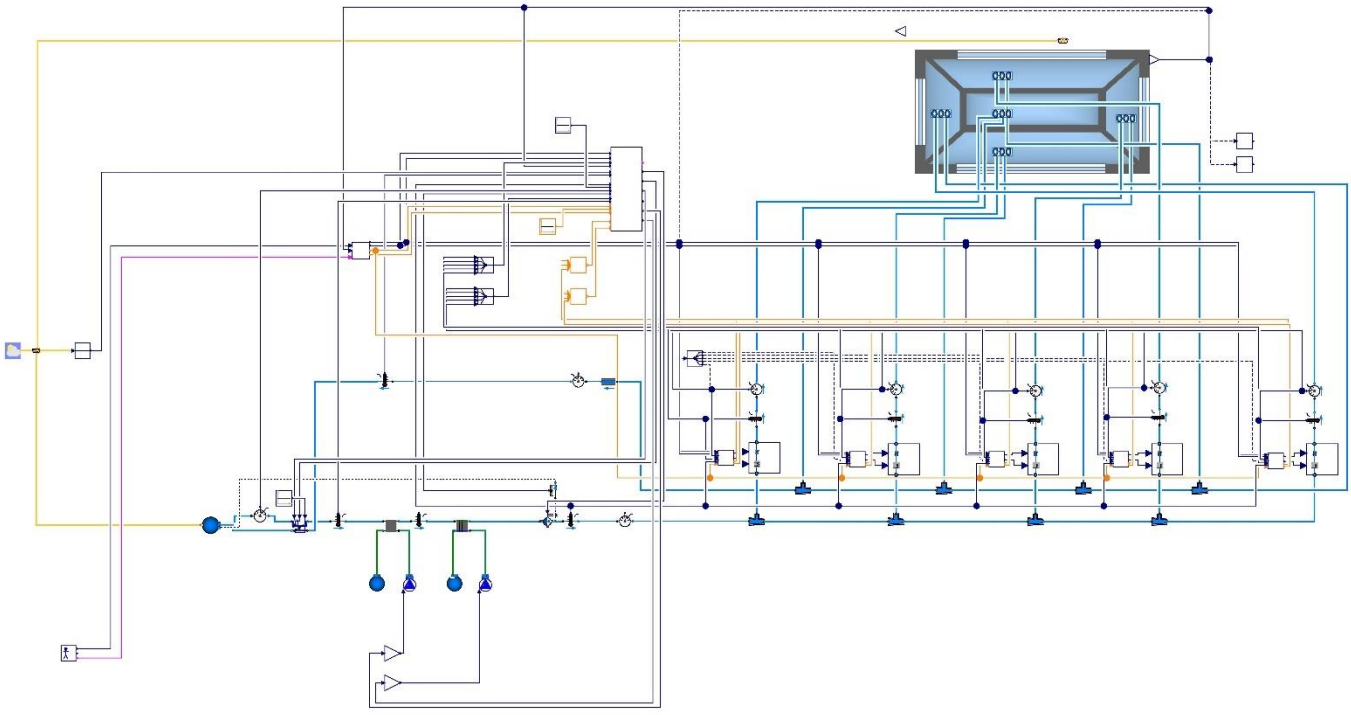


Fig. 8.7 : *Top level view of Modelica model for the Guideline 36 case.*

8.3 Performance Comparison

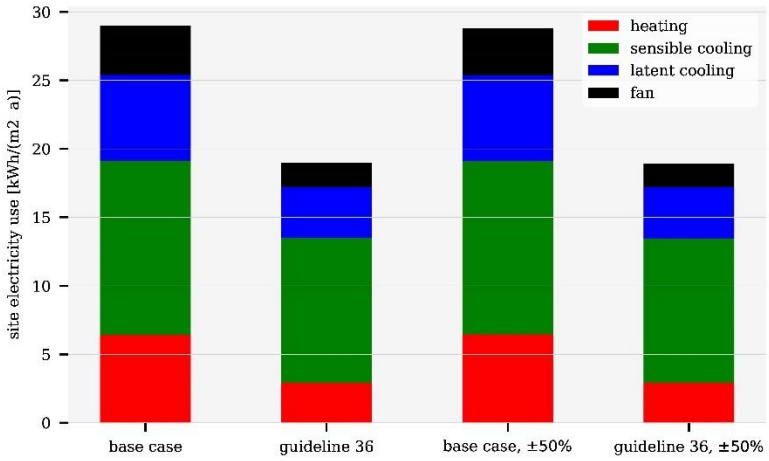


Fig. 8.8: Comparison of energy use. For the cases labeled ±50%, the internal gains have been increased and decreased as described in Section 8.2.3.

Table 8.2: 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. 8.8 and Table 8.2 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. 8.9 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. 8.10 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.

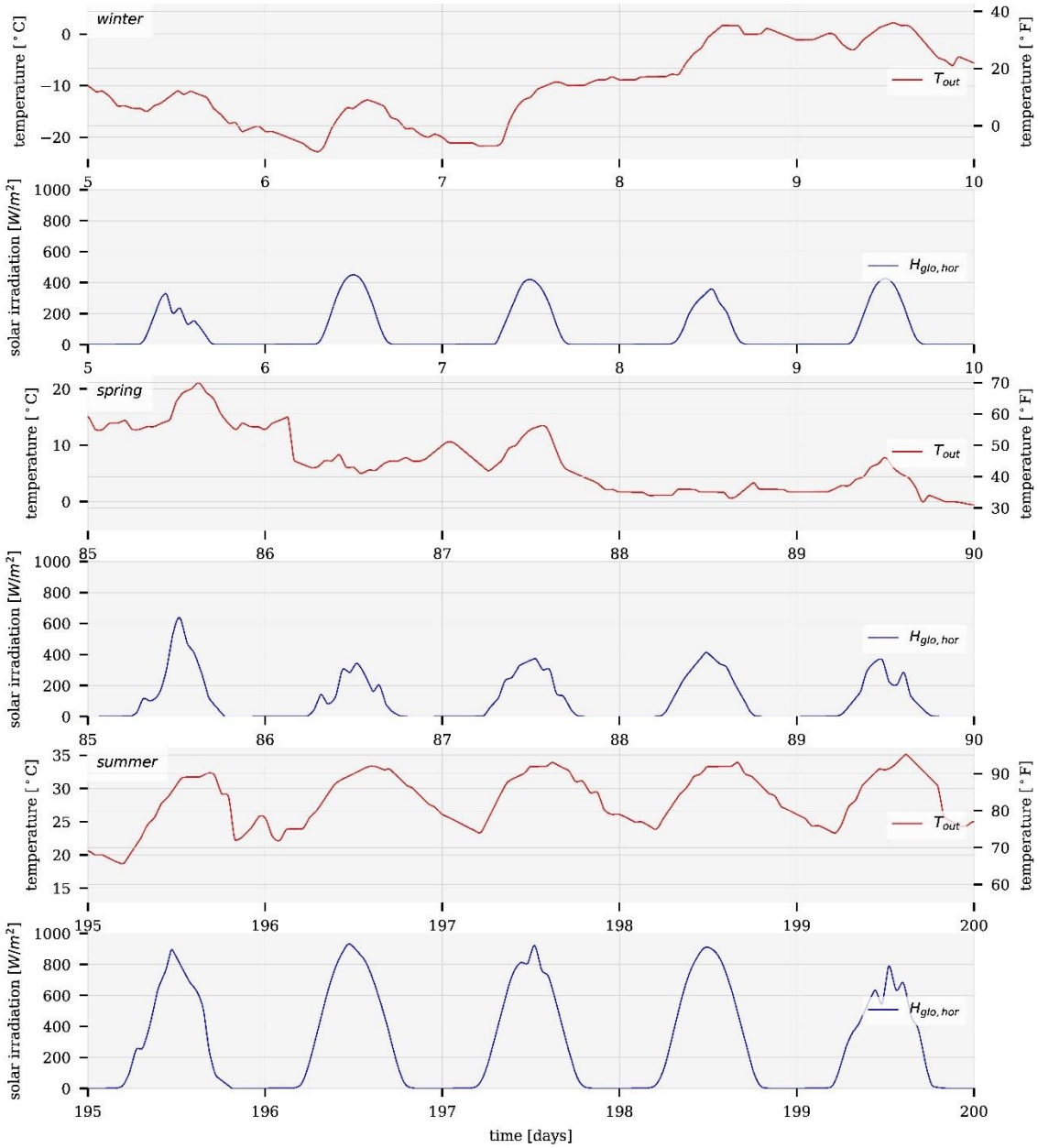


Fig. 8.9: *Outside air temperature and global horizontal irradiation for the three periods that will be further used in the analysis.*

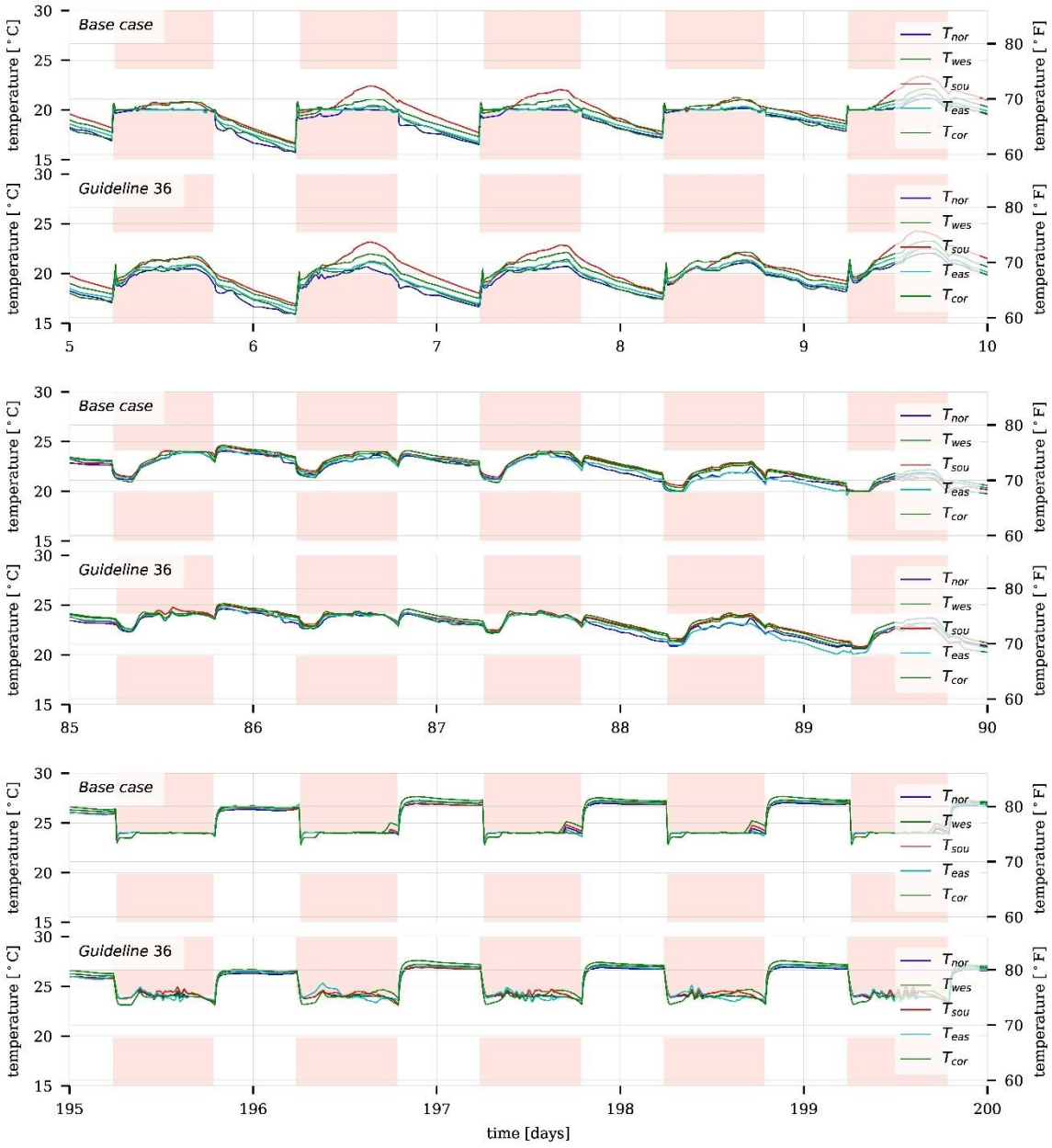


Fig. 8.10: *Room air temperatures. The white area indicates the region between the heating and cooling setpoint temperatures.*

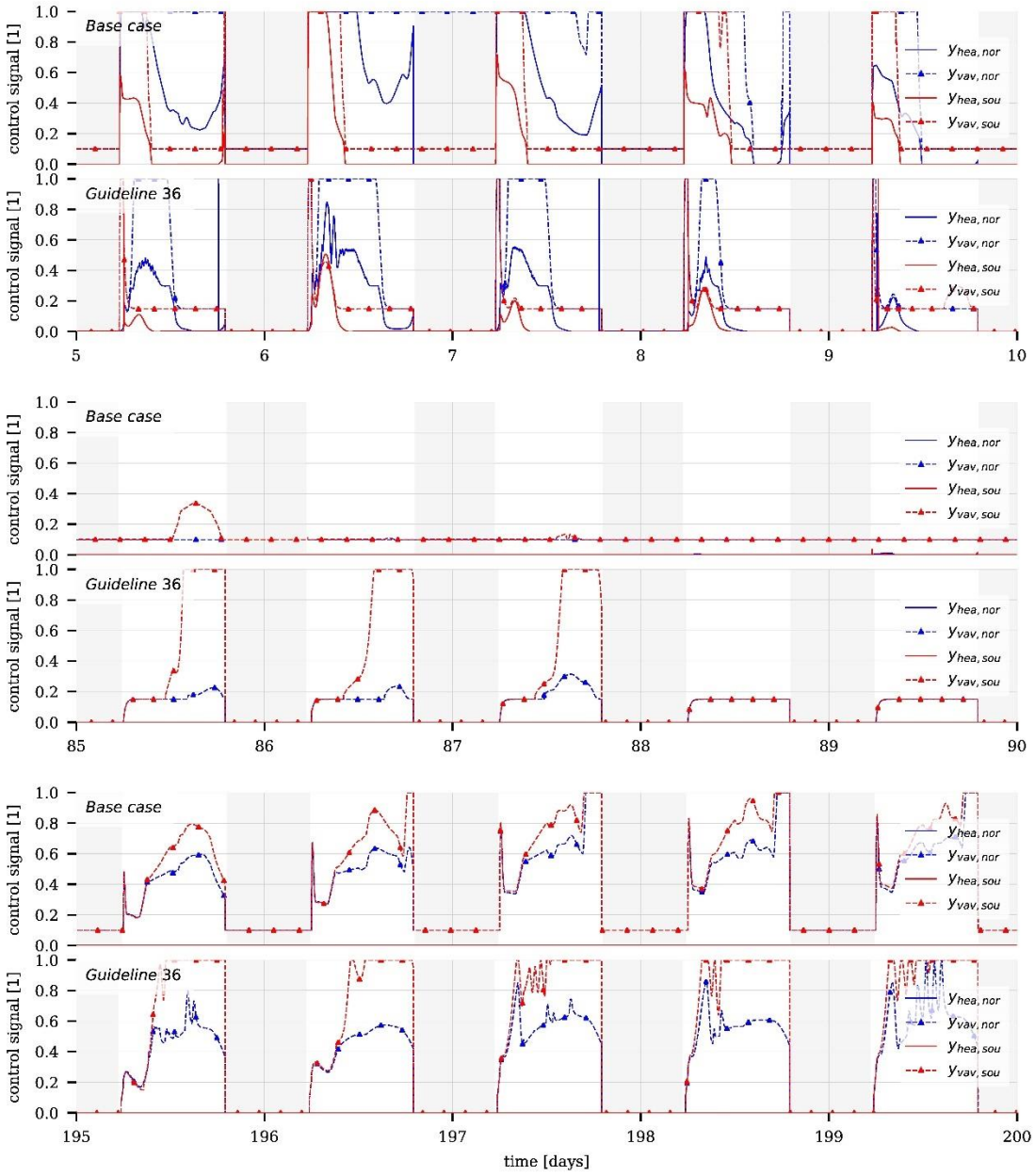


Fig. 8.11: *VAV control signals for the north and south zones. The white areas indicate the day-time operation.*

Fig. 8.11 shows the control signals of the reheat coils y_{hea} and the VAV damper y_{vav} for the north and south zones.

Fig. 8.12 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. 8.13 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. 8.14 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. 8.15 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. 8.8, their energy consumption is not noticeably affected.

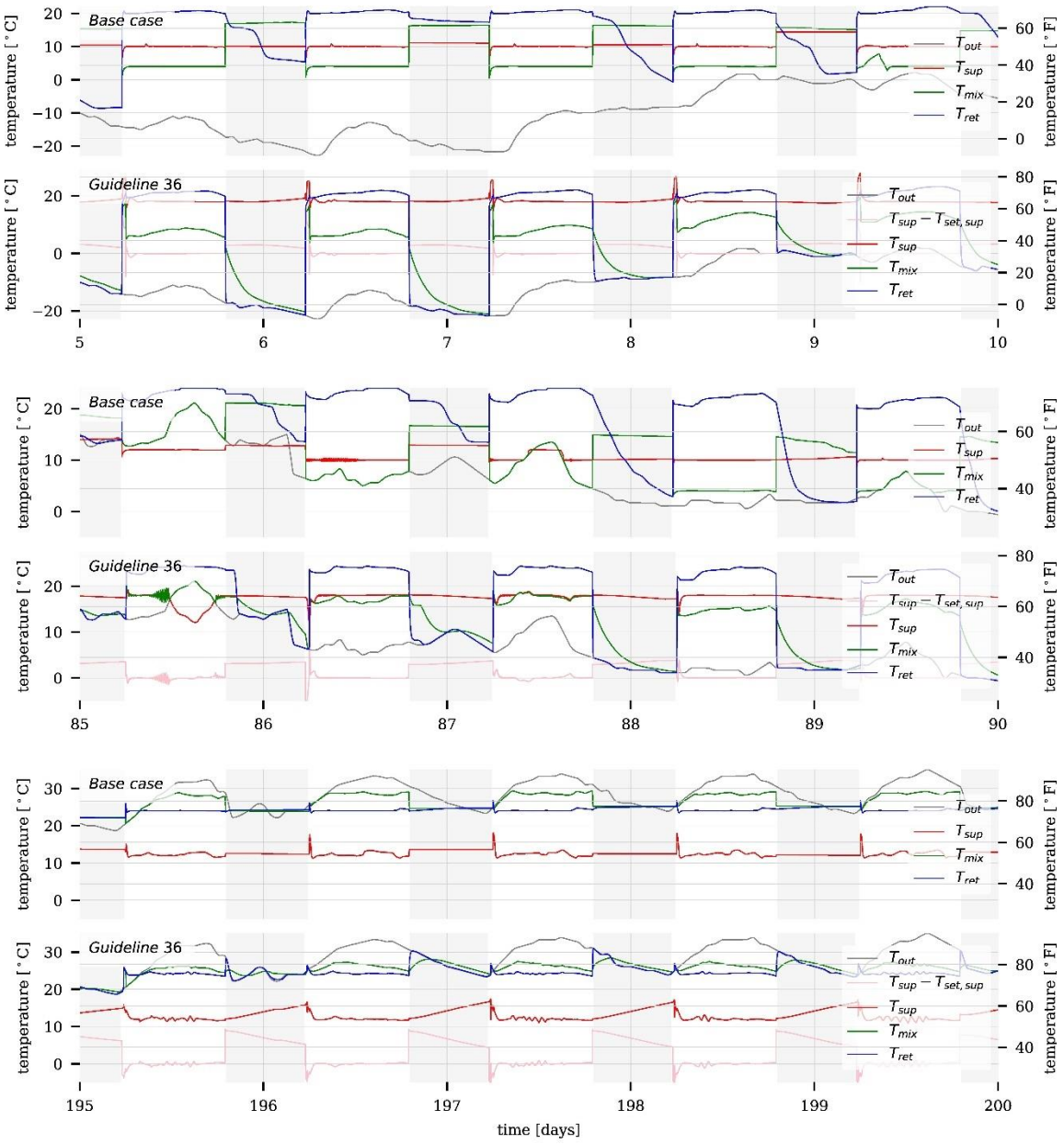


Fig. 8.12: *AHU temperatures.*

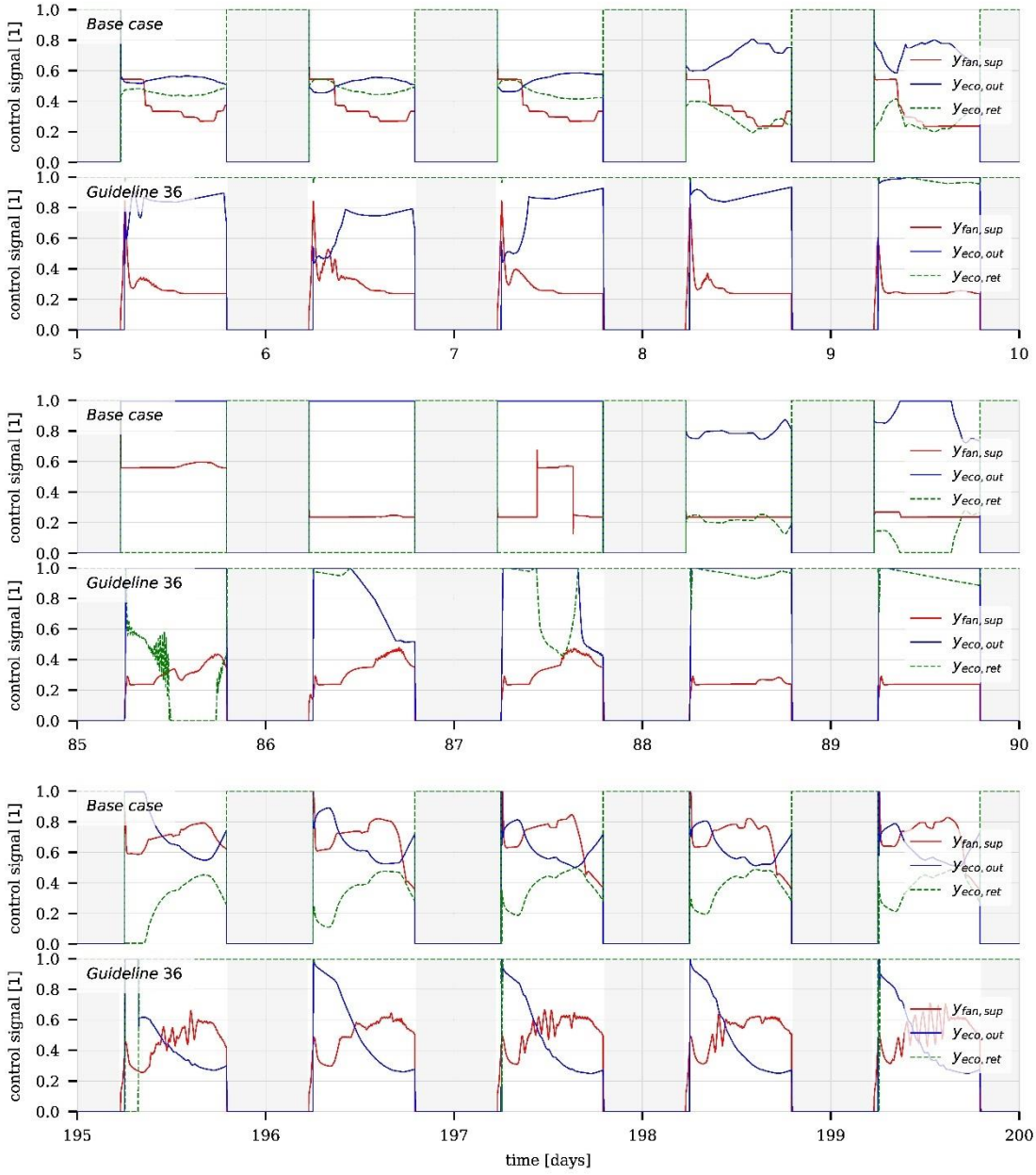


Fig. 8.13: *Control signals for the supply fan, outside air damper and return air damper.*

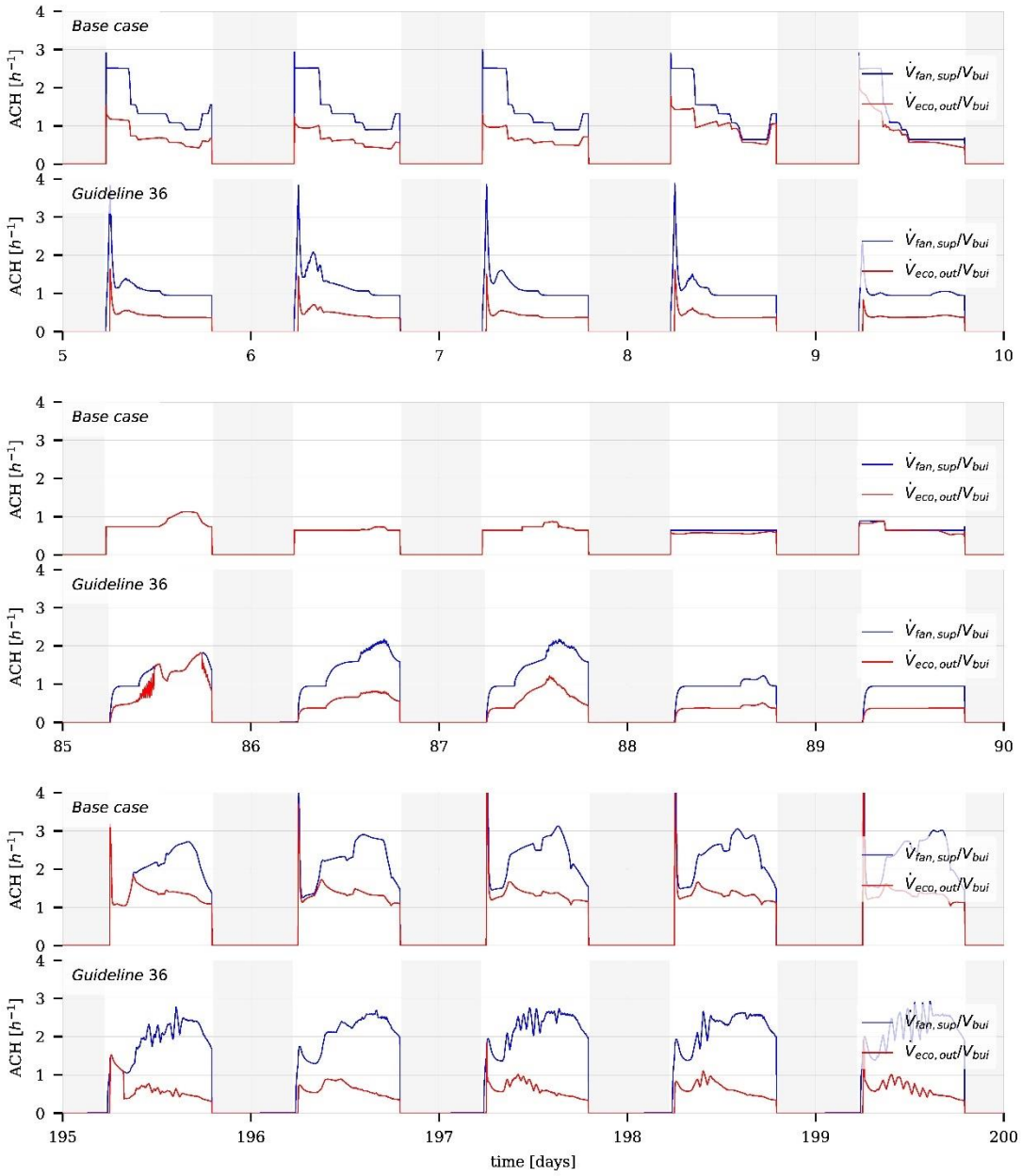


Fig. 8.14: *Fan and outside air volume flow rates, normalized by the room air volume.*

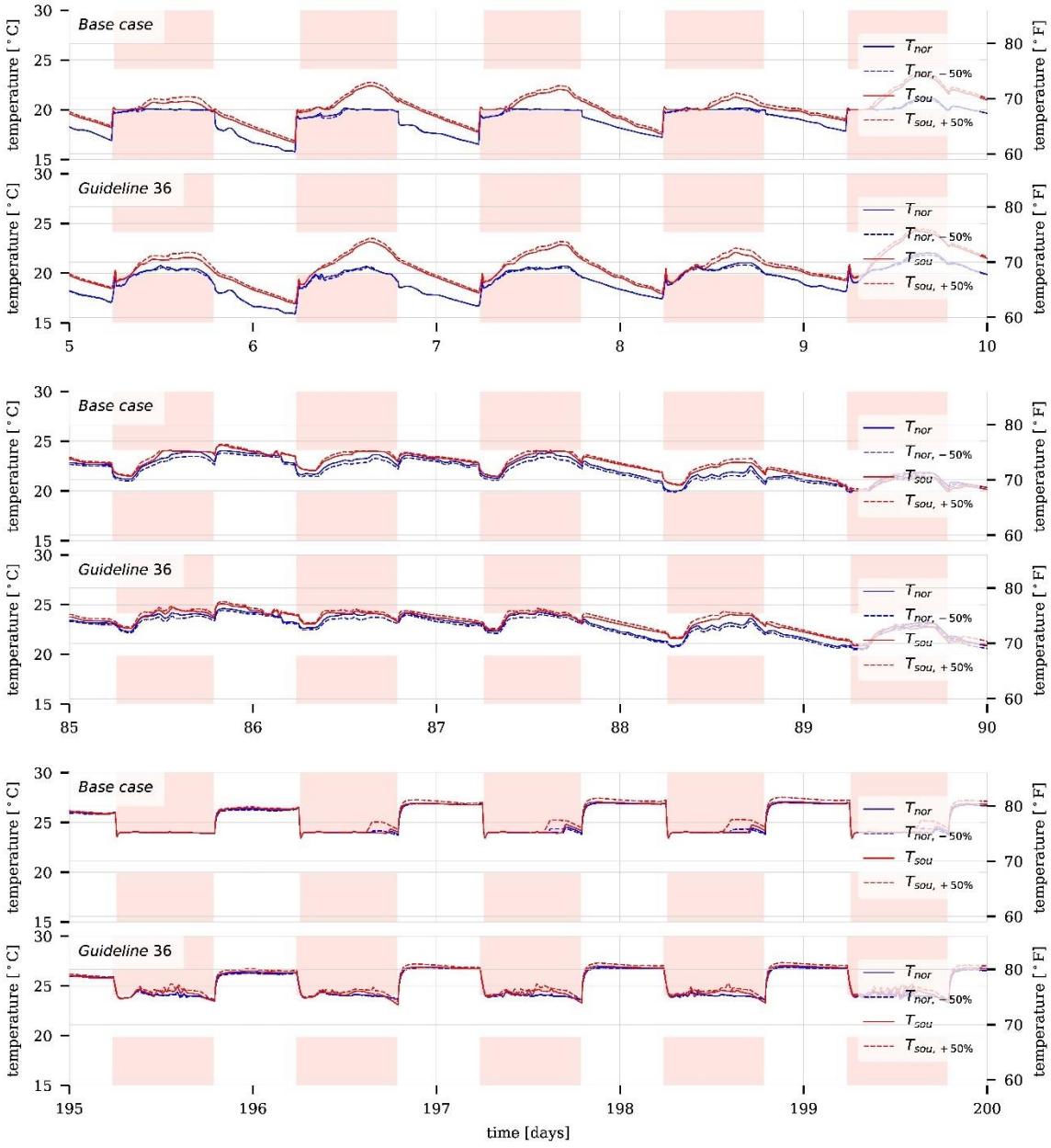


Fig. 8.15: *Outdoor air and room air temperatures for the north and south zone with equal internal loads, and with diversity added to the internal loads. The white area indicates the region between the heating and cooling setpoint temperatures.*

8.4 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].

8.4.1 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. 8.16 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. 8.16 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.

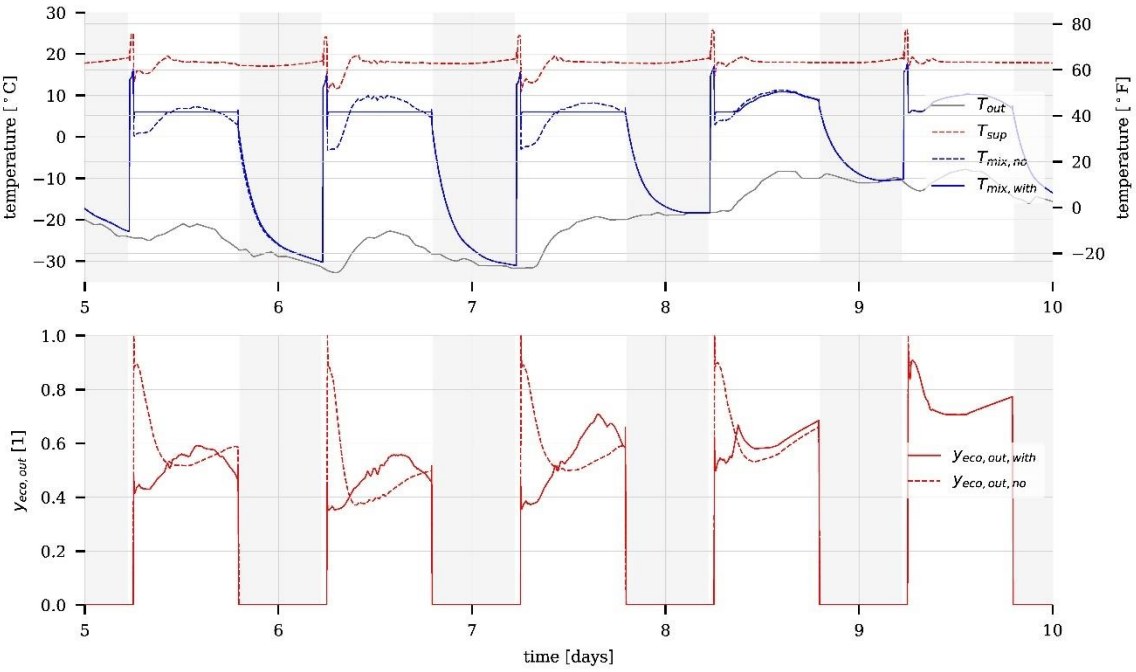


Fig. 8.16: *Mixed air temperature and economizer control signal for Guideline 36 case with and without freeze protection.*

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.

8.4.2 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. 8.5. 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.

8.4.3 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).

8.4.4 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.

8.4.5 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 http://simulationresearch.lbl.gov/modelica/userGuide/bestPractice.html

- *Fan with prescribed mass flow rate:* In earlier implementations, we converted the control signal for the fan to a mass flow rate, and used a fan model whose mass flow rate is equal to its control input, regardless of the pressure head. During start of the system, this caused a unrealistic large fan head of 4000 Pa (16 inch of water) because 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 presssure head:* We also tried to use a fan with prescribed pressure head. Similarly as above, the fan maintains the presssure head as obtained from its control signal, regardless of the volume flow rate. This caused unrealistic large flow rates in the return duct which has very small pressure drops. (Subsequently, we removed the return fan as it is not needed for this system.)

- *Time sampling certain physical calculations:* Dymola 2018FD01 uses the same time step for all continuous-time equations. Depending on the dynamics, this can be smaller than a second. Since some of the control samples every 2 minutes, it has shown to be favorable to also time sample the 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.

8.5 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_PR1*, we made a start on such an implementation and laid out the structure and conventions, but have not yet covered all of Guideline 36. Furthermore, conducting field validations would be useful too.

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

A benefit of the simulation based assessment was that it allowed detecting potential issues such as a mixed air temperature below the freezing point (Section 8.4.1) and chattering due to hard switches (Section 8.4.2). 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 9

Benefits to Rate Payers

9.1 Estimates of Potential Benefits

The adoption of the technology developed in the OpenBuildingControl project will result in improved design and implementation of building controls, resulting in more robust and efficient operation of commercial buildings. This will lead to more reliable provision of thermal and visual comfort while reducing both energy costs and equipment maintenance costs [FXK+17].

The energy savings from widespread adoption of the processes and tools can be estimated as follows. A major barrier to achieving the state's statutory energy goals is the failure of most commercial buildings to perform close to the technical potential of the design and its associated equipment. An LBNL meta-study identified 16% median actual savings from retro-commissioning [Mil11] and a study of 481 operational issues identified in existing commercial buildings found that control problems accounted for more than 75% of the potential energy savings [EFM+09]. Therefore, we assume that around 75% of the 16% expected energy savings associated with commissioning relate to controls, i.e. 12%. Assuming that the technologies being developed in the project can save 12% in the 50% of the commercial building floor area that is in buildings larger than 50,000 sf, the estimated savings average 6% across all commercial buildings.

We assume a value of 2.8 for the ratio of site energy to source energy for electricity, both for California and nationally.[1]

The California savings are estimated as follows: The annual energy consumption of California commercial buildings is about 67.1 TWh of electricity, equivalent to 0.64 quads (188 TWh) of source energy, and 1278.6 Mtherms (0.13 quads, 37.4 TWh) of natural gas (see Table 8-1 in [Com06]). The estimated 6% savings correspond to 4.03 TWh of electricity, equivalent to 0.038 quads (11.28 TWh) of source energy, and 0.00764 quads (2.24 TWh) of natural gas. Assuming a price of 0.17 $/kWh for electricity and 8 $/(1000 ft3) for natural gas (corresponding to 0.027 $/kWh), the cost savings would be $0.69B in electricity and $0.064B in natural gas.

The US national savings are estimated as follows: The annual energy consumption of US commercial buildings is about 1240 TWh of electricity, equivalent to 11.9 quads (3472 TWh) of source energy, and about 22,500 MTherms (2.25 quads, 659 TWh) of natural gas.[2] The estimated 6% savings correspond to 74.4 TWh of electricity, equivalent to 0.71 quads (208 TWh) of source energy, and 0.135 quads (39.5 TWh) of natural gas. Assuming a price of 0.11 $/kWh for electricity and 8 $/(1000 ft3) for natural gas, the electricity cost savings would be $8.2B and the natural gas savings $1.07B.

[1] See https://portfoliomanager.energystar.gov/pdf/reference/Source%20Energy.pdf

[2] See Table 1 in https://www.eia.gov/consumption/commercial/reports/2012/energyusage/.

These electricity savings correspond to 25 Rosenfelds in the US and 1.5 Rosenfelds in California.[3]

9.2 Timeframe and Assumptions for Estimated Benefits

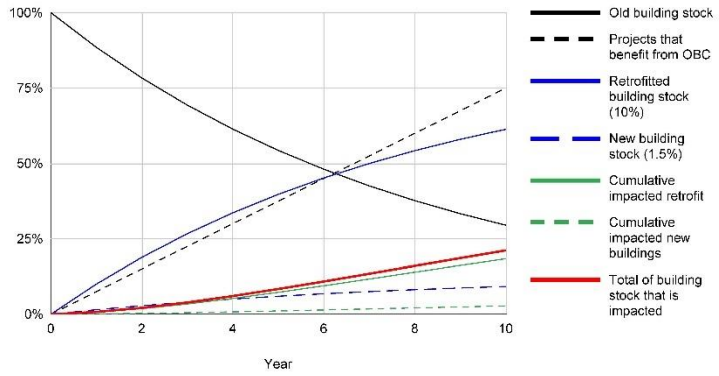


Fig. 9.1: *Estimated benefits of OpenBuildingControl.*

If we assume 75% adoption of OpenBuildingControl over the next ten years, a controls retrofit rate of 10% per year and a new building construction rate of 1.5% per year, then, after 10 years, the fraction of the building stock, weighted by floor area, that is impacted by OpenBuildingControl is 21% (see Fig. 9.1). Assuming the potential benefit of $0.69B savings in electricity, the estimated benefits are $146M/yr savings for California ratepayers.

[3] The unit of one Rosenfeld is equal to the electricity savings of 3 billion kWh per year, the amount needed to replace the annual generation of a 500 megawatt coal-fired power plant.

Chapter 10

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 the language that is used to express control sequences and requirements. It is a declarative language based on a subset of the Modelica language and specified in Section 4.

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

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

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

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

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

Functional Mockup Interface The Functional Mockup Interface (FMI) standard defines an open interface to be implemented by an executable called *Functional Mockup Unit* (FMU). The FMI functions are called by a simulator to create one or more instances of the FMU, called models, and to run these models, typically together with other models. An FMU may either be self-integrating (*co-simulation*) or require the simulator to perform the numerical integration (*model-exchange*). The first are sometimes called FMU-CS, while the second are called FMU-ME. See further http://fmi-standard.org/.

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

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

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

HVAC System Any HVAC plant coupled with the control system.

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

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

OBC See *Open Building Controls*.

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

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

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

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

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

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

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

has a state event when $x = 1$.

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 11

References

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

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

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

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

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

[BHK+02] Floyd E. Barwig, John M. House, Curtis J. Klaassen, Morteza M. Ardehali, and Theodore F. Smith. The national building controls information program. In *Summer Study on Energy Efficiency in Buildings*. Pacific Grove, CA, August 2002. ACEEE. URL: http://aceee.org/files/proceedings/2002/data/papers/SS02_Panel3_ Paper01.pdf.

[Com06] California Energy Commission. California commercial end-use survey. March 2006. URL: https://ww2.energy. ca.gov/2006publications/CEC-400-2006-005/CEC-400-2006-005.PDF.

[DFS+11] Michael Deru, Kristin Field, Daniel Studer, Kyle Benne, Brent Griffith, Paul Torcellini, Bing Liu, Mark Halver- son, Dave Winiarski, Michael Rosenberg, Mehry Yazdanian, Joe Huang, and Drury Crawley. U.S. De- partment 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.

[EFM+09] J. Effinger, H. Friedman, C. Morales, E. Sibley, and S. Tingey. A study on energy savings and measure cost effectiveness of existing building commissioning. Technical Report, IEA Annex 47, 2009.

[FXK+17] N. Fernandez, Y. Xie, S. Katipamula, M. Zhao, W. Wang, and C. Corbin. Impacts of commercial building controls on energy savings and peak load reduction. Technical Report 25985, PNNL, May 2017. URL: https: //buildingretuning.pnnl.gov/publications/PNNL-25985.pdf.

[FPA+19] Gabe Fierro, Marco Pritoni, Moustafa Abdelbaky, Daniel Lengyel, John Leyden, Anand Krishnan Prakash, Pranav Gupta, Paul Raftery, Therese Peffer, Greg Thomson, and David E. Culler. Mortar: an open testbed for portable building analytics. *ACM Transactions on Sensor Networks*, 16(1):7:1–7:31, 2019. doi:10.1145/3366375.

[GF17] Bill Gnerre and Kevin Fuller. When building controls veer off course. In *Facility Executive*, volume 462707, pages 32. Group C Media, Inc., Tinton Falls, NY, dec 2017. URL: https://facilityexecutive.com/2017/12/ building- automation-veers-off-course/.

[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.

[ME97] Sven Erik Mattsson and Hilding Elmqvist. Modelica – An international effort to design the next generation modeling language. In L. Boullart, M. Loccufier, and Sven Erik Mattsson, editors, *7th IFAC Symposium on*

Computer Aided Control Systems Design, 1–5. Gent, Belgium, April 1997. URL: http://www.modelica.org/publications/papers/CACSD97Modelica.pdf.

[Mil11] Evan Mills. Building commissioning: a golden opportunity for reducing energy costs and greenhouse gas emissions in the united states. *Energy Efficiency*, 4:145–173, 2011. URL: https://doi.org/10.1007/ s12053-011-9116-8, doi:10.1007/s12053-011-9116-8.

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

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

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

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

[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.

[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 Model-ica “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. De-velopment 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.