Control Description Language

Michael Wetter Milica Grahovac Jianjun Hu

Lawrence Berkeley National Laboratory
Energy Technologies Area
Building Technology and Urban Systems Division
Berkeley, CA, USA
{mwetter,mgrahovac,jianjunhu}@lbl.gov

Abstract

Properly designed and implemented building control sequences can significantly reduce energy consumption. However, there is currently no process with supporting tools that allows the assessment of the performance of different control sequences, export the control sequences in a vendor-neutral format for cost estimation and for implementation on a building automation system through machine-to-machine translation, and reuse the sequences for verification during commissioning.

This paper describes a Control Description Language (CDL) that we developed to create such a process. For CDL, we selected a subset of Modelica that allows a convenient representation of control sequences, simulation of the control sequence coupled to a building energy model, and development of translators from CDL to building automation systems. To aid in the development of such translators, we created a translator from CDL to a JSON intermediate format. In future work, we seek to work with building control providers to develop translators from CDL to commercial building automation systems.

Through a case study, we show that CDL suffices for simulation-based performance assessment of two ASHRAE-published control sequences for a variable air volume flow system of an office building. Moreover, the case study showed that merely due to differences in the control sequences, annual HVAC energy use was reduced by 30%. This difference is larger than the accuracy required when comparing different HVAC systems, thereby questioning the current practice of idealizing control sequences in building energy simulations, and demonstrating the importance of ensuring that the control sequence used during design simulations corresponds to the control sequence that will be implemented in the real building.

Keywords: controls, buildings, HVAC

1 Introduction

The building control industry has a standard for data communication called BACNet that is supported by all major control vendors (ASHRAE, 2004). However, there is no standard for expressing the control logic, despite the situation that control is often not implemented as specified during design, and the savings potential due to better control sequences is significant but not widely realized. The purpose of this paper is to describe a first implementation of a language with the intent to develop a standard for expressing building control sequences. This standard should support the mechanical designer in developing and testing control sequences within building energy simulations, and exporting these sequences to create unambiguous specifications for the control provider. It should support control providers in cost-estimation and in implementation of the control sequence on their control platform through machine-to-machine translation, and it should support the commissioning agent when verifying that the implemented control sequence meets the original specification.

It is generally recognized that properly designed and implemented control sequences can reduce energy consumption around 20% to 30% (Fernandez et al., 2017). Implementation errors in control sequences are in particular common in large buildings as they typically have built-up heating, ventilation and air-conditioning (HVAC) systems that require custom control sequences. The need for correct design and implementation of energy-saving and load-shifting control sequences is increasing because more stringent demands on energy savings and energy flexibility for the grid leads to increased complexity of control sequences.

For built-up HVAC systems, the current process is generally as follows: The mechanical engineer writes the control sequence in English language. This typically involves copying and adapting a part of the control sequence from similar projects. The document is then sent to a control provider. The control provider uses this English language description for cost estimation, and later for implementation. The control provider typically implements the control sequence by combining parts of sequences from previous projects that appear to have a similar controls intent. During commissioning, the commissioning agent conducts a limited number of tests to verify that the operation conforms to the commissioning agents' understanding of the control sequence.

The quality of the English language descriptions that are underlying this process varies largely. Our observation

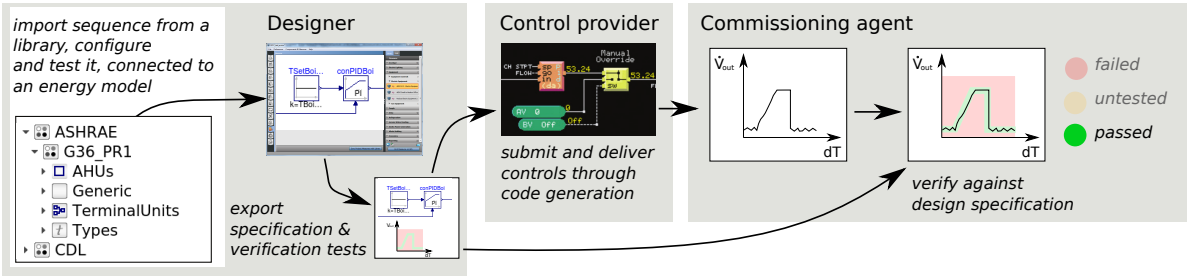

Figure 1. Overview of process for control sequence design, export of a specification, implementation on a control platform and verification against the specification.

is that best-in-class sequence specifications are often ambiguous, leave considerable room for interpretation, and may miss part of the sequence. This is not surprising because the control sequences are rather complex, because the English language representation aims to convey the control intent rather than how to realize it, and the mechanical engineer who wrote the control sequences is neither trained in, nor reimbursed for, the implementation of control sequences. As a consequence, such control specifications are not executable and hence do not allow for formal testing.

To realize the energy savings potential of building control sequences, we are working on developing tools and a process that will allow a mechanical engineer to select and adapt a control sequence from a library of sequences, connect it to a simulation model of the HVAC and building, test the performance of the control sequence coupled to the simulation model of the HVAC system and the building, export the control sequence in a control-vendor neutral format that allows control providers to conduct cost estimates and ultimately implement the sequence on their control platform through machine-to-machine translation. Figure 1 shows an overview of such a design flow.

To enable such a design flow, we are developing a language that we call Control Description Language (CDL), whose description is the main subject of this paper. We are also working on a project called "Spawn of EnergyPlus" that redesign EnergyPlus so that it supports this process (see `https://lbl-srg.github.io/soep/`).

CDL needs to satisfy these high level requirements:

- It must be independent of any control-vendor specific platform.
- It must be declarative to facilitate its translation to other languages.
- It must be possible to simulate controls expressed in the language within an annual building energy simulation.
- It must be deterministic, e.g., for given inputs and states, different implementations of sequences expressed must yield the same output and state updates (within the precision of ordinary differential equation solvers that may integrate PID controllers).
- It should be possible to translate the sequence to a variety of building control platforms.
- It must allow identification of cyclic graphs that would require iterative solutions and hence are not suited for implementation in building automation systems.

Related work in our application domain includes the following: Husaunndee et al. (1997) developed a MATLAB/Simulink-based toolbox of models of HVAC components and plants for the design and test of control systems called SIMBAD. SIMBAD has been used for testing and emulation of building control sequences, and is commercially distributed by CSTB France. Bonvini and Leva (2012) developed an industrial control library in Modelica that contains a variety of blocks with the intent to allow modelers to replicate industrial control sequences, including vendor-specific peculiarities. Yang et al. (2010) developed a tool chain that maps Simulink and Modelica models into an intermediate format, and then refined it for implementation in distributed controllers. Our approach borrows from their methodology in that we also use an intermediate format and restrict the language to make such a translation possible. Schneider et al. (2017) implemented a Modelica library with standardized control functions for building automation. They use control functions from VDI 3813-2:2011 and state graph representations from VDI 3814-6:2009. Our approach differs from their work as they document the control sequence using Unified Modeling Language (UML) class and activity diagrams. Also, they used semantic control connectors, which they subsequently removed for version 1.0.0. To generate English language representations together with a process diagram, Automated Logic Control developed CtrlSpecBuilder (ALC, 2018). CtrlSpecBuilder allows mechanical engineers to select the desired control functionality by answering a set of questions. The software then outputs a Microsoft Word document that specifies the control sequence in a vendor-neutral way together with a process diagram. Our approach differs from Bonvini and Leva (2012), Yang et al. (2010) and Schneider et al. (2017) in that we use elementary control blocks that form a basic library of control functions, and simple composition rules that we believe suffice for composing building control sequences.

As of this writing, we implemented CDL, used it

to implement control sequences that were developed by a project conducted for the American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE), tested these sequences in simulation, and developed an export program that converts the CDL representation to a JSON and an English language representation. Work in progress and not reported in this paper includes the use of CDL to compare a simulated versus an actual building control system response. Also ongoing are discussions with control providers to see if they can prototype a translator from CDL to their commercial product line. Based on this feedback, the CDL language may further evolve.

2 Discussion of the Target Platform

To put this work in context, one has to recognize that building control systems largely vary in how they implement control sequences. Commercial products range from textual languages that combine the functionality of FORTRAN with programming structures similar to BASIC (Siemens, 2000) to graphical block-diagram languages where blocks can be used from a library and new blocks can be provided in a component-oriented programming language similar to Java or C# (Thomas, 2016). Furthermore, different building control systems use different native data types; some allow boolean signals to take on `true` or `false` only, while others also allow the value of `null`. Also, control sequences often contain proprietary algorithms, such as the computation of the start time for a warm-up after a room temperature setback. Furthermore, specification for the programming languages are hard if not impossible to find for many systems. Thus, the space of target platforms to which our language will need to be translated is heterogeneous and often proprietary. We therefore only intent to translate CDL to building automation systems, but do not attempt to translate a particular control implementation to CDL.

Control providers typically also include blocks for communication with hardware, and for sending messages to the operator, such as through logging, sending email, or displaying a value in an operator workstation. For example, Contemporary Controls' Sedona platform contains a block called `CControls_BASR8M_Platform` that advises the programmer how much usable memory is available for application programming, a block to monitor the execution time of a Sedona logic (`ScanTim`) and blocks to communicate with BACNet or with web pages (Contemporary Controls, 2017). CDL does not attempt to support such specialized blocks. Rather, the intention of CDL is to support the declaration of the control logic in a vendor neutral way. This is also required because during the design of a building, the control provider may not yet be known and thus the specification should be independent of any control product line. Code that provides input/output functionality with hardware or web services will need to be added when a CDL-conformant control sequence specification is implemented on a particular control platform.

3 CDL Language

We will now describe the Control Description Language (CDL). To develop CDL, we identified a small subset of basic control functionalities that will need to be provided, together with rules that prescribe how to compose sequences and rules that prescribe the mathematical behavior of these basic control functionalities and composite sequences. Specifically, we formulated CDL as a block diagram language that consists of the following elements:

- Permissible data types.
- Elementary control blocks, each of which encapsulates an elementary calculation performed on a signal in a control sequence, such as a block that adds two signals and outputs the sum.
- Input and output connectors through which these blocks receive values and send values.
- Syntax to specify
  - how to instantiate control blocks and assign values to 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, and
  - how to specify composite blocks.
- A model of computation that describes when blocks are executed and when outputs are assigned to inputs.

The following sections further explain these elements.

3.1 Syntax

In order to use CDL with building energy simulation programs, and to not invent yet another language with a new syntax, we selected a subset of the Modelica 3.3 specification for the implementation of CDL (Modelica Association, 2012). The selected subset is needed to instantiate classes, assign parameters, connect objects and document classes. This subset is fully compatible with Modelica, e.g., no other information that violates the Modelica Standard has been added, thereby allowing users to view, modify and simulate CDL-conformant control sequences with any Modelica-compliant simulation environment.

To simplify the support of CDL for tools and control systems, the following Modelica keywords are not supported in CDL: `extends`, `redeclare` and `constrainedby`, `inner` and `outer`.

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

1. Clocks, as the use of clocks would complicate translation to building automation systems that often distribute the control sequences to different field devices.
2. `algorithm` sections, because the elementary building blocks are black-box models as far as CDL

is concerned and thus there is no need to support `algorithm` sections.

3. `initial equation` and `initial algorithm` sections, because these are not needed when composing sequences using the elementary building blocks explained in Section 3.4.

3.2 Permissible Data Types

The basic data types are, in addition to the elementary building blocks, parameters of type `Real`, `Integer`, `Boolean`, `String`, and `enumeration`. All specifications in CDL are declarations of blocks, instances of blocks, or declarations of type `parameter`, `constant`, or `enumeration`. Variables are not allowed.[1] The declaration of such types is identical to the declaration in Modelica.

Each of these data types, including the elementary building blocks, can be a single instance or one-dimensional array. Array indices shall be of type `Integer` only. The first element of an array has index `1`. An array of size `0` is an empty array. `enumeration` or `Boolean` data types are not permitted as array indices.

3.3 Encapsulation of Functionality

All computations are encapsulated in a `block`. Blocks expose parameters, and they expose inputs and outputs using connectors.

Blocks are either *elementary building blocks* (see Section 3.4) or *composite blocks* (see Section 3.9).

3.4 Elementary Building Blocks

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

$$(p,t,u(t),x(t)) \mapsto (y(t),x'(t)). \qquad (1)$$

Control providers who support CDL need to be able to implement the same functionality as is provided by the elementary CDL blocks. CDL implementations are allowed to use a different implementation of the elementary building blocks, because the implementation is language specific. However, implementations shall have the same inputs, outputs and parameters, and they shall compute the same response for the same value of inputs and state variables.

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

The elementary building blocks are implemented in subpackages of the package `CDL`. For each elementary building block, there is an example that demonstrates its use.

[1] Variables are used in the elementary building blocks, but these can only be used as inputs to other blocks if they are declared as an output.

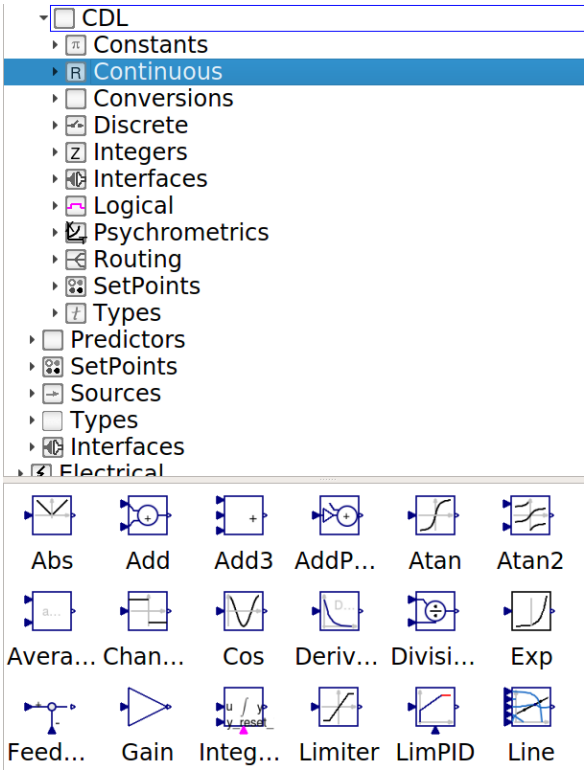

Figure 2. Screenshot of CDL library.

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 ... [omitted]
 </p>
 </html>"));
end AddParameter;
```

3.5 Instantiation

The instantiation of blocks is identical to Modelica. In the assignment of `parameters`, calculations are allowed. For example, a hysteresis block could be configured as follows

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

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

Instances can conditionally be removed by using an `if` clause. This allows, for instance, to have a single implementation of an economizer enable/disable control sequence that can be configured to optionally take the specific enthalpy as an input signal. An example code snippet is

```
parameter Boolean use_enthalpy = true
 "Set to true to evaluate outdoor air
    enthalpy in addition to temperature"
    ;
CDL.Interfaces.RealInput hOut
  if use_enthalpy
  "Outdoor air enthalpy";
```

3.6 Connectors

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

3.7 Connections

Connections connect input to output connectors. For scalar connectors, each input connector of a block needs to be connected to exactly one output connector of a block. For vectorized connectors, each (element of an) input connector needs to be connected to exactly one (element of an) output connector. Vectorized input connectors can be connected to vectorized output connectors using one connection statement, provided that they have the same number of elements.

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

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

where `annotation(...)` is used to declare the graphical rendering of the connection (see Section 3.8). The order of the connections and the order of the arguments in the `connect` statement does not matter.

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

3.8 Annotations

Annotations follow the same rules as described in the following sections of the Modelica 3.3 Specification:

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

Hence, for CDL, annotations are primarily used to graphically visualize block layouts and input and output signal connections, and to declare vendor annotations (see § 18.1 in Modelica 3.3 Specification).

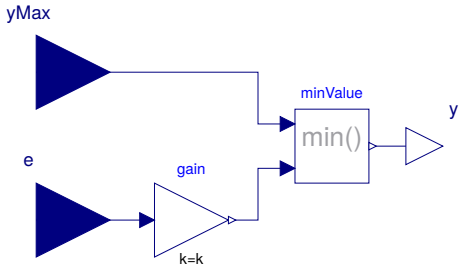

Figure 3. Example of a composite control block that outputs $y = \min(k\,e, y_{max})$, where k is a parameter.

3.9 Composite Blocks

CDL allows building composite blocks such as shown in Figure 3. Composite blocks are needed to preserve grouping of control blocks and their connections, and are needed for hierarchical composition of control sequences.

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. Appendix A shows how to declare the block shown in Figure 3.

3.10 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 § 8.4, and is as follows:

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.
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.

3.11 Inferred Properties

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.

Note that 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.

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.

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 use of display unit 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 Control Sequence Implementation

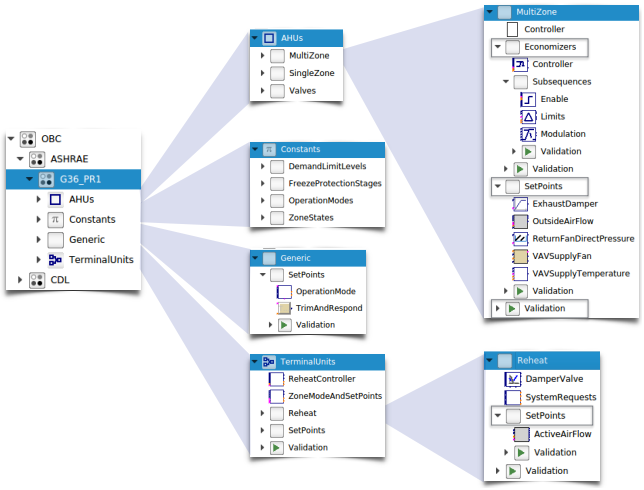

Figure 4. Overview of the ASHRAE Guideline 36 package implemented in the Modelica Buildings library 5.0.0

To test CDL, we used it to implement control sequences for variable air volume flow systems as specified in ASHRAE Guideline 36, public review draft 1 (ASHRAE, 2016). Figure 4 shows an overview of the package structure. The implementation is structured hierarchically into packages for air handler units, into constants that indicate operation modes, into generic sequences such as for a trim and respond logic, and into sequences for terminal units. For every sequence, there is a validation package that illustrates its use.

For implementation of these sequences, we had to make the following main design decisions:

For the PID controller, we used the same implementation as is used in the Modelica Buildings library. This implementation is identical to the one from the Modelica Standard Library, except that it adds an option to reset the control output when a boolean input switches to `true`. This controller is in the standard form

$$y(t) = k\left(e(t) + \frac{1}{T_i}\int e(s)ds + T_d\frac{de(t)}{dt}\right), \qquad (2)$$

where we omitted for simplicity features of the implemented controller such as anti-windup, and where $y(t)$ is the control signal, $e(t) = u_s(t) - u_m(t)$ is the control error, with $u_s(t)$ being the set point and $u_m(t)$ being the measured quantity, k is the gain, T_i is the time constant of the integral term and T_d is the time constant of the derivative term. Note that the units of k are the inverse of the units of the control error, while the units of T_i and T_d are seconds.

As the units of flow rates and pressure can vary between orders of magnitude, for example depending on whether cfm, m^3/s or m^3/h are used for flow measurements, we decided to normalize the control error as follows: For temperatures, no normalization is used, and the units of k are 1/Kelvin. No normalization is used because 1 Kelvin is 1.8 Fahrenheit, and hence these are of the same order of magnitude. For air flow rate control, the design flow rate is used to normalize the control error, and hence k is unitless. This also allows for using the same control gain for flows of different magnitudes, for example for a VAV box of a large and a small room, provided the rooms have similar transient response. For pressure control, the pressure difference is used to normalize the control error, and hence k is unitless.

Guideline 36 is specific as to where a P or a PI controller should be used. We used these recommendations as the default control configuration. However, all controllers can be configured as P, PI or PID controller. This allows for example to temporarily configure a PI controller as a P controller during the tuning process.

As Guideline 36 is written to convey the control intent rather than the actual implementation, it does not discuss how to avoid chattering of control due to sensor noise or numerical integration error. Therefore, for the part of the control sequences that use continuous time semantics, we added either hysteresis blocks or timers wherever the control or measurement signal is used as an input to a switch.[2]

5 Export of Control Sequences

We are currently developing a parser that exports CDL-conformant control sequences for the following use cases:

1. For human-readable documentation, the parser converts the sequences to html, similar to how Modelica tools generate html documentation.

[2]During the initial testing of the sequences, we indeed observed chattering and non-convergence of the solver as we missed a few of these switches.

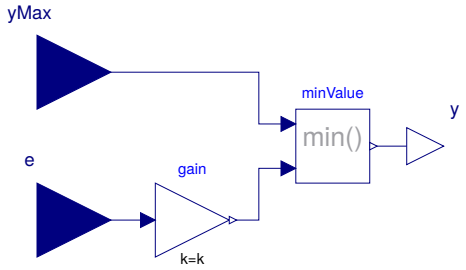

Figure 5. Graphical rendering of a the composite control block shown in Appendix A.

2. For further translation to control product lines, the parser converts the sequences to two JSON formats: One is an intermediate format that is close to the abstract syntax of Modelica, the other is generated by simplifying the former for easier processing by downstream applications. The latter representation is also used to generate html documentation of the control sequence.

The parser is currently being developed at `https://github.com/lbl-srg/modelica-json`. As an illustrative example, consider the composite block shown graphically in Figure 5 and textually using the CDL in Appendix A. The parser can export this specification to the JSON format shown in Appendix B.

The parser is implemented using ANTLR (ANother Tool for Language Recognition, `http://www.antlr.org/`) which converts CDL to a JSON format, which then is further simplified using JavaScript. For the html output, we use the mustache templating engine (`https://github.com/janl/mustache.js`).

6 Case Study

To test the suitability of CDL for simulation, we conducted closed loop simulations of multizone VAV sequences coupled to a whole building energy model. We will now summarize the experiment, and refer the reader for a more detailed description to Wetter et al. (2018) and `http://obc.lbl.gov/`. For the simulations, we used a model of one floor of the new construction medium office building for Chicago, IL, as described in the set of DOE Commercial Building Benchmarks (Deru et al., 2011). For all simulations, we used the same building and the same variable air volume flow system, but with two different control sequences. One sequence is based on the above described ASHRAE Guideline 36, whereas the other is the control sequence VAV 2A2-21232 of the Sequences of Operation for Common HVAC Systems (ASHRAE, 2006). All models are available in the Modelica Buildings library (Wetter et al., 2014), version 5.0.0, in the package `Buildings.Examples.VAVReheat`.

It turns out that changing the control sequence from VAV 2A2-21232 to the one published in Guideline 36 saves around 30% annual site HVAC energy under comparable thermal comfort. These are significant savings that can be achieved through software only, without the need for additional hardware or equipment. Moreover, the magnitude of these savings also questions how controls are typically represented in building energy simulation programs. Building energy simulation programs typically use idealized control sequences. These programs may then be used to compare the energy performance of different HVAC systems, such as a VAV system versus a radiant cooling system. However, such differences frequently are also in the order of 30%. Thus, to compare the energy performance of HVAC systems, control sequences must be represented adequately in the simulation, and the authors question the validity of the control idealizations that are commonly used in building energy simulation. If the variability due to controls is in the order of 30%, one cannot discern what apparent savings can be attributed to the change in HVAC system. Moreover, a process is needed that ensures that the control sequences will be implemented correctly and thus savings identified during design are realized during operation.

7 Conclusion

With the implementation of the Guideline 36 sequences and the case study, we have shown that our subset of the Modelica language that we identified for CDL suffices to implement control sequences for simulation. Ongoing work attempts to put in place a translator to a commercially available building automation system to see if unexpected issues arise that may require changes to CDL.

Our case study indicated that annual HVAC energy use can be reduced by 30% simply through the use of more sophisticated conventional control sequences. These sequences are however more complicated to specify and implement, and therefore we believe that for their proper use in design and actual operation of buildings, a process that allows their use in design, their (semi-automatic) translation to control product lines, and their verification relative to design specification is essential.

A key language issue that we selected to not support in the first version of CDL are state machines, for example as implemented in the `Modelica.StateGraph` package of the Modelica Standard Library 3.2 or as described in the Chapter 17 in the Modelica Language Definition (Modelica Association, 2012). The use of state machines would have made the implementation of control sequences considerably easier for blocks whose output is computed using various time delays, interlocks and modes of operation, which are frequently used in Guideline 36. As state machines are not universally supported in building automation systems, and as there are various flavors of state machines, we decided to currently not support them.

Selecting a subset of Modelica, in particular not supporting replaceable classes and multiple inheritance, considerably simplified the development of a translator from CDL to JSON. We believe that this also makes it easier for

control providers to support CDL.

At present, CDL supports conventional control sequences. In the future, CDL will also need to support control sequences that use Model Predictive Control or other advanced mathematical methods. How to provide blocks that can interface with such methods, or how to add vendor-specific packages that provide such advanced methods that are typically proprietary will be subject of future work.

8 Acknowledgment

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

References

ALC, 2018. CtrlSpecBuilder, 2018. URL https://www.ctrlspecbuilder.com.

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

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

ASHRAE, 2016. *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.

Marco Bonvini and Alberto Leva. A modelica library for industrial control systems. In *Proc. of the 9-th Int. Modelica Conf.*, pages 477–484, Munich, Germany, September 2012. Modelica Association. doi:DOI:10.3384/ecp12076477.

Contemporary Controls, 2017. *Sedona Open Control – Reference Manual*. Contemporary Controls, September 2017. URL https://www.ccontrols.com/pdf/RM-SEDONA00.pdf.

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

Nicholas E.P. Fernandez, Srinivas Katipamula, Weimin Wang, YuLong Xie, Mingjie Zhao, and Charles D. Corbin. Impacts of commercial building controls on energy savings and peak load reduction. Technical Report 25985, PNNL, 5 2017.

A. Husaunndee, R. Lahrech, H. Vaezi-Nejad, and J.C. Visier. Simbad: A simulation toolbox for the design and test of HVAC control systems. In Jean Jacques Roux and Monika Woloszyn, editors, *Proc. of the 5-th IBPSA Conf.*, pages 269–276, 1997. URL www.ibpsa.org/proceedings/bs1997/bs97_p022.pdf.

Modelica Association, 2012. *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.

Georg Ferdinand Schneider, Georg Ambrosius Peßler, and Simone Steiger. Modelling and simulation of standardised control functions from building automation. In *Proc. of the 12-th Int. Modelica Conf.*, pages 209–218, Prague, Czech Republic, may 2017. Modelica Association. doi:DOI:10.3384/ecp17132209.

Siemens, 2000. *APOGEE Powers Process Control Language (PPCL) User's Manual*. Siemens Building Technologies, October 2000. URL https://www.quia.com/files/quia/users/hpiracer/AIRC65/PPCL_Users_Manual.

George Thomas. *Creating an Open Controller with Sedona Framework*™. Contemporary Controls, February 2016. URL https://sedona-alliance.org/pdf/WPSEDONAAA0.pdf.

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.

Michael Wetter, Jianjun Hu, Milica Grahovac, Brent Eubanks, and Philip Haves. OpenBuildingControl: Modeling feedback control as a step towards formal design, specification, deployment and verification of building control sequences. In *To appear in: 2018 Building Performance Modeling Conference and SimBuild*, September 2018.

Y. Yang, A. Pinto, A. Sangiovanni-Vincentelli, and Q. Zhu. A design flow for building automation and control systems. In *2010 31st IEEE Real-Time Systems Symposium*, pages 105–116, November 2010. doi:10.1109/RTSS.2010.26.

Appendix A

The following statement, when saved as `CustomPWithLimiter.mo`, is the declaration of the composite block shown in Figure 3

```
block CustomPWithLimiter
  "Custom implementation of a P controller with variable output limiter"
  parameter Real k "Constant gain";
  CDL.Interfaces.RealInput yMax "Maximum value of output signal"
    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}));
  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;
```

Appendix B

The JSON representation of the composite control block shown in Figure 5 is as follows, where we omitted the graphical annotations to keep the listing short.

```
[
 {
  "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",
      "comment": "Outputs the minimum of its inputs"
     }
    ]
   },
   "info": "<html>[omitted for briefity]</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" }
    ]
   ]
  }
]
```