LinkageJS Requirements Specification

(Working Draft)

Oct 17, 2019

Contents

Chapter 1

Preamble

This documentation specifies the requirements for LinkageJS software.

It is still work in progress, intended to be used as a discussion basis.

The proposed design should not be considered finalized and will evolve as the development progresses.

Chapter 2

Process Workflow

To be updated.

Chapter 3

Requirements

Todo: Conventions refer to RFC 2119 [RFC2119].

Look into https://jsonapi.org/

How to extract CDL part (to be translated) from whole system model?

- Specification with the configuration widget of "unbreakable" parts of the control sequence.
- Also need to 1) expand arrays into scalar instances and connections 2) suppress bus gateway variables `connect(v1, bus.v); connect(v2, bus.v)` into `connect(v1, v2)` unless supported by CDL spec. & translator?

Devise how return status signals can be generated for components that do not have built-in output status variables.

- Typically the pump model: difficult to implement without state event, otherwise short circuit the input control signal potentially with a `pre` operator.

Allow for top level controller XOR individual controllers.

Specify more the tagging process and how it can support mapping with equipment characteristics and sizing from data & sizing sheets.

- EIKON uses two concepts: *reference* (used programmatically) and *display* (used for UI only) name.
- HVAC zone, floor, room name is enough?

3.1 General Description

3.1.1 Main Requirements

The software is primarily a graphical user interface for editing Modelica models in a diagrammatic form: see Section 3.2 and Section 3.3.

Built around this core functionality the following additional features are required:

1. A configuration widget supporting assisted modeling based on a simple HTML input form: see Section 3.4
2. A schematics export functionality: see Section 3.5
3. A set of functionalities to enable working with tagged variables: see Section 3.6

In terms of software design:

- The software must rely on client side JS code with minimal dependencies and must be built down to a single page HTML document (SPA).
- A widget structure is required that allows seamless embedding into:
  - a desktop app – with standard access to the local file system,
  - a standalone web app – with access to the local file system limited to Download & Upload functions of the web browser (potentially with an additional sandbox file system to secure backup in case the app enters an unknown state),
  - any third party application with the suitable framework to serve a single page HTML document executing JS code – with access to the local file system through the API of the third party application:
    - the primary target is OpenStudio® (OS),
    - an example of a JS application embedded in OS is FloorspaceJS. The standalone SPA lives here: https://nrel.github.io/floorspace.js. FloorspaceJS can be considered as a reference for the development.

  Note: Those three integration targets are actual deliverables.

- A Python or Ruby API is required to access the data model and leverage the main functionalities of the software in a programmatic way e.g. by OpenStudio measures.

3.1.2 Software Compatibility

The software requirements regarding platform and environment compatibility are presented in Table 3.1.

Table 3.1: Requirements for software compatibility

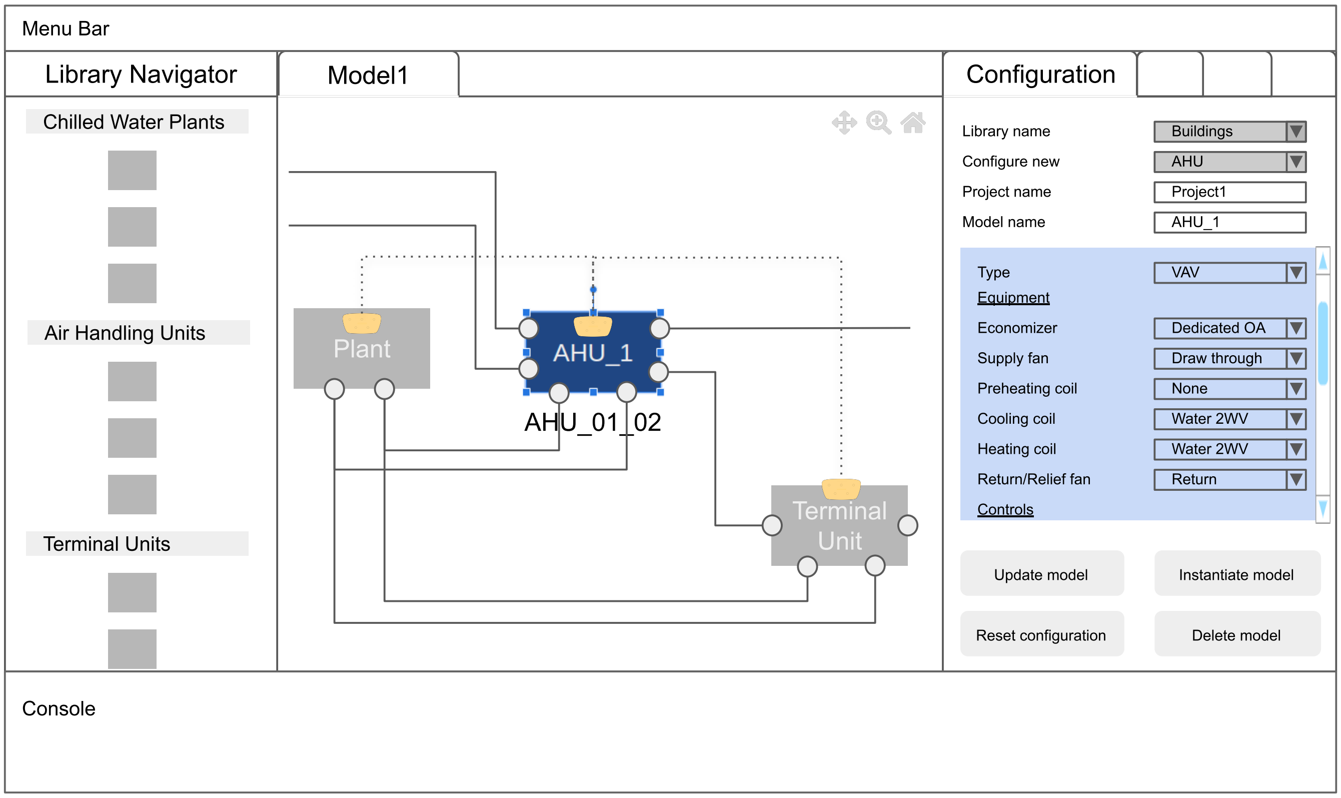
Feature	Support
Platform (minimum version)	Windows (10), Linux Ubuntu (18.04), OS X (10.10)
Mobile device & responsive design ?	iOS, Android?
Web browser	Chrome, Firefox, Safari

3.1.3 UI Visual Structure

A minimal mockup of the UI is presented Fig. 3.1.

The minimal requirements are:

- Left panel: library navigator
- Main panel: model editor with diagram, icon, documentation or code view
- Right panel:
  - Configuration tab, see Section 3.4
  - Connections tab, see Section 3.4.3.4
  - Parameters tab, see Section 3.4.3.5



Library name	Buildings
Configure new	AHU
Project name	Project1
Model name	AHU_1
Type	VAV
Economizer	Dedicated OA
Supply fan	Draw through
Preheating coil	None
Cooling coil	Water 2WV
Heating coil	Water 2WV
Return/Relief fan	Return

Fig. 3.1: UI Visual Structure

- Menu bar
- Bottom panel: console

The placement of the different UI elements might be different than the one proposed here above but the user must have access to all those elements. Ideally a toggle feature should be implemented to show or hide each side panel, either by user click if the panel is pinned or automatically. Optionally a fully customizable workspace may be implemented.

3.2 Detailed Functionalities

Table 3.2: Functionalities of the software – R: required, P: required partially, O: optional, N: not required

Feature	V0	V1	Comment
Main functionalities			(as per Section 3.1)
Diagram editor for Modelica models	R		See detailed requirements below.
Configuration widget	P	R	An alpha version of the widget is required in V0 for testing and refining the requirements.
Schematics export	N	R	
Working with tagged variables	N	R	
I/O			
Load `mo` file	P	R	To be updated cf. different integration targets Simple Modelica model or full package (V0). If the model contains annotations specific to the configuration widget (see Section 3.4), the corresponding data are loaded in memory for further configuration. If the model contains the Modelica annotation `uses` the corresponding library is loaded. If a package is loaded the structure of the package and sub packages is checked against *Chapter 13 Packages* (V1).
Export *mo* file	R		"Total model" export option?
Export simulation results	R		Export in the following format: `mat, csv`. All variables or selection based on variables browser (see below).
Variables browser	P	R	Query selection of model variables based on regular expression (V0) or Brick/Haystack tag [Bri] [Hay] (V1)
Plot simulation results	N	O	
Export control points summary	R		Relies on LBL module to generate the list of A/B I/O variables.
Export schematics	P	R	Only the equipment drawing in V0. Control points and SOO description in V1 see Fig. 3.16. Relies on LBL module CDL to Word translator.

Continued on next page

Table 3.2 – continued from previous page

Feature	V0	V1	Comment
Import/Export data sheet?	P	R	Additional module to 1) generate a file in CSV or JSON format from the configuration data (V0) 2) populate the configuration data based on a file input in CSV or JSON format (V1).
Modelica features			
Checking the compliance with Modelica standard	P	R	Real-time checking of syntax (V0) and connection (V1).
Translate model	P		The software settings allow the user to specify a command for translating the model with a third party Modelica tool e.g. JModelica. The output of the translation routine is logged in LinkageJS console.
Simulate model	P		The software settings allow the user to specify a command for simulating the model with a third party Modelica ool .g. JModelica. The output of the simulation routine is logged in LinkageJS console.
Automatic medium propagation between connected components	P	P	Partially supported because only the configuration widget integrates that feature. When generating `connect` equation manually a similar approach as the *fluid path* used by the configuration widget could be developed, see components with 4 ports and 2 medium. Expected as a future enhancement of Modelica standard[1]: should we anticipate or wait and see?
Support of Modelica graphical annotations	R		
Code editor	P	R	Raw text editor (V0) with linter and Modelica specification check upon save (V1)
Icon editor	O	R	Editing functionalities similar to diagram editor
Documentation view	R		
Library version management	O	R	If a loaded model contains the Modelica annotation `uses` e.g. `uses(Buildings(version="6.0.0")` the software checks the version number of the stored library, prompts the user for update if the version number does not match, executes the conversion script per user request.
Object manipulation			
Vectorized instances	R		An array dimension descriptor appending the name of an object is interpreted as an array declaration. Further connections to the connectors of that object must comply with the array structure.
Expandable connectors	R		

Continued on next page

Table 3.2 – continued from previous page

Feature	V0	V1	Comment
Navigation in object composition	R		Right clicking an icon in the diagram view offers the option to open the model in another tab
Multiple objects selection for input of common parameters	O	R	If several objects are selected only their common parameters are listed in the Parameters panel. If a parameter alue s modified, all the selected objects will have their parameter value change.
Avoiding duplicate names	R		When instantiating a component, if the default name is already used in the model the software automatically appends he name with the lowest integer value that would ensure uniqueness. When copying and pasting a set of objects connected together, the set of connect equations is updated to ensure consistency with the appended object names.
Graphical features			A user experience similar to modern web based diagramming applications is expected e.g. draw.io.
Tab view	R		The diagram view is organized in tabs that can be manipulated, created and deleted typically as navigation tabs n a eb browser.
Diagram split view	N	R	The diagram view can be split (horizontally and vertically) into several views. Each tab can be dragged and dropped from one view to another. The views are synchronized so that if the same model is open in different views and gets modified, all the views of the model are updated to reflect the modifications.
Copy/Paste objects	R		Copying and pasting a set of objects connected together copies the objects declarations and the corresponding connect equations.
Pan and zoom on mouse actions	R		
Undo/Redo	R		
Draw shape, text box	O	R	
Start connection line when hovering connectors	O	R	
Connection line jumps	O	R	Gap jump at crossing
Customize connection lines	O	R	Color, width and line can be specified in the annotations panel
Hover information	R		Class path when hovering an object in the diagram view and tooltip help for each GUI element
Color and style of connection lines	P	R	Allow the user to manually specify (right click menu) the style of the connections lines (V0). When generating a `connect` equation automatically select a line style based on some heuristic to be further specified (V1).

Continued on next page

Table 3.2 – continued from previous page

Feature	V0	V1	Comment
Fancy connection lines?	N	O	Gridified layout https://ialab.it.monash.edu/webcola/examples/dotpowergraph.html Orthogonal edge route layout https://www.visual-paradigm.com/support/documents/vpuserguide/1283/28/ 6047_automaticdia.html
Miscellaneous			
Choice of units SI / IP	?	?	

3.3 Modelica Graphical User Interface

The software must comply with the Modelica language specification [Mod17] for every aspect relating to (the chapter numbers refer to [Mod17]):

- validating the syntax of the user inputs: see *Chapter 2 Lexical Structure* and *Chapter 3 Operators and Expressions*,
- the connection between objects: see *Chapter 9 Connectors and Connections*,
- the structure of packages: see *Chapter 13 Packages*,
- the annotations: see *Chapter 18 Annotations*.

Note: When drawing a connection line between two connector icons in the diagram view:

- a `connect` equation with the references to the two connectors must be created,
- with a graphical annotation defining the connection path as an array of points and providing an optional smoothing function e.g. Bezier.
- When no smoothing function is specified the connection path must be rendered graphically as a set of segments.
- The array of points must be either:
  - created fully automatically when the next user's click after having started a connection is made on a connector icon. The function call `create_new_path(connector1, connector2)` creates the minimum number of *vertical or horizontal* segments to link the two connector icons with the constraint of avoiding overlaying any instantiated object,
  - created semi automatically based on the input points corresponding to the user clicks outside any connector icon: the function call `create_new_path(point[i], point[i+1])` is called to generate the path linking each pair of points together.
- The first and last couple of points must be so that the connection line does not overlap the component icon but rather grows the distance to it, see Fig. 3.2.

3.4 Configuration Widget

[1] From https://build.openmodelica.org/Documentation/Modelica.Fluid.UsersGuide.ComponentDefinition.FluidConnectors.html

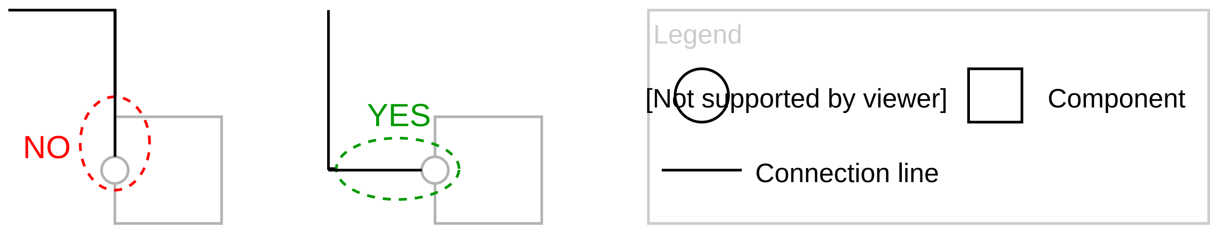


Fig. 3.2: Connection line drawing logic near a connector

3.4.1 Functionalities

The configuration widget allows the user to generate a Modelica model of an HVAC system and its controls by filling up a simple input form. It is mostly needed for integrating advanced control sequences that can have dozens of I/O variables. The intent is to reduce the complexity to the mere definition of the system's layout and the selection of standard control sequences already transcribed in Modelica [LBNL19]. CtrlSpecBuilder is a tool widely used in the HVAC controls industry, which typically provides the same kind of functionality.

There are fundamental requirements regarding the Modelica model generated by the configuration widget:

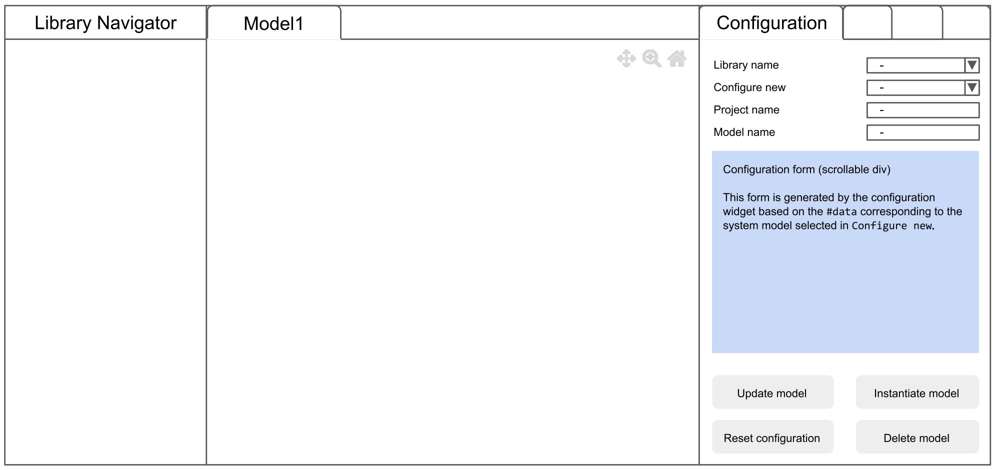
1. It must be "graphically readable" (both within LinkageJS and within any third-party Modelica GUI e.g. Dymola): this is a strong constraint regarding the placement of the composing objects and the connections that must be generated automatically.
2. It must be ready to simulate: no additional modeling work or parameters setting is needed outside the configuration widget.
3. It must contain all annotations needed to regenerate the HTML input form when loaded, with all entries corresponding to the actual state of the model.
   - Manual modifications of the Modelica model made by the user are not supported by the configuration widget: an additional annotation should be included in the Modelica file to flag that the model has deviated from the template. In this case the configuration widget is disabled when loading that model.
4. The implementation of control sequences must comply with OpenBuildingControl requirements, see *§7 Control Description Language* and *§8 Code Generation* in [LBNL19]. Especially:
   - It is required that the CDL part of the model can be programmatically isolated from the rest of the model in order to be translated into vendor-specific code (by means of a third-party translator).
   - The expandable connectors (control bus) are not part of CDL specification: **how to represent communication between sub-systems?**

The input form is provided by the template developer (e.g. LBL) in a data model with a format that is to be further specified in collaboration with the software developer.

The data model typically provides for each entry:

- the HTML widget and populating data to be used for requesting user input,
- the modeling data required to instantiate, position and set up the parameters of the different components,
- some tags to be used to automatically generate the connections between the different components connectors.

The user interface logic is illustrated in figures Fig. 3.3 and Fig. 3.4.



Configuration form (scrollable div)

This form is generated by the configuration widget based on the `#data` corresponding to the system model selected in `Configure new`.

`Update model`
Implement the configuration options specified by the user in the `Configuration form` by instantiating and connecting objects in the file `#library/User/#project/#model.`

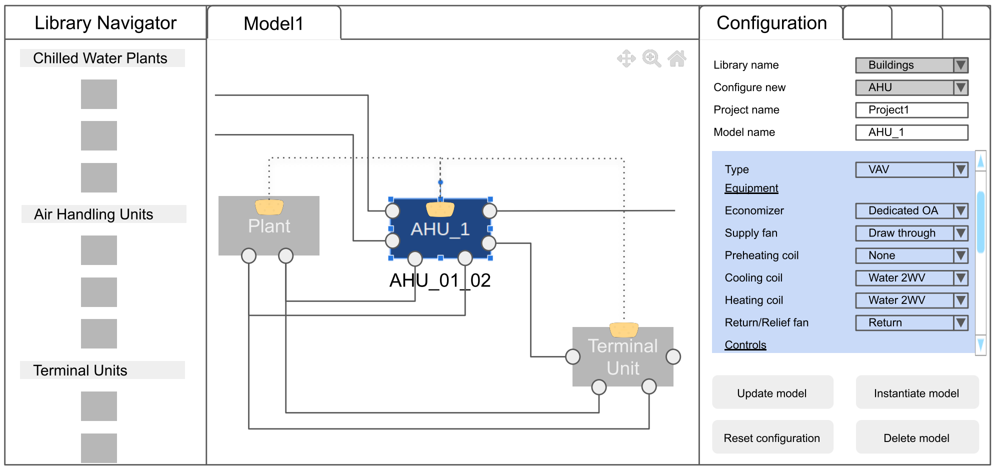
`Instantiate model`
Update and instantiate the class `#library/User/#project/#model` in the model loaded in the active tab of the main panel (here `Model1`).

`Reset configuration`
Reset the configuration options to the default values.

`Delete model`
Delete the class `#library/User/#project/#model` and all its instances in the model loaded in the active tab of the main panel (here `Model1`).

When no object is selected in the diagram view this is the default view for the `Configuration` panel.
The `Library name` is the last value selected (further referenced as `#library`). The drop down menu allows selecting between loaded libraries. The `Library name` is used to 1) load the configuration data stored in `#library/Configuration` directory, 2) define the root path of the directory where the built models will be saved i.e. `#library/User/*/`.
The `Configure new` drop down menu allows selecting the type of system model to configure. The menu is populated by `#data/#system.value` for all configuration data files in `#library/Configuration`.
The `Project name` is the last value entered (further referenced as `#project`). A real-time form test is required to validate the user input against syntax requirements and avoid duplicate in `#library/User`. The path of the directory where the built models will be saved is `#library/User/#project`.
The `Model name` is by default `#data/#name.value` (further referred to as `#model`). It can be modified by the user (call a `rename_class` function if the model has already been saved). A real-time form test is required to validate the user input against syntax requirements and avoid duplicate in `#library/User/#project`.

Fig. 3.3: Configuration widget – Configuring a new model



Library name	Buildings
Configure new	AHU
Project name	Project1
Model name	AHU_1
Type	VAV
Equipment	
Economizer	Dedicated OA
Supply fan	Draw through
Preheating coil	None
Cooling coil	Water 2WV
Heating coil	Water 2WV
Return/Relief fan	Return
Controls	

`Update model`
Implement the configuration options specified by the user in the `Configuration form` by instantiating and connecting objects in the file `#library/User/#project/#model.`

`Instantiate model`
Update and instantiate the class `#library/User/#project/#model` in the model loaded in the active tab of the main panel (here `Model1`).

`Reset configuration`
Reset the configuration options to the default values.

`Delete model`
Delete the class `#library/User/#project/#model` and all its instances in the model loaded in the active tab of the main panel (here `Model1`).

This is the view for the `Configuration` panel if:
- one object is selected in the main panel,
- and the corresponding class contains a model annotation `__Linkage_data(...)` providing the configuration data in a JSON-serialized format (further referred to as `#data`).

The `Configuration` panel is populated with the values from `#data`.
The `Library name` and `Configure new` fields are locked.
The `Project name` can be modified: when clicking `Update model` this will call a `move_class` function.
The `Model name` can be modified: when clicking `Update model` this will call a `rename_class` function.
All configuration options can be modified: when clicking `Update model` this will update the class `#library/User/#project/#model`.

Fig. 3.4: Configuration widget – Configuring an existing model

The envisioned data structure supporting this logic is illustrated in Listing 5.1 (pseudo code) where:

- the placement coordinates are provided relatively to a simplified grid, see Fig. 3.5 – those are to be mapped to Modelica diagram coordinates by the widget,
- the components referenced under the `equipment` name are connected together with fluid connectors, see Section 3.4.2,
- the components referenced under the `controls` name are connected together with signal connectors, see Section 3.4.3,
- the components referenced under the `dependencies` name are part of the equipment section:
  - they typically correspond to sensors and outside fluid connectors,
  - the model completeness depends on their presence,
  - the requirements for their presence can be deduced from the equipment and controls options,
  - they do not need additional fields in the user form of the configuration widget.
- the equipment and controls models are connected together by means of a *control bus*, see Fig. 3.16: the upper-level model including the equipment and controls models is the ultimate output of the configuration widget (see Fig. 3.4 where the component named `AHU_1_01_02` represents an instance of the upper-level model `AHU_1` generated by the widget). That component exposes the outside fluid connectors as well as the top level control bus.

The logic for instantiating classes from the library is straightforward. Each field of the form specifies:

- the path of the class to be instantiated depending on the user input;
- the position of the component in simplified grid coordinates to be converted in diagram view coordinates.

The next paragraphs address how the connections between the connectors of the different components are generated automatically based on this initial model structure.

Note: Test/issue

- Headered VS dedicated chilled water pump: conditional number of instances, placement and fluid path. Backup strategy: the first dedicated pump can be instantiated in the equipment section, the others in the dependencies section.
- A `RelativePressure` sensor requires the specification of two derived paths which is cumbersome since the fluid component around which the differential pressure is sensed belongs to a fluid path which depends on the sensor option e.g. AFMS (main path) or differential pressure (derived path). Backup strategy: considering an additional `junction` tag or specifying a tagging logic to determine if the parent fluid path gets interrupted or not at each fork. . .

Best format

- JSON
  - Expensive syntax especially for boolean conditions or auto-referencing the data structure: is there any standard syntax?
  - Is a JSON schema needed to eventually validate the user inputs? In that case the template developer would have to write the boolean conditions twice with two different syntaxes: once in the template and once in the JSON schema (typically with the standard syntax `if then else` introduced in *Draft 7*)?
- Specific format to be defined in collaboration with the UI developer and depending on the selected UI framework
  A robust syntax is required for:
  - auto-referencing the data structure e.g. `#type.value` refers to the value of the field `value` of the object which `$id` is `type`,

  - conditional statements: potentially every field might require a conditional statement – either data fields (e.g. the model to be instantiated and its placement) or UI fields (e.g. the condition to enable a widget itself or the different options of a menu widget).
  - Ideally the syntax should also allow iteration `for` loops to instantiate a given number (as parameter) of objects with an offset applied to the placement coordinates e.g. chiller plant with `n` chillers. Backup strategy: define all (e.g. 10) possible instances and enable only the first `n` ones based on a condition.

Reference guideline for controls specification

- Providing a reference guideline for controls specification conditionally disables all controls options that do not comply with that guideline.

Parameters exposed by the configuration widget

- The template developer is free to integrate in the template any parameter of the composing components e.g. `V_flowSup_nominal` and reference them in the model declaration e.g. `Buildings.Fluid.Movers.SpeedControlled_y(m_flow_nominal=(#air_supply.medium).rho_default / 3600 * #V_flowSup_nominal.value).` The configuration widget must replace the referenced names by their actual values (literal or numerical). The user will be able to override those values in the parameters panel e.g. if he wants to specify a different nominal air flow rate for the heating or cooling coil.
- Some parameters *need* to be integrated in the template (examples are provided in reference to `Buildings.Controls.OBC.ASHRAE.G36_PR1.AHUs.MultiZone.VAV.Controller`):
  - when they impact the model structure e.g. `use_enthalpy` requires an additional enthalpy sensor,
  - when they impact the dimension or instanciation of some connectors e.g. `numZon`, `have_occSen`,
  - when no default value is provided e.g. `AFlo` cf. requirement that the model generated by the configuration widget must be ready to simulate.

  In the first two cases the model declaration must use the `final` qualifier for the corresponding parameters to prevent the user from overriding those values in the parameters panel.

For each object, the fields are defined as follows. When the type of a field is specified as a string marked with (C) it may correspond to:

- a conditional statement provided as a string that must be interpreted by the UI engine,
- a reference to another field value of type boolean (that may itself correspond to a conditional statement provided as a string).
- The syntax supporting this feature shall be specified in collaboration with the UI developer.

Note: The syntax must support e.g. `(#air_supply.medium).rho_default` where the first dot is used to access the property `medium` of the object with `$id == #air_supply` (which must be replaced by its value) while the second dot is used to access Modelica property `rho_default` of the class `Medium` (which must be kept literal).

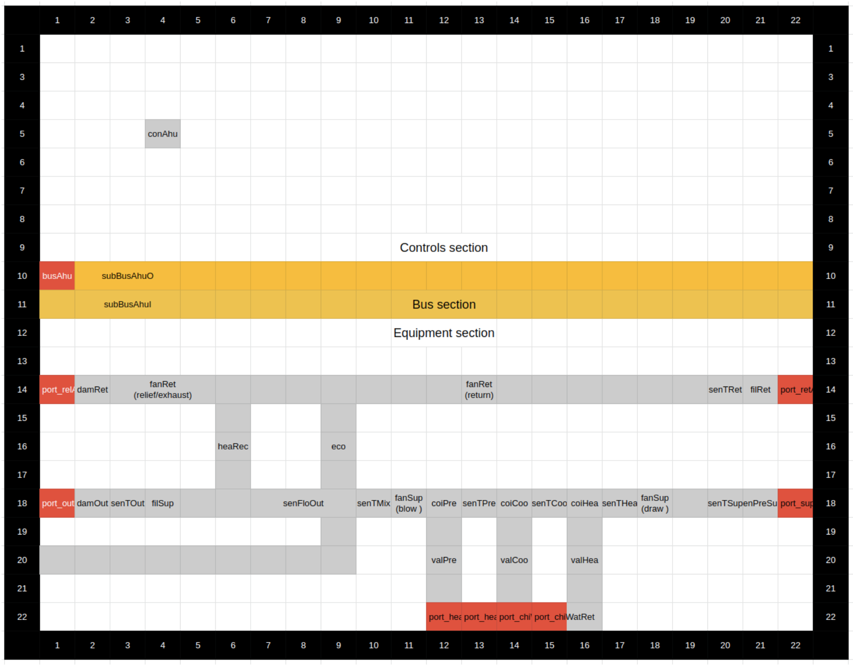
Configuration object definition

`system` : string : required

System to configure e.g. air handling unit, chilled water plant.

`icon` : string : required

Path to icon file.



Equipment		Sensors		Controls		Connectors	
damOut	AO Damper	senTSup	Supply temperatu	conAhu	AHU top level	port_outAir	Outdoor air
filSup	Filter Supply	senTOut	Outdoor air temperature			port_supAir	Supply air
heaRec	Heat Recove	senTMix	Mixed air temperature			port_retAir	Return air
eco	Economizer	senTPre	coiPre outlet temperature			port_relAir	Relief air
fanSup	Supply Fan	senTHea	coiHea outlet temperature			port_heaWatS	Heating water supply
coiPre	Preheating C	senTCoo	coiCoo outlet temperature			port_heaWatR	Heating water return
coiCoo	Cooling Coil	senPreDuc	Duct static pressure			port_chiWatS	Chilled water supply
coiHea	Heating Coil	senFloOut	OA flow rate			port_chiWatR	Chilled water return
filRet	Filter Return					busAhu	Control bus
fanRet	Return Fan					**private**	
damRet	RA Damper					subBusAhuI	Sub bus A/BI
valPre	coiPre valve					subBusAhuO	Sub bus A/BO
valHea	coiHea valve						
valCoo	coiCoo valve						

Fig. 3.5: Simplified grid providing placement coordinates for all objects to be instantiated when configuring an AHU model

`diagram` : object : required

Object defined as follows.

`configuration` : array : required

items : integer

Array on length 2, providing the number of lines and columns of the simplified grid layout.

`model` : array : required

items : array

items : integer

Array on length 2 providing the coordinates of one corner of the diagram rectangular layout.

`name` : object : required

Name of the component. Must be stored in the Modelica annotation `defaultComponentName`. Object defined as *elementary object*.

required : `[$id, description, widget, value]`

`type` : object : required

Type of system e.g. for an air handling unit: variable air volume or dedicated outdoor air. Object defined as *elementary object*.

required : `[$id, description, widget, value]`

`fluid_paths` : array : required

items : object

Object defined as follows.

`$id` : string : required

Unique string identifier starting with `#`.

`direction` : string : required

enum : `["north", "south", "east", "west"]`

Direction indicating the order in which the components must be connected along the path.

`medium` : string : required

Common medium for that fluid path and all derived paths e.g. `"Buildings.Media.Air"`

`equipment` : array : optional

items : object

Object defined as *elementary object*.

`controls` : array : optional

items : object

Object defined as *elementary object*.

`dependencies` : array : optional

items : object

Object defined as *elementary object*.

Elementary object definition

`$id` : string : required

Unique string identifier starting with `#`.
Used for referencing the object properties e.g. `id_value.property`.
If the object has a `declare` field, the name of the declared component is the value of `$id` without the dash character.
Must be suffixed with brackets e.g. `[2]` in case of array variables.

`description` : string : required

Descriptive string.
If the object has a `declaration` field, the descriptive string appends the component declaration in the Modelica source file (referred to as *comment* in *§4.4.1 Syntax and Examples of Component Declarations* of [Mod17]).

`enabled` : boolean, string (C) : optional, default `true`

Indicates if the object must be used or not. If not, the UI does not display the corresponding widget, no modification to the model is done and the object field `value` gets assigned its **default value or null?**.

`widget` : object : optional

Object defined as follows.

`type` : string : required

Type of UI widget.

`options` : array : optional

items : string

Options to be displayed by certain widgets e.g. dropdown menu.

`options.enabled` : array : optional

items : boolean, string (C)

Indicates which option can be selected by the user. Must be the same size as `widget.options`.

`value` : string (C), number, boolean, null : required

[*enum* : `widget.options` (if provided)]

Value of the object (default value prior to user input).

May be provided as a literal expression in which all literal references to object properties (prefixed with `#`) must be replaced by their numerical value.

`unit` : string : optional

Unit of the value. Must be displayed in the UI.

`declaration` : array, string (C), null : optional

[*items* : string (C)]

Any valid Modelica declaration (component or parameter) or an array of those which must have the same size as `widget.options` if the latter is provided (in which case the elements of `declaration` get mapped with the elements of `widget.options` based on their indices).

Note: If one option requires multiple declarations, the first one will typically be specified here and the other ones as dependencies.

`annotation` : array, string (C), null : optional

[*items* : string (C)]

Any valid Modelica annotation or an array of those which must have the same size as `widget.options` if the latter is provided (in which case the elements of `annotation` get mapped with the elements of `widget.options` based on their indices).

`protected` : boolean : optional, default `false`

Indicates if the declaration should be public or protected.

`icon_transformation` : string (C) : optional

Graphical transformation that must be applied to the component icon e.g. `"flipHorizontal"`.

`placement` : array, string (C) : optional

[*items* : array, integer]

[*items* : integer]

Placement of the component icon provided in simplified grid coordinates `[line, column]` to be mapped with the model diagram coordinates.

Can be an array of arrays where the main array must have the same size as `widget.options` if the latter is provided (in which case the elements of `placement` get mapped with the elements of `widget.options` based on their indices).

`connect` : object : optional

Data required to generate the connect equations involving the connectors of the component, see Section 3.4.2.

Object defined as follows.

`type` : string : optional, default `path`

enum : `["path", "tags", "explicit"]`

Type of connection logic.

`value` : string (C), object : required

If `type == "path"`: fluid path (string) that must be used to generate the tags in case of two connectors only. It must not be used if the component has more than two connectors or a non standard connectors scheme (different from one instance of `Modelica.Fluid.Interfaces.FluidPort_a` and one instance of `Modelica.Fluid.Interfaces.FluidPort_b`).

If `type == "tags"`: object providing for each connector (referenced by its instance name) the tag to be applied.

If `type == "explicit"`: object providing for each connector (referenced by instance name) the connector to be connected to, using explicit names e.g. `fanSup.port_a`.

An example of the resulting data structure is provided in annex, see Section 5.1.

3.4.2 Fluid Connectors

The fluid connections (`connect` equations involving two fluid connectors) must be generated based on either:

- an explicit connection logic relying on one-to-one relationships between connectors (see Section 3.4.2.1) or,

- a heuristic connection logic (see Section 3.4.2.2) based on:
  - the coordinates of the components in the diagram layout i.e. after converting the coordinates provided relatively to the simplified grid,
  - a tag applied to the fluid connectors (or fluid ports) of the components.

3.4.2.1 Explicit Connection Logic

That logic is activated at the component level by the keyword `connect.type == "explicit"`.

The user provides for each connector the name of the component instance and connector instance to be connected to e.g. `"port_1": "component1.connector2`.

3.4.2.2 Heuristic Connection Logic

That logic relies on connectors tagging which supports two modes:

1. Default mode (`connect.type == "path"`)
   - By default the instances of `Modelica.Fluid.Interfaces.FluidPort_a` and `Modelica.Fluid.Interfaces.FluidPort_b` will be suffixed `inlet` and `outlet` respectively.
   - The prefix tag is provided at the component level to specify the fluid path e.g. `air_supply` or `air_return`.
   - The fluid connectors are then tagged by concatenating the previous strings e.g. `air_supply_inlet` or `air_return_outlet`.
2. Detailed mode (`connect.type == "tags"`)
   - We need an additional mechanism to allow tagging each fluid port individually. Typically for a three way valve, the bypass port should be on a different fluid path than the inlet and outlet ports see Fig. 3.6. Hence we need a mapping dictionary at the connector level which, if provided, takes precedence on the default logic specified above.
   - Furthermore a fluid connector can be connected to more than one other fluid connector (fork configuration). To support that feature the concept of *derived path* is introduced: if `fluid_path` is the name of a fluid path, each fluid path named `/^fluid_path_((?!_).)*$/gm` is considered a *derived path*. The original (derived from) path is the *parent path*. A path with no parent path is referred to as *main path*.
   - For instance in case of a three way valve without any flow splitter to explicitly model the fluid junction, the mapping dictionary could be:
     `{"port_1": "hotwater_return_inlet", "port_2": "hotwater_return_outlet", "port_3": "hotwater_supply_bypass_inlet"}` where `hotwater_supply_bypass` is a derived path from `hotwater_supply`.
3. Explicit mode (`connect.type == "explicit"`)
   - Eventually it may be more convenient in certain cases to specify explicitly a one-to-one connection scheme between the connectors of the model e.g. a differential pressure sensor to be connected with the outlet port of a fan model and a port of a fluid source providing the reference pressure.

The conversion script throws an exception if an instantiated class has some fluid ports that cannot be tagged nor connected with the previous logic e.g. non default names and no (or incomplete) mapping dictionary provided.

If the tagging is resolved for all fluid connectors of the instantiated objects, the connector tags are stored in a hierarchical vendor annotation at the model level e.g. `__Linkage(Connect(tags="{object_name1:`

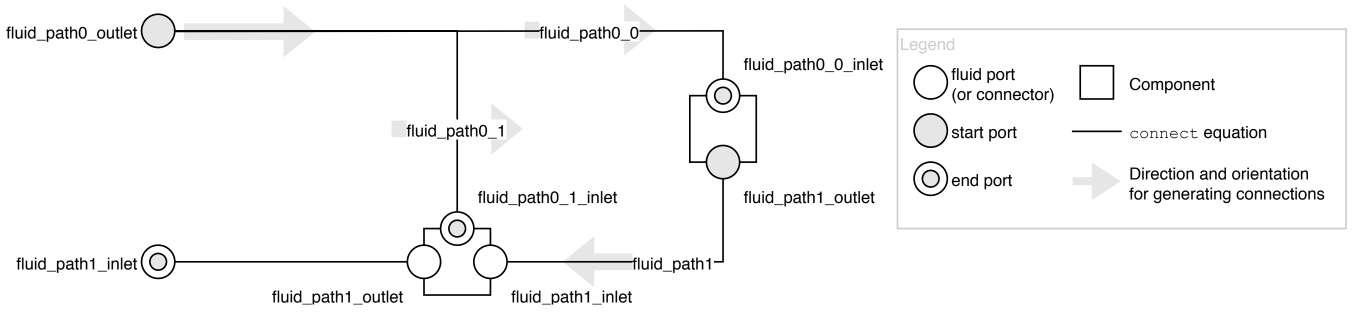


Fig. 3.6: Connection scheme with a fluid junction not modeled explicitly e.g. three-way valve. In this example the bypass and direct branches are derived paths from `fluid_path0` which consists only in one connector.

`{connector_name1: air_supply_inlet, connector_name2: air_supply_outlet, ...}, ... }")).`

All object names in `__Linkage(Connect(tags="{...}"))` annotation thus reference instantiated objects with fluid ports that have to be connected to each other. To build the full connection set, the direction (north, south, east, west) along which the objects must be connected needs to be provided.

Note: The direction (as well as the fluid medium) of a derived path are inherited from the parent path.

Modelica `connect` construct is symmetric so at first glance only the vertical / horizontal direction of a fluid path seems enough. However the actual orientation along the fluid path is needed in order to identify the start and end connectors, see below.

That information is stored in `__Linkage(Connect(paths="{fluid_path1: {direction: horizontal_or_vertical, ...}, ...}"))` for all main (not derived) fluid paths.

The connection logic is then as follows:

- List all the different fluid paths in `__Linkage(Connect(tags="{...}"))` as obtained by truncating `_inlet` and `_outlet` from each connector name. Get the orientation and direction of the main fluid paths from `__Linkage(Connect(paths="{...}"))` and finally reconstruct the tree structure of the fluid paths based on their names:

```
└── fluid_path0 (direction: east): [connectors list]
  ├── fluid_path0_0 (inherited direction: east): [connectors list]
  └── fluid_path0_1 (inherited direction: east): [connectors list]
    ├── fluid_path0_1_0 (inherited direction: east): [connectors list]
    └── fluid_path0_1_1 (inherited direction: east): [connectors list]
├── fluid_path1 (direction: west): [connectors list]
├── fluid_path3 (direction: north): [connectors list]
└── fluid_path4 (direction: south): [connectors list]
```

- For each fluid path:
  - Order all the connectors in the connectors list according to the direction of the fluid path and based on the position of the corresponding *objects* (not connectors) with the constraint that for each object `inlet` has to be listed first and `outlet` last.

- For each *derived path* find the start and end connectors as described hereunder and prepend / append the connectors list.
    - If the first (resp. last) connector in the ordered list is an outlet (resp. inlet), it is the start (resp. end) connector. (Note that the reciprocal is not true: a start port can be either an inlet or an outlet see Fig. 3.7.)
    - Otherwise the start (resp. end) connector is the outlet (resp. inlet) connector of the object in the parent path placed immediately before (resp. after) the object corresponding to the first (resp. last) connector – where before and after are relative to the direction and orientation of the fluid path (which are the same for the parent path).
- For each *parent path* split the path into several *sub paths* whenever a connector corresponds to the start or end port of a derived path.
- Throw an exception if one of the following rules is not verified:
    - Derived paths must start *or* end with a connector from a parent path.
    - Each branch of a fork must be a derived path, it cannot belong to the parent path: so no object from the parent path can be positioned between the objects corresponding to the first and last connector of any derived path.
- Generate the `connect` equations by iterating on the ordered list of connectors and generate the connection path and the corresponding graphical annotation. The only valid connection along a fluid path is `outlet` with `inlet`.
- Populate the `iconTransformation` annotation of each outside connector instantiated as a dependency so that they belong to the same border (top, left, bottom, right) as in the diagram layer and be evenly positioned considering the icon's dimensions. The bus connector is an exception and will always be positioned at the top center of the icon.

The implications of that logic are the following:

- Within the same fluid path, objects are connected in a given direction and orientation: to represent a fluid loop (graphically) at least two fluid paths must be defined, typically `supply` and `return`.

Fig. 3.7 to Fig. 3.9 further illustrate the connection logic on different test cases.

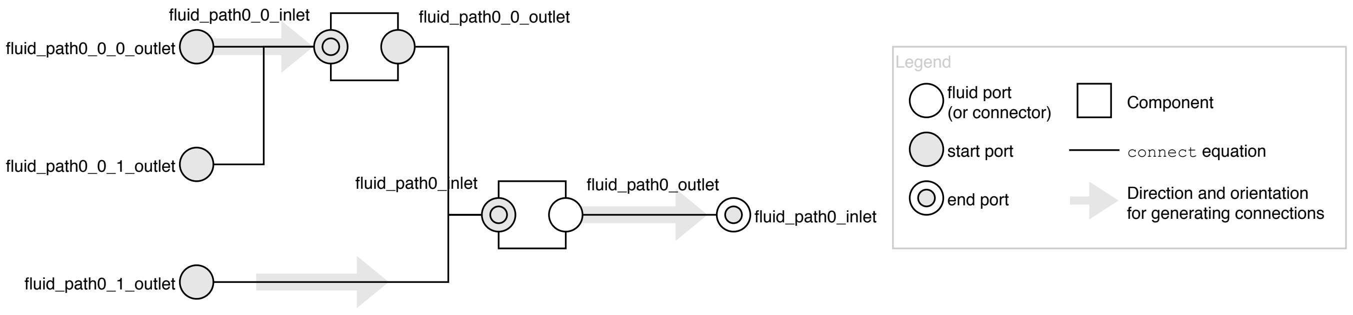


Fig. 3.7: Connection scheme with nested fluid junctions not modeled explicitly

3.4.3 Signal Connectors

3.4.3.1 General Principles

Generating the `connect` equations for signal variables relies on:

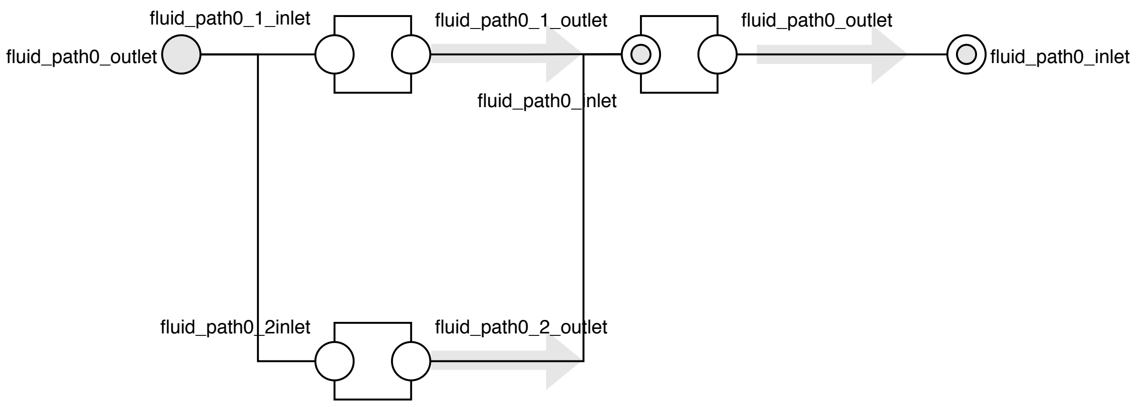


Fig. 3.8: *Connection scheme with fluid branches with identical directions e.g. AHU with dedicated outdoor air damper for economizer*

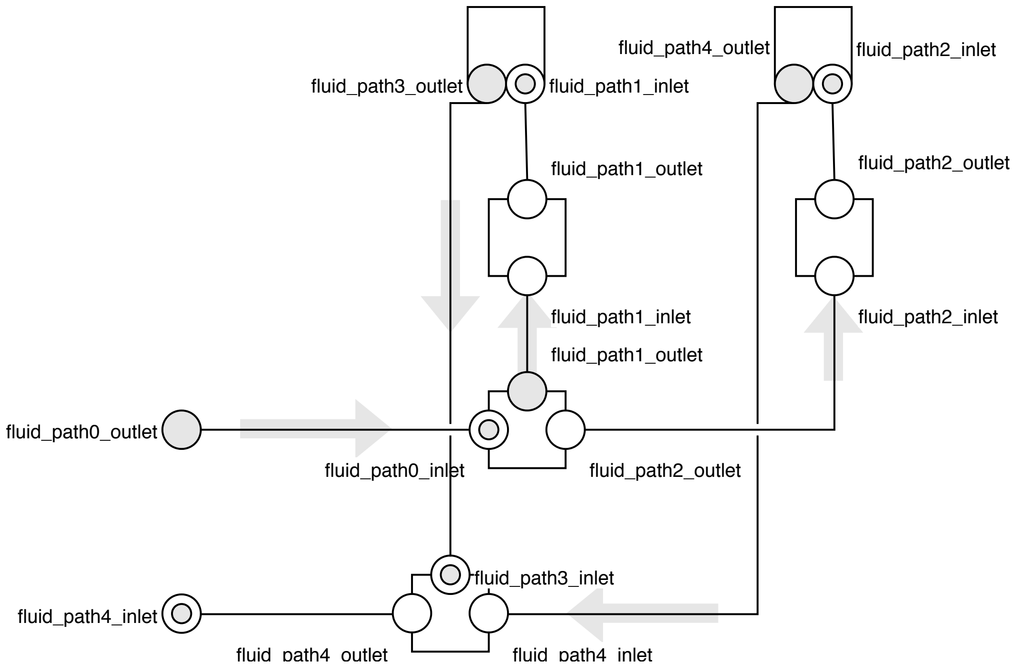


Fig. 3.9: *Connection scheme with fluid branches with different directions e.g. VAV duct system. Here a flow splitter is used to start several main fluid paths with a vertical connection direction.*

- a (fuzzy) string matching principle applied to the names of the connector variables and their components e.g. `com.y` for the output connector `y` of the component `com`,
- a so-called *control bus* which has the type of an expandable connector, see *§9.1.3 Expandable Connectors* in [Mod17].
  (For clarity it might be useful to group control input variables in one sub-bus and control output variables in another sub-bus. The experience feedback on bus usage in Modelica shows that restricting the number of sub-buses and the use of bus variables to sensed and actuated signals only is a preferred option.)

The following features of the expandable connector are leveraged:

1. All components in an expandable connector are seen as connector instances even if they are not declared as such. In comparison to a non expandable connector, that means that each variable (even of type `Real`) can be connected i.e. be part of a `connect` equation.

   Note: Connecting a non connector variable to a connector variable with `connect(non_connector_var, connector_var)` yields a warning but not an error in Dymola. It is considered bad practice though and a standard equation should be used in place `non_connector_var = connector_var`.
   Using a `connect` equation allows to draw a connection line which makes the model structure explicit to the user. Furthermore it avoids mixing `connect` equations and standard equations within the same equation set, which has been adopted as a best practice in the Modelica Buildings library.

2. The causality (input or output) of each variable inside an expandable connector is not predefined but rather set by the `connect` equation where the variable is first being used. For instance when the variable of an expandable connector is first connected to an inside connector `Modelica.Blocks.Interfaces.RealOutput` it gets the same causality i.e. output. The same variable can then be connected to another inside connector `Modelica.Blocks.Interfaces.RealInput`.
3. Potentially present but not connected variables are eventually considered as undefined i.e. a tool may remove them or set them to the default value (Dymola treat them as not declared: they are not listed in `dsin.txt`): all variables need not be connected so the control bus does not have to be reconfigured depending on the model structure.
4. The variables set of a class of type expandable connector is augmented whenever a new variable gets connected to any *instance* of the class. Though that feature is not needed by the configuration widget (we will have a predefined control bus with declared variables), it is needed to allow the user further modifying the control sequence. Adding new control variables is simply done by connecting them to the control bus.

Those features are illustrated with a minimal example in annex, see Section 5.2.

3.4.3.2 Generating Connections by Approximate String Matching

To support automatic connections of signal variables a predefined control bus will be defined for each type of system (e.g. VAV, CHW plant) with a set of predeclared variables. The names of the variables must allow a one-to-one correspondence between:

- the control sequence input variables and the outputs of the equipment model e.g. sensed quantities and actuators returned positions,
- the control sequence output variables and the inputs of the equipment model e.g. actuators commanded positions.

Thus the control bus variables are used as “gateways” to stream values between the controlled system and the controller system.

However an exact string matching is not conceivable. An approximate (or fuzzy) string matching algorithm must be used instead.

Listing 3.1: Example of a Python function used for fuzzy string matching

```python
from fuzzywuzzy import fuzz
from fuzzywuzzy import process

import itertools as it
import re


def return_best(string, choices):
    # Constrain array to array and scalar (or array element) to scalar.
    # Need to specify a logic for tagging scalar variables that should be connected to array␣
↪elements e.g. '*_zon*.y'.
    # Guard against array element A[i] to be connected to scalar variable.
    if bool(re.search('\[.+\]|_zon.*\.', string)) and not bool(re.search('\[\d+\]', string)):
        choices = [el for el in choices if re.search('\[.+\]', el)]
        # Replace [.*] by [:]
        string = re.sub('\[.*\]', '[:]', string, flags=re.I)
        string = re.sub('_zon.*\.', '[:].', string, flags=re.I)
    else:
        choices = [el for el in choices if not re.search('\[.+\]', el)]

    # Replace pre by p and tem by t.
    string = re.sub('pre', 'P', string, flags=re.I)
    string = re.sub('tem', 'T', string, flags=re.I)

    # Perform comparison.
    res = process.extract(string, choices, limit=2, scorer=fuzz.token_sort_ratio)

    return list(it.chain(*res))
```

Results in Fig. 3.10.

3.4.3.3 Validation and Additional Requirements

The use of expandable connectors (control bus) is validated in case of a complex controller, see Section 5.3.

Note: Connectors with conditional instances must be connected to the bus variables with the same conditional statement e.g.

```
if have_occSen then
    connect(ahuSubBusI.nOcc[1:numZon], nOcc[1:numZon])
end if;
```

With Dymola, bus variables cannot be connected to array connectors without explicitly specifying the indices range. Using

Connector	Variable to connect to	Bus variable	IO	match	score	sec_score	match_to	score_to	sec_score_to
TDis[numZon]	TDis_zonA.T	TDis[:]	I	TDis[:]	100	46	TDis[:]	80	40
TMix	TMix.T	TMix	I	TMix	100	25	TMix	80	40
TOut	TOut.T	TOut	I	TOut	100	75	TOut	80	62
TOutCut	TOutCut.y	TOutCut	I	TOutCut	100	86	TOutCut	88	75
TSup	TSup.T	TSup	I	TSup	100	73	TSup	80	62
TZonCooSet	modSetZon.TZonCooSet[1]	TZonCooSet	I	TZonCooSet	100	70	TZonCooSet	62	44
TZonHeaSet	modSetZon.TZonHeaSet[numZom]	TZonHeaSet	I	TZonHeaSet	100	70	TZon[:]	33	33
TZonResReq[nin].u	conVAVBox_zonA.yZonTemResReq	reqResT[:]	I	TZon[:]	50	42	reqResT[:]	29	26
TZon[numZon]	TZon_zonA.T	TZon[:]	I	TZon[:]	100	57	TZon[:]	80	50
VDis_flow[numZon]	V_flowDis_zonA.V_flow	V_flowDis[:]	I	V_flowDis[:]	67	46	V_flowDis[:]	72	30
VOut_flow	V_flowOut.V_flow	V_flowOut	I	V_flowOut	67	46	V_flowOut	72	35
ducStaPre	pStaDuc.p_rel	pStaDuc	I	staFrePro	50	43	pStaDuc	70	70
hOut	hOut.h	hOut	I	hOut	100	75	hOut	80	62
hOutCut	hOutCut.y	hOutCut	I	hOutCut	100	86	hOutCut	88	75
nOcc[numZon]	nOcc[numZom].y	nOcc[:]	I	nOcc[:]	100	25	nOcc[:]	80	20
uFreProSta	frePro.ySta	staFrePro	I	staFrePro	63	59	staFrePro	60	56
uOpeMod	modSetZon.yOpeMod	modOpe	I	modOpe	46	43	modOpe	52	44
uWin[numZon]	staWin_zonA.y	staWin[:]	I	staWin[:]	60	31	staWin[:]	86	33
uZonPreResReq	reqResPZon.y	reqResPZon	I	TZonHeaSet	57	48	reqResPZon	91	82
uZonTemResReq	reqResTZon.y	reqResTZon	I	TZonHeaSet	57	48	reqResTZon	91	82
TSupSet	TSupSet.T	TSupSet	O	TSupSet	100	73	TSupSet	88	62
yCoo	valCoo.y	yValCoo	O	yValCoo	73	43	yValCoo	80	47
yHea	valHea.y	yValHea	O	yValHea	73	43	yValHea	80	40
yOutDamPos	eco.yOut	yDamOut	O	yDamOut	59	47	yDamOut	53	50
yRetDamPos	eco.yRet	yDamRet	O	yDamOut	59	47	reqResT	53	53
ySupFanSpe	fanSup.y	yFanSup	O	yFanSup	71	59	yFanSup	80	50
pStaDuc	pStaDuc.p_rel	pStaDuc	I	pStaDuc	100	100	pStaDuc	70	70
reqResTZon[nin].u	conVAVBox_zonA.yReqResT	reqResTZon[:]	I	reqResTZon[:]	91	74	reqResT[:]	56	50
uReqResP	reqResP.y	reqResP	I	reqResP	93	80	reqResP	88	75
uReqResT	reqResT.y	reqResT	I	reqResT	93	80	reqResT	88	75

Fig. 3.10: Fuzzy string matching test case – G36 VAV AHU Controller

the unspecified `[:]` syntax yields the following translation error.

```
Failed to expand conAHU.ahuSubBusI.nOcc[:] (since element does not exist) in connect(conAHU.
↪ahuSubBusI.nOcc[:], conAHU.nOcc[:]);
```

Providing an explicit indices range e.g. `[1:numZon]` like in the previous code snippet only causes a translation warning: Dymola seems to allocate a default dimension of **20** to the connector, the unused indices (from 3 to 20 in the example hereunder) are then removed from the simulation problem since they are not used in the model.

```
Warning: The bus-input conAHU.ahuSubBusI.VDis_flow[3] matches multiple top-level connectors␣
↪in the connection sets.

Bus-signal: ahuI.VDis_flow[3]

Connected bus variables:
ahuSubBusI.VDis_flow[3] (connect) "Connector of Real output signal"
conAHU.ahuBus.ahuI.VDis_flow[3] (connect) "Primary airflow rate to the ventilation zone from␣
↪the air handler, including  outdoor air and recirculated air"
ahuBus.ahuI.VDis_flow[3] (connect)
conAHU.ahuSubBusI.VDis_flow[3] (connect)
```

This is a strange behavior in Dymola. On the other hand JModelica 1) allows the unspecified `[:]` syntax and 2) does not generate any translation warning when explicitly specifying the indices range. JModelica's behavior seems more aligned with [Mod17] *§9.1.3 Expandable Connectors* that states: "A non-parameter array element may be declared with array dimensions ":" indicating that the size is unknown." The same logic as JModelica for array variables connections to expandable connectors is required for LinkageJS.

3.4.3.4 Additional Requirements for the UI

Based on the previous validation case, Fig. 3.11 presents the Dymola pop-up window displayed when connecting the sub-bus of input control variables to the main control bus. A similar view of the connections set must be implemented with the additional requirements listed below. That view is displayed in the connections tab of the right panel.

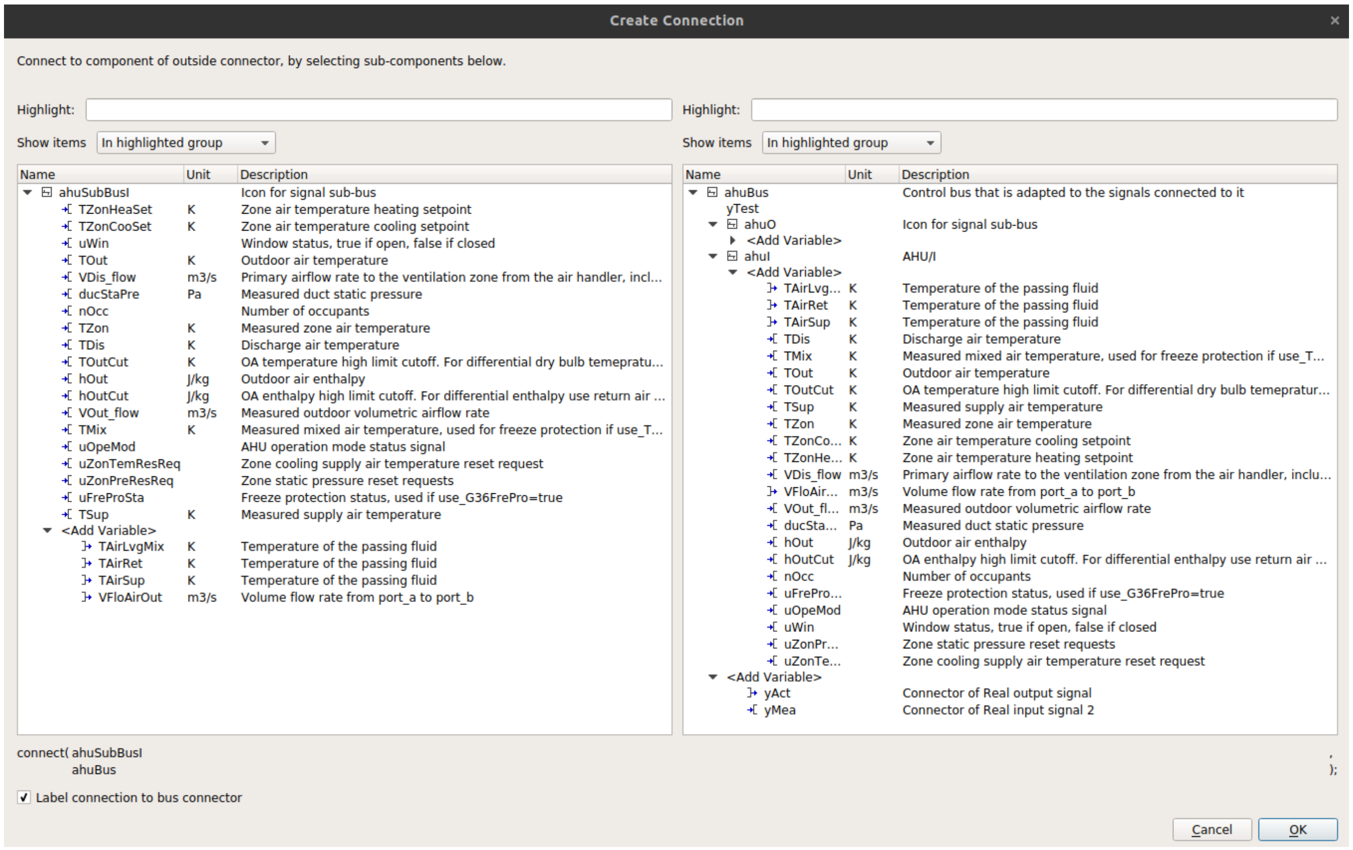
The variables listed immediately after the bus name are either:

- *declared variables* that are not connected e.g. `ahuBus.yTest` (declared as `Real` in the bus definition): those variables are only *potentially present* and eventually considered as *undefined* when translating the model (treated by Dymola as if they were never declared) or,
- *present variables* i.e. variables that appear in a connect equation e.g. `ahuSubBusI.TZonHeaSet`: the icon next to each variable then indicates the causality. Those variables can originally be either declared variables or variables elaborated by the augmentation process for *that instance* of the expandable connector i.e. variables that are declared in another component and connected to the connector's instance.

The variables listed under `Add variable` are the remaining *potentially present variables* (in addition to the declared but not connected variables). Those variables are elaborated by the augmentation process for *all instances* of the expandable connector, however they are not connected in that instance of the connector.

In addition to Dymola's features for handling the bus connections, LinkageJS requires the following:

- Color code to distinguish between:


Create Connection

Connect to component of outside connector, by selecting sub-components below.

Highlight:

Show items In highlighted group

Name	Unit	Description
ahuSubBusI		Icon for signal sub-bus
TZonHeaSet	K	Zone air temperature heating setpoint
TZonCooSet	K	Zone air temperature cooling setpoint
uWin		Window status, true if open, false if closed
TOut	K	Outdoor air temperature
VDis_flow	m3/s	Primary airflow rate to the ventilation zone from the air handler, incl...
ducStaPre	Pa	Measured duct static pressure
nOcc		Number of occupants
TZon	K	Measured zone air temperature
TDis	K	Discharge air temperature
TOutCut	K	OA temperature high limit cutoff. For differential dry bulb temepratu...
hOut	J/kg	Outdoor air enthalpy
hOutCut	J/kg	OA enthalpy high limit cutoff. For differential enthalpy use return air ...
VOut_flow	m3/s	Measured outdoor volumetric airflow rate
TMix	K	Measured mixed air temperature, used for freeze protection if use_T...
uOpeMod		AHU operation mode status signal
uZonTemResReq		Zone cooling supply air temperature reset request
uZonPreResReq		Zone static pressure reset requests
uFreProSta		Freeze protection status, used if use_G36FrePro=true
TSup	K	Measured supply air temperature
<Add Variable>		
TAirLvgMix	K	Temperature of the passing fluid
TAirRet	K	Temperature of the passing fluid
TAirSup	K	Temperature of the passing fluid
VFloAirOut	m3/s	Volume flow rate from port_a to port_b

Highlight:

Show items In highlighted group

Name	Unit	Description
ahuBus		Control bus that is adapted to the signals connected to it
yTest		
ahuO		Icon for signal sub-bus
<Add Variable>		
ahuI		AHU/I
<Add Variable>		
TAirLvg...	K	Temperature of the passing fluid
TAirRet	K	Temperature of the passing fluid
TAirSup	K	Temperature of the passing fluid
TDis	K	Discharge air temperature
TMix	K	Measured mixed air temperature, used for freeze protection if use_T...
TOut	K	Outdoor air temperature
TOutCut	K	OA temperature high limit cutoff. For differential dry bulb temepratur...
TSup	K	Measured supply air temperature
TZon	K	Measured zone air temperature
TZonCo...	K	Zone air temperature cooling setpoint
TZonHe...	K	Zone air temperature heating setpoint
VDis_flow	m3/s	Primary airflow rate to the ventilation zone from the air handler, inclu...
VFloAir...	m3/s	Volume flow rate from port_a to port_b
VOut_fl...	m3/s	Measured outdoor volumetric airflow rate
ducSta...	Pa	Measured duct static pressure
hOut	J/kg	Outdoor air enthalpy
hOutCut	J/kg	OA enthalpy high limit cutoff. For differential enthalpy use return air ...
nOcc		Number of occupants
uFrePro...		Freeze protection status, used if use_G36FrePro=true
uOpeMod		AHU operation mode status signal
uWin		Window status, true if open, false if closed
uZonPr...		Zone static pressure reset requests
uZonTe...		Zone cooling supply air temperature reset request
<Add Variable>		
yAct		Connector of Real output signal
yMea		Connector of Real input signal 2

```
connect( ahuSubBusI                                                    ,
         ahuBus                                                        );
```

✓ Label connection to bus connector

Fig. 3.11: Dymola pop-up window when connecting the sub-bus of input control variables (left) to the main control bus (right) – case of outside connectors

  - Variables connected only once (within the entire augmentation set): those variables should be listed first and in red color. This is needed so that the user immediately identify which connections are still required for the model to be complete.

> **Warning:** Dymola does not throw any exception when a *declared* bus variable is connected to an input (resp. output) variable but not connected to any other non input (resp. non output) variable. It then uses the default value (0 for `Real`) to feed the connected variable.
> That is not the case if the variable is not declared i.e. elaborated by augmentation: in that case it has to be connected in a consistent way.
> JModelica throws an exception in any case with the message `The following variable(s) could not be matched to any equation`.

  - Declared variables which are only potentially present (not connected): those variables should be listed last (not first as in Dymola) and in light grey color. That behavior is also closer to [Mod17] *§9.1.3 Expandable Connectors*: "variables and non-parameter array elements declared in expandable connectors are marked as only being potentially present. [. . .] elements that are only potentially present are not seen as declared."

- View the "expanded" connection set of an expandable connector in each level of composition – that covers several topics:
  - The user can view the connection set of a connector simply by selecting it and without having to make an actual connection (as in Dymola).
  - The user can view the name of component and connector variable to which the expandable connector's variables are connected: similar to Dymola's function `Find Connection` accessible by right-clicking on a connection line.
  - From [Mod17] *§9.1.3 Expandable Connectors*: "When two expandable connectors are connected, each is augmented with the variables that are only declared in the other expandable connector (the new variables are neither input nor output)."

    That feature is illustrated in the minimal example Fig. 3.12 where a sub-bus `subBus` with declared variables `yDeclaredPresent` and `yDeclaredNotPresent` is connected to the declared sub-bus `bus.ahuI` of a bus. `yDeclaredPresent` is connected to another variable so it is considered present. `yDeclaredNotPresent` is not connected so it is only considered potentially present. Finally `yNotDeclaredPresent` is connected but not declared which makes it a present variable. Fig. 3.13 to Fig. 3.15 then show which variables are exposed to the user. In consistency with [Mod17] the declared variables of `subBus` are considered declared variables in `bus.ahuI` due to the connect equation between those two instances and they are neither input nor output. Furthermore the present variable `yNotDeclaredPresent` appears in `bus.ahuI` under `Add variable` i.e. as a potentially present variable whereas it is a present variable in the connected sub-bus `subBus`.
    - ⋆ This is an issue for the user who will not have the information at the bus level of the connections which are required by the sub-bus variables e.g. Dymola will allow connecting an output connector to `bus.ahuI.yDeclaredPresent` but the translation of the model will fail due to `Multiple sources for causal signal in the same connection set`.
    - ⋆ Directly connecting variables to the bus (without intermediary sub-bus) can solve that issue for outside connectors but not for inside connectors, see below.
  - Another issue is illustrated Fig. 3.15 where the connection to the bus is now made from an outside component for which the bus is considered as an inside connector. Here Dymola only displays declared variables of the bus (but not of the sub-bus) but without the causality information and even if it is only potentially present (not connected). Present variables of the bus or sub-bus which are not declared are not

displayed. Contrary to Dymola, LinkageJS requires that the “expanded” connection set of an expandable connector be exposed, independently from the level of composition. That means exposing all the variables of the *augmentation set* as defined in [Mod17] *9.1.3 Expandable Connectors*. In our example the same information displayed in Fig. 3.13 for the original sub-bus should be accessible when displaying the connection set of `bus.ahuI` whatever the current status (inside or outside) of the connector `bus`. A typical view of the connection set of expandable connectors for LinkageJS could be:

Table 3.3: Typical view of the connection set of expandable connectors – visible from outside component (connector is inside), “Present” and “I/O” columns display the connection status over the full augmentation set

Variable	Present	Declared	I/O	Description
bus				
`var1` (present variable connected only once: red color)	x	O	→ `comp1.var1`	…
`var2` (present variable connected twice: default color)	x	O	`comp2.var1` → `comp1.var2`	…
`var3` (declared variable not connected: light grey color)	O	x		…
Add variable				
`var4` (variable elaborated by augmentation from *all instances* of the connector: light grey color)	O	O		…
subBus				
`var5` (present variable connected only once: red color)	x	O	`comp3.var5` →	…
Add variable				
`var6` (variable elaborated by augmentation from *all instances* of the connector: light grey color)	O	O		…

3.4.3.5 Parameters Setting

To be updated.

The name and comment of the instance must be displayed in the parameters tab.

3.5 Schematics Export

3.6 Working with Tagged Variables

To be updated: specify the requirements for tagging variables and performing some queries of the set of tagged variables

Set up parameters values with OS measures e.g. nominal electrical loads or boiler efficiency

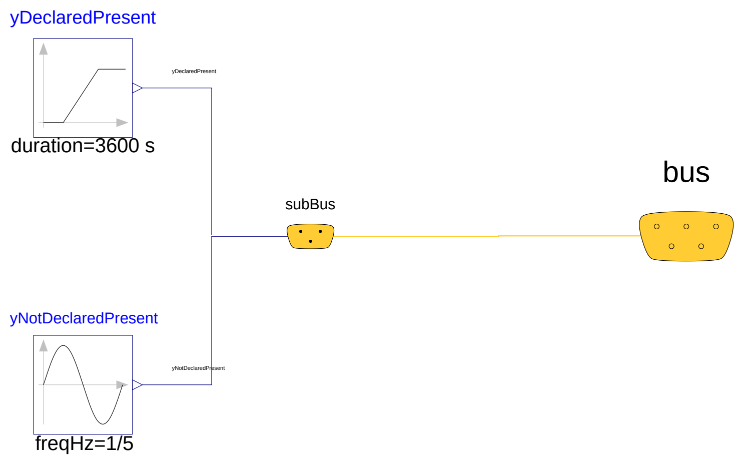
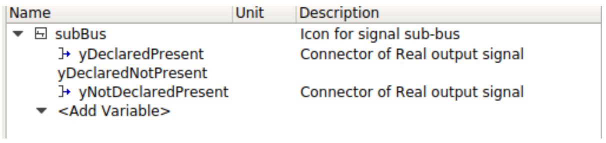
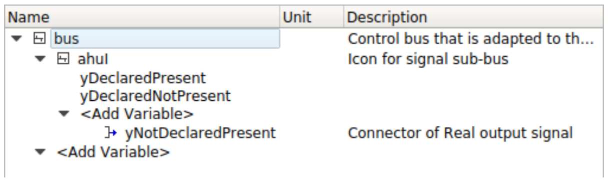


Fig. 3.12: Minimal example of sub-bus to bus connection illustrating how the bus variables are exposed in Dymola – case of outside connectors



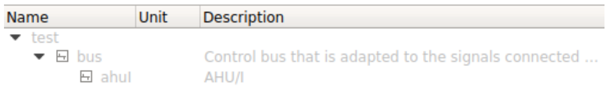
Name	Unit	Description
subBus		Icon for signal sub-bus
yDeclaredPresent		Connector of Real output signal
yDeclaredNotPresent		
yNotDeclaredPresent		Connector of Real output signal
<Add Variable>		

Fig. 3.13: Sub-bus variables being exposed in case the sub-bus is an outside connector



Name	Unit	Description
bus		Control bus that is adapted to th...
ahuI		Icon for signal sub-bus
yDeclaredPresent		
yDeclaredNotPresent		
<Add Variable>		
yNotDeclaredPresent		Connector of Real output signal
<Add Variable>		

Fig. 3.14: Bus variables being exposed in case the bus is an outside connector



Name	Unit	Description
test		
bus		Control bus that is adapted to the signals connected ...
ahuI		AHU/I

Fig. 3.15: Bus variables being exposed in case the bus is an inside connector

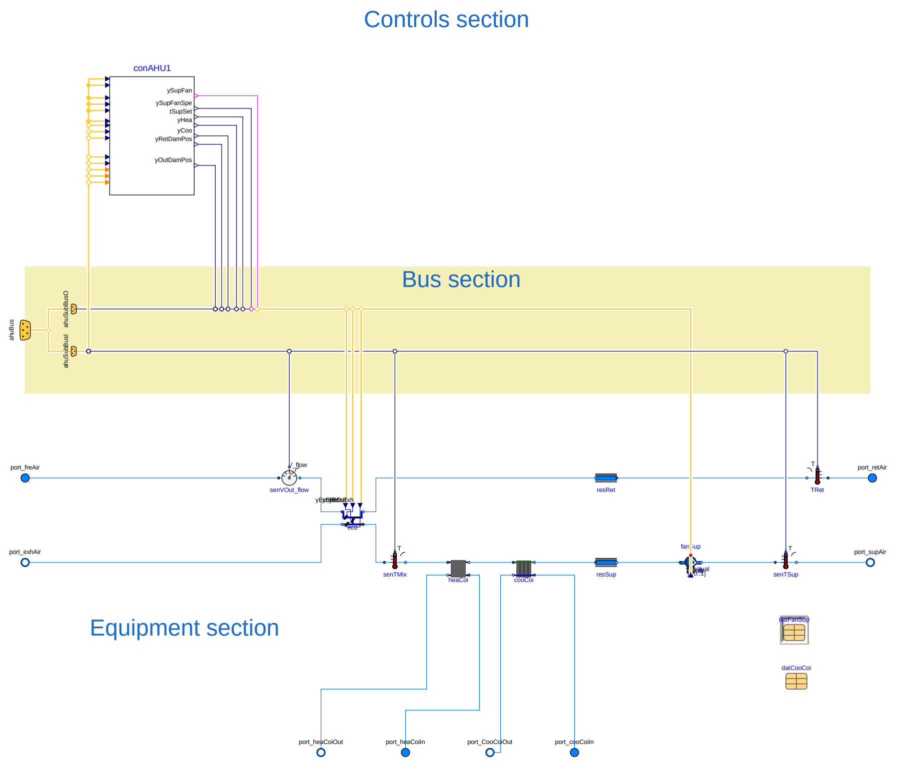
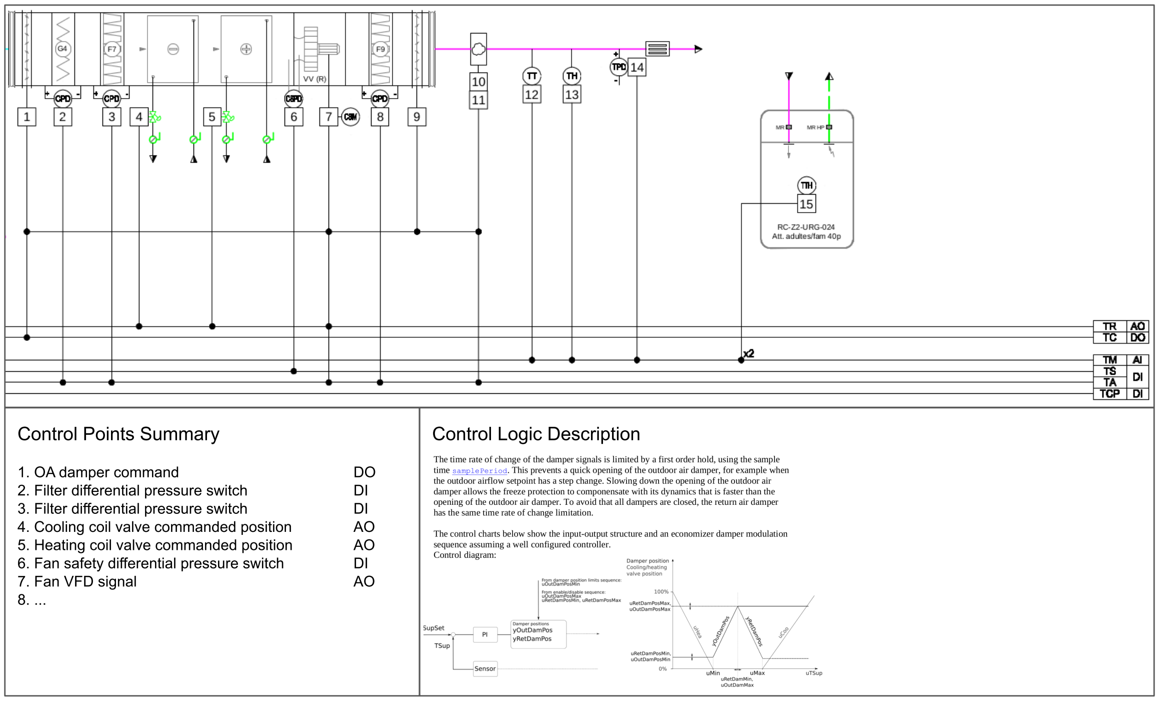


Fig. 3.16: *Mockup of the schematics export – Input Modelica file*



Control Points Summary

1. OA damper command	DO
2. Filter differential pressure switch	DI
3. Filter differential pressure switch	DI
4. Cooling coil valve commanded position	AO
5. Heating coil valve commanded position	AO
6. Fan safety differential pressure switch	DI
7. Fan VFD signal	AO
8. ...	

Control Logic Description

The time rate of change of the damper signals is limited by a first order hold, using the sample time `samplePeriod`. This prevents a quick opening of the outdoor air damper, for example when the outdoor airflow setpoint has a step change. Slowing down the opening of the outdoor air damper allows the freeze protection to componensate with its dynamics that is faster than the opening of the outdoor air damper. To avoid that all dampers are closed, the return air damper has the same time rate of change limitation.

The control charts below show the input-output structure and an economizer damper modulation sequence assuming a well configured controller.
Control diagram:

Fig. 3.17: *Mockup of the schematics export – Output file (format to be specified: Word or PDF)*

3.7 OpenStudio Integration

To be updated.

3.8 Interface with URBANopt GeoJSON

To be updated.

3.9 Encryption

See current standardization effort in #1868.

3.10 Licensing

To be updated cf. licensing strategy different for each integration target

Chapter 4

Software Architecture

Fig. 4.1 presents the envisioned software architecture for the desktop app.

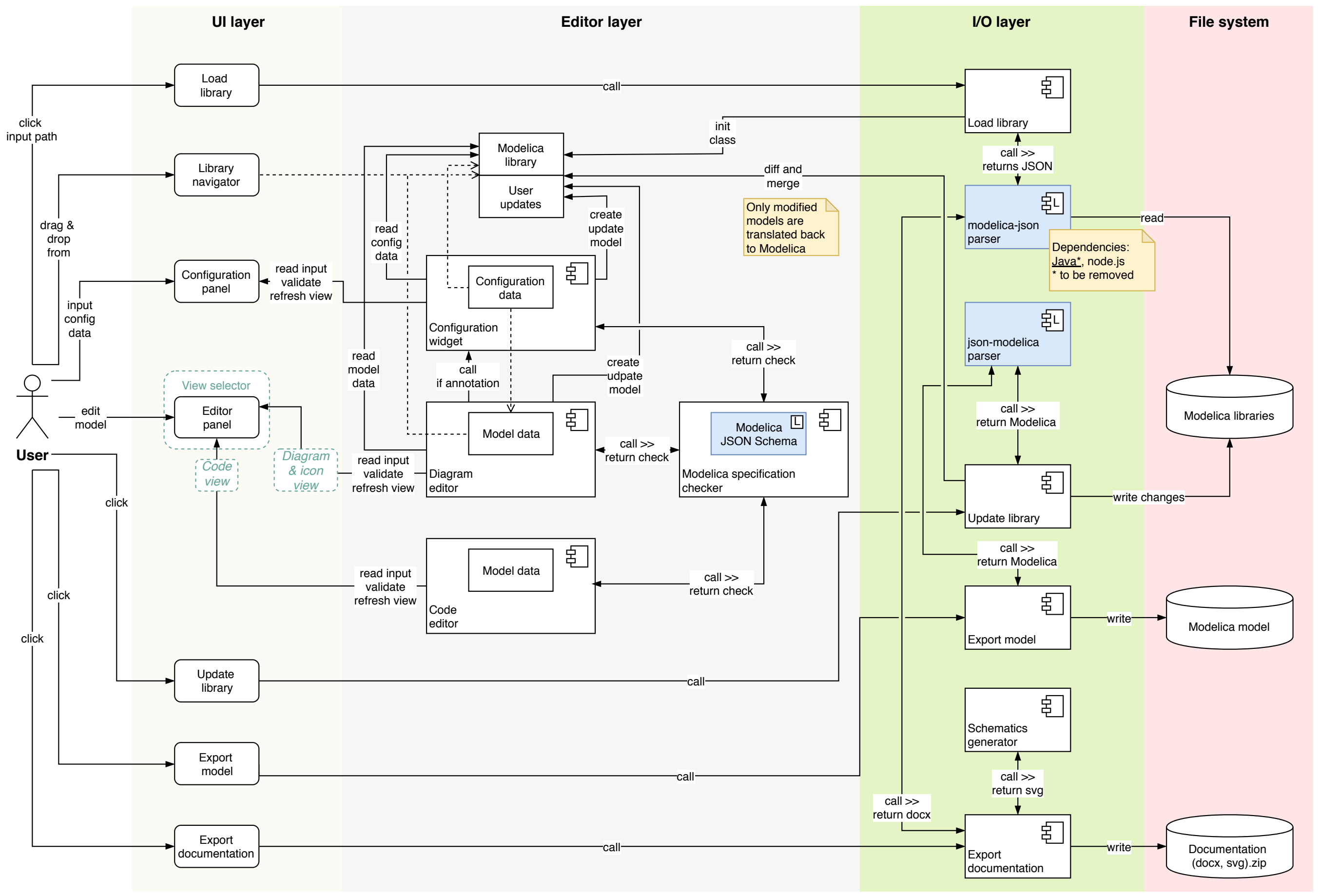


Fig. 4.1: Software architecture - Desktop app

Chapter 5

Annex

5.1 Example of the Configuration Data Structure

Listing 5.1: Partial example of the configuration data structure for an air handling unit (pseudo-code, especially for autoreferencing the data structure and writing conditional statements)

```
{
    "system": {
        "$id": "#system",
        "description": "System type",
        "value": "AHU"
    },
    "icon": "path of icon.mo",
    "diagram": {
        "configuration": [24, 24],
        "modelica": [[-120,-200], [120,120]]
    },
    "name": {
        "$id": "#name",
        "description": "Model name",
        "widget": {
            "type": "Text input"
        },
        "value": "AHU"
    },
    "type": {
        "$id": "#type",
        "description": "Type of AHU",
        "widget": {
            "type": "Dropdown",
            "options": ["VAV", "DOA", "Supply only", "Exhaust only"]
        },
```

(continues on next page)

(continued from previous page)

```
        "value": "VAV"
    },
    "fluid_paths": [
        {
            "$id": "#air_supply",
            "direction": "east",
            "medium": "Buildings.Media.Air"
        },
        {
            "$id": "#air_return",
            "direction": "west",
            "medium": "Buildings.Media.Air"
        }
    ],
    "equipment": [
        {
            "$id": "#heaRec",
            "description": "Heat recovery",
            "enabled": "#type.value == 'DOA'",
            "widget": {
                "type": "Dropdown menu",
                "options": ["None", "Fixed plate", "Enthalpy wheel", "Sensible wheel"]
            },
            "value": "None",
            "declaration": [
                null,
                "Buildings.Fluid.HeatExchangers.PlateHeatExchangerEffectivenessNTU",
                "Buildings.Fluid.HeatExchangers.EnthalpyWheel",
                "Buildings.Fluid.HeatExchangers.EnthalpyWheel(sensible=true)"
            ],
            "icon_transformation": "flipHorizontal",
            "placement": [18, 6],
            "connect": {
                "type": "tags",
                "value": {
                    "port_a1": "air_return_inlet", "port_a2": "air_supply_inlet", "port_b1":
↪"air_return_outlet", "port_b2": "air_supply_outlet"
                }
            }
        },
        {
            "$id": "#eco",
            "description": "Economizer",
            "enabled": "#type.value == 'VAV'",
            "widget": {
                "type": "Dropdown menu",
                "options": ["None", "Dedicated OA damper", "Common OA damper"]
            },
            "value": "None",
```

(continues on next page)

(continued from previous page)

```
            "declaration": [
                null,
                "Buildings.Fluid.Actuators.Dampers.MixingBoxMinimumFlow",
                "Buildings.Fluid.Actuators.Dampers.MixingBox"
            ],
            "icon_transformation": "flipVertical",
            "placement": [18, 9],
            "connect": {
                "type": "tags",
                "value": {
                    "port_Out": "air_supply_out_inlet", "port_OutMin": "air_supply_min_inlet
↪", "port_Sup": "air_supply_outlet",
                    "port_Exh": "air_return_outlet", "port_Ret": "air_return_inlet"
                }
            }
        },
        {
            "$id": "#V_flowOut_nominal",
            "description": "Nominal outdoor air volumetric flow rate",
            "enabled": "#eco.value != 'None'",
            "widget": {
                "type": "Numeric input"
            },
            "declaration": "Modelica.SIunits.VolumeFlowRate",
            "value": 0,
            "unit": "m3/s"
        },
        {
            "$id": "#fanSup",
            "description": "Supply fan",
            "enabled": "#type.value != 'Exhaust only'",
            "widget": {
                "type": "Dropdown menu",
                "options": ["None", "Draw through", "Blow through"]
            },
            "value": "Draw through",
            "declaration": "Buildings.Fluid.Movers.SpeedControlled_y(m_flow_nominal=m_
↪flowSup_nominal)",
            "placement": [null, [18, 11], [18, 18]],
            "connect": {
                "value": "air_supply"
            }
        },
        {
            "$id": "#V_flowSup_nominal",
            "description": "Nominal supply air volumetric flow rate",
            "enabled": "#fanSup.value != 'None'",
            "widget": {
                "type": "Numeric input"
```

(continues on next page)

(continued from previous page)

```
            },
            "value": 0,
            "unit": "m3/h"
        },
        {
            "$id": "#fanRet",
            "description": "Return/Relief fan",
            "enabled": "#type.value != 'Supply only'",
            "widget": {
                "type": "Dropdown menu",
                "options": ["None", "Return", "Relief"]
            },
            "value": "Relief",
            "declaration": [
                null,
                "Buildings.Fluid.Movers.SpeedControlled_y(m_flow_nominal=m_flowRet_nominal)",
                "Buildings.Fluid.Movers.SpeedControlled_y(m_flow_nominal=m_flowRel_nominal)"
            ],
            "icon_transformation": "flipHorizontal",
            "placement": [null, [14, 13], [14, 4]],
            "connect": {
                "value": "air_return"
            }
        },
        {
            "$id": "#V_flowRet_nominal",
            "description": "Nominal return air volumetric flow rate",
            "enabled": "#fanRet.value != 'None'",
            "widget": {
                "type": "Numeric input"
            },
            "value": 0,
            "unit": "m3/h"
        }
    ],
    "controls": [
        {
            "$id": "#conAHURef",
            "description": "Reference guideline for control sequences",
            "widget": {
                "type": "Dropdown menu",
                "options": ["None", "ASHRAE 2006", "ASHRAE G36"]
            },
            "value": "None"
        },
        {
            "$id": "#conAHUOpt",
            "description": "Optimal start up",
            "widget": {
```

(continues on next page)

(continued from previous page)

```
                "type": "Dropdown menu",
                "options": ["None", "Heating", "Cooling", "Heating and cooling"],
            },
            "value": "None",
            "declaration": [null,
                "Buildings.Controls.OBC.CDL.Utilities.OptimalStartUp(mode=#conAHUOpt.value)",
                "Buildings.Controls.OBC.CDL.Utilities.OptimalStartUp(mode=#conAHUOpt.value)",
                "Buildings.Controls.OBC.CDL.Utilities.OptimalStartUp(mode=#conAHUOpt.value)"
            ]
        },
        {
            "$id": "#conFanSup",
            "description": "Supply fan control",
            "enabled": "#fanSup.value != 'None'",
            "widget": {
                "type": "Text"
            }
        },
        {
            "$id": "#conFanSupStaSto",
            "description": "Supply fan start/stop control",
            "enabled": "#conFanSup.widget.enabled",
            "widget": {
                "type": "Dropdown menu",
                "options": ["On-Off", "Static Pressure Control"],
                "options.enabled": ["#conAHURef.value != 'ASHRAE G36'", true]
            },
            "value": "if #conAHURef.value == null then 'On-Off' elseif #conAHURef.value ==
↪'ASHRAE G36' then 'Static Pressure Control'",
            "declaration": ["..."]
        },
        {
            "$id": "#resPreStaSet",
            "description": "Static pressure set point reset",
            "enabled": "#conFanSup.widget.enabled",
            "widget": {
                "type": "Dropdown menu",
                "options": ["None", "T&R"],
                "options.enabled": ["#conAHURef.value != 'ASHRAE G36'", true]
            },
            "value": "if #conAHURef.value == null then 'None' elseif #conAHURef.value ==
↪'ASHRAE G36' then 'T&R'",
            "declaration": ["..."]
        },
        {
            "$id": "#conTAirSup",
            "description": "Supply Air Temperature Control",
            "enabled": "#fanSup.value != 'None'",
            "widget": {
```

(continues on next page)

(continued from previous page)

```
                "type": "Text"
            }
        },
        {
            "$id": "#resTSupSet",
            "description": "Supply air temperature set point reset",
            "enabled": "#conTAirSup.widget.enabled",
            "widget": {
                "type": "Dropdown menu",
                "options": ["None", "OAT Reset", "OAT and T&R"],
                "options.enabled": ["#conAHURef.value != ('ASHRAE G36' or 'ASHRAE 2006')", "
↪#conAHURef.value != 'ASHRAE G36'", true]
            },
            "value": "if #conAHURef.value == null then 'None' else 'OAT and T&R'",
            "declaration": ["..."]
        },
        {
            "$id": "#numZon",
            "description": "Number of served VAV boxes",
            "enabled": "#conTAirSup.widget.enabled and #resTSupSet.value == 'OAT and T&R'",
            "widget": {
                "type": "Numeric input"
            },
            "value": null
        }
    ],
    "dependencies": [
        {
            "$id": "#port_outAir",
            "description": "Outside air port",
            "enabled": "#type.value != 'Exhaust only'",
            "declaration": "Modelica.Fluid.Interfaces.FluidPort_a(redeclare package Medium=
↪#air_supply.medium)",
            "placement": [18, 1],
            "connect": {
                "value": "air_supply"
            }
        },
        {
            "$id": "#port_supAir",
            "description": "Supply air port",
            "enabled": "#type.value != 'Exhaust only'",
            "declaration": "Modelica.Fluid.Interfaces.FluidPort_b(redeclare package Medium=
↪#air_supply.medium)",
            "placement": [18, 24],
            "connect": {
                "value": "air_supply"
            }
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "$id": "#senFloOut",
            "description": "Outdoor airflow measurement station",
            "enabled": "#eco.value != 'None'",
            "declaration": "Buildings.Fluid.Sensors.VolumeFlowRate(redeclare package Medium=
↪#air_supply.medium)",
            "placement": "if #eco.value == 'Dedicated OA damper' then [20, 5] else [18, 5]",
            "connect": {
                "value": "if #eco.value == 'Dedicated OA damper' then 'air_supply_min' else
↪'air_supply_out'"
            }
        },
        {
            "$id": "#m_flowOut_nominal",
            "description": "Nominal outdoor air mass flow rate",
            "enabled": "#V_flowOut_nominal.enabled",
            "declaration": "Modelica.SIunits.MassFlowRate",
            "value": "(#air_supply.medium).rho_default / 3600 * V_flowOut_nominal",
            "protected": true
        },
        {
            "$id": "#m_flowSup_nominal",
            "description": "Nominal supply air mass flow rate",
            "enabled": "#V_flowSup_nominal.enabled",
            "declaration": "Modelica.SIunits.MassFlowRate",
            "value": "(#air_supply.medium).rho_default / 3600 * V_flowSup_nominal",
            "protected": true
        },
        {
            "$id": "#m_flowRet_nominal",
            "description": "Nominal return air mass flow rate",
            "enabled": "#V_flowRet_nominal.enabled",
            "declaration": "Modelica.SIunits.MassFlowRate",
            "value": "(#air_supply.medium).rho_default / 3600 * V_flowRet_nominal",
            "protected": true
        },
        {
            "$id": "#m_flowRel_nominal",
            "description": "Nominal relief air mass flow rate",
            "enabled": "#eco.value != 'None'",
            "declaration": "Modelica.SIunits.MassFlowRate",
            "value": "m_flowRet_nominal - m_flowSup_nominal + m_flowOut_nominal",
            "protected": true
        }
    ]
}
```

5.2 Illustration of the Main Features of the Expandable Connectors

The main features of the expandable connectors (as described in Section 3.4.3) are illustrated with a minimal example described in the figures below where:

- a controlled system consisting in a sensor (idealized with a real expression) and an actuator (idealized with a simple block passing through the value of the input control signal) is connected with,
- a controller system which divides the input variable (measurement) by itself and thus outputs a control variable equal to one.
- The same model is first implemented with an expandable connector and then with a standard connector.

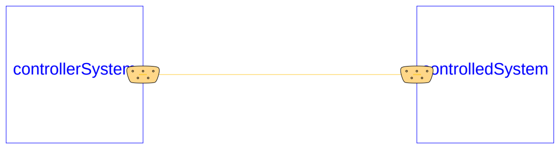


Fig. 5.1: Minimal example illustrating the connection scheme with an expandable connector – Top level

```
model BusTestExp
BusTestControllerExp controllerSystem;
BusTestControlledExp controlledSystem;
equation
      connect(controllerSystem.ahuBus, controlledSystem.ahuBus);
end BusTestExp;
```

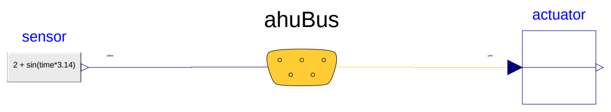


Fig. 5.2: Minimal example illustrating the connection scheme with an expandable connector – Controlled component sublevel

```
model BusTestControlledExp
Modelica.Blocks.Sources.RealExpression sensor(y=2 + sin(time*3.14));
Buildings.Experimental.Templates.BaseClasses.AhuBus ahuBus;
Modelica.Blocks.Routing.RealPassThrough actuator;
equation
      connect(sensor.y, ahuBus.yMea);
```

(continues on next page)

(continued from previous page)

```
      connect(ahuBus.yAct, actuator.u);
end BusTestControlledExp;
```

```
expandable connector AhuBus
extends Modelica.Icons.SignalBus;
end AhuBus;
```

Note: The definition of `AhuBus` in the code snippet here above does not include any variable declaration. However the variables `ahuBus.yAct` and `ahuBus.yMea` are used in `connect` equations. That is only possible with an expandable connector.

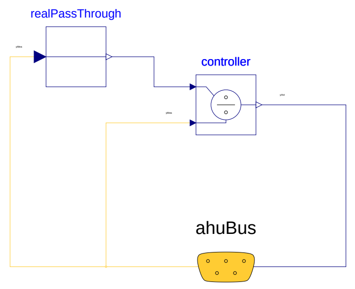


Fig. 5.3: Minimal example illustrating the connection scheme with an expandable connector – Controller component sublevel

```
model BusTestControlledExp
      Modelica.Blocks.Sources.RealExpression sensor(y=2 + sin(time*3.14));
      Buildings.Experimental.Templates.BaseClasses.AhuBus ahuBus;
      Modelica.Blocks.Routing.RealPassThrough actuator;
equation
      connect(ahuBus.yAct, actuator.u);
      connect(sensor.y, ahuBus.yMea)
end BusTestControlledExp;
```

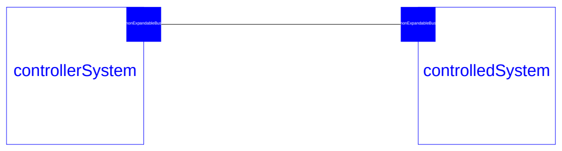


Fig. 5.4: Minimal example illustrating the connection scheme with a standard connector – Top level

```
model BusTestNonExp
BusTestControllerNonExp controllerSystem;
BusTestControlledNonExp controlledSystem;
equation
      connect(controllerSystem.nonExpandableBus, controlledSystem.nonExpandableBus);
end BusTestNonExp;
```

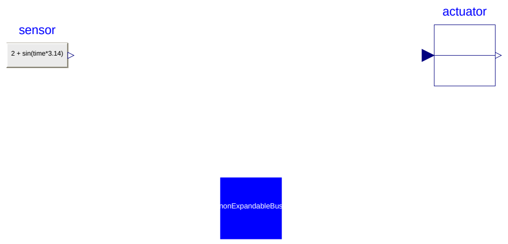


Fig. 5.5: Minimal example illustrating the connection scheme with a standard connector – Controlled component sublevel

```
model BusTestControlledNonExp
Modelica.Blocks.Sources.RealExpression sensor(y=2 + sin(time*3.14));
Modelica.Blocks.Routing.RealPassThrough actuator;
BaseClasses.NonExpandableBus nonExpandableBus;
equation
      nonExpandableBus.yMea = sensor.y;
      actuator.u = nonExpandableBus.yAct;
end BusTestControlledNonExp;
```

```
connector NonExpandableBus
// The following declarations are required.
// The variables are not considered as connectors: they cannot be part of connect equations.
Real yMea;
Real yAct;
end NonExpandableBus;
```

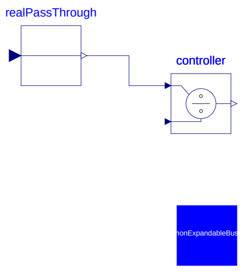


Fig. 5.6: Minimal example illustrating the connection scheme with a standard connector – Controller component sublevel

```
model BusTestControllerNonExp
Controls.OBC.CDL.Continuous.Division controller;
Modelica.Blocks.Routing.RealPassThrough realPassThrough;
BaseClasses.NonExpandableBus nonExpandableBus;
equation
      connect(realPassThrough.y, controller.u1);
      controller.u2 = nonExpandableBus.yMea;
      nonExpandableBus.yAct = controller.y;
      realPassThrough.u = nonExpandableBus.yMea;
end BusTestControllerNonExp;
```

5.3 Validation of the Use of Expandable Connectors

The use of expandable connectors (control bus) is validated in case of a complex controller (`Buildings.Controls.OBC.ASHRAE.G36_PR1.AHUs.MultiZone.VAV.Controller`).

The validation is performed:

- with Dymola (Version 2020, 64-bit, 2019-04-10) and JModelica (revision numbers from svn: JModelica 12903, Assimulo 873);
- first with a single instance of the controller and then with multiple instances corresponding to different parameters set up (see validation cases of the original controller `Validation.Controller` and `Validation.ControllerConfigurationTest`),
- with nested expandable connectors: a top-level control bus composed of a first sub-level control bus for control output variables and another for control input variables.

Simulation succeeds for the two tests cases with the two simulation tools. The results comparison to the original test case (without control bus) is presented in Fig. 5.7 for Dymola.

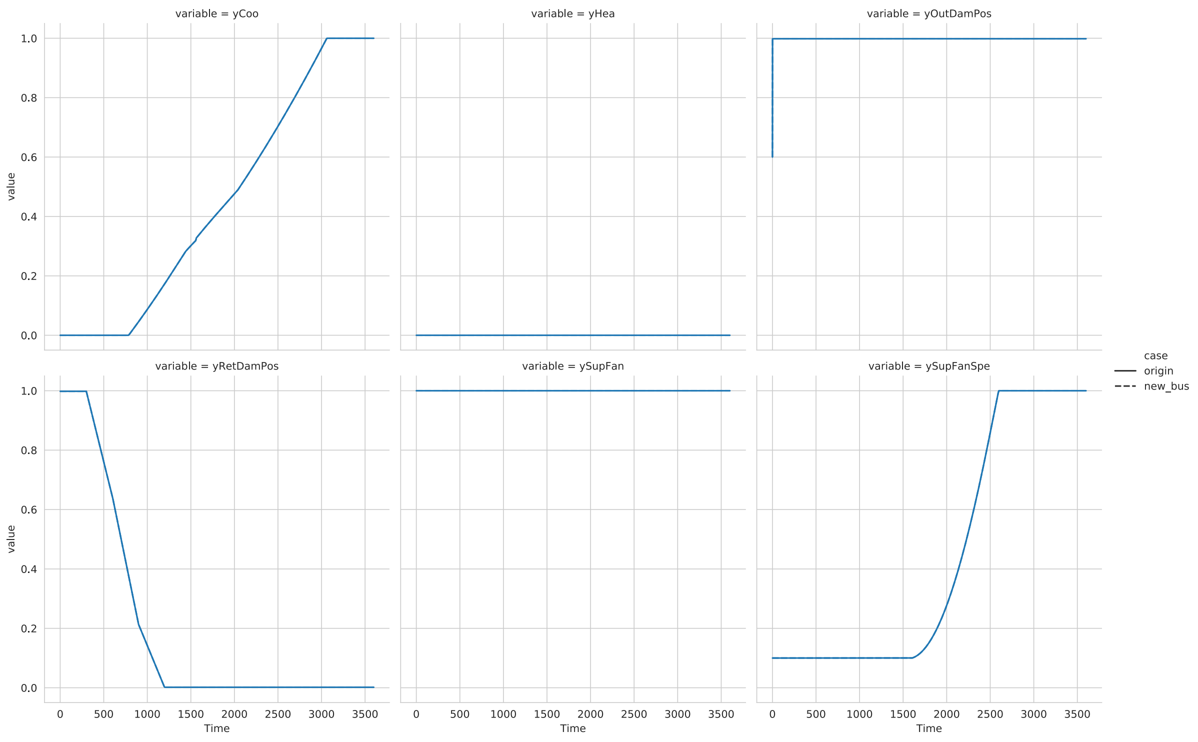


Fig. 5.7: *G36 AHU controller model: comparison of simulation results (Dymola) between implementation without (`origin`) and with (`new_bus`) expandable connectors*

Chapter 6

Glossary

To be updated.

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.

Chapter 7

Acknowledgments

To be updated.

Chapter 8

References

[Bri] *Brick – A Uniform Metadata Schema for Buildings*. URL: https://brickschema.org/#home.

[Hay] *Project Haystack 4 – An Open Source initiative to streamline working with IoT Data*. URL: https://project-haystack.dev.

[Mod17] *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification, Version 3.4*. Modelica Association, April 2017. URL: https://www.modelica.org/documents/ModelicaSpec34.pdf.

[LBNL19] *OpenBuildingControl Specification*. LBNL, 2019. URL: https://obc.lbl.gov/specification/index.html.

Index