LinkageJS Requirements Specification

Contents

Chapter 1

Preamble

1.1   Purpose of the Document

This document specifies the requirements for LinkageJS software.

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

Chapter 2

Process Workflow

To be updated.

Chapter 3

Requirements

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 3.3.

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 relies on client side JS code with minimal dependencies and is 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 based on system calls,
  - 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 – with access to the local file system through the file system 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 needed 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

Table 3.1: Requirements for software compatibility

Feature	Support
Platform (minimum version)	Windows (10), Linux Ubuntu (16.04), OS X (10.10)
Mobile device & responsive design ?	iOS, Android?
Web browser	Chrome, Firefox, Safari

3.1.3 UI Visual Structure

See figure Fig. 3.2:

- Left panel: library navigator
- Main panel: diagram view of the model
- Right panel:
  - Configuration widget
  - Connections widget
  - Annotations widget
  - Parameters widget
- Menu bar
- Bottom panel: console

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			

Table 3.2 – continued from previous page

Feature	V0	V1	Comment
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 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	
Text editor	N	O	
Export control points summary	R		Relies on LBNL 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 numref:*screen_schematics_modelica*. Relies on LBNL module CDL to Word translator.
Import/Export data sheet	P	R	Additional module to 1) generate a file in CSV (or Excel) format from the configuration data (V0) 2) populate the configuration data based on a file input in CSV (or Excel) 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 ol 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.

Table 3.2 – continued from previous page

Feature	V0	V1	Comment
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		
Icon layer	O	R	
Version checking and upgrade	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		
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.

Table 3.2 – continued from previous page

Feature	V0	V1	Comment
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).
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*.

---

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

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

- a `connect` equation with the references to the two connectors is 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 is 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.

3.4 Configuration Widget

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[2]. CtrlSpecBuilder is a tool widely used in the HVAC controls industry, which typically provides the same kind of functionality.

There are three 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.

The input form is provided by the template developer (e.g. LBNL) 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,

---

[2] From Taylor Engineering: "For standard systems, it might be possible to simply include in their specifications a table of ASHRAE Guideline 36 sequences with check boxes for the paragraph numbers that are applicable to their project."

- 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. 3.1 and Fig. 3.2.

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. 3.1: *Configuration widget – Configuring a new model*

The envisioned data structure supporting this logic is illustrated in Listing 3.1 (pseudo code) where:

- the placement coordinates are provided relatively to a simplified grid, see Fig. 3.3, 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. 3.20: the upper-level model including the equipment and controls models is the ultimate output of the configuration widget (see Fig. 3.2 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:
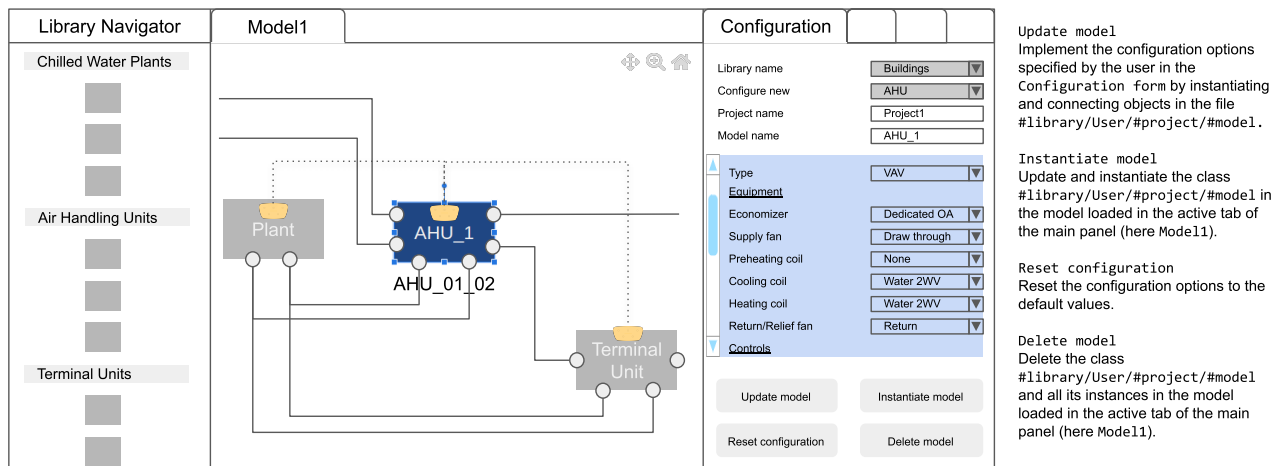
This is the view for the `Configuration` panel if:
- one object is selected in the main panel,
- and the corresponding class contains a model annotation `__Linkage_data(...)` providing the configuration data in a JSON-serialized format (further referred to as #data).

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. 3.2: Configuration widget – Configuring an existing model

- 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 would 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.)
- Providing a reference guideline for the controls specification conditionally disables all controls options that do not comply with that guideline.
- Parameters specified in 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.
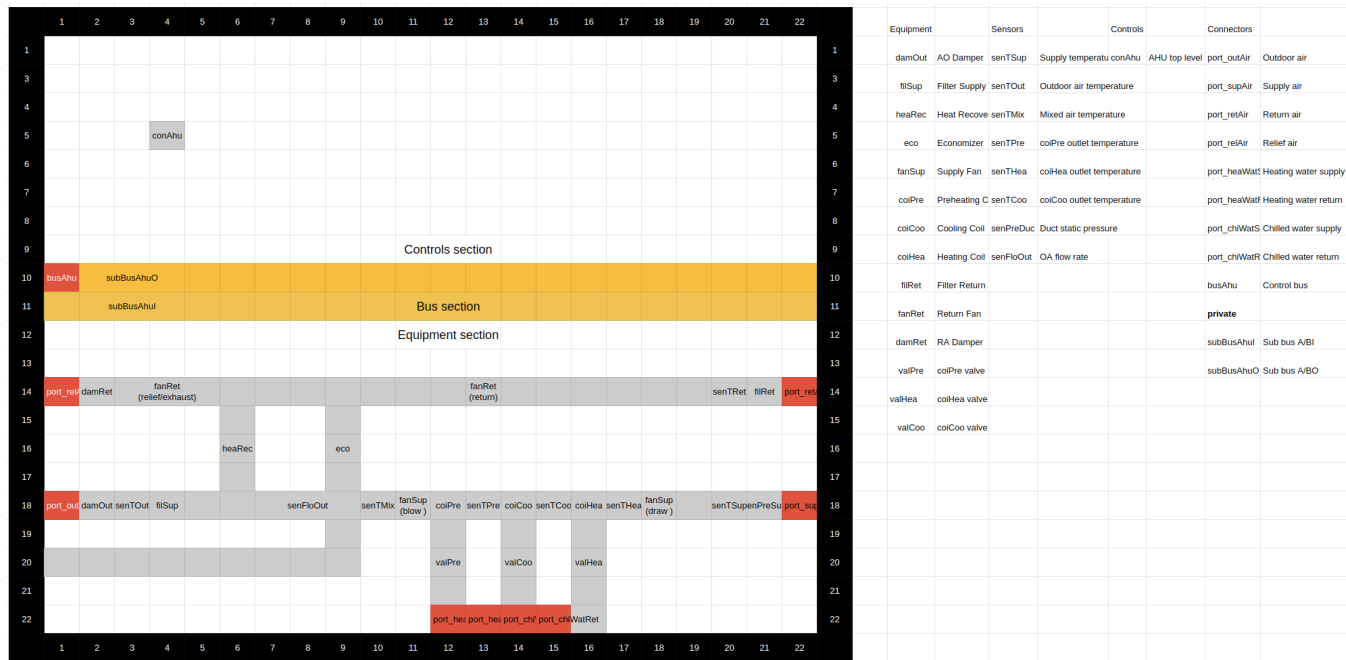
Fig. 3.3: *Simplified grid providing placement coordinates for all objects to be instantiated when configuring an AHU model*

Listing 3.1: 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": "Text",
        "value": "AHU_#i"
    },

    "type": {
        "$id": "#type",
        "description": "Type of AHU",
        "widget": "Dropdown",
        "options": ["VAV", "DOA", "Supply only", "Exhaust only"],
        "value": "VAV"
    },

    "fluid_path": [
        {
            "$id": "#air_supply",
            "direction": "horizontal",
            "orientation": "right",
            "medium": "Buildings.Media.Air"
        },
        {
            "$id": "#air_return",
            "direction": "horizontal",
            "orientation": "left",
            "medium": "Buildings.Media.Air"
        }
    ],

    "equipment": [
        {
            "$id": "#heaRec",
            "description": "Heat recovery",
            "ui::widget": "Dropdown",
            "ui::widget::enabled": "#type.value == 'DOA'",
            "options": ["None", "Fixed plate", "Enthalpy wheel", "Sensible wheel"],
            "value": "None",
            "model": [
                null,
                "Buildings.Fluid.HeatExchangers.PlateHeatExchangerEffectivenessNTU",
                "Buildings.Fluid.HeatExchangers.EnthalpyWheel",
                "Buildings.Fluid.HeatExchangers.EnthalpyWheel(sensible=true)"
            ],
            "icon_transformation": "flipHorizontal",
            "placement": [18, 6],
            "connectors": {
                "port_a1": "air_return_inlet", "port_a2": "air_supply_inlet", "port_
→b1": "air_return_outlet", "port_b2": "air_supply_outlet"
            }
```

```
            },
            {
                    "$id": "#eco",
                    "description": "Economizer",
                    "ui::widget": "Dropdown",
                    "ui::widget::enabled": "#type.value == 'VAV'",
                    "options": ["None", "Dedicated OA damper", "Common OA damper"],
                    "value": "None",
                    "model": [
                            null,
                            "Buildings.Fluid.Actuators.Dampers.MixingBoxMinimumFlow",
                            "Buildings.Fluid.Actuators.Dampers.MixingBox"
                    ],
                    "icon_transformation": "flipVertical",
                    "placement": [18, 9],
                    "connectors": {
                            "port_Out": "air_supply_out_inlet", "port_OutMin": "air_supply_min_
↪inlet", "port_Sup": "air_supply_outlet",
                            "port_Exh": "air_return_outlet", "port_Ret": "air_return_inlet"
                    }
            },
            {
                    "$id": "#V_flowOut_nominal",
                    "description": "Nominal outdoor air volumetric flow rate",
                    "ui::widget": "Input real",
                    "ui::widget::enabled": "#eco.value != 'None'",
                    "value": 0,
                    "unit": "m3/h"
            },
            {
                    "$id": "#fanSup",
                    "description": "Supply fan",
                    "ui::widget": "Dropdown",
                    "ui::widget::enabled": "#type.value != 'Exhaust only'",
                    "options": ["None", "Draw through", "Blow through"],
                    "value": "Draw through",
                    "model": "Buildings.Fluid.Movers.SpeedControlled_y(m_flow_nominal=(#air_
↪supply.medium).rho_default / 3600 * #V_flowSup_nominal.value)",
                    "icon_transformation": null,
                    "placement": [null, [18, 11], [18, 18]],
                    "fluid_path": "air_supply"
            },
            {
                    "$id": "#V_flowSup_nominal",
                    "description": "Nominal supply air volumetric flow rate",
                    "ui::widget": "Input real",
                    "ui::widget::enabled": "#fanSup.value != 'None'",
                    "value": 0,
                    "unit": "m3/h"
```

```
        },
        {
                "$id": "#fanRet",
                "description": "Return/Relief fan",
                "ui::widget": "Dropdown",
                "ui::widget::enabled": "#type.value != 'Supply only'",
                "options": ["None", "Return", "Relief"],
                "value": "Relief",
                "model": [
                        null,
                        "Buildings.Fluid.Movers.SpeedControlled_y((#air_return.medium).rho_
↪default / 3600 * #V_flowRet_nominal.value)",
                        "Buildings.Fluid.Movers.SpeedControlled_y(m_flow_nominal=(#air_
↪return.medium).rho_default / 3600 * (#V_flowRet_nominal.value - #V_flowSup_nominal.value +
↪#V_flowOut_nominal.value))"
                ],
                "icon_transformation": "flipHorizontal",
                "placement": [null, [14, 13], [14, 4]],
                "fluid_path": "air_return"
        }
        {
                "$id": "#V_flowRet_nominal",
                "description": "Nominal return air volumetric flow rate",
                "ui::widget": "Input real",
                "ui::widget::enabled": "#fanRet.value != 'None'",
                "value": 0,
                "unit": "m3/h"
        }
    ],

    "controls": [
        {
                "$id": "#conAHURef",
                "description": "Reference guideline for control sequences",
                "ui::widget": "Dropdown",
                "options": ["ASHRAE G36"],
                "value": null
        },
        {
                "$id": "#conAHUOpt",
                "description": "Optimal start up",
                "ui::widget": "Checkbox",
                "value": false
        },
        {
                "$id": "#conAHUDemLim",
                "description": "Demand limit set point adjustment",
                "ui::widget": "Checkbox",
                "value": false
```

```
            },
            {
                    "description": "Supply fan control",
                    "ui::widget": "Text"
            },
            {
                    "$id": "#conFanSupStaSto",
                    "description": "Supply fan start/stop control",
                    "ui::widget": "Dropdown",
                    "ui::widget::enabled": "#fanSup.value != 'None'",
                    "options": ["On-Off", "Static Pressure Control"],
                    "ui::widget::option::disabled": ["#conAHURef.value == 'ASHRAE G36'", ""],
                    "value": "if #conAHURef.value == null then 'On-Off' elseif #conAHURef.
→value == 'ASHRAE G36' then 'Static Pressure Control'"
            },
            {
                    "$id": "#resPreStaSet",
                    "description": "Static pressure set point reset",
                    "ui::widget": "Dropdown",
                    "ui::widget::enabled": "#fanSup.value != 'None'",
                    "options": ["None", "T&R"],
                    "ui::widget::option::disabled": ["#conAHURef.value == 'ASHRAE G36'", ""],
                    "value": "if #conAHURef.value == null then 'None' elseif #conAHURef.value␣
→== 'ASHRAE G36' then 'T&R'"
            },
            {
                    "description": "Supply Air Temperature Control",
                    "ui::widget": "Text"
            },
            {
                    "$id": "#resTSupSet",
                    "description": "Supply air temperature set point reset",
                    "ui::widget": "Dropdown",
                    "ui::widget::enabled": "#type.value != 'Exhaust only'",
                    "options": ["None", "OAT Reset", "OAT and T&R"],
                    "ui::widget::option::disabled": ["#conAHURef.value == 'ASHRAE G36'", "
→#conAHURef.value == 'ASHRAE G36'", ""],
                    "value": "if #conAHURef.value == null then 'None' elseif #conAHURef.value␣
→== 'ASHRAE G36' then 'OAT and T&R'"
            },
            {
                    "$id": "#numZon",
                    "description": "Number of served VAV boxes",
                    "ui::widget": "Input integer",
                    "ui::widget::enabled": "#resTSupSet.value == 'OAT and T&R'",
                    "value": null
            }
    ],
```

```
    "dependencies": [
            {
                "$id": "#port_outAir",
                "description": "Outside air port",
                "enabled": "#type.value != 'Exhaust only'",
                "model": "Modelica.Fluid.Interfaces.FluidPort_a(redeclare package Medium=
↪#air_supply.medium)",
                "placement": [18, 1],
                "fluid_path": "air_supply"
            },
            {
                "$id": "#port_supAir",
                "description": "Supply air port",
                "enabled": "#type.value != 'Exhaust only'",
                "model": "Modelica.Fluid.Interfaces.FluidPort_b(redeclare package Medium=
↪#air_supply.medium)",
                "placement": [18, 24],
                "fluid_path": "air_supply"
            },
            {
                "$id": "#senFloOut",
                "description": "Outdoor airflow measurement station",
                "enabled": "#ecoCon.value == 'ASHRAE G36'",
                "model": "Buildings.Fluid.Sensors.VolumeFlowRate(redeclare package Medium=
↪#air_supply.medium)",
                "placement": "if #eco.value == 'Dedicated OA damper' then [18, 5] else [20,
↪ 5]",
                "fluid_path": ""
            },
            {
                "$id": "#conAHU",
                "description": "AHU top level controller",
                "enabled": "#conFanSupRef.value == 'ASHRAE G36'",
                "model": "Buildings.Controls.OBC.ASHRAE.G36_PR1.AHUs.MultiZone.VAV.
↪Controller",
                "placement": "[5, 4]",
                "fluid_path": ""
            }
    ]
}
```

3.4.2 Fluid Connectors

The fluid connections (`connect` equations involving two fluid connectors) are generated based on :

- the coordinates of the components in the *diagram view* 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.

That tag can be automatically generated for components with the two following fluid ports (most common case):

- `Modelica.Fluid.Interfaces.FluidPort_a`: inlet
- `Modelica.Fluid.Interfaces.FluidPort_b`: outlet

For components with more than two fluid ports e.g. coil, the variable name could be used:

- `Modelica.Fluid.Interfaces.FluidPort_a port_a1`: primary fluid (liquid) inlet
- `Modelica.Fluid.Interfaces.FluidPort_a port_a2`: secondary fluid (air) inlet

However that logic fails when the ports correspond to the same medium e.g.:

- `Buildings.Fluid.Actuators.Dampers.MixingBox`: `port_Out, port_Exh, port_Ret, port_Sup`
- `Buildings.Fluid.Actuators.Valves.ThreeWayEqualPercentageLinear`: `port_1, port_2, port_3`
- `Buildings.Fluid.HeatExchangers.PlateHeatExchangerEffectivenessNTU`: `port_a1, port_a2, port_b1, port_b2`

So the following logic is considered:

1. Default mode
   - By default `port_a` and `port_b` will be tagged as `inlet` and `outlet` respectively.
   - An additional tag is provided at the component level to specify the fluid path e.g. `air_supply` or `air_return`.
   - All fluid connectors are then tagged by concatenating the previous tags e.g. `air_supply_inlet` or `air_return_outlet`.
2. Detailed mode
   - 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. 3.4. 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`.

The conversion script throws an exception if an instantiated class has some fluid ports that cannot be tagged 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: {connector_name1: air_supply_inlet, connector_name2: air_supply_outlet, ...}, ... }"))`. This is done when updating the model.

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, two additional inputs are needed:
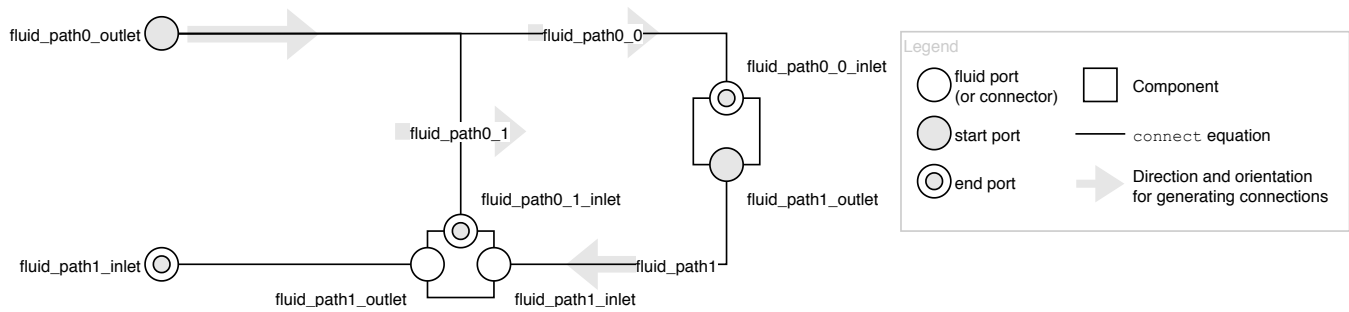
Fig. 3.4: *Connection scheme with a fluid junction not modeled explicitly e.g. three-way valve. In this example the bypass and direct branches are derived paths from* `fluid_path0` *which consists only in one connector.*

- The direction (horizontal or vertical) of the connection path.
- The orientation (up, down, right, left) of the connection path.

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

That information is stored in `__Linkage(Connect(paths="{fluid_path1: {direction: horizontal_or_vertical, orientation: right, ...}, ...}"))` 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 (horizontal, right): [connectors list]
    ├── fluid_path0_0 (inherited direction and orientation): [connectors list]
    └── fluid_path0_1 (inherited direction and orientation): [connectors list]
        ├── fluid_path0_1_0 (inherited direction and orientation): [connectors list]
        └── fluid_path0_1_1 (inherited direction and orientation): [connectors list]
├── fluid_path1 (horizontal, left): [connectors list]
├── fluid_path3 (vertical, up): [connectors list]
└── fluid_path4 (vertical, down): [connectors list]
```

- For each fluid path:
    - Order all the connectors in the connectors list according to the direction and orientation 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. 3.5.)
        * 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. 3.5 to Fig. 3.7 further illustrate the connection logic on different test cases.
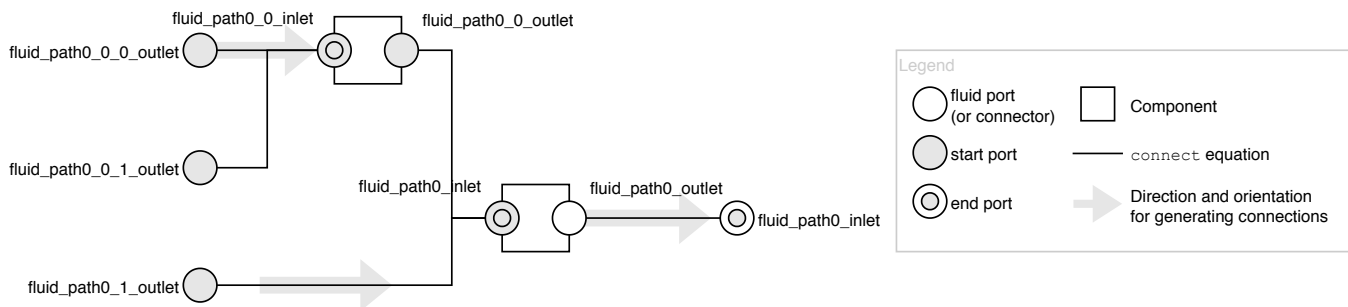

Fig. 3.5: *Connection scheme with nested fluid junctions not modeled explicitly*

3.4.3 Signal Connectors

3.4.3.1 General Principles

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

• a 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].

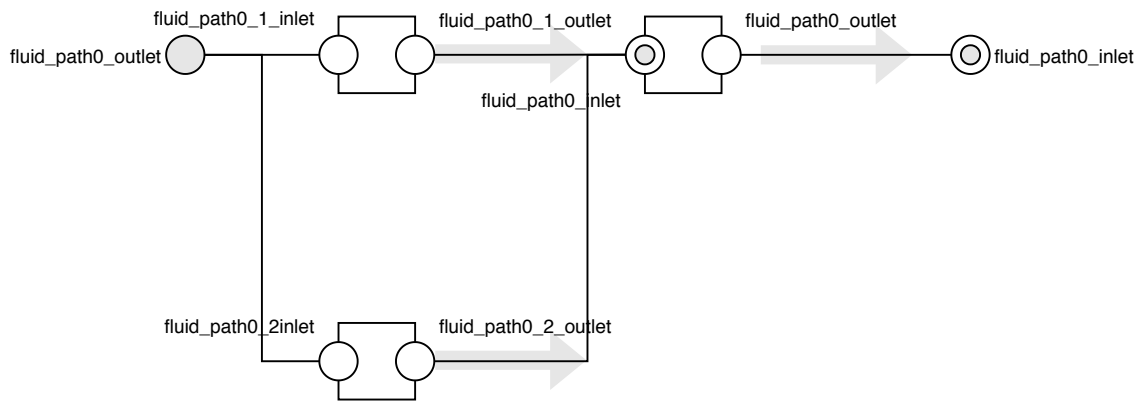The following features of the expandable connector are leveraged:

Fig. 3.6: *Connection scheme with fluid branches with identical directions e.g. AHU with dedicated outdoor air damper for economizer*
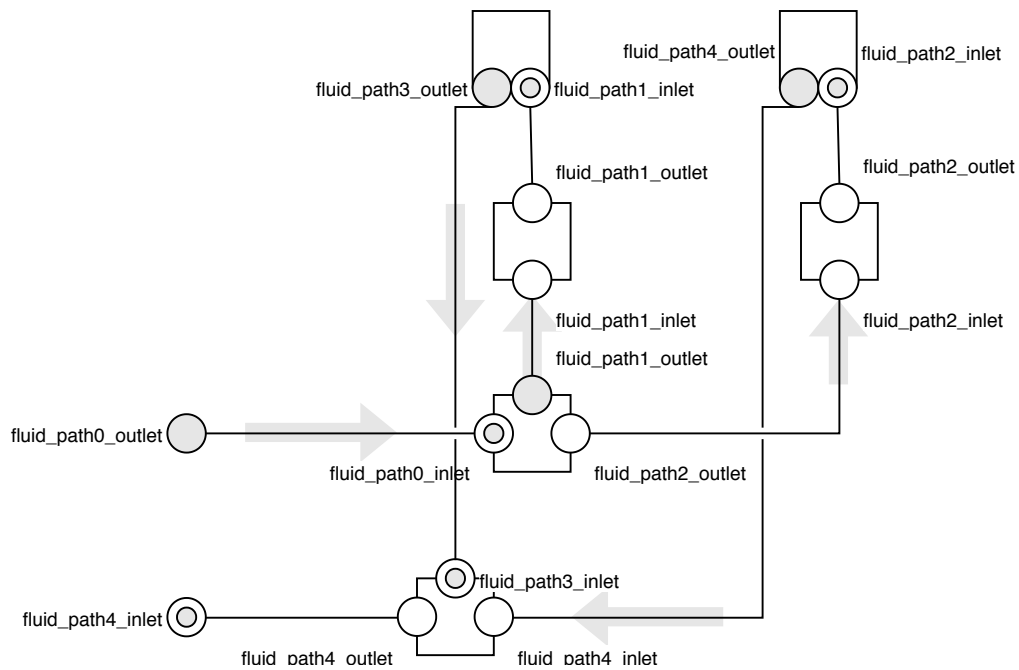

Fig. 3.7: *Connection scheme with fluid branches with different directions e.g. VAV duct system. Here a flow splitter is used to start several main fluid paths with a vertical connection direction.*

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 corresponding to the control sequences implemented for each system), 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 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. 3.8: *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;
```
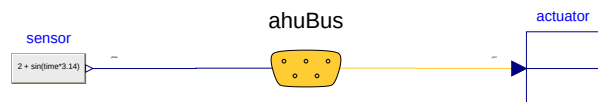

Fig. 3.9: *Minimal example illustrating the connection scheme with an expandable connector – Controlled component sublevel*

```
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);
       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.

For the configuration widget we will have predeclared variables with names allowing 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.

The control bus variables are used as "gateways" to stream values between the controlled and controller systems.

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 number of signals passing through busses has an impact on the number of equations and the simulation time.
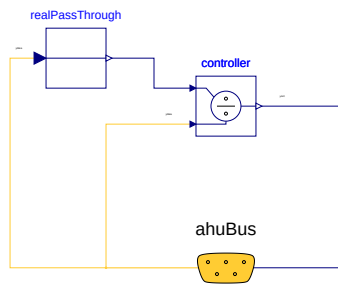
Fig. 3.10: *Minimal example illustrating the connection scheme with an expandable connector – Controller component sublevel*

```
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. 3.11: *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;
```

```
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;
```
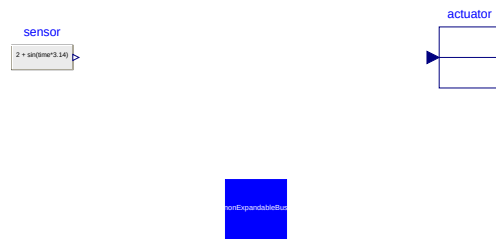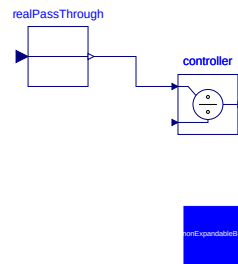
Fig. 3.12: *Minimal example illustrating the connection scheme with a standard connector – Controlled component sublevel*

```
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. 3.13: *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;
```

3.4.3.2 Validation and Additional Requirements

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.

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 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.

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. 3.14 for Dymola.
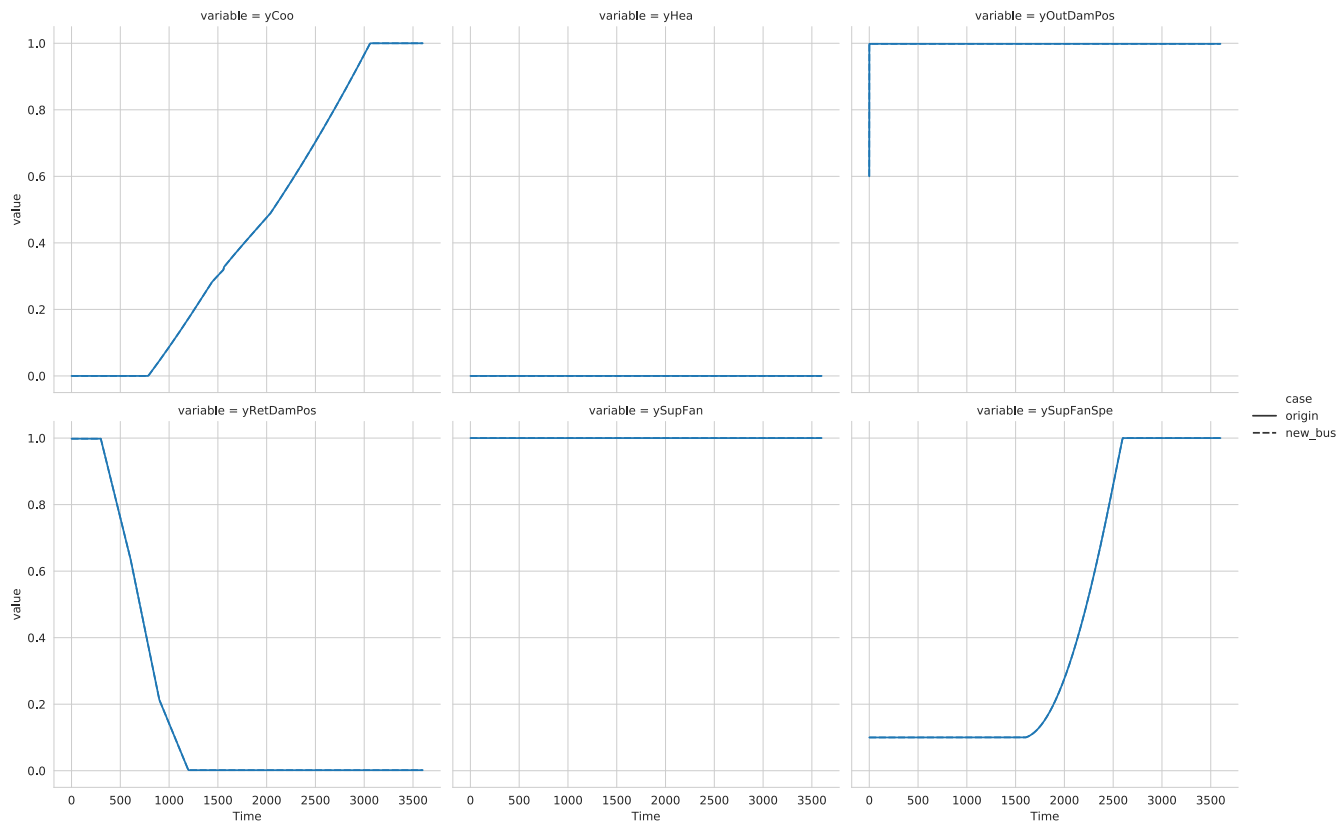

Fig. 3.14: *G36 AHU controller model: comparison of simulation results (Dymola) between implementation without (`origin`) and with (`new_bus`) expandable connectors*

3.4.3.3 Additional Requirements for the UI

Based on the previous validation case, Fig. 3.15 presents the Dymola pop-up window displayed when connecting the sub-bus of input control variables to the main control bus.

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

Fig. 3.15: Dymola pop-up window when connecting the sub-bus of input control variables (left) to the main control bus (right) – case of outside connectors

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:
  - 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.
    - Remark: 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. 3.16 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. 3.17 to Fig. 3.19 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. 3.19 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. 3.17 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	$\rightarrow$ `comp1.var1`	...
`var2` (present variable connected twice: default color)	x	O	`comp2.var1` $\rightarrow$ `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` $\rightarrow$	...
Add variable				
`var6` (variable elaborated by augmentation from *all instances* of the connector: light grey color)	O	O		...

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

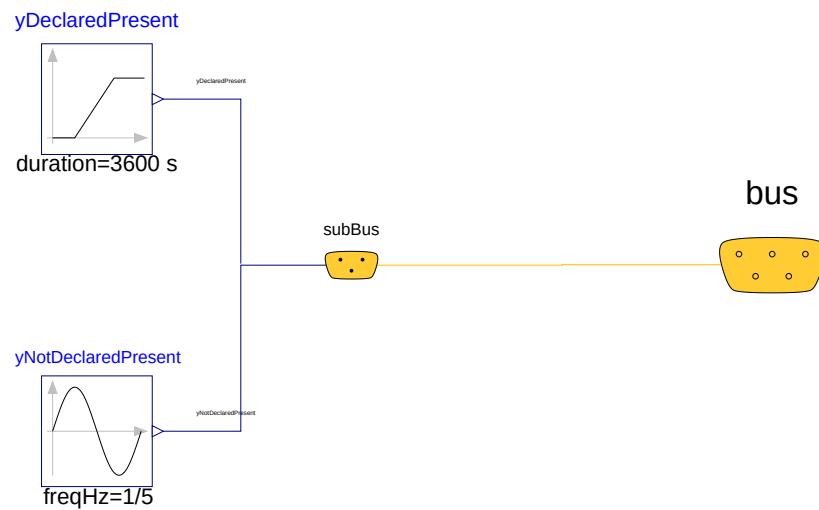## 3.7   OpenStudio Integration

To be updated.

Fig. 3.16: *Minimal example of sub-bus to bus connection illustrating how the bus variables are exposed in Dymola – case of outside connectors*

Fig. 3.17: *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. 3.18: 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. 3.19: Bus variables being exposed in case the bus is an inside connector

3.8 Interface with URBANopt GeoJSON

To be updated.

3.9 Encryption

See current standardization effort in #1868.

3.10 Licensing

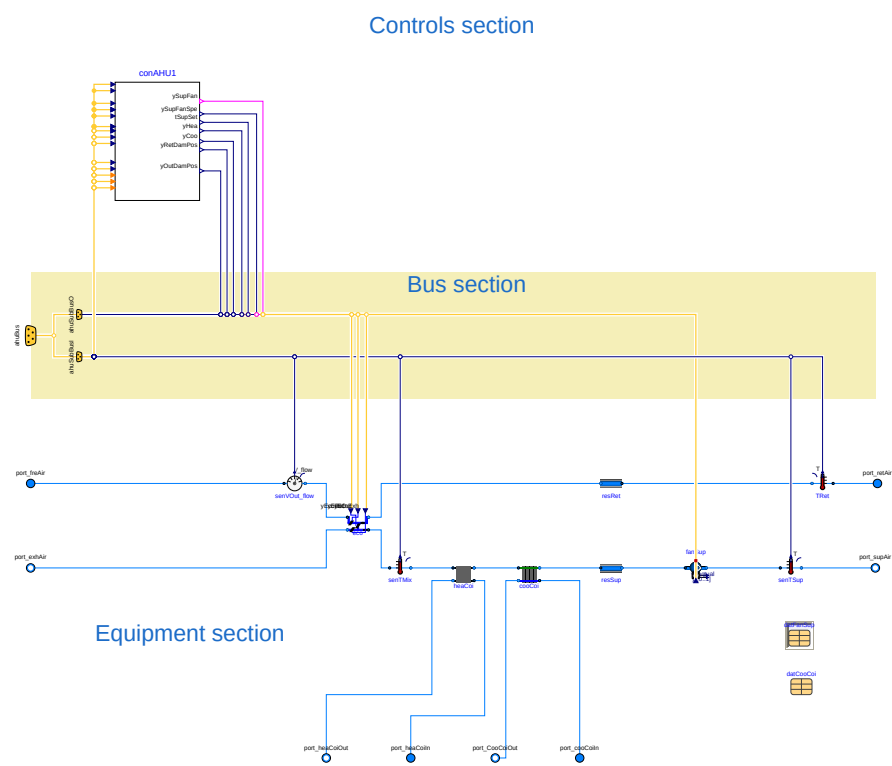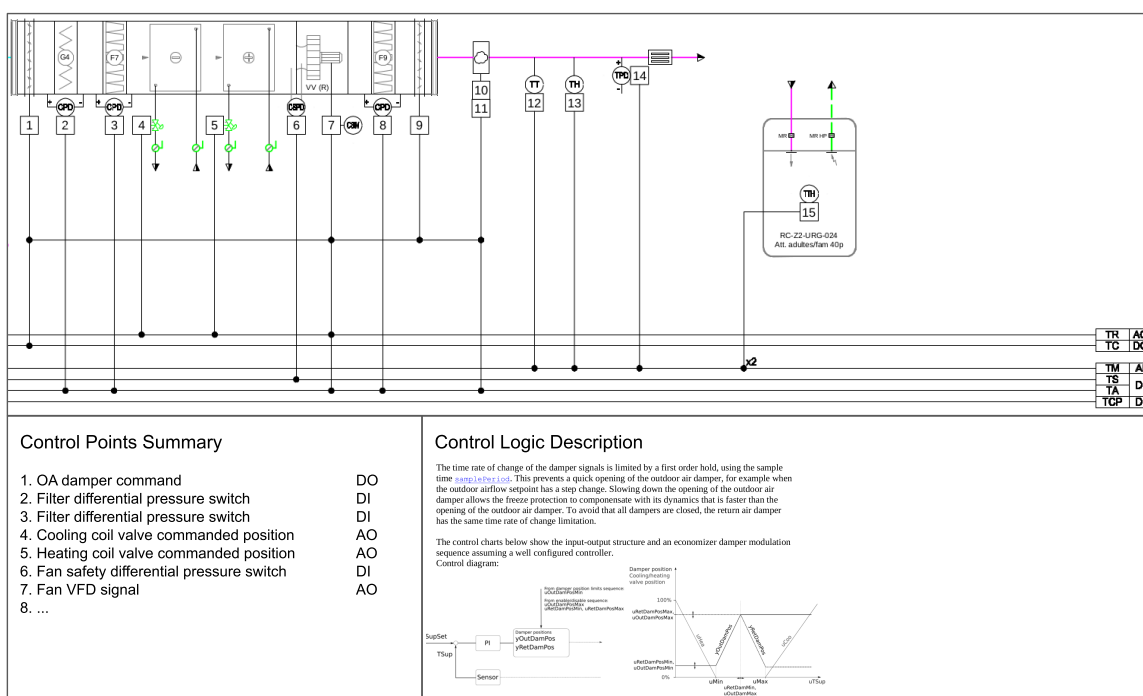To be updated cf. licensing strategy different for each integration target

Controls section

conAHU1

ySupFan
ySupFanSpe
tSupSet
yHea
yCoo
yRetDamPos
yOutDamPos

Bus section

ahuBus

Equipment section

port_freAir

senVOut_flow

port_exhAir

senTMix

heaCoi

cooCoi

resRet

TRet

port_retAir

resSup

fan_sup

senTSup

port_supAir

datCooCoi

port_heaCoiOut   port_heaCoiIn   port_CooCoiOut   port_cooCoiIn

Fig. 3.20: *Mockup of the schematics export – Input Modelica file*

Fig. 3.21: *Mockup of the schematics export – Output file (format to be specified: Word or PDF)*

Chapter 4

Software Architecture

This section describes the software architecture of LinkageJS.

To be updated.

Chapter 5

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 6

Acknowledgments

Chapter 7

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.

Index

A

B