LinkageJS Specification

Sep 09, 2019

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

Conventions

1. We write a requirement *shall* be met if it must be fulfilled. If the feature that implements a shall requirement is not in the final system, then the system does not meet this requirement. We write a requirement *should* be met if it is not critical to the system working, but is still desirable.
2. Text in bracket such as "[. . .]" denotes informative text that is not part of the specification.
3. Courier font names such as `input` denote variables or statements used in computer code.

Chapter 3

Process Workflow

a design engineer selects, configures, tests and evaluates the performance of a control sequence using building energy simulation (2), starting from a control sequence library that contains ASHRAE GPC 36 sequences, as well as user-added sequences (3), linked to a model of the mechanical system and the building (4). If the sequences meet closed-loop performance requirements, the designer exports a control specification, including the sequences and functional verification tests expressed in the Controls Description Language CDL (5). Optionally, for reuse in similar projects,

Chapter 4

Requirements

This section describes the functional, mathematical and software requirements.

4.1 General description

The software is a graphical user interface for editing Modelica models. In this respect it 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*,
- *to be updated*.

4.1.1 Software Compatibility

Table 4.1: Requirements for software compatibility

Feature	Support
Platform (minimum version)	Windows (10), Linux Ubuntu (16.04), OS X (10.10)
Web browser	Chrome, Firefox, Safari
Offline version	Running on local server with access to resources hosted on the file system

4.2 Modelica Graphical User Interface

4.2.1 Structure

See figure Fig. 4.2:

- Left panel: library navigator

- Main panel: diagram view of the model
- Right panel:
  - Configuration widget
  - Connection widget
  - Annotation widget
  - Parameters widget
- Menu bar
- Bottom panel: console

4.2.2 Functionalities

Table 4.2: Functionalities of the user interface – R: required, P: required partially, O: optional, N: not required

Feature	V0	V1	Comment
I/O			
Load `mo` file	R		Simple Modelica model or full package
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.
Simulate model	P		The software settings allow the user to specify a command for simulating the model with a third party Modelica tool e.g. JModelica.
Export simulation results	R		Export in the following format: `mat, csv`
Variables browser	P	R	Query selection of model variables based on regular expression (V0) or Brick tag [Bri] (V1)
Plot simulation results	N	O	
Text editor	N	O	
Object manipulation			
Vectorized instances	R		
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
Graphical features			
Support of Modelica graphical annotations	R		
Icon layer	O	O	
Draw shape, text box	O	R	
Customize connection lines	O	R	Color, width and line can be specified in the *Annotation Panel*
Hover information	R		Class path when hovering an object in the diagram view and tooltip for each GUI element

4.2.3 Generating Connections

When drawing a connection line between two connector icons:

- 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.

4.3 Configuration Widget

4.3.1 Functionalities

The configuration widget allows the user to generate a model of an HVAC system by filling up a simple form.

The form is provided by the developer as a JSON file or a dictionary within a Python script (easier to program the modeling logic). It provides for each field:

- the HTML widget and populating data to be used for 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. 4.1 and Fig. 4.2.

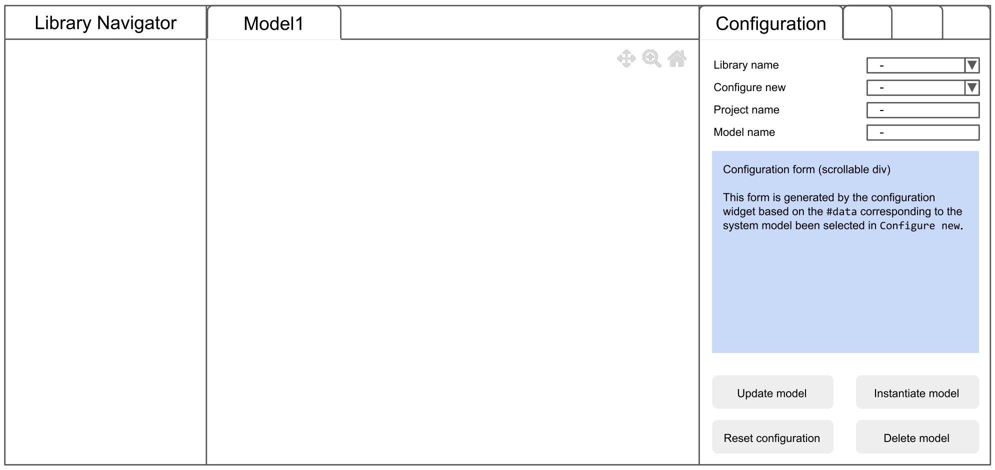
The envisioned schema supporting this logic is illustrated in Listing 4.1 where:

- the components referenced under the `equipment` name are connected together in one dedicated equipment model,
- the components referenced under the `controls` name are connected together in one dedicated controls model,
- the equipment and controls models are connected together by means of a *control bus*, see Fig. 4.6.

Listing 4.1: Partial example of the configuration data model for an air handling unit

```
{
        "system": {
                "description": "System type",
                "value": "AHU"
        },
```

(continues on next page)


Library Navigator
Model1
Configuration
Library name
Configure new
Project name
Model name
Configuration form (scrollable div)
This form is generated by the configuration widget based on the #data corresponding to the system model been selected in Configure new.
Update model
Instantiate model
Reset configuration
Delete model

`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.mo`.

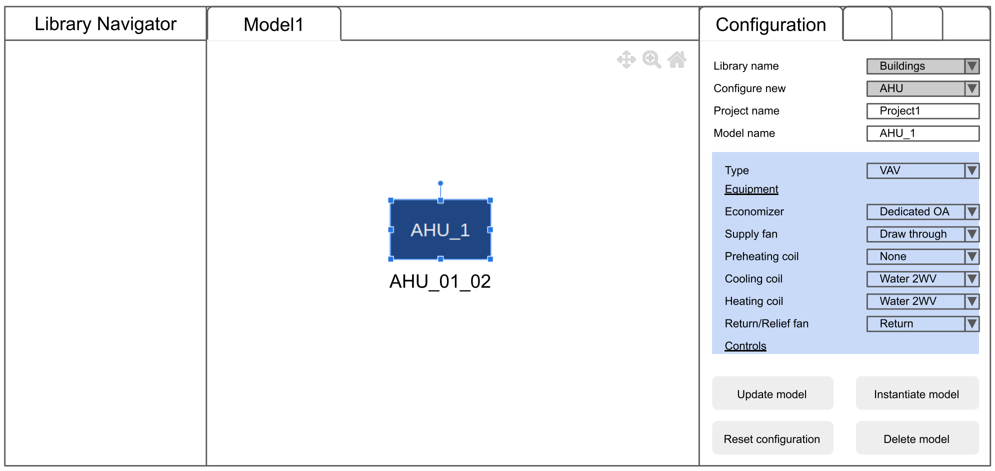
`Instantiate model`
Update and instantiate the class `#library/User/#project/#model` in the model loaded in the active tab of the main panel (here `Model1`).

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

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

When no object is selected 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 full 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. 4.1: Configuration widget – Configuring a new model



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

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

`Instantiate model`
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.mo`.

Fig. 4.2: Configuration widget – Configuring an existing model

(continued from previous page)

```
      "icon": "path of icon.mo",

      "diagram": {
            "configuration": [20, 20],
            "modelica": [[-120,-200], [120,120]]
      },

      "name": {
            "description": "Model name",
            "widget": "Text",
            "value": "AHU_#i"
      },

      "type": {
            "description": "Type of AHU",
            "widget": "Dropdown",
            "options": ["VAV", "DOA", "Supply Only", "Exhaust Only"]
      },

      "fluid_path": {
            "air_supply": [
                  {"start_port": "port_outAir"},
                  {"end_port": "port_supAir"},
                  {"direction": "horizontal"},
                  {"medium": "Buildings.Media.Air"}
            ]
      },

      "equipment": [
            {
                  "name": ["port_outAir", "port_supAir"],
                  "description": ["Outside air port", "Supply air port"],
                  "condition": [
                        "NOT",
                        {"#type": "Exhaust Only"},
                  ],
                  "model": ["Modelica.Fluid.Interfaces.FluidPort_a", "Modelica.Fluid.
↪Interfaces.FluidPort_b"]
                  "placement": [[12, 1], [12,20]]
                  "connect_tags": {"fluid_path": "air_supply"}
            },
            {
                  "name": "heaRec",
                  "description": "Heat recovery",
                  "widget": "Dropdown",
                  "condition": [
                        {"#type": "DOA"}
                  ],
                  "options": ["None", "Fixed plate", "Enthalpy wheel", "Sensible wheel"],
```

(continues on next page)

(continued from previous page)

```
                "value": "None",
                "model": [
                    null,
                    "Buildings.Fluid.HeatExchangers.PlateHeatExchangerEffectivenessNTU",
                    "Buildings.Fluid.HeatExchangers.EnthalpyWheel",
                    "Buildings.Fluid.HeatExchangers.EnthalpyWheel(sensible=true)"
                ],
                "icon_transformation": "flipHorizontal",
                "placement": [12, 9],
                "connect_tags": {"connectors": {
                    "port_a1": "air_return_inlet", "port_a2": "air_supply_inlet", "port_
↪b1": "air_return_outlet", "port_b2": "air_supply_outlet"
                }}
            },
            {
                "name": "eco",
                "description": "Economizer",
                "widget": "Dropdown",
                "options": ["None", "Separate dedicated OA dampers", "Single common OA␣
↪damper"],
                "condition": [
                    {"#type": "VAV"}
                ],
                "model": [
                    null,
                    "Buildings.Fluid.Actuators.Dampers.MixingBoxMinimumFlow",
                    "Buildings.Fluid.Actuators.Dampers.MixingBox"
                ],
                "icon_transformation": "flipVertical",
                "placement": [12, 6],
                "connect_tags": {"connectors": {
                    "port_Out": "air_supply_junction", "port_OutMin": "air_supply_
↪junction", "port_Sup": "air_supply_outlet",
                    "port_Exh": "air_return_outlet", "port_Ret": "air_return_inlet"
                }}
            },
            {
                "name": "supFan",
                "description": "Supply fan",
                "widget": "Dropdown",
                "options": ["None", "Draw through", "Blow through"],
                "value": "Draw through",
                "model": "Buildings.Fluid.Movers.SpeedControlled_y",
                "icon_transformation": null,
                "placement": [null, [16, 11], [16, 18]],
                "connect_tags": {"fluid_path": "air_supply"}
            },
            {
                "name": "retFan",
```

(continues on next page)

(continued from previous page)

```
                "description": "Return/Relief fan",
                "widget": "Dropdown",
                "options": ["None", "Return", "Relief"],
                "value": "Relief",
                "model": "Buildings.Fluid.Movers.SpeedControlled_y",
                "icon_transformation": "flipHorizontal",
                "placement": [null, [16, 11], [16, 18]],
                "connect_tags": {"fluid_path": "air_return"}
            }
    ],

    "controls": [

            {
                "description": "Economizer",
                "widget": "Dropdown",
                "condition": [
                    {"#equipment[id=economizer].value": "True"}
                ],
                "options": ["ASHRAE 2006", "ASHRAE G36"]
            }
    ],

    "parameters": [
            {
                "name": "V_flowSup_nominal",
                "description": "Nominal supply air volumetric flow rate",
                "value": 0,
                "unit": "m3/h"
            },
            {
                "name": "V_flowRet_nominal",
                "description": "Nominal return air volumetric flow rate",
                "value": 0,
                "unit": "m3/h"
            }
            {
                "name": "V_flowOut_nominal",
                "description": "Nominal outdoor air volumetric flow rate",
                "value": 0,
                "condition": [
                    {"#equipment[id=economizer].value": "True"}
                ],
                "unit": "m3/h"
            }
    ]
}
```

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 (note that some classes e.g. `FluidPort` can be instantiated as dependencies of others and have no corresponding input in the form);
- 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 have to be generated automatically based on the model structure.

4.3.2 Fluid Connectors

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

- the coordinates of the components in the diagram view,
- a tag applied to the components' connectors.

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 optional 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. 4.3. 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. To support that feature another connector tag value is needed: `junction`.
   - For 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_junction"}`

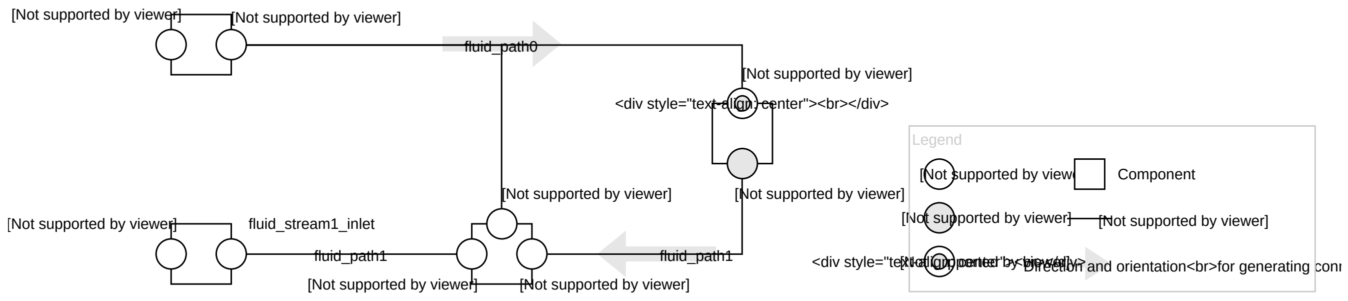

[Not supported by viewer]
[Not supported by viewer]
fluid_path0
[Not supported by viewer]
<div style="text-align: center"><br></div>
Legend
[Not supported by viewer]
Component
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
fluid_stream1_inlet
fluid_path1
fluid_path1
[Not supported by viewer]
[Not supported by viewer]
<div style="text-align: center"><br></div>
Direction and orientation<br>for generating con

Fig. 4.3: *Connection scheme with a fluid junction not modeled explicitly, using the connector tag* `junction`

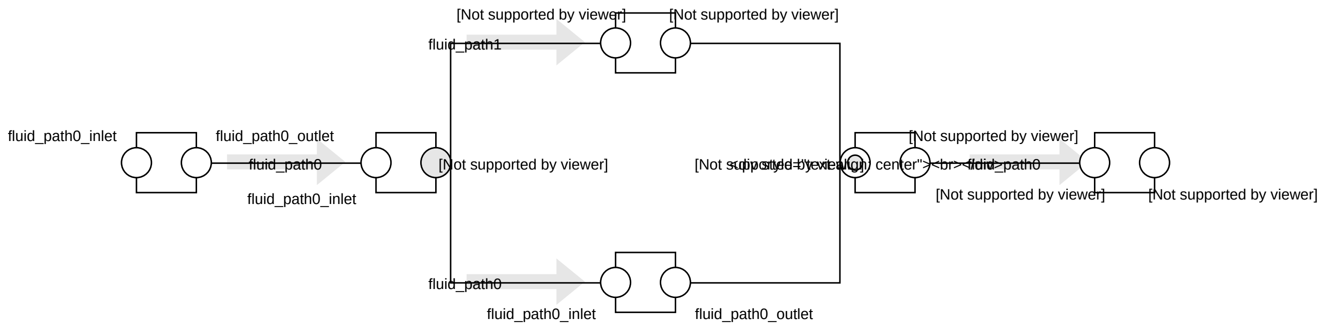

[Not supported by viewer]
[Not supported by viewer]
fluid_path1
fluid_path0_inlet
fluid_path0_outlet
fluid_path0
fluid_path0_inlet
[Not supported by viewer]
[Not supported by viewer]
<div style="text-align: center"><br></div>
fluid_path0
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
fluid_path0
fluid_path0_inlet
fluid_path0_outlet

Fig. 4.4: *Connection scheme with a fluid junction not modeled explicitly, using a dedicated fluid path*

The conversion script throws an exception if the 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_tags(Tags())` annotation 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:

1. The names of the start port and the end port for each fluid path.

   Note: Those ports may be part of a different fluid path see `fluid_path1` in Fig. 4.4.
   Furthermore this a stringent limitation of that approach: if the structure of the model is complex, the start or end port of a specific fluid path can be hard to determine due to intricate boolean conditions. Allowing multiple tags for a single connector could be a better solution?

2. The direction (horizontal or vertical) of the connection path.

Those inputs are stored in `__Linkage_connect(Direction(fluid_path1={start_connector_name, end_connector_name, horizontal_or_vertical}))`.

The connection logic is then as follows:

- List all the different fluid paths in `__Linkage_connect(Tags())` corresponding to each tuple `{fluid}_{path}` in all the connector tags `{fluid}_{path}_{port}`.
- For each fluid path:
  - Find the position of the objects corresponding to the start and end ports specified in `__Linkage_connect(Direction(fluid_path1={start_connector_name, end_connector_name}))`. Those are further referred to as start and end position.
  - Find the orientation (up, down, right, left) of the direction (horizontal, vertical) of the connection path by comparing the `x` (resp. `y`) coordinate values of the start and end position if the direction is horizontal (resp. vertical).

    Throw an exception if the orientation cannot be resolved due to identical coordinate values.

  - Order all the connectors belonging to that fluid path according to the orientation defined here above and based on the position of the corresponding objects with the constraint that for each object `inlet` has to be listed first and `outlet` last. Prepend / append that list with the start and end connectors.
  - Generate the `connect` equations by iterating on the ordered list of connectors as illustrated in the pseudo code below. And generate the connection path and the corresponding graphical annotation:

```
i = 1
while i < n
j = i + 1
if type(ordered_connector[i]) == "junction"
      while type(ordered_connector[j]) == "junction"
            connect(ordered_connector[i], ordered_connector[j])
            annotation(Line(points=create_new_path(ordered_connector[i], ordered_
↪connector[j])))
```

(continues on next page)

(continued from previous page)

```
            j = j + 1
      i = j
else
      connect(ordered_connector[i], ordered_connector[j])
      annotation(Line(points=create_new_path(ordered_connector[i], ordered_
↪connector[j])))
      i = j + 1
```

The only valid connections are `junction` with `junction` and `outlet` with `inlet`: throw an exception otherwise.
Fig. 4.5 further illustrates the logic for connecting `junction` ports.

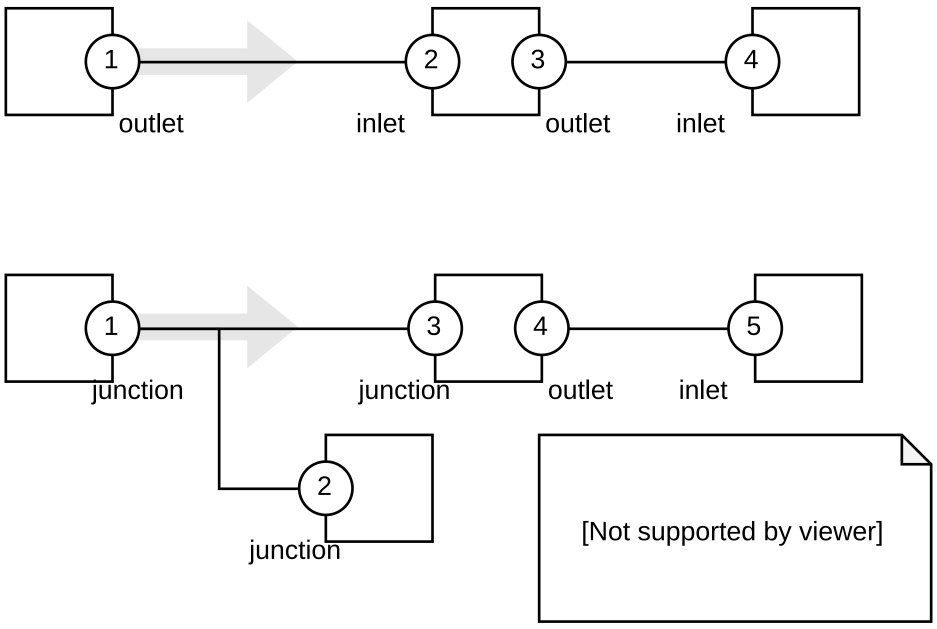


Fig. 4.5: Logic of ports connection in case of `inlet` and `outlet` ports (top) and `junction` ports (bottom)

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`.
- A same fluid path does not necessarily imply a uniform flow rate.
- Among the multiple connectors which are part of the same junction, only one can have a paired `outlet` connector (part of the same component) within the same fluid path and the corresponding component must be placed further downstream. There is always the modeling alternative which consists in representing a junction by introducing a new fluid path as in Fig. 4.4.

4.3.3 Signal Connectors

Generating the `connect` equations for signal variables relies on string matching with the names of the variables declared in a a so-called *control bus* which has the type of an expandable connector type, see *§9.1.3 Expandable Connectors* in [Mod17].

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. 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 a variable 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. Possibly present but not connected variable is considered not declared: 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 expanded 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 outputs a control variable equal to one.
- The same model is first implemented with an expandable connector and then with a standard connector.

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

```
model BusTestControlledExp
Modelica.Blocks.Sources.RealExpression sensor(y=2 + sin(time*3.14));
```

(continues on next page)

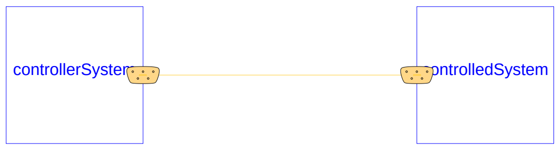


Fig. 4.6: *Minimal example illustrating the connection scheme with an expandable connector – Top level*

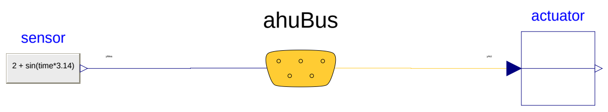


Fig. 4.7: *Minimal example illustrating the connection scheme with an expandable connector – Controlled component sublevel*

(continued from previous page)

```
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. measured 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.

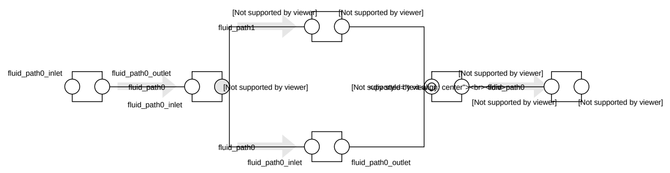

[Not supported by viewer]
[Not supported by viewer]
fluid_path1
fluid_path0_inlet
fluid_path0_outlet
fluid_path0
fluid_path0_inlet
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
fluid_path0
fluid_path0_inlet
fluid_path0_outlet

Fig. 4.8: 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;
```

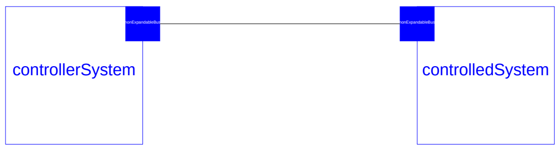


Fig. 4.9: Minimal example illustrating the connection scheme with a standard connector – Top level

```
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
```

(continues on next page)

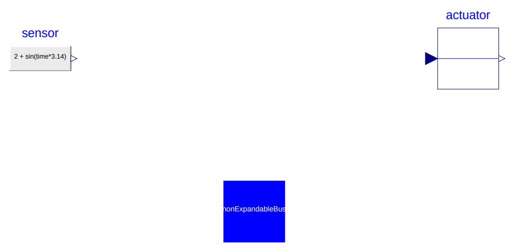


Fig. 4.10: Minimal example illustrating the connection scheme with a standard connector – Controlled component sublevel

(continued from previous page)

```
      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;
```

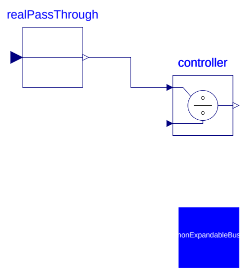


Fig. 4.11: Minimal example illustrating the connection scheme with a standard connector – Controller component sublevel

```
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;
```

The algorithm is as follows:

-

Chapter 5

Issues

We have a linked modelica model residing on disk. When loading that model, LinkageJS must be able to:

- identify which object and `connect` statement can be modified with the template script: declaration/statement annotation `__Linkage_modify=true`
- generate the JSON configuration file:
  - automatically from the model structure? Non working examples:
  - Supply fan/Draw through: if the user has modified the `Placement` we have no one-to-one correspondance with JSON file. Also relying on the `connect` statements involving the object seems to complex.

Chapter 6

Questions

- Validation upon submit (export/generate) VS real-time
- Routine to add sensors for control sequences
- Routine to add fluid ports: part of data model?

Choice of units: SI / IP

Launch simulation integrated

- At least compilation required to validate the model?
- For control sequence configuration the model may not need to be fully specified.

Visualize results: variable browser (with Brick/Haystack option similar as `re` option)

No icon layer: just diagram layer showing graphical objects, component icons, connectors and connection lines

Automatic medium propagation between connected components

- Expected as a future enhancement of Modelica standard: should we anticipate or wait and see?

Note: Brick and tagging

Set up parameters values like OS measures enable cf. electrical loads…

From Taylor Eng.

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.

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

With the current library design, it is necessary to explicitly select the medium model for each component in a circuit. This model is then propagated to the ports, and a Modelica translator will check that the quantity and unit attributes of connected interfaces are identical. Therefore, an error occurs, if connected FluidPorts do not have a medium with the same medium name. In the future, automatic propagation of fluid models through the ports will be introduced, but this still not possible with Modelica 3.0.

Chapter 7

Software Architecture

This section describes the software architecture of the LinkageJS tool and the functional verification tool. In the text below, we mean by *plant* the HVAC and building system, and by *control* the controls other than product integrated controllers (PIC). Thus, the HVAC or building system model may, and likely will, contain product integrated controllers, which will be out of scope for CDL apart from reading measured values from PICs and sending setpoints to PICs.

Chapter 8

Glossary

This section provides definitions for abbreviations and terms introduced in the Open Building Controls project.

Analog Value In CDL, we say a value is analog if it represents a continuous number. The value may be presented by an analog signal such as voltage, or by a digital signal.

Binary Value In CDL, we say a value is binary if it can take on the values 0 and 1. The value may however be presented by an analog signal that can take on two values (within some tolerance) in order to communicate the binary value.

Building Model Digital model of the physical behavior of a given building over time, which accounts for any elements of the building envelope and includes a representation of internal gains and occupancy. Building model has connectors to be coupled with an environment model and any HVAC and non-HVAC system models pertaining to the building.

Chapter 9

Acknowledgments

This research was supported by

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

Chapter 10

References

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

[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