Linkage Software Requirements Specification

V1.2 - Release after External Review

Mar 17, 2021

Contents

Chapter 1

Preamble and Conventions

This documentation specifies the requirements for Linkage software: *a graphical user interface for editing Modelica models of HVAC and control systems*.

The following convention is used throughout the specification.

> The words *must, must not, required, shall, shall not, should, should not, recommended, may, optional* in this document must be interpreted as described in RFC2119.

Chapter 2

Overview

The software to develop is a graphical user interface for editing Modelica models of HVAC and control systems.

The development targets two main categories of end users, associated with two main use cases.

1. Heating Ventilating and Air Conditioning (HVAC) engineers and designers who will utilize the tool to specify the controls of HVAC systems in commercial buildings. In terms of use case, we will refer to this category as the *control specification workflow*.
2. Building Energy Modeling (BEM) engineers who will utilize the tool to assess the energy use of HVAC systems based on a detailed representation of the equipment and controls that the tool will enable, and simulations that will be run using third party Modelica tools. In terms of use case, we will refer to this category as the *modeling and simulation workflow*.

The software relies on two main components.

1. A configuration widget supporting assisted modeling based on a simple HTML input form. This widget 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 layout and the selection of standard control sequences already transcribed in Modelica.
2. A graphical user interface for editing Modelica classes in a diagrammatic form.

We plan a phased development where

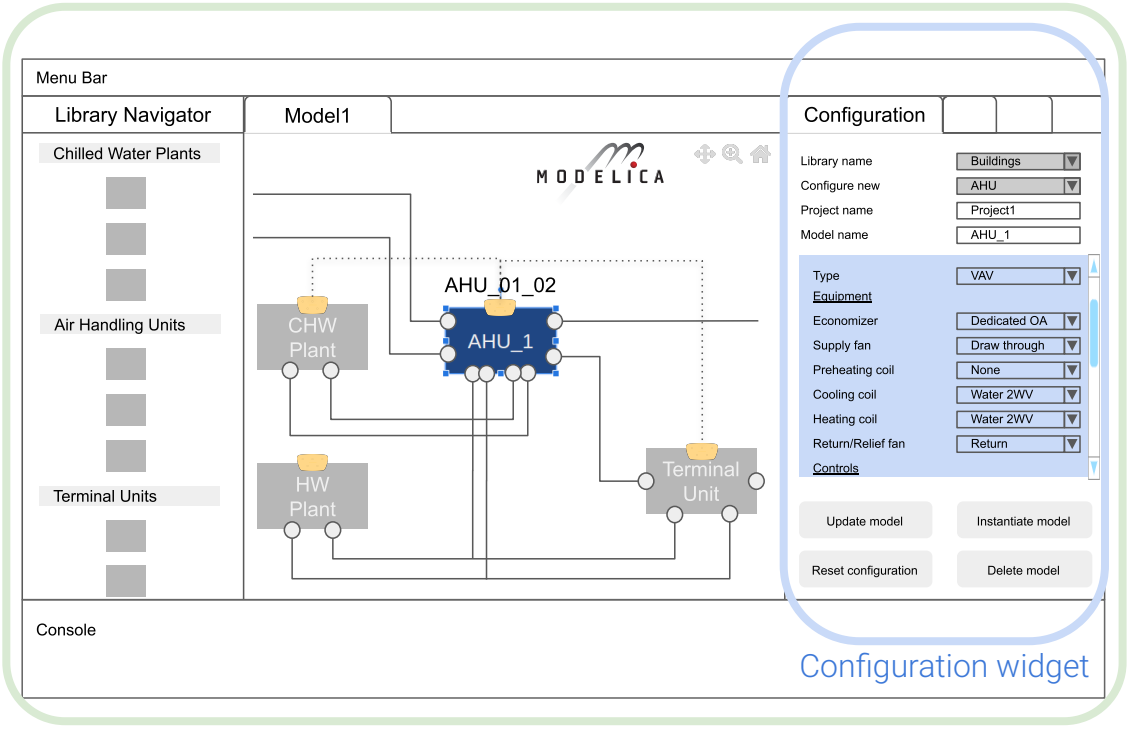
1. the configuration widget will be first implemented as part of Phase 1 and integrated into an existing graphical editor for Modelica—the *control specification workflow* is the prioritized use case in Phase 1,
2. the full-featured editor will be developed in a future phase, providing diagrammatic editing capabilities and integrating the configuration widget natively—the *modeling and simulation workflow* is the prioritized use case in the future phase.

Note: The current version of the specification is limited to Phase 1. Each part related to the full-featured editor is provided for informative purposes only.

Revision Note

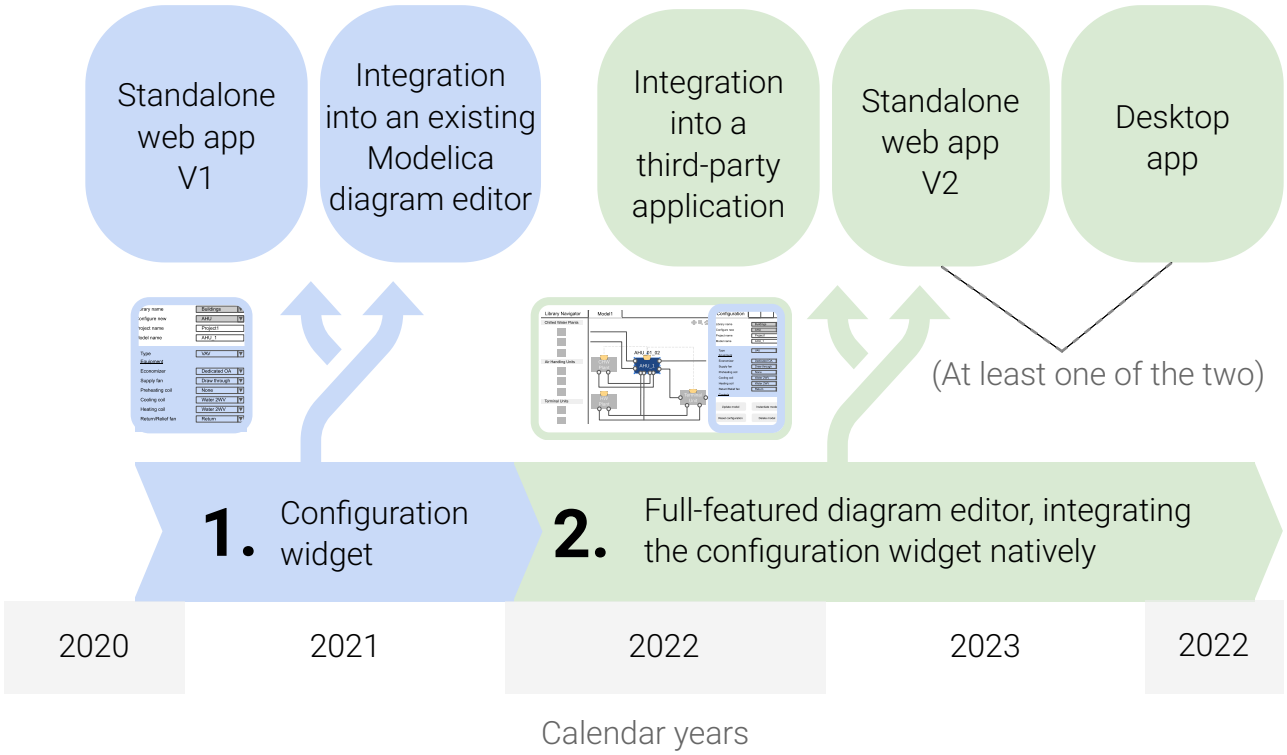
The third slide is modified to illustrate the configuration workflow based on Modelica.

What We Want to Develop

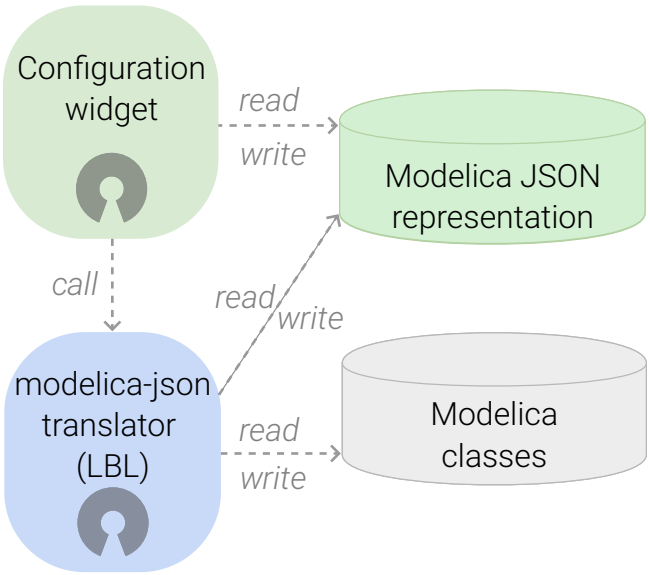


Full featured editor

In Two Phases



Requirement Overview



- Based on Modelica JSON schema and parser developed by LBL
- JS libraries can be leveraged

- Based on Modelica specification

- Client-side JS code with minimal dependencies
- Open source (3 or 4-clause BSD)
- Available as a free software

Full specification available at github.com/lbl-srg/linkage.js

Chapter 3

Requirements

3.1 General Description

3.1.1 Main Requirements

The following requirements apply to both the configuration widget (Phase 1 of the development) and the diagram editor (future phase of the development).

- The software must rely on client side JS code with minimal dependencies and must be built down to a single page HTML document (SPA).
- A widget structure is required that allows seamless embedding into:
  - a desktop app—with standard access to the local file system,
  - a standalone web app—with access to the local file system limited to Download & Upload functions of the web browser (potentially with an additional sandbox file system to secure backup in case the app enters an unknown state),
  - any third-party application with the suitable framework to serve a single page HTML document executing JS code—with access to the local file system through the API of the third-party application:
    * For the first development phase pertaining to the configuration widget, the third-party application for the widget integration is an existing graphical editor for Modelica. To demonstrate the feasibility of this integration, a proof of concept shall be developed, together with the documentation describing how this workflow can be accomplished. This will include the creation of a prototype where the host application may be an actual Modelica editor or a rudimentary emulator of such.
    * For the second development phase, the primary integration target is OpenStudio® (OS) while the widget to be integrated is now the full-featured editor (including the configuration widget). An example of a JS application embedded in OS is FloorspaceJS. The standalone SPA lives here: https://nrel.github.io/floorspace.js. FloorspaceJS may be considered as a reference for the development.

Revision Note

The following part is modified to make it clearer that the execution of the Modelica to JSON translator must be handled by the configuration widget.

- The core components parsing and generating Modelica classes must rely on JSON-formatted Modelica. For this purpose, LBL has developed a Modelica to JSON translator, based on the definition of two JSON schemas:
  - Schema-modelica.json validates the JSON files parsed from Modelica.
  - Schema-CDL.json validates the JSON files parsed from CDL (subset of Modelica language used for control sequence implementation).

  The software should call the Modelica to JSON translator for interfacing with native Modelica code.

3.1.2 Software Compatibility

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

Table 3.1: Requirements for software compatibility

Feature	Support
Platform (minimum version)	Windows (10), Linux Ubuntu (18.04), OS X (10.10)
Mobile device	The software may support iOS and Android integration though this is not an absolute requirement.
Web browser	Chrome, Firefox, Safari

3.1.3 UI Visual Structure

A responsive design is required.

Revision Note

The following part is modified to add the library navigator into the scope. See also Section 3.3.4.

A mockup of the UI for the full-featured editor is presented Fig. 3.1. For Phase 1, only the graphical features pertaining to the configuration widget in the right panel and the library navigator in the left panel (see Section 3.3.4) should be considered.

3.2 High-Level 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.

Note: CtrlSpecBuilder is a tool widely used in the HVAC industry for specifying control systems. It may be used as a reference for the development in terms of user experience minimal functionalities. Note that this software does not provide any Modelica modeling functionality.

The implementation of control sequences must comply with OpenBuildingControl requirements, see *§7 Control Description Language* and *§8 Code Generation* in [LBNL19]. Especially:

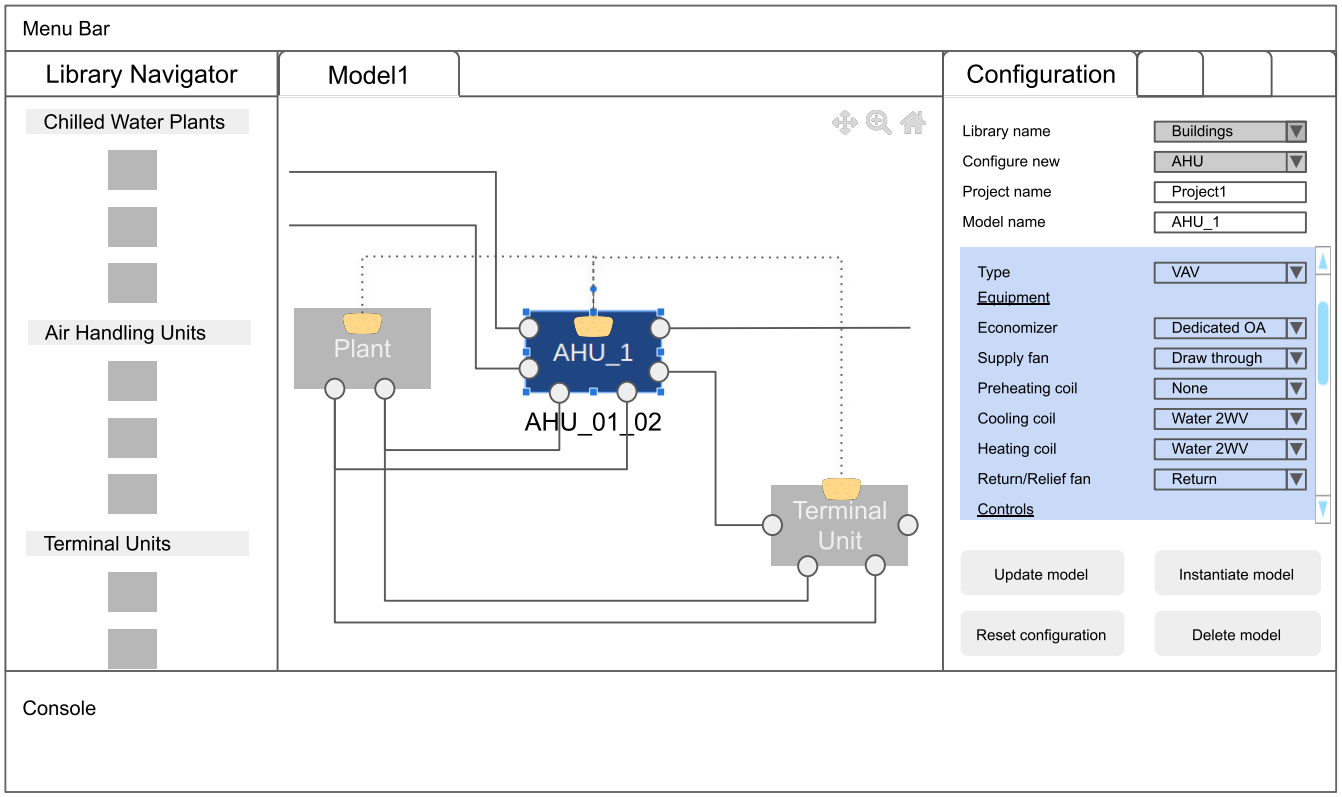


Fig. 3.1: UI Visual Structure

- It is required that the CDL part of the model can be programmatically isolated from the rest of the model in order to be translated into vendor-specific code (by means of a third-party translator).
- The expandable connectors (control bus) are not part of CDL specification. Those are used to connect
  - control blocks and equipment models within a composed sub-system model, e.g., AHU or terminal unit,
  - different sub-system models together to compose a whole system model, e.g., VAV system serving different rooms.

  This is consistent with OpenBuildingControl requirement to provide control sequence specification at the equipment level only (controller programming), not for system level applications (system programming).

Table 3.2 provides a list of the functionalities that the software must support. Phase 1 refers to the configuration widget, future work refers to the full-featured editor and is provided for informative purposes only.

Revision Note

The requirement for automatic medium propagation between connected components is removed. The requirement for executing the conversion scripts is removed.

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

Feature	Phase 1	Future	Comment
Main functionalities			See Section 3.1 for reference.
Diagram editor for Modelica classes	N	R	In the first phase, the configuration widget must be integrated into an existing diagram editor for Modelica, but only as a proof a concept of such an integration. Such an editor must be developed in the second phase.
Configuration widget	R	R	
I/O			
Modelica export	R	R	See Section 3.5
Documentation export	R	R	Control points, sequence of operation description (based on CDL to Word translator developed by LBL), and equipment schematics see Section 3.6
Modelica features			
Modelica code editor	N	R	Raw text editor with linter and Modelica specification checking upon save Note that this functionality requires translation and reverse translation of JSON to Modelica (those translators are developed by LBL).
Library version management	P	R	If a loaded class contains the Modelica annotation `uses` (e.g., `uses(Buildings(version="6.0.0"))` the software checks the version number of the stored library. If the version number does not match, the tool simply alerts the user of version incompatibility.

continues on next page

Table 3.2 – continued from previous page

Feature	Phase 1	Future	Comment
Path discovery	R	R	A routine to reconstruct the path or URL of a referenced resource within the loaded Modelica libraries is required. Typically a resource can be referenced with the following syntax `modelica://Buildings.Air.Systems.SingleZone.VAV`.
Object manipulation			
Avoiding duplicate names	R	R	When instantiating a component or assigning a name through the configuration widget, if the default name is already used in the class the software automatically appends the 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 apps is expected e.g. tranedesignassist.com.
Pan and zoom on mouse actions	N	R	
Help tooltip	R	R	Provide contextual help information to the user during each step of the workflow
Miscellaneous			
Internationalization	R	R	The software will be delivered in US English only, but it must be architectured to allow seamless integration of additional languages in the future. The choice between I-P and SI units must be possible. The mechanism supporting different units will be specified by LBL in a later version of this document.
User documentation	R	R	User manual of the GUI and the corresponding API Both an HTML version and a PDF version are required (may rely on Sphinx).
Developer documentation	R	R	All classes, methods, free functions, and schemas must be documented with an exhaustive description of the functionalities, parameters, return values, etc. UML diagrams should also be provided. At least an HTML version is required, PDF version is optional (may rely on Sphinx or VuePress).

3.3 Modelica-Based Templating

Revision Note

This paragraph is added. It replaces the former paragraph *Configuration Widget*.

The templates used by the configuration widget will be developed in Modelica.

A prototype of a template for an air handling unit is available in the feature branch `issue1374_templateVAV` of Modelica Buildings Library within the package `Buildings.Experimental.Templates.AHUs`.

To support the use of Modelica, the software must comply with the language specification [Mod17] for every aspect pertaining 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*,
  - In Phase 1, only literals, operations with literals and arrays constructors with literals need to be supported.
- the class names: see *Chapter 5 Scoping, Name Lookup, and Flattening*,
  - In Phase 1, short class names may be used in the templates and must be supported.
- the structure of packages: see *Chapter 13 Packages*,
  - In Phase 1, the tool generates a package `UserProject` which structure must comply with the specification.
- the annotations: see *Chapter 18 Annotations*.
  - In Phase 1, this is required since the UI relies on the annotations specified in Section 3.3.2 and Section 3.3.3.

Furthermore, in a control specification workflow only a subset of all the user inputs of a Modelica model are needed. For instance the type of medium, the nominal values of physical quantities, various modeling assumptions, etc. are only needed in the modeling and simulation workflow. Therefore, the configuration widget must include a mechanism to select the subset of user inputs that must be exposed in the UI. For this purpose a vendor-specific annotation should be used, see Section 3.3.3.

Eventually, the core components developed in Phase 1 must be reusable for the development of a full-featured parameter dialog widget in Phase 2, with the ability to switch between a control specification mode—with only a subset of the user inputs being exposed—and a modeling and simulation workflow—with the complete set of the user inputs being exposed.

3.3.1 Input Fields

Each input field described in this paragraph must be rendered in the UI with the description string provided at the declaration level. Optionally a software setting parameter (accessible to the user) should enable hiding the instance name, which is not needed in the control specification workflow.

TODO: Flesh out the requirement for highlighting missing parameter values (no default) or best-guess (or default?) values that need to be further specified (based on user selection?). How the latter is supported since a declaration annotation cannot be added/modified when redeclaring the enclosing class? Maybe through a class annotation referencing the instance name?

3.3.1.1 Validation

Values entered by the user must be validated *upon submit* against the Modelica language specification [Mod17]—syntax and type check, with an additional dimension check for arrays—and parameter attributes such as `min` and `max`. In Phase 1, only literals, operations with literals and arrays constructors with literals need to be supported.

A color code may be used to identify the fields with incorrect values that will be discarded upon save, and the corresponding error message may be displayed on hover.

TODO: Specify how manual edits of the control description can be added by the user. For instance, store them in a `String` variable for each part (such as supply temperature control, duct pressure control, etc.) of the control block, with a specific annotation so that they are highlighted in the generated documentation.

3.3.1.2 Variables

Each variable declared as a parameter without a `final` modifier must have a corresponding input field in the UI.

If the variable has the type Boolean a dropdown menu must be used and populated with `true`, `false` and `Unspecified` (no default). The latter option may be simply rendered as blank.

If the variable has the type of an enumeration a dropdown menu must be used. The dropdown menu must display the description string of each enumeration element and fallback to the name of each element. In addition an `Unspecified` (no default) option must be included, which may be simply rendered as blank.

If the variable is an array, a minimum requirement is that its value can be input using any array constructor specified in [Mod17]. Optionally a tailored input field for arrays may be made available *in addition*, for instance to allow the input of each array element within a cell of a table. However, the previous input logic based on a literal array constructor must always be available.

3.3.1.3 Record Type

All the declarations within a parameter of type record, and recursively of all the enclosed record parameters, must have a corresponding input field in the UI. An indentation may be used to show the different levels of composition.

3.3.1.4 Replaceable Keyword

Each declaration with the keyword `replaceable` and a choices annotation—either from the Modelica specification or a vendor-specific annotation, see Section 3.3.3—must have a corresponding dropdown menu in the UI. See Table 3.3 for additional requirements for how to populate the dropdown menu.

In addition, if the declaration corresponds to the instantiation of a model, a block, or a record, the previous logic must be applied recursively at each level of composition. An indentation may be used to show the different levels of composition.

Note that each variable may potentially be declared as replaceable. So the dropdown menu logic shall be not exclusive of the input field logic. Typically a user may specify the type through the dropdown menu and enter the value through the input field.

3.3.1.5 Final Keyword

The `final` prefix must result in no item being rendered in the UI for the corresponding declaration.

3.3.2 Parameter Dialog Annotations

The UI of the configuration widget must comply with the specification of the *parameter dialog annotations* from [Mod17] §18.7. Table 3.3 specifies how each feature of this part of the Modelica specification must be addressed.

Table 3.3: Parameter dialog annotations

Feature	Comment
Modelica annotations for the GUI	See [Mod17] §18.7 for reference.
`Dialog(tab|group)`	The UI must render the structure in groups and tabs as specified by this annotation. The groups may be collapsible with a button to expand or collapse all the groups.
`Dialog(enable)`	`Dialog(enable=false)` must result in no item being rendered in the UI for the corresponding declaration—as opposed to being only greyed out but still visible in Dymola.
`Dialog(showStartAttribute)`	The configuration widget should not display the input for the start value of a variable, this is not required in Phase 1.
`Dialog(colorSelector)`	This is not required in Phase 1.
`Dialog(loadSelector|saveSelector)` and `Selector(filter|caption)`	A mechanism to display a file dialog to select a file is required. The `filter` and `caption` attributes must also be interpreted as specified in [Mod17].
Annotation Choices for Modifications and Redeclarations	See [Mod17] §18.11 and §7.3.4 for reference.
`choicesAllMatching`	A discovery mechanism is required to enumerate all class subtypes (where subtyping is possible through multiple inheritances or nested function calls to a class constructor, such as `class A = B(...);`) given a constraining class. The enumeration must display the description string of the class and fallback to the simple name of the class. Once a selection is made by the user, the UI must display the description string of the redeclared class (as opposed to the literal redeclare statement in Dymola), with the same fallback logic as before. In Phase 1, this discovery mechanism will be implemented in the Modelica to JSON translator that will convert the `choicesAllMatching` annotation into a `choices(choice)` annotation when a specific flag is set to true. (Note that the tool in Phase 1 is an application tool, not a development tool: it works with static libraries that cannot be edited. So the list of allowable classes may be precomputed.)
`choices(choice)`	The enumeration must display the description string provided within each inner `choice` and fallback to the description string of the redeclared class, and ultimately fallback to the simple name of the redeclared class. Once a selection is made by the user, the UI must display the description string of the redeclared class (as opposed to the literal redeclare statement in Dymola), with the same fallback logic as before.

3.3.3 Vendor-Specific Annotations

Some vendor-specific annotations are required to facilitate the use of the templates. Those annotations are specified below using the lexical conventions from [Mod17] Appendix B.1.

Note that some annotations require to interpret some redeclare statements prior to compile time, in order to “visit” the redeclared classes and evaluate clauses like `coiCoo.typHex <> Types.HeatExchanger.None`—which Dymola

does not support, see for instance `annotation(Dialog(enable=typHex<>Types.HeatExchanger.None))` which has no effect. The UI must dynamically evaluate such clauses and update the parameter dialog accordingly.

3.3.3.1 Declaration Annotation

Each declaration may have a hierarchical vendor-specific annotation `"__Linkage" class-modification` that must be interpreted, with the following possible attributes.

`"choicesConditional" "(" [ "condition" "=" logical-expression "," choices-annotation ] { "," "condition" "=" logical-expression "," choices-annotation } ")"`

> Description: This annotation enables specifying a Modelica choices annotation (see [Mod17] §7.3.4) *conditionally* to any logical expression. Both the logical expression and the argument specified within the choices annotation must be valid in the variable scope of the class where they are used. This annotation takes precedence on Modelica `choices` and `choicesAllMatching` annotation. The UI must render the choices corresponding to the condition evaluated as true, with the same logic as the one described for the choices annotation in Table 3.3. If no condition is evaluated as true of if `choices()` is empty for the condition evaluated as true, no enumeration shall be rendered. If multiple conditions are evaluated as true, no enumeration shall be rendered and a message shall be printed to the standard error.
>
> Example: See the declaration `replaceable Economizers.None eco` in VAVSingleDuct.mo.

`"modification" "(" [ "condition" "=" logical-expression "," argument ] { "," "condition" "=" logical-expression "," argument } ")"`

> Description: This annotation enables a programmatic element modification or redeclaration based on any logical expression. Both the logical expression and the argument must be valid in the variable scope of the class where they are used. This annotation takes precedence on Modelica `choices` and `choicesAllMatching` annotation. No enumeration shall be rendered in the UI for any declaration containing this annotation.
>
> Example: See the declaration `replaceable record RecordEco = Economizers.Data.None` in VAVSingleDuct.mo.

`"display" "=" logical-expression`

> Description: This annotation enables displaying (or hiding) some input fields in the UI. It takes precedence on Modelica `Dialog(enable)` annotation, and must be interpreted with the same logic as the one described for the latter in Table 3.3. This annotation adds another level of flexibility to the built-in Modelica `Dialog(enable)` annotation, typically needed to render only a subset of the user input fields in a control specification workflow.

`"displayOrder" "=" UNSIGNED-INTEGER`

> Description: This annotation enables specifying how to order the input fields in the UI, independently from the declaration order. The input fields of all the declarations with this annotation should be positioned at the top of each group—or at the top of the parameter dialog if no group is specified—and ordered according to the value of `UNSIGNED-INTEGER`.
>
> *Is it really needed? Alternatively, and specifically for the templates, we could allow declaring parameters after instantiating replaceable components (those replaceable components must often appear at the top of a group as they often define the equipment type).*

3.3.3.2 Class Annotation

`"__LinkageTemplate"`

Description: This annotation identifies either a template or a package containing templates. It is used by the tool to simplify the tree view of the loaded libraries and only display the templates, see Section 3.3.4.

3.3.4 Class Manipulation and Workflow

From the original template classes, the configuration workflow enables generating classes representing specific system configurations. Those specific classes must be organized in a package structure (the user projects) complying with the Modelica specification. Note that according to the specification, a package can be either a single file (for instance `NameOfPackage.mo`) or a directory containing a `package.mo` file, and the package file may itself include some definitions of subpackages.

The UI must provide a means to explore both the package containing the template classes and the package containing the specific classes (the user projects).

- A file explorer with a tree view should reveal the package structure in a left panel.
- Only the classes defined in the package file, or enumerated in the `package.order` file shall be displayed. And they shall be displayed in the same order as the one specified by those two files.
- The left panel is divided vertically in two parts: the upper part for the templates, the lower part for the user projects.
- The description string of each class must be displayed, for instance when hovering a package or a model in the file explorer.

The following example illustrates typical package structures and the way they should be displayed in the UI.

Listing 3.1: Example of the package structure for the templates and user projects (in the file system)

```
Buildings
├── Templates
│   ├── AHUs
│   │   ├── Data
│   │   ├── package.mo          # Contains __Linkage_Template annotation.
│   │   ├── package.order
│   │   └── VAVSingleDuct.mo    # Contains __Linkage_Template annotation.
│   ├── BoilerPlants
│   │   └── ...                 # Enclosed file package.mo contains __Linkage_Template␣
↪annotation.
│   ├── ChillerPlants
│   │   └── ...                 # Enclosed file package.mo contains __Linkage_Template␣
↪annotation.
│   ├── TerminalUnits
│   │   └── ...                 # Enclosed file package.mo contains __Linkage_Template␣
↪annotation.
│   ├── package.mo
│   └── package.order
├── ...
├── package.mo
```

(continues on next page)

(continued from previous page)

```
└── package.order

UserProjects
├── Project_1
│   ├── AHUs
│   │   ├── Data
│   │   ├── package.mo
│   │   ├── package.order
│   │   └── VAV_1.mo
│   ├── BoilerPlants
│   │   └── ...
│   ├── ChillerPlants
│   │   └── ...
│   ├── TerminalUnits
│   │   └── ...
│   ├── package.mo
│   └── package.order
├── {Project_i}
│   └── ...
├── package.mo
└── package.order
```

This should be rendered in the UI as follows.

Listing 3.2: Example of the rendering of the package structure in the UI

```
Buildings
├── AHUs
│   └── VAVSingleDuct
├── BoilerPlants
│   └── ...
├── ChillerPlants
│   └── ...
└── TerminalUnits
    └── ...

UserProjects
├── Project_1
│   ├── AHUs
│   │   ├── VAV_1
│   │   └── Data
│   ├── BoilerPlants
│   │   └── ...
│   ├── ChillerPlants
│   │   └── ...
│   └── TerminalUnits
│       └── ...
└── {Project_i}
    └── ...
```

The suggested workflow is as follows.

1. The template package of the Modelica Buildings Library is preloaded. The tool provides the option to load additional template packages from third-party libraries. A template package is identified by the class annotation `__Linkage_Template` in the package file.
   - Only the classes with the annotation `__Linkage_Template` should be displayed in the template file explorer.
2. The user can select whether to create a `UserProjects` from scratch or to load a package stored locally on the device.
   - If a new package is created, it must contain the class annotation `uses(Buildings(version="..."), ...)` with the version of all loaded libraries.
   - When loading a package with the class annotation `uses(Buildings(version="..."), ...)` refer to Table 3.2 for the library version management.
3. The user can create a new project, for instance by right clicking on `UserProjects` which renders a menu with the options *Add New*, etc.
4. The user can select the working project to save the new specific classes, for instance by right clicking on `Project_1` which renders a menu with the options *Set Working Project*, *Rename*, *Delete*, etc.
   - The current working project must be clearly highlighted in the user projects file explorer.
5. The user selects a template to start the configuration workflow, for instance by right clicking on `VAVSingleDuct` which renders a menu with the option *Start Configuring*, etc.
6. A specific class is created under the corresponding subpackage (for instance `AHUs`) of the current working project in the `UserProjects` package.
   - The new class is constructed by extending the original template `extends type-specifier [ class-modification ] [annotation]`.
   - The parameter dialog of the template class is generated in the configuration panel. In addition, two input fields allows specifying the simple name and the description string of the specific class to be generated.
   - The tree view of the `UserProjects` package is updated dynamically, based on the class name and the class description string input by the user.
   - Any user input leads to updating the specific class definition. The full composed name (dot notation starting from the top-level library package, for instance `Buildings`) shall be used to reference each class used in the class definition.
7. Optionally, a record class with the same simple name is created under the corresponding subpackage, for instance `AHUs.Data`. The record contains the same class modifications as the ones applied to the records of the specific class. This will allow the user to further use this record to propagate the parameters of an instance of the specific class to a top-level simulation model.
8. At least two action buttons *Save* and *Cancel* are required in the configuration panel. The class within the `UserProjects` package is only modified upon *Save*. All the modifications are reset to the last saved state upon *Cancel*.
9. Once created, the user can select each specific class from the user projects file explorer and further modify it, for instance by right clicking on the corresponding class which renders a menu with the options *Edit Class*, *Delete*, *Rename*, *Duplicate*, etc.
10. Export functionalities (Modelica code, see Section 3.5, or documentation, see Section 3.6) are available from the user projects file explorer, at the level of the package and at the level of the specific class.

3.4 Standard Streams and Error Logging

Revision Note

This paragraph is added.

An error logging mechanism is required.

Standard output and standard error streams must support redirecting to any file descriptor when integrating the widget into a third party application.

3.5 Modelica Export

Revision Note

This paragraph is added.

Exporting Modelica code requires the following steps.

1. Converting the JSON-formatted Modelica into Modelica code. This should be done by calling Modelica to JSON translator.
2. Formatting the generated Modelica code. This should be done by calling https://github.com/urbanopt/modelica-fmt. *Implemented in Go: can it run on client side? With WebAssembly?*.
3. Optional compression. File compression is only required for exporting a package. A single class should be exported as a single uncompressed `mo` file.
4. Launch downloading on the client.

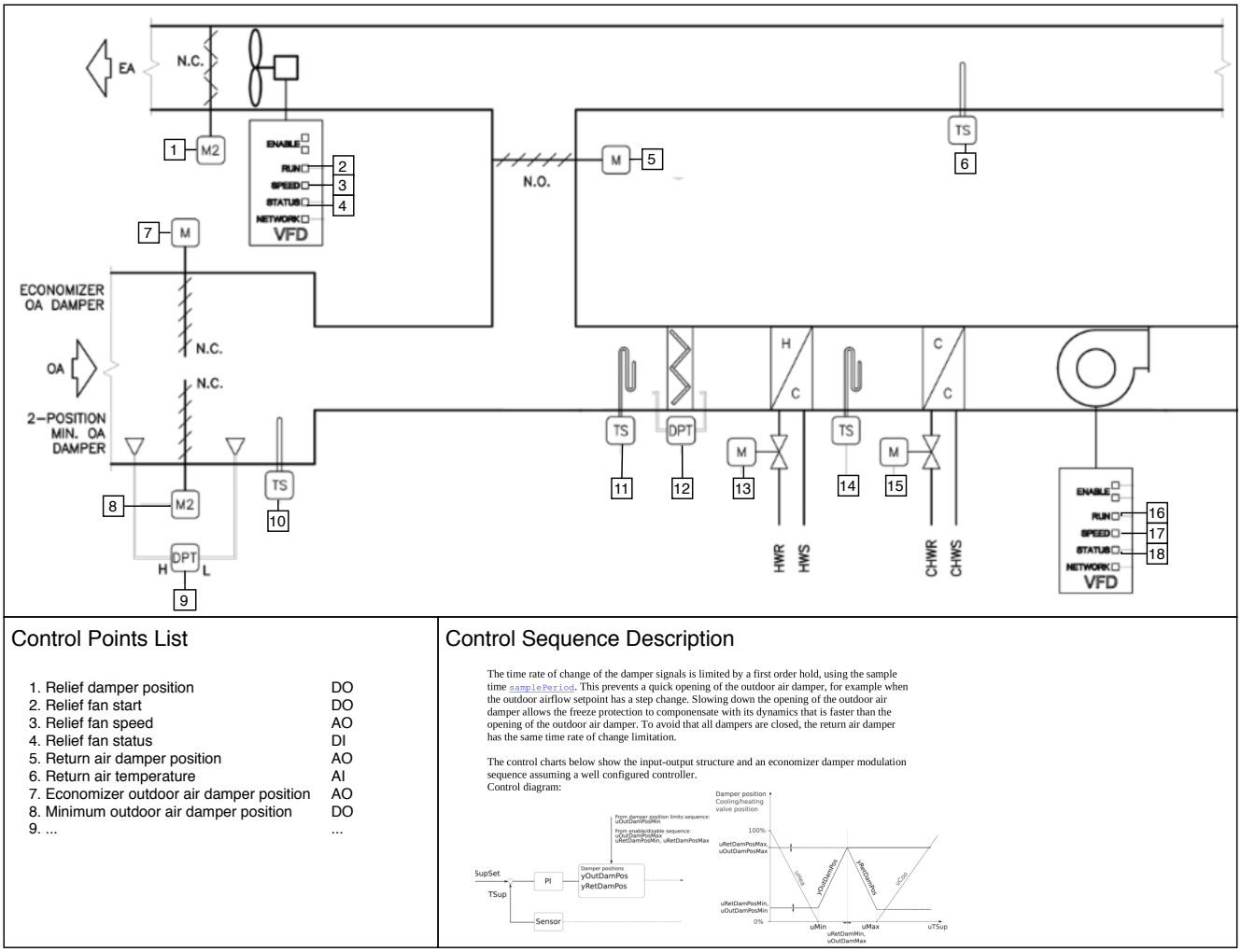
3.6 Documentation Export

The documentation export encompasses three items.

1. Engineering schematics of the equipment including the controls points
2. Control points list
3. Control sequence description

The composition level at which the functionality will typically be used is the same as the one considered for the configuration widget, for instance primary plant, air handling unit, terminal unit, etc. No specific mechanism to guard against an export call at different levels is required.

Fig. 3.2 provides an example of the documentation to be generated in case of an air handling unit. The documentation export may consist in three different files but must contain all the material described in the following paragraphs.



Control Points List

1. Relief damper position	DO
2. Relief fan start	DO
3. Relief fan speed	AO
4. Relief fan status	DI
5. Return air damper position	AO
6. Return air temperature	AI
7. Economizer outdoor air damper position	AO
8. Minimum outdoor air damper position	DO
9. ...	...

Control Sequence Description

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

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

Fig. 3.2: *Mockup of the documentation export*

3.6.1 Engineering Schematics

Objects of the original model to be included in the schematics export must have a declaration annotation providing the SVG file path for the corresponding engineering symbol e.g. `annotation(__Linkage(symbol_path="value of symbol_path"))`.

Note: It is expected that Linkage will eventually be used to generate design documents included in the invitation to tender for HVAC control systems. The exported schematics should meet the industry standards and they must allow for further editing in CAD softwares, e.g., AutoCAD®.

Due to geometry discrepancies between Modelica icons and engineering symbols a perfect alignment of the latter is not expected by simply mapping the diagram coordinates of the former to the SVG layout. A mechanism should be developed to automatically correct small alignment defaults.

For the exported objects

- the connectors connected to the control input and output sub-buses must be split into two groups depending on their type—Boolean or numeric,
- an index tag is then generated based on the object position, from top to bottom and left to right,
- eventually connection lines are drawn to link those tags to the four different control points buses (AI, AO, DI, DO). The line must be vertical, with an optional horizontal offset from the index tag to avoid overlapping any other object.

SVG is the required output format.

See Fig. 3.2 for the typical output of the schematics export process.

3.6.2 Control Points List

Generating the control points list is done by calling a module developed by LBL (ongoing development) which returns an HTML or Word document.

3.6.3 Control Sequence Description

Generating the control sequence description is done by calling a module developed by LBL which returns an HTML or Word document.

3.7 Licensing

The software is developed under funding from the U.S. Department of Energy and the U.S. Government consequently retains certain rights. As such, the U.S. Government has been granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software to reproduce, distribute copies to the public, prepare derivative works, and perform publicly and display publicly, and to permit other to do so.

The main software components built as part of this development project must be open sourced under BSD 3 or 4-clause, with possible additions to make it easy to accept improvements. Licensing under GPL or LGPL will not be accepted.

Different licensing options are then envisioned depending on the integration target and the engagement of third-party developers and distributors. The minimum requirement is that at least one integration target be made available as a free software.

- Desktop app
  Subscription-based
- Standalone web app
  - Free account allowing access to Modelica libraries preloaded by default, for instance Modelica Standard and Buildings: the user can only upload and download single Modelica files (not a package).
  - Pro account allowing access to server storage of Modelica files (packages uploaded and models saved by the user): the user can update the stored libraries and reopen saved models between sessions.
- Third-party application embedding
  Licensing will depend on the application distribution model.
  For OpenStudio there is currently a shift in the licensing strategy. The specification will be updated to comply with the distribution options after the transition period (no entity has yet announced specific plans to continue support for the OS app).

Chapter 4

Annex

This annex is informative only. It presents various validation cases and additional information for the development of the templates in Phase 1 and the future development of a diagram editor in Phase 2.

4.1 Using Expandable Connectors in Templates

Revision Note

This paragraph is moved from the *Requirements* section to the *Annex* section.

4.1.1 General Principles

The `connect` equations for signal variables in the Modelica templates rely on expandable connectors (also referred to as control bus), see *§9.1.3 Expandable Connectors* in [Mod17].

The following features of the expandable connectors are leveraged. They are illustrated with minimal examples in Section 4.2.

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

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

2. The causality (input or output) of each variable inside an expandable connector is not predefined but rather set by the `connect` equation where the variable is first being used. For instance when the variable of an expandable connector is first connected to an inside connector `Modelica.Blocks.Interfaces.RealOutput` it gets the same causality i.e. output. The same variable can then be connected to another inside connector `Modelica.Blocks.Interfaces.RealInput`.
3. Potentially present but not connected variables are eventually considered as undefined i.e. a tool may remove them or set them to the default value (Dymola treat them as not declared: they are not listed in `dsin.txt`): all variables need not be connected so the control bus does not have to be reconfigured depending on the model structure.
4. The variables set of a class of type expandable connector is augmented whenever a new variable gets connected to any *instance* of the class. Though that feature is not needed by the configuration widget (we will have a predefined control bus with declared variables), it is needed to allow the user further modifying the control sequence. Adding new control variables is simply done by connecting them to the control bus.
5. Expandable connectors can be used in arrays, as any other Modelica type. A typical use case is the connection of control input signals from a set of terminal units to a supervisory controller at the AHU or at the plant level. This use case has been validated on minimal examples in Section 4.4.

4.1.2 Additional Requirements for the UI in Phase 2

Fig. 4.1 presents the Dymola pop-up window displayed when connecting the sub-bus of input control variables to the main control bus (based on the validation case in Section 4.3). A similar view of the connections set must be implemented with the additional requirements listed below. That view is displayed in the connections tab of the right panel.

The variables listed immediately after the bus name are either

- *declared variables* that are not connected, for instance `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, for instance `ahuSubBusI.TZonHeaSet`: the icon next to each variable then indicates the causality. Those variables can originally be either declared variables or variables elaborated by the augmentation process for *that instance* of the expandable connector i.e. variables that are declared in another component and connected to the connector's instance.

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

In addition to Dymola's features for handling the bus connections, Linkage Phase 2 will require 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.

    Note: 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.

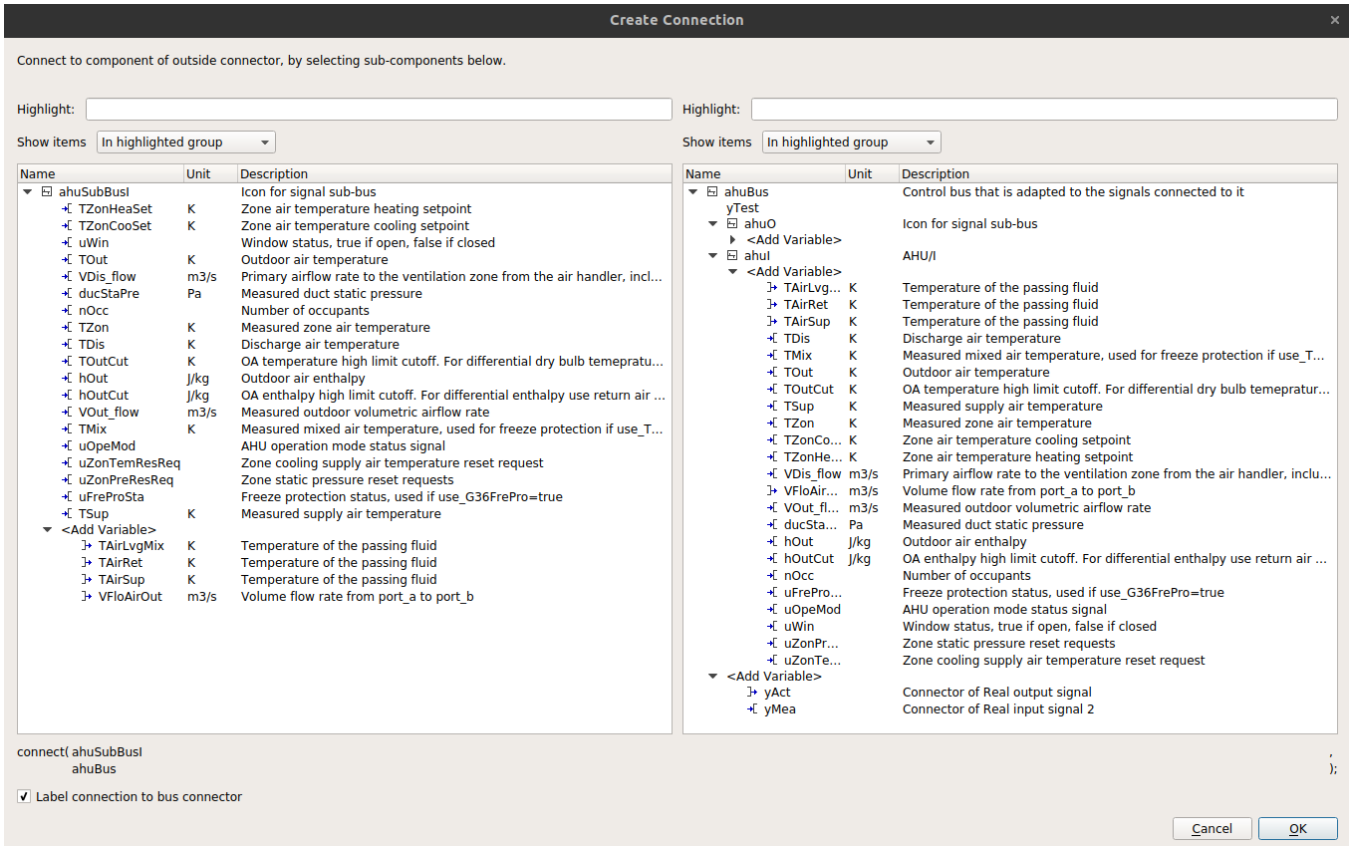


Fig. 4.1: Dymola pop-up window when connecting the sub-bus of input control variables (left) to the main control bus (right) – case of outside connectors

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 the 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. 4.2 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. 4.3 to Fig. 4.5 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. 4.5 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, Linkage 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. 4.3 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 Linkage could be:

Table 4.1: Typical view of the connection set of expandable connectors – visible from outside component (connector is inside), "Present" and "I/O" columns display the connection status over the full augmentation set

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

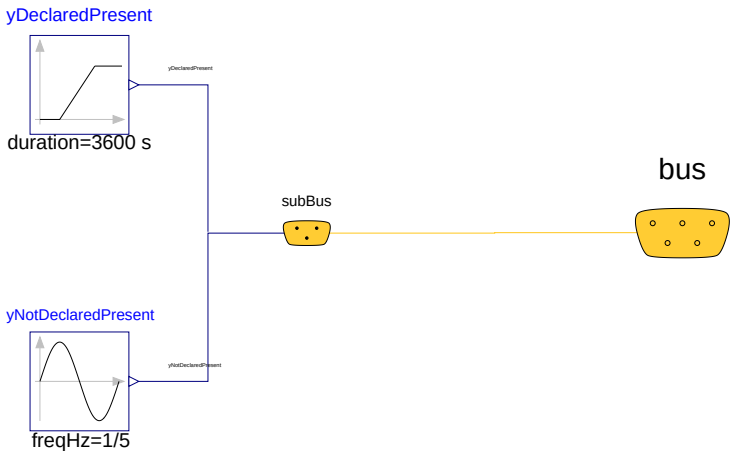


Fig. 4.2: Minimal example of sub-bus to bus connection illustrating how the bus variables are exposed in Dymola – case of outside connectors

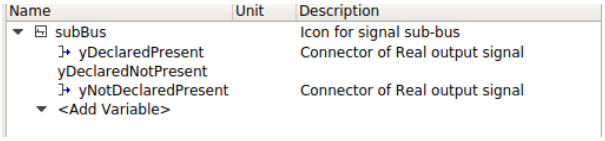


Fig. 4.3: Sub-bus variables being exposed in case the sub-bus is an outside connector

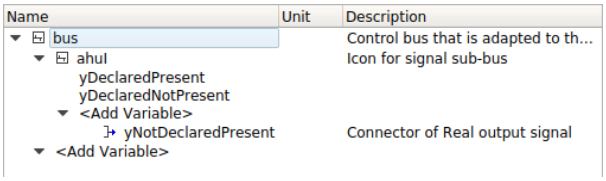


Fig. 4.4: Bus variables being exposed in case the bus is an outside connector

4.2 Main Features of the Expandable Connectors

The main features of the expandable connectors are illustrated with a minimal example described in the figures below where

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

```
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));
Buildings.Experimental.Templates.BaseClasses.AhuBus ahuBus;
Modelica.Blocks.Routing.RealPassThrough actuator;
equation
      connect(sensor.y, ahuBus.yMea);
      connect(ahuBus.yAct, actuator.u);
end BusTestControlledExp;
```

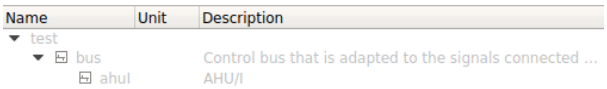


Fig. 4.5: Bus variables being exposed in case the bus is an inside connector

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

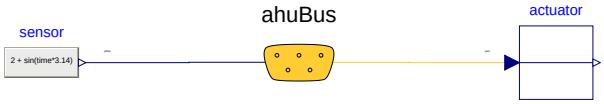


Fig. 4.7: Minimal example illustrating the connection scheme with an expandable connector – Controlled component sublevel

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

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

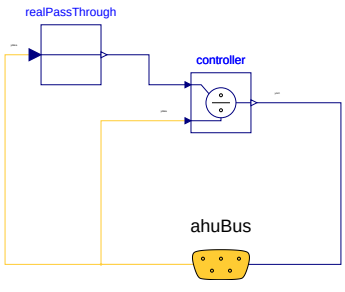


Fig. 4.8: Minimal example illustrating the connection scheme with an expandable connector – Controller component sublevel

```
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;
```

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

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

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

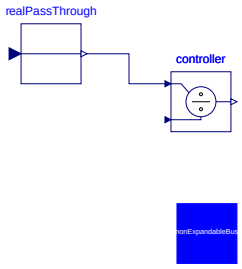


Fig. 4.11: Minimal example illustrating the connection scheme with a standard connector – Controller component sublevel

```
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;
```

4.3 Validating the Use of Expandable Connectors

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

The validation is performed

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

Simulation succeeds for the two tests cases with the two simulation tools. The results comparison to the original test case (without control bus) is presented in Fig. 4.12 for Dymola.

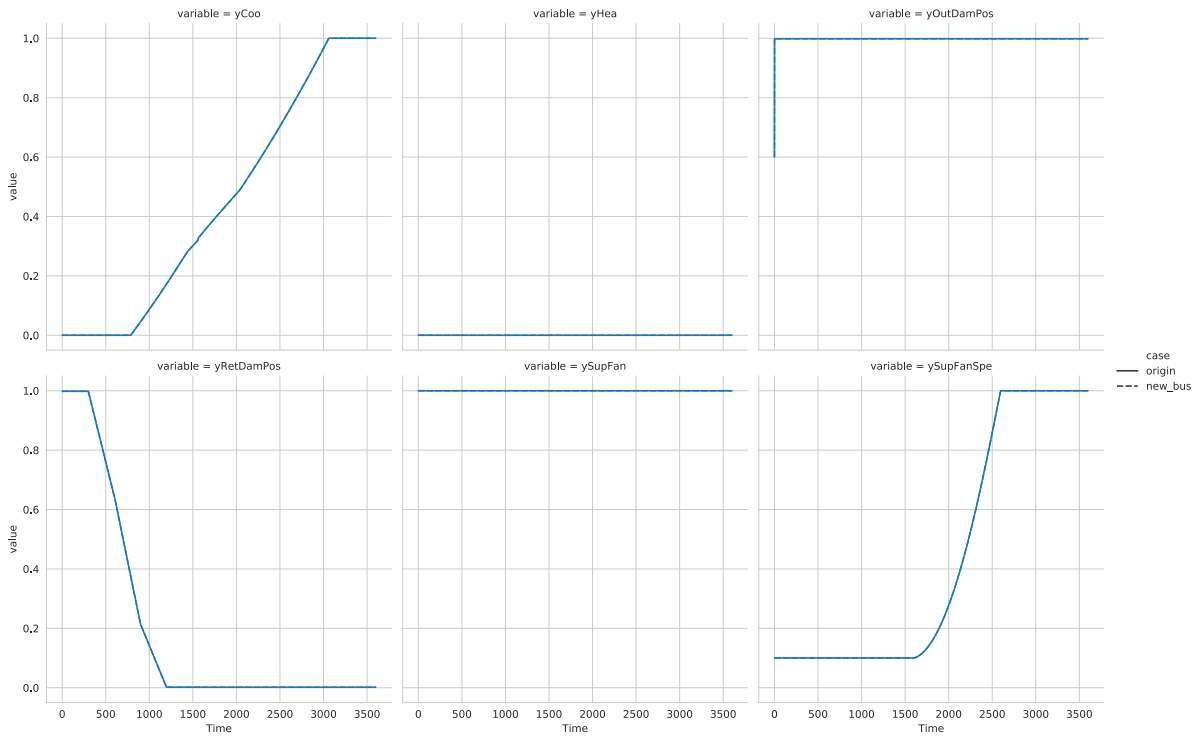


Fig. 4.12: G36 AHU controller model: comparison of simulation results (Dymola) between implementation without (`origin`) and with (`new_bus`) expandable connectors

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 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:

- allows the unspecified `[:]` syntax and,
- 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 Linkage.

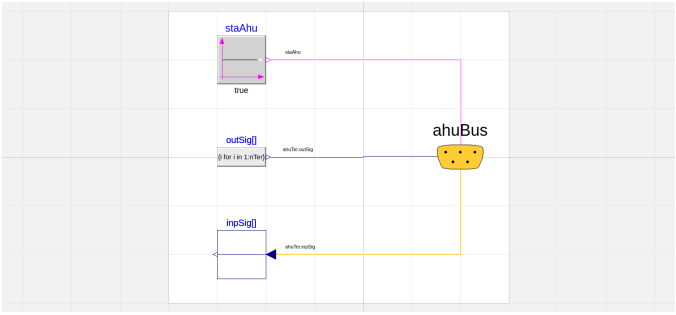
4.4 Validating the Use of Expandable Connector Arrays

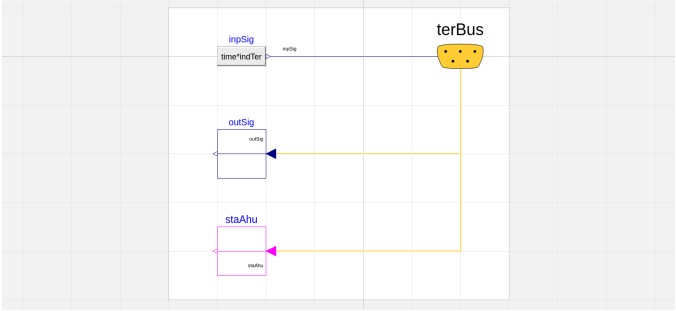
Minimum examples illustrate that arrays of expandable connectors are differentially supported between Dymola and OCT. None of the tested Modelica tools seems to have a fully robust support. However, by reporting those bugs, it seems as a feature we can leverage for Linkage.

We start with the basic definition of an expandable connector `AhuBus` containing the declaration of an array of expandable connectors `ahuTer` that will be used to connect the signal variables from the terminal unit model. In addition we build dummy models for a central system (e.g. VAV AHU) and a terminal system (e.g. VAV box) as illustrated in the figures

hereafter. The input signal `inpSig` is typically generated by a sensor from the terminal system and must be passed on to the central system which, in response, outputs the signal `outsig` typically used to control an actuator position in the terminal unit.

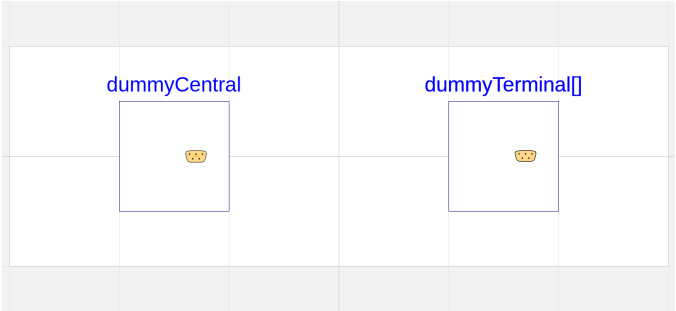
```
expandable connector AhuBus
   extends Modelica.Icons.SignalBus;
   parameter Integer nTer=0
      "Number of terminal units";
      // annotation(Dialog(connectorSizing=true)) is not interpreted properly in Dymola.
   Buildings.Experimental.Templates.BaseClasses.TerminalBus ahuTer[nTer] if  nTer > 0
      "Terminal unit sub-bus";
end AhuBus;
```





4.4.1 Connecting One Central System Model to an Array of Terminal System Models

The first test is illustrated in the figure below.



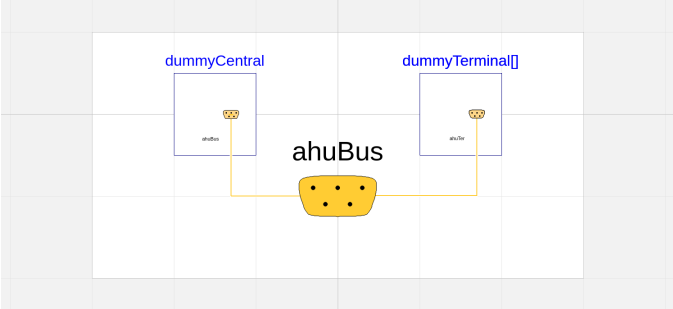
Bug in Dymola

Dymola GUI does not allow graphically generating the statement `connect(dummyTerminal.terBus, dummyCentral.ahuBus.ahuTer)`. The GUI returns the error message `Incompatible connectors`.

However, we cannot find which part of the specification [Mod17] this statement would violate. To the contrary, the specification states that "expandable connectors can be connected even if they do not contain the same components".

Additionally, when manually adding this `connect` statement in the code, the model simulates (with correct results) with OCT. Dymola fails to translate the model and returns the error message `Connect argument was not one of the valid forms, since dummyCentral is not a connector`.

Based on various tests we performed, it seems that Dymola supports connecting *inside* expandable connectors together only when they are instances of the same class. Again, we cannot find such a requirement in Modelica specification. To allow such a connection in Dymola, we need to rely on an *outside* expandable connector as illustrated below.



Bug in OCT

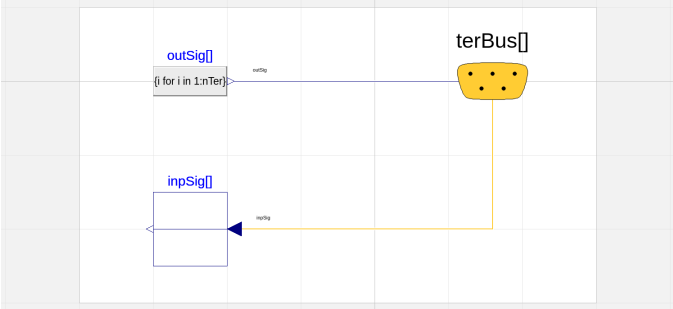
With this connection layout, the model simulates with Dymola but no more with OCT which returns the following error message.

```
Error at line 296, column 5, in file '/opt/oct/ThirdParty/MSL/Modelica/Blocks/Interfaces.mo':
Cannot find class declaration for RealInput
```

Bug in Dymola

Incidentally we observe other bugs in Dymola related to the elaboration process leading to a variable being marked as present in the expandable connector variable set.

- When connecting a non declared variable to a sub-bus, e.g., `connect(ahuBus.ahuTer.inpSig, inpSig.u)`, the corresponding expandable connector variable list (visible in Dymola GUI under `<Add Variable>` when drawing a connection to the connector) does not get augmented with the variable name.
- When connecting a non declared variable directly to an array of expandable connectors as in the figure below, the dimensionality may be wrong depending on the first connection being established. Indeed, `terBus.inpSig` is considered as an array if `terBus[:].inpSig` is first connected to a one-dimensional array of scalar variables. The code needs to be updated manually to suppress the array index and simulate. If the first connection of `inpSig` variable to the connector is made at the terminal unit level (scalar to scalar) then the dimensionality is correctly established.
- In several use cases, we noticed similar issues related to the dimensionality of variables in presence of nested expandable connectors. In that respect OCT appears more robust.

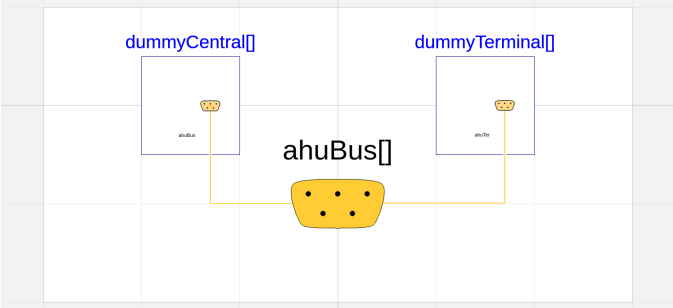


4.4.2 Connecting an Array of Central System Models to an Array of Terminal System Models

We now try to connect an one-dimensional array of central system models `DummyCentral dummyCentral[nAhu]` to a two-dimensional array of terminal system models `DummyTerminal dummyTerminal[nAhu, nTerAhu]`.

Bug in Dymola

As explained before, in Dymola, we need to rely to an *outside* expandable connector to connect the two *inside* expandable connectors.



However, despite the connection being made properly through the GUI, the model fails to translate.

```
Unmatched dimension in connect(ahuBus.ahuTer, dummyTerminal.terBus);

The first argument, ahuBus.ahuTer, is a connector with 1 dimensions
and the second, dummyTerminal.terBus, is a connector with 2 dimensions.
```

The error message is incorrect as in this case `ahuBus.ahuTer` has two dimensions.

OCT also fails to translate the model but for a different reason, see error message previously mentioned. However, when manually adding the connect statement between the two *inside* connectors `connect(dummyTerminal.terBus, dummyCentral.ahuBus.ahuTer)`, the model simulates with OCT.

4.4.3 Passing on a Scalar Variable to an Array of System Models

The typical use case is a schedule, set point, or central system status value that is used as a common input to a set of terminal units. Two programmatic options are obviously available.

1. Instantiating a replicator (routing) component to connect the variable to the expandable connector array. After discussion with the team, it seems like the best approach to use in production.
2. Looping over the expandable connector array elements to connect each of them to the variable.

The test performed here aims to provide a more "user-friendly" way of achieving the same result with only one connection being made (either graphically or programmatically).

The best approach would be a binding of the variable in the declaration of the expandable connector array.

```
expandable connector AhuBus
   parameter Integer nTer
      "Number of terminal units";
   Boolean staAhu
      "Test how a scalar variable can be passed on to an array of connected units";
   Buildings.Experimental.Templates.BaseClasses.TerminalBus ahuTer[nTer](
      each staAhu=staAhu) if nTer > 0
      "Terminal unit sub-bus";
end AhuBus;
```

However that syntax is against the Modelica language specification. It is indeed equivalent to an equation, and equations are not allowed in an expandable connector class.

The approach eventually tested relies on a so-called "gateway" model composed of several instances of expandable connectors and an equation section used to establish the needed connect statements. Note that if a variable is left unconnected then it is considered undefined, so the corresponding connect statement is automatically removed by Modelica tools.

```
model AhuBusGateway
   "Model to connect scalar variables from main bus to an array of sub-bus"
   parameter Integer nTer
      "Number of terminal units";
   AhuBus ahuBus(nTer=nTer);
   TerminalBus terBus[nTer];
equation
   for i in 1:nTer loop
      connect(ahuBus.staAhu, ahuBus.ahuTer[i].staAhu);
   end for;
   connect(ahuBus.ahuTer, terBus);
end AhuBusGateway;
```

Bug in Dymola

When trying to simulate a model using such a component Dymola fails to translate and returns:

```
The bus-input dummyTerminal[1].terBus.staAhu lacks a matching non-input in the connection␣
↪sets.
This means that it lacks a source writing the signal to the bus.
```

However, OCT simulates the model properly.

Chapter 5

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.

Chapter 6

References

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

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