Building Technologies & Urban Systems Division
Energy Technologies Area
Lawrence Berkeley National Laboratory

HVAC and Control Templates for the Modelica Buildings Library

Antoine Gautier[1], Michael Wetter[2], Jianjun Hu[2], Hubertus Tummescheit[3]

[1]Solamen, France, [2]Lawrence Berkeley National Laboratory, Berkeley, CA, [3]Modelon, Hartford, CT

Energy Technologies Area
October 2023

https://doi.org/10.3384/ecp204217

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Building Technologies Office, of the US Department of Energy under Contract No. DE-AC02-05CH11231.

Disclaimer:

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

HVAC and Control Templates for the Modelica Buildings Library

Antoine Gautier[1] Michael Wetter[2] Jianjun Hu[2] Hubertus Tummescheit[3]

[1]Solamen, France, agautier@solamen.fr
[2]Lawrence Berkeley National Laboratory, Berkeley, CA
[3]Modelon, Hartford, CT

Abstract

This article reports on our experience in creating Modelica classes that serve as templates for modeling HVAC systems with thousands of configurations and closed-loop controls. Our motivation is to reduce model creation and parameterization time, provide access to state-of-the-art control sequences, while limiting the risk of error and enforcing modeling best practices. The development of such templates required exploration of class parameterization techniques and data structures for handling large sets of equipment parameters. By describing these issues and the approach taken, we show how the Modelica language can support advanced templating logic. The main limitation we encountered relates to parameter assignment and propagation. The interpretation of parameter attributes at user interface runtime, or the handling of non-trivial constructs involving record classes at compile time is not consistently supported by Modelica tools. This leads to choices that are difficult to make when looking for a generic implementation.

Keywords: Modelica Buildings Library, template, class parameterization

1 Introduction

Modeling Heating, Ventilation and Air-Conditioning (HVAC) systems with the Modelica Buildings Library (Wetter, Zuo, et al. 2014) usually relies on a component-by-component approach that is both time-consuming and error-prone, requiring expertise in configuring HVAC systems and designing and implementing the appropriate feedback control logic. This motivates the development of pre-built Modelica models that can be easily reconfigured and serve as templates for a variety of HVAC systems.

In our experience, developing Modelica-based templates for complex systems requires not only advanced knowledge of the language, but also experience in template development in particular. To circumvent a high level of complexity, or to allow for a more straightforward development process, some tool developers instead resorted to external templating engines, such as Mako (Nytsch-Geusen et al. 2017) or Jinja (Long et al. 2021). In the latter application, the highly variable network topology of district heating and cooling systems is captured by a GeoJSON parameter schema, from which a Python-based templating layer generates Modelica code. For use of templating engines, in addition to the underlying Modelica models that serve as building blocks for such a workflow, a parameter schema must be developed, along with template files and the software that does the translation into Modelica. This leads to more dependencies to manage, and we believe it also increases the maintenance overhead compared to Modelica-based templating. For example, a change in the underlying Modelica library may require an update of the templates or the translator itself. Moreover, such tools usually implement a one-way trip from the parameter schema to Modelica. Any subsequent change to the Modelica model will therefore result in the configuration workflow no longer being applicable. If the configuration workflow involves programmatic creation of `connect` statements, this will most likely affect the graphical aspect of the model. Finally, a lack of reusability becomes apparent. There are as many template schemas and translators as there are development projects, with no clear way for other applications—even from the same domain—to leverage the existing work.

Alternatively, some developments rely exclusively on the Modelica language and its parametric polymorphism (Broman, Fritzson, and Furic 2006). For example, the Vehicle Dynamics Library developed by Modelon achieves a high degree of configurability through the use of class parameterization techniques, which are described in more detail in this paper. The Vehicle Dynamics Library also provides the ability to specify system parameters via XML, JSON, MAT, or Adams data properties files. The library covers both the chassis and the powertrain configurations of all current architectures of passenger vehicles, as well as many classes of trucks. The hierarchical use of class parameterization gives a high flexibility to include new technologies, and also allows to use different levels of modeling details during the design process. Another example is given by Greenwood et al. (2017) for modeling power plants. The approach uses replaceable elements to configure subsystems and controls, and a record class for parameter assignment within each subsystem. More recently, Wüllhorst et al. (2023) introduced BESMod, an open-source Modelica library for research and teaching purposes that provides a modular approach to domain-coupled simulations of building energy systems. The library is structured with modules that represent the various systems, e.g., demand, ventilation, hydraulic system. Each module is built using expandable connectors,

vector-sized ports, and a unified parameterization framework based on Modelica records. The modules are agnostic of the component models, which can come from various open source libraries such as Buildings or IBPSA (Wetter, Blum, et al. 2019). An example illustrates how building models are interchangeable from one library to another. However, for each module, the configuration options are limited to those of the underlying libraries, and there is still a need for system-level templates. For example, it is not possible to change the system layout and control options of an air handling unit if those features are not present in the library that provides that component.

In this paper we will go over the advantages and disadvantages of Modelica-based templating, which is the method we use. In addition to providing insights to help future template developers, we discuss the main constructs of the Modelica language that serve the purpose of templating, and we point out the limitations and possible language extensions or tool improvements that could make the task easier. We start with some important definitions in section 2 and the key requirements guiding our development in section 3. The core concepts that support templating are introduced in section 4. Some implementation choices related to connecting signal variables, structuring system parameters and integrating graphical elements for control diagrams are then presented in section 5, section 6 and section 7, respectively. Finally, an overview of the test workflow we use to validate the numerous configurations covered by the templates is given in section 8.

2 Definitions

Throughout this article, the following terms are used according to the definitions given here.

Configuration. A system configuration corresponds to the specification of the type and layout of the equipment and the corresponding control logic. Systems with different capacities may have the same configuration, provided they have the same control software and hardware type.

Parameterization. By parameterization we mean all possible class modifications, such as changing parameter values and redeclaring components or classes, which we refer to as class parameterization (Zimmer 2010).

Structural and value parameters. We use the term structural parameters if a parameter affects the number and structure of the equations, and value parameters if they do not. An example of a structural parameter is a parameter used to specify an array size. The use of these terms is consistent with Kågedal and Fritzson (1998).

System. By system we mean a set of components that "share a load in common, i.e., collectively act as a source to downstream equipment, such as a set of chillers in a lead/lag relationship serving air handlers", whereas "each air handler constitutes its own separate system because it does not share a load (terminal unit) in common with the other air handlers". Our use of the term "system" is adopted from ASHRAE (2021).

Template. A template, or template class, is defined as a Modelica model that can be parameterized (as defined above) *to represent a particular system configuration.*

3 Requirements

We will now present key requirements that guided the development of the templates to provide the necessary context for understanding the main implementation choices.

3.1 Tool Compatibility

Our main requirement is that the language constructs used to create the templates are supported by various Modelica compilers. This appeared particularly constraining when dealing with nested expandable connections (see section 5) or choosing the right data structure for system parameters (see section 6). Our test workflow (see section 8) currently includes Dymola (Dassault Systèmes AB 2023), Modelon Impact (Modelon AB 2023b; Modelon AB 2023a), and we are working on support with OpenModelica. In addition, the graphical primitives used for icons and diagrams (see section 7) should also be supported by various Modelica tools, especially if they include a `visible` attribute that requires the evaluation of Boolean expressions at user interface (UI) runtime.

3.2 Diversity of Equipment and Controls

To illustrate the diversity that must be represented, it should be noted that a simple air handling unit can have thousands of possible combinations of equipment, not counting the various control options and the type and placement of sensors required for them.

In addition, there is a strong dependency between the different types of equipment and control logic. For instance, ASHRAE (2021) specifies that the primary hot water flow sensor in a boiler plant is "required for primary-only plants", that the sensor is "optional for variable primary-variable secondary plants" and "not required nor recommended for constant primary-variable secondary plants." In this case, the specification of some equipment (the primary and secondary hot water pumps) together with a control option (the type of sensors used to control the primary recirculation in variable primary-variable secondary systems) constrains the possible options for another piece of equipment (the primary flow sensor), which, if present, can be located either in the supply or in the return pipe.

Conceptually, this means that the user's choices can affect the possible options that are exposed at another level of the model's composition. We will see in subsection 4.3, with a concrete example, the language constructs that are used to support this process.

3.3 System Parameters

The data structure containing the design and operating parameters should allow parameter values to be assigned via a unique object at the top level of the simulation model. Such an object can be viewed as a digital avatar of the

manufacturer's data sheets for a complete HVAC system, from plant to zone equipment.

System parameters usually run into the hundreds and are highly dependent from one device to another, so it should be possible to express these relationships in binding equations. In addition, a mechanism should be available to expose only the parameters required for a project's specific system configurations.

Finally, it should be possible to reuse existing equipment datasets implemented as Modelica `record` classes from the library on which the templates are based, e.g., pumps, fans, chillers and boilers for the Buildings library.

We will see in section 6 the resulting implementation choices.

3.4 System Level Templates

The templates need to be provided at the system level, e.g., a central plant or a air handler. Therefore, creating a simulation model for a complete HVAC system involves multiple instances of templates and multiple connections between physical connectors (for fluid circuits) and input/output connectors (for controls). This task should be achievable without need of an automation tool. In practice, this leads to the use of expandable connectors (Modelica Association 2021) to connect control inputs and outputs between systems (see section 5), as otherwise a large set of connections would be required, and this set would vary with each system configuration.

3.5 Scalability

Any number of identical devices must be supported. In practice, this leads to use of array instances for models that can represent multiple units, such as pumps, chillers or zone equipment. The main difficulty then lies in managing this dimensionality for non-trivial constructs such as nested expandable connectors (see section 5) or record classes (see subsection 6.2).

3.6 Integration With OpenBuildingControl

The OpenBuildingControl project aims to digitize the control delivery process based on control specifications that are a subset of Modelica and now being standardized through ASHRAE Standard 231P (Wetter, Grahovac, and Hu 2018; Wetter, Ehrlich, et al. 2022). To support this workflow, the templates shall contain the information necessary to prepare the documents required for the bidding and project execution of HVAC systems.

The ability to generate a control diagram is of particular importance to our development, see section 7. The main requirement is that the data used to create control diagrams be provided as graphical annotations that a Modelica tool with a graphical user interface (GUI) can interpret. This way, when a template is configured in a Modelica tool, the user can get direct graphical feedback on the system layout.

Although outside the scope of this work, additional resources can be exported by dedicated tools, all of which use a JSON representation of Modelica templates as a central digital resource (Wetter, Hu, et al. 2021). These resources include documentation of the sequence of operation, the control point list, or an executable version of the control sequence that control vendors can translate into their product line and commissioning agents can use to verify implementation of the control logic.

4 Structural Changes to a Model

In this section, we describe the mechanism by which we enable structural changes to a model directly through the parameter dialog, i.e., without manual user intervention on the model components. These structural changes allow representing a variety of system layouts and control options—and corresponding sensors and actuators—with a single pre-built model. This mechanism is thus at the core of template design and is based on the concept of class parameterization, which we first introduce in subsection 4.1. To support class parameterization in practice, template components typically need to be derived from interface classes designed to ensure *plug-compatibitility* as described in subsection 4.2. In subsection 4.3, we then present a concrete example of how parameterization constructs are used in conjunction with element annotations to represent dependencies between options for different components of a model and, more broadly, to cover the diversity of equipment and controls as first described in subsection 3.2.

4.1 Class Parameterization

The concept of class parameterization is central to template development. Class parameterization allows a class or component to be used as a parameter of another class. Zimmer (2010) gives an overview of the main language constructs supporting class parameterization in Modelica. Class parameterization can be accomplished via the following alternative approaches:

Container class. A container class, also called a wrapper class, is a class that contains structural parameters that are used to conditionally instantiate the components of the class. The main advantage is that a simple parameter binding with possible expressions can be used to reconfigure the class when instantiating or extending it. The main disadvantage is that the instance tree becomes more complex with additional nesting levels and instance names that vary depending on the system configuration.

Replaceable elements. Replaceable elements, either instances or classes, provide an alternative in which the instance tree is preserved, at least down to the level of the object being replaced. However, as pointed out by Zimmer (2010), these elements cannot be manipulated as standard parameters and require specific syntax (using the keywords `replaceable`, `constrainedby` and `redeclare`) that precludes conditional redeclarations involving expressions and parameters.

There is no single way to achieve the same goals in creating templates, and our developments use a variety of the

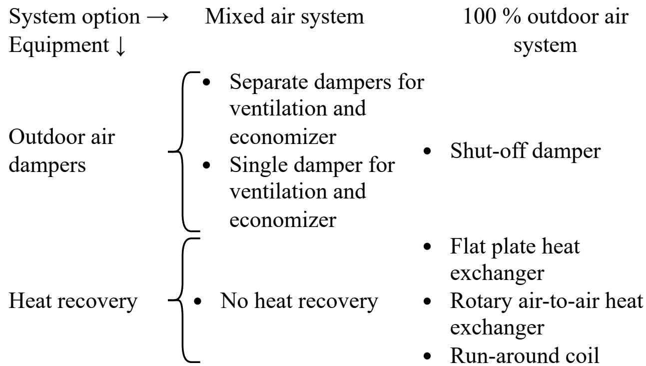

Figure 1. Example of equipment options depending on a high-level configuration option for an air handling unit air intake.

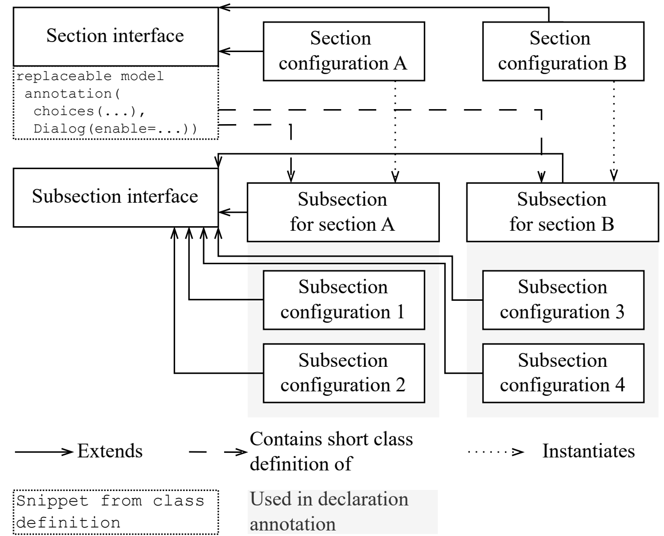

Figure 2. Class diagram for implementing the air intake section for an air handling unit template.

above constructs. In this section, we will try to illustrate how these concepts can be put into practice to solve some challenging use cases.

4.2 Interface Class

Designing appropriate interface classes is of paramount importance when creating templates. The main goal is to achieve *plug-compatibility* (Modelica Association 2021) for each component model created by extending such interface classes. Applying this concept ensures that all possible connections and parameter assignments can be specified in advance in a template class, so that each time a component is redeclared, no change to the `connect` clauses or binding equations is required.

This differs from the usual practice where interface classes typically contain the minimum common set of elements (e.g., outside connectors and parameters) required by all derived classes, which then extend this set as needed and are thus *type compatible*. In our templates, all outside connectors are declared within the interface class, with the appropriate conditional instance statements. Any class that extends an interface class does not declare any outside connector, but rather conditionally removes inherited connectors. Similarly, the interface class instantiates a record containing the full set of system parameters covering all possible configurations, see section 6.

4.3 Redeclarations and Choices Annotations

4.3.1 Concrete Example

To illustrate the use of class parameterization, let us consider the possible equipment options when specifying the air intake section of a multiple-zone air handling unit. Figure 1 shows these options: In a mixed air system, there are several options for the outdoor air dampers, but usually no heat recovery. In contrast, a 100 % outdoor air system offers several heat recovery options, but should be equipped with a shut-off outdoor air damper. The challenge is to fully cover these options and their constraints, and to select appropriate control logic while minimizing code duplication and maintenance overhead.

One might see the lack of conditional class definitions (or redeclarations) in the Modelica language as a limitation when trying to represent dependencies between options for different components of a model. For example, the following construct, although appealing, is not a valid short class definition.

```
class-prefixes IDENT "=" if expression
then type-specifier [ class-modification ]
else type-specifier [ class-modification ]
```

Similarly, conditional `choices` annotations, in which the exposed options depend on a Boolean expression, are not allowed. However, by using different levels of inheritance we can achieve almost the same intent, at the price of a more complicated class structure though.

Figure 2 gives an overview of the classes developed for such purpose, focusing on the options for one type of equipment (e.g., the outdoor air dampers) and considering the most generic case where multiple options exist for each system configuration, i.e., options `1` and `2` for system configuration `A` (mixed air) and options `2` and `3` for system configuration `B` (100 % outdoor air). At the top level of the template, a so-called "section" is declared, which contains all the interdependent components (labelled "subsection" in the figure), e.g., in our case the outdoor air dampers and the heat recovery (not shown in the figure for conciseness). This section derives from an interface containing short class definitions, optionally as replaceable models, and uses these definitions as constructors for its components. As illustrated in Listing 1 the choices specified in the annotation of the replaceable model `Subsection_A` (resp. `Subsection_B`) are only exposed for configuration `"A"` (resp. `"B"`) due to the `enable` attribute in the dialog annotation. Then the model selected from the given choices is used by the derived class `Section_A` (resp. `Section_B`) to create the actual object representing the subsection `subSec`.

Listing 1. Minimal working example illustrating the structure of a template class.

```
model Template "Template"
 replaceable Section_A sec
  constrainedby PartialSection "Section"
  annotation(choices(
   choice(redeclare replaceable Section_A
       sec),
   choice(redeclare replaceable Section_B
       sec)));
end Template;

partial model PartialSection
 "Section interface"
 parameter String config;

 replaceable model Subsection_A = Config1
  constrainedby PartialSubsection
  "Subsection"  annotation(choices(
   choice(redeclare replaceable model
       Subsection_A = Config1),
   choice(redeclare replaceable model
       Subsection_A = Config2)),
   Dialog(enable=config=="A"));

 replaceable model Subsection_B = Config3
  constrainedby PartialSubsection
  "Subsection"   annotation(choices(
   choice(redeclare replaceable model
       Subsection_B = Config3),
   choice(redeclare replaceable model
       Subsection_B = Config4)),
   Dialog(enable=config=="B"));
end PartialSection;

model Section_A "Section config A"
 extends PartialSection(final config="A");
 Subsection_A subSec "Subsection";
end Section_A;

model Section_B "Section config B"
 extends PartialSection(final config="B");
 Subsection_B subSec "Subsection";
end Section_B;

partial model PartialSubsection
 "Subsection interface"
 // Instantiate all possible connectors and
     parameters.
end PartialSubsection;

model Config1 "Subsection config 1"
 // Set parameter values needed to remove
     non-needed connectors, and implement
     actual components.
end Config1;
model Config2 "Subsection config 2"
end Config2;
model Config3 "Subsection config 3"
end Config3;
model Config4 "Subsection config 4"
end Config4;

replaceable Template system;
```

Zimmer (2010) mentions that replaceable *models* are most commonly used instead of replaceable *components* when multiple instances are to be redeclared with a unique statement. Replaceable packages are typically used for medium models because access to enclosed elements (e.g., constants and functions) is required. Our use case differs and replaceable models are used here in conjunction with inheritance and "deferred" instantiation as a practical means of achieving conditional class parameterization and conditional choices for replaceable elements. From a user experience (UX) perspective, this is identical to manipulating a replaceable component. Note that we systematically use `choices` annotations with **`redeclare replaceable`** to support further editing of the template after a configuration workflow.

4.3.2 Caveats and Alternatives

Resorting to UI features does not provide the same degree of robustness as using pure language constructs for object manipulation. For example, with a single line of code, one could manually redeclare `Subsection_A` with any type compatible model and violate the constraints imposed by the `choices` annotation.

One could also argue that for the configurations described in Figure 1, where there is either an option list or a *unique* option, as opposed to another option list, a simpler construct is to declare a replaceable *component* in the air intake section to represent the outdoor air dampers, with an `enable` attribute that evaluates to `true` for the mixed air configuration, and to `false` for the 100 % outdoor air configuration. For the latter case, an additional **`redeclare`** `final` statement then enforces the unique option for the outdoor air dampers (i.e., shut-off damper). However, additional scrutiny on the configuration workflow is here necessary. Indeed, an issue appears if a system model is first created by extending the template with a class modification that pertains to the mixed air configuration. Any further modification of the air intake section, either during inheritance or instantiation, risks an error due to a final override in the merging of modifiers, as shown below.

```
partial model PartialSection
 "Section interface"
 parameter String config;

 replaceable Config1 subSec "Subsection"
 annotation(choices(
  choice(redeclare replaceable Config1
      subSec),
  choice(redeclare replaceable Config2
      subSec)),
  Dialog(enable=config=="A"));
end PartialSection;

model Section_A "Section config A"
 extends PartialSection(final config="A");
end Section_A;

model Section_B "Section config B"
 extends PartialSection(final config="B",
  redeclare final Config3 subSec);
end Section_B;
```

```
model System1
 extends Template(sec(redeclare replaceable
     Config2 subSec));
end System1;
// The following yields a final override
    error.
System1 system(redeclare Section_B sec);
```

Another limitation arises from the fact that no parameter dialog is generated for the subcomponent redeclared as final, so that other configuration options nested below it are not accessible to the user.

As an alternative, an annotation override may be considered. Indeed, Modelica Association (2021) specifies that a `description` is allowed as part of an `element-redeclaration`. So, if the original component is replaceable, the concrete syntax allows an `annotation-clause` when redeclaring the component. If a replaceable component `subSec` is declared inside `PartialSection` as before, the same configuration logic as in Listing 1 could be implemented as follows.

```
model Section_B "Section config B"
 extends PartialSection(
  redeclare replaceable Config3 subSec
  annotation(choices(
   choice(redeclare replaceable Config3
       subSec),
   choice(redeclare replaceable Config4
       subSec))));
end Section_B;
```

However, although it conforms with Modelica Association (2021), the above syntax is not interpreted by any of the Modelica tools we tested.

5 Connections

The `connect` equations for signal variables in the templates use expandable connectors, also called control busses, which have the following useful properties. The set of variables in an expandable connector is augmented whenever a new variable is connected to an instance of the class. Thus, there is no requirement to pre-declare variables, and in fact we do not pre-declare any variable within the control bus. Variables that are potentially present but not connected are eventually considered as undefined, i.e., a tool may remove them or set them to a default value. Not all variables need to be connected, and therefore the control bus does not need to be reconfigured depending on the model structure.

Like any other Modelica type, expandable connectors can be used in array instances. A typical use case is to connect control signals from a set of terminal units to a supervisory controller of an air handling unit. In our experience, some care is required when handling such array instances to maximize support by different Modelica compilers, especially when nested expandable connectors are involved. We have opted for the pragmatic rule of limiting the number of nested expandable connectors to one. In other words, a control bus may have none or one sub-bus.

Also, we use local instances of array sub-busses to force the compilers to assign the dimensionality to the correct variable. For instance, let us consider the following model where the variables nested under the `bus` object are not pre-declared.

```
model Control
 parameter Integer nDim1 = 2, nDim2 = 1;
 Modelica.Blocks.Examples.
     BusUsage_Utilities.Interfaces.
     ControlBus bus;
 Modelica.Blocks.Sources.RealExpression exp
     [nDim1, nDim2](y=fill(fill(1, nDim2),
     nDim1));
equation
 connect(exp.y, bus.subbus.y);
end Control;
```

Some compilers assign the dimensionality of `exp.y` (that is equal to two) to `bus.subbus.y`, while the template developer may expect both `bus.subbus` and `bus.subbus.y` to have a dimensionality of one. Other compilers will reject such a model. Therefore, the following implementation is used instead.

```
model Control
 parameter Integer nDim1 = 2, nDim2 = 1;
 Modelica.Blocks.Examples.
     BusUsage_Utilities.Interfaces.
     ControlBus bus;
 Modelica.Blocks.Sources.RealExpression exp
     [nDim1, nDim2](y=fill(fill(1, nDim2),
     nDim1));
protected
 Modelica.Blocks.Examples.
     BusUsage_Utilities.Interfaces.
     SubControlBus subbus[nDim1];
equation
 connect(exp.y, subbus.y);
 connect(subbus, bus.subbus);
end Control;
```

Most compilers we have tested can handle the above implementation, and `bus.subbus` and `bus.subbus.y` are necessarily assigned a dimensionality of one. Furthermore, having the instance of the sub-bus as a protected element of the control block instead of a pre-declared variable inside the main control bus avoids binding equations for the dimension parameters wherever the control bus is used.

Finally, we use strict naming conventions for all components, including signal variables, which support a natural syntax for the `connect` equations. For example, connecting the measurement signal yielded by the supply air temperature sensor component to the bus variable used by the controller is done with: `connect(TAirSup.y, bus.TAirSup)`. Connecting the supply fan command and feedback signals to the corresponding sub-bus is done with: `connect(fanSup.bus, bus.fanSup)`. Connecting all signal variables for the air intake section described in subsection 4.3 is done with: `connect(secAirInt.bus, bus)`, where the main bus of the template is used in the section class, because the section itself contains nested

components with sub-busses (such as dampers), so the number of nested levels is effectively limited to one.

6 Design and Operating Parameters

When trying to meet the requirements of subsection 3.3, the main difficulty is that conditional declarations cannot be used for parameters, since a "component declared with a condition attribute can only be modified and/or used in connections" (Modelica Association 2021). We therefore considered the use of an external data file and the use of record classes and chose the latter in our implementation for the reasons explained below.

6.1 External Data File

At first glance, using an external data source, such as a JSON parameter file, is promising because it eliminates the need for parameter propagation. Instead, each component can retrieve the required parameter values by invoking accessor functions. The Modelica library ExternData (Beutlich and Winkler 2021), which we used for this purpose, provides accessor functions for each predefined variable type (`Real`, `Integer`, `Boolean`, `String`) with dimensionality up to 2.

However, we had to scale back our original requirements from subsection 3.3 due to the inherent limitations of using literal constants from an external file. For instance, referencing existing equipment data records from the Modelica Buildings Library is not possible because a class cannot be instantiated by passing the class name as a string. Similarly, binding equations cannot be used to express relationships between parameters of different systems because there is no built-in function to interpret a string as Modelica code and evaluate it. Also left open is the question of how to create the structure of this external data file so that only the parameters required for the actual system configurations are exposed, although automation could address this problem.

More importantly, many structural parameters need to be stored in the external data file, for example, to specify the size of the parameter array for a multi-unit component. Ideally, the value for these parameters should be assigned in this file. However, as structural parameters, they must be evaluated at compile time. FMI-compliant compilers (such as Modelon Impact) can handle this well as the compiler flags that the accessor function must be executed at compile time. This is not the case with other compilers. Dymola, for example, requires the function annotation `__Dymola_translate = true` to force compile time function execution, even though the parameter declaration is already annotated with `Evaluate = true`. This raises concerns about the impact on the translation time, since *each* function call requires the creation of an external object and access to the external file with `fopen`, even though all function calls target the same external file.

It gets even worse when the compiler also stores the values for some variable attributes in the translated model, either by legacy (e.g., `nominal`) or because they are used in symbolic processing (e.g., `min` and `max`). Then all *value* parameters used in bindings of these attributes have to be evaluated as well, causing a significant translation overhead. This marked the end of our attempt to use an external parameter file.

6.2 Record Class

As an alternative, we resorted to record classes to handle parameter assignment from the top level and propagation throughout the instance tree. The use of record classes benefits from the following features of the Modelica language. Records are the only composite components allowed in bindings, and the only composite components of an instance that can be accessed by another instance. Thus, the following conforms with Modelica Association (2021) and records are the only specialized classes that support such constructs, which largely help reduce the number of binding equations when propagating parameters.

```
record Rt "Template record"
 parameter Rc c;
end Rt;

record Rc "Component record"
  parameter Real p=1;
end Rc;

model Template
 parameter Rt dat;
 Component c(final dat=dat.c);
end Template;

model Component "Component"
 parameter Rc dat;
end Component;
```

Specifically, in our case, two record classes are developed for each component model and instantiated within the model's interface class. The first record contains all configuration parameters (structural parameters). This can be considered as the "signature" for a given system configuration, accessible from any component and any template. The second record contains the *full set* of design and operating parameters (value parameters) covering all possible configurations, as well as an instance of the first configuration record.

The inclusion of configuration parameters makes it possible to disable input fields in the parameter dialog if the parameters are not needed for a particular configuration by using the annotation attribute `enable`. Also, if `enable = false`, no value can be assigned to this parameter (Modelica Association 2021) although this is a non-normative part of the language specification, and some compilers may issue warnings or errors if no value is assigned to a parameter, even with `enable = false`. In addition, some compilers require that the `start` attribute be assigned for parameters that have no assignment.

The inclusion of all value parameters is a requirement to ensure plug compatibility with the interface class, which

serves as the constraining type and supports parameter propagation via a single binding equation.

Although simple by design, the implementation of this parameterization logic proves tricky, as shown in Listing 2.

Listing 2. Minimal working example illustrating the use of parameter records.

```
model Template
 final parameter ConfigTemplate config(
  comp=comp.config) "Configuration record";
 parameter DataTemplate data(
  config=config) "Data record";
 Component comp(final data=data.comp)
  "Component";
end Template;

model Component
 parameter Integer typ;
 final parameter ConfigComponent config(
  typ=typ);
 parameter DataComponent data(config=config
    );
 // Parameter myPar needed only if typ==1.
 SubComponent1 sub1(myPar=data.myPar)
  if typ==1;
end Component;

model SubComponent1
 parameter Real myPar;
end SubComponent1;

record DataTemplate
 // Annotation enable used in lieu of final
     binding in instance to avoid final
    override.
 parameter ConfigTemplate config
  annotation(Dialog(enable=false));
 parameter DataComponent comp(
  config=config.comp);
end DataTemplate;

record DataComponent
 parameter ConfigComponent config
  annotation(Dialog(enable=false));
 // Explicit start attribute is needed to
    avoid the warning: "The following
    parameters don't have any value"
 parameter Real myPar(start=0)
  annotation(Dialog(enable=config.typ==1));
end DataComponent;

record ConfigTemplate
 parameter ConfigComponent comp;
end ConfigTemplate;

record ConfigComponent
 // Annotation Evaluate is needed to avoid
    the warning: "The following parameters
     don't have any value"
 parameter Integer typ
  annotation(Evaluate=true);
end ConfigComponent;

// To use the template, they can be
    instantiated as follows:
// If typ=1, the data need to be set to
    assign parameter myPar.
replaceable Template system1(
 comp(typ=1), data(comp(myPar=1)));

// The following instance does not require
    assigning parameter myPar.
replaceable Template system2(comp(typ=2));
```

The comments inserted in this listing give an insight into our experience and show that some ad hoc rules from the various Modelica tool vendors make a generic implementation difficult and lead to a lengthy trial-and-error process. Also, a parameter with `enable = false` remains in the variable namespace, and compilers use the value of the `start` attribute to initialize the parameter when it is unassigned. So we still need to assign a value to this attribute and guard against corner cases, such as division by zero of another variable attribute that has a binding with this parameter, or zero-sized arrays of records. The situation gets worse when dealing with UI/UX features as, in the above example, the input field for the parameter `system.data.comp.m_flow_nominal` may remain enabled in the parameter dialog, even though the configuration parameter `system.data.comp.config.typ` evaluates to `2`. This is the case with many Modelica tools we tested, even for such a minimal example where class name lookup is kept as simple as possible. In practice, and as illustrated in subsection 4.3, templates require complex class structures with multiple levels of inheritance and composition, or the use of **`outer`** components that further limit the support by various tools.

Our ultimate goal entails even more demanding requirements for interpreting annotation attributes at UI runtime. Specifically, we want to read the configuration parameters of template instances from an object at the top level of the simulation model, such as with the following construct. Note that we use a generic **`class`** construct as opposed to the specialized class **`record`** because the latter does not allow for **`outer`** elements.

```
class DataAllSystems
 outer Template system;
 parameter DataTemplate data_system(
  config=system.config);
end DataAllSystems;

parameter DataAllSystems data;

inner replaceable Template system(
 comp(typ=2),
 final data=data.data_system);
```

The top-level component `data` can thus serve as a single object for storing all design and operating parameters, displaying only those required for the particular system configurations, thus satisfying the requirements of subsection 3.3. This structure is composed of records for each system and its components, so existing equipment data records from the library can be easily reused. Vectorized instances are also possible for systems with multiple

units, and different *classes* can be used in them as long as they have the same *type*. Parameterization of multiple units with different properties is therefore straightforward and most compilers allow these arrays to be populated on-the-fly using record functions as shown below.

```
parameter ElectricReformulatedEIR.Generic
    data[2] = {
  ElectricReformulatedEIR.
      Carrier_19XR_1234kW_5_39COP_VSD(),
  ElectricReformulatedEIR.
      Carrier_19XR_1143kW_6_57COP_VSD()};
```

All in all, the use of Modelica records offers great potential, but suffers from uneven support from various compilers, especially when using array instances or evaluating the `enable` attribute at UI runtime, and for use cases requiring complex class structures. At the beginning of the development of system templates, we certainly did not expect that the handling of *value* parameters would be the most difficult part and take the most development time—estimated to be over 30 %—with a final result that we still judge to be far from optimal. We believe that the lack of conditional declarations of *value* parameters is the biggest obstacle to the development of complex system templates and makes the parametric polymorphism of the Modelica language cumbersome in practice.

7 Control Diagram

To meet the requirements of subsection 3.6, all data needed to create the diagram for a given system are included in the template class in the form of Modelica graphical annotations. Figure 3 shows the diagram view of the boiler plant template and gives an example of the direct graphical feedback that can be obtained about a particular system configuration. It also illustrates the capability of the template to adapt to different equipment and control specifications by programmatically generating the necessary objects for equipment, actuators and sensors, and resolving the hydronic routing of components and control signal connections without user intervention.

The template components include `Bitmap` primitives in the icon layer to reference equipment symbols provided as vector graphics in SVG format (W3C 2003). Although Modelica tools render them as raster images in the diagram view, a tool can use the referenced SVG files to create diagrams that conform to industry standards and are accurate at the pixel level. The visibility of these graphical objects is controlled with the annotation attribute `visible` and bindings to the template class configuration parameters. Due to the lack of a "group" element in the Modelica Language Specification—as opposed to the SVG specification (W3C 2003) which includes the `'g'` element—dealing with complex graphical objects using only native graphical primitives would require many duplicates of the `visible` attribute and its binding equation, which is the main reason we rely on external SVG files. Text in equipment symbols is handled separately, so it can be flipped or rotated independently of the component symbol.

Piping systems are represented directly by the graphical annotations of the `connect` equations, with an explicit `visible` attribute added in the case of conditional components. This is necessary because Modelica tools, while removing the corresponding `connect` statements at translation, generally do not provide direct graphical feedback and the connection lines remain visible in the diagram view. In the case of air systems, we use separate graphical annotations in the diagram layer to represent the ductwork. The connection lines are then graphical artifacts that should be deleted when creating the final control diagram.

8 Validation

We have implemented a comprehensive test workflow to verify that all system configurations supported by a given template are implemented correctly. However, generating the list of these supported configurations was not straightforward, considering that elaborate class parameterization techniques are used together with `choices` annotations to implement the actual decision tree (see subsection 4.3). Thus, we recreated this set of options in a standalone script that first builds the list of all possible combinations of structural parameters and **`redeclare`** clauses, and then prunes this list based on exclusion patterns that must be manually specified by the template developer. Then, simulations are run for all the resulting class modifications. For example, this results in over 2000 simulations for the chiller plant template and nearly 1000 simulations for the boiler plant template. Due to the computation load, we only trigger these tests in our continuous integration workflow when the checksum computed for all classes in the `Templates` package changes.

Currently, the percentage of tool coverage is about 60 %,[1] but is steadily growing with the updates of the Modelica compilers released by various vendors.

9 Conclusion and Future Work

Our experience with relying entirely on the Modelica language to create user-friendly models for systems with thousands of configurations and closed-loop controls has shown that complexity is not necessarily where one expects it to be. Originally, we saw the lack of conditional element redeclaration as the main obstacle, especially since there is convincing work already pointing out this deficiency and proposing some changes to allow for better class parameterization (Zimmer 2010). Although these proposed changes would have made our task easier, it appeared to us that advanced model configuration could be achieved with the current syntax, provided that com-

[1] The percentage of tool coverage is calculated as the ratio between the number of successful tests and the number of tests, where the number of tests is equal to the number of templates times the number of compilers tested.

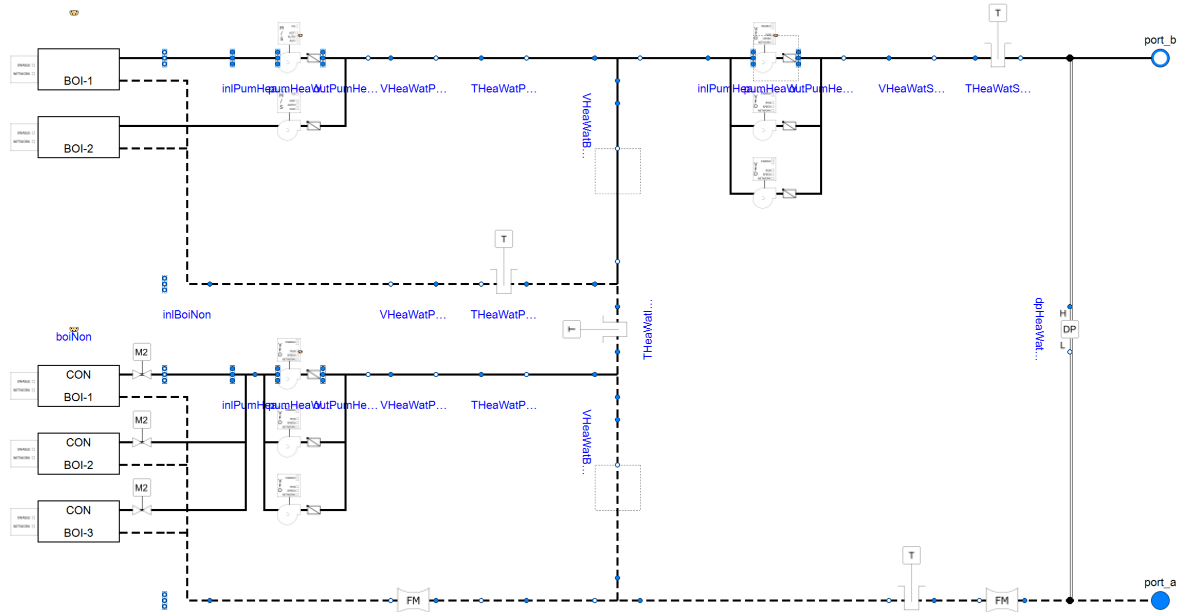

Figure 3. Diagram view of the boiler plant template (as rendered by Dymola) in a primary-secondary configuration, with three condensing boilers equipped with headered variable speed pumps and two non-condensing boilers equipped with dedicated constant speed pumps.

ponents have well-designed interface classes and that UI features are used in conjunction with pure language constructs for object manipulation.

However, the assignment of equipment parameters and their propagation throughout the instance tree proved to be the most problematic. The main reason is that the conditional declaration of parameters used in binding equations, as opposed to connectable components, is not allowed. We believe that this restriction is what most limits the templating capabilities of the Modelica language. In the absence of such a feature, we are left with only a non-normative part of the specification, namely the use of the attributes `enable` and `start` in parameter declarations. With this pattern, a parameter can remain unassigned if it is not needed for a given configuration where `enable` evaluates to `false`. But this approach is fraught with some issues. First, the fact that it is a non-normative feature limits the support of our templates by various Modelica compilers, which may issue a warning or error in case of unassigned parameters. Second, a disabled parameter remains in the variable namespace and compilers use the value of the `start` attribute to initialize the parameter if it is unassigned. So we still need to assign a value to this attribute, and several corner cases occur that we need to guard against. Finally, the behavior of the UI is almost unpredictable because the interpretation of the `enable` attribute at UI runtime to generate the parameter dialog is not specified. Thus, if the logic to disable a parameter input field fails, it is difficult to determine which constructs are the cause. We believe that a unified UX matters, especially for templating, and that it would be good for the Modelica community if more UI features were normative and much more clearly specified than currently. The uneven support by Modelica compilers of another fundamental structure for handling parameters further complicates the task. Indeed, we have observed and reported many compiler failures with record classes used in non-trivial constructs such as array instances, composite component bindings, or on-the-fly instantiation with record functions. Thus, development work often becomes a tricky navigation around the specific limitations of the various Modelica compilers.

Our next step is to implement templates for entire HVAC systems, from plant to zone equipment, with coupling to thermal zone models. This means assembling templates from templates, and for highly scalable systems. The main difficulty that we foresee arises from the constraint that array elements must be of the same type, which is rarely the case with class parameterization techniques that only aim at type or plug compatibility. Again, we think that the suggestions from Zimmer (2010) would be very valuable to use an array of model parameters for this purpose. We have some alternatives, such as building container classes around templates, or using multiple array instances for the same system type. This also means that our developments have not yet reached the highest level of complexity, even though it sometimes seems that they have already exceeded the complexity that most compilers can handle.

10 Data Availability

The templates discussed in this paper are available in the feature branch `issue3266_template_HW_plant` from commit `e15d845`, and are planned to be released in future versions of the Modelica Buildings Library.

Acknowledgements

This work 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.

References

ASHRAE (2021). *Guideline 36: High-Performance Sequences of Operation for HVAC Systems*. Guideline. Atlanta, GA.

Beutlich, Thomas and Dietmar Winkler (2021). "Efficient Parameterization of Modelica Models". In: *Proceedings of 14th Modelica Conference*. Linköping, Sweden. DOI: 10.3384/ecp21181141.

Broman, David, Peter Fritzson, and Sébastien Furic (2006-09). "Types in the Modelica Language". In: *Proceedings of the 5th International Modelica Conference*. Vienna, Austria: The Modelica Association and arsenal research. URL: https://modelica.org/events/modelica2006/Proceedings/sessions/Session3c3.pdf.

Dassault Systèmes AB (2023-03). *Dymola: Dynamic Modeling Laboratory. Full User Manual (Dymola 2023x Refresh 1)*. Lund, Sweden.

Greenwood, Michael Scott et al. (2017). "A Templated Approach for Multi-Physics Modeling of Hybrid Energy Systems in Modelica". In: DOI: 10.2172/1427611.

Kågedal, David and Peter Fritzson (1998-07). "Generating a Modelica Compiler From Natural Semantics Specifications". In: *Proceedings of the Summer Computer Simulation Conference*. Reno, Nevada.

Long, Nicholas et al. (2021). "Modeling District Heating and Cooling Systems With URBANopt, GeoJSON to Modelica Translator, and the Modelica Buildings Library". In: *Proceedings of the Building Simulation Conference*. DOI: 10.26868/25222708.2021.30943.

Modelica Association (2021-02). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.5*. Tech. rep. Linköping: Modelica Association. URL: https://specification.modelica.org/maint/3.5/MLS.html.

Modelon AB (2023a). *Modelon Impact Helpcenter*. Lund, Sweden. URL: https://help.modelon.com/latest/.

Modelon AB (2023b). *OPTIMICA Compiler Toolkit. Version 1.43.4*. Lund, Sweden.

Nytsch-Geusen, Christoph et al. (2017). "Template Based Code Generation of Modelica Building Energy Simulation Models". In: *Proceedings of the 12th International Modelica Conference*. Prague, Czech Republic: Linköping University Electronic Press. DOI: 10.3384/ecp17132199.

W3C (2003-01). *Scalable Vector Graphics (SVG) 1.1 Specification*. Tech. rep. URL: https://www.w3.org/TR/2003/REC-SVG11-20030114/REC-SVG11-20030114.pdf.

Wetter, Michael, David Blum, et al. (2019-05). *Modelica IBPSA Library v1*. DOI: 10.11578/dc.20190520.1.

Wetter, Michael, Paul Ehrlich, et al. (2022). "OpenBuildingControl: Digitizing the Control Delivery From Building Energy Modeling to Specification, Implementation and Formal Verification". In: *Energy* 238, p. 121501. DOI: https://doi.org/10.1016/j.energy.2021.121501.

Wetter, Michael, Milica Grahovac, and Jianjun Hu (2018-08). "Control Description Language". In: *1st American Modelica Conference*. Cambridge, MA, USA. DOI: 10.3384/ecp1815417.

Wetter, Michael, Jianjun Hu, et al. (2021). "Modelica-json: Transforming Energy Models to Digitize the Control Delivery Process". In: *Proceedings of the IBPSA Building Simulation Conference*. Brugge, Belgium.

Wetter, Michael, Wangda Zuo, et al. (2014). "Modelica Buildings library". In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506.

Wüllhorst, Fabian et al. (2023-02). "BESMod - A Modelica Library pProviding Building Energy System Modules". In: pp. 9–18. DOI: 10.3384/ECP211869.

Zimmer, Dirk (2010). "Towards Improved Class Parameterization and Class Generation in Modelica". In: *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Oslo, Norway: Linköping University Electronic Press. URL: https://ep.liu.se/ecp/047/004/ecp4710004.pdf.