Bloqqi: Modular Feature-Based Block Diagram Programming

Fors, Niklas; Hedin, Görel

Published in:
International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)

DOI:
10.1145/2986012.2986026

2016

Document Version:
Peer reviewed version (aka post-print)

Bloqqi: Modular Feature-Based Block Diagram Programming

Niklas Fors Görel Hedin

Department of Computer Science, Lund University, Sweden
{niklas.fors, gorel.hedin}@cs.lth.se

Abstract

Automation programming is typically done using blocks and dataflow connections, in diagram languages that support user-defined block types. Often, these types are intended to be instantiated and connected to other blocks in common patterns, corresponding to anticipated variability. We present the new language mechanisms of *wirings* and *recommendations* that allow these intentions to be encoded as features in libraries. A wiring describes *how* a given block is typically connected to other blocks, and a recommendation describes *where* such a wiring is typically applied as a feature. This allows feature-based wizards to be generated for user-defined libraries, making it easy to construct applications that make use of the encoded patterns.

Categories and Subject Descriptors D.3 [*Programming Languages*]

Keywords dataflow, inheritance, variability, visual, wiring, recommendations, redeclare, intercept, control systems, process automation

1. Introduction

Block diagrams, with blocks and connections, are common in modeling, simulation, and automation programming. Example languages include both proprietary languages like *LabView* from National Instruments, *Simulink* from MathWorks, and *ControlBuilder* from ABB, as well as open community-developed languages like *Modelica* (Modelica 2012), and *SysML* (SysML 2015). Typically, the connections are directed and describe dataflow between the blocks.[1] These languages typically support *user-defined blocks* to encourage hierarchical decomposition and the construction of reusable libraries, for example for controllers, motors, valves, tanks, pumps, etc. The content of a block can be defined by an inner block diagram, or using some other notation, like structured text[2]. These user-defined blocks are usually intended to be combined in specific predefined ways. For example, in Proportional Integral Derivative (PID) control, there are many different ways in which a basic controller block can be combined with other blocks to improve control. Examples include support for feed-forward, gain scheduling, cascade control with master-slave controllers, override control, etc. (Åström and Hägglund 2006). While any such combination can be coded up as wired blocks, the problem is that libraries of components do not encode the intended variability, so the domain engineer will need to manually select and wire all individual components, which is both time-consuming and error-prone. In process automation, this is an important problem, as programming a control system for an industrial plant is a very large engineering effort.

One possible work-around is to provide a number of pre-configured block types in the library, one for each combination of features. However, this leads code duplication and to an exponential number of block types in the library, making it impractical. Another more practical but still insufficient work-around is to provide parameterized block types, that contain support for all features, but where the actual features used are selected by extra input parameters. However, this leads to diagrams that are very complex to understand and use, and where only a subset of the functionality is actually used at runtime. Furthermore, all variability needs to be anticipated in advance with this solution.

In this paper, we provide a solution to this problem by allowing the variability to be explicitly encoded, making it easy for the domain engineer to select the desired features, and resulting in simple diagrams that only contain the desired functionality. Furthermore, the encoding is modular, so all features do not have to be anticipated when constructing a library: additional features can be added in separate library modules, and the resulting diagrams can be edited to add special features that are not in any library at all.

In our solution, we propose the novel language constructs of *wirings* that describe how blocks are typically connected,

[1] Modelica is an exception in this respect, as it uses undirected connections that correspond to equations.

[2] Structured text is one of the languages in the IEC 61131 standard for programmable logic controllers.

http://dx.doi.org/10.1145/2986012.2986026

and *recommendations* that describe where such wirings are typically applied. A given block could be inserted at many different places in a particular block diagram, but often, it is advantageous to insert it at a specific place, serving the role of a *feature* that the diagram can include or not. An example could be a controller diagram that includes a feed-forward block or not. If the feed-forward block is included, it is intended to be inserted in a specific way. A *wiring* defines *how* a particular block, like the feed-forward block, should be connected in a diagram. A *recommendation* for a diagram defines *where* in the diagram a particular wiring is intended to be inserted.

The recommendations can be used to create smart editing support in the form of a feature-based wizard. The wizard can be automatically derived, and the user can use it to select or change the desired combination of features for a particular diagram. These mechanisms do not hinder the user in making other modifications, or in wiring together blocks in unanticipated ways. However, being able to capture anticipated variability this way can simplify and speed up program construction substantially, by allowing common patterns to be applied very quickly.

The new mechanisms are general programming constructs, applicable to data-flow block diagram programming, supporting the construction of modular extensible libraries for different domains. The modularity is very important as it allows libraries, and therefore the derived wizards, to be extended with new patterns of interest for a given application domain or even for an individual plant.

The language mechanisms proposed in this paper are based on *diagram inheritance*: a subtype can extend a supertype diagram with additional connections and blocks, and can also specialize the behavior in the supertype, in analogy to method overriding in object-oriented languages. In particular, specialization is supported by *connection interception* (Fors and Hedin 2014) and *block redeclaration* (Modelica 2012). Connection interception allows the subtype to *intercept* connections defined in the supertype, that is, to reroute the connection to go via another block or subnet. Block redeclaration allows the subtype to replace the type of a block defined in the supertype by a more specialized type.

As a proof of concept, we have implemented the proposed language mechanisms, *recommendations* and *wirings*, in an experimental data-flow based language, Bloqqi, used for programming automation control systems. Bloqqi has both textual and visual syntax. We have implemented a compiler and a visual editor for the language that is released as open source.[3] The mechanisms have been developed in collaboration with ABB Control Technologies, with the goal of improving reusability mechanisms in the control domain for the process industry. We have been collaborating with ABB for almost four years with monthly meetings. The language has been designed iteratively using their feedback as input and has been inspired by their examples.

[3] `https://bitbucket.org/bloqqi`

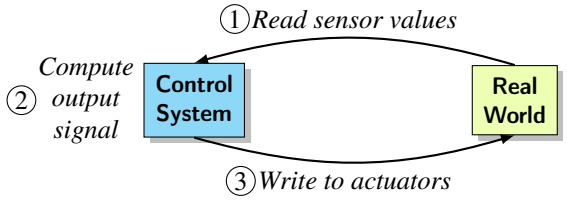


Figure 1. Periodic execution. All steps are performed in each period.

Like typical control systems, programs in Bloqqi are executed periodically, for example, 10 or 100 times per second. Sensor values are read in the beginning of the period, and are used to compute output values that are sent to actuators to control the process. This is illustrated in Figure 1. The control program may also have states that are stored between the periods. For instance, integrating controllers may use old sensor values to improve the control signal.

Before continuing with a motivating example (Section 2) and a discussion of the Bloqqi language (Section 3), we will first summarize the main contributions of this paper. They are:

- **Wirings.** A new language mechanism that describes how blocks of a user-defined type can be inserted and connected (Section 4).
- **Recommendations.** A new language mechanism that modularly describes where a wiring is recommended to be applied, serving the role of an optional feature (Section 5).
- **Recommendation composition**. An explanation of how recommendations can be used together with subtyping to automatically compute hierarchical feature wizards, and how feature interactions can be automatically discovered and modularly resolved (Section 6).
- **Source interception.** A generalization that allows interceptions to be applied not only at the target of a connection, but also at the source (Section 6.3).
- **Editing support.** Editing support for feature-based programming, to create recommendations by example, and to support staged configuration (Section 7).
- **Evaluation.** We have implemented the new mechanisms, and show that our approach requires much less effort in constructing diagram variants, compared to manual editing (Section 8).

We end with a discussion of related work (Section 9) and a concluding discussion (Section 10).

2. Motivating Example

A basic PID controller periodically reads a sensor value and computes an actuator value (using the history of sensor values) in order to control a system towards a *set point*, i.e.,

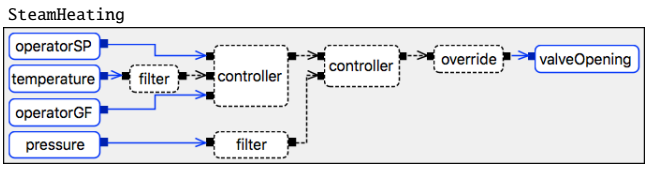


Figure 2. Diagram for temperature control using steam. Solid parts are created manually. Dashed parts via the wizard.

a desired value. For real systems, there are many different variants on this basic structure.

As a motivating example, we will consider controlling the temperature of a fluid in a tank, by using steam that is let in through a valve. When the valve is opened, the effect of heating will depend on the current steam pressure, which in turn may vary substantially since the steam may be used elsewhere in the factory, for example, to heat other tanks.

This is a classic example of when *cascade control* is needed, i.e., where two controllers are connected in a master-slave configuration (Åström and Hägglund 2006). The master controller reads the current temperature to compute how much steam is needed. The slave controller uses this value as its set point, and reads the current steam pressure to compute how much to open the valve in order to obtain that amount of steam.

A Bloqqi diagram for temperature control using steam is shown in Figure 2. There are two sensors, `temperature` and `pressure`, one actuator, `valveOpening`, and two values that can be set by the human operator: `operatorSP` for setting the desired temperature, and `operatorGF` to set a suitable gain factor for the master controller, controlling how fast to heat. All these blocks are implemented using primitives to communicate with the sensors, actuators, and operator interface.

The rightmost `controller` (the slave) takes the output of the leftmost `controller` (the master) as its set point. Further embellishments include `filters` on both controller inputs, an extra input port on the master controller to receive the gain factor, and an `override` filter before sending the signal to the `valveOpening` actuator, to protect the valve from opening too much which might damage the equipment.

Without our new language constructs, the user would construct the control program from scratch in the visual editor, by instantiating existing block types for different kinds of filters and controllers, creating specialized types as necessary, and explicitly wiring the components together. However, this requires a lot of detailed knowledge of many different block types, and knowledge of how they are intended to be combined. Instead, we provide the possibility of embedding this knowledge into library types from which a wizard can be automatically generated, allowing the user to simply select the desired parts. These parts are then automatically inserted in the correct way into the program, wired together, and combined in the right order.

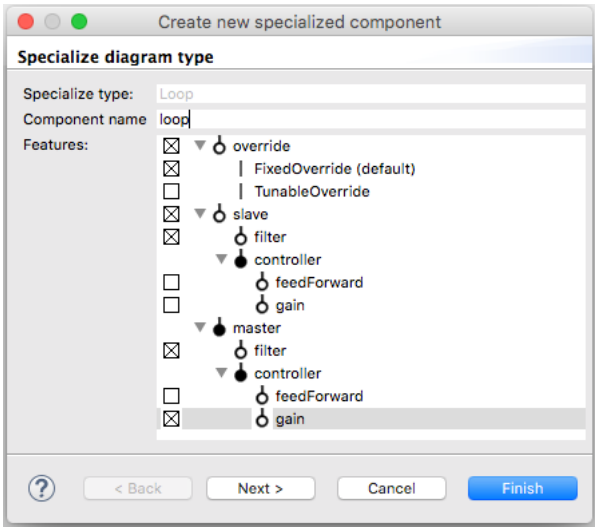


Figure 3. Wizard for creating a control loop.

In this example, the wizard is used to create the control logic (dashed). This is done by creating an instance of `Loop` (a library block type), and selecting appropriate features in the wizard. The sensors, actuators, and the operator-set value blocks (solid), are created manually, as in a usual block diagram editor.

Figure 3 shows the wizard for `Loop`. The possible selections are shown in a structure similar to a feature diagram with mandatory, optional, and alternative parts (Kang et al. 1990). Black circles correspond to mandatory parts, white to optional parts, and vertical bars are used for showing alternatives. In this case, the `master` part is mandatory, whereas the `slave` part is optional. Both the master and slave parts have a `controller` and an optional `filter`. The controllers have optional `feedForward` and `gain` parts. Furthermore, there is an optional `override` part for which there are two alternatives: `FixedOverride` and `TunableOverride`. A corresponding feature model for the `Loop` type is shown in Figure 4. By selecting the parts as shown in Figure 3, the dashed parts of the diagram in Figure 2 are created, complete with all the internal wiring (dashed arrows). The user completes the diagram by connecting the sensors, actuators and operator values to the control logic (blue solid arrows). The example illustrates how a user can construct a complex diagram very quickly and easily, just by selecting features in the wizard, and without having to remember what block types to use and how to wire them together.

The diagram in Figure 2 is an interactive view of an underlying control program, and all details in this program are not immediately visible, but can be accessed by interactive means. For example, the names of ports can be viewed by hovering over the ports, and the type and content of a block can be viewed by double-clicking on the block.

The dashed parts of the diagram is actually an instance of an anonymous subtype of the block type `Loop`, but which is

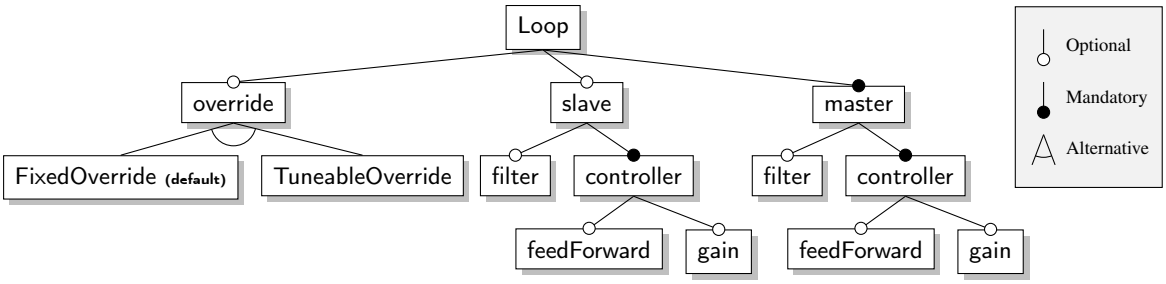


Figure 4. Corresponding feature model for the type `Loop` in Figure 3.

automatically inlined to show interesting parts of the inner structure. For the `Loop` type, the library developer has chosen to expose the controllers and the filters. However, the selected `gain` component is shown only as an extra port on the master controller, and the corresponding block is visible only if the user double-clicks on the master block to show its interior components. This ability to selectively do visual inlining is important because it allows the final diagrams to look like manually drawn diagrams, where the important components are visible, rather than reflecting the reusability-based type structure which is usually not of interest to the end users (domain engineers) (Fors and Hedin 2016).

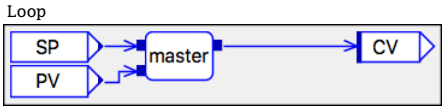
The library is constructed with novel language constructs that allow new features to be added modularly, and which will then automatically turn up in the wizards. The technique is general, and modeling a control loop is just an example. Another example could be to model an engine with one or more motors, different alternatives for start-up and shut-down logic, etc.

3. The Bloqqi Language

To be able to experiment with our new language constructs for variability, we have developed the language *Bloqqi*. Bloqqi has blocks and directed connections inspired by ABB's Control Builder tool which in turn builds on Function Block Diagrams in the IEC 61131 standard. Additionally, Bloqqi has block type inheritance and redeclaration inspired by mechanisms in Modelica (Modelica 2012). Bloqqi furthermore supports *connection interception*, where a connection in a supertype can be intercepted and rerouted through a block subnet (Fors and Hedin 2014).[4] Similar to Modelica, Bloqqi has both a textual syntax that covers the complete language and a visual syntax that covers block configuration.

In this section, we will introduce the basic language constructs in Bloqqi, and discuss how inheritance, redeclaration, and connection interception can be used to encode a variant as a subtype. The subsequent sections will then introduce the new language mechanisms of wirings and recommendations that allow variants to be optionally applied and combined.

[4] (Fors and Hedin 2014) describes an earlier version of Bloqqi, then called PicoDiagram.




```
diagramtype Loop(SP: Int, PV: Int => CV: Int) {
  master: ControllerPart;
  connect(SP, master.SP);
  connect(PV, master.PV);
  connect(master, CV);
}
```

Figure 5. Visual and textual representation of a basic control loop with input parameters `SP` and `PV` (left of =>), output parameter `CV` (right of =>), and containing a `master` controller part. Connections are used for sending the `SP` and `PV` values to the master, and its output to `CV`.

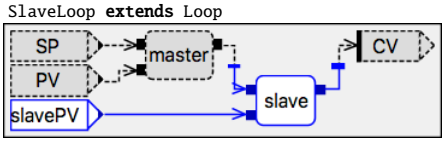
Bloqqi computations can be programmed using diagrams, describing the relation between input and output values. A diagram consists of input parameters, blocks, output parameters, and connections between these entities. Connections are directed, forming a partial directed graph that describes data-flow. A diagram is also a block type, and can be instantiated in another diagram as a block, leading to hierarchical structures. The input and output parameters of the block type will then be shown as input and output ports on the block.

Figure 5 shows an example diagram in both visual and textual syntax. The diagram describes the basic structure of the type `Loop` that was specialized in section 2. The diagram computes the control value (`CV`) by sending the set point (`SP`) and the process value (`PV`) to the master part (`master`) that contains a controller. The block `master` has the type `ControllerPart`, which is also defined by a diagram, similar to `Loop`. The input and output parameters of `ControllerPart` are shown as input and output ports on the block `master`. Thus, the interface of the block is dependent on its type.

3.1 Inheritance

Bloqqi supports inheritance, where a diagram type S can extend another diagram type T. We say that S is a subtype of T and that T is a supertype of S. The subtype *inherits* all parameters, blocks and connections that are declared locally in its supertypes (transitively), which means that these elements are implicitly copied from the supertypes

SlaveLoop **extends** Loop




```
diagramtype SlaveLoop(slavePV: Int) extends Loop {
  slave: ControllerPart;
  intercept CV with slave.SP, slave.CV;
  connect(slavePV, slave.PV);
}
```

Figure 6. The type Loop is extended with a `slave` controller part. Inherited elements are dashed/black/grey and local elements are solid/blue.

to the subtypes. The subtype can also declare new parameters, blocks and connections, and specialize existing elements. Since a subtype can declare new parameters, the interface of the subtype can be larger (contain more parameters) compared to its supertype.

Ordinary object-oriented programming languages support dynamic allocation and references to objects with static and dynamic types. In contrast, Bloqqi (like Modelica) supports only static allocation of blocks, and there are no reference variables. Therefore, the types of all blocks are known statically, and the compiler can report all type and flow errors.

3.2 Interception: Specializing Connections

A diagram subtype can specialize the behaviour of its supertype using *connection interception* (Fors and Hedin 2014). A connection interception allows a connection defined in a supertype to be intercepted, that is, to reroute it to go via another block or network of blocks. In effect, the intercepted connection is replaced by two new connections. This way, a connection can be specialized.

The connection that is intercepted can be identified by either the source or the target of the connection, which we call *source interception* and *target interception*, respectively. This is a generalization of the previous work, where only target interception is presented. Source interception is described in Section 6.3.

For example, Figure 6 shows a new subtype of `Loop` from Figure 5, which adds a `slave` controller part. The connection to the output parameter `CV` (control value) defined in the supertype is intercepted, that is, the connection will go through the new `slave` block. The connection is identified by the target of the connection, thus the interception is a target interception. The `slave` contains a controller that uses the output from the master controller as its set point, and together with an additional sensor value, decides what control value to actually output.

3.3 Redeclare: Specializing Blocks

Another way for the subtype to specialize the behaviour of its supertype is by *redeclaring* blocks, similar to how this is done in Modelica (Modelica 2012). A subtype can redeclare an inherited block of type T, to be of a type S, where S is a subtype of T.

For example, assume that there is a type `FilterControllerPart` that is a subtype of `ControllerPart`, and which filters the process value before sending it to the controller. We can then create a new subtype of `Loop` from Figure 5 that redeclares the block `master` to the specialized type, as the following code illustrates:

```
diagramtype FilterLoop() extends Loop {
  redeclare master: FilterControllerPart;
}
```

As the examples in this section have shown, inheritance and composition can be used for specifying different variants of diagram types by introducing subtypes. For example, `SlaveLoop` and `FilterLoop` can be seen as variants on `Loop`. However, if all possible variants would be predefined in library classes, this would lead to combinatorial explosion. If instead only basic types, like `Loop`, were available in the library, the domain engineer would need to explicitly add a number of detailed elements, like blocks, parameters, connections, and intercepts, in order to create a specific variant. To support a better way of defining and using variants, we introduce the new constructs of *wirings* and *recommendations*, as described in the following sections.

4. Wirings

A *wiring* describes how blocks of a certain type are intended to be inserted into a diagram. For example, in Figure 6, we can see that an interception and a connection is used for connecting the new `slave` block. With wirings, we can move these two flow statements to a *wiring declaration* for the `ControllerPart` type. We can then *apply* the wiring to insert the `slave` block in a simpler way, without having to explicitly wire it into the diagram.

4.1 Wiring Declaration

A wiring declaration for a type T has a number of formal parameters and a number of flow statements. The formal parameters refer to connection points in the diagram that applies the wiring, and they each have a type and a data-flow direction (input or output). The flow statements describe connections and interceptions involving these formal parameters and ports on T. In the current Bloqqi implementation there can be at most one wiring declaration for each block type.

For example, the type `ControllerPart` used in Figure 6 has the following interface:

```
diagramtype ControllerPart(SP: Int, PV: Int => CV: Int){
  ...
}
```

and the wiring for `ControllerPart` can then be defined as follows:

```
wiring ControllerPart[=>c: Int, p: Int] {
  intercept c with ControllerPart.SP, ControllerPart.CV;
  connect(p, ControllerPart.PV);
}
```

This means that when the wiring is applied, a block of type `ControllerPart` is added at a specific location in a diagram (indicated by the parameters `c` and `p`), at which the flow statements (the interception and the connection) are added. The parameter `c` has *output* direction (indicated by =>), meaning that it should be bound to an actual parameter that is at the target end of a connection, and can therefore be intercepted. The parameter `p` has *input* direction (which is the default), meaning that it can be used as the source of connections.

4.2 Wiring Application

A wiring application can be used to add a block, including all its wiring, at a given location. The location is indicated by passing actual parameters to the wiring. For example, instead of defining `SlaveLoop` as was done in Figure 6, we can equivalently define it in the following simpler way:

```
diagramtype SlaveLoop(slavePV: Int) extends Loop {
  slave: ControllerPart[CV, slavePV];
}
```

In applying a wiring, like above, the actual parameters are bound to the formal parameters. Thus, the following wiring application

```
slave: ControllerPart[CV, slavePV];
```

is equivalent to

```
slave: ControllerPart;
intercept CV with slave.SP, slave.CV;
connect(slavePV, slave.PV);
```

Note that it is possible to apply a wiring in several different places, even within the same diagram.

Note also that both wirings and the application of them is optional. Even if a wiring is defined, it is not necessary to use it to add a block. For example, the following code will add a block of type `ControllerPart` without applying the wiring:

```
slave: ControllerPart;
```

The user will then have to explicitly add the wiring in order to connect the block to the surrounding network.

4.3 Declaring Parameters in Wiring Parameters

Sometimes, we need to add parameters to the enclosing type of the wiring application. An example is the type `Loop` that we have seen before. It is possible to declare a parameter at the same time as applying a wiring, as the following code illustrates, which is again equivalent to the definition in Figure 6.

```
diagramtype SlaveLoop() extends Loop {
  slave: ControllerPart[CV, slavePV: Int];
}
```

Here, the input parameter `slavePV` of `SlaveLoop` is declared in the wiring application instead of in the diagram type header. We can see that `slavePV` is a parameter declaration because it has a type. We can see that it is an input parameter rather than an output parameter, because it lacks the => modifier. An output parameter would be declared as `=>Name: Type`.

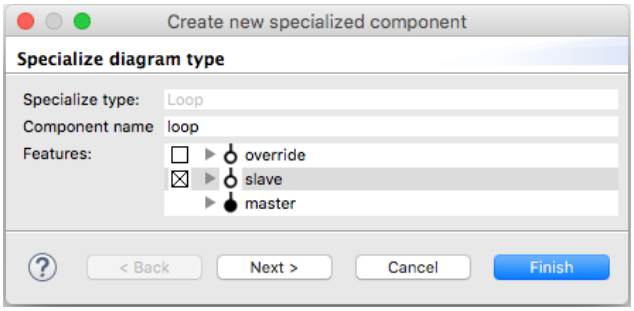


Figure 7. Automatically derived wizard for the type `Loop`, based on recommendations. Using this wizard, a new block can be created with the desired functionality.

Wirings are intended to be used in libraries, together with recommendations which will be described in the next section.

5. Recommendations

A *recommendation* describes how a diagram type can be specialized with optional functionality. These recommendations will then be used to derive wizards, like the one in Figure 3. The wizards are used for creating specialized types for a particular situation, selecting the desired features to be included. For example, a recommendation can suggest that a slave controller part can be added to the type `Loop` in Figure 5. This can be described as follows:

```
recommendation Loop {
  slave: ControllerPart[CV, slavePV: Int];
}
```

A recommendation consists of the name of a diagram type (`Loop` in this case) and recommended features, each expressed as a wiring application. We refer to the block, `slave` in the above example, as a *feature*, and we say that `slave` is a *recommended feature* for type `Loop`. From this recommendation, together with other recommendations for this type, we can automatically derive a wizard for creating specializations of the type `Loop`. Figure 7 shows the derived wizard, where the optional `slave` feature is shown together with an optional `override` feature (added with another recommendation), and the mandatory feature `master`. The latter feature is mandatory since it is declared in `Loop`, whereas the others are just recommended options. The wizard in Figure 7 is the same as the one in Figure 3, but where the suboptions are not yet opened.

Instead of creating one library type for each possible combination: one plain loop, one with the `slave` feature only, one with the `override` feature only, and one with both features, it is sufficient to have a single type `Loop` in the library. To create a `Loop` block, say `loop`, the domain engineer can select in the wizard which of the features to include. The

type of `loop` will then be a generated anonymous subtype of `Loop`, containing the selected features:

```
loop: Loop { slave: ControllerPart[CV, slavePV: Int];};
```

This is equivalent to first defining an explicit subtype, e.g., `SlaveLoop`, as before, and then instantiating it:

```
diagramtype SlaveLoop() extends Loop {
  slave: ControllerPart[CV, slavePV: Int];
}
loop: SlaveLoop;
```

By letting the domain engineer do this choice, the relevant types can be constructed on a demand basis rather than having to predefine all possible combinations, which would have led to combinatorial explosion.

Note that in an application, there might be a need for several loops that have the same set of features. Instead of selecting the features for each of those loop instances, it is possible to give the generated loop subtype a name, and then instantiate it multiple times.

5.1 Modular Recommendations

There can be several recommendations for the same diagram type, defined independently in different library modules. This allows domain library developers to add recommendations to general library types, without having to modify the general libraries. When a wizard is requested for specializing a type, all recommendations will be collected for the type and the wizard is automatically derived based on this information. For example, the recommendation for `override`, as shown in Figure 7, can be defined separately from the `slave` recommendation as follows:

```
recommendation Loop {
  override: Override[CV];
}
```

5.2 Alternative Features

Given a recommended feature `f: F[...]` for a type `T`, the wizard for `T` may include several *alternatives* for `f`. An alternative is added by simply creating a new subtype of `F`. This subtype will then be shown as an alternative in the wizard. Using subtyping for defining alternatives allows them to be defined modularly: an existing library can be extended with new feature alternatives without touching the library.

Note that while mandatory and optional features correspond to block names, alternative features correspond to (sub-)type names. For example, in the feature diagram in Figure 4, the alternative features `FixedOverride` and `TunableOverride` are type names, whereas the other features are mandatory or optional features, indicated by block names.

Bloqqi supports alternatives with or without new parameters, alternative wirings for subtypes, and default alternatives.

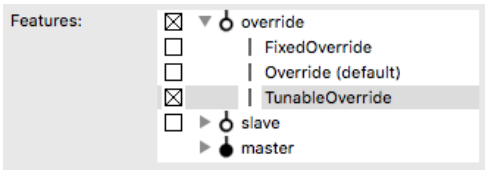


Figure 8. Recommended features for `Loop`. The feature `override` has three alternatives, with one default choice.

Figure 9. Unconnected inner parameters can be exposed as outer parameters.

Simple Alternative

A simple alternative is a new subtype that does not introduce any new parameters. Consider again the recommended feature `override` for `Loop`:

```
recommendation Loop {
  override: Override[CV]
}
```

We can add a new alternative for `override` simply by adding a subtype to `Override` as the following code illustrates.

```
diagramtype FixedOverride() extends Override { ... }
```

The new subtype `FixedOverride` makes sure that the control value does not exceed a predefined threshold. The subtype will be shown as an alternative for `override` in the derived wizard for `Loop`.

Alternative with Extra Parameters

A subtype may introduce extra parameters. For example, the following subtype for `Override` introduces an extra input parameter:

```
diagramtype TuneableOverride(limit:Int) extends Override{
  ...
}
```

The new subtype `TuneableOverride` makes sure that the control value does not exceed a threshold defined by the input parameter `limit`. The subtype will appear in the wizard as another alternative for `override`, as shown in Figure 8.

Since the `override` recommendation uses the wiring for `Override`, it does not cover the new parameter introduced in the subtype. If nothing more is done, the new parameter will be hidden inside the new `loop` block, with no connections to/from it. There are several ways of dealing with this. The wizard will detect which parameters are not covered by the wiring, and allow the user to *expose* them as parameters on the created block. This means that a new parameter is created for the specialized type, and it is connected to the extra subtype parameter of the feature.

The mechanism is illustrated in Figure 9, which shows how the user can select to expose the `limit` parameter after selecting the `TuneableOverride` alternative in the wizard of Figure 8. This results in the following `loop` block with an extra parameter `overridelimit` which is connected to the `limit` of the `TuneableOverride` feature:

```
loop: Loop (overridelimit: Int) {
  override: TunableOverride[CV];
  connect(overridelimit, override.limit);
};
```

This way, the new `loop` block gets an extra parameter through which the user can connect some other value to set the `override`'s `limit`.

Alternative Wiring

If a subtype for a feature introduces new parameters, it is possible to automatically expose them and wire them by introducing a wiring for the subtype, and adding a recommendation that uses the new wiring. The new recommendation should have the same feature name as the original feature, and because of the subtype relation, the two recommendations are identified as alternatives.

For `TuneableOverride` the following wiring could be added:

```
wiring TuneableOverride[=>CV: Int, limit: Int] {
  ...
}
```

We can then define a new recommendation that will always expose the parameter:

```
recommendation Loop {
  override: TuneableOverride[CV, limit: Int];
}
```

By using the same feature name as before (`override`), and because `TuneableOverride` is a subtype of `Override`, the two recommendations will be identified as alternatives for the `override` feature. Note that it is also possible to add a recommendation for `TuneableOverride` with a different feature name. It will then appear in the wizard as another independent feature.

Default Alternative

It is useful to be able to have a default alternative for a feature. Normally, the common supertype of all alternatives is automatically chosen as the default, which was done in Figure 8. However, abstract types are not shown as alternatives at all, so if the common supertype is abstract, it is useful to define the default explicitly instead. This is what has been done in Figure 2. There, `Override` is an abstract type, and the `FixedOverride` is specified as the default in the recommendation, as follows:

```
recommendation Loop {
  override: Override[CV] default FixedOverride;
}
```

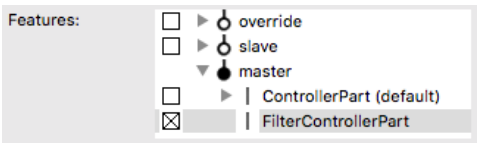


Figure 10. Wizard recommendations for `Loop`. The block `master` is `replaceable` and is specialized to an existing subtype of `ControllerPart`.

5.3 Specializing Mandatory Features

So far, we have described how alternatives to *optional* features can be added to a wizard using subtyping. We will now describe how alternatives to *mandatory* features can be added.

Whereas an *optional feature* is a feature declared in a recommendation, a *mandatory feature* is a block that is declared in a type. As discussed in Section 3.3, the type of an existing block can be specialized by using the `redeclare` construct. By declaring a mandatory feature as `replaceable` in a recommendation, subtypes of the declared feature type will be shown in the wizard, allowing a user to replace the existing feature with a more specialized type.

One variation of the `Loop` is that the `master` part can be replaced by a more specialized type. This is specified by adding a recommendation for it to be replaceable:

```
recommendation Loop {
  replaceable master;
}
```

Suppose `ControllerPart` has a subtype `FilterControllerPart`, which filters the process value. In creating a new specialization of `Loop`, the wizard shows the opportunity of replacing `master` with `FilterControllerPart`, as shown in Figure 10. This generates the following declaration of the block `loop`, as an anonymous subtype of `Loop`:

```
loop: Loop {
  redeclare master: FilterControllerPart;
}
```

The replaceable blocks correspond to mandatory features that have a default type that can be replaced. The wizard can thus support both mandatory and optional features.

The `replaceable` construct comes from Modelica (Modelica 2012), where it is used for defining which blocks are allowed to be redeclared in subtypes. In contrast, all blocks in Bloqqi can be redeclared, and the `replaceable` construct is only used in recommendations, in order to guide the wizard.

5.4 Inheriting Recommendations

Recommendations are not inherited by default. The reason is that the subtype might implement some of the optional features defined for its supertype, and perhaps in an alternative way than described in the recommendations. This could be done to support special cases not common enough to be part of the general recommendations in a library. However, another common case could be to create a subtype that adds behavior orthogonal to the recommendations of its supertype.

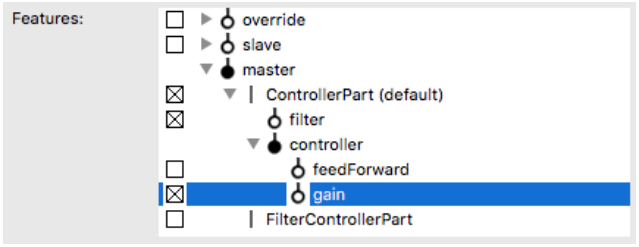


Figure 11. Wizard recommendations for `Loop`. The block `master` is specialized to a new anonymous subtype of `ControllerPart` that has support for proportional gain.

In this case it is possible to explicity let the subtype inherit all the recommendations for its supertype as follows:

```
recommendation T extends super;
```

This means that the recommendations for the diagram type `T` will include the recommendations from its supertype, transitively. That is, if the supertype of `T` also includes recommendations from its supertype, then these recommendations will also be included in the recommendations for `T`.

6. Combining Recommendations

We will now discuss more complex examples where recommendations are combined to give a hierarchy of features, what happens when different features are applied at the same place in a diagram, how source interception works when features are combined, and how hierarchical features can be visually inlined to expose the parts of most interest.

6.1 Hierarchical Recommendations

The type of a feature may itself have recommendations. In the previous example, we specialized a mandatory feature to an existing subtype. Another way to specialize a feature (whether mandatory or optional) is to select some of the recommended features for its declared type. If these features in turn have recommendations or subtypes, they can in turn be selected, and so on. This leads to a hierarchy of recommendations.

Consider again `Loop`'s mandatory block

```
master: ControllerPart;
```

The type `ControllerPart` contains the mandatory feature `controller: Controller`, and, due to a recommendation, an optional feature `filter`. (The `filter` filters the process value before sending it to the `controller` block.) The `controller` feature is declared as replaceable in a recommendation, so it can be specialized. In recommendations, two optional features of `Controller` are defined: `feedForward`, and `gain`. This gives the hierarchical wizard shown in Figure 11. In the wizard, the features `filter` and `gain` have been selected, resulting in the following generated code:

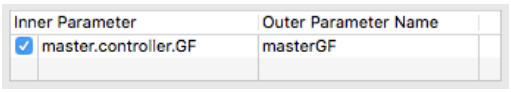


Figure 12. Exposure of inner unconnected parameter `GF`.

```
loop: Loop (masterGF: Int) {
  redeclare master: ControllerPart (controllerGF: Int) {
    filter: Filter[controller.PV];
    redeclare controller: Controller {
      gain: Gain[sub.out, GF: Int];
    };
    connect(controllerGF, controller.GF);
  };
  connect(masterGF, master.controllerGF);
};
```

Figure 13. Generated code when making the parameter `master.controller.GF` (Figure 12) accessible as an outer parameter (highlighted).

```
loop: Loop {
  redeclare master: ControllerPart {
    filter: Filter[controller.PV];
    redeclare controller: Controller {
      gain: Gain[sub.out, GF: Int];
    };
  };
};
```

Here, the type of `master` is redeclared to an anonymous subtype `ControllerPart { ... }`. The anonymous subtype includes the `filter` feature and redeclares the block `controller`. The `controller` block is redeclared to an anonymous subtype of `Controller` and includes the feature `gain`.

The example illustrates *hierarchical recommendations*, i.e., the user selects one feature, and then additional subfeatures of that feature. Note that the subfeatures are only shown if the main mandatory feature (`controller` in this case) is declared as `replaceable` in a recommendation. For optional features, subfeatures are always shown.

As described earlier, the wizard can infer which parameters that are not covered by the wirings, and allow the user to expose them as outer parameters. Exposing a parameter of a subfeature down the hierarchy will expose it all the way up to the currently constructed block, i.e., to the `loop` in this case. In this example, the `controller` feature has an input parameter `GF` not covered by the wiring, and the user has selected to expose it, see Figure 12. This results in the generated code shown in Figure 13, where additional parameters and connections have been generated, connecting `GF` from the inner `controller` feature, all the way to a parameter on `loop`.

6.2 Ordering Recommendations

If several recommendations are applied that intercept the *same* port, the order of application is significant, and may affect the meaning. This is an example of *feature interaction*.

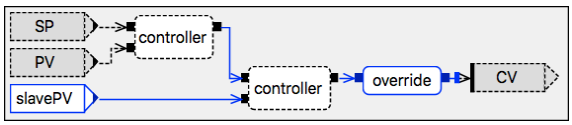


Figure 14. Specialization of `Loop`. The slave and override features intercept the same parameter (`CV`), however, the slave is applied before override, resulting in its inlined `controller` appearing before the `override` block.

Recall the definition of `Loop` that contains a `master` block (Figure 5). Additional common features of control loops include a slave part and an override filter.

Applying both these features in a new specialization will intercept the same parameter `CV`, and the order in which the features are applied is thus significant. In practice, we always want to apply the `slave` before the `override`, as shown in Figure 14, since it is the actual output to the process that we want to put a threshold on, not the set point of the slave controller.

To achieve the desired ordering, a recommendation can explicitly order features using `before` statements:

```
recommendation Loop {
  slave before override;
}
```

A statement `f1 before f2` specifies that the feature `f1` will be applied before `f2`, if both are added in a new specialization.

The compiler analyzes all recommendations to find out in what order to apply them. If two recommendations intercept the same port and are not ordered using `before`, then the compiler will report an error, to indicate the feature interaction. To resolve the problem, a new recommendation can be added that uses `before` to specify an explicit order.

Thus, all recommendations that intercept the same port need to be totally ordered using `before` statements. The ordering obtained by `before` statements is transitive. Hence, if `a` is declared to be before `b` and `b` is declared to be before `c`, then `a` is implicitly before `c` as well. Cycles are not allowed, and should they occur, the compiler reports an error.

By automatically identifying conflicts, two recommendations can be developed in parallel, independently from each other, and when they are combined, conflicts are reported and can then be solved explicitly and modularly.

While the ordering needs to be complete for all interceptions on the same port, the `before` statements need only give a partial order of all the features that can be applied for a given type: the order of application is irrelevant if they do not intercept the same port. Nevertheless, a total order of application is desirable, in order to generate normalized code when selecting features in the wizard. To achieve this, a total application order is computed by using alphabetic order for features not ordered by `before` statements.

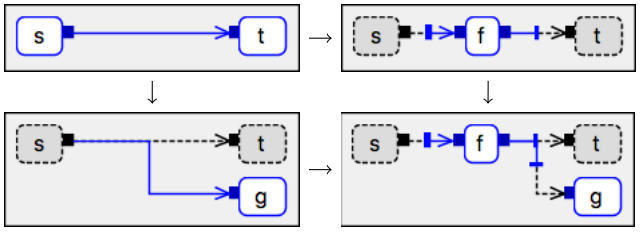


Figure 15. Base diagram extended with two features, `f` and `g`, where `f` uses *source interception* on the output port on block `s`. When the features `f` and `g` are combined, feature `g` will use the value from `f`, and not the value from block `s`.

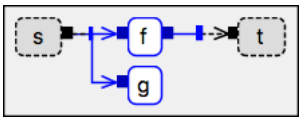


Figure 16. Same as Figure 15, but where feature `f` uses target interception instead of source interception, resulting in that feature `g` will use the value from the block `s`.

6.3 Source Interception

As mentioned earlier, an interception can be done either at the source or at the target of a connection. Distinguishing between source and target interception has advantages when combining features.

First, we can note that if two recommendations intercept the same connection, but one uses source interception and the other uses target interception, they are implicitly ordered, and no `before` statement is needed.

Source interception has an additional advantage in that it applies to *all* outgoing connections from that port. In contrast, a target interception can only apply to one connection since an input port can have at most one incoming connection. The difference between source and target interception is illustrated in the following example.

Consider two recommended features, one feature f that source intercepts port p on a block s and another feature g that uses the value from the same port p (by adding a connection). When both these features are selected, the second feature g will automatically use the value from f, and *not* the original value from the block s. This is because the first feature f uses source interception, which is applied for all outgoing connections from the port p, including the connection for feature g. This is illustrated in Figure 15.

If `f` had instead used target interception, g would have used the original value from s, as shown in Figure 16.

Target interception thus places the interception right before the value enters a target port, whereas source interception places it right after the value leaves a source port, i.e., before being distributed through connections to other blocks. For the control loop library in Figure 3, the `gain` feature uses source interception, so that all other components will use the

```
SteamHeating
   operatorSP
   temperature
   operatorGF
   pressure
   loop              // inlined
      master         // inlined
         filter
         controller
            gain
      slave          // inlined
         filter
         controller
      override
   valveOpening
```

Figure 17. Block composition hierarchy for the `SteamHeating` in Figure 2, indicating which blocks are visually inlined.

amplified input value instead of the raw input value. However, all other interceptions in that library are target interceptions.

In the editor, the user can interactively add source and target interceptions to a diagram, and the editor then generates a corresponding intercept statement. The textual syntax for source interception is similar to target interception, but with the extra keyword `source`. This is illustrated in the following code that source intercepts port `out` on block `s`.

```
intercept source s.out with b.in, b.out;
```

6.4 Visual Instance Inlining

The default visual view of a diagram shows the blocks declared at the top level of the diagram, but not their contents. However, it is often the case that a domain engineer would like to see a more flat view showing some but not all blocks in the complete hierarchy. This way, the most interesting blocks can be shown, giving a view similar to that which would be drawn manually, or to those appearing in textbooks on automatic control. To support this, it is possible in Bloqqi to *visually inline* a block (Fors and Hedin 2016). For a block that is visually inlined, the contents of the block are displayed in the diagram, instead of displaying the block itself.

As an example, consider the `SteamHeating` diagram from Figure 2 and its corresponding block composition hierarchy in Figure 17. In this case, the `loop`, `master`, and `slave` blocks are inlined, but not the `controller` blocks, so the `gain` block inside the master `controller` is not visible in the diagram. This way, the diagram shows the structure that is most important, and it is similar to a typical cascade control diagram from a textbook.

A block is inlined by adding a modifier `inline` to its declaration:

```
inline block: BlockType;
```

It is also possible for the user of the editor to manually select a block and inline it, which will add the `inline` modifier to the declaration of the selected block. When designing a library of block types and recommendations, it can be anticipated what blocks should typically be inlined. For the example in Figure 2, the `loop` block was interactively created, and also interactively inlined. However, the `master` and `slave` blocks are inlined because the library declared them as inlined in the `Loop` type and in the `Loop` recommendation:

```
diagramtype Loop(...) {
  inline master: ControllerPart;
}
...
recommendation Loop {
  inline slave: ControllerPart[CV];
}
```

When the user edits a diagram with inlined blocks in it, the addition of what looks like a single connection in the diagram may actually correspond to adding several connections and sometimes extra parameters to the inlined blocks in the underlying textual version of the program. These additions are done automatically by the editor.

In Figure 2, the master (the leftmost `controller`) is augmented with one extra parameter `GF` since it includes the feature `gain`. In the editor, the user can connect the operator value `operatorGF` to the parameter `GF`, even if the block is inlined in two steps. When this happens the editor will detect if the parameter `GF` is already directly accessible as an outer parameter on the new `Loop` specialization. If this is the case, a connection is added to the outer parameter. If the parameter `GF` is not directly accessible, then the editor will automatically create parameters and connections in the anonymous subtypes in order to connect `operatorGF` to the parameter `GF`. This is similar to when inner parameters are exposed as outer parameters during specialization creation (see Figure 13).

7. Editing Support

The editor has additional editing support to make it easier to work with recommendations. In particular, the editor supports creating recommendations by example as well as staged configuration (Czarnecki et al. 2004).

7.1 Creating Recommendations By Example

So far, we have described the textual representation for how to specify recommendations. In practice, we have found that it is often simpler to create the recommendations by example than to write them down as text. To support this, the user creates a temporary subtype, and visually adds desired blocks and connections. The subtype can then be extracted to a recommendation using an editor operation.

For example, suppose we want to add a recommended feature `f` for a type `T`. We can do this by creating a temporary subtype of `T`, and adding blocks to it, and connecting them to existing elements inherited from `T`. To extract the subtype to a new recommended feature `f`, the editor generates a new type `F`, a wiring declaration for `F`, and a recommendation for `T`. The new type `F` will contain all the local blocks and local connections between them. The wiring for `F` will capture

the remaining connections as parameters and connections between the parameters to the local blocks of `F`. Finally, the recommendation will declare the feature `f`, and apply the wiring to encode how the feature is to be connected to the relevant elements of `T`:

```
recommendation T {
  f: F[...];
}
```

Once this is done, the temporary subtype is no longer needed. Note that the generated wiring `F[...]` can be used for defining features also for other types than `T`, and even for other features of `T` by applying it at a different place in `T`.

7.2 Staged Configuration

As we have seen, the derived wizard can be used to create specializations of a type with the desired features. The editor also supports wizard state inferencing, so that the wizards can be used for editing existing blocks, and not just generating them. This is very important for practical use, and allows staged configuration (Czarnecki et al. 2004), where users specialize blocks in several steps.

For example, the control loop in Figure 2 has filters on both the input to the master and slave controller. After the creation of this block, we may later realize that these filters are unnecessary and should be removed. The user can then select the control loop in the editor and open the wizard again, where all previously selected features are inferred and automatically shown as selected. The user can then deselect the filter features and apply the changes.

When the user wants to change the specialization of a block, the editor will match the content of the block with recommendations to infer which features that are selected. As described earlier, when a new block b is created and specialized of type T using the wizard, the editor will automatically create an anonymous subtype of T. This anonymous subtype is the block type for b and contains the selected features. When the user requests changing the specialization for the block b, the editor will compare the content of the anonymous subtype with the recommendations for T, in a hierarchical manner. If the anonymous subtype contains a block with a name and a block type that matches a recommendation for T, then this block is considered as a feature that has previously been selected, and will be shown as selected in the wizard.

As described earlier, when a new specialization is created, inner parameters can be exposed as outer parameters. This is also inferred by the editor when the wizard for editing specializations is shown. Thus, previously exposed parameters will show up as already exposed in the wizard as well. This is implemented by analyzing the data-flow between the inner parameters and the outer parameters.

8. Evaluation

As a proof of concept, we have implemented the new language constructs and the wizard support, extending our Bloqqi tools which include a compiler and a visual editor. All the screenshots in this paper are from our tools, and all the examples given are runnable code. To evaluate the language constructs, we provide a comparison between using the feature wizard as compared to manually constructing the corresponding block diagrams.

8.1 Implementation

The Bloqqi compiler and editor share a core implementation of the language, developed using reference attribute grammars (RAGs) (Hedin 2000), using the metacompilation system JastAdd (Ekman and Hedin 2007).

The core implementation includes a parser, and RAG computations for static semantics like name resolution, type checking, and direction checking of the connected ports and parameters. To support the new language constructs, additional semantic checking was added, for example, to check that wiring applications are done with ports of the correct type and direction, that there is sufficient ordering of recommended features, as was discussed in Section 5, etc.

The compiler extends the core with RAG modules for C-code generation. The visual editor extends the core with RAG modules that compute the visual rendering of diagrams, combining the type and supertype definitions of each block. There are many computed properties in the visual presentation: the number of input and output ports on a specific block, the rendering of inherited and local information in a diagram, etc. The actual rendering is done using GEF (Graphical Eclipse Framework). To support the wizards, RAG modules were added that use the static semantics to compute the wizard content, and with computations of how to generate and insert the Bloqqi code corresponding to selections in the wizard.

We have executed Bloqqi programs on both specific controller hardware and together with simulated models. The models simulated have been specified as Modelica models and exported as executable simulations using the Functional Mockup Interface standard (Blochwitz et al. 2012). Thus, in Figure 1, we replaced the *Real World* with a simulation of the real world. We did this by specifying the inputs and outputs for the control program to be the outputs and inputs from the simulation.

8.2 Comparison

The examples in this paper are downscaled examples of how feature-based block libraries can be constructed and used. To get an impression of the advantages, we provide a comparison between constructing a small control system using the wizard, as compared to constructing it manually, i.e., creating the blocks and connections one by one.

The diagram in Figure 2 includes 5 *environmental* blocks and connections, i.e., blocks that communicate with the environment (sensors, actuators, operator), and a number of blocks and connections representing the control loop. The control loop logic is created by selecting 5 features in the

Control loop library			
	Environmental blocks	**Select features**	**Editing operations**
$\mathbf{W}_{none}$	3	0	1
$\mathbf{M}_{none}$	3	0	1
$\mathbf{W}_{e}$	5	5	1
$\mathbf{M}_{e}$	5	0	14
$\mathbf{W}_{all}$	11	9	1
$\mathbf{M}_{all}$	11	0	31

Table 1. Comparison of effort to create a control loop diagram using the wizard in Figure 3 (**W**) and creating it manually (**M**). Compares three different variants of the diagram: $none$ for selecting no optional features, i.e., only using a plain master controller, e for the `SteamHeating` example in Figure 2, and all for selecting all features. Note that an editing operation involves much more effort than a feature selection operation.

Control loop library		
	#Types	**#Recommendations**
Wizard	9	9
Explicit types	223	-

Table 2. Number of types and recommendations for the control loop wizard in Figure 3 as compared to having one explicit type for each variant.

wizard[5]. This corresponds to 14 manual editing operations (adding blocks, parameters, subtypes, connections, and interceptions). Table 1 compares the selections with corresponding manual editing operations for three examples: *none* means no selections were made in the wizard, i.e., only the basic loop logic is used, *e* is the example in Figure 2 and Figure 3, and *all* means doing as many selections as possible in the wizard.

The table shows how the advantage of using the wizard grows with the complexity of the design. It is important to note that selecting features in the wizard is *much* easier than to do an editing operation. Selecting a feature is done with one click, and requires only expertise in control systems concepts, whereas each editing operation involves several interactive steps (e.g., several clicks), and furthermore requires much more cognitive effort in locating the block types to instantiate, and deciding on how to do the wiring. We therefore expect the effort for building the control logic to be much lower for the wizard solution. Future user studies could be done to confirm this.

When the user creates a block instance by using the wizard, the particular feature selection is used to automatically compute the (anonymous) type for the block instance declaration. An alternative to using feature wizards would be to create one explicit type for each variant in advance. For the wizard in Figure 3, this would lead to 223 types, see Table 2. The total number of types includes the component types `Controller`, `Filter`, etc., and subtracting them gives 216, which is the number of variants. Having one type for each variant is not practical, but the comparison illustrates how fast the number of variants grow, even for a example that is quite small.

[5] `FixedOverride` is selected as default and therefore counted together with `override`

Tank library			
	Environmental blocks	**Select features**	**Editing operations**
$\mathbf{W}_{none}$	4	0	1
$\mathbf{M}_{none}$	4	0	1
$\mathbf{W}_{e}$	8	4	1
$\mathbf{M}_{e}$	8	0	42
$\mathbf{W}_{all}$	10	6	1
$\mathbf{M}_{all}$	10	0	50

Table 3. Comparison of effort to create a tank diagram using the wizard (**W**) and creating it manually (**M**). $none$ corresponds to selecting no optional features, e to the example in Figure 18, and all to selecting all features.

Tank library		
	#Types	**#Recommendations**
Wizard	4	6
Explicit types	48	-

Table 4. Number of types and recommendations for the tank library wizard in Figure 18 as compared to having one explicit type for each variant.

As an additional example, we have implemented a library for controlling the liquid level in a tank with a pump and a valve, and with optional features for heating, agitating, and for adding extra pumps and valves, as well as selecting between different types of valves. The heating feature may be implemented as steam heating, similar to the one in Figure 2. The result, measured in the same way as for the control loop library, is shown in Table 3 and Table 4. Again, we see that the effort of creating a diagram is much lower when using the wizard than when constructing it manually. The wizard for the library and the diagram for an example variant is shown in Figure 18.

9. Related Work

9.1 Variability Support for Diagram Languages

There are several diagram languages that have some kind of support for variability.

Simulink is a popular block diagram language with some built-in support for handling variability (Weiland and Manhart 2013). It supports *conditional* blocks which are executed conditionally, and *model variant* blocks which conditionally select a block from a set of variants. In contrast to our approach, this approach does not handle wirings, and all variants have to be anticipated in advance.

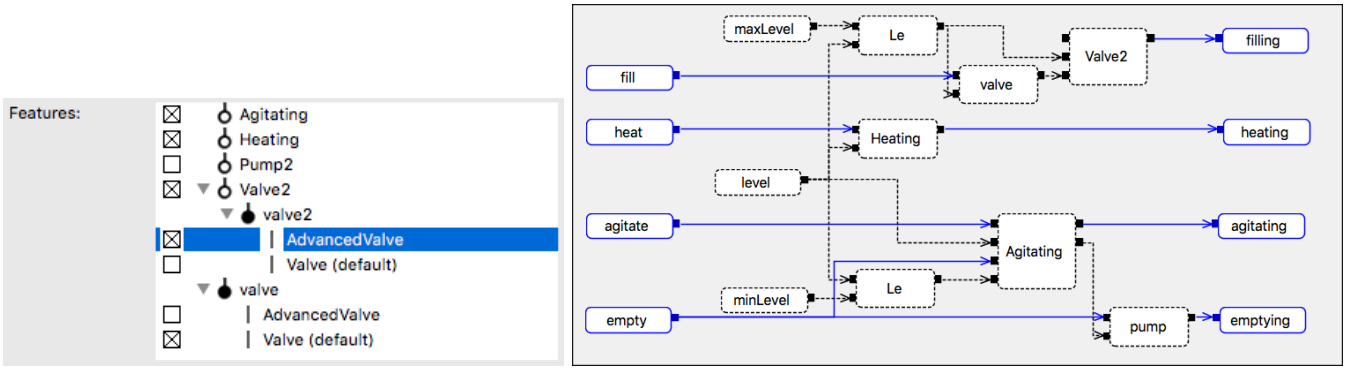


Figure 18. Tank specialized with agitation, heating, and a second valve and connected to environmental blocks (solid).

Dieumegard et al. (2014) present a textual domain specific language for defining blocks with variants in Simulink. An example is a sum block that sums the input for different structures, such as scalars and matrices. The variants of a block are specified with imperative code with additional constraints given in a subset of OCL. Again, all variants need to be anticipated in advance.

Haber et al. (2013) add delta modeling to Simulink to handle variability. In delta modeling, a set of deltas are specified relative to a base model. A delta may contain additions, removals, and modifications of blocks and connections. A variant is then created by selecting a set of deltas that are applied in a linear order on the base model. Deltas may have order constraints that describe how they can be applied, for example, that one delta is required to be applied before another delta, or that one delta requires another delta. Deltas are more flexible than recommendations in that they allow removals and arbitrary modifications. However, this also makes them less composable. In many other ways, deltas are much less flexible than our approach: they are specified relative to a base model, whereas our wirings can be applied at several locations, even within the same block. Deltas do not support explicit interception, but need to model such modifications using explicit connection removals and additions, and has no automatic detection of interfering intercepts as in our approach. The deltas are patch-based rather than type based, making them more low level to work with. There is no support for feature-selection wizards or modular additions to wizards, like in our approach.

Other examples of variability for diagrams include variant support for UML activity and class models (Czarnecki and Antkiewicz 2005) and for SysML (Trujillo et al. 2010), but these also require all variants to be anticipated in advance.

As mentioned, Bloqqi has been substantially influenced by Modelica (Modelica 2012), a language used for modeling and simulating physical systems. Important influences include the use of combined textual and visual syntax, and the use of object-oriented subtyping for blocks with connections. Bloqqi's `redeclare` and `replaceable` constructs were also heavily inspired by Modelica. However, the languages have different purposes, and there are otherwise more differences than similarities. In particular, Modelica is an equational language with undirected connections, solved symbolically or numerically during simulation. In contrast, Bloqqi's connections are directed, corresponding to data-flow directed execution. The wirings and recommendations are designed for data-flow based languages, where connections are directed. It would be interesting to investigate if and how they could be adapted to equation-based languages such as Modelica.

9.2 Inheritance

In earlier work (Fors and Hedin 2014), we introduced the interception construct, and then used multiple inheritance to combine features into a desired variant. While this is a possible way to modularize and reuse components, it has several drawbacks. First, it uses the inheritance construct for two different things: composing features and specializing blocks. This makes it difficult to make use of it in a wizard, as the wizard does not know which use of inheritance is intended for what. It also goes against the general object-oriented principle of favoring composition over inheritance, as described, for example, by Gamma et al. (1994). Second, by using a subtype to define a feature, it needs to wire the feature in a given inherited context. In contrast, the wiring construct introduced in this paper allows a feature to be applied in several different contexts, even within the same diagram.

Multiple inheritance using traits (Schärli et al. 2003) has a similar problem, in that they cannot be applied at several different places in the same type (diagram), which wirings can.

9.3 Aspects

Aspect-oriented programming (Kiczales et al. 2001) (AOP) allows crosscutting concerns to be defined in a modular way. Both recommendations and wirings have similarities with inter-type declarations in AspectJ, in that they allow declarations about existing types to be added modularly. However, in contrast to AOP, recommendations and wiring declarations do not change the behavior of existing types.

Wirings just declare how a block is typically instantiated, and recommendations just recommend new (anonymous) subtypes to be created with a desired behavior.

Another important difference is that AOP usually works on methods, like *advice*, which is code that is running after or before a method and is defined modularly in an aspect. Wirings and recommendations on the other hand allows functionality to be added inside diagrams, that is, at the statement level, rather at the method level.

9.4 Feature Models

Our derived wizards have similarities to feature models (Kang et al. 1990), a technique for describing variability. A feature model has the form of a tree, often depicted as a *feature diagram*, with features represented by tree nodes. The parent/child relation can be of different kinds:

optional: The child feature *may* be selected.

mandatory: The child feature *must* be selected.

alternative: There is a set of child features of which *exactly one* must be selected (exclusive-or).

Some feature systems also include an *inclusive-or* kind, where *one or more* child features must be selected. Many feature model systems additionally support various kinds of constraints. For example, requires/excludes cross-tree constraints, cardinalities on features (Czarnecki et al. 2005), or general propositional formulas (Batory 2005).

We have been inspired by the feature diagram syntax, and wizards for feature diagrams (Czarnecki and Antkiewicz 2005), to present the choices in our wizards. Figure 4 shows an example of drawing one of our wizards as a feature diagram. However, while it is possible to draw the wizard structure this way, there are several important differences from feature models. In particular, feature models model a set of global features, and each feature occurs only once in the tree. They are typically used in product lines, to describe at a global level which features a product includes. In contrast, we have one feature diagram per block type, and our features are local to a context, so a feature can occur in several places in the tree. For example, both the slave and the master feature have a local filter and controller feature. In fact, we can create recursive feature structures. For example, we could create a type `T` that has an optional feature of type `T`. The wizard expands dynamically, as the user opens features to see the sub-features, so the tool can handle this, even if the complete feature diagram would be infinite. Our wizard structure is thus like an ordinary context-free grammar, whereas feature models are typically restricted to tree grammars, with each nonterminal occurring only in the right-hand side of one production (Batory 2005). In contrast to most feature models, we do not use any constraints that restrict the choices more than the tree structure does. This could, however, be an interesting topic to investigate.

An important difference from ordinary feature models is that recommendations allow our feature models to be constructed modularly. For example, we saw in Section 5 how the subtype `FixedOverride` could be added, and will extend the wizard/feature model, without the need to touch the code defining the `override` feature.

9.5 Feature-Oriented Programming

Prehofer (1997) introduced the notion of Feature-oriented programming (FOP), and applied it to Java. FOP is related to feature modeling described above, but takes a more technical approach by adding first-class language support for features in a programming language. In this approach, the features are specified in a similar way as classes, with methods and fields, but can be combined together. To handle feature interaction, it is possible to define *lifters* between two features. A lifter is a piece of code that describes how the two features interact with each other and is defined outside the features. Apart from being designed for a programming language rather than a diagram language, Prehofer's FOP differs from Bloqqi in that it works at the method level rather than at the statement level, and it has no support for selecting features in a wizard.

Both FOP and feature modeling are part of the paradigm of Feature-oriented software development (FOSD) (Apel and Kästner 2009), which focuses on building large-scale software systems with the use of features, typically with the goal of constructing software product lines.

10. Conclusion

We have introduced the new language constructs of *wirings* and *recommendations* for block diagram languages to support the description of intended variability. We have shown how these constructs can be used to automatically generate smart editing support, in the form of wizards that suggest features for diagram types, aiding domain engineers in doing visual configuration of automation systems.

An important property of the new language constructs is that they support modular and extensible definition of variability: variability of diagram types can be defined separately from the libraries defining the diagram types. This is achieved partly by the recommendation construct that allows new recommendations to be added to existing diagram types, and partly by subtyping of diagram types, automatically extending all relevant recommendations.

Another important property of recommendations is that they are composable: recommended features that intercept the same connection are automatically identified, and can, if needed, be ordered explicitly by additional recommendations. This allows independently developed libraries or library extensions to be composed.

We have evaluated the new language constructs by implementing a proof-of-concept compiler and visual editor for them. We have also evaluated the constructs by comparing the effort of creating diagrams using the wizard, as compared

to creating them manually, concluding that using the wizard takes much less effort.

It turns out that recommendations have many similarities with feature models which also aim at supporting variability. An important difference is that whereas feature models are often aimed at global features of a product, recommendations are aimed at local features within a hierarchical configuration, and with modularity and composability as key concerns.

10.1 Future Directions

Although our focus has been on the application area of automatic control systems, the presented language constructs are general, and could be used for any language that has diagram types with blocks and directed connections. For example, it could be interesting to investigate how the constructs could be applied to languages based on Petri nets (Murata 1989) or state charts (Harel 1987). Within the automation domain there is, for example, the Grafchart language which focuses on supervisory control (Johnsson and Årzén 1998), and which combines features from both Petri nets and Statecharts. Other interesting application areas could be simulation and stream processing where data-flow programming is common. For block languages with undirected connections, like Modelica, the notions of source and target interception would be irrelevant, but the idea of modularly expressing features in the form of wired blocks might be interesting to investigate.

Other ideas from feature models could be investigated, possibly using them to enhance our experimental language. Examples include *require* and *exclude* statements, to express dependencies between different features.

It could be investigated how to use our proposed mechanisms for feature-oriented programming at a detailed algorithmic level. We have made experiments with programming a PID controller in Bloqqi, expressing the I and D parts as optional features of the algorithm. To apply a similar idea to general algorithmic code, generalizations would be needed, for example, to handle algorithmic loops.

The Bloqqi implementation is just a proof-of-concept prototype intended for experimenting with language constructs, and there are several ways to further extend and generalize both the language and its implementation. One direction is to support execution of multiple distributed diagrams, running with individual sampling speeds. Another direction is to extend the language with support for *control connections*: structured data where some elements can be *reversed*, allowing data to flow backwards along connections. This allows certain control diagrams to be substantially simplified, and improving control performance (Pernebo and Hansson 2002). Finally, larger case studies should be performed.

Acknowledgments

We would like to thank Ulf Hagberg, Christina Persson, Stefan Sällberg and Alfred Theorin at ABB for sharing their expertise about the ABB tools for building control systems. This work was partly financed by the Swedish Research Council under grant 621-2012-4727.

References

S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

K. J. Åström and T. Hägglund. *Advanced PID control*. ISA-The Instrumentation, Systems, and Automation Society; Research Triangle Park, NC 27709, 2006.

D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *9th International Modelica Conference*, 2012.

K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, LNCS, pages 422–437, 2005.

K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *Proc. 3rd Int. Conf. Software Product Lines*, pages 266–283, 2004.

K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

A. Dieumegard, A. Toom, and M. Pantel. A software product line approach for semantic specification of block libraries in dataflow languages. In *Proc. 18th Int. Software Product Line Conference*, pages 217–226. ACM, 2014.

T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Progr.*, 69:14–26, 2007.

N. Fors and G. Hedin. Intercepting dataflow connections in diagrams with inheritance. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 21–24, 2014.

N. Fors and G. Hedin. Visual instance inlining and specialization: Building domain-specific diagrams from reusable types. In *Proceedings of the 1st International Workshop on Real World Domain Specific Languages*, RWDSL '16, pages 4:1–4:10. ACM, 2016.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

A. Haber et al. First-class variability modeling in matlab/simulink. In *7th Int. Workshop on Variability Modelling of Software-intensive Systems*, pages 4:1–4:8. ACM, 2013.

D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

C. Johnsson and K.-E. Årzén. Grafchart for recipe-based batch control. *Computers & Chemical Engineering*, 22(12):1811 – 1828, 1998.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001*, pages 327–354, 2001.

Modelica. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3*. Modelica Association, 2012. Available from `modelica.org`.

T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

L. Pernebo and B. Hansson. Plug and play in control loop design. In *Proceedings for Control Meeting*, Linköping, Sweden, 2002.

C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP*, pages 248–274, 2003.

SysML. *SysML Open Source Specification Project*, 2015.

S. Trujillo et al. Coping with variability in model-based systems engineering: An experience in green energy. In *Modelling Foundations and Applications*, volume 6138 of *LNCS*, pages 293–304. Springer, 2010.

J. Weiland and P. Manhart. A classification of modeling variability in simulink. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 7:1–7:8. ACM, 2013.