Benefits of Feature-Oriented Block Diagram Programming

Niklas Fors and Görel Hedin

Department of Computer Science, Lund University, Sweden
(niklas.fors|gorel.hedin)@cs.lth.se

May 26, 2017

Abstract. Feature-oriented block diagram programming is a new way to program automation systems. It allows feature libraries to be created that support multiple variants with a high degree of code reuse. From the library specification, a feature wizard is automatically derived in which the user easily can get the desired variant by selecting appropriate features. This reports evaluates the benfits of using this new approach with respect to engineering- and testing costs on two benchmark examples.

1 Introduction

Automation programming of process industry plants is often done using *function block diagrams* in the style of the 61131-3 IEC standard [1]. A diagram consists of several function blocks whose input and output ports are wired to form a dataflow network. Examples of languages following this idea include Mathwork's Simulink [2] and ABB's ControlBuilder [3]. The programs for a plant typically consist of hundreds of such diagrams, each with tens or hundreds of wired blocks.

By constructing libraries of diagram types, for example for motors, tanks, and other aggregate parts of a plant, it is possible to encode and reuse engineering knowledge and reuse it for several plants, or in several places in the same plant. However, there are often many possible variants of such aggregate parts. For example, a motor may have one or two speeds, and it may optionally include detection of a minimal fluid level. A tank may have optional heating and agitation, etc. While the different variants are similar to each other, it is difficult to capture the variability in a library of an ordinary diagram language. Until recently, there were only two main approaches to this problem. One was to manually create each specific variant by instantiating and wiring subcomponents, and the other one was to create a complex template library type that captures all variants, and use boolean selectors to turn on or off parts to get the desired variant. Both these approaches have severe drawbacks. The manual approach requires a large engineering effort. The template approach results in complex diagrams that are difficult to understand, and a non-trivial engineering effort to set the selectors in the right way.

A recent approach, feature-oriented block diagram programming [4], introduces a number of new language constructs to solve this problem, allowing variants to be expressed as combinations of *features*, and allowing features to be

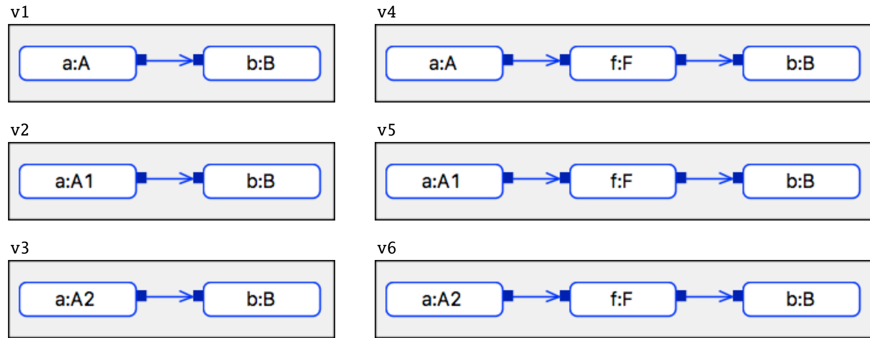


Fig. 1. Six variants (`v1`-`v6`) of a simple block diagram.

expressed in libraries. This allows a specific variant to be very easily created by selecting a number of optional features for a base diagram. The approach has been implemented in an open-source prototype language and tool, Bloqqi [5].

In this report, we analyze the benefits of the new approach with regards to engineering effort and testing. We begin by reviewing feature-oriented block diagram programming, contrasting it to earlier approaches for handling variants (section 2). We will then introduce two benchmark examples: a simple one with tank control, and a more complex one with PID and cascade control (section 3). In section 4, we introduce metrics for evaluating engineering costs, and compare the methods on the benchmark examples. In section 5, we discuss how features can be used for test coverage of block diagram libraries. Finally, Section 6 concludes the report.

2 Block diagram variants

Figure 1 shows a simple block diagram in 6 different variants. The base variant `v1` contains a local block `a:A` (block name `a` of block type `A`) connected to another block `b:B`. In `v2`, the block type of `a` is replaced with a similar block type `A1`, and in `v3` by another similar block type `A2`. In `v4`, an extra block `f:F` has been added between `a:A` and `b:B`, and `v5` and `v6` show the combinations of `v2` and `v4`, and `v3` and `v4` respectively.

We can view each variant as a combination of *features*, and depict all possible variants in a *feature diagram* [6, 7], see Figure 2. A feature diagram depicts mandatory features (`a` in this case), alternative features (`A1` and `A2` are alternative block types to `A`), and optional features (like `f`). Mandatory features are only depicted if they have selectable subfeatures. This is why `f` is depicted but not `b`, although `b` is also a mandatory feature. The total number of variants is all possible combinations of selected or not selected features ($3 * 2 = 6$ in this case).

In a normal block diagram language, we would need to manually construct the desired variant from scratch using an editor. If a specific variant is often used,

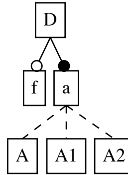


Fig. 2. Feature diagram representing the variants in Figure 1. Optional features are depicted with white circles and mandatory features with black circles. Alternative features are depicted with dashed lines.

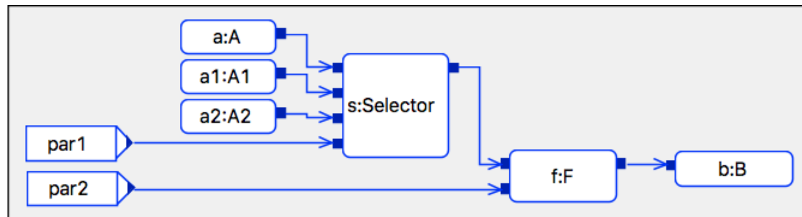


Fig. 3. Using the template approach to support all variants in Figure 1 in one diagram. The parameters `par1` and `par2` are used to decide what features to use.

we could create a library type for it, and instantiate it, instead of redrawing the diagram for each use. However, as the number of alternative and optional features increases, the number of possible variants increases combinatorially, so storing them all in a library is not a viable solution.

A possible solution for supporting all variants by a library, is to use a *template* approach. In this approach, a library type is created that contains *all* features, as well as extra parameters and logic that is used to select what part of the diagram is actually used. Figure 3 shows an example. This library type can be instantiated, and by passing `2` to `par1`, and `true` to `par2`, this would correspond to variant `v6` in Figure 1.

The template approach has several drawbacks:

- the diagram is more complex than the individual variants
- we cannot immediately see which variant is used
- the diagram is inefficient, because it needs to contain all the logic, even if only some of it is actually used
- the diagram is not extensible: if we come to think of a new optional feature we would like to add, we cannot do so without changing the library type

The language Bloqqi [4], is a function block diagram language that has specific constructs to explicitly support features and variants. Figure 4 shows the Bloqqi code for the `A`, `B`, `F` example. Here, the diagram type (block type) `D` contains an `a:A` and a `b:B` block. Subtyping is used to obtain alternatives: `A1` and `A2` are subtypes to `A`. The feature `f:F` is added through an *intercept* mechanism [8] that allows a connection to be specialized by letting it go via two other ports. In

```
diagramtype D {
  a: A;
  b: B;
  connect(a.out, b.in);
}
diagramtype A( => out: Int) { ... }
diagramtype A1 extends A { ... }
diagramtype A2 extends A { ... }
diagramtype B(in: Int) { ... }
diagramtype F(in: Int => out: Int) { ... }
wiring F[=>v: Int] {
  intercept v with F.in, F.out;
}
recommendation D {
  replaceable a;
  f: F[b.in];
}
```

Fig. 4. Bloqqi code defining the variants in Figure 1 as features using recommendations.

this case, the connection flowing into `b.in` (port `in` on block `b`) is specialized by going via the `in` and `out` ports of the `f` block. The *recommendation* construct says that `f` is an optional feature for `D`, and if it is selected, it is added to the `D` diagram by intercepting the A→B connection.

The user constructs a desired variant of the base diagram `D` by selecting in an automatically generated wizard, see Figure 5. The wizard is analogous to the feature diagram, showing alternative and optional features as check boxes. By selecting `A2` and `F` we get a diagram corresponding to `v6`. Figure 6 shows both the visual and the textual representation of the diagram variant. Visually, the base diagram is depicted using dashed lines (indicating the mandatory parts that cannot be changed). Textually, the selected features are represented as an anonymous subtype of `D`. The user can bring up the wizard again and change the selection, if desired.

The feature-oriented approach combines the advantages of the manual and template approaches, and avoids their drawbacks. In particular:

- the visualization of each diagram variant is as simple as if it had been created manually.
- it is immediately clear which variant is used.
- it is much quicker to create the diagram variant (simply click a few check boxes) than if we would have had to create the diagram manually.
- it is much easier to create the diagram variant through the wizard, since we do not have to know exactly how to wire the individual components. For example, the filter `F` always goes between `A` and `B`.
- the diagram is efficient, since it contains only the logic actually used.

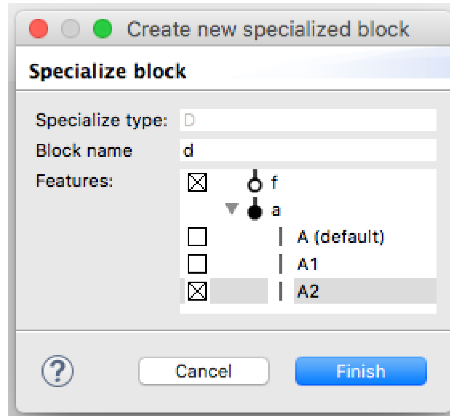
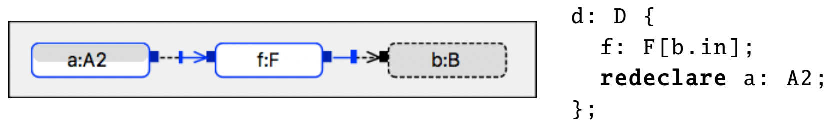


Fig. 5. Automatically derived feature wizard from the Bloqqi code specified in Figure 4. The selected features correspond to the variant `v6`.




```
d: D {
  f: F[b.in];
  redeclare a: A2;
};
```

Fig. 6. Automatically generated Bloqqi program when selecting the features in the wizard (Figure 5) corresponding to the variant `v6`.

– all variants are encoded in library types. We can easily change to another variant by bringing up the wizard again.
– we can easily extend the library without changing it. For example, we can add a new subtype A3. It will automatically appear in the wizard.

In the following, we will present a number of realistic libraries using the feature-based approach. We will evaluate the engineering cost as compared to the manual and the template approach, and we will discuss how the features can be used in testing.

3 Benchmark examples

To evaluate the feature-oriented approach, we have developed two library types. One smaller, `Tank` for tank control, with optional valves and agitation, and one larger, `Loop` for control loops with P/PI/PD/PID control, cascade control, and filters. Figures 7 and 8 show the feature diagrams for these library types. The `Tank` and `Loop` types capture 48 and 1800 distinct variants, respectively.

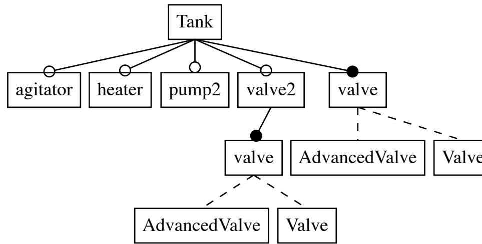


Fig. 7. Feature diagram for the Tank library.

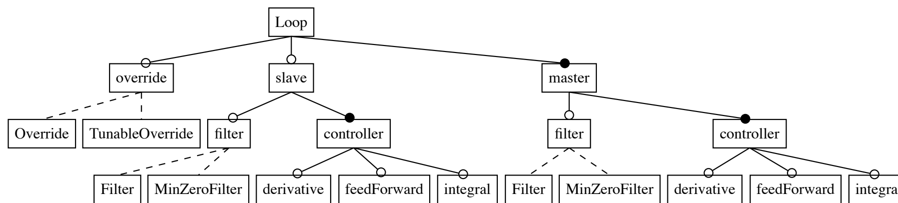


Fig. 8. Feature diagram for the Control loop library.

4 Engineering costs

To estimate engineering costs, we introduce a metric based on the number of edit operations needed to construct a diagram. For the manual approach, this includes creating each block, parameter and connection in the diagram, and we count the sum of these. For the feature selection approach, we count each box checked in the wizard. The results are shown in Table 1. Even with few features selected, the cost is much lower for the feature-oriented approach than the manual approach, and as more features are used, the win is even greater.

The comparison is crude since it does not take into account the knowledge needed to perform the operations. For the manual approach, knowledge is needed to select the appropriate block to instantiate, and to select between which ports to draw connections. For the feature-based approach, much less knowledge is needed, since it is only necessary to understand the different features, and not exactly how they are implemented using blocks and connections. Therefore, the actual difference between using the manual approach and the feature-based approach is greater than the metric shows.

5 Testing

A process control system needs to be tested thoroughly before taken into operation. By using reusable libraries in an application, as opposed to constructing the whole system from scratch, the test burden can be reduced significantly. However, it is then important that the library itself is well tested. Complete testing

	Feature	Manual			
Example	**Selections**	**Edits**	Parameters	Blocks	Connections
Tank_{none}	0	21	4	7	10
$\text{Tank}_{example}$	2	44	7	14	23
Tank_{all}	6	69	10	23	36
Loop_{none}	0	11	4	2	5
$\text{Loop}_{example}$	3	40	8	11	21
Loop_{all}	13	85	15	25	45

Table 1. Comparison between using the feature wizard and manually constructing a variant. Three variants are compared for each library: *none*, *example* and *all*. The variant *none* corresponds to the base diagram with no features selected, *example* with some of the features selected and *all* with all features selected. The features selected for $\text{Tank}_{example}$ are {`agitator`, `pump2`} and the features selected for $\text{Loop}_{example}$ are {`derivative`, `feedForward`, `integral`} for the `master` controller.

is typically not feasible since the number of possible uses is much too large. Instead, some level of test coverage is typically aimed for. It is also desirable that the tests are related to each other in such a way that it is easy to pinpoint an error when tests fail.

To measure how complete testing is, it is useful to make use of test coverage metrics.

We propose to use the feature-oriented constructs of Bloqqi to formulate coverage criteria for libraries. We consider the construction of test models to test the library, where each test model is a type variant (characterized by a set of selected features).

Feature coverage A suite of test models has full *feature coverage* when each alternative or optional feature is present in at least one model, and absent in at least one model.

Order coverage A *connection interaction* occurs when two features `f1` and `f2` include an interception on the same connection. In Bloqqi, each such pair needs to be ordered, so that it is clear in what order the features intercept the connection. This can be done using `before` statements. If more than two features intercept the same connection, it is sufficient to declare enough `before` statements so that all features are ordered. For example, if we declare `f1 before f2`, and `f2 before f3`, then implicitly, `f1` is before `f3`. A suite of test models has full *order coverage* if, for each `before` statement (`f before g`), there is a model including both `f` and `g`.

Feature coverage is a very basic coverage criterion in that it does not take feature interaction into account at all. Order coverage is also very basic in that it considers only the most explicit interactions between two features. On top of these criteria, more elaborate criteria can be formulated to take more implicit interactions into account. For example, we could consider all pairs of features, or triplets of features, etc. Or we could consider all combinations of features intercepting the same connection, etc.

		Number of test cases	
Library	**Variants**	MaxAlt	EachFeature
Tank	48	3	7
Loop	1800	3	14

Table 2. The number of test cases for the test generation methods MaxAlt and EachFeature.

The coverage metrics can be fulfilled in many different ways. For example for feature coverage, it would be sufficient with one test model that includes no features, and an additional n models where n is the largest number of alternatives for a single feature of the tested type. Another way of getting feature coverage would be to have one test model that includes no features, and an additional model for each alternative and optional feature, including other features only as necessary. The former suite (called MaxAlt) would be good for testing feature interactions, but the latter suite (called EachFeature) would be good for pinpointing errors that have to do with a single feature. The number of test cases in these suites for the libraries Tank and Loop are shown in Table 2. As we can see in the table, the number of variants is much higher than the number of test cases in the test suites.

There are thus many opportunities to explore the use of features to generate test models for libraries.

6 Concluding discussion

We have in this report evaluated the benefits of using feature-oriented block diagram programming on two benchmark library examples. We have created variants using the derived feature wizard and compared it to creating the variants manually. The number of editing operations is on average a magnitude higher in the manual case compared to the number of selections in the feature wizard. This indicates that the feature wizard is much easier to use and that it reduces engineering costs. We believe that these numbers underestimate the savings since selecting features in a feature wizard is much easier than creating blocks and connections manually, since the latter requires much more detailed domain knowledge.

We have also described how features can be used for testing and introduced two test coverage metrics: feature coverage and order coverage. We have then described two ways to generate test suites from feature models that fulfill the metric feature coverage, where the number of test cases is proportional to the number of features. Generating test suites like this would not be possible using the template approach described earlier, since it is impossible to know which of the parameters that are used for feature selection.

Acknowledgments

This work has been done within the project *Feature-Oriented Automation Programming*, 2016-03414, a project supported by the strategic innovation program *Process Industrial IT and Automation* (PiiA), financed by VINNOVA, Sweden's innovation agency.

References

1. Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids.* Springer Science & Business Media, 2010.
2. MathWorks. *Simulink documentation*, r2017a edition.
3. ControlBuilder. *Compact Product Suite. Compact Control Builder AC 800M. Product Guide. Version 6.0.* ABB, 2016. Available from abb.com. Document number: 3BSE041586-600 A.
4. Niklas Fors and Görel Hedin. Bloqqi: Modular feature-based automation programming. In *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*, pages 57–73, 2016.
5. Niklas Fors. *The Design and Implementation of Bloqqi - A Feature-Based Diagram Programming Language.* PhD thesis, Lund University, October 2016.
6. Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.
7. Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.
8. Niklas Fors and Görel Hedin. Intercepting dataflow connections in diagrams with inheritance. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 21–24, 2014.