The Design and Implementation of Bloqqi - A Feature-Based Diagram Programming Language

Fors, Niklas

2016

The Design and Implementation of Bloqqi – A Feature-Based Diagram Programming Language

Niklas Fors

Doctoral Dissertation, 2016

Department of Computer Science
Lund University

ISBN 978-91-7623-998-8 (printed version)
ISBN 978-91-7623-999-5 (electronic version)
ISSN 1404-1219
Dissertation 53, 2016
LU-CS-DISS:2016-5

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: niklas.fors@gmail.com

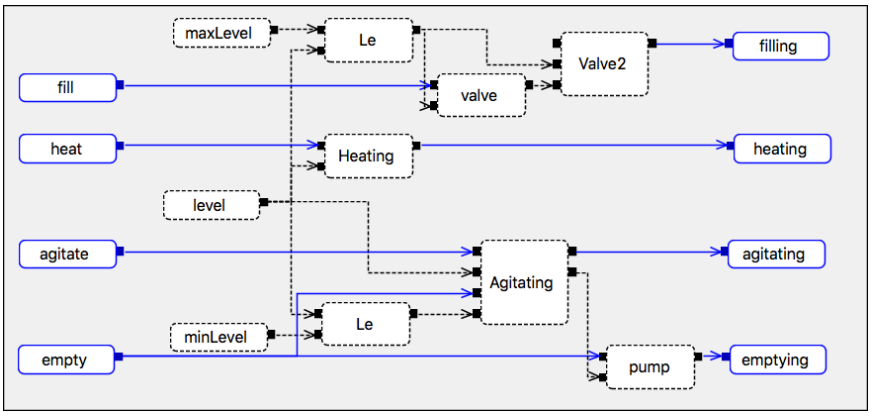
Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2016

Abstract

This dissertation presents the design and implementation of a new block diagram programming language, *Bloqqi*, for building control systems with focus on variability. The language has been developed in collaboration with industry with the goal of reducing engineering time and improving reuse of functionality.

When building a control system for a plant, there are typically different variants of the same base functionality. A plant may have several variants of a tank, for example, one variant with heating and another one without. This dissertation presents novel language mechanisms for describing this kind of variability, based on diagram inheritance. For instance, Bloqqi supports specifying what features, like heating, the base functionality can have. These specifications are then used to automatically derive smart-editing support in the form of a *feature-based wizard*. In this wizard, the user can select what features the base functionality should have, and code is generated corresponding to these features. The new language mechanisms allow feature-based libraries to be created and extended in a modular way.

This dissertation presents techniques for implementing rich graphical editors with smart editing support based on semantic analysis. A prototype compiler and graphical editor have been implemented for the language, using the semantic formalism *reference attribute grammars* (RAGs). RAGs allow tools to share the semantic specifications, which makes it possible to modularly extend the compiler with support for advanced semantic feedback to the user of the graphical editor.



Figur 1: Styrprogram för en tank i Bloqqi.

Bloqqi: nytt diagramspråk för programmering av styrsystem

Den här avhandlingen presenterar ett nytt programmeringsspråk, *Bloqqi*, som gör det lättare att skapa och återanvända styrprogram inom automation, såsom för fabriker. Målet är att minska ingenjörstiden det tar att bygga ett styrsystem. Särskilt fokus för språket är *variabilitet*, det vill säga, hur man på ett enkelt sätt stödjer olika varianter av samma grundfunktionalitet, exempelvis om ett styrprogram för en tank har omrörning eller ej. Språket har utvecklats i samarbete med industrin och företaget ABB som verkar inom automation.

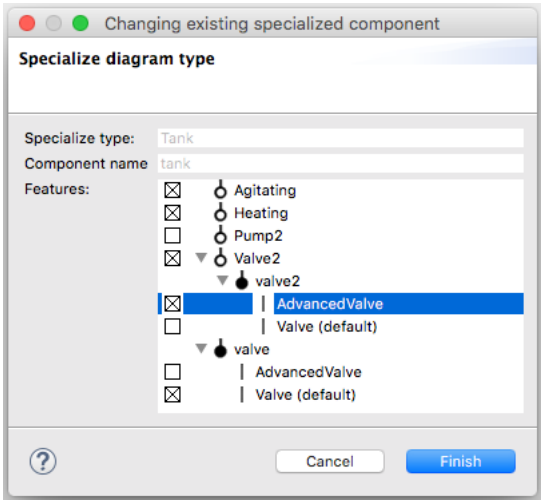
Introduktion till Bloqqi

ABB har idag ett grafiskt programmeringsspråk för att bygga styrsystem, där ett program är indelat i diagram, och varje diagram består av boxar och pilar. Boxarna beskriver beräkningar och pilarna beskriver hur beräkningarna beror på varandra. Bloqqi fungerar på samma sätt, men har byggt vidare och infört nya språkmekanismer som stödjer variabilitet.

Ett exempeldiagram i Bloqqi visas i Figur 1, som beskriver ett styrprogram för en tank. Tanken har stöd för fyllning, tömning, värmning och omrörning. Exempelvis beskriver boxen `Agitating` hur omrörning sker och boxen `agitate` beskriver om omrörning ska ske just nu, vilket bestäms av en operatör i fabriken.

Varianthantering

När man skapar ett styrsystem för en hel fabrik är det vanligt att det finns olika varianter av samma grundstruktur. Exempelvis kan det finnas flera olika tankar



Figur 2: Utvecklaren skapar styrprogram för en tank och kan välja vilken funktionalitet tanken ska ha.

som fungerar ungefär likadant, till exempel några med värmning och några utan värmning. Istället för att utvecklaren skapar ett diagram, liknande diagrammet i Figur 1, för varje tank, så erbjuder Bloqqi möjligheten att generera delar av ett sådant diagram automatiskt.

Bloqqi introducerar flera nya språkmekanismer för att underlätta varianthantering. Den som utvecklar ett styrsystem kan lätt välja vilken funktionalitet en grundstruktur ska ha i ett genererat grafiskt användargränssnitt. Detta visas i Figur 2, där styrprogrammet för en tank skapas. Här kan utvecklaren välja vilken funktionalitet tanken ska ha, till exempel omrörning och värmning. Väljer utvecklaren valen enligt figuren genereras det delar av ett diagram, motsvarande de streckade blocken i Figur 1. Även om de nya språkmekanismerna är utvecklade i kontexten av styrsystem, så kan de appliceras för andra liknande diagramspråk också.

Utvecklingsverktyg

Två utvecklingsverktyg har tagits fram för språket: en grafisk utvecklingsmiljö (figurerna är skärmdumpar) och en kompilator som genererar exekverbar kod. Avhandlingen presenterar hur man kan utveckla sådana verktyg med hjälp av moderna datavetenskapliga tekniker. Dessa tekniker möjliggör att de båda verktygen återanvänder samma språkspecifikationer för att effektivisera verktygsutvecklingen. Teknikerna möjliggör även avancerat editeringsstöd i den grafiska utvecklingsmiljön för den som utvecklar styrsystem.

Acknowledgements

First of all, I would like to thank my supervisor Görel Hedin, who has been indispensable during the journey of this dissertation. Görel has taught me how to think about research, introduced me to attribute grammars, and helped me with improving my writing and presentation skills. I would also like to thank my co-supervisor Boris Magnusson for his support.

The language mechanisms presented in this dissertation have been motivated by examples provided in a collaboration with the company ABB. I want to thank Ulf Hagberg, Christina Persson, Stefan Sällberg, and Alfred Theorin from ABB for a fun and interesting collaboration. Their expertise about control systems has been very useful during the work of this dissertation. We have collaborated for over four years with monthly meetings with intense discussions.

Furthermore, I would like to thank current and previous members of the JastAdd group: Jesper Öqvist, Emma Söderberg, and Christoff Bürger. Also, I had the pleasure to work with Gustav Cedersjö on JavaRAG, a Java library for the semantic formalism reference attribute grammars. I have been supervisor for several Master's theses that have been about the modeling language Modelica, which gave valuable knowledge about the language, together with the supervisors Jon Sten, Jesper Mattsson, and Jonathan Kämpe at the company Modelon. I would also like to thank Johan Åkesson for introducing me to Modelica and the Functional Mockup Interface.

Finally, I would like to thank my family, my friends, and my colleagues for their help and support.

This work was partly financed by the Swedish Research Council under the project "ITRAG: Interactive Tools based on Reference Attribute Grammars", by ABB under the project "Semantic Tools for Control Languages", and by the ELLIIT Excellence Center at Linköping and Lund.

CONTENTS

1 Introduction 1
2 Background 2
3 Introduction to Bloqqi 7
4 The Implementation of the Bloqqi Compiler and Editor 11
5 Contributions 15
6 Included Papers 16
7 Related Papers 18
8 Conclusion 20
References 21

I Intercepting Dataflow Connections in Diagrams with Inheritance 25
1 Introduction 25
2 Connection Interception 26
3 Implementation 30
4 Evaluation 31
5 Related Work 33
6 Conclusion 33
References 34

II Visual Instance Inlining and Specialization – Building Domain-Specific Diagrams from Reusable Types 35
1 Introduction 35
2 The Bloqqi Language 37
3 Visual Instance Inlining 42
4 Instance Specialization 46
5 Implementation 50
6 Related Work 53
7 Conclusions 55
8 Acknowledgements 56

References 56

III Bloqqi: Modular Feature-Based Block Diagram Programming 59
1 Introduction 59
2 Motivating Example 62
3 The Bloqqi Language 65
4 Wirings 68
5 Recommendations 70
6 Combining Recommendations 76
7 Editing Support 82
8 Evaluation 83
9 Related Work 86
10 Conclusion 91
References 92

IV Implementation of the Bloqqi compiler and editor 95
1 Introduction 95
2 Introduction to Bloqqi 97
3 Overall architecture 103
4 Reference Attribute Grammars 105
5 The Bloqqi Compiler 113
6 The Bloqqi Editor 117
7 Code Generation 121
8 Discussion 129
9 Related Work 134
10 Conclusions 135
References 136

1 Introduction

Domain-specific languages (DSLs) are becoming more and more important. A DSL targets a specific application domain and has a notation that is adapted for the domain [Hud97; Deu+00; Mer+05], usually at a higher abstraction level than ordinary programming languages. The adapted notation makes it easier to express programs in the domain. Sometimes, this even allows people without any programming background to write programs, making the DSL more accessible. In contrast, a general-purpose language (GPL) targets a wide variety of domains. Java, C, and C++ are all examples of GPLs.

There is a wide range of DSLs that target different domains, such as for graphics, hardware description, databases, mathematics and automation. Example languages include VHDL for hardware description, Matlab for numerical computations, SQL for database queries, and Modelica for modeling and simulation of physical systems [Mod12].

Several DSLs are based on the paradigm of *data-flow programming*. Programs in these languages are typically expressed visually with boxes and arrows. The boxes represent some kind of computation, like a simple addition or a complex computation like controller logic. The arrows describe how the computations depend on each other, that is, the data flow between the computations. An example of a data-flow programming language is ABB's ControlBuilder which is used in automation [Con16], and which is based on the language Function Block Diagram in the IEC-61131 standard. Other examples include Simulink from MathWorks and LabVIEW from National Instruments.

Often when developing programs, there are different *variants* of the same base functionality. For example, in the automation domain, we may have a tank regulator with different features, like with or without heating, and with or without agitation. Each combination of features constitutes a variant, and the number of possible variants increases exponentially with the number of features [AK09]. It is a challenge to specify the desired variants without introducing unnecessary code duplication and without having complex diagrams capturing all the anticipated features. In the automation domain, there two common ways to solve this challenge: *copy-modify* and *parameterization*. Copy-modify means that each new variant is copied from an existing variant and changed for the desired functionality, leading to code duplication. Parameterization means that a diagram covers all interesting features which are turned on and off using parameters, leading to runtime overhead and complicated diagrams, and requires all variants to be anticipated in advanced. This dissertation tries to solve these problems with new language features that allow modular feature-based specifications, avoiding the drawbacks of the other approaches.

This dissertation presents the design and implementation of a data-flow language called *Bloqqi*. The language includes novel language mechanisms for describing variability based on diagram inheritance. These new language mecha-

nisms make it possible to avoid code duplication and avoid the need for anticipating all variants in complex diagrams. The language has been developed in collaboration with the company ABB with the goal of improving reusability within the automation domain.

The compiler and the graphical editor for Bloqqi have been implemented using the metacompilation system JastAdd [EH07b] and the semantic formalism *reference attribute grammars* [Hed00] (RAGs). RAGs allow semantic analyses to be shared between tools, reducing code duplication, and making it possible to reuse the semantic analyses in the editor to provide semantic feedback to the user. As an example of semantic feedback, the editor colors arrows red that connect incompatible data types. One interesting property of the language and the implementation is that the visual syntax depends on user-defined types, and is *computed* using RAGs.

We will now describe the background for the new language, with focus on the languages Modelica and ControlBuilder, which both have inspired Bloqqi. We will then continue with an introduction to Bloqqi and an overview of the implementation for the compiler and editor. Then, the contributions of this dissertation are presented, followed by a description of the included and related papers to this dissertation. The iterative methodology used when developing the language mechanisms is then shortly described. Finally, we conclude with future directions.

2 Background

During the development of Bloqqi, we have been inspired by several languages, especially by the data-flow programming language ControlBuilder and the equation-based language Modelica. This section gives a brief overview of ControlBuilder and Modelica, and also describes other related languages.

The goal of ControlBuilder is to program automation systems, whereas the goal of Modelica is to model and simulate physical systems, to understand the dynamics of a system. Simulations can be used to test automation systems, where the automation system controls a simulation of the physical process. Bloqqi has the same goal as ControlBuilder: to program automation systems, but it borrows constructs from both ControlBuilder and Modelica.

2.1 ControlBuilder

ControlBuilder is an engineering tool from ABB used within the automation domain to program distributed control systems (DCSs) [Con16]. The tool provides programming languages according to the standard IEC 61131-3, and also extensions to the standard to support the needs in the DCS domain. One extension is *Diagrams* that extends the visual data-flow programming language Function Block Diagram (FBD) defined by the standard IEC 61131-3. This standard defines five programming languages, including FBD, for programmable logic controllers

(PLCs). Both PLCs and DCSs are used for automation. Traditionally, PLCs are used for manufacturing of items, while DCSs are used for continuous process control, for example, in chemical plants. DCSs typically have higher degree of redundancy, are running continuously, which requires online updating, etc., and are more expensive [NS07].

A DCS has many distributed controllers, and each controller can run *tasks*. Each task is executed periodically with a fixed frequency. In each period, there are three steps that are carried out. In the first step, sensor values, or values from other tasks, are read. In the second step, the control values are computed based on the sensor values and state values. In the last step, the computed control values are sent to actuators to control the process.

The computation of the control values is programmed using the diagrams in the ControlBuilder Diagram language. Like many other data-flow programming languages, ControlBuilder supports *user-defined block types*. A diagram in ControlBuilder can have blocks and connections between the blocks. The blocks are instances of *block types*. A block type can be defined as another diagram, or using structured text (similar to Pascal), or some other language. This makes it possible to create hierarchical programs and abstracting common functionality into block types and putting them into libraries for reuse.

Plants in the process industry typically run for a long period of time, potentially for several years. One reason for this is that the startup time for a plant may be quite long, like days or weeks. Also, many plants in the process industry have high requirements on the safety for the control system. For example, in some cases, it is required that several control systems are run in parallel for redundancy. Running a control system for a long period of time usually requires online updating of the software, which is supported in ControlBuilder. However, this dissertation focuses on diagram languages, and will not consider online updates or distributed control.

Tank Regulator Example

Figure 3 shows a simple example diagram in ControlBuilder. The diagram controls the amount of liquid in a tank. The tank has a valve that is used to fill the tank and a pump to empty it. A diagram can have input parameters, blocks, output parameters and connections between them that describe the data-flow. The data-flow in the diagram is from left to right. The boolean input parameters `fill` and `empty` are to open the valve and pump, respectively. However, the valve is only opened if there is room for more liquid and the pump is only opened if there is enough liquid to empty. The `Le` blocks (Less or Equal) compare the input values and return true if the first input value is less or equal to the second input value. This is used when comparing the current liquid level, measured by a sensor, with the maximum liquid level, which is set by a human operator. The diagram contains similar logic for the pump. The boolean output parameters `filling` and `emptying` describe the status and say if the valve or pump actually are open and turned on, respectively.

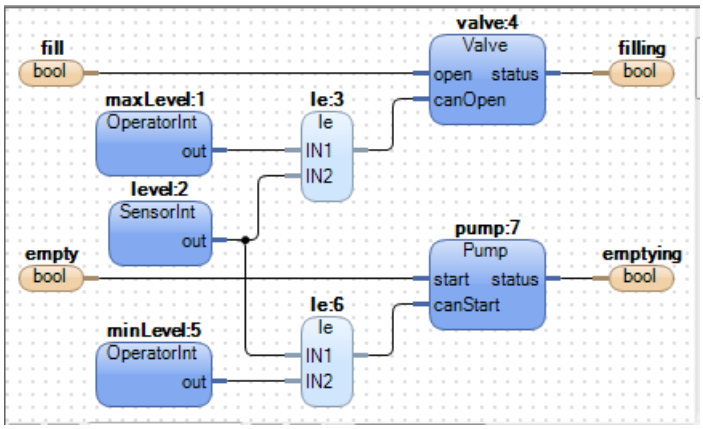


Figure 3: Simple tank regulator in ControlBuilder.

2.2 Modelica

Modelica is an equation-based and object-oriented language for modeling and simulating physical systems [Mod12]. The language is community developed under the organization Modelica Association. There are several tools supporting Modelica, for instance, the commercial tool Dymola from the company Dassault Systemes and the open source tools OpenModelica [Fri+05] and JModelica.org [Åke+10b].

Models in Modelica are described by algebraic differential equations that are solved by a numerical solver during simulation. Before the simulation, the Modelica compiler takes the Modelica model as input and removes the object-orientation and transforms the model to a flat equation system. This equation system is then used in the simulation. The result of the simulation is usually plotted as a graph describing how the interesting variables vary over time. The connections in Modelica are *undirected*, and correspond to equations. This is in contrast to the connections in ControlBuilder which are *directed*, and correspond to dataflow.

The object-oriented mechanisms in Modelica make it possible to create reusable components, for instance, to build libraries for different domains. These mechanisms are also useful in structuring the models, to decompose problems into smaller parts.

Modelica has both a visual and a textual syntax. The textual syntax is complete and covers the whole language and is used as the serialization format. The visual syntax is only partial and is mostly for users that connect existing component types, for instance, from libraries. The library can specify how these components are visualized using *graphical annotations*. These annotations are attached to component types and describe how instances of the component types are visualized. This makes it possible, for instance, to have a library for electrical circuits

```
model Tank
  parameter Real Area=1;
  Real level;
  Stream Inlet, Outlet;
equation
  Area*der(level) = Inlet.volumeFlowRate + Outlet.volumeFlowRate;
end Tank;
```

Figure 4: Tank model in Modelica describing the liquid in a tank. (Example adapted from [Mod00].)

with the visual notation that is standardized and used in the application domain. Library developers, on the other hand, typically use the textual syntax since they use more advanced language mechanisms.

Modelica has also been used for safety-related control applications [Thi15], where the language has been restricted to be suitable for safety-related development. For example, in the restricted version of the language, only directed connections are allowed, like ControlBuilder, and algebraic loops are forbidden.

Modelica Examples

Figure 4 shows an example Modelica model describing the liquid level in a tank [Mod00]. A Modelica model consists of two parts, the component part with model instances and variables, and the equation part that expresses relations between values. In the component part, the tank model declares two `real` variables, one describing the area of the tank and one describing the current liquid level, which is unknown. The tank has also an inlet and an outlet, of type `Stream` that is defined elsewhere, describing the incoming and outgoing volume flow, respectively. The equation part describes how the liquid level relates to the incoming and outgoing volume flow. This kind of equation is called a *mass balance equation*, which describes the mass of a system using the inflow and outflow. The equation uses the built-in operator `der`, which is the time derivative of a given variable. This model can be used to simulate how the liquid level varies over time and the result is typically plotted in a graph.

Like ControlBuilder diagrams, models in Modelica are user-defined types, which allow them to be instantiated as components in other models. For instance, we can create a new model that has two components of the `Tank` model we just defined.

Modelica's object-orientation allows models to extend other models. For example, the tank model can be extended to describe the temperature of the inlet and/or outlet streams as well. Modelica introduced the inheritance mechanism *redeclare*, which allows a subtype to change the type for a component defined in the

```
model circuit
  Resistor R1(R=10);
  Capacitor C(C=0.01);
  Resistor R2(R=100);
  Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.p);
end circuit;
```

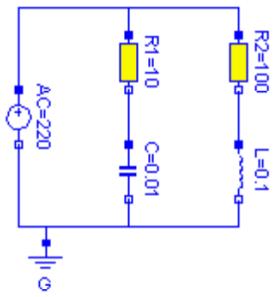


Figure 5: Simple electrical circuit in Modelica. Both the textual and visual syntax are shown.

supertype. For the tank example, the new subtype can redeclare the type for the components `Inlet` and `Outlet` to a type that includes the temperature.

Another example from [Mod00] that describes a simple electrical circuit is shown in Figure 5, where both the textual and the visual syntax are shown. This model contains components from the standard library and connects them using `connect` statements. The `connect` statements provide a shorthand for relating components and correspond to ordinary algebraic equations. What these corresponding equations look like depends on what is connected.

2.3 Object-Orientation in IEC 61131-3

The latest edition of the standard IEC 61131-3 from 2013 includes object-oriented mechanisms [Wer09]. As described earlier, this standard includes five programming languages used in automation to program PLCs. In this standard, block types are called function blocks. The new edition takes inspiration from ordinary object-orientation languages, like Java and C++, and adds methods to function blocks. The methods can then be overridden in subtypes. The new standard also adds interfaces, like in Java, that consists of method declarations, which can then be implemented by function blocks. The standard supports references to interfaces with reference semantics, which allows references to be changed dynamically and to be passed around. These references can be used for dynamic dispatch on methods. Bloqqi is quite different from this approach, and instead adds inheritance to diagrams, following the approach by Modelica, including the inheritance mecha-

nism block redeclaration. References and dynamic dispatch are not supported in Bloqqi, and all objects are known statically.

2.4 State-Based Languages

Another way to specify behavior and control systems is to use *state-based languages*, based on finite state machines (FSMs). In these languages, nodes represent states and connections represent state transitions. Thus, the connections describe control-flow rather than data-flow. There are several extensions to finite state machines, like *Statecharts* by Harel [Har87], which adds, for instance, nested states. Modelica has also introduced nested state machines [Elm+12], similar to Statecharts, that can be used to specify and generate control systems.

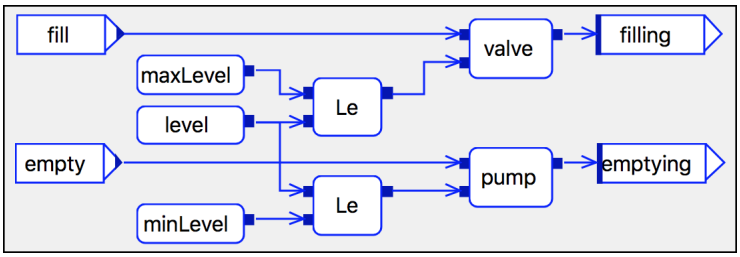
An ordinary finite state machine can only be in one state at a time. There are other languages that allows the program to be in several states at the same time, one example is Petri nets [Pet77]. One of the languages in the IEC 61131-3 standard is *Sequential Function Charts* (SFC) [JT10], which is based on Petri nets. Grafchart is a state-based language for automation that is based on both Petri nets and Statecharts [JÅ98; The14].

2.5 Feature-Oriented Software Development

The feature-based mechanisms in Bloqqi are related to *feature-oriented software development* (FOSD) [AK09], like *feature diagrams* [Kan+90] and *feature-oriented programming* [Pre97] (FOP). The idea in FOSD is to describe a software system in terms of what features it provides, which can be used for defining *software product lines*. These features can be described in a *feature model*, often visualized as a feature diagram, which has the form of a tree with mandatory, optional, and alternative features. The features in these models are often global for the whole system. In contrast, the feature-based mechanisms in Bloqqi are local, and are described in terms of block types. Our approach is more similar to FOP, which allow features to be described in an ordinary object-oriented language. The features in FOP are specified on the method level, whereas they are specified on the statement level in Bloqqi. The automatically derived wizard in Bloqqi was initially inspired by FeaturePlugin [AC04], an Eclipse plugin for feature modeling.

3 Introduction to Bloqqi

This section introduces Bloqqi, by first describing the influences from ControlBuilder and Modelica and then describing the language using a tank regulator example. We will describe what a diagram looks like, then extend the diagram using inheritance, and finally illustrate recommendations, our novel mechanism for feature-based programming.




```
diagramtype Tank(fill: Bool, empty: Bool =>
    filling: Bool, emptying: Bool) {
  maxLevel: OperatorInt;
  level: SensorInt;
  minLevel: OperatorInt;
  Le_1; Le_2;
  valve: Valve;
  pump: Pump;
  connect(fill, valve.open);
  ...
}
```

Figure 6: Simple tank regulator in Bloqqi.

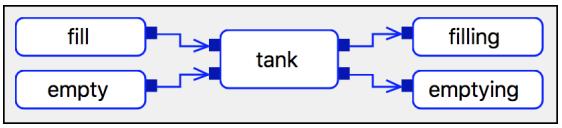
Bloqqi is a data-flow programming language with focus on automation. The execution scheme in Bloqqi is periodic, like ControlBuilder, but without the support for distributed execution or online update. In contrast to ControlBuilder, Bloqqi is object-oriented, like Modelica, and allows diagrams to be extended using inheritance. The inheritance mechanism of *block redeclaration* found in Modelica is also supported in Bloqqi, but in a slightly different form. Like Modelica, Bloqqi supports a textual syntax for the complete language, and a graphical syntax for showing the contents of a diagram.

In contrast to ControlBuilder and Modelica, Bloqqi supports the inheritance mechanism *connection interception* (Paper I & III), *visual instance inlining* (Paper II), and the feature-based mechanisms *wirings* and *recommendations* (Paper III).

3.1 Tank Regulator Example

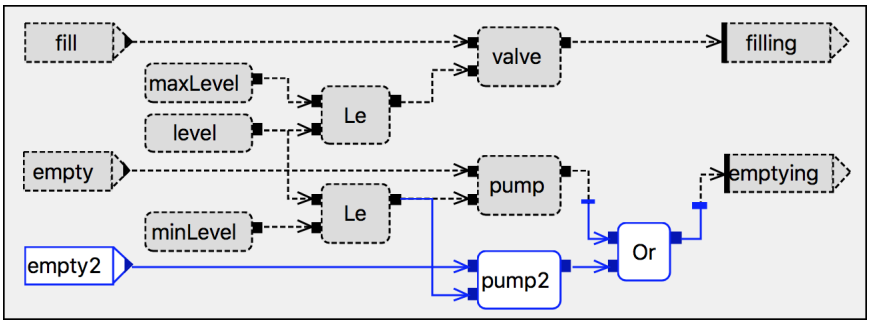
Figure 6 shows the same tank regulator in Bloqqi as we saw for ControlBuilder in Figure 3. As we can see, the diagram has both a visual and a textual syntax.

Having defined the simple tank regulator as a diagram, we can now instantiate it as a block in other diagrams. This is shown in Figure 7, where the diagram `Main` is defined, which contains the block `tank` with the block type of the previously defined diagram `Tank`. The tank is connected to the operator value blocks `fill` and `empty` and the actuator blocks `filling` and `emptying`. The parameters defined in `Tank` are shown as input and output ports on the diagram. In the editor, the user can hover the mouse over the ports to see their name and data type (in this case, `Bool`).




```
diagramtype Main() {
  ...
  tank: Tank;
  ...
}
```

Figure 7: Using the diagram `Tank` defined in Figure 6.




```
diagramtype TankTwoPumps(empty2: Bool) extends Tank {
  pump2: Pump;
  Or_2;
  intercept emptying with Or_2.in1, Or_2.out;
  ...
}
```

Figure 8: Using the diagram `Tank` defined in Figure 6.

3.2 Extending the Tank

Having defined the tank regulator as a diagram, the diagram can now be extended with new functionality using the inheritance construct in Bloqqi. In this example, we will extend the diagram with one more pump for emptying the tank.

When a diagram S extends another diagram T, we say that S is a subtype of T and T is a supertype of S. The subtype will inherit all parameters, blocks, and connections, but can also declare new parameters, blocks and connections. The subtype can also specialize the behavior defined in the supertype with the mechanisms *connection interception* (Paper I & III) and *block redeclaration* [Mod12]. Connection interception allows a connection defined in a supertype to be intercepted, that is, the connection will go via another block. Block redeclaration allows the block type for a block declared in a supertype to be specialized.

Figure 8 shows the diagram `TankTwoPumps` which extends the diagram `Tank` and adds a second pump for emptying the tank. The diagram also adds an input parameter `empty2` that controls the second pump. We can see that the visualization of the diagram shows all parameters, blocks and connections, including the

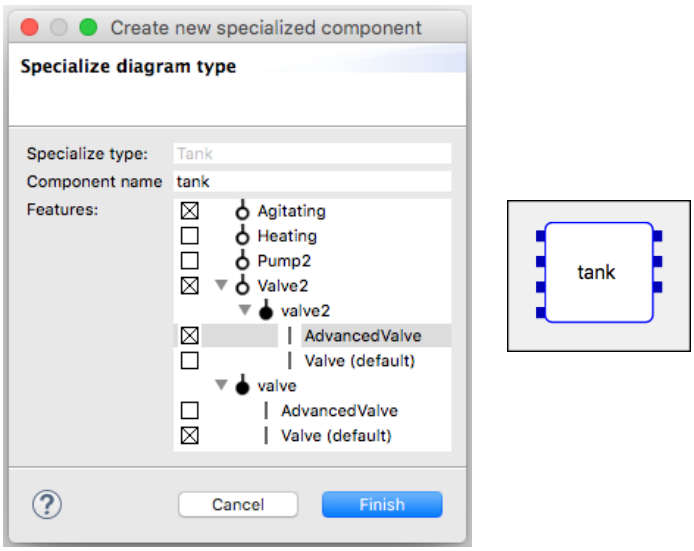


Figure 9: Automatically derived wizard when creating a block of the diagram `Tank`, which has recommendations. Optional features have hollow circles and recommended features have filled circles. There are two alternative implementation for the valves.

inherited ones. The user cannot edit the inherited parts, and they are therefore visualized differently from the local parts.The inherited parts are visualized as grey (like stone) with dashed lines, and the local parts are visualized as blue with solid lines. The diagram `TankTwoPumps` specializes the behavior defined in the supertype `Tank` by intercepting the connection to the output parameter `emptying`. In the supertype, there is a connection from the block `pump` to the parameter `emptying`, but in the subtype, the connection is replaced with two connections so that the data-flow will go via the local block `Or`.

3.3 Features using Recommendations

The previous example illustrated how a diagram can be extended with more functionality; but having one subtype for each supported variant will lead to a combinatorial explosion of diagrams. The Bloqqi language also supports the feature-based language mechanisms *wirings* and *recommendations*. These mechanisms are used in the editor to automatically derive smart-editing support for the user, in the form of a wizard. In the wizard, the user can select which features a block should have.

Features for the tank example include a second pump for emptying, a second valve for filling, and heating functionality. Assuming that there are recommendations for the diagram `Tank`, the derived wizard is displayed when the user creates a block of the type `Tank`. The derived wizard is shown in Figure 9. In this wizard,

the user can select which features the block should have. Optional features are depicted with hollow circles and mandatory features are depicted with filled circles. Sometimes there are alternatives for a feature, indicated by vertical bars. In this case, the valve can be replaced with a more advanced valve, which can be done for both the mandatory feature `valve` and the optional feature `valve2`. The figure also shows the block with the selected features, which has been generated by the editor.

The recommendations for the diagram `Tank` are specified as follows.

```
recommendation Tank {
  Valve2: ValveExtension[...];
  Pump2: PumpExtension[...];
  Heating: HeatingExtension[...];
  Agitating: AgitationExtension[...];
  replaceable valve;
}
```

There are four optional features for `Tank`, each having a name and a type with a *wiring application* (within the square brackets (`[]`)). The wiring application tells how the block is inserted into the diagram `Tank`, if the feature is selected. The keyword `replaceable` says that the block `valve` defined in `Tank` can be specialized, that is, redeclared to a subtype of the type for `valve`. In this example, the type for `valve` is `Valve`, and there exists one subtype of `Valve` called `AdvancedValve`. The editor will automatically find all subtypes of the type for `valve` and show them as alternatives in the wizard.

One important property of recommendations is that they are specified *modularly*, that is, separate from the type for which they add recommendations to. Furthermore, recommendations for a diagram can be divided into separate recommendation statements. For example, the recommendations above could be split into five different recommendation statements defined in different files. Also, alternatives are added by subtyping, which are automatically found by the editor, which makes it possible to add new alternatives modularly. The modularity aspect of recommendations allows libraries to be extended with more recommendations that are defined outside the library.

4 The Implementation of the Bloqqi Compiler and Editor

Paper IV is about the implementation of the Bloqqi compiler and editor. The tools serve different roles but share a common core. The compiler takes a Bloqqi program as input, in the form of a file, and generates C code as output, which can be compiled into an executable file. If the Bloqqi program contains errors, the compiler will instead print the errors and exit. Like the compiler, the editor reads a Bloqqi program from a file, but instead of generating code, the editor visualizes

the program. All Bloqqi diagrams in this dissertation are screenshots from the editor. In the editor, the user can then change the program interactively and store the changed program back to the file. In order to generate code or to visualize a diagram, many of the same analyses are needed. Thus, the specification of the analyses are shared by both the compiler and the editor, in order to avoid duplicated code. The editor uses these analyses to provide semantic feedback for the user.

4.1 Reference Attribute Grammars

The compiler and the editor have been implemented using the metacompilation system JastAdd [EH07b] and the semantic formalism *reference attribute grammars* (RAGs) [Hed00]. JastAdd takes a set of semantic specifications and generates Java code implementing these. Thus, both the compiler and the editor are Java programs.

One big advantage of using RAGs is its support for extensibility and modularity, allowing tools to developed as modules and later extended with more modules. This can be used, for example, to extend the tool to perform more advanced semantic analysis or extending the language with more language features. An example of an extensible compiler built using RAGs is the Java compiler *ExtendJ*[1] which is organized by having one module for each version of the language [EH07c]. The ExtendJ compiler has also been extended with, for example, refactoring support [Sch+08; Sch+09] and non-null analysis [EH07a]. Another example is the JModelica.org compiler that is implemented using JastAdd [Åke+10a], and later extended with language mechanisms for optimization of dynamic models as well [Åke+10b]. In a similar way as these examples, the Bloqqi compiler and editor share a common semantic core, and the editor extends the core with analyses useful in the editor, such as advanced editing operations. For an introduction to RAGs, see [Hed11; FH15].

The central data structure in tools developed with JastAdd is the *decorated abstract syntax tree* (AST). Typically, tools developed with JastAdd parse the input files into ASTs, which are then decorated with *attributes*. Attributes are attached to tree nodes and are *computed node properties*, defined by *equations*. The attributes are computed on demand, that is, when accessed, and memoized for subsequent accesses to avoid computing the same attribute value several times. The equations defining the attributes are *declarative*, in the sense that they are free from externally observable side effects. RAGs is an extension to ordinary *attribute grammars* by Knuth [Knu68] and adds reference attributes, allowing attributes to refer to nodes in the tree. This makes it easy to super-impose graphs over the AST, for instance, to describe name binding relations and data-flow graphs, which makes it suitable for graph-based languages like Bloqqi.

A typical example of a reference attribute is from the name analysis, where nodes representing name uses have a reference attribute called `decl()` that refers

[1] See `http://extendj.org`. ExtendJ was previously called JastAddJ.

to the corresponding name declaration. Another example of a reference attribute is the set of successors of a block, which is called `succ()` and declared for block nodes in the AST. The value of this attribute is a set of references that refer to other block nodes in the AST. The attribute value is computed based on name analysis, since the information depends on `connect` statements, which refer to blocks by name and are resolved by name analysis.

4.2 Computed Structures

There are different kinds of attributes, with distinct characteristics. One important kind used in the compiler and editor is *non-terminal attributes* (NTAs) [Vog+89] (also known as *higher-order attributes*). An NTA is an attribute whose value is a new fresh subtree, defined by an equation like any other attribute. This in contrast to ordinary subtrees in the AST, which are typically created by the parser. NTAs make it possible to augment the AST with new structures that are *computed*, which can be based on other attributes, for instance, on attributes describing the type and name analysis.

As an example of using NTAs in the Bloqqi tools, each diagram node has an NTA called `blocks()`. As we can see in Figure 8, when a diagram is visualized, all blocks are shown, both local and inherited, but with different colors. To do this, the editor needs to perform name and type analysis, in order to compute which blocks to visualize. This computation is defined as the NTA `blocks()`, which includes all blocks for the diagram, including the inherited ones. This attribute is an example of a computed property that is used by both the compiler and the editor. The compiler uses the attribute to find semantic errors and to generate C code. The attribute is needed in the code generation to remove inheritance since inheritance is not supported in C.

4.3 Visualizing Computed Structures

The editor is implemented as an Eclipse plugin using the Graphical Editing Framework (GEF)[2]. This framework is based on the model-view-controller (MVC) pattern. In the Bloqqi editor, we use the decorated AST as the model, and the view objects, which are written in plain Java, describe how this model is visualized. When a diagram is visualized, the NTA `blocks()` is used, together with other NTAs describing all connections, all parameters, and so on, which all include the inherited ones. There are also other attributes that are used in the visualization, such as boolean properties for if a block is local or inherited, which is used to color the block white or grey.

Most of the visualization of a diagram is computed. In addition to inheritance, the ports on a block are also computed. The ports correspond to the parameters in the block type. Since the visualization of diagrams is computed, we have not found

[2] `http://www.eclipse.org/gef/`

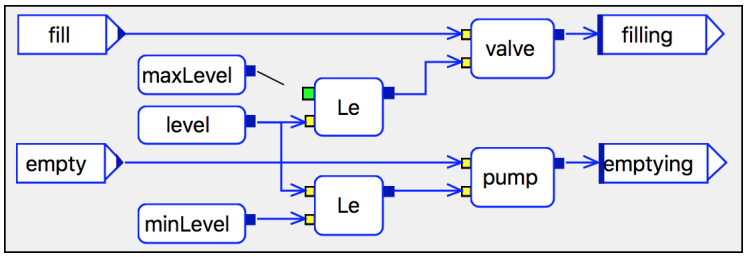


Figure 10: Semantic feedback when the user creates a connection. Connecting it to a green port will not introduce any semantic errors, but connecting it to a yellow port will introduce semantic errors.

existing visual syntax formalisms, like constraint multiset grammars [Mar94] and graph grammars-based formalisms [Min02; Min06], suitable for defining the visual syntax of Bloqqi.

4.4 Semantic Feedback

By reusing the decorated AST as the model in the editor, we can use semantic analysis defined for the compiler to provide semantic feedback to the user. By extending the compiler specification, we can also define new analysis that builds on the semantic analysis defined for the compiler, to provide even more semantic feedback. There are two kinds of semantic feedback, one that we call *permanent semantic feedback* and the second one is called *transient semantic feedback*, which is only shown temporarily while the user performs an edit operation.

An example of permanent semantic feedback is when a connection that has incompatible types is visualized and the editor colors it red. An example of transient semantic feedback is when the user creates a connection. This is shown in Figure 10, where the user has started to create a connection from the output port on the `maxLevel` block. The editor colors input ports yellow or green, depending on if connecting to them would give semantic errors or not. Semantic errors include that the types are incompatible and that the input port does not already have an incoming connection, since only one incoming connection is allowed for input ports.

4.5 Integration with FMI and ControlBuilder

We have experimented with integrating the C code generated by the Bloqqi compiler with the standard Functional Mockup Interface [Blo+12] (FMI). This is a standard that describes a common export format for dynamic models, which allows models to be used together even if they have been exported with different tools. For example, many Modelica compilers support exporting models as simu-

lation binaries in the FMI format. We have used this to connect Bloqqi programs to simulations described by Modelica models. For instance, we have run a Bloqqi program to control a simulated tank process.

We have also experimented with integrating Bloqqi with ControlBuilder, with two alternative approaches. One approach is to generate ControlBuilder programs from Bloqqi programs and the other approach is to integrate the generated C code from the Bloqqi compiler as external executables in ControlBuilder. We have used this to run Bloqqi programs on specialized controller hardware provided by ABB.

5 Contributions

This section describes the contributions of this dissertation. Paper I-III describe the novel language mechanisms in Bloqqi and Paper IV describes the implementation of the compiler and editor for Bloqqi. The main contributions of this dissertation are:

- **Connection interception (Paper I & III).** Connection interception is an inheritance mechanism for block diagrams that allows subtypes to intercept connections defined supertypes, making the connections go via other blocks. Paper I introduces *target interception*, where the connection to be intercepted is identified by the target of the connection. Paper III introduces *source interception*, where the connection to be intercepted is identified by the source of the connection. Source interception may replace several connections, since a source port may have several outgoing connections. In contrast, target interception only replaces one connection, since a destination port can only have one incoming connection.

- **Visual instance inlining (Paper II).** Visual instance inlining allows a block to be visually inlined, replacing the block with its content. The domain engineer can use this mechanism to make the diagrams look like how they are drawn on paper. The editor provides editing operations for the user to connect to the inlined block's content. When the user connects to the content of an inlined block, the editor will implicitly create anonymous subtypes and use block redeclaration.

- **Wirings and recommendations (Paper III).** Wirings and recommendations are feature-based language mechanisms that modularly describe variability of a base structure. This is used to derive smart-editing support in the form of a wizard, where the user can select the desired features. After the specialized block has been created, the user can change the feature configuration of the block. Thus, Bloqqi supports *staged configuration* [Cza+04]. The modular property of the mechanisms allows libraries to be created and extended with more features specified outside the library.

- **Prototype implementation (Paper IV).** The language mechanisms have been implemented in a prototype compiler and editor. Paper IV presents the architecture for the tools and how they can be implemented using RAGs, including techniques for implementing computed visual syntax and semantic feedback. The paper also presents what the generated C code looks like for Bloqqi programs. The generated C code has been used to integrate Bloqqi programs with FMI and ControlBuilder, to control simulated models for testing purposes and to execute Bloqqi programs on controller hardware, respectively. The implementation is released as open source under the license Modified BSD License. The source code can be accessed at: `https://bitbucket.org/bloqqi/`

6 Included Papers

The following papers are included in this dissertation.

Paper I. Niklas Fors and Görel Hedin. "Intercepting dataflow connections in diagrams with inheritance". In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2014, pp. 21-24.

This paper introduces the concept of (target) connection interception. The paper was written before the invention of the wiring and recommendation constructs, and instead uses multiple inheritance to model variants. The paper describes an early version of Bloqqi, called *PicoDiagram*.

Paper II. Niklas Fors and Görel 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. ACM, 2016, 4:1-4:10.

This paper describes visual instance inlining.

Paper III. Niklas Fors and Görel Hedin. "Bloqqi: Modular Feature-Based Block Diagram Programming". In: *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* 2016, Amsterdam, Netherlands, 30 October - 4 November, 2016. In press.

This paper describes the feature-based constructs wirings and recommendations. In this paper, we use composition to describe variability instead of multiple inheritance that we used in Paper I. This paper also introduces source interception that complements target interception described in Paper I.

Paper IV. Niklas Fors. "Implementation of the Bloqqi compiler and editor". Technical Report 98. LU-CS-TR:2016-252, ISSN 1404-1200. Lund University, 2016.

This report describes the implementation of the Bloqqi compiler and editor, including the architecture for the tools. The report is partially based on the related Papers VII-IX. The work carried out for the related Papers X-XI gave useful insights on how to specify compilers using reference attribute grammars, which have been used in the implementation of the Bloqqi tools.

6.1 Methodology

Denning et al. [Den+89] describe the computing discipline as a combination of three paradigms: *theory*, *abstractions*, and *design* [TS08]. These paradigms have their roots in mathematics, science, and engineering, respectively. We have in this dissertation used the design paradigm. This paradigm is about constructing a system for a given problem, thus, constructing *useful* artifacts. According to Denning et al., the steps for following the design paradigm are: 1) state requirements, 2) state specifications, 3) design and implement the system, and 4) test the system. These steps should be carried out in an iterative manner.

The new language mechanisms presented in this dissertation have been developed iteratively, with monthly meetings with ABB over several years, where they presented challenges they faced. The challenges were illustrated with example programs that were difficult to program in ControlBuilder in a reusable way. The challenges and the examples formed the requirements for the new mechanisms. The new mechanisms have been presented at meetings with ABB and refined many times. During the meetings, we discussed the mechanisms, which gave useful feedback for refining and improving the mechanisms. We also found it useful to have both a textual and visual syntax during the discussions, to concretize what we were talking about to increase our understanding of the mechanisms and the examples. The mechanisms have been evaluated by a prototype implementation, thus, we have designed and implemented the mechanisms as a proof of concept, and tested them. Even though the mechanisms were motivated by concrete examples from ABB, we have made the mechanisms as general as possible, so that they can be applied for other languages as well.

6.2 Contribution Statement

Niklas Fors is the main author of Papers I-III, which have been co-written with Görel Hedin. For these papers, Niklas Fors drafted the first version of the paper with text for all parts, which was then revised to final form together with Görel Hedin in a close collaboration. For Paper IV, Niklas Fors is the sole author. The overall ideas for the proposed language mechanisms presented in Papers I-III have been jointly developed with Görel Hedin in an iterative manner, motivated by examples provided by the company ABB. Niklas Fors is the main contributor to the design of the language mechanisms in Bloqqi capturing these ideas. Niklas Fors is

also the main contributor of the implementation techniques described in Paper IV. All the implementation work has been carried out by Niklas Fors.

7 Related Papers

The following papers are related, but not included, in this dissertation.

Paper V. Niklas Fors and Görel Hedin. ”Handling of layout-sensitive semantics in a visual control language”. In: 2012 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2012, Innsbruck, Austria, September 30 - October 4, 2012. 2012, pp. 249-250.

This is the first paper about PicoDiagram. Inspired by ControlBuilder, the execution order of the blocks depends partially on the visual layout of the blocks, since the connections do not give a total order.

Paper VI. Niklas Fors. ”Supporting Visual Editors using Reference Attributed Grammars”. In: Proceedings of the Doctoral Symposium of the 5th International Conference on Software Language Engineering 2012, Dresden, Germany, Sep 25, 2012. 2012, pp. 15-22.

This is a paper for a doctoral symposium that describes the ideas of using RAGs to share semantic specifications between tools and use the semantic specification in the graphical editor to provide semantic feedback.

Paper VII. Niklas Fors and Görel Hedin. ”Reusing Semantics in Visual Editors: A Case for Reference Attribute Grammars”. In: ECEASST 58 (2013).

This paper describes in more detail the execution order of blocks that partially depends on the visual layout of the blocks. To compute the execution order, first the connections are considered, and second, the layout of the blocks is considered, to give a total execution order. The execution order is defined using RAGs and is used in the visual editor to give semantic feedback. For example, the execution order number is shown on the blocks. The user also gets semantic feedback when moving a block, by showing in which area the block can be moved without changing the execution order.

Paper VIII. Niklas Fors and Görel Hedin. ”Implementing Semantic Feedback in a Diagram Editor”. In: Proceedings of the Second Workshop on Graphical Modeling Language Development. GMLD ’13. Montpellier, France: ACM, 2013, pp. 42-50.

This paper describes more examples of semantic feedback and the implementation of it using RAGs. More specifically, the paper deals with, among other things, diagrams that contain data-flow cycles and how these cycles are handled using the visual layout. Later, after the publication of this paper, it

was decided to forbid data-flow cycles in Bloqqi, and instead handle data-flow cycles using variables defined by the user. The values of the variables are then stored between the execution periods.

Paper IX. Niklas Fors and Görel Hedin. "Using refactoring techniques for visual editing of hybrid languages". In: Proceedings of the 2013 ACM Workshop on Refactoring Tools, WRT@SPLASH 2013, Indianapolis, IN, USA, October 27, 2013. 2013, pp. 17-20.

This paper describes how refactoring techniques for textual languages can be used for implementing visual editing operations for *hybrid languages*, that is, languages with both textual and visual syntax. Textual languages have name binding rules that define how name uses are bound to name declarations. These rules need to be considered when implementing editing operations correctly.

Paper X. Niklas Fors and Görel Hedin. "A JastAdd implementation of Oberon-0". In: Science of Computer Programming 114 (2015). pp. 74-84.

This paper presents the implementation of a compiler specified in JastAdd for the language Oberon-0. The language is a small Pascal-like language that was introduced by Niklaus Wirth for teaching compilers [Wir96]. The paper was part of the LDTA 2011 Tool Challenge. The goal of the tool challenge was to compare different compiler tools and techniques for the same language.

Paper XI. Niklas Fors, Gustav Cedersjö, and Görel Hedin. "JavaRAG: A Java Library for Reference Attribute Grammars". In: Proceedings of the 14th International Conference on Modularity. MODULARITY 2015. Fort Collins, CO, USA: ACM, 2015, pp. 55-67.

Developing a tool using JastAdd requires the tool to use the data structures generated by JastAdd. Sometimes, this is not feasible. For example, we may already have an existing data structure, for instance, an EMF model[3], that we want to add attributes to, or maybe we do not want to add an extra step in the build process. To alleviate these issues, this paper presents a library in Java called JavaRAG, which allows arbitrary data structures in Java to be decorated with attributes. The requirement is that a spanning tree can be formed over the data structure. JavaRAG is implemented using Java annotations and reflection, and have no other dependencies.

[3] `http://www.eclipse.org/modeling/emf/`

8 Conclusion

This dissertation presents the design and implementation of the feature-based data-flow language *Bloqqi*.

The proposed language mechanisms in Bloqqi focus on reusability and variability. Diagrams can be extended and specialized using *connection interception*. The connection to be intercepted is identified either by the source or the target of the connection. *Visual instance inlining* allows blocks to be visually inlined, visually replacing the block with its content. This enables the domain engineer to create diagrams like how they are drawn on paper. The feature-based mechanisms *wirings* and *recommendations* are used to specify variability to derive smart-editing support in the form of a wizard, where the user can select the desired features. Recommendations are specified in a modular way, so that they can be added and extended for existing libraries.

The compiler and editor for Bloqqi have been implemented using the semantic formalism reference attribute grammars, where the semantic specification is shared by both the tools to avoid code duplication. One interesting thing with the Bloqqi language is that the visualization of diagrams are computed, based on the semantic analysis. For example, to visualize diagrams, inherited blocks and ports on blocks are dependent on the semantic analysis. By reusing the semantic specification defined for the compiler, the editor can give advanced semantic feedback to the user for better user interaction.

The proposed language mechanisms have been tested within the automation domain, but we think the mechanisms are useful for other data-flow programming languages as well, which would be an interesting area of future work. It would also be interesting to combine the mechanisms with state-based languages, like Grafchart [JÅ98].

Bloqqi is a prototype language for experimenting with new language mechanisms. Many of the features in ControlBuilder are used for building distributed control systems, like online updating. It would be interesting to investigate how Bloqqi can be used for distributed control. Another thing ControlBuilder supports is user interfaces for the human operators. It would be interesting to see how these user interfaces could be generated in combination with the language mechanisms presented in this dissertation. Also, ControlBuilder supports *control connections*, where parts of the structured data can be reversed, allowing the data to flow backwards along the connections. This is used to simplify diagrams and to improve control performance [PH02].

The Functional Mockup Interface [Blo+12] (FMI) allows models defined in different tools to be exported as Functional Mockup Units (FMUs) and used together. In this dissertation, we have integrated the generated C code by the Bloqqi compiler with Modelica models exported as FMUs. It would be useful to export Bloqqi programs as FMUs, so that Bloqqi programs easily can be integrated in other tools. When several FMUs are connected, these FMUs need to be connected

in some way, and it would be interesting to investigate if Bloqqi could be used for connecting them (an FMU can be seen as a block with input and output).

The visual syntax of Bloqqi is computed and we have implemented the visualization in the graphical editor with ordinary Java code. It would be interesting to investigate how a more higher-level specification could be used instead. Also, currently in the graphical editor, when a change is made by the user, the values of all attributes are cleared and recomputed again when they are needed, for example, when the visualization is re-rendered. To improve this, incremental evaluation could be used [Rep82; SH12], where only attribute values that are affected by the change are cleared. One challenge in incremental evaluation is the tradeoff between precision and overhead.

References

[Åke+10a] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. "Implementation of a Modelica Compiler Using JastAdd Attribute Grammars". In: *Science of Computer Programming* 75.1-2 (Jan. 2010), pp. 21–38.

[Åke+10b] Johan Åkesson et al. "Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problem". In: *Computers and Chemical Engineering* 34.11 (Nov. 2010), pp. 1737–1749.

[AC04] Michal Antkiewicz and Krzysztof Czarnecki. "FeaturePlugin: feature modeling plug-in for Eclipse". In: *Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange, ETX 2004, Vancouver, British Columbia, Canada, October 24, 2004*. 2004, pp. 67–72.

[AK09] Sven Apel and Christian Kästner. "An Overview of Feature-Oriented Software Development". In: *Journal of Object Technology* 8.5 (2009), pp. 49–84.

[Blo+12] Torsten Blochwitz et al. "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models". In: *9th International Modelica Conference*. 2012.

[Con16] ControlBuilder. *Compact Product Suite. Compact Control Builder AC 800M. Product Guide. Version 6.0.* Available from `abb.com`. Document number: 3BSE041586-600 A. ABB. 2016.

[Cza+04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. "Staged Configuration Using Feature Models". In: *Proc. 3rd Int. Conf. Software Product Lines*. 2004, pp. 266–283.

[Den+89] Peter J. Denning et al. "Computing as a Discipline". In: *Commun. ACM* 32.1 (1989), pp. 9–23.

[Deu+00] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *SIGPLAN Notices* 35.6 (2000), pp. 26–36.

[EH07a] Torbjörn Ekman and Görel Hedin. "Pluggable checking and inferencing of nonnull types for Java". In: *Journal of Object Technology* 6.9 (2007), pp. 455–475.

[EH07b] Torbjörn Ekman and Görel Hedin. "The JastAdd system - modular extensible compiler construction". In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[EH07c] Torbjörn Ekman and Görel Hedin. "The Jastadd Extensible Java Compiler". In: *OOPSLA 2007*. ACM, 2007, pp. 1–18.

[Elm+12] Hilding Elmqvist et al. "State machines in modelica". In: *Proceedings of the 9th International MODELICA Conference*. 76. Linköping University Electronic Press. 2012, pp. 37–46.

[FH15] Niklas Fors and Görel Hedin. "A JastAdd implementation of Oberon-0". In: *Science of Computer Programming* 114 (2015). {LDTA} (Language Descriptions, Tools, and Applications) Tool Challenge, pp. 74 –84.

[Fri+05] Peter Fritzson et al. "The OpenModelica Modeling, Simulation, and Software Development Environment". In: *Simulation News Europe* 15.44/45 (2005), pp. 8–16.

[Har87] David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231 –274.

[Hed00] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)*. 24(3). 2000, pp. 301–317.

[Hed11] Görel Hedin. "An Introductory Tutorial on JastAdd Attribute Grammars". In: *Generative and Transformational Techniques in Software Engineering III*. Vol. 6491. LNCS. Springer, 2011, pp. 166–200.

[Hud97] Paul Hudak. "Domain-specific languages". In: *Handbook of Programming Languages* 3.39-60 (1997), p. 21.

[JT10] 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.

[JÅ98] Charlotta Johnsson and Karl-Erik Årzén. "Grafchart for recipe-based batch control". In: *Computers & Chemical Engineering* 22.12 (1998), pp. 1811 –1828.

[Kan+90] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. DTIC Document, 1990.

[Knu68] Donald E. Knuth. “Semantics of Context-free Languages”. In: *Math. Sys. Theory* 2.2 (1968). Correction: *Math. Sys. Theory* 5(1):95–96, 1971, pp. 127–145.

[Mar94] K. Marriott. “Constraint multiset grammars”. In: *Visual Languages, 1994. Proceedings., IEEE Symposium on*. IEEE. 1994, pp. 118–125.

[Mer+05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and how to develop domain-specific languages”. In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344.

[Min02] M. Minas. “Concepts and realization of a diagram editor generator based on hypergraph transformation”. In: *Science of Computer Programming* 44.2 (2002), pp. 157–180.

[Min06] M. Minas. “Generating meta-model-based freehand editors”. In: *Proc. of 3rd Intl. Workshop on Graph Based Tools*. Electronic Communications of the EASST, 2006.

[Mod00] Modelica. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling v.1.4 - Tutorial*. Available from `modelica.org`. Modelica Association. 2000.

[Mod12] Modelica. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3*. Available from `modelica.org`. Modelica Association. 2012.

[NS07] Bob Nelson and Todd Stauffer. *DCS or PLC? Seven Questions to Help You Select the Best Solution*. Tech. rep. Siemens Energy & Automation, 2007.

[PH02] Lars Pernebo and Bengt Hansson. “Plug and play in control loop design”. In: *Proceedings for Control Meeting*. Linköping, Sweden, 2002.

[Pet77] James L. Peterson. “Petri Nets”. In: *ACM Comput. Surv.* 9.3 (Sept. 1977), pp. 223–252.

[Pre97] Christian Prehofer. “Feature-Oriented Programming: A Fresh Look at Objects”. In: *ECOOP*. 1997, pp. 419–443.

[Rep82] Thomas W. Reps. “Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors”. In: *POPL*. 1982, pp. 169–176.

[Sch+08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. “Sound and Extensible Renaming for Java”. In: *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. Ed. by Gregor Kiczales. ACM Press, 2008.

[Sch+09] Max Schäfer et al. “Stepping Stones over the Refactoring Rubicon”. In: *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. 2009, pp. 369–393.

[SH12] Emma Söderberg and Görel Hedin. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Technical Report 98. LU-CS-TR:2012-249, ISSN 1404-1200. Lund University, Apr. 2012.

[TS08] Matti Tedre and Erkki Sutinen. “Three traditions of computing: what educators should know”. In: *Computer Science Education* 18.3 (2008), pp. 153–170.

[The14] Alfred Theorin. “A Sequential Control Language for Industrial Automation”. PhD thesis. Department of Automatic Control, Lund University, Sweden, Nov. 2014.

[Thi15] Bernhard Amadeus Thiele. “Framework for Modelica Based Function Development”. PhD thesis. Universität München, 2015.

[Vog+89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *PLDI*. 1989, pp. 131–145.

[Wer09] Bernhard Werner. “Object-oriented extensions for IEC 61131-3”. In: *IEEE Industrial Electronics Magazine* 3.4 (2009), pp. 36–39.

[Wir96] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.

Intercepting Dataflow Connections in Diagrams with Inheritance

Abstract

Control systems are often built using visual dataflow-based languages, and supporting different variants may be challenging. We introduce the concept of *connection interception* based on inheritance. This mechanism allows a diagram to extend another diagram and intercept connections defined in the supertype, that is, to replace it by two other connections, in order to specialize the behavior. This can be used to create extensible libraries that support different variants.

1 Introduction

Control systems can be built using visual dataflow-based languages, such as the Function Block Diagram in the IEC 61131-3 standard or the Control Builder from ABB. A diagram in these languages contains input and output parameters, blocks with ports and connections that model the dataflow between the ports and parameters. These diagrams are executed in a periodic manner, for example, 100 times per second. For each period, sensor values are read to compute control signals that are sent to actuators. The Control Builder supports *visual user-defined types* in the form of diagrams, called diagram types, that can be instantiated as blocks in other diagram types. This makes it possible to create (non-recursive) hierarchies of blocks. An example of a diagram type is a proportional feedback controller that can be instantiated for several uses.

Niklas Fors and Görel Hedin. "Intercepting dataflow connections in diagrams with inheritance". In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2014, pp. 21-24.

Building a control system for a plant typically involves creating several controllers for different physical entities, such as tanks and engines. These controllers can be abstracted into libraries to enable reuse of functionality between similar entities in the same or different plants. Some entities both share similarities and have differences, and there is a need for managing these variabilities. An example of variants is a proportional feedback controller with or without gain scheduling, and with or without feedforward. There are two common ways for handling variability: copy-modify and parameterization. *Copy-modify* means that an existing diagram type is copied, and then modified to obtain the desired variant. This gives the well-known "code smell" of duplicated code, with the drawbacks of code bloat and double maintenance problems. *Parameterization* means that all interesting variants are anticipated and input parameters are used to decide which functionality to enable. This leads to complicated diagrams where often only a subset of the diagram is used at runtime.

In this paper, we introduce a third way of handling variability that overcomes the drawbacks of the copy-modify and parameterization approaches. Our approach is based on inheritance and a new concept of *connection interception* that allows a connection defined in a supertype to be replaced in a subtype by two other connections.

We have implemented a prototype compiler, which generates executable C code, and a visual editor for a language that supports diagram types, inheritance and connection interceptions. We have defined a textual syntax for the language, which both the compiler and visual editor use as the serialization format. The mechanism of connection interception has been developed in collaboration with ABB with the goal of improving reusability mechanisms in the control domain for the process industry. We have used existing libraries to identify which problems arise when handling variability.

The rest of the paper describes how inheritance and connection interception works and how multiple inheritance is handled, some notes on the implementation, and an evaluation, comparing the new approach with copy-modify and parameterization. The paper ends with related work and a conclusion.

2 Connection Interception

We will use inheritance to manage variabilities, where new elements can be added in subtypes. But only adding elements is not enough, subtypes need also the possibility to specialize the behavior. We introduce the mechanism of *connection interception* that allows subtypes to specialize the connection behavior in dataflow-based languages. This mechanism plays a similar role as method overriding in object-oriented languages.

With (multiple) inheritance, a diagram type D can *extend* other diagram types $B_1, ..., B_n$, and we say that D is a *subtype* of $B_1, ..., B_n$ and that $B_1, ..., B_n$ are

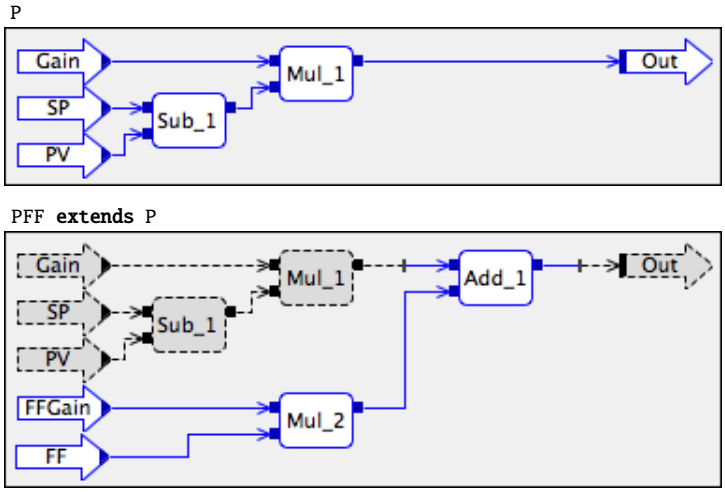


Figure 1: A P-controller (`P`) is extended with feedforward (`PFF`). Black/grey/-dashed indicates inherited elements, and blue/solid indicates elements that are locally defined. The connection from `Mul_1` to `Out` in `P` is intercepted in `PFF`, causing it to go via the new `Add` block.

supertypes of D. The subtype and supertype relationships are transitive. The subtype D *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 to the subtypes. For each supertype, all elements are only copied once to the subtype, even if there is more than one path in the type hierarchy from a subtype to a supertype, i.e., corresponding to so called *virtual inheritance* in languages like C++. The subtype D can also declare new parameters, blocks and connections. With connection interception, the subtype D can intercept a connection declared in a supertype, causing the connection to go via other blocks.

Suppose a diagram type B contains a connection from port p to port q, where q is either an output port (parameter) of B, or an input port on a block in B. In this case, a subtype D can intercept the connection by stating that q is intercepted by the pair (t, s), where t and s are ports declared in B and/or D. For a block of type D the connection (p, q) will then be replaced by the connections (p, t) and (s, q). This way, the connection to q is specialized by one or more other blocks before reaching q. An invariant of interception is that if a port in a diagram has incoming data, then it will have incoming data in all subtypes, but maybe from a different source.

As an example, Figure 1 shows two diagram types: one type that models a proportional feedback controller (`P`) and a subtype that extends it with feedfor-

```
diagramtype PFF(Int FFGain, Int FF) extends P {
  Mul Mul_2;
  Add Add_1;
  connect(FFGain, Mul_2.in1);
  connect(FF, Mul_2.in2);
  connect(Mul_2.out, Add_1.in2);
  intercept Out with Add_1.in1, Add_1.out;
}
```

Figure 2: Textual syntax of PFF in Figure 1

ward (PFF). The textual syntax of PFF is shown in Figure 2, and the textual syntax of P is similar. The P controller computes the error, that is, the difference between the set point (SP) and the process value (PV), and then multiplies the error with the proportional gain (Gain), which is sent to the output parameter Out. The subtype PFF adds input parameters, blocks, connections and a connection interception. The latter intercepts the connection to the output parameter Out with the pair (Add_1.in1, Add_1.out) that adds the feedforward value to Out. Other diagram types may also extend P and add, for example, gain scheduling, which allows several values for the gain.

In Figure 2, the intercepted connection is identified using only the target of the connection (Out). This works for ports/parameters with only one incoming connection. For ports/parameters with several incoming connections, a connection can be identified using both the source and the target of the connection, as the following code illustrates:

```
intercept Mul_1.out, Out with Add1.in1, Add1.out;
```

In this language, all diagram types, parameters, blocks (instances of diagram types) and connections are defined at compile time. Since the language does not include dynamic dispatch, the compiler can generate flattened code with no extra runtime support needed for inheritance.

2.1 Visual Appearance

The idea behind the visual appearance is that black and grey symbolizes stone, something old, something that you inherit and generally cannot change. For intercepted connections, the two replacing connections each have a dashed black part symbolizing the unchanged end of the connection and a blue part symbolizing the intercepting end. The notation has been guided by the principles described by Moody [Moo09]. For example, we use both color and texture to improve discriminability between inherited and local elements. Color makes it easier to differentiate between the two kinds of elements, but the coding is redundant, which allows diagrams to be printed black-and-white without loss of information.

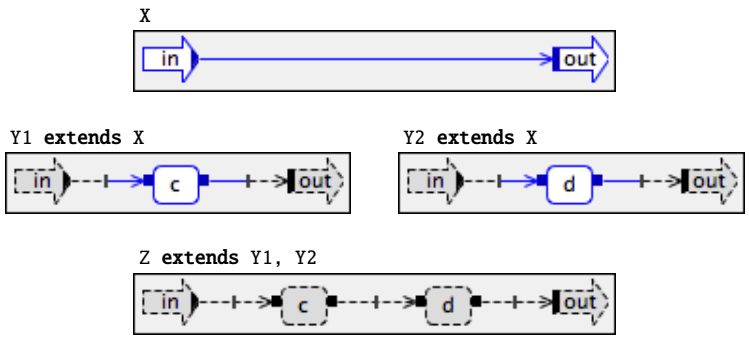


Figure 3: Both `Y1` and `Y2` intercept the connection between the parameters `in` and `out`. The order in the `extends` clause determines in which order interceptions are applied.

Some of the visual elements in the visual editor are computed using the semantics of the language, rather than being purely syntactical. We will call these *computed visual elements*. For instance, inherited blocks and connections are computed, as shown in Figure 1, and visualized in another way than local elements. Local interceptions is another example, where the resulting connections are computed, and shown as having an inherited part (dashed black) and a local part (solid blue). Computed visual elements are implemented using the semantic formalism reference attribute grammars [Hed00], which will be discussed more below.

2.2 Linearization

Figure 1 shows an extension to the P-controller with feed forward (`PFF`). Another extension may add, for example, gain scheduling (`PGS`). To get both feed forward and gain scheduling, we can create a fourth type `PFFGS` that extends both `PFF` and `PGS`. When combining several supertypes, the supertypes may intercept the same connection, and we need a way to resolve such conflicts and combine those interceptions. We do this by defining a linearization of a diagram type, which decides in what order interceptions are added in the subtype. The linearization corresponds to a depth-first search in the type hierarchy, starting at the node to be linearized, where successors are visited from left to right according to the `extends` clause.

This combination problem is an instance of the well-known diamond problem in multiple inheritance, and is illustrated in Figure 3. The type `Z` inherits from two other types, `Y1` and `Y2`, that both inherit from a common supertype `X`, and both `Y1` and `Y2` intercept the same connection in `X`. The order in the `extends` clause in `Z` determines that the interception in `Y1` is applied before the interception in `Y2` resulting in the linearization $\mathrm{L}(\mathtt{Z}) = \mathtt{X}, \mathtt{Y1}, \mathtt{Y2}, \mathtt{Z}$. We use this linearization to

determine in which order the interceptions are added. First, all blocks and connections are computed, both those declared locally and those inherited from supertypes. Then the connections are replaced according to the interceptions in the linearization order. This means that the interception in `Y1` is performed first, and then the interception in `Y2` is performed. That is, first the connection (in, out) is replaced with the connections $(in, c.in)$ and $(c.out, out)$, and then the connection $(c.out, out)$ is replaced with the connections $(c.out, d.in)$ and $(d.out, out)$. The resulting connections are $(in, c.in)$, $(c.out, d.in)$ and $(d.out, out)$, and we can see that block `c` is before block `d`.

Formally, the type hierarchy for a diagram type d with acyclic supertypes $b_1, b_2, ..., b_n$ is linearized as the following sequence:

$$\mathrm{L}(d) = \mathrm{L}(b_1) \oplus \mathrm{L}(b_2) \oplus ... \oplus \mathrm{L}(b_n)\ ,\ d$$

where

$$A \oplus (b, B) = \begin{cases} (A, b) \oplus B & \text{if } b \notin A \\ A \oplus B & \text{if } b \in A \end{cases}$$

$$A = a_1, ..., a_n \text{ and } B = b_1, ..., b_m$$

The operator $\oplus$ combines two sequences. It removes duplicates and favors elements on the left-hand side.

The linearization order we have chosen is just one of many possible orders. We think it is the most natural one, but we have no solid empirical evidence for this claim. The order has been inspired by the language Scala, but with the order reversed. In object-oriented languages, the linearization is used for selecting the method implementation to invoke when there are several possibilities. In contrast, for our visual language *all* interceptions are added, and the linearization order only decides in what order this is done.

3 Implementation

We have implemented a compiler and a visual editor as a prototype for the intercept mechanism. The compiler generates executable C programs. The textual representation, as illustrated in Figure 2, is used as the serialization format for both the compiler and the visual editor. The semantics of the language is implemented using the metacompilation system JastAdd [EH07b]. JastAdd supports the semantic formalism *reference attribute grammars* (RAGs) [Hed00] that extends *attribute grammars* [Knu68] with reference attributes. Using RAGs, the semantics can be implemented once and be reused and extended by different tools, in this case the compiler and the visual editor. In both tools, the textual representation is parsed into an abstract syntax tree (AST) and attribute values of the AST nodes are then computed automatically by JastAdd, as defined in the RAG. Reference

Table 1: Code cost

	Intercept	**Copy-modify**	**Param-**`PFF`	**Param-**`PFFGS`
`P`	11	11		
`PFF`	8	19	23	
`PGS`	8	19		
`PFFGS`	0	27		31
Total	***27***	***76***		

attributes allow an attribute value to be a reference to another AST node, meaning that graphs can be super-imposed on the AST. This makes RAGs suitable for modeling graph-like properties like name and type bindings. Other attributes represent the resulting connections and blocks in a diagram, taking into account inheritance and interception. The *computed visual elements* correspond to attributes and are shown in the visual editor.

When adding a new block in the visual editor, the name is either given explicitly by the user, or it is generated by the editor, based on the block type and a sequence number local to the block. To avoid name conflicts in subtypes (in the textual syntax), the access to an inherited block is always qualified by the supertype. For example, if a block `T_1` has been added to both `Y1` and `Y2`, the subclass `Z` can access these blocks by `Y1:T_1` and `Y2:T_1` respectively.

4 Evaluation

Table 1 compares the three approaches of interception, copy-modify, and parameterization, with respect to code cost. We compute an abstract code cost per diagram type as the sum of all local elements (parameters, blocks, connections, interceptions). For the interception approach, the table shows the cost of the types `P` and `PFF`, according to the diagrams shown in Figure 1, and for `PGS` and `PFFGS` (diagrams not shown). `PGS` extends `P` with gain scheduling. `PFFGS` extends `PFF` and `PGS`, but without adding any further elements. The diagram types for the copy-modify approach are similar to those for interception, but without the use of inheritance, that is, the elements are manually copied between diagram types, leading to code duplication and thus higher code costs.

For the parameterization approach, two different cases are shown. In **Param-**`PFF`, the diagram type models a P controller with or without feedforward, using an extra input parameter `EnableFF` to select the variant, see Figure 4. In **Param-**`PFFGS`, a more complex controller is modelled, allowing both feedforward and gain scheduling to be individually enabled or disabled (diagram not shown). We

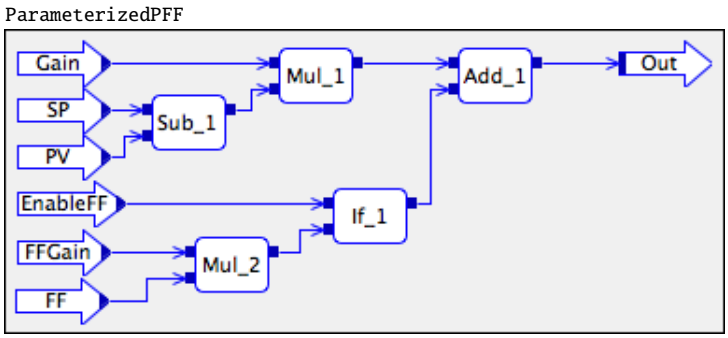


Figure 4: Using the parameterization approach for PFF. The boolean parameter EnableFF is used for enabling or disabling the feedforward functionality.

Table 2: Runtime cost

	Intercept	**Copy-modify**	**Param-PFF**	**Param-PFFGS**
P	11	11	23	31
PFF	19	19	23	31
PGS	19	19		31
PFFGS	27	27		31

can see that the code cost for the intercept solution is the lowest (27). We also see that the code cost goes up rapidly for the copy-modify approach, when the number of variants increases (76). For the parameterized approach, we see that all interesting variants need to be anticipated in advance. Thus, when a simple P controller is needed, we need to instantiate a more complex one and disable the undesired functionality.

In Table 2 an abstract runtime cost is shown for instances of type P, PFF, PGS, and PFFGS. The runtime cost is computed by summing all the instance's elements (parameters, blocks, connections, interceptions), including the inherited ones. We can see that the runtime cost increases with the number of variants for the parameterization approach, and a simple P has a runtime cost of 31 in the PFFGS parameterized case, compared to 11 for the intercept and copy-modify approaches.

The tables illustrate the drawbacks of copy-modify and parameterization. Copy-modify leads to code duplication, and parameterization leads to extra runtime costs and requires that all interesting variants are anticipated. The interception approach avoids these drawbacks. It gives both the lowest code cost and the lowest runtime cost, and furthermore enables extensible libraries. A possible drawback of the interception approach is that the language becomes more complicated, affecting both the language user and the language developer.

5 Related Work

The focus of our work is on visual dataflow-based languages, where data is moved along a connection from the producer to the consumer.

Similar work has been done for Simulink [KM07], where connections can be intercepted. However, only single inheritance is supported and corresponding blocks are instantiated by the editor when they are created. The latter means that if a block type change, all blocks of that type need to be updated, which is done manually using an editor command.

Modelica is a language for modeling and simulating complex systems [Mod16]. The language is equation-based and object-oriented. Systems are described by differential equations and connections correspond to equations rather than dataflow. Modelica supports multiple inheritance and *redeclaration* that allows a block to be replaced in a subtype, where the type of the replacing block must be a subtype of the original block's type. Redeclaration is complementary to interception, and would be interesting to add to our dataflow-based language.

Ptolemy II is an actor-based language that supports inheritance [Lee+09], where language constructs that correspond to blocks and connections can be added in subtypes. The language also supports ports with multiple incoming connections. Ptolemy II has, however, no support for connection interception and cannot handle the example shown in Figure 1.

There has also been work on inheritance for languages based on control-flow graphs, such as Activity diagrams [SP05] and Statecharts [Sim+02]. In these languages, connections correspond to state transitions rather than dataflow. However, interception maintains flow of data to in-ports for all subtypes, which is not relevant for control-flow graphs.

Parameterization in this paper refers to using input parameters for variability. Another approach is to have language support for parameterization, for example preprocessed annotations, in order to statically determine which variants to use [CA05]. This approach can lower the runtime overhead of the parameterized approach, but still requires that all variants are anticipated in advance.

6 Conclusion

We have introduced the concept of *connection interception* based on inheritance to support variability in dataflow-based languages. Connection interception allows a subtype to replace a connection declared in a supertype with two other connections. The specialized behavior maintains the invariant that the original target port still has incoming data. This mechanism has low runtime costs, it avoids code duplication and it enables extensible libraries. We have shown how inheritance and connection interception can be visualized, and how conflicts due to multiple inheritance can be handled. Furthermore, we have constructed an editor and compiler for a prototype language supporting these concepts, to illustrate the ideas.

Acknowledgment

We would like to thank Ulf Hagberg, Christina Persson and Stefan Sällberg at ABB. This work was partly financed by the Swedish Research Council under grant 621-2012-4727.

References

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. “Mapping Features to Models: A Template Approach Based on Superimposed Variants”. In: *GPCE*. LNCS. 2005, pp. 422–437.

[EH07b] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)*. 24(3). 2000, pp. 301–317.

[KM07] Paul Kinnucan and Pieter J Mosterman. “A graphical variant approach to object-oriented modeling of dynamic systems”. In: *Proc. Summer Computer Simulation Conference*. 2007, pp. 513–521.

[Knu68] Donald E. Knuth. “Semantics of Context-free Languages”. In: *Math. Sys. Theory* 2.2 (1968). Correction: *Math. Sys. Theory* 5(1):95–96, 1971, pp. 127–145.

[Lee+09] Edward A. Lee, Xiaojun Liu, and Stephen Neuendorffer. “Classes and Inheritance in Actor-oriented Design”. In: *ACM Trans. Embed. Comput. Syst.* 8.4 (July 2009), 29:1–29:26.

[Mod16] Modelica. *The Modelica Association*. `http://www.modelica.org`. 2016.

[Moo09] Daniel L. Moody. “The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering”. In: *IEEE Trans. Software Eng.* 35.6 (2009), pp. 756–779.

[SP05] Arnd Schnieders and Frank Puhlmann. “Activity Diagram Inheritance”. In: *Proc. 8th Int. Conf. Business Information Systems*. 2005.

[Sim+02] AJH Simons et al. “Plug and play safely: Rules for behavioural compatibility”. In: *Proc. 6th IASTED Int. Conf. Software Engineering and Applications*. 2002, pp. 263–268.

Visual Instance Inlining and Specialization – Building Domain-Specific Diagrams from Reusable Types

Abstract

Block diagram languages are often used for physical modeling and automation, where end users are domain engineers that instantiate block types and wire them together. Abstraction mechanisms in these languages allow specialists to build advanced reusable domain-specific libraries. However, this brings a tension between supporting reusability and making the language simple to comprehend for domain engineers. We propose a technique for relieving this tension by supporting visual instance inlining and smart editing mechanisms based on instance specialization. The new technique allows end users to visually edit and comprehend diagrams in terms of domain blocks only, while the underlying program makes use of reusable library types.

1 Introduction

Block diagrams are used in many domains, such as modeling, simulation and automation. A block diagram consists of blocks and connections between the blocks that model the data flow. Example languages include proprietary ones, such as LabView from National Instruments and ControlBuilder from ABB, and community-developed languages, such as Modelica and SysML. Many of these

Niklas Fors and Görel 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. ACM, 2016, 4:1-4:10.

languages support *user-defined types*, where a block can be defined by another block diagram, making it possible to create hierarchical structures. These hierarchies allow users to manage complexity by organizing programs at different levels of detail, making each abstraction level easier to comprehend [Moo09]. User-defined types also allow commonly used functionality to be factored out and reused many times, making it possible to create libraries that capture common functionality.

However, these two objectives, comprehension and reusability, may sometimes be conflicting. Typically, the end user of the language is a domain engineer who does not necessarily have much knowledge of programming abstraction mechanisms. He/she wishes to interact only with blocks that make sense in terms of the domain, typically corresponding to physical components such as valves, motors, controllers, and the like, and does not wish to see blocks whose existence is only motivated by reuse. At the same time, it is very important that specialists can construct reusable block libraries for different domains, to allow engineers to efficiently build working models. Furthermore, it is often the case that local libraries are built and modified for a particular application, such as the automation of a particular factory plant. Copying and changing sample models is therefore not a good alternative, since modifications to the local libraries would then not have any effect on existing diagrams. We have previously experienced this reuse/comprehensibility conflict when we worked on supporting variability in block diagrams.

In this paper, we introduce a technique for resolving this conflict between domain comprehensibility and the need for reusable libraries. In particular, we introduce a mechanism for *visual instance inlining*, and smart editing mechanisms based on instance specialization that allow connections in such visually inlined diagrams to be edited.

A visually inlined block is visually replaced by the content of the block, in a transitive manner. This allows the user to view and edit diagrams without having to understand the types introduced for reusability, and thereby increase diagram comprehension. Note that this kind of inlining is about visual appearance, rather than execution performance. The proposed inlining mechanism is different from a Fisheye view for hierarchical models [Fur86; Fre+14; Rei+08; Rei+07], which creates a bounding box around the inlined blocks. Instead, we allow the inlined blocks to be moved around in the diagram without any restrictions.

Visual inlining does not lead to copying code in the underlying program. Instead, the underlying program retains its structure with reuse-motivated blocks. Thus, changes to library types will have effect on all existing programs that use them, whether the instances are inlined or not, which is important for code maintenance.

In the underlying program, connections go only between blocks at the same level, i.e., between blocks that are siblings in the code. To allow the user to create connections between any blocks in the visually inlined view, we introduce smart editing mechanisms that automatically introduce ports on intermediate blocks as

needed.

Our underlying diagram language makes use of advanced abstraction mechanisms for diagrams, with inheritance, anonymous subtypes, interception of connections [FH14], and block redeclaration [Mod10]. Inheritance allows a diagram type S to be extended to another diagram subtype T, meaning that all blocks and connections in S are implicitly copied to T. Such a subtype can be anonymous, given at the declaration of a block. *Interception* allows a connection in S to be intercepted in T, redirecting the dataflow through a subnet of blocks in T. *Block redeclaration* allows T to specialize blocks declared in S.

We use these mechanisms to represent connections that the user creates between blocks at different levels, creating anonymous subtypes for the involved blocks.

The new language mechanisms have been developed in collaboration with the ABB company[1]. The goal of the collaboration has been to improve reusability for block diagram languages in the domain of process industry automation. We have implemented the proposed language mechanisms in a prototype language called *Bloqqi*. The language has both a textual and a visual syntax, and we have implemented both a compiler and a visual editor for the language. The compiler generates C code that can be compiled to executable programs for controlling industrial processes.

The outline of this paper is the following. We will first describe the Bloqqi language including a motivating example (Section 2), and then make the following contributions:

- **Visual instance inlining.** A mechanism that visually inlines instances of reusable block types to increase diagram comprehension. (Section 3)
- **Smart connections.** Smart editing support that allows the user to connect to the content of the inlined block by automatically specializing the block in the underlying program. (Section 4)
- **Implementation.** We have implemented compilation and editor support for the new mechanisms using the metacompilation system JastAdd [EH07b], which supports *Reference attribute grammars* [Hed00], and we describe the implementation techniques used. (Section 5)

We end the paper by discussing related work (Section 6) and wrap up with a conclusion (Section 7).

2 The Bloqqi Language

We have developed a visual data-flow language called *Bloqqi* to experiment with language constructs for building control systems. The focus has been on reusabil-

[1] `http://www.abb.com`

```
Initialize memory
for each clock tick do
  Read inputs
  Compute outputs
  Update memory
end
```

Figure 1: The execution scheme for programs in Bloqqi.

ity and supporting variability for such systems. The constructs in Bloqqi are, however, not limited to the control languages, but could be applied for other data-flow languages as well.

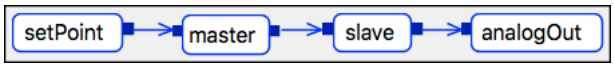
Bloqqi programs are executed periodically, for example 10 times per second. This execution scheme is called *sample-driven* execution, see Figure 1 [Ben+03]. In each period, sensor values are read (inputs), and which are used to compute control signals (outputs) that are sent to actuators. As can be seen in Figure 1, diagrams may also have states (memory) that are stored between the periods. As an example, a PID controller[2] may use the error (the difference between the set point and the process value) from previous periods to improve the control signal.

Bloqqi has been developed in collaboration with the company ABB, with monthly meetings over several years. ABB has their own visual data-flow language, ControlBuilder, that is used to build control systems for the process industry. Bloqqi has been influenced by ControlBuilder, and uses a similar sample-driven execution scheme, and similar visual data-flow diagrams. In contrast to ControlBuilder, Bloqqi uses diagram inheritance and subtyping. For this part, Bloqqi is influenced by Modelica, and uses a redeclaration mechanism similar to that in Modelica [Mod10]. However, Modelica is different in many other aspects. It uses equational diagrams with undirected connections, rather than data-flow diagrams, and it supports differential equations and is primarily used for simulation of physical processes rather than for controlling them.

2.1 Motivating Example

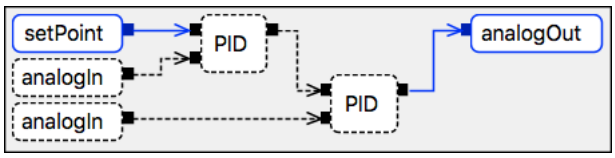
We will now motivate visual instance inlining with a simple example from the control domain. A typical controller solution is to connect two PID controllers in a master-slave configuration, where the control signal from the master is used as the set point for the slave. This way, better control performance can be achieved in many situations [ÅH06]. Figure 2 shows an example diagram for such a setup. To model this, two instances of a reusable type `ControllerPart` is used: a `master` and a `slave`. However, the `ControllerPart` type is an abstraction introduced just to support reusability, and the control engineer would rather like to see the

[2] proportional–integral–derivative controller




```
diagramtype Cascade {
  setPoint: OperatorValue;
  master: ControllerPart;
  slave: ControllerPart;
  analogOut: AnalogOut;
  connect(setPoint.out, master.SP);
  connect(master.CV, slave.SP);
  connect(slave.CV, analogOut.in);
}
```

Figure 2: Visual and textual representation for a simple cascade control loop with a master and a slave controller. The set point is set by the operator.



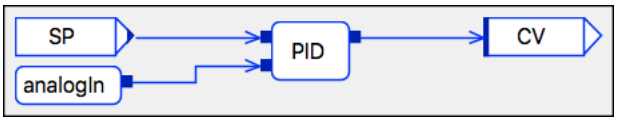

```
diagramtype Cascade {
  ...
  inline master: ControllerPart;
  inline slave: ControllerPart;
  ...
}
```

Figure 3: Inlining the controller blocks in Figure 2. The analog input blocks are moved to the left part of the diagram.

diagram shown in Figure 3, where the two `ControllerPart` instances (blocks) have been visually inlined, showing also their analog input parameters.

Note that inlining *all* blocks is seldom desired. For example, it would typically not be relevant for the control engineer to see the inner structure of the PID block, in case it also has a sub structure of blocks. It is therefore important to support selective inlining.

In Bloqqi, each visual diagram has an underlying textual program from which the diagram is rendered, and which is used when serializing diagrams. When the user opens a file in the editor, the file is parsed into an abstract syntax tree (AST). The AST is decorated with computed attributes that, for example, take into account visual instance inlining. This structure is then used to render the diagram. When the user makes a change in the diagram, the editor changes the AST accordingly and the diagram is updated. The user can serialize the AST back to the file by performing a save operation.




```
diagramtype ControllerPart(SP: Int => CV: Int) {
  analogIn: AnalogIn;
  PID: PID;
  connect(SP, PID.SP);
  connect(analogIn.out, PID.PV);
  connect(PID.CV, CV);
}
```

Figure 4: Consists of one input parameter `SP` (set point), two blocks, and one output parameter `CV` (control value).

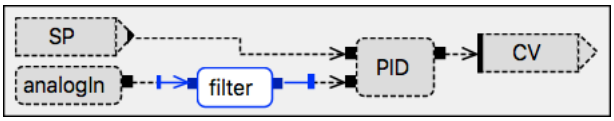
To support selective visual inlining, blocks can be declared with the modifier `inline`, as shown in Figure 3. In the diagram view of this code, the `master` and `slave` blocks have been replaced with the content of the type `ControllerPart`, that is, with the two blocks `analogIn` and `PID`. This way, the `analogIn` blocks are both at the same visual level as the `setPoint` block, and can be visually aligned with it, just like in a diagram that a control engineer would draw by hand.

The inlined blocks and connections, depicted by dashed lines in the editor, can be moved around, but can otherwise not be modified. Editing the blue parts of the diagram results in the underlying textual code to be updated (and vice versa—editing the text causes the diagram to be updated).

The definition of `ControllerPart` is shown in Figure 4. This diagram consists of both parameters and blocks. The parameters are shown as ports when the diagram is instantiated as a block, as can be seen on the `master` and `slave` blocks in Figure 2. Note that when the blocks are inlined, like in Figure 3, the parameters are removed and their connections are unified with the corresponding connections at the outer level.

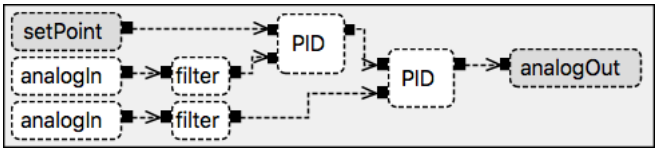
2.2 Inheritance

Bloqqi is based on inheritance, where a diagram T can be *extend* another diagram S. This means that all parameters, blocks and connections in S are also in T (transitively). We say that T is a subtype of S and S is a supertype of T. In ordinary object-oriented languages, a subtype can override a method defined in a supertype. However, since Bloqqi does not have methods, other inheritance mechanisms are available: a subtype can *intercept connections* defined in a supertype [FH14] and *redeclare blocks* defined in a supertype [Mod10]. Later in this paper, we will describe how the user can connect to inlined blocks, and this is based on inheritance.




```
diagramtype FilterControllerPart extends ControllerPart {
  filter: Filter;
  intercept PID.PV with filter.in, filter.out;
}
```

Figure 5: Extends diagram `ControllerPart`. Intercepts the connection to the block `PID` with a block `filter`.




```
diagramtype FilterCascade extends Cascade {
  redeclare master: FilterControllerPart;
  redeclare slave: FilterControllerPart;
}
```

Figure 6: Extends diagram `Cascade`. Redeclares the blocks `master` and `slave`, resulting in extra `filter` blocks.

As an example of inheritance, consider the diagram `FilterControllerPart` in Figure 5 that extends the diagram `ControllerPart`. The new diagram adds noise filtering on the sensor value. This is achieved by intercepting the connection to the second port on the `PID` block, meaning that the value from `analogIn` will now go through the filter before reaching the `PID` block. Inherited blocks are depicted as grey in the visual syntax. Like inlined blocks, they can be moved around, but not modified.

Block redeclaration is illustrated in Figure 6. Here, `FilterCascade` extends `Cascade` and specializes the blocks `master` and `slave` by redeclaring them to the type `FilterControllerPart`, adding noise filtering to the sensors in both the master and slave. A block defined in a supertype can be redeclared in a subtype. The requirement is that the new block type is a subtype of the old block type, and which is true for this example. In the visual representation, we can see that two `filter` blocks are added, since the new block type also contains one extra `filter` block.

In the previous example in Figure 6, the blocks `master` and `slave` were redeclared to an existing named subtype. It is also possible to define *anonymous subtypes* and to redeclare blocks to anonymous subtypes. Instead of redeclaring the

```
diagramtype FilterCascade2 extends Cascade {
  redeclare master: ControllerPart {
    filter: Filter;
    intercept PID.PV with filter.in, filter.out;
  };
  redeclare slave: ControllerPart {
    filter: Filter;
    intercept PID.PV with filter.in, filter.out;
  };
}
```

Figure 7: Redeclares `master` and `slave` to anonymous subtypes of `ControllerPart` (highlighted), in contrast to a named subtype as in Figure 6. The two diagrams are behaviourally equivalent.

blocks `master` and `slave` to `FilterControllerPart`, we can redeclare them to anonymous subtypes of `ControllerPart` and add the filter in the anonymous subtype. This is done in Figure 7 and this diagram is behaviourally equivalent to the diagram in Figure 6. In this example, only one block and one interception is added, but it is also possible to add parameters in an anonymous subtype, which we will use later in this paper to allow the user to connect to the content of inlined blocks.

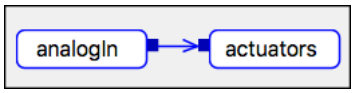
3 Visual Instance Inlining

We will now describe visual instance inlining in more detail. The rationale for visual instance inlining is that some parts of the type structure may be created to support reusability, and is not optimal for diagram comprehension. The goal of visual instance inlining is to improve diagram comprehension and at the same time retain the type structure, and thus retaining the reusability aspect.

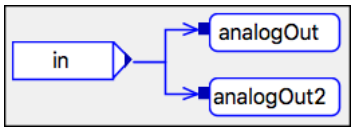
Visual instance inlining means that an instance (block) is replaced by its contents in the editor view. This brings a number of issues to resolve: How should connections to inlined blocks be handled? What kind of interaction is allowed with the inlined block content? How should the inlined block components be named? Can the content of an inlined block be transitively inlined? Are there restrictions on what can be inlined? We will now deal with these questions.

3.1 Connections to Inlined Blocks

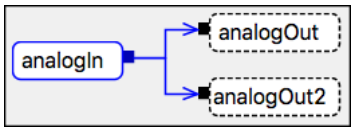
The content of a block is defined by its diagram type. As mentioned earlier, parameters in the diagram type are displayed as ports on the block. When inlining the block, the ports/parameters are removed, and if there is a connection to the port




```
diagramtype Main {
  analogIn: AnalogIn;
  actuators: Actuators;
  connect(analogIn.out,
          actuators.in);
}
```




```
diagramtype Actuators(in: Int){
  analogOut: AnalogOut;
  analogOut2: AnalogOut;
  connect(in, analogOut.in);
  connect(in, analogOut2.in);
}
```




```
diagramtype Main {
  analogIn: AnalogIn;
  inline actuators: Actuators;
  connect(analogIn.out,
          actuators.in);
}
```

Figure 8: The diagram `Main` consists of two blocks, `analogIn` and `actuators`. When the block `actuators` in `Main` is inlined (highlighted), there will be two connections from the block `analogIn`.

and a connection from its corresponding parameter inside the inlined block, these two connections are replaced by one connection.

An example of this can be seen in Figure 3. Here, the connection from `setPoint` to `master` in `Cascade` before inlining (Figure 2) and the connection from `SP` to `PID` in `ControllerPart` (Figure 4) is replaced with one connection from `setPoint` to `PID`, when the block `master` is inlined in diagram `Cascade`.

An input parameter may, in general, have n outgoing connections ($n \geq 0$). If there is a connection to the corresponding inport port on the block, then the connection to the port will be replaced with n connections when the block is inlined. This is illustrated in Figure 8, where the diagram `Main` consists of two blocks, `analogIn` and `actuators`. The diagram of the latter block consists in turn of two actuators, `analogOut` and `analogOut2`. There is also one connection from the block `analogIn` to `actuators` in `Main`. If we inline the block `actuators`, then there will be two connections from `analogIn`, one connection to each `AnalogOut` block. Output parameters/ports are handled analogously.

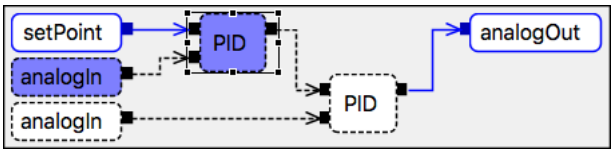


Figure 9: Selecting an inlined blocks highlights the blocks that have been inlined by the same block.

3.2 Interacting with Inlined Blocks

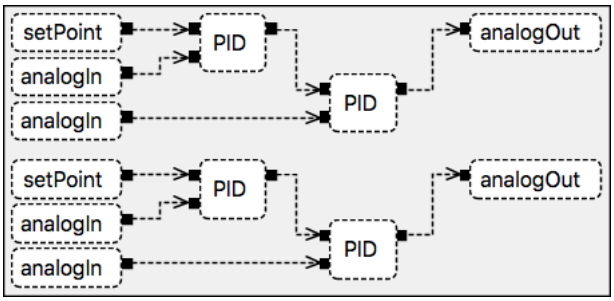
All inlined blocks are depicted as white rectangles with dashed borders in the visual syntax. The inlined blocks can be moved freely without any restrictions as illustrated in Figure 3. However, the content of the inlined blocks (connections and blocks) cannot be removed or modified, since this would correspond to editing the type of the inlined block. If such changes are desired, the user would have to bring up that type into the editor.

It is, however, possible to interact with the complete inlined block, to remove it or to make it not inlined, i.e., replacing the inlined content with the block with ports. Such interaction corresponds to editing the diagram, for example removing the `master` block from the `Cascade` diagram, or removing the `inline` modifier on the block. It is also possible to add connections to the inner blocks that come into view when a block is inlined, as will be discussed in Section 4.

3.3 Naming Inlined Content

When a block is inlined, the editor will show the local names of the inner blocks by default. For example, in Figure 3, the names that are displayed are `analogIn` and `PID`, which are not unique names in the context of `Main`. A control engineer would normally prefer this default view, since it is apparent from the connection structure which block is which, and more detailed names would take up precious screen space and clutter the view. However, if the user desires, the editor can also show unique names that are inferred by how the blocks are inlined. For example, the full name of the blocks in `Main` are `master$analogIn`, `master$PID`, `slave$analogIn` and `slave$PID`.

As an additional interactive aid, to differentiate content coming from different inlined blocks, the user can select a block in the editor and the editor will then highlight all blocks that originate from the same inlined block. This is shown in Figure 9, where one of the `PID` blocks has been selected. The editor has visually highlighted both the `PID` and its related `analogIn`, since both belong to the same inlined block (`master`).




```
diagramtype TwoCascades {
  inline cascade: Cascade;
  inline cascade2: Cascade;
}
```

Figure 10: Transitive inlining. This diagram consists of two blocks of the type `Cascade` (Figure 3) and which are inlined. Since `Cascade` also contains inlined blocks, the blocks `analogIn` and `PID` are inlined in two steps.

3.4 Transitivity

If a block is marked using the modifier `inline`, this means that it is has been decided that its contents are more meaningful to see than the block itself. It is therefore natural that visual inlining is transitive. Thus, if a block is inlined and the content of the block also contains blocks that are inlined, then blocks are inlined in more than one step. This is shown in diagram `TwoCascades` in Figure 10, where two blocks of the diagram `Cascade` (Figure 3) are inlined. The diagram `Cascade` in turn inlines the blocks `master` and `slave`, resulting in that the blocks `analogIn` and `PID` in `TwoCascades` are inlined in two steps, thus, illustrating that inlining is transitive.

There are, however, two cases when a block marked as `inline` is actually not visually inlined. The first is when the block does not contain any inner blocks. Visually inlining it would then result in the block being removed altogether from view. The other case is when a diagram type T contains (directly or transitively) a block of the same type T. This is a compile time error, and is detected by the compiler as such. In this case, visual inlining is inhibited since it would result in an endless recursion.

3.5 Alternative Solution: Bounding Box

As explained earlier, the content of a visually inlined block can be moved around freely. An alternative solution would be to have a bounding box around the block that is inlined. This is a common way of supporting visual inlining in interactive editors (for example, see [Sch+14]) and is illustrated in Figure 11. Here, both the

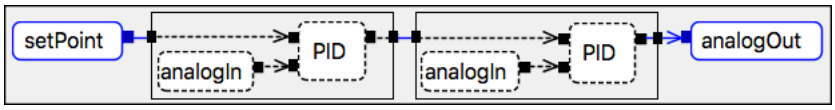


Figure 11: Bounding box, an alternative inlining technqiue. Compare to Figure 3.

blocks `master` and `slave` have a bounding box around the inlined content. This kind of visual inline can be useful to be able to see many levels at the same time, and thus has another purpose than our inlining. Our purpose is that the control engineer should be able to see diagrams that look just like the ones usually drawn in that domain, e.g., like the ones shown in control textbooks. Furthermore, often, the end diagrams are printed and used for reference for a running plant, and it is important to be able to make full use of all available layout space. Being able to move the inlined content in an unrestricted way is therefore very important. An example was shown in Figure 3 where the `analogIn` blocks were visually aligned with the `setPoint` block.

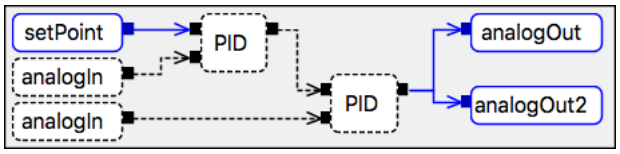
4 Instance Specialization

When the user edits a diagram with a visually inlined block, he/she may wish to create connections to *content blocks*, i.e. to blocks in the inlined content. However, in order to not break encapsulation, such connections must go via parameters of the inlined block.

We solve this by *instance specialization*: An instance can be specialized by adding an anonymous subtype to its declaration, and adding parameters and connections to that subtype. The editor automatically adds the subtype, parameters, and connections, as the user creates cross-level connections to content blocks. When the user modifies these connections, the anonymous subtype is modified accordingly. This way, visual edits in the diagram are represented by local edits in the corresponding diagram code. We illustrate the mechanism through a series of increasingly complex examples.

4.1 Existing Parameters

The simplest case for connecting to the content of an inlined block is when the content is already accessible through existing parameters. As an example, consider the diagram `Cascade` in Figure 3. For this diagram, the user may, for example, want to use the value from `PID` block that orignates from the `slave` block to a second actuator. To do this, the user first adds the block `analogOut2` representing the actuator and then creates a connection from the output port on the `PID` block to the newly created block. This can be seen in Figure 12. When the user adds the connection, the editor detects that the output from the `PID` block in the diagram `ControllerPart` (Figure 4) is connected to the output parameter `CV`, and




```
diagramtype Cascade {
  ...
  analogOut2: AnalogOut;
  ...
  connect(slave.CV, analogOut2.in);
}
```

Figure 12: First, the block `analogOut2` is added to the diagram `Cascade` (Figure 3) and then the user connects the second `PID` to the newly created block. The editor infers that the output of the `PID` block is the same as the output port `CV` on the `slave` block.

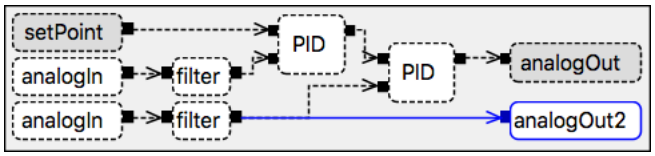
then the editor simply adds a connection from the `CV` port on the `slave` block to the newly created block. Thus, the editor adds a connection from `slave.CV` to `analogOut2.in`.

4.2 Implicitly Creating Anonymous Subtypes

A slightly more complex example is when the user would like to create a connection from a port on a content block that is not connected to any output parameter of the inlined block. In this case, the editor first creates an anonymous subtype for the inlined block, unless it already has one, and then adds parameters and connections in that subtype, in order to then be able to add the connection in the edited diagram.

For example, in diagram `FilterCascade` in Figure 6, the user may want to connect directly from the `filter` block to a second actuator `analogOut2`. This is shown in Figure 13. Since the output from the `filter` block is not directly accessible, an anonymous subtype of `FilterControllerPart` is automatically created by the editor. The anonymous subtype makes the value from the `filter` block accessible by adding an output parameter. Thus, the anonymous subtype contains one output parameter `filterout` and one connection from the `filter` block to the new output parameter. Then, a connection is created in `FilterCascade`: from the newly created parameter `filterout` on the `slave` block to the `analogOut2` block.

In Figure 13, the type of the `slave` block was a named type, and the editor therefore created an anonymous subtype. However, in diagram `FilterCascadeAnonymous` (Figure 7), both the blocks `master` and `slave` already have an anonymous type. If this is the case, the editor only adds parameters and connections when the user connects to the content of such a block. This is shown in Figure 14, where one parameter and one connection are added to the existing anonymous type (highlighted).

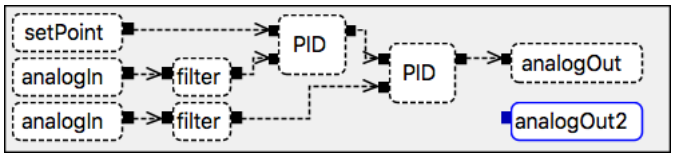



```
diagramtype FilterCascade extends Cascade {
  redeclare master: FilterControllerPart;
  redeclare slave:
    FilterControllerPart (=> filterout: Int) {
      connect(filter.out, filterout);
    };
  analogOut2: AnalogOut;
  connect(slave.filterout, analogOut2.in);
}
```

Figure 13: The user has added the blue parts of the diagram: the block `analogOut2` and the connection to it from one of the `filters`. The additions cause the editor to change the underlying code. For the new connection the editor adds (highlighted): 1) An anonymous subtype to the inlined `slave` block, with a new parameter `filterout`, and a connection to it from the `filter`. 2) A connection from `filterout` to `analogOut2`.

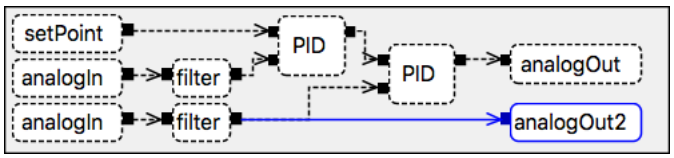
```
diagramtype FilterCascadeAnonymous extends Cascade {
  redeclare master: ControllerPart {
    filter: Filter;
    intercept PID.PV with filter.in, filter.out;
  };
  redeclare slave: ControllerPart (=> filterout: Int) {
    filter: Filter;
    intercept PID.PV with filter.in, filter.out;
    connect(filter.out, filterout);
  };
  analogOut2: AnalogOut;
  connect(slave.filterout, analogOut2.in);
}
```

Figure 14: Connecting from `filter` to `analogOut2`, as in Figure 13. In this example, however, the `slave` block already has an anonymous type, and in this case, the editor only adds one parameter and one connection to the existing anonymous type.




```
diagramtype InliningFilterCascade {
  inline filterCascade: FilterCascade;
  analogOut2: AnalogOut;
}
```

Figure 15: A diagram consisting of a block of `FilterCascade` that is inlined and an actuator block.




```
diagramtype InliningFilterCascade {
  inline filterCascade: FilterCascade (=> slaveout: Int) {
    redeclare slave: super (=> filterout: Int) {
      connect(filter.out, filterout);
    };
    connect(slave.filterout, slaveout);
  };
  analogOut2: AnalogOut;
  connect(filterCascade.slaveout, analogOut2.in);
}
```

Figure 16: Connecting the second `filter` block to the `analogOut2` block makes the editor to implicitly create two anonymous subtypes (highlighted) that connect the value from the `filter` block to the `analogOut2` block.

4.3 Transitive Inlining

As described in the previous section, inlining can be transitive. For example, in Figure 15, the diagram `InliningFilterCascade` consists of two blocks: `filterCascade` of type `FilterCascade` (Figure 6) and `analogOut2`. The block `filterCascade` is inlined, resulting in that the blocks `analogIn`, `filter` and `PID` are inlined in two steps. If the user wants to connect from the output on one of the `filter` blocks, then the editor needs to take into account the transitivity of inlining.

Figure 16 shows the result of when the second `filter` block is connected to the `analogOut2` block. We can see that two anonymous subtypes are created, one for the `filterCascade` block and one for the `slave` block residing inside the `filterCascade` block. The value from the `filter` block is connected through parameters and connections in two steps, flowing through the two anonymous types, ending as an output parameter of the `filterCascade` block.

```
diagramtype A {
  block: Block;
}
diagramtype B extends A {
  redeclare block: Block { ... };
}
diagramtype C extends B {
  redeclare block: super { ... };
}
```

Figure 17: Redeclaring the block `block` in two steps. The keyword `super` is used in `C` to redeclare the block to an anonymous subtype of the anonymous subtype declared in `B`.

4.4 Super in Redeclare

We can see in Figure 16 that the inner block `slave` is redeclared to `super { ... }`. In Bloqqi, `super` is a keyword that can be used inside the type part of a `redeclare` statement. The meaning of `super` is that an anonymous subtype will be created that is a subtype of the type for the block that is redeclared. This is needed when redeclaring an anonymous subtype to another anonymous subtype, since redeclare requires that a block is redeclared to a subtype of the type for the existing block.

The need for supporting `super` in redeclare is illustrated in Figure 17. Here, the block `block` is redeclared in two steps, first in diagram `B` and then in diagram `C`. The diagram `B` creates an anonymous subtype of `Block`, and to create an anonymous subtype of the anonymous subtype in `B`, the `super` keyword is used in diagram `C`.

4.5 Editing the anonymous type

Sometimes, it may be desirable to add local blocks and connections to an inlined block, or to redeclare an existing component to a more specialized type. This can be done by bringing up the anonymous subtype in an editor window, and add or edit the local parts there.

5 Implementation

We have implemented a compiler and a visual editor for the Bloqqi language. The semantics has been defined using the metacompilation tool JastAdd [EH07b] which is based on Reference Attribute Grammars [Hed00]. JastAdd allows the semantics to be defined once and to be used both in the compiler and the visual editor. It makes it easy to reuse the semantic specifications between several tools,

avoiding double maintenance. The interaction parts of the editor have been implemented using the Eclipse Graphical Framework (GEF). All the diagrams shown in this paper are screenshots from our tool.

The serialization format for Bloqqi is the textual syntax. The compiler reads the textual syntax and outputs C code or possible compile-time errors. The visual editor also reads the textual syntax but instead visualizes the program and lets the user interact with it and change it. When the user saves the program in the editor, the program is serialized using a pretty printer.

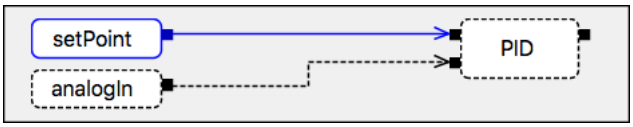
Both the compiler and the visual editor parses the textual syntax into an abstract syntax tree (AST). This tree is decorated with attributes using *Reference attribute grammars* (RAGs) [Hed00], an extension to Attribute grammars (AGs) [Knu68] that adds reference attributes. This extension makes it possible to superimpose graphs on the AST, making RAGs suitable for representing graph-based languages like Bloqqi. A typical example of reference attributes is name binding, where the tree node representing the use of a name has a reference attribute pointing to the tree node representing the declaration of the name. Attributes are attached to tree nodes and are *computed* node properties defined by equations. These equations are declarative in the sense that they are free from externally visible side effects. In effect, the equations form a complex function from the AST to the attribute values. The attributes are demand-driven, meaning that they are only computed when they are requested and cached for subsequent accesses. When a diagram is modified in the editor, these caches are then cleared.

Most of the complex computations in the compiler and editor are done using attributes. The compiler then uses attributes to show compile-time errors or to generate C code. The editor uses attributes for rendering the diagrams, for computing smart interaction support, and for showing compile-time errors. GEF is based on the Model-view-controller (MVC) pattern and we use the decorated AST as the model.

5.1 Inlined Blocks as Non-Terminal Attributes

The rendered diagram has a quite different structure from the corresponding underlying code: it shows inherited blocks and connections, and it shows the content of the inlined blocks. To compute a suitable representation of it, we use *Non-terminal attributes* (NTAs) [Vog+89]. An NTA is an attribute whose value is a subtree defined by an equation, rather than constructed by the parser like ordinary subtrees. For each node in the AST that represents a diagram, we define two NTAs: `blocks()` and `connections()` that represent its visible parts.

The `blocks()` attribute is the computed set of blocks for the diagram, taking into account inlining and inheritance. For example, consider the diagram `Simple` in Figure 18, which consists of two blocks: `setPoint` and `master`. The latter block is inlined. A simplified AST for this diagram is shown in Figure 19. We can see that the node representing the diagram `Simple` has two children, one ordinary




```
diagramtype Simple {
  setPoint: OperatorValue;
  inline master: ControllerPart;
  connect(setPoint.out, master.SP);
}
```

Figure 18: A simple diagram consisting of two blocks and one connection.

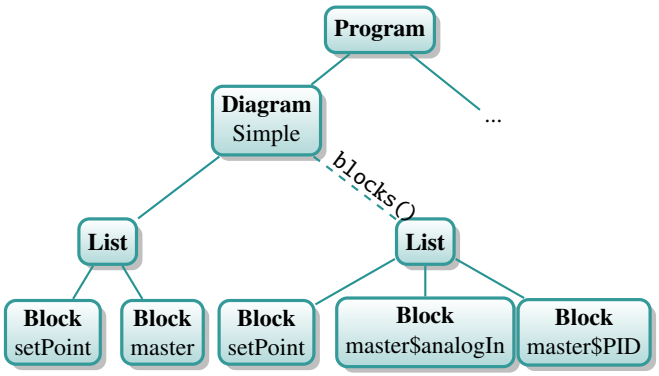


Figure 19: Simplified AST for the diagram `Simple` in Figure 18. The diagram node has an NTA `blocks()` that is the computed set of blocks considering inlining and inheritance.

child representing the list of local block declarations and the other child is the NTA `blocks()`. The NTA contains all three blocks that are seen in Figure 18. The names for the inlined blocks are prefixed with `master$` to get unique names. The attribute `blocks()` is used by the editor when a diagram is rendered.

The AST in Figure 19 is highly simplified, and the real implementation has many more children and attributes. For example, each block node has a child representing the type of the block and two NTAs representing the input and output ports. The value of these two NTAs are computed based on the parameters of the type for the block. In turn, a port node has an attribute that is the set of incoming or outgoing connections for that port, and is used by the editor for rendering the connections between the ports.

In analogy to the `blocks()` NTA of the diagram node, the `connections()` attribute of the diagram node contains a computed set of connections considering inlining and inheritance. For this example, this attribute contains two connections: one from `setPoint.out` to `master$PID.SP` and one from `master$analogIn`.

`out` to `master$PID.PV`. These connection nodes have reference attributes that point to the corresponding blocks in the NTA `blocks()` that they refer to, for example, the `master$PID` block.

In Section 4, we described how the user can connect to the content of inlined blocks. This is an example of a smart editing operation where the editor uses attributes to compute how to change the AST. This add-connection operation is broken down in several steps. The first step is adding anonymous types, the second step is adding parameters and connections, and the third step is adding the outermost connection. Steps one and two are not always needed, and depend on the program.

In each of these steps, attributes are used for computing how to change the AST. After each step, the AST is modified, and all cached attributes are then cleared. In principle, it would be possible to compute the change in one step, but this would lead to a more complex specification, since the steps are dependent on each other.

The clearing of cached attributes after each modification to the AST could lead to problems with scalability when editing large programs. However, the caching demand evaluation counters this problem, and we have run the editor with diagrams of 500 blocks and connections, and interaction is still perceived as immediate.

5.2 Layout

The user can freely move around blocks, and we store the block coordinates as annotations in the textual representation. Connections are rendered using a simple manhattan algorithm, breaking on the middle of the connection if needed, so data about connection layout does not need to be stored. Allowing more advanced connection layout would, however, yield a better layout.

For a newly created subtype or inlined block, we use automatic layout, and we also provide automatic layout as an explicit command. The tool we use for automatic layout is called KLay [Sch+14], which is specialized for block diagrams with ports on the blocks. Currently, we only make use of their automatic layout of the blocks. Allowing more advanced layout of the connections, as well as using KLay for automatic computation of such layout, is a matter of future investigation.

6 Related Work

A related concept to visual instance inlining presented in this paper is the Fisheye view [Fur86], which is a technique for showing local and global information at the same time. The user has a focus point and local information to that point is shown in greater detail compared to the global context information that is shown in less detail. The Fisheye view has been used for hierarchical models where composite

nodes can be expanded with a bounding box around the content [Fre+14; Rei+08; Rei+07], allowing the content only to be moved within the box. In contrast, visual instance inlining allows the content blocks to be moved freely without any restrictions. Also, in many of the examples that use the Fisheye view use models that are hierarchical but without any types. In constrast, we do inlining on instances of diagram types. Our approach also includes ways for allowing the user to access values from the content blocks, where the editor automatically creates anonymous subtypes, parameters and connections.

Several block diagram languages support user-defined types, and one way to generate code is to *flatten* the programs [Lub+09], that is, replacing all composite blocks (instances of user-defined types) with the content of the blocks transitively. This results in a program with only atomic blocks. Visual inlining is similar to flattening in that ports and parameters are removed. But in visual inlining, they are removed only from view, and they remain in the code. Other ways visual inlining differs from flattening is that it is specified on a subset of the blocks, and not all blocks, and that the goal is to improve diagram comprehension rather than generating code. ([Lub+09] suggests an alternative code generation approach than flattening to support separate compilation.)

Bloqqi is inspired in several ways by the language Modelica [Mod10]. However, there are also many fundamental differences between the languages. In particular, Modelica is an equation-based simulation language with undirected connections between blocks, whereas Bloqqi is a data-flow based control language, with directed connections between blocks.

The `redeclare` construct in Bloqqi has been inspired by a similar construct in Modelica, but which is written in a slightly different way, as exemplified in Figure 20. When declaring a Modelica block, like `c` in the example, the local blocks can be redeclared to be of more specific subtypes. This is called *modifications* in Modelica terminology. This can be seen as introducing an anonymous subtype to the original type of the local block, but with a different syntax.

A modification can be done in several steps at once, like in the following Modelica snippet.

```
a: A(b(c(redeclare d: DPrime)))
```

Here, the inner block `d` is redeclared. This is possible to express in Bloqqi as well, but with the introduction of several anonymous subtypes. Another difference is that Bloqqi allows new blocks and connections to be declared in the anonymous subtype, which is not allowed in Modelica.

To generate simulation code for a Modelica model, the model is typically first flattened, removing all object-oriented structures and resulting in a large set of differential equations. The Modelica compiler JModelica.org is also implemented using JastAdd, and uses NTAs to implement the flattening [Åke+10a]. Our approach of using NTAs for implementing the visual inlining has some similarities to this approach.

```
diagramtype ControlLoop {
  c: Cascade(redeclare master: FilterControllerPart,
             redeclare slave: FilterControllerPart);
  ...
}
diagramtype FilterCascade
  extends Cascade(redeclare master: FilterControllerPart,
                  redeclare slave: FilterControllerPart){
  ...
}
```

Figure 20: Redeclare in Modelica. Diagram `ControlLoop` redeclares the inner content of the block `c` of type `Cascade`. Diagram `FilterCascade` extends a subtype of `Cascade` where the blocks `master` and `slave` are redeclared.

Another inspiration from Modelica is that Bloqqi has both a textual and a visual syntax. Like in Modelica, the textual syntax is complete, whereas the visual syntax is only partial, supporting the construction of diagrams from predefined library types. For both languages, the textual syntax is typically used by specialists to create reusable libraries, whereas the visual syntax is used by domain engineers, to create concrete models. A complete textual syntax has many advantages. It allows the compiler and the semantics to be developed without the support for complex user interaction. It also makes it easy to add new test cases, just add new text files. Furthermore, it allows general purpose development tools, like version control tools, to be used. Another example of an implementation of a language with both textual and visual syntax is van Rest's work on behavior trees [Res+13].

7 Conclusions

In this paper we have introduced *visual instance inlining* with smart editing support for typed block diagrams. This technique allows reusable block libraries to be supported, while allowing domain engineers to view and edit diagrams in a natural way, in terms of the domain blocks they would draw on paper, i.e., without having to view types that only have to do with reusability.

With visual instance inlining, a block type instance is replaced visually by its content, and the content parts can be freely moved around, without being restricted by a bounding box, and connections can be added to the content parts. In the underlying program, the inlined block is represented as an anonymous subtype of the original block type. Added connections to content parts are represented by ports and connections in the anonymous subtype. Since the type structure is retained in the program, changes to the library block types will affect all block type instances, even if they are inlined and connections to the inlined parts have been added.

As a feasibility study, we have implemented a compiler and a visual editor supporting visual instance inlining and implemented smart editing support in the editor.

There are several ways to continue this work. One is to be able to define diagram *types* as `inlined`, and not only individual blocks. Each block of such a type would then be visually inlined.

Another useful thing would be to support additional context-dependent names to be set on inlined block content. Currently, the inlined parts are labelled with the local name declared in the inlined type, e.g., `PID` in our examples. A control engineer would often like to see a name corresponding to a global physical entity, e.g., `PID354`, corresponding to that particular `PID` component in a factory plant. Such names could also be represented in the anonymous subtypes of the inlined instances.

It would also be useful to introduce access restriction constructs, to be able to limit which content blocks the user can access values from.

8 Acknowledgements

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

[Åke+10a] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. “Implementation of a Modelica Compiler Using JastAdd Attribute Grammars”. In: *Science of Computer Programming* 75.1-2 (Jan. 2010), pp. 21–38.

[ÅH06] Karl Johan Åström and Tore Hägglund. *Advanced PID control*. ISA-The Instrumentation, Systems, and Automation Society; Research Triangle Park, NC 27709, 2006.

[Ben+03] Albert Benveniste et al. “The synchronous languages 12 years later”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[EH07b] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[FH14] Niklas Fors and Görel Hedin. “Intercepting dataflow connections in diagrams with inheritance”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2014, pp. 21–24.

[Fre+14] Patrick Frey et al. “Efficient Exploration of Complex Data Flow Models”. In: *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*. 2014, pp. 321–336.

[Fur86] George W. Furnas. “Generalized Fisheye Views”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Boston, Massachusetts, USA: ACM, 1986, pp. 16–23.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)*. 24(3). 2000, pp. 301–317.

[Knu68] Donald E. Knuth. “Semantics of Context-free Languages”. In: *Math. Sys. Theory* 2.2 (1968). Correction: *Math. Sys. Theory* 5(1):95–96, 1971, pp. 127–145.

[Lub+09] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. “Modular code generation from synchronous block diagrams: modularity vs. code size”. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 2009, pp. 78–89.

[Mod10] Modelica. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.2*. Available from `www.modelica.org`. Modelica Association. 2010.

[Moo09] Daniel L. Moody. “The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering”. In: *IEEE Trans. Software Eng.* 35.6 (2009), pp. 756–779.

[Rei+07] T. Reinhard, S. Meier, and M. Glinz. “An Improved Fisheye Zoom Algorithm for Visualizing and Editing Hierarchical Models”. In: *Requirements Engineering Visualization, 2007. REV 2007. Second International Workshop on*. Oct. 2007, pp. 9–9.

[Rei+08] Tobias Reinhard et al. “Tool support for the navigation in graphical models”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. 2008, pp. 823–826.

[Res+13] Oskar van Rest et al. “Robust Real-Time Synchronization between Textual and Graphical Editors”. In: *ICMT*. 2013.

[Sch+14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. “Drawing layered graphs with port constraints”. In: *Journal of Visual Languages & Computing* 25.2 (2014), pp. 89–106.

[Vog+89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *PLDI*. 1989, pp. 131–145.

Bloqqi: Modular Feature-Based Block Diagram Programming

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.

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* [Mod12], and *SysML* [Sys15]. Typically, the connections

Niklas Fors and Görel Hedin. "Bloqqi: Modular Feature-Based Block Diagram Programming". In: *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* 2016, Amsterdam, Netherlands, 30 October - 4 November, 2016. In press.

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. [ÅH06]. 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 preconfigured 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, 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,

[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.

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* [FH14] and *block redeclaration* [Mod12]. 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.

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:

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

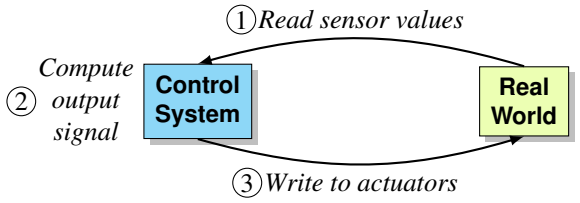


Figure 1: Periodic execution. All steps are performed in each period.

- **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., 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

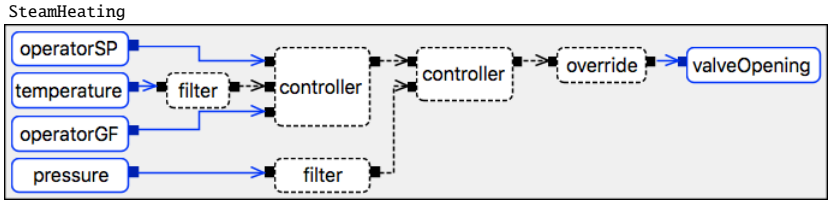


Figure 2: Diagram for temperature control using steam. Solid parts are created manually. Dashed parts via the wizard.

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 [ÅH06]. 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.

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 ap-

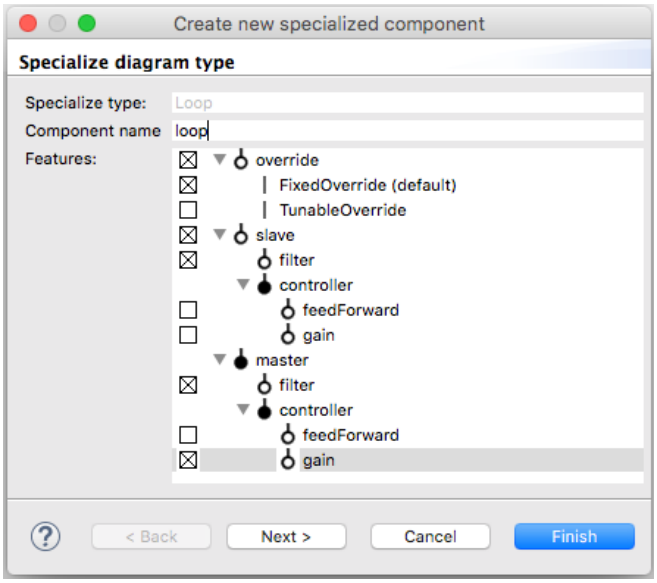


Figure 3: Wizard for creating a control loop.

propriate 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 [Kan+90]. 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.

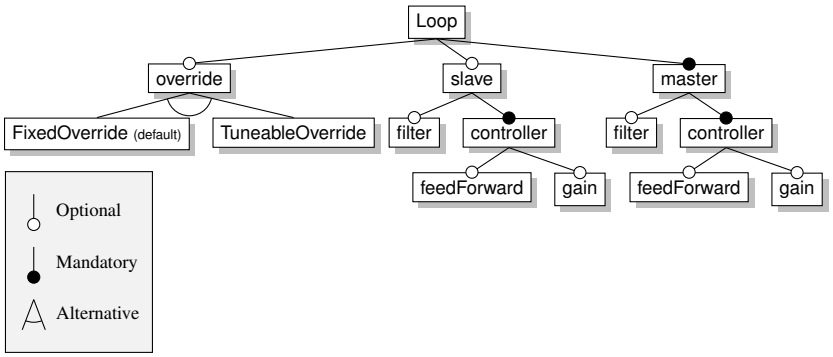


Figure 4: Corresponding feature model for the type `Loop` in Figure 3.

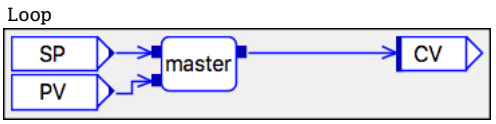
The dashed parts of the diagram is actually an instance of an anonymous subtype of the block type `Loop`, but which is automatically inlined to show interesting parts of the inner structure. For the `Loop` type, the library developer has chosen to expose the controllers and the filters. However, the 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) [FH16b].

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 [Mod12]. Bloqqi furthermore supports *connection interception*, where a connection in a supertype can be intercepted and rerouted through a block subnet [FH14].[4] Similar to Modelica, Bloqqi has both a textual syntax that covers the complete language and a visual syntax that covers block configuration.

[4] [FH14] 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`.

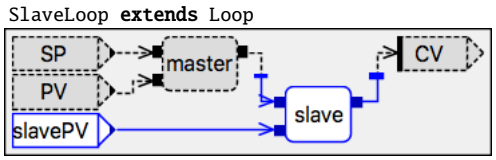
In this section, we will introduce the basic language constructs in Bloqqi, and discuss how inheritance, redeclaration, and connection interception can be used to encode a variant as a subtype. The subsequent sections will then introduce the new language mechanisms of wirings and recommendations that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

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




```
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.

from the supertypes 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* [FH14]. 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 [Mod12]. 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`.

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

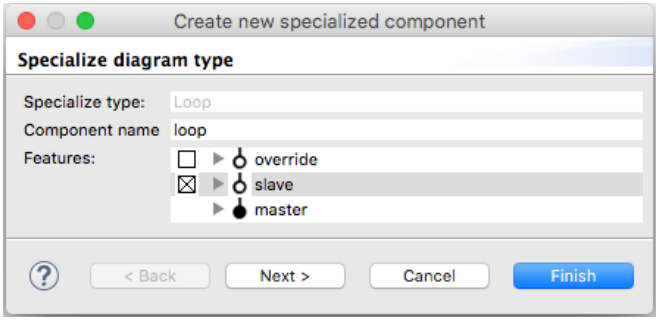


Figure 7: Automatically derived wizard for the type `Loop`, based on recommendations. Using this wizard, a new block can be created with the desired functionality.

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 gen-

eral 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.

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`.

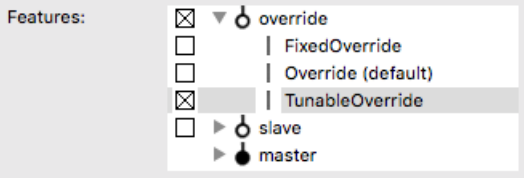


Figure 8: Recommended features for `Loop`. The feature `override` has three alternatives, with one default choice.

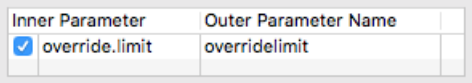


Figure 9: Unconnected inner parameters can be exposed as outer parameters.

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;
}
```

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,

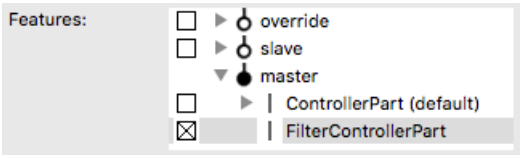


Figure 10: Wizard recommendations for `Loop`. The block `master` is `replaceable` and is specialized to an existing subtype of `ControllerPart`.

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 [Mod12], 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. 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:

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

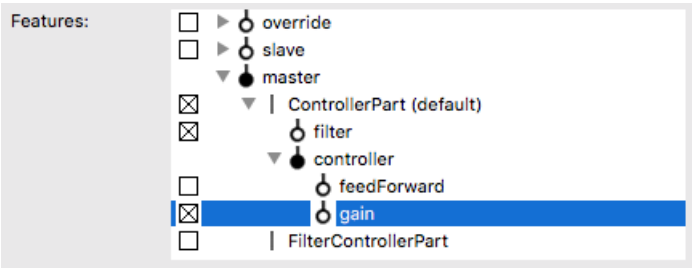


Figure 11: Wizard recommendations for `Loop`. The block `master` is specialized to a new anonymous subtype of `ControllerPart` that has support for proportional gain.

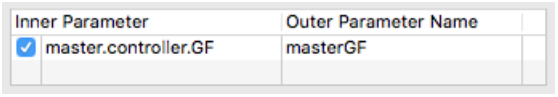


Figure 12: Exposure of inner unconnected parameter `GF`.

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*.

```
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).

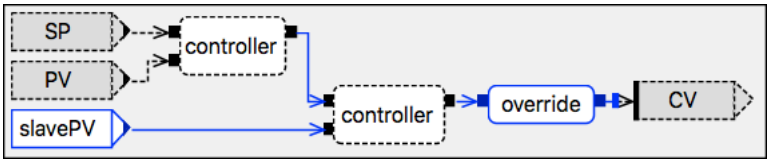


Figure 14: Specialization of `Loop`. The slave and override features intercept the same parameter (`CV`), however, the slave is applied before override, resulting in its inlined `controller` appearing before the `override` block.

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.

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

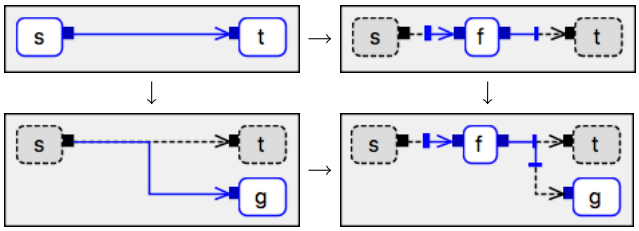


Figure 15: Base diagram extended with two features, `f` and `g`, where `f` uses *source interception* on the output port on block `s`. When the features `f` and `g` are combined, feature `g` will use the value from `f`, and not the value from block `s`.

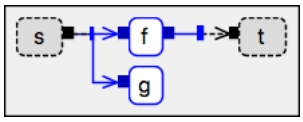


Figure 16: Same as Figure 15, but where feature `f` uses target interception instead of source interception, resulting in that feature `g` will use the value from the block `s`.

that all other components will use the 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 [FH16b]. 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`,

```
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.

`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 [Cza+04].

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 [Cza+04], 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) [Hed00], using the metacom-

pilation system JastAdd [EH07b].

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 [Blo+12]. 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 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

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

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.

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.

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.

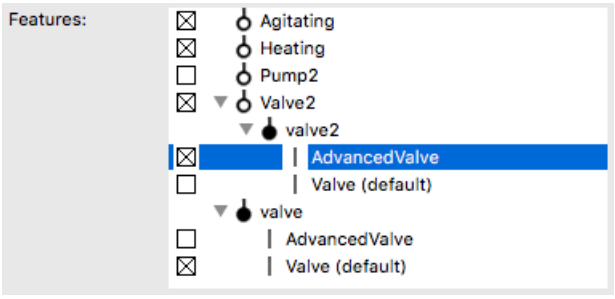
As an additional example, we have implemented a library for controlling the liquid level in a tank with a pump and a valve, and with optional features for heating, agitating, and for adding extra pumps and valves, as well as selecting between different types of valves. The heating feature may be implemented 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 [WM13]. 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.



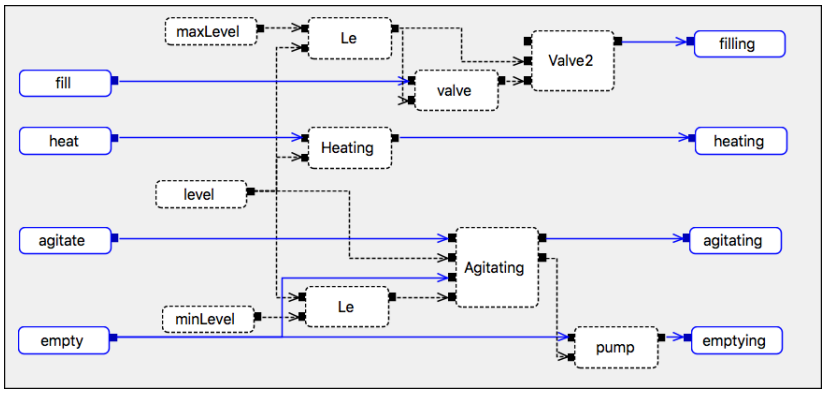


Figure 18: Tank specialized with agitation, heating, and a second valve and connected to environmental blocks (solid).

[Die+14] 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.

[Hab+13] 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 [CA05] and for SysML [Tru10], but these also require all variants to be anticipated in advance.

As mentioned, Bloqqi has been substantially influenced by Modelica [Mod12], 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 [FH14], 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 [Gam+94]. 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 [Sch+03] 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 [Kic+01] (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 [Kan+90], 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 [Cza+05], or general propositional formulas [Bat05].

We have been inspired by the feature diagram syntax, and wizards for feature diagrams [CA05], 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 [Bat05]. 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 `Fixed0verride` 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

[Pre97] 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) [AK09], 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 [Mur89] or state charts [Har87]. Within the automation domain there is, for example, the Grafchart language which focuses on supervisory control [JÅ98], 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 [PH02]. Finally, larger case studies should be performed.

Acknowledgment

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

[AK09] Sven Apel and Christian Kästner. “An Overview of Feature-Oriented Software Development”. In: *Journal of Object Technology* 8.5 (2009), pp. 49–84.

[ÅH06] Karl Johan Åström and Tore Hägglund. *Advanced PID control*. ISA-The Instrumentation, Systems, and Automation Society; Research Triangle Park, NC 27709, 2006.

[Bat05] Don Batory. “Feature Models, Grammars, and Propositional Formulas”. English. In: *Software Product Lines*. Vol. 3714. LNCS. Springer, 2005, pp. 7–20.

[Blo+12] Torsten Blochwitz et al. “Functional mockup interface 2.0: The standard for tool independent exchange of simulation models”. In: *9th International Modelica Conference*. 2012.

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. “Mapping Features to Models: A Template Approach Based on Superimposed Variants”. In: *GPCE*. LNCS. 2005, pp. 422–437.

[Cza+04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. “Staged Configuration Using Feature Models”. In: *Proc. 3rd Int. Conf. Software Product Lines*. 2004, pp. 266–283.

[Cza+05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. “Formalizing cardinality-based feature models and their specialization”. In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29.

[Die+14] Arnaud Dieumegard, Andres Toom, and Marc Pantel. “A Software Product Line Approach for Semantic Specification of Block Libraries in Dataflow Languages”. In: *Proc. 18th Int. Software Product Line Conference*. Florence, Italy: ACM, 2014, pp. 217–226.

[EH07b] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[FH14] Niklas Fors and Görel Hedin. “Intercepting dataflow connections in diagrams with inheritance”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2014, pp. 21–24.

[FH16b] Niklas Fors and Görel 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. ACM, 2016, 4:1–4:10.

[Gam+94] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[Hab+13] Arne Haber et al. “First-class Variability Modeling in Matlab/Simulink”. In: *7th Int. Workshop on Variability Modelling of Software-intensive Systems*. Pisa, Italy: ACM, 2013, 4:1–4:8.

[Har87] David Harel. “Statecharts: a visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (1987), pp. 231 –274.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)*. 24(3). 2000, pp. 301–317.

[JÅ98] Charlotta Johnsson and Karl-Erik Årzén. “Grafchart for recipe-based batch control”. In: *Computers & Chemical Engineering* 22.12 (1998), pp. 1811 –1828.

[Kan+90] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. DTIC Document, 1990.

[Kic+01] Gregor Kiczales et al. "An Overview of AspectJ". In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*. Vol. 2072. LNCS. Springer, 2001, pp. 327–353.

[Mod12] Modelica. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3*. Available from `modelica.org`. Modelica Association. 2012.

[Mur89] Tadao Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pp. 541–580.

[PH02] Lars Pernebo and Bengt Hansson. "Plug and play in control loop design". In: *Proceedings for Control Meeting*. Linköping, Sweden, 2002.

[Pre97] Christian Prehofer. "Feature-Oriented Programming: A Fresh Look at Objects". In: *ECOOP*. 1997, pp. 419–443.

[Sch+03] Nathanael Schärli et al. "Traits: Composable Units of Behaviour". In: *ECOOP*. 2003, pp. 248–274.

[Sys15] SyslML. *SysML Open Source Specification Project*. `http://sysml.org`. 2015.

[Tru10] Salvador Trujillo et al. "Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy". English. In: *Modelling Foundations and Applications*. Vol. 6138. LNCS. Springer, 2010, pp. 293–304.

[WM13] Jens Weiland and Peter Manhart. "A Classification of Modeling Variability in Simulink". In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 7:1–7:8.

Implementation of the Bloqqi compiler and editor

1 Introduction

Data-flow languages are commonly used in a variety of domains, such as automation, modeling, image processing, etc. They are used both in industry and in academia, and examples include Simulink from MathWorks, LabVIEW from National Instruments, ControlBuilder from ABB, and Function Block Diagrams in the IEC-61131 standard. Typically, programs are expressed visually as diagrams with blocks and connections between the blocks that describe the data-flow. Many data-flow languages support *user-defined block types*, where a block is an instance of a block type. A block type can be defined as another diagram, making it possible to create hierarchical programs, or it can be defined externally or by an algorithmic function. User-defined block types facilitate reuse of functionality and allow libraries to be created. They also encourage program decomposition into manageable and understandable program units.

Although user-defined block types aid the creation of libraries, they do not support creating different *variants* of the same basic structure in a good way. For example, for a controller library, the user sometimes would like the controller to have gain scheduling functionality and sometimes not. Using ordinary user-defined block types, this can be solved in two ways, either by having one block that supports all features that can be turned on and off using parameters, or by having one block for each variant. The first technique means that all variants need to be anticipated and leads to complicated diagrams and runtime overhead. The second technique works only for a small number of variants and leads to duplicated code.

Niklas Fors. "Implementation of the Bloqqi compiler and editor". Technical Report 98.
LU-CS-TR:2016-252, ISSN 1404-1200. Lund University, 2016.

Bloqqi is a data-flow language with language constructs that allows variants to be created of the same basic structure, which avoids the drawbacks of the previous techniques. The language is based on user-defined block types which are defined as diagrams with diagram inheritance. The inheritance mechanisms *connection interception* [FH14] and *block redeclaration* [Mod12] allow diagram behavior to be specialized, similar to virtual methods in ordinary object-oriented programming. In addition to this, Bloqqi has feature-oriented language constructs for describing variants and which are used by the editor to automatically derive smart editing support in the form of wizards. The user can then use the wizard for feature selection when creating a block.

Bloqqi has been developed in collaboration with the company ABB in the automation domain, in the context of process industry. The focus has been on reusability and how to describe variability within the language. The language has been inspired by the automation language ControlBuilder from ABB [Con16] and the simulation and modeling language Modelica [Mod12]. Like in ControlBuilder, the execution scheme in Bloqqi is periodic, where a program is executed a number of times per second. Many of the language constructs in Bloqqi are novel and general for data-flow languages, and not only applicable in the automation domain.

We have implemented a compiler and a graphical editor for the Bloqqi language. The compiler generates C code that can be compiled to executable binaries and the editor visualizes programs and allows the user to change them. Bloqqi has both a textual and a visual syntax, and the textual syntax is used as a serialization format when programs are stored on file.

This report describes how the compiler and the editor have been implemented using the metacompilation system JastAdd [EH07b] and the semantic formalism reference attribute grammars [Hed00]. The semantic specification is specified once and used in both the compiler and the editor, for generating code or visualizing diagrams. The semantic analysis is needed for visualizing diagrams, since the visualized content of a diagram is *computed*, based on, for example, user-defined block types, inheritance and visual instance inlining. When the user changes a diagram in the editor, the editor maps the changes in the computed structure back to the source program.

This report is structured as follows. Section 2 introduces Bloqqi using an example. Section 3 describes the overall architecture for the compiler and the editor. Section 4 gives background for JastAdd and reference attribute grammars. Section 5 and Section 6 describe the implementation of the compiler and the editor, respectively. Section 7 explains how code generation is done in the compiler, and how Bloqqi programs can be integrated with simulated models using the Function Mockup Interface (FMI) [Blo+12] and integration with ABB controllers. Section 8 discusses some of the implementation design choices. Section 9 presents related work. Finally, Section 10 concludes the report.

2 Introduction to Bloqqi

This section introduces Bloqqi using the textbook definition of a proportional-integral-derivative (PID) controller [ÅM12]. These controllers are commonly used in industry[1]. A PID-controller consists of three parts: the proportional part, the integral part, and the derivative part. The integral and derivative parts can be seen as extensions, or features, to a P-controller. With these extensions, the following variants can be created: P, PI, PD, PID. This section will first show what a P-controller looks like, then extend this base structure using inheritance to a PI-controller. We will then describe how all the four variants can be described using the feature-based language mechanisms *wirings* and *recommendations*, where the user can select what variant to use in an automatically derived wizard [FH16a].

A P-controller computes the control signal based on the current error, that is, the difference between the reference value and the measured value. The control signal $u(t)$ for a P-controller can be described as follows [ÅM12].

$$u(t) = K_p e(t) \tag{1}$$

Here, $e(t)$ is the error at time t. The coefficient K_p describes how much the error should be amplified in the control signal.

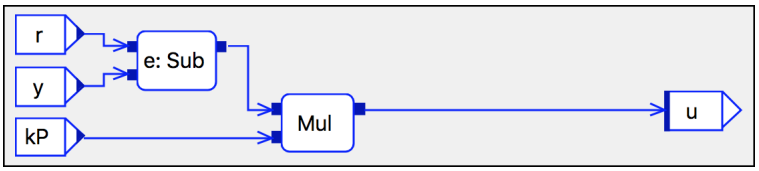
A PI-controller adds an integrator term that takes into account the history of the error as well. The control signal for a PI-controller can be described as follows [ÅM12].

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau \tag{2}$$

The coefficients K_p and K_i describe how much the proportional part and the integral part should be taken into account in the control signal.

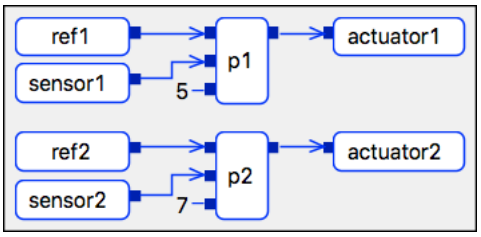
Using the above definition of a P-controller, we can now model it as a diagram in Bloqqi. A diagram can have input parameters, blocks, state variables, output parameters and connections between them. Figure 1 shows the diagram `P` for the P-controller, where both the textual and the visual syntax is shown. The diagram has three input parameters: the reference value (`r`), the measured value (`y`) and the proportional coefficient (`kP`). The error is computed by taking the difference between reference value and the measured value. The control value is exposed as an output parameter (`u`) and computed by multiplying the error with the coefficient. The diagram makes use of the standard blocks `Sub` for subtraction and `Mul` for multiplication. The subtraction block has been given the explicit name `e` (for the control error) to make the diagram easier to read and extend. The connections describe how the data flows in the diagram, from the source of the connection to the target of the connection (arrow head), thus, the data-flow is directed.

[1] With some extensions to the textbook definition, which will not be covered in this section




```
diagramtype P(r: Int, y: Int, kP: Int => u: Int) {
  e: Sub;
  Mul_1;
  connect(r, e.in1);
  connect(y, e.in2);
  connect(e.out, Mul_1.in1);
  connect(kP, Mul_1.in2);
  connect(Mul_1.out, u);
}
```

Figure 1: P-controller in Bloqqi.

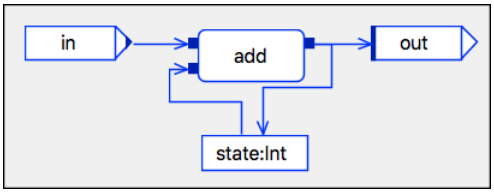



```
diagramtype Main {
  ref1: OperatorValue;
  sensor1: Sensor;
  p1: P;
  actuator1: Actuator;
  ...
  connect(ref1.out, p1.r);
  ...
}
```

Figure 2: A diagram that uses the P-controller defined in Figure 1.

Programs in Bloqqi are executed in a periodic manner, for example, 10 times per second. In each period, sensor values are read, which are used together with state variables to compute control signals. After each period, the control signals are sent to actuators that affect the physical system, for example, to open or close a valve in a tank. In each period, all blocks are executed and executed in an order that does not violate the data-flow order. Thus, if block a produces data for block b, then block a will execute before b.

Having defined the P-controller as a diagram, we can now use it in another diagram as a block, which illustrates user-defined block types. Figure 2 shows how the P-controller is instantiated as two blocks, `p1` and `p2`, in the diagram `Main`. These two blocks are then connected to operator values, sensors, and actuators. An operator value is like a sensor, but the value is set by a human operator in the plant. When a diagram is instantiated as a block, the input parameters are shown as input ports (left side) and output parameters are shown as output ports (right side) on the block. The ports can then be connected using connections. For example, there is a




```
diagramtype Acc
    (in: Int => out: Int) {
  var state: Int;
  add: Add;
  connect(in, add.in1);
  connect(state, add.in2);
  connect(add.out, out);
  connect(add.out, state);
}
```

Figure 3: Accumulator implemented using state variable (`state`).

connection to the input port `r` on the block `p1`.

The execution of a diagram depends on sensor values, but also on state values that may be embedded inside other blocks. The P-controller does not use any state variables, but the PI-controller does, in order to store the history of the error. We will show this in the next section.

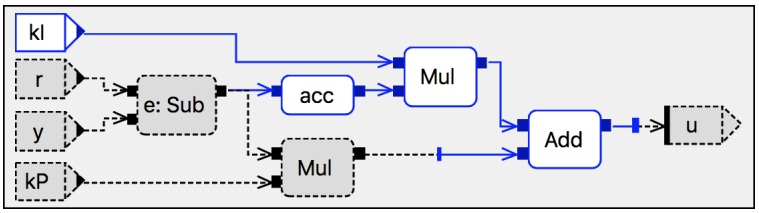
2.1 Diagram Inheritance

Bloqqi supports inheritance between diagrams, that is, a diagram S can *extend* another diagram T. This means that all parameters, blocks, state variables, and connections in T are also in S, in a transitive manner. We say that S is a subtype of T and T is a supertype of S. A subtype can add new parameters, blocks, state variables and connections, and it can also specialize the behavior defined in the supertype by using *connection interception* [FH14] and *block redeclaration* [Mod12]. Connection interception allows a subtype to intercept a connection defined in a supertype to instead go via a local block. Block redeclaration allows a subtype to redeclare (specialize) the type of a block defined in a supertype.

We can use these mechanisms to implement the PI-controller as an extension of the P-controller, instead of as a separate diagram type. The PI-controller adds an integrator part that is added to the control value (the output parameter `u`). In computerized control, the integrator can be approximated as an accumulator of the error [ÅW96]. An accumulator is a block that has an internal state and that is incremented with the input value at each period. The internal state is implemented as a state variable (has prefix `var`) that is stored between the execution periods. Figure 3 shows how the diagram `Acc` is implemented.

Figure 4 shows the diagram `PI`, which extends the diagram `P`. Inherited parts are visualized as grey/dashed (grey as in stone, and which cannot be changed) and local parts as blue/solid. An input parameter is added for the integral coefficient (`kI`)[2]. The accumulator block (`acc`) is incremented each period with the error value. The accumulated value is then multiplied with the coefficient and

[2]The sampling time is included in the integral coefficient.




```
diagramtype PI(kI: Int) extends P {
  acc: Acc;
  Mul_2;
  Add_1;
  connect(e.out, acc.in);
  connect(kI, Mul_2.in1);
  connect(acc.out, Mul_2.in2);
  connect(Mul_2.out, Add_1.in1);
  intercept u with Add_1.in2, Add_1.out;
}
```

Figure 4: A diagram describing a PI-controller by extending the diagram `P` defined in Figure 1.

added to the control value. The new term is added to the control value by intercepting the connection to `u` defined in the supertype, so that the connection now will go through the local `Add` block.

Having defined the subtype `PI` of type `P`, we can now replace blocks of type `P` with type `PI` using block redeclaration, if we want a PI-controller instead of a P-controller. For example, we can create a subtype of `Main` and *redeclare* the blocks `p1` and `p2` to the type `PI`, as the following code illustrates.

```
diagramtype SubMain() extends Main {
  redeclare p1: PI;
  redeclare p2: PI;
  ...
}
```

The inheritance mechanism `redeclare` requires that the new block type (`PI`) is a subtype of the old block type (`P`). The subtype `PI` adds the input parameter `kI`, so we need to connect the new corresponding ports as well. For example, by adding two more operator value blocks and connect them to the new ports.

2.2 Sensors and actuators

In addition to state variables (`var`), Bloqqi also supports communication variables that are used to communicate with the environment, modelling sensors and actuators. A communication variable has either the dataflow direction `input` or `output`. Sensors use `input` variables and actuators use `output` variables. These

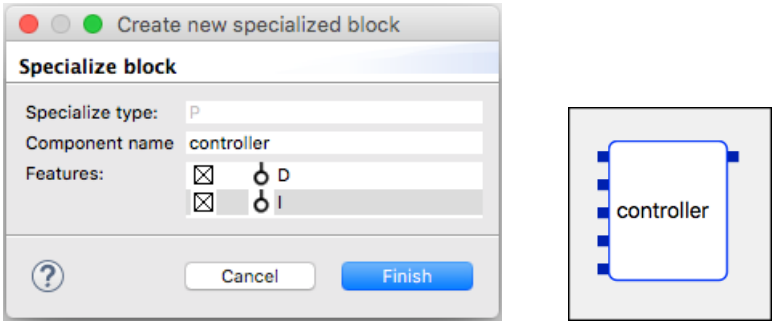


Figure 5: A PID-controller block is created using an automatically derived wizard based on recommendations.

communication variables are used to implement the block types `OperatorValue`, `Sensor` and `Actuator` that are used in diagram `Main` (Figure 2).

2.3 Recommendations

The previous example used inheritance to model two variants of a controller, but Bloqqi also supports the feature-based language constructs *recommendations* and *wirings* [FH16a]. This allows variants of the same base structure to be specified and the user can select which variant to use. The base structure is defined as a diagram and recommendations define *recommended features* for the base structure. For the controller example, the P-controller is the base structure, and the integral part and the derivative part are recommended features. Recommended features are specified as blocks and include information about how these blocks are connected into the base structure. The editor then uses the recommendations to automatically derive smart-editing support in the form of a wizard, which lists all recommended features for a given diagram. When a user creates a block of a diagram with recommendations, the user can specialize the block using the wizard.

For example, assume that the P-controller defined in Figure 1 has two recommended features, one for the integral part and one for the derivative part. When the user creates a block of the `P` diagram, a wizard is shown with available features that can be selected. If both features are selected, a PID-controller is created, and which is illustrated in Figure 5, where both the wizard and the PID-block are shown. Using this wizard, several controller variants can be created from the same base structure: P, PI, PD and PID. In this case, the number of variants is only four, but when the number of features increase, the number of variants grow exponentially.

The following code shows how the recommendations for the diagram `P` are specified.

```
recommendation P {
  D: DPart[e.out, kD: Int, u];
  I: IPart[e.out, kI: Int, u];
  I before D;
}
```

The diagram `P` has two recommended features, `D` and `I`, which are of block types `DPart` and `IPart`, respectively. The type `DPart` implements the derivative part and the type `IPart` implements the integral part. The code within the square brackets (`[]`) describes how the features are connected into the `P` diagram, if they are selected. We can see that both features use the error value `e` and that they affect the control signal `u`. Exactly how this is done is defined as a *wiring* in the block type (which is not shown). In this case, both the wirings specify that the parameter `u` will be intercepted. Since both features intercept the same output parameter, the application of the interceptions need to be ordered if both are selected. The statement `before` tells that if both `D` and `I` are selected, then `I` is applied before `D`.

Recommendations are modular: the code for the recommendations for diagram `P` above can be split into three different recommendation statements. This allows libraries to be created with recommendations that are extended with more features defined by the user, which are specified as recommendations outside the library.

A feature may also have alternatives, for example, there may be several implementations of the integral functionality which have different characteristics. Alternatives are specified as subtypes. Thus, to add an alternative to the `I` feature, a subtype to `IPart` needs to be created. The wizard will then collect all subtypes for a feature and list them as alternatives.

The wizard shows features hierarchically. This means that if a block is created of a diagram with recommendations and this diagram has blocks with recommendations, then the features for the inner blocks will be shown in the wizard as well. See [FH16a] for more information about the hierarchical property of recommendations.

2.4 Visual Instance Inlining

User-defined block types allow common functionality to be abstracted into block types to increase reuse, but sometimes the abstractions make the diagrams harder to understand. Bloqqi allows a block that is an instance of another diagram to be visually inlined [FH16b], meaning that the block is visually replaced with the content of the block type. This is shown in Figure 6, where the block `controller` in Figure 5 has been inlined, and with some additional blocks. The inlined blocks are white with dashed black borders. Inlined blocks are allowed to be moved freely without any restrictions, making it possible for the user to align the blocks in a way that she wants. The user can also create connections to and from the inlined blocks, in which case the editor will implicitly create parameters and connections.

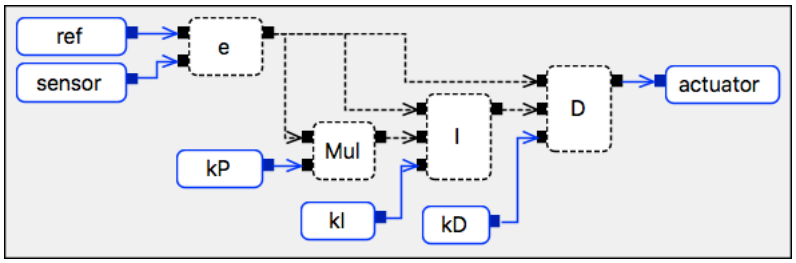


Figure 6: The block in Figure 5 is visually inlined (white/dashed).

2.5 Functions and External Procedures

Bloqqi does not only support user-defined block types as diagrams, but block types can also be defined as algorithmic code or as external procedures. The standard blocks in Bloqqi, like `Add` and `Sub`, are defined as algorithmic functions, and which are free from side effects. For example, the function `Div` for division is implemented as follows.

```
function Div(Int in1, Int in2 => Int out) {
  if (in2 != 0) {
    out = in1 / in2;
  } else {
    out = 0;
  }
}
```

Algorithmic functions are easily mapped to ordinary programming languages like C.

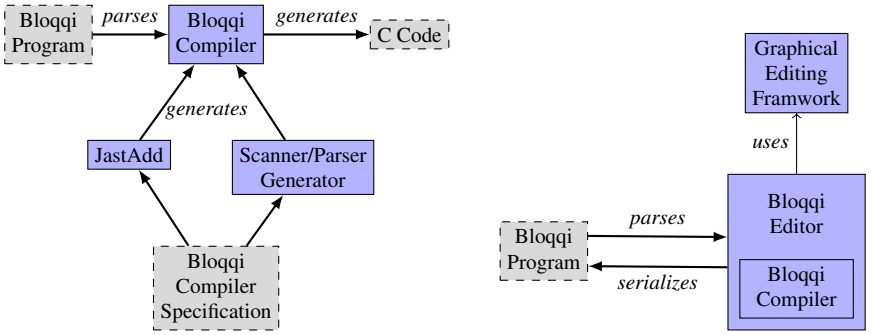
Algorithmic code can be defined externally, using an external procedure declaration. The external procedure is implemented using a C function that is linked together with the generated C code for the diagram. For example, we can define an external procedure that prints an integer as follows.

```
external Print(in: Int);
```

3 Overall architecture

The Bloqqi compiler and editor use a common semantic analysis that is specified once and reused by both the tools. The compiler generates C code that can be compiled to execute Bloqqi programs. The editor renders Bloqqi programs visually and allows the user to change them. Both these computations are based on the same semantic specification.

Bloqqi programs are stored on file using a textual syntax defined by a context-free grammar. Both the compiler and the editor load Bloqqi programs in the same



(a) Compiler architecture. **(b)** Editor architecture.

Figure 7: Architecture overview for the Bloqqi compiler and editor.

way, by first parsing the textual syntax of the program, and then analyzing the parsed structure in order to report errors, generate C code or render diagrams visually. When the user changes a program in the editor and saves the change, the editor will serialize the program back to file again using the textual syntax.

Figure 7 shows the overall architecture for the compiler and the editor. The figure shows that the compiler is generated by the metacompilation tool JastAdd [EH07b] together with a scanner and parser generator, called JFlex and Beaver. The semantic analysis is specified in JastAdd and the syntactical analysis is specified using the scanner and parser generators. JastAdd represents programs as abstract syntax trees (ASTs) and the output of JastAdd is Java classes representing the ASTs. The parser then constructs ASTs by creating instances of the generated Java classes. We will in later sections describe what semantic specifications look like in JastAdd.

The figure also shows that the compiler is used by the editor, since many of the semantic analyses done in the compiler are also useful in the editor. Examples include rendering of diagrams and showing of error messages. In addition to the compiler analyses, the editor includes advanced editing operations and additional semantic analysis only used in the editor, which both are defined using JastAdd. The editor also includes plain Java code to, for example, implement the visualization in terms of shapes or colors and implement the interaction with the user.

The editor uses the Graphical Editing Framework (GEF)[3], which is an Eclipse framework for building graphical editors. This means that the Bloqqi editor runs within Eclipse as a plugin. GEF uses the design pattern Model-View-Controller, where the model can be any model defined as Java classes. For the Bloqqi editor, the model is the AST defined by JastAdd, and the view describes how the AST is visualized.

[3] `https://eclipse.org/gef/`

Compiler	SLOC
Abstract grammar	124
Frontend	
Scanner specification	99
Parser specification	447
Semantic specification (JastAdd)	3314
Backend (JastAdd)	1203
Main program (Java)	580
Total	5767

Editor	SLOC
Editing operations (JastAdd)	1088
Java	7482
Total	8570

Table 1: Source lines of code (SLOC) for the Bloqqi compiler and editor.

The number of source lines of code (SLOC) for the compiler and editor are shown in Table 1, which have been measured with the tool `cloc`[4]. The compiler is divided into two parts: the frontend and the backend. The syntactical and semantic analyses are performed in the frontend, and the backend is responsible for pretty printing and generating C code. The *abstract grammar* defines the AST structure. As described earlier, the editor reuses the compiler, but adds additional JastAdd specifications for editing operations and also Java code that describes the visualization, user interaction, etc.

4 Reference Attribute Grammars

The semantics for the compiler and the editor are specified in the metacompilation tool JastAdd [EH07b] with the declarative semantic formalism *Reference Attribute Grammars* (RAGs) [Hed00]. One strength of RAGs is specifying the semantics of software languages in a modular and extensible way. RAGs have previously been used to specify compilers for complex languages like Java [EH07c] and Modelica [Mod12; Åke+10b]. The Java compiler first supported version 4 and was later extended with support for Java 5, 6, 7 and 8, where each extension is specified as a separate module that extends the previous modules [ÖH13; Hog14]. The Java compiler has also been extended with other kinds of analysis, for example, data-flow analysis [Söd+13] and refactoring support [Sch+08; Sch+09; SM10]. The Modelica compiler *JModelica.org* has also been extended in a modular way, with language constructs for model optimization [Hed+10]. This exemplifies that RAGs

[4] `https://github.com/AlDanial/cloc`

is a powerful semantic formalism useful for specifying extensible tools for complex software languages.

Reference attribute grammars [Hed00] is an extension to Attribute grammars (AGs) by Knuth [Knu68]. Attribute grammars decorate trees with *attributes* that are *computed properties* of nodes in a tree. These attributes are defined *declaratively* by directed *equations*. The left-hand side of an equation is an attribute, and the right-hand side is an expression over the attributed AST for computing the value of the left-hand side attribute. The expression must be free from externally observable side effects. Thus, computing the value of the same attribute several times will yield the same value. In JastAdd, the attributes are computed on-demand, that is, the attributes are computed when they are accessed, and cached for subsequent accesses so that each attribute value is computed at most once.

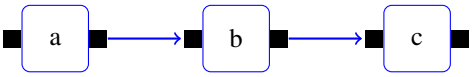
RAGs adds *reference attributes* to AGs, allowing the value of an attribute to refer to other tree nodes. This makes it possible to super-impose graphs over the tree and to more easily express non-local dependencies, such as that the type of a name use is the same as the type of the corresponding name declaration, which is usually a distant node. A RAG is specified in an object-oriented manner using inheritance for node types and where equations can be overridden, like virtual methods. A typical example of reference attributes is the name binding relation, which can be expressed as a reference attribute `decl` on the use node and that refers to the corresponding declaration node. Another example in the Bloqqi compiler is the data-flow graph in a diagram.

JastAdd is a metacompilation system based on aspect-orientation and RAGs [HM03; EH07b]. The central data structure in JastAdd is the *abstract syntax tree* (AST), which is decorated with attributes. ASTs are defined by an *abstract grammar*, which describes the structure of the AST in an object-oriented way. The attributes are defined in static *aspects* in terms of the abstract grammar. Static aspects allow related attributes to be grouped together. For example, we can have one aspect for all attributes concerning name analysis and another aspect for attributes that support editor operations.

From an abstract grammar specification and an attribute specification, JastAdd generates Java code, where each node type in the abstract grammar is translated to a Java class and each attribute is translated to a method in the corresponding class. JastAdd does not support parsing. Instead, any parser can be used as long as it can create objects of the generated Java classes. For the Bloqqi compiler, we have used the scanner generator JFlex[5] and the parser generator Beaver[6], both allowing arbitrary Java code to be specified as semantic actions.

[5] `http://jflex.de/`
[6] `http://beaver.sourceforge.net/`

```
diagram MyDiagram {
  a;
  b;
  c;
  connect(a, b);
  connect(b, c);
}
```



(a) Textual representation **(b)** Visual representation

Figure 8: An example program in PicoBloqqi

⟨*program*⟩ ::= ⟨*diagram*⟩*

⟨*diagram*⟩ ::= 'diagram' ⟨*id*⟩ '{' ⟨*stmt*⟩* '}'

⟨*stmt*⟩ ::= ⟨*block*⟩ | ⟨*connection*⟩

⟨*block*⟩ ::= ⟨*id*⟩ ';'

⟨*connection*⟩ ::= 'connect' '(' ⟨*id*⟩ ',' ⟨*id*⟩ ')' ';'

Figure 9: Context-free grammar for PicoBloqqi. Kleene star (*) means zero or more.

4.1 PicoBloqqi

RAGs and JastAdd will be introduced for a simplified variant of Bloqqi called *PicoBloqqi*. A program in PicoBloqqi consists of diagrams, which in turn consist of blocks and connections between the blocks that describe the data-flow. An example program is shown in Figure 8. As we can see, the language has both a textual and a visual syntax. The blocks implicitly have one input port (on the left side in the visual syntax) and one output port (on the right side). The statement `connect(a, b)` has two parts: `a` is the source and `b` is the target. The source implicitly refers to the output port on the block referenced by `a` and the target implicitly refers to the target port on `b`, as we can see in Figure 8b. PicoBloqqi does not support user-defined block types.

A context-free grammar (CFG) expressed in EBNF for the PicoBloqqi language is shown in Figure 9. A program consists of zero or more diagrams. A diagram starts with the keyword `diagram` followed by an identifier and then statements surrounded by curly braces. A statement can be either a block or a connection. A block has an identifier. A connection uses the keyword `connect` followed by the source and the target specified as identifiers. The CFG only specifies the concrete textual syntax of the language, and to analyze a program with JastAdd,

```
Program ::= Diagram*;
Diagram ::= <Name:String> Stmt*;
abstract Stmt;
Block : Stmt ::= <ID:String>;
Connection : Stmt ::= Source:VarUse Target:VarUse;
VarUse ::= <ID:String>;
```

Figure 10: Abstract grammar for PicoBloqqi.

```
public class Program extends ASTNode {
  public List<Diagram> getDiagrams();
}
public class Diagram extends ASTNode {
  public String getName();
  public List<Stmt> getBlocks();
}
public abstract class Stmt extends ASTNode {
}
public class Block extends Stmt {
  public String getID();
}
```

Figure 11: Part of API for generated Java classes from Figure 10.

we need to create an abstract grammar as well and a parser that transforms the textual syntax into an AST according to the abstract grammar.

The abstract grammar for PicoBloqqi is shown in Figure 10. Each rule in the abstract grammar defines an AST class, where the left-hand side is the name of the class and the right-hand side describes the children and the tokens (the latter enclosed by <>) for the class. For example, the AST class `Program` consists of zero or more diagrams. Tokens and children can be named using colon (`:`), where the left-hand side is the name and the right hand-side is the type. For example, the class `Diagram` has one token called `Name` of type `String`. When a child is specified without an explicit name, the name will implicitly be the same as the type. The production for `stmt` in the CFG contains an alternative, and this production is translated to an abstract class `Stmt` with two subclasses `Block` and `Connection` in the abstract grammar. JastAdd supports arbitrary inheritance levels as long it is single inheritance. This is useful, for example, when modelling expressions, where we can have an abstract superclass `Expr`, and another abstract superclass `BinExpr` that extends `Expr`, and then concrete subclasses like `AddExpr` that extends `BinExpr`. Multilevel inheritance allows more code reuse since common code can be moved to the right level in the inheritance chain.

From the abstract grammar in Figure 10, JastAdd generates Java classes, and part of the API for the generated classes is shown in Figure 11, where some getter methods are shown. We can see that Kleene star (*) is translated to a method with `List` as return type (and the method name ending with `s`). JastAdd also generates

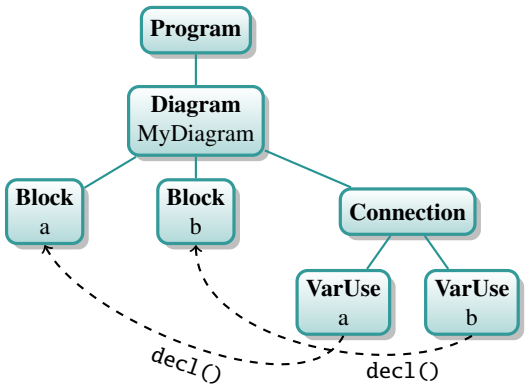


Figure 12: Simple AST. The `VarUse` nodes have a reference attribute `decl` that refer to their declarations.

```
aspect NameAnalysisDecl {
  // Declarations
  syn Block VarUse.decl();

  // Equations
  eq VarUse.decl() = lookup(getID());
}
```

Figure 13: Attribute `decl`.

other kinds of methods, for example, setter methods that are used by the parser during the construction of the AST. All generated classes are subtypes to the class `ASTNode`, which can be used to specify attributes and methods that are valid for all nodes.

Defining Static Semantics using Attributes

We can now analyze programs by defining the semantics using attributes over ASTs. For example, we can define an attribute for name uses that describes the name binding relation, so that each `VarUse` node has an attribute `decl` that refers to its declaration. This is illustrated in Figure 12, which consists of one diagram with two blocks and one connection. The value of the `decl` attribute for the two `VarUse` nodes are shown in the figure as dashed lines.

The attribute `decl` is specified in Figure 13, and consists of two parts, first the attribute declaration and then the attribute equation. The attribute is declared as a *synthesized attribute* (using the keyword `syn`) with the type `Block` on `VarUse` nodes with the attribute name `decl`. A synthesized attribute means that the equation is defined by the node *itself*. The equation can then use information from the node, thus, tokens, children and other attributes declared on the node. In this

example, we define the `eq`uation to use another attribute `lookup` and we pass the token `ID` as parameter. Thus, the right-hand side of the equal sign (=) specifies how the value of the attribute is computed. The `decl` attribute is a reference attribute, since the value is a reference to another AST node (the type is `Block`).

Since we use the attribute `lookup` in the equation for `decl`, the `lookup` attribute is also declared on `VarUse`. The `lookup` attribute will search for a `Block` node that matches the parameter `name`. Since we are looking for `Block` nodes, the equation for `lookup` attribute needs to be defined by a node that has access to `Block` nodes. In this case, `Diagram` nodes have access to `Block` nodes, so the `Diagram` node can ask its children if they declare the name in question. Thus, the `lookup` attribute is declared on `VarUse` nodes, but the equation is defined by a `Diagram` node. When we want the equation for an attribute to be defined by an ancestor node in the tree, we can use *inherited attributes*[7] (using the keyword `inh`). We declare the inherited attribute `lookup` (line 3) and the equation for it (line 8-9) in Figure 14. The equation simply delegates to a synthesized attribute `localLookup` passing the same parameter. Equations for inherited attributes have the form `eq T.getC().a() = ...`, where `T` is the node type defining the equation, `C` is the child of `T` for which the equation is valid (it is possible to have different equations for different children), and `a` is the name of the attribute.

The equation for the synthesized attribute `localLookup` simply traverses the statement list in order to find out which of the statements that declares the given name, using the synthesized attribute `declares` declared on `Stmt`. Synthesized attributes are similar to virtual methods: each subtype can have its own equation defining the attribute. This allows an equation for the attribute `declares` to be defined on each subtype of `Stmt` (including `Stmt`). On lines 20-22, one equation is defined for `Block` and another one is defined for `Connection`. The equation for `Block` simply compares the given name with the value of the token `ID` and returns itself if they match.

Figure 15 illustrates how the attribute `decl` is computed for the `VarUse` node with the identifier `b`. For synthesized attributes, the attribute and the equation are depicted together. For the inherited attribute `lookup`, the attribute declaration and the equation are depicted separately. We can see that all attributes except `decl` are parameterized and that the parameter `b` is passed around. The computation for the `VarUse` with the identifier `a` is similar, but with the parameter `a` instead and the search will stop at the first `Block` node, since it declares the name `a`.

To summarize how attributes work, we declare attributes on tree nodes that describe node properties. The attributes are defined by equations. The equations are free from externally observable side-effects (we can compute the equation several times and get the same result). Where the equation is defined depends on the attribute kind. For synthesized attributes, the equation is defined by the node itself. For inherited attributes, the equation is defined by an ancestor in the tree.

[7] Not to be confused with inheritance in object-orientation

```
1  aspect NameAnalysisLookup {
2    // Declarations
3    inh Block VarUse.lookup(String name);
4    syn Block Diagram.localLookup(String name);
5    syn Block Stmt.declares(String name);
6
7    // Equations
8    eq Diagram.getStmt().lookup(String name)
9      = localLookup(name);
10
11   eq Diagram.localLookup(String name) {
12     for (Stmt s: getStmts()) {
13       if (s.declares(name) != null) {
14         return s.declares(name);
15       }
16     }
17     return null;
18   }
19
20   eq Block.declares(String name)
21     = getID().equals(name) ? this : null;
22   eq Connection.declares(String name) = null;
23 }
```

Figure 14: Attribute `lookup`.

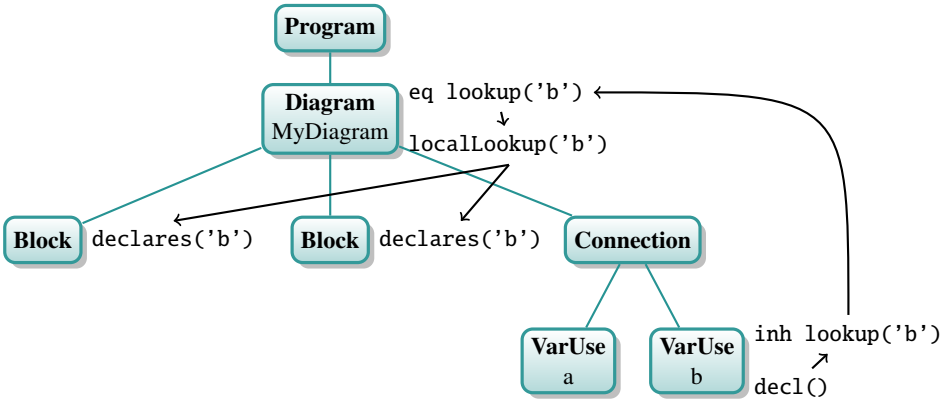


Figure 15: Illustrates how the attribute `decl` is computed for the rightmost `VarUse` node.

Non-Terminal Attributes

There are other kinds of attributes than synthesized and inherited attributes. We will now describe *non-terminal attributes* (NTAs) [Vog+89], which are also known as *higher-order attributes*. NTAs are used heavily in the Bloqqi compiler and editor. An NTA is an attribute whose value is a subtree that can itself have attributes, and defined by an equation like any other attribute. In contrast to normal reference attributes, the equation on the right-hand side for an NTA must return a *fresh* subtree.

For example, we can use an NTA to represent undeclared names. The name analysis in Figure 14 uses the value `null` to represent when a block is not found. Instead of using the value `null`, we can use the *Null Object pattern* [Woo97] for undeclared names, and declare an NTA that represents the null object. This can be specified as follows.

```
syn nta Block Program.unknownBlock() = new UnknownBlock("unknown");
```

Here, we declare the synthesized NTA `unknownBlock` on the root node, that is, on the type `Program`, so that the NTA can be shared by the whole tree. The equation creates an object of the type `UnknownBlock`, which is a subtype of `Block`, and is specified as follows in the abstract grammar.

```
UnknownBlock : Block;
```

We use a subtype so that we can have different equations for undeclared names and declared names for *other* attributes, which is often useful during semantic analysis. What remains is to change the name analysis in Figure 14 to use the new NTA. Since it is defined on the root node, we can declare an inherited attribute on all nodes that gives access to the NTA, and whose equation is defined by the root node. The inherited attribute can then be used in the equations in the name analysis. This will lead to that the value of the `decl` attribute will refer to the `unknownBlock()` NTA value for all name uses that cannot be resolved.

NTAs are thus useful for representing undeclared names. Other use cases are representing predefined names and predefined types. These examples do not depend on information in the tree. However, the equations for NTAs may use other attributes, and thus, the NTAs are also very useful for augmenting the AST with new subtrees that depend on computed information. Examples in the Bloqqi compiler include an attribute that describes the list of all blocks for a diagram, including the inherited ones.

Other Attributes

JastAdd supports more kinds of attributes than the ones already discussed. *Collection attributes* [Boy96; Mag+09] and *circular attributes* [Far86; MH07] are two attribute kinds that are useful and that are used in the Bloqqi compiler.

```
1  aspect Errors {
2    coll TreeSet<String> Program.errors();
3
4    VarUse contributes "Block \"" + getID() + "\" is not declared"
5      when decl() != null
6      to Program.errors();
7  }
```

Figure 16: Collection attribute `errors` and a contribution rule.

A collection attribute is declared on a node, and the value is defined by *contributions* that are collected and combined together. For example, the set of all errors in a program can be defined as a collection attribute on the `Program` node, and each kind of error is then defined as a contribution rule. This can be seen in Figure 16. First, we declare the `coll`ection attribute `errors` on `Program` as a set of strings (line 2). Then, we define a contribution rule that reports if a `VarUse` node uses a name that is not declared. When a collection attribute is accessed, the evaluation engine will traverse the tree and look for contributions, and if they match, they will be added to the collection attribute.

A circular attribute is an attribute whose value may depend on itself. For example, in a graph of nodes, we can have an attribute that represents the set of reachable nodes from a node. Computing this set for a node may be circular, since graphs can contain cycles. This issue can be solved by declaring the attribute as `circular`, which will make the attribute engine to solve the equation system using fix-point iteration. A circular attribute is well-defined if the domain of the attributes involved in the cycle can be arranged in a lattice of finite height and all equations involved are monotonic functions.

For more information about RAGs, see [Hed11; FH15].

5 The Bloqqi Compiler

The Bloqqi compiler analyzes programs syntactically and semantically to report errors and generate code. The compiler specification is divided into two parts, the frontend and the backend, both defined by a set of aspects. Most of the semantic analysis is done in the frontend, and is reused in the backend to generate code. The frontend includes, for example, aspects for name analysis and type analysis.

The name analysis is implemented in a similar fashion as the previous section described, with a reference attribute `decl()` on identifier nodes that refer to the corresponding declaration. The actual search for the declaration follows the `lookup` pattern described earlier, but taking inheritance into account. Since Bloqqi supports user-defined block types, the compiler does name lookup for types as well, which follows the same `lookup` pattern.

```
diagramtype Main() {
  t: T;
}
diagramtype T
  (in: Int => out: Int) {
}
```

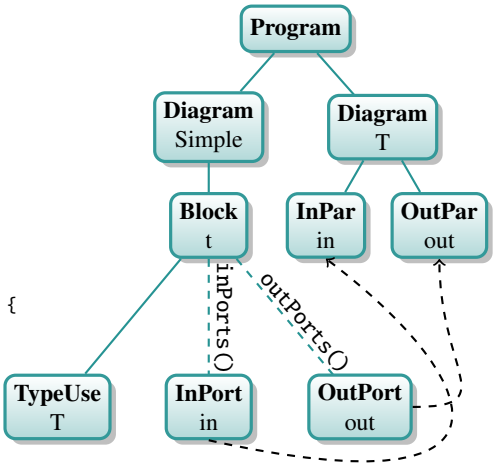


Figure 17: The ports for a block are computed based on the parameters in the type of the block. The ports are represented in the compiler as the NTAs `inPorts()` and `outPorts()`.

5.1 Computed Structures

The compiler makes extensive use of *computed structures* in the form of *non-terminal attributes* (NTAs).

As described in Section 2, the parameters in a diagram will show up as ports when the diagram is instantiated as a block. Since the ports depend on the block type, they are computed based on the type analysis, and modelled as NTAs. Each block has two NTAs, `inPorts()` and `outPorts()`, which is the list of input ports and the list of output ports, respectively. These attributes are illustrated in Figure 17, where the diagram `Main` has one block `t` of type `T`. This block will then have the NTAs `inPorts()` and `outPorts()` that reflect the parameters in the type `T`, and the ports will have a reference to their corresponding parameter declaration. The NTAs are defined as follows.

```
syn nta List<InPort> Block.inPorts() = type().createInPorts();
syn nta List<OutPort> Block.outPorts() = type().createOutPorts();
```

Thus, the equation asks the type for a list of input ports and output ports using the methods `createInPorts()` and `createOutPorts()`. These methods must return a fresh subtree, since a diagram can be instantiated as many blocks, and each block needs to have its own list of input ports and output ports. Then each `InPort` and `OutPort` node have attributes describing the dataflow, that is, the incoming and outgoing connections. These attributes are, for example, used to

check that input ports only have one incoming connection, to check type errors, and to generate code.

The example in Figure 17 is very simple and does not make use of inheritance. When a diagram S extends another diagram T, then all parameters, blocks and connections in T are also in S. Thus, if T defines a block b, then there are two versions of this block, one in T and one in S, which may have different data-flow. For example, S may add a connection from block b to a local block. This is also modelled as NTAs in the compiler. Thus, each diagram has an NTA `blocks()` for its blocks taking into account inheritance. Then each block in `blocks()` has the NTAs `inPorts()` and `outPorts()`, as described earlier. Similar to the computed ports, the computed blocks have references back to the declared block. Each diagram also has NTAs for the input parameters, output parameters and the connections.

The equations for the attributes `blocks()` and `connections()` take inheritance into account, and will include the blocks and connections defined in a supertype. These attributes also consider the mechanisms of block redeclaration and connection interception. Block redeclaration will change the type of a block defined in a supertype, which is reflected in the equation for `blocks()`. Connection interception will replace one connection defined in a supertype with two new connections, and the two new connections will be in the attribute `connections()`.

In addition to inheritance, the attributes `blocks()` and `connections()` will also consider visual inlining, and expand an inlined block with its content, in a transitive manner. Like with inheritance, the computed blocks will have references to the blocks that are inlined and expanded, which may be several blocks since inlining is transitive. These attributes are then used for editing operations in the editor.

5.2 Semantic Analysis

The compiler performs several semantic analyses to find compile time errors before generating code, and some of them are discussed below.

Block Type Analysis. The type of a block may refer to another diagram, an algorithmic function or an external procedure. If the block type refers to a primitive type, like an `Int`, or to a structured type (similar to structs in C), a type error is reported. When a block is redeclared, the compiler will check that the new block type is a subtype of the previous type. Redeclare only works when the block type is a diagram, since inheritance is only supported for diagrams.

Data Type Analysis. The types that describe the data are called *data types*, and they are different from block types, although the same name space is used. Data types are explicitly attached to parameters and variables, and implicitly attached to literals. A data type may be either a primitive type or a structured

type, otherwise a type error is reported. Note that a diagram is not a valid data type, which means that diagrams are not allowed to be passed around as data. Nor does the language support references to diagrams, or any other block type. The compiler will also check that the data types of the end points of connections and interceptions are type compatible. An end point may be attached to an *anchor*, like a parameter, a port or a variable.

Direction Analysis. Since the data-flow is directed, it is important that the end points of connections and interceptions follow the data-flow rules for Bloqqi. The source of a connection may only refer to an *anchor* that is a source anchor, and the target of a connection may only refer to a target anchor. Source anchors include input parameters and output ports on blocks, and target anchors include output parameters and input ports on blocks. Variables are also anchors, and there are different rules depending of what kind of variable. State variables are both source and target anchors, `input` variables are source anchors, and `output` variables are target anchor. The compiler checks that these restrictions are not violated.

Unique Input Analysis. For a target anchor, like an output parameter and an input port on a block, the number of incoming connections should be either zero or one. However, when generating code, the number of incoming connections to output parameters and input ports on blocks should be equal to one, since default values are currently not supported for parameters. This is only checked when generating code, and not in the editor, since otherwise too many errors would be reported during the development of a program.

Finite Block Structure Analysis. It is a compile-time error to have recursion in user-defined block types, since it would lead to infinite structures. An example is a diagram consisting of a block of the same block type as the diagram itself. The compiler does reachability analysis over the block types to detect circular type definitions, and if found, reports an error. The reachability analysis is defined using circular attributes, so that each diagram has an attribute defining the set of reachable block types from that diagram. The value of the attribute is defined as a set so that the blocks involved in a circular block type definition can be highlighted as red in the editor.

Data-Flow Cycles Analysis. It is a compile-time error if the data-flow graph in a diagram contains cycles. To analyze this, the compiler represents the blocks as nodes and the connections as edges in a graph. The compiler then uses this graph to detect cycles using reachability analysis, similar to the analysis for circular block types. Note that state variables are not represented in the graph, which means that cycles can be broken by introducing a state variable in a data-flow cycle. However, this will delay the value from the state variable by one execution period.

Circular Inheritance Analysis. Since diagram inheritance is supported, the compiler needs to check that the inheritance chain is not circular. As an example, diagram `A` extends diagram `B` and `B` extends `A`, leading to a cycle. This is detected by the compiler and is again implemented using circular attributes, but in this case, the attribute value is of boolean type, only saying if there is a cycle or not.

6 The Bloqqi Editor

The editor is implemented as an Eclipse plugin using the Graphical Editing Framework (GEF) in Eclipse. GEF is based on the model-view-controller (MVC) pattern, where the model can be any arbitrary Java objects. The Bloqqi editor reuses the attributed AST from the compiler as the model, with some additional attributes and methods that are specifically defined for the editor, like editing operations for changing the AST. Since JastAdd generates ordinary Java classes and translates attributes to methods, the AST can easily be used in GEF. The view part of the editor describes how the diagrams are visualized, like shapes, colors, etc. The controller part describes the interaction with the user and how the interaction affects the AST, like adding a new block or intercepting a connection. The editor also includes advanced editing support in form of a wizard for feature selection.

When a file is opened in the editor, the editor reads the file by parsing it into an AST that is decorated with attributes. The attributes are then used to visualize the diagram. When the user saves the diagram, it is serialized back to the file using the pretty printer defined for the compiler.

6.1 Visualizing Computed Structures

The visualization of a diagram requires semantic analysis, since the content of a diagram is computed. For example, the blocks and connections in a diagram depends on inheritance. Also, the number of input ports and output ports a block has depends on the block type. In addition to this, visual inlining needs to be considered.

As described earlier, the compiler already computes these structures as NTAs. So instead of reimplementing these computations in the editor as well, we reuse these NTAs in the editor. For example, the attributes `blocks()` and `connections()` are used when a diagram is visualized.

In addition to the computed structures, the editor needs some computed properties, such as if a block is inherited or local, or if a connection is intercepted with two new connections. These computed properties are then used in the visualization. For example, a block that is local is visualized as blue with solid lines and an inherited block is visualized as grey with dashed lines.

When a file is opened, the decorated AST is mapped to data structures required by the MVC pattern in GEF. Thus, the editor will create controller objects for the

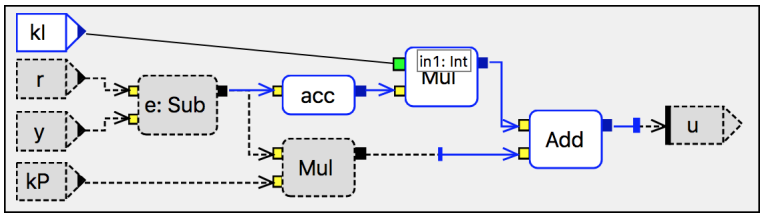


Figure 18: Semantic feedback when a connection is created. Connecting to a green port will not introduce any semantic errors. The mouse hovers over the port on the `Mul` block, and the editor shows the port name and type.

interesting nodes in the decorated AST (from the editor perspective). The controller objects describe the interaction with the user, and also how the view objects are created. The controller and the view objects have the form of a tree, matching the form of the AST. A controller object knows both about the corresponding view object and the node in the AST, and uses attributes in the AST node to set properties in the view object, such as if a block is local or inherited.

6.2 Semantic Feedback

Since all attributes defined in the compiler are accessible in the editor, we can reuse them for more than visualization. In particular, we can use attributes for giving semantic feedback. There are two kinds of semantic feedback, *permanent* and *transient semantic feedback*. Examples of permanent semantic feedback include connections that have type errors or are involved in a data-flow cycle, and which are colored red. Transient semantic feedback is typically given by the editor while the user is editing a diagram.

One example of transient semantic feedback is when creating a new connection or when intercepting a connection. A connection is type correct if the types of the connection end points are type compatible. When a connection is created, the user first points out the source end point, then the target end point. When the user is about to point the target end point, the editor will give semantic feedback by coloring the ports, either yellow or green. Green ports mean that connecting to this port will not introduce any new semantic errors, and yellow means that a semantic error will be introduced. Furthermore, during connection creation, the editor will color those input ports yellow that already have one or more incoming connection. This is illustrated in Figure 18.

Another example of transient semantic feedback is to show the name and data type of a block port when the user hovers with the mouse over it. This is also shown in Figure 18. These port properties depend on the block type, and the feedback is implemented using attributes already defined in the compiler.

6.3 Name Bindings

Since the visualization of a diagram is computed, changes in diagrams made by the user need to be mapped back by the editor to the source AST. For example, connections are visualized as arrows between two ports, but they are represented in the source AST as two qualified names that are resolved by the name analysis. Thus, when an endpoint of a connection is defined by the user clicking on a port `p` on block `b`, a qualified name `b.p` is created that will be part of the connection, for instance, `connect(b.p, ...)`. When the user clicks on the port, the editor asks the corresponding port node in the AST how it can be accessed using the method `access`, and a qualified name is returned. This technique is similar to the technique for refactoring presented by Max Schäfer [Sch+08; FH13], see related work for details (Section 9).

The same block name could be declared both in a subtype and a supertype, representing two different blocks. To disambiguate the access, the name can be qualified. For example, consider a block `b` in type `T` and a subtype `S` of `T`, then block `b` can be accessed as `T::b` in `S`. Note that the user sees all blocks and can distinguish between them simply by clicking on them, and does not have to be aware of how the disambiguation is done textually. When a connection is created to a block declared in a supertype, the editor will automatically qualify it with the supertype to avoid future name conflicts. It is, however, a compile-time error to declare two blocks with the same name in the same diagram type.

Some block names are not interesting for the user, for example, the name of an addition block. When a simple block, like an addition, is created, the editor will automatically create a locally unique name for it. The name will be on the form `T_number`, where `T` is the type of the block and `number` is an integer number that is generated by the editor, for example, `Add_1`.

6.4 Preserving Name Bindings

When a declaration is introduced or renamed, it is important that the editor preserves existing name bindings. This is challenging when names can be shadowed, like with block names in combination with inheritance. Here, introducing a new declaration could shadow declarations in supertypes, and the name bindings of existing accesses could change if no special action was taken.

Consider the example in Figure 19, where the type `B` accesses the block `b` declared in the supertype `A`. Thus, there is a name binding between the access of the block and the declaration of the block, and it is important that this name binding is preserved when the user changes the diagram. In this example, there are two kinds of changes that may affect the name binding. One way is to introduce a new block in `B` also called `b`, and another way is to rename the existing block `b2` to `b`.

The editor preserves name bindings by rewriting the source program to use fully qualified names for name uses that access blocks declared in a supertype.

```
diagramtype A {
  b: ...;
}
diagramtype B extends A {
  b2: ...;
  connect(b.out, b2.in);
}
```

Figure 19: In the compiler, the accesses are fully qualified, and the access to `b.out` is rewritten to `A::b.out`.

JastAdd supports *rewrites* [EH04] that allow AST nodes to be conditionally rewritten. A rewrite rule is specified for an AST class and consists of a rewrite condition and an expression for the new node. The rewrite is activated on demand, that is, when the node is accessed for the first time and the rewrite condition evaluates to true. Both the rewrite condition and rewrite expression may use attributes, and in this case, attributes from the name analysis and type analysis are used. For the example in Figure 19, the access to `b.out` is rewritten to `A::b.out`. Having fully qualified names makes it easy to implement rename and operations for adding new blocks. Shadowing is only permitted for blocks, and is not allowed, for instance, for parameters and types. When the diagram is saved and stored to file, the pretty printer will simplify accesses by removing the type qualifier if the block can be accessed without it.

6.5 Updating the view

When the user changes a diagram, the editor maps the change to the AST. As described earlier, attributes are computed when needed and cached for subsequent accesses to improve performance. Since the AST is changed (structure or tokens may have changed), cached attribute values may become inconsistent, and all attribute caches are therefore cleared. Then the AST tells the editor that a change has occurred using the Observer pattern and the editor responds to this by updating the view. When the view is updated, the diagram is revisualized and the attributes needed are recomputed. Clearing all attributes may lead to costly recomputations, an issue we discuss in further detail in Section 8.4.

6.6 Layout

The user can move blocks around freely and change the size of them. The coordinates and the sizes for the blocks are stored as annotations in the same file as the diagram. The layout is often used by users as a secondary notation that encodes information and makes the diagram more readable [Moo09]. The graphical information is only used in the editor and will not affect the meaning of the diagram,

thus, the compiler will just ignore this information. When a diagram is opened, the editor will do automatic layout if the diagram lacks coordinates. The user can also enforce the editor to do automatic layout as an editing operation.

We use the tool KLay [Sch+14] for automatic layout, which has specialized layout algorithms for data-flow diagrams. KLay will also layout the connections, but currently, the editor does not store coordinates for the connections. Instead, the editor use Manhattan layout to layout the connection between two endpoints, which will possibly visually break the connection in the middle. By adding coordinates for connections, more advanced automatic layout could be used, yielding even better results.

7 Code Generation

Bloqqi programs can be executed by first generating C code using the Bloqqi compiler, then compile the generated C code using a C compiler, and finally run the compiled C code.

The execution scheme is periodic, for example, a program is executed 10 or 100 times per second. There is one execution entry point for each program, and currently, that is the diagram type with the name `Main`. The execution scheme is illustrated in the following C code.

```
Main_STATE state = ...;
while (...) {
  read_sensor_values(&state);
  Main(&state);
  send_to_actuators(&state);
  sleep();
}
```

First, state variables are initialized, which are stored between the periods. Then in each period, first sensor values are read, then the control values are computed based on state variables and sensor values, and finally the control values are sent to actuators. The task of the code generation is to generate code for the `Main` function from the corresponding diagram type (and all diagram types it uses, transitively).

The code generation has been implemented as *inter-type methods* [Kic+01; EH07b], which are imperative Java methods that are defined in aspects, like attributes. Inter-type methods may have side effects and can use attributes. The inter-type methods for the code generation traverse the AST and use attributes to compute what C code to generate. This is a common pattern for code generation when using RAGs; attributes specified for the frontend are reused by the backend when generating code. For example, when generating code for a diagram type, the attribute `blocks()` is used, which includes all blocks, both local and inherited blocks, thus flattening the diagram inheritance. Also, each port on a block has attributes describing the ingoing and outgoing connections, which are used to translate connections to C code. In addition to the attributes defined in the fron-

```
diagramtype Inc(in: Int=>out: Int);

diagramtype Main() {
  inc: Inc;
  print: Print;
  connect(2, inc.in);
  connect(inc.out, print.in);
}
```

(a) Bloqqi code

```
typedef struct {
  Int out;
} Inc_RES;
Inc_RES Inc(Int in);

void Main(Main_STATE* _state){
  Inc_RES inc = Inc(2);
  Print(inc.out);


}
```

(b) Generated C code

Figure 20: A simple Bloqqi program is translated to C code. The definition of diagram type `Inc` has been left out.

tend, some additional attributes are defined specifically for the code generation. For example, an attribute is defined to compute if a diagram type contains state variables directly or indirectly, this by checking if the types of the blocks contain state variables, in a transitive manner.

7.1 Simple Code Generation Scheme

The basic code generation scheme is translating all diagram types to functions, and all blocks to function calls and storing the results in local variables. The connections between the blocks describe the data-flow, and thus the dependencies between the blocks constrains in which order the blocks can be executed. For example, when there is a connection from a block `A` to a block `B`, then `A` needs to be executed before `B`, and the result of executing `A` is used when executing `B`.

The data-flow is a partial order, thus, for most diagrams, there are many ways to execute the same diagram. The Bloqqi compiler creates a total execution order by using the data-flow together with the block declaration order in the diagram type. This causes the blocks for the same diagram to always be executed in the same order, thus, the execution order is deterministic. Having a deterministic execution order is useful when debugging programs. For example, while debugging, we may use an external function that prints values for debugging purposes.

The data-flow in Bloqqi is acyclic. If a diagram contains data-flow cycles, a compile-time error is reported. However, the user can break data-flow cycles by explicitly introducing variables.

Figure 20 shows how a simple Bloqqi program is translated to C code. The diagram type `Main` will print out the value 3 in each period. Note that no state variables or sensor values are used in this example. The `Main` diagram consists of two blocks: the block `inc` that increments the input value by one and the block `print` that prints out the value. These two blocks are translated to two function calls. We can see that the block `inc` is executed before the block `print`, since `inc` produces data to `print`. We can also see that the function calls contain actual

parameters that match the incoming connections for the corresponding block. For example, the literal 2 is connected to the block `inc`. Since diagram types can have several output parameters, and thus several return values, a `struct` containing the return values is generated for each diagram type with at least one output parameter. In this example, the struct `Inc_RES` has been generated for the diagram type `Inc`. A local variable of this struct is then used to store the result from the function call to `Inc`, so that the value can be used in the call to `Print`. The function `Main` has an implicit parameter `_state` that is used to store variables between the periods, and which will be described next.

For a diagram type `S` that extends another diagram type `T`, then when generating code for `S`, the content of `T` is copied to `S`, in a transitive manner. Thus, the subtype relation is not directly visible in the generated code. This is possible since Bloqqi does not support references or dynamic dispatch. Instead all types can be determined statically by the compiler. Anonymous subtypes are translated to corresponding top-level diagram types with generated names that are unique to avoid name conflicts.

7.2 Variables

Bloqqi supports variables that can be used to store values between periods. The code generation handles variables by generating structs representing the variables and passing around pointers to values of these structs as implicit function parameters. We have already seen the implicit state variable in the `Main` function.

Figure 21a shows an example Bloqqi program that makes use of variables. The `Main` diagram consists of one accumulator block and one block that prints the accumulated value. The accumulator block has an internal state variable that is increased each period with the value of the input parameter. In this case, the accumulator will in each period increase the value by 1. Thus, when executing this program, the output will be the sequence of numbers starting from 1, that is, 1, 2, 3, etc. The diagram type `Acc` implementing the accumulator consists of one `var`iable `state` and one addition block. Every time a block of this type is executed, the value of the variable `state` is added together with the input parameter `in`. The result of the addition is stored in the variable `state` again and is exposed as the output parameter `out`. The visual representation of the diagram type `Acc` can be seen in Figure 3.

The generated C code for Figure 21a is shown in Figure 21b. The variable inside `Acc` is represented with the struct `Acc_STATE`. Since `Main` has an accumulator block, the state struct for `Main` consists of a field `acc` with the type `Acc_STATE`. If, for example, `Main` instead had two accumulators, then the state struct would have two fields with the type `Acc_STATE`. When the function `Acc` is called, the field `acc` is passed as an implicit parameter. In general, the state structs are generated by flattening out the blocks that directly or indirectly contain variables. Structs correspond to diagrams and field names correspond to block names.

```
diagramtype Main() {
  acc: Acc;
  print: Print;
  connect(1, acc.in);
  connect(acc.out, print.in);
}

diagramtype Acc
    (in: Int => out: Int) {
  var state: Int;
  add: Add;
  connect(in, add.in1);
  connect(state, add.in2);
  connect(add.out, out);
  connect(add.out, state);
}
```

(a) Bloqqi code

```
typedef struct {
  Int state;
} Acc_STATE;
typedef struct {
  Acc_STATE acc;
} Main_STATE;

void Main(Main_STATE* _state) {
  Acc_RES acc = Acc(1, &_state->acc);
  Print(acc.out);


}

Acc_RES Acc
    (Int in, Acc_STATE* _state) {
  Add_RES add = Add(in, _state->state);
  _state->state = add.out;
  Acc_RES _result;
  _result.out = add.out;
  return _result;

}
```

(b) Generated C code

Figure 21: A Bloqqi program that contains a variable.

```
diagramtype Main {
  sensor: Sensor;
  actuator: Actuator;
  connect(sensor.out, actuator.in);
}
diagramtype Sensor(=> out: Int) {
  input value: Int;
  connect(value, out);
}
diagramtype Actuator(in: Int) {
  output value: Int;
  connect(in, value);
}
```

(a) Bloqqi program

```
typedef struct {
  Int value;
} Sensor_STATE;
typedef struct {
  Int value;
} Actuator_STATE;
typedef struct {
  Sensor_STATE sensor;
  Actuator_STATE actuator;
} Main_STATE;
```

(b) Generated struct

Figure 22: A diagram describing a sensor uses an `input` variable to communicate with the environment.

7.3 Communication Variables

As described in Section 2.2, Bloqqi also supports communication variables that are used to communicate with the environment.

Figure 22a shows an example program that uses an `input` variable and an `output` variable, encapsulated in the diagram types `Sensor` and `Actuator`, respectively. These diagram types are then used in the diagram `Main`. In the generated code, the `input` and `output` variables will be in the same `struct` as state variables. For this example, the generated `struct` can be seen in Figure 22b. Using this `struct`, we can now write a program that lets the Bloqqi program communicate with the environment, as follows.

```
Main_STATE state = ...;
while (...) {
  state.sensor.value = sensor_read();
  Main(&state);
  actuator_write(state.actuator.value);
  sleep();
}
```

7.4 FMI Integration

Testing a control system before deployment is a useful technique to reduce the risk of unexpected behavior in the real setting. To test the control system, the real setting needs to be modelled and simulated in some way. Modelica [Mod12] is a good choice when the real setting can be described by differential algebraic equations. The Functional Mockup Interface (FMI) is a standard that allows dynamic models from different tools to be used together [Blo+12], and which is supported by many Modelica tools. Thus, Modelica models can be exported as simulation executables, called Functional Mockup Units (FMUs). FMUs can be used to simu-

late the real setting and be used together with Bloqqi programs. The FMU consists of binaries to simulate the models and XML files describing the models, and may also contain the source code in C for the binaries.

The FMI standard has two ways that models from different tools can be used together: *model exchange* and *co-simulation*. Model exchange can be seen as white box. When several model-exchange FMUs are used together, they make up an equation system that is solved as one system with one numerical solver, external to the FMUs. A co-simulation FMU, on the other hand, can be seen as a black box with input and output, and contains an internal numerical solver. When several co-simulation FMUs are used together, they are solved independently and communicate with each other at discrete communication points. The program that connects several FMUs and manages the communication is called the *master*. The master can then tell an FMU to simulate for a given time. Co-simulation fits well with testing Bloqqi programs. This is because a control system communicates at discrete points in time with the environment by using sensors and actuators, and we want the simulated model to be solved independently and to only interact periodically with the Bloqqi program.

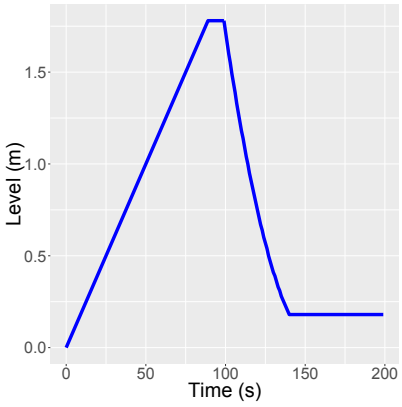
Tank example

We have experimented with FMI and Bloqqi by modelling a tank in Modelica and a tank regulator in Bloqqi. The model has been exported as a co-simulation FMU using the Modelica compiler JModelica.org [Åke+10b]. The master program has been written in C, which communicates both with the FMU and the Bloqqi program. We have used the C library FMILibrary[8] to access the FMU.

The tank contains some kind of liquid. It has one input valve that can be opened to fill the tank and one output valve to empty the tank. The tank also has one sensor to measure the current liquid level. The tank regulator uses the level sensor to decide when to open and close the valves, in order to achieve a given set point level. The Bloqqi program will first fill the tank to the liquid level 1.8 meters and stay there for some time, and then empty it to 0.2 meters liquid. The result of running the Bloqqi program with the simulated model is plotted in Figure 23a.

The tank model in Modelica is shown in Figure 23b, which has been adapted from an example in [Fri10]. The model consists of variables and equations. Some of the variables are constant and some vary over time. When the input valve is open, the inflow of liquid is constant (5 liters/s). When the output valve is open, the outflow depends on the pressure, and thus on the current liquid level and the area of the output valve ($5\text{cm} * 5\text{cm}$). The liquid level is expressed as a *mass balance* equation, that is, expressed in terms of how much liquid that is entering or leaving the tank. The `input` variables `upperValveOpen` and `lowerValveOpen` specify if the valves are open (value above 0.0). The values of these variables can be changed during simulation, and the variables act as actuators that are set

[8] `http://www.jmodelica.org/FMILibrary`



(a) Simulation results

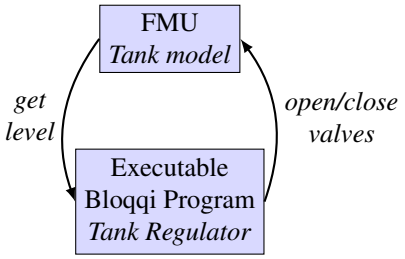
```
model Tank
  Real inFlow(unit="m3/s") = 0.005;
  Real area(unit="m2")     = 0.5*0.5;
  Real outArea(unit="m2")  = 0.05*0.05;

  Real level(start=0, unit="m");
  Real qInFlow(unit="m3/s");
  Real qOutFlow(unit="m3/s");

  input Real upperValveOpen;
  input Real lowerValveOpen;
equation
  der(level) = (qInFlow-qOutFlow)/area;
  qInFlow =
    if upperValveOpen > 0.0 then
      inFlow
    else 0.0;
  qOutFlow =
    if lowerValveOpen > 0.0 then
      (outArea*sqrt(2*9.82*level))
    else 0.0;
end Tank;
```

(b) Tank model in Modelica

Figure 23: Using a FMU to simulate a tank and a tank regulator in Bloqqi.



(a) Illustration

```
while (tcur < tend) {
  // Read sensor values
  state.tank.levelSensor.value
    = get_real(fmu, "level");

  // Run tank regulator
  Main(&state);

  // Send to actuators
  set_real(fmu, "upperValveOpen",
    state.tank.upperValve.open);
  set_real(fmu, "lowerValveOpen",
    state.tank.lowerValve.open);

  // Run FMU for one second
  do_step(fmu, tcur, 1.0);

  // Increase current time with 1
  tcur += 1.0;
}
```

(b) C code

Figure 24: Master program for the tank example.

by the tank regulator. The actual inflow (`qInFlow`) and outflow (`qOutFlow`) are described in terms of the `input` variables.

The master program is illustrated in Figure 24a. The Bloqqi program is retrieving the current liquid level, computing if the valves should be open or closed and sending this information to the FMU, and then the FMU is running for some time to simulate the tank. The C code for the master program is illustrated in Figure 24b. The C variable `state` contains the variables representing the sensors and actuators in the Bloqqi program. First the tank regulator is run and then the FMU is run for 1 second in simulated time. Thus, the tank regulator is simulated to run once every second and is assumed to take zero time.

In this example, the master program is written in C, and it connects the Bloqqi program with the FMU. It would also be possible to generate FMUs for Bloqqi programs, allowing Bloqqi programs to be imported to any tool that supports FMI import. For example, it would then be possible to import a Bloqqi as a block in Simulink using the FMI toolbox[9]. However, there is a difference between dynamic models that are described by equations like in Modelica and the programs in Bloqqi, since the equations in Modelica are typically continuous and the variables vary continuously over time, but the values in Bloqqi are computed at discrete points in time.

[9] `http://www.modelon.com/products/fmi-tools/fmi-toolbox-for-matlabsimulink/`

7.5 ABB ControlBuilder Integration

We have experimented with integrating Bloqqi programs with ABB ControlBuilder [Con16], in two alternative ways. First, Bloqqi programs can be exported in the XML-format used in ControlBuilder, which allows Bloqqi programs to be imported to ControlBuilder. Second, we have integrated the C code that the Bloqqi compiler generates with programs defined in ControlBuilder, in an experimental version of it, removing the extra step of importing the program. The ControlBuilder integration has been defined in a separate module that extends the Bloqqi compiler, and which contains around 300 lines of JastAdd code and around 60 lines of Java code.

We have also experimented with running Bloqqi programs on controller hardware from ABB, called *AC 800M controllers*, which can be seen in Figure 25. ControlBuilder can deploy programs on controller hardware, like AC 800M. When the program is running on the hardware, the values that are computed in the program can be seen in ControlBuilder, for instance, for debugging. We have been able to see what values the Bloqqi programs computed in ControlBuilder while the programs were running.

8 Discussion

During the development of Bloqqi and its tools, we made some design choices which are discussed in this section.

8.1 Texual Syntax

Bloqqi has already from the start had a textual syntax that is complete for the language and that is used as the serialization format. The visual syntax is only partial and not all language features are supported by the editor. We believe that the textual syntax has been helpful, for example, when developing the language, but also since it makes the programs more concise and it allows existing text-based tools to be used.

Developing A New Language

When developing a new language, it is useful to experiment with different language constructs in an iterative manner. Developing a graphical editor, including determining how constructs are visualized and how the user interacts with the program, requires much effort. Typically, during the development of new language constructs, we developed them gradually, starting with a problem we tried to solve and discussions about how to solve it. After reaching some conclusions, we tried out the ideas by implementing the new language constructs in the compiler, by first defining the textual syntax for them and then the semantic analysis. After the

Figure 25: We have performed test runs of Bloqqi programs on AC 800M controllers by ABB. The controller was not connected to any physical process, but we were able to see the values computed by the Bloqqi program by using the ControlBuilder software.

new constructs were implemented in the compiler, we implemented them in the editor, by first defining the visualization for them, and then the user interaction. Typically, the diagrams were changed textually during the development when only the visualization had been implemented.

With a textual syntax and a compiler, it is easy to write test cases, in the form of input (the program) and output, where the output can be an error message or the output when executing the program. Before implementing the language constructs in the editor, the semantic analysis was tested thoroughly in the compiler. This way, the implementation in the editor could focus on only the visualization and the user interaction.

During the discussions about new language constructs, we often used both textual notation and visual notation to better understand each other. Not that these notations used in the discussions became final in the language, but it made the communication easier. The visual notation is useful to describe graph-based structures, like the data-flow. The data-flow is difficult to understand from only looking at the `connect` statements. Examples of when we used only the textual notation include when we discussed how to specify recommendations, and still there is no visual notation for specifying them. It is hard to exactly define when visual notation is useful and when textual notation is useful, but we have found them both valuable.

Serialization

In some way, we need to serialize the programs so that we can store them. There are alternatives to having a custom textual syntax. One alternative is to store the diagrams in a binary serialization format like Java serialization. Another is to use a general textual serialization format like the Human-Usable Textual Notation [OMG04] (HUTN). We believe that programs should be stored textually, and not binary, so that programs can be created and changed with ordinary text editors. Having a textual syntax also allows existing text-based tools to be used, such as version-control systems and text-based merge tools.

The question is then if the serialization format should be automatically derived or custom made. A benefit of using a custom textual syntax is that it is more concise and readable, and more adapted for the domain. A drawback is that it requires more implementation effort.

Having a serialization format that is directly derived from the abstract grammar or meta-model makes stored programs more fragile, since changing the meta-model will break the compatibility with existing programs. Having a custom textual syntax allows the internal representation to be changed without any backward compatibility issues. One example of a fragile serialization technique is to refer to nodes in lists using node indices, which makes a local reordering change in one file unnecessarily affect other files that refer to the changed node.

8.2 Simple Prototype Language

The Bloqqi language is a prototype language, and during the development of a prototype language, we believe that one should strive to make the language as simple as possible, by reducing the number of nice-to-have features. One example of a language feature in the Bloqqi language that was introduced because it was nice to have is the simplification of block accesses, which is illustrated with the following code.

```
diagramtype Main {
  a: T;
  b: T;
  connect(a, b);
}
```

Here, the block type `T` is assumed to have one input parameter and one output parameter, with the names `in` and `out`, respectively. However, in the `connect` statements, only the block names are used, `a` and `b`. This is valid in Bloqqi, since there is only one input parameter and only one output parameter in `T`, thus, the accesses will implicitly refer to the ports corresponding to these parameters. This is implemented using rewrites, so that the access `a` is rewritten to `a.out` and the access `b` is rewritten to `b.in`. This language feature was introduced in the early days of Bloqqi and has not yet been removed, because many test cases use the feature. In retrospect, it would be better if this feature was never introduced, since it only makes the compiler implementation more difficult.

8.3 The Choice of Editor Framework

The graphical editor is implemented using the Graphical Editing Framework (GEF) in Eclipse. During the development of the editor, we evaluated and tested different graphical editor frameworks.

There are several frameworks built on top of GEF, for example, Graphiti[10] and GMF[11]. These frameworks try to make it easier to create graphical editors, since GEF requires much boilerplate code and has a steep learning curve. One requirement that we had was that we wanted to use the JastAdd AST as the model, thus, the framework should support arbitrary Java objects as the model. The GMF framework requires the model to be an EMF model, defined by an EMF meta-model[12]. It would be possible to use JastEMF [Bür+11], a tool that enables semantics to be defined over EMF models with the use of JastAdd, but this adds an additional generation step. Also, EMF models have higher memory footprint than JastAdd models. Furthermore, we could not find any straight-forward way in GMF for supporting the computed structures that we needed in Bloqqi.

[10] `https://eclipse.org/graphiti/`
[11] `https://eclipse.org/modeling/gmp/`
[12] `https://eclipse.org/modeling/emf/`

We tested the Graphiti framework, which removes boilerplate and makes it easy for creating graphical editors for simple languages. However, creating an editor using Graphiti for a more complicated language and features not anticipated by the frameworks developers, leads to writing GEF code. Because of this, we chose to use GEF directly instead, since it offers more flexibility. One observation is that if one wants to create a novel language with new features, it is better to chose a framework with a high degree of flexibility, since there is otherwise a risk of the framework being too limiting.

Other approaches to creating graphical editors is to define the visual syntax using a formal visual grammar. However, there is no standard way of doing this, like context-free grammars for textual languages. Approaches have been suggested that use constraint multiset grammars [Mar94] and graph grammars [Min02; Min06]. These approaches generate visual parsers that take a diagram and determine whether it is syntactically correct or not. One benefit of this is that it allows *free-hand editing*, making it possible to temporarily have diagrams that are syntactically incorrect. In comparison, the Bloqqi editor is a syntax-directed editor, but the visual syntax for Bloqqi is rather simple and does not restrict the user very much.

The challenge with these formal approaches is to model the computed visual syntax of Bloqqi. For example, the ports on a block are computed based on the block type, which in turn depend on inheritance since subtypes can have new parameters, and the blocks are computed based on inheritance. Thus, the visual syntax in Bloqqi is *dependent* on the semantics. To check if a diagram is valid visually (for example, how many ports a block has), the semantics needs to be analyzed first. Free-hand editing would allow the user to add more ports to a block than the block type has parameters, which does not make sense.

8.4 Incremental Evaluation

As described in Section 6, the editor flushes all attribute caches whenever a diagram is changed, and the attributes are recomputed again when they are needed. We have run the editor with diagrams of 500 blocks and connections, and we perceived the interaction as immediate. However, flushing all attribute caches for each change could be problematic for large programs.

Performance can be improved by adding incremental evaluation [SH12], which reduces the number of attribute caches that are flushed. This can be done by tracking the dependencies between the attributes dynamically during the attribute evaluation, and then use this dependency information to limit the flushing to attributes that are affected by the change. The dependency tracking can be implemented with different levels of granularity, where the tradeoff is between precision and performance overhead.

We have experimented with very a coarse level granularity, where the program is divided into two parts, the user code and the library code. It is possible to write

the compiler attributes in such a way that the library code is not dependent on the user code. When a change is made, the attribute computations for the library code are not flushed, only the attributes for the user code. We believe that this is a good tradeoff in many cases, since most code will be in libraries, which is not changed during a normal editing session. However, it is part of future work to study this idea in more detail.

9 Related Work

We will now relate the implementation of the Bloqqi compiler and editor to previous work on using attribute grammars in visual editors, or editors with semantic feedback.

Schmidt et. al. [SK03] have also used attribute grammars to implement structured editors for visual languages. However, they focus on general and predefined visual patterns, such as lists, graphs, tables and line connections, which can be reused by different visual languages. The user can refine these patterns and also create new visual patterns. Similar to our approach, they use an abstract syntax tree as the central data structure. The task for the language developer is then to reuse and assign visual patterns to tree nodes, which describe how the nodes are visualized. The visual patterns can be specialized using attribute equations, such as, defining the size of list nodes or how much padding there is between nodes. However, their attribute grammars do not support reference attributes, and they do not support visualization of computed structures, which the Bloqqi language and editor are heavily based on.

Schäfer et. al. [Sch+08; Sch+09; SM10] have used RAGs to implement refactoring for Java, like renaming and extract method. Especially relevant for this report is the presented renaming technique. The authors introduce the concept of an *access function*, which is the inverse of the lookup function. The lookup function is like the `lookup` attribute, described earlier in this report. A lookup function takes a program point and an access node, and gives back the corresponding declaration, if found. The access function on the other hand takes a program point and a declaration and gives back an access node, that can be used to access the given declaration at the given program point. The access function may introduce qualified names, like adding the qualifier `this` in the returned access node to resolve name conflicts. Schäfer et. al. also observed that the invariant of preserving name bindings is useful when specifying and implementing renaming. If the invariant is violated during a rename operation, the rename is aborted and all changes are undone. The technique for the access function is implemented in the Bloqqi editor for renaming, but is also used when creating and editing connections. A difference is that the AST in Bloqqi is rewritten to always use fully qualified names, in order to avoid name collisions and issues with shadowing.

Another related compiler is the JModelica.org compiler by Åkesson et. al. [Åke+10b], which is also implemented using RAGs and JastAdd [Åke+10a]. Similar to Bloqqi, Modelica is object-oriented with inheritance between models. The task for the Modelica compiler is to take a Modelica program and transform it into one equation system, this by removing the object-orientation and flattening the objects. Central to the JModelica.org compiler are also NTAs that are used to flatten the object structure.

There are other attribute systems than JastAdd that supports RAGs. Notable examples include Silver [VW+10], Kiama[Slo+13], RACR [Bür15], and JavaRAG [For+15]. Silver is similar to JastAdd in that it has its own specification language and generates code, however, it is based on functional programming rather than object-orientation like JastAdd. Kiama, RACR and JavaRAG on the other hand are libraries: Kiama in Scala, RACR in Scheme and JavaRAG in Java. This makes it easier to integrate attribute grammar with existing data structures.

10 Conclusions

We have in this report described how the compiler and the graphical editor for the data-flow language Bloqqi have been implemented. The Bloqqi language has many advanced language features that have been specified using the semantic formalism reference attribute grammars, which represents programs as abstract syntax trees that are decorated with computed node properties called attributes. The semantic specification is specified once and is reused by both the compiler end the editor, avoiding double maintenance. The visualization of a diagram is computed based on the semantic analysis, for example, blocks are inherited and the ports on a block depend on the block type. These computed properties of the visualizations match well with how attributes work, and the editor uses mostly the attributes to visualize a diagram.

We have also presented how Bloqqi programs are executed, by describing what C code that is generated from Bloqqi programs. Using the code generation, Bloqqi programs have been used together with simulated models specified in Modelica and integrated using the functional mockup interface (FMI). This way, we have been able to test Bloqqi on realistic examples, without having to integrate with real world systems.

Future work includes both extensions to the language, and improvement of the implementation techniques. An example of an interesting language extension would be to add support for *control connections*, where part of the structured data can be reversed, allowing the data to flow backwards along the connection. This is used to simplify diagrams and improve control performance [PH02]. We have used Bloqqi programs together with Modelica models exported using FMI, but it would also be valuable to export Bloqqi programs using FMI so that Bloqqi programs can be used in other tools. An example of interesting future work for the editor is to

experiment with incremental evaluation of the attributes, allowing more efficient attribute evaluation when diagrams are edited [SH12]. It would also be interesting to investigate how the computed visual syntax could be specified using a more higher-level specification, compared to plain Java that the editor uses today.

References

[Åke+10a] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. "Implementation of a Modelica Compiler Using JastAdd Attribute Grammars". In: *Science of Computer Programming* 75.1-2 (Jan. 2010), pp. 21–38.

[Åke+10b] Johan Åkesson et al. "Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problem". In: *Computers and Chemical Engineering* 34.11 (Nov. 2010), pp. 1737–1749.

[ÅM12] Karl Johan Åström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, Sept. 2012.

[ÅW96] Karl Johan Åström and Björn Wittenmark. *Computer-controlled systems: theory and design*. 3rd ed. Prentice Hall, 1996.

[Blo+12] Torsten Blochwitz et al. "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models". In: *9th International Modelica Conference*. 2012.

[Boy96] John Tang Boyland. "Descriptional Composition of Compiler Components". PhD thesis. University of California, Berkeley, Sept. 1996.

[Bür15] Christoff Bürger. "Reference attribute grammar controlled graph rewriting: motivation and overview". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*. 2015, pp. 89–100.

[Bür+11] Christoff Bürger et al. "Reference Attribute Grammars for Metamodel Semantics". In: *Software Language Engineering*. Vol. 6563. LNCS. Springer, 2011, pp. 22–41.

[Con16] ControlBuilder. *Compact Product Suite. Compact Control Builder AC 800M. Product Guide. Version 6.0.* Available from `abb.com`. Document number: 3BSE041586-600 A. ABB. 2016.

[EH04] Torbjörn Ekman and Görel Hedin. "Rewritable Reference Attributed Grammars". In: *Proceedings of ECOOP 2004*. Vol. 3086. Lecture Notes in Computer Science. Springer-Verlag, 2004.

[EH07b] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[EH07c] Torbjörn Ekman and Görel Hedin. “The Jastadd Extensible Java Compiler”. In: *OOPSLA 2007*. ACM, 2007, pp. 1–18.

[Far86] Rodney Farrow. “Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars”. In: *SIGPLAN ’86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*. Palo Alto, California, United States: ACM, 1986, pp. 85–98.

[FH13] Niklas Fors and Görel Hedin. “Using refactoring techniques for visual editing of hybrid languages”. In: *Proceedings of the 2013 ACM Workshop on Refactoring Tools, WRT@SPLASH 2013, Indianapolis, IN, USA, October 27, 2013*. 2013, pp. 17–20.

[FH14] Niklas Fors and Görel Hedin. “Intercepting dataflow connections in diagrams with inheritance”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2014, pp. 21–24.

[FH15] Niklas Fors and Görel Hedin. “A JastAdd implementation of Oberon-0”. In: *Science of Computer Programming* 114 (2015). {LDTA} (Language Descriptions, Tools, and Applications) Tool Challenge, pp. 74 –84.

[FH16a] 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, Netherlands, 30 October - 4 November, 2016*. In press. 2016.

[FH16b] Niklas Fors and Görel 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. ACM, 2016, 4:1–4:10.

[For+15] Niklas Fors, Gustav Cedersjö, and Görel Hedin. “JavaRAG: A Java Library for Reference Attribute Grammars”. In: *Proceedings of the 14th International Conference on Modularity*. MODULARITY 2015. Fort Collins, CO, USA: ACM, 2015, pp. 55–67.

[Fri10] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)*. 24(3). 2000, pp. 301–317.

[Hed11] Görel Hedin. "An Introductory Tutorial on JastAdd Attribute Grammars". In: *Generative and Transformational Techniques in Software Engineering III*. Vol. 6491. LNCS. Springer, 2011, pp. 166–200.

[HM03] Görel Hedin and Eva Magnusson. "JastAdd: an aspect-oriented compiler construction system". In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[Hed+10] Görel Hedin, Johan Åkesson, and Torbjörn Ekman. "Extending Languages by Leveraging Compilers: from Modelica to Optimica". In: *IEEE Software* (Mar. 2010).

[Hog14] Erik Hogeman. *Extending JastAddJ to Java 8*. Tech. rep. Master's Thesis. LU-CS-EX 2014-14. Sweden: Department of Computer Science, Lund University, 2014.

[Kic+01] Gregor Kiczales et al. "An Overview of AspectJ". In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*. Vol. 2072. LNCS. Springer, 2001, pp. 327–353.

[Knu68] Donald E. Knuth. "Semantics of Context-free Languages". In: *Math. Sys. Theory* 2.2 (1968). Correction: *Math. Sys. Theory* 5(1):95–96, 1971, pp. 127–145.

[MH07] Eva Magnusson and Görel Hedin. "Circular Reference Attributed Grammars - Their Evaluation and Applications". In: *Science of Computer Programming* 68.1 (2007), pp. 21–37.

[Mag+09] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. "Demand-driven evaluation of collection attributes". In: *Automated Software Engineering* 16.2 (2009), pp. 291–322.

[Mar94] K. Marriott. "Constraint multiset grammars". In: *Visual Languages, 1994. Proceedings., IEEE Symposium on*. IEEE. 1994, pp. 118–125.

[Min02] M. Minas. "Concepts and realization of a diagram editor generator based on hypergraph transformation". In: *Science of Computer Programming* 44.2 (2002), pp. 157–180.

[Min06] M. Minas. "Generating meta-model-based freehand editors". In: *Proc. of 3rd Intl. Workshop on Graph Based Tools*. Electronic Communications of the EASST, 2006.

[Mod12] Modelica. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3*. Available from `modelica.org`. Modelica Association. 2012.

[Moo09] Daniel L. Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Trans. Software Eng.* 35.6 (2009), pp. 756–779.

[OMG04] OMG. *Human-Usable Textual Notation (HUTN) Specification Version 1.0*. Available from `www.omg.org/spec/HUTN/`. Object Management Group. 2004.

[ÖH13] Jesper Öqvist and Görel Hedin. "Extending the JastAdd extensible Java compiler to Java 7". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*. 2013, pp. 147–152.

[PH02] Lars Pernebo and Bengt Hansson. "Plug and play in control loop design". In: *Proceedings for Control Meeting*. Linköping, Sweden, 2002.

[SM10] Max Schäfer and Oege de Moor. "Specifying and implementing refactorings". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 2010, pp. 286–301.

[Sch+08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. "Sound and Extensible Renaming for Java". In: *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. Ed. by Gregor Kiczales. ACM Press, 2008.

[Sch+09] Max Schäfer et al. "Stepping Stones over the Refactoring Rubicon". In: *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. 2009, pp. 369–393.

[SK03] Carsten Schmidt and Uwe Kastens. "Implementation of visual languages using pattern-based specifications". In: *Softw., Pract. Exper.* 33.15 (2003), pp. 1471–1505.

[Sch+14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing layered graphs with port constraints". In: *Journal of Visual Languages & Computing* 25.2 (2014), pp. 89–106.

[Slo+13] Anthony M. Sloane, Lennart C.L. Kats, and Eelco Visser. "A pure embedding of attribute grammars". In: *Science of Computer Programming* 78.10 (2013), pp. 1752 –1769.

[SH12] Emma Söderberg and Görel Hedin. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Technical Report 98. LU-CS-TR:2012-249, ISSN 1404-1200. Lund University, Apr. 2012.

[Söd+13] Emma Söderberg et al. "Extensible intraprocedural flow analysis at the abstract syntax tree level". In: *Sci. Comput. Program.* 78.10 (2013), pp. 1809–1827.

[VW+10] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1-2 (2010), pp. 39–54.

[Vog+89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *PLDI*. 1989, pp. 131–145.

[Woo97] Bobby Woolf. “Null object”. In: *Pattern languages of program design 3*. Addison-Wesley. 1997, pp. 5–18.