(empty backward page)

OFB – Basic considerations

Dr. Hartmut Schorrig
www.vishia.org 2024-09-28

Table of Contents

1 Open/Libre Office for Graphical programming

One of the advantages of textual programming is: You can visit your program code with any desired editor, such as Notepad++, or VIM on Linux or just a powerful *Integrated Development Environment*. For development of course, compiler tool suites are necessary. But to discuss content, behavior, look whats happen you need only standard tools. For long time maintenance it means it may be sufficient only to have the source code itself, if maintenance actions can be done by parametrization (with given *Operation and Monitoring* tools), or for update the program you need only the compilation tools or possible use newer versions of compilation tools which are compatible.

If you use graphical programming, then the graphical sources can be viewed often only with the original tools which may be vendor specific, need licenses to use etc. Sometimes older source files cannot be opened with newer (currently in use) versions of the tools. It means only for view what is contained in your device you need a specific tool. Additional often code changes are sophisticated in the tool chain, needs specific knowledge (about set options etc.).

This may be one reason that textual programming is preferred, though for the graphical programming it was rumored also for more as 30 years, it would be replace the textual programming because of some advantages.

That's why graphical programming is the playground for some big tool providers, whereas different approaches are given with the tools which are not compatible. Whereas textual programming is also familiar for common software, sometimes Open Source.

The second reason to favor textual programming is: The sources are immediately comparable with simple text diff tools. And the third reason is: Tools are interchangeable, the source is always understandable as text source.

Now, to favor the graphical programming, this paper offers the idea and shows approaches related with usable software for content evaluation to use a common graphical draw tool for the graphical programming, which is usable for everybody without effort, which is compatible also with some other tools and which is enough powerful to use. For that ***LibreOffice*** and also ***OpenOffice*** was tested to draw the diagrams, and a translator to evaluate the content was written (just in progress). This concept is presented here.

Some basic ideas are:

- Use Style Sheets to designate semantic information to graphical blocks,
- Evaluate it reading information from the odg file, it is a simple zip file containing XML information
- Translate the content to other graphic formats for the specific tool or make the own code generation.

A second approach of this work is: For graphical programming the familiar idea to use Function Block Diagrams (FBD) to present functional content are combined with important features of the UML class diagrams. All in all the Function Blocks (FBlocks) are seen as instances of classes, which is self evident often for code implementation (in C++) but also in C where Object Oriented classes can be implement with `struct` data and the appropriate operations for this data. It means the FBlock Diagrams are advanced with UML features of class diagrams.

And also, UML class diagrams (without the FBlock idea) can be drawn and translated also with this approach.

2 Join FBlock Diagrams and UML-Class Diagrams

The ***Unified Modeling Language*** (UML) was created in the beginning of the 1990th based on different existing modeling approaches, firstly by Grady Booch, Ivar Jacobson and James Rumbaugh wiki. Another contribution to UML comes from David Harel wiki who had development state machine technology firstly introduced with his own tool "*Statemate*" and then applied to the UML tool *Rhapsody* (original from I-Logix, now IBM).

The focus of UML was also code generation for particular devices, but also the approach of commonly describing of systems which can be applied to particular software, with focus of Object Orientation.

In opposite, the technology for ***Function Block Diagrams*** (FBD) inclusively code generation for particular usual firstly automation devices was created already in the 1960th with the IEC 61131 Norm for "*Programmable Logic Controllers*". It was also similar used for some other approaches such as LabVIEW wiki or simulation tools. Especially Simulink from Mathworks wiki is used here for some comparisons with the here shown technology. This tools has its basics in the 1980th but currently further developed and used.

Both approaches, the UML and the FBD tools are designated as "*model driven development*". But there are not related. The FBD tools does not use diagrams from the UML, and it is usual not seen as "Object Oriented" and the UML seems not accept a diagram kind which is firstly for a particular software or device and not for a commonly described system.

Usual the code generation is familiar from the FBD tools. In UML code generation generates only the frames of the classes respectively instances, it is not so frequently used.

The FBD tools focus only to the functional aspect of a device or a software. The operation system and managing to properly run the software (organization of threads, hardware access etc.) is usual done by specific settings (for example the "*hardware config*" part of configuration for automation devices with the Siemens TIA portal). The system itself is hard coded given and does not need an elaborately description presentation.

In opposite, the UML focuses to the whole system. For example the operation system itself is a "*component*", which is presented with interactions etc. in the component diagram. Also some hardware components.

In this manner the here presented combination of the UML Class and the FBlock diagram is only a part of a possible "UML 3.0". It does not replace all basics from UML, of course. It is only a contribution for this imagined UML 3.0.

How to name this combination of a FBlock and Class Diagram ... Let's use the abbreviation ***UFB***. The "*U*" comes from the UML influence, also means *“Unified”*. The diagram, graphical programming is named ***UFBgl*** with *“gl”* as *“graphic language”*. A textual representation of the same content should be named ***FBcL*** as *”Function Block connection Language”*. The focus to the UML is not presented in this abbreviation, but UML is familiar and recognizable.

What else: The **event connection** between FBlocks are also used here as important part. Events are familiar in UML for state machines. An Event connection is also used in FBlock Diagrams with the standard IEC61499 for automation devices as a basically feature. Also in Simulink events are designated and used for "*triggered subsystems*" as well as for state machines. Events should be familiar in Object Orientation.

3 Approaches for the graphic, basic consideration

Table of Contents

This chapter shows how capabilities of ***Open-*** or ***LibreOffice*** are used to draw diagrams.

3.1 Question of sizes and grid snapping in diagram

Commercial tools for graphical programming have often not a proper answers to this question. Often sizes are scalable in any kind, as the user want to have. Grid snapping is sometimes supported or not, and, sometimes sophisticated algorithm are implemented which avoids lines through blocks and make instead mad ways around all blocks. LibreOffice is here more friendly, it let the user decide about the connection path. This may be only a marginalia.

Let’s think about font sizes and grid, requirements:

- In a usual document a proper font size is 9..11 pt, this document uses 9 pt but for A5 page format. A smaller font (pt, 6 pt) is not suitable for reading because of the recognizability of the words and their contexts, it is only for read the package leaflet of medical products.
- A diagram should have place in a document on a A4 or size-B page (~ 18 cm text width). It means the size of a proper view is **~18 x 10..12 cm**. Using a whole side in landscape orientation may have a size of 25 x 17 cm, but in landscape mode the document must be rotated only for this page, this is not suitable for reading a PDF document on the screen.
- A diagram has two tasks:

  a)\t Documentation

  b)\t Base for the software

For the approach b) the diagram may be well editable as a whole on a large screen, for example with resolution 2650 x 1200 pixel. To document this complex diagram it can be shown in landscape orientation in a document, or better: It should be reduced in size to fit on a normal page in portrait format. Details are then no longer legible, but important things and orientation should be shown in larger font. Then the overview can be explained and details can be shown as part from exact the same diagram in a higher resolution.

- A common and contradictory question for diagrams is: How comprehensive should it be. Should it contain only one block and some less aggregated ones? Or should it contain the whole truth of a module? The answer of this question depends on the available size for presentation. There should not be to less content.

The UML has the advantage that you can use more as one class diagrams to explain the same class in different contexts. That is a very great advantage and it should be usable also for some Function Block presentations! (Not yet in professional tools). This helps to decide how many content a diagram should contain.

- The readability of a word which is isolated of a sentence, an identifier of a block or line or such one is given also with a smaller font size than 11 pt, especially if it is present in bold font or maybe also in a non proportional font (as for programming language source code). Because in proportional fonts often important small characters such as “il” are to small and bad visible

- For positioning a proper grid size and the **possibility of positioning with cursor keys (!)** is essential. LibreOffice has the property that the step size for the cursor key is anytime 1 mm, independent of other settings. It's possible use cursor keys for fine positioning (Alt-Cursor...) but this is too fine.

There is a specific property of LibreOffice: The step width by moving with cursor keys is normally 1 mm. You can do fine adjusting in combination with the Alt-key, but this is too fine. If also a grid fine spacing with snap points of 1 mm is selected (a 5 mm grid with 5 fine divisions), then the placing is very proper. All elements are placed in a 1 mm grid, the 1 mm is enough fine for details and enough raw to simple snap in the grid points.

From that, the idea comes to have a standard size of small elements of 2 mm. The mid point is also in 1 mm grid snapping raster. You can have a near distance of lines of 1 mm, well obviously.

To show enough content in a diagram you may use an A3 paper in landscape orientation. On a larger monitor (2560 or 3280 pixel width) it is editable in entire page mode. The diagram has a width of ~40 cm. 1 mm space is ~ 6 pixel on the screen.

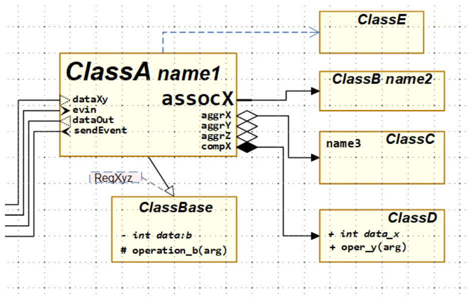


Figure 1: View A4-width as Part (280 DPI)

If you present the whole diagram in a document in portrait format, it is demagnified to ~ 17..18 cm, it means ~40%. As you see right side, the name of `ClassA` is readable, also the "`assocX`" with a font size of 10 pt Consolas bold in the original. Here it is presented with ~ 4 pt because of the demagnification. The others or not readable, but you can recognize the aggregations, compositions and associations. The symbols may be obviously though they have a size of only 0.8 mm height.

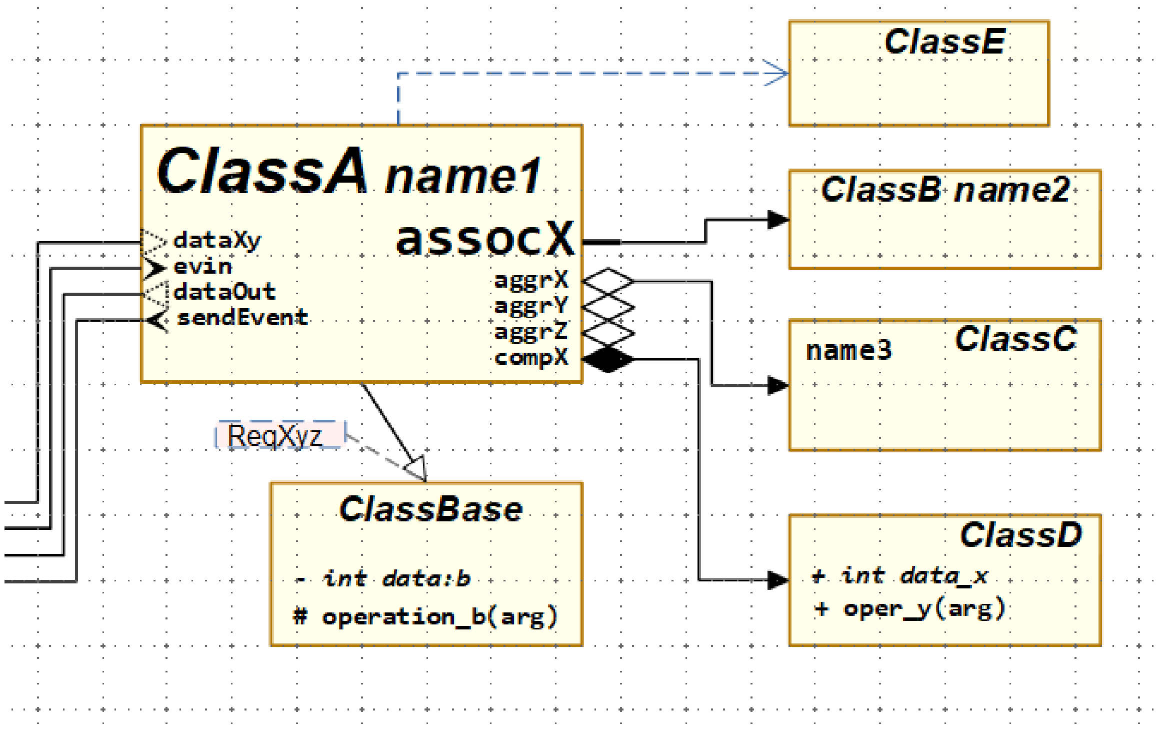


Figure 2: View in original size if this document is displayed with 2 pages on screen (112 DPI)

The same content is presented here right side in original magnification. The font size of 6 pt for the most elements is just readable. It is Consolas bold. The type names of the classes are Arial 8 pt, the name of ClassA is Arial 14 pt. This is a 1:1 presentation, drawn in portrait A4 it is really 1/1 site width.

It means you can have an overview, but you don't see some details in the documentation. Parts of the same diagram can be shown in original size, then all is readable.

You should place different approaches of the same module in more as one diagram. This is definitely supported by UML, and should also be usable for function block presentations. In commercial tools such as Simulink it is not possible, but here it is.

As living example look on the following Class-Object-diagram:

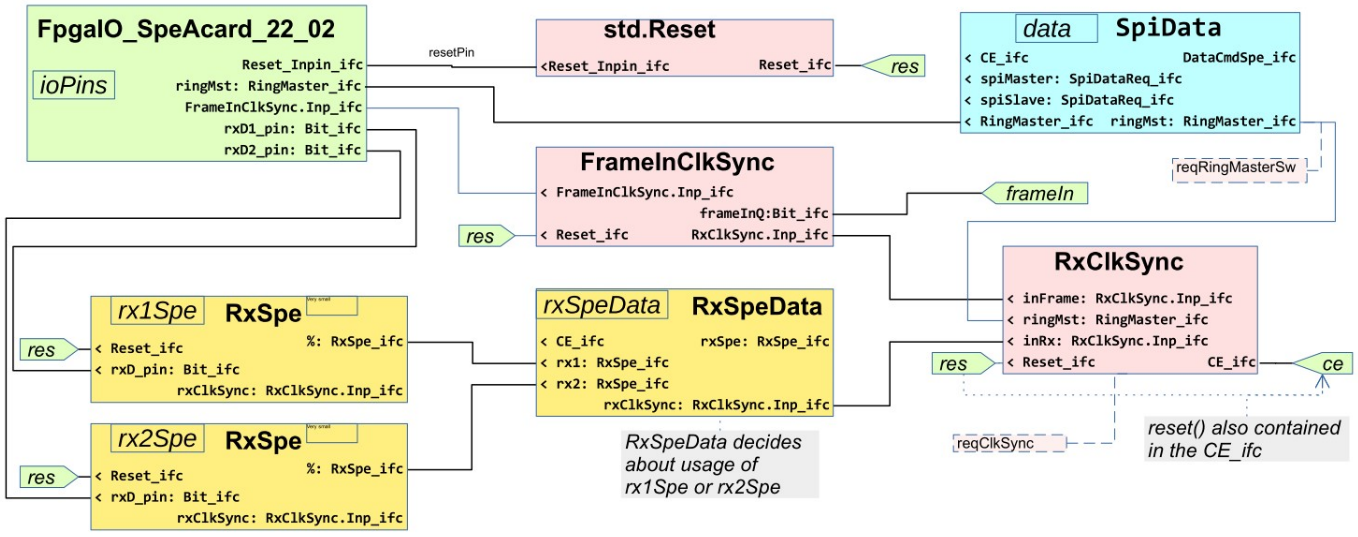


Figure 3: Example for a Module Diagram

This diagram should be well readable in normal view of a pdf viewer. The font and size of the names is consolas 6 pt bold. The original draw area is the width of a A4 page. The pixel solution is 1351 x 480, results from a Zoom of 200 % on a 1980 pixel width monitor.

The diagram shows a coherence of different blocks to build a synchronized *clock enable* (ce) in a FPGA. You see two receiver (Rx) modules, which are combined with a third module, with equal light-brown colors. Its a selection of the active input. The output of this third module has the same interface type `RxClkSync.Inp_ifc` as the module in the mid. Both are selected from the red right module. With less explanations the coherence should be understandable.

3.2 Using figures with styles (indirect formatted) for element

The first used is a rectangle shape which presents a class or Function Block (FBlock). The rectangle should be marked with the style for indirect formatting ofbClass or also ofbFBlock. This formatting style results in a predefined appearance (color, line width, text font etc.). But not the appearance determines the kind of the shape, **the name of the style defines its semantic**.

With given indirect formatting style, you can modify the appearance with additional direct formatting, for example change the color of the shape. You can also define your own style. If this style starts with the identifier of the semantic defining style, followed by a “-” and then your own name, it works proper. This may be interesting for specific solutions, showing a special type of shapes only in appearance, which are all of the same kind.

For possible styles of FBlock shapes see ***Error: Reference source not found Error: Reference source not found*** on page ***Error: Reference source not found***

From view of UML class diagrams:

A class or FBlock should have a name and a type designation. This can be written either as text in the FBlock (class) shape, as also in an extra shape `ofnClassObjName` for more free positioning. The text of the `ofbFBock` is positioned right top in the shape area. *Maybe press ctrl-M to remove other automatic formatting informations.*

The original UML class diagram has the following approach:

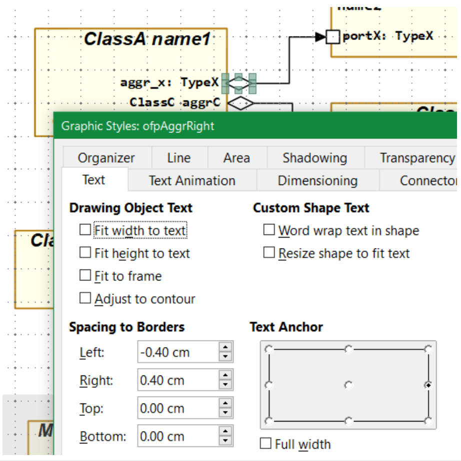
- A class is a rectangle box containing the type name of the class.
- Some data or operations may be named inside the class box, it does not need to be completely.
- All relations to other classes are shown with references to the other classes. This references are often non directed, but sometimes only in a specific direction marked with a little arrow on end. This relations are associations, aggregations, compositions, inheritance, dependencies.

The last point is not mapped to the languages which presents the software which is presented by the UML diagrams. Because: The fact that a class has an aggregation to any other class is a property of the class, and not a property of relations between the classes. It is exactly the same as for data. A data element has a type, and a reference has also a type, the type (or super/basic type) of the referenced class. The name and type of a reference is a property of the class, it is not a property of the relation between the classes.

For that reason the shown relations to other classes are assigned to the class itself. They are existing also if there is no connection. Then, of course in the implementation it's a null or nil pointer. Or it is just not shown here in this diagram, instead shown in another diagram, but nevertheless it is an element of the class. Look on the images on the page before. There are some not connected aggregations, which may have a meaning on explanation to the diagram.

The pin contains a text, which is the identifier for the pin and can also contain a type specification, a constant value or also a connection information. The text is written outside left or right from the small pin shape by using the LibreOffice property, that a text can exceed the bounds of the element's graphic. More as that, the left or right margin of the text is set to a value greater or equal the size of the element, and in this kind the text is written outside, left or right next to the element. If you want to have a little more distance, you can also insert spaces left or right of the text. The spaces are removed while evaluation of the text.

Why it is necessary in LibreOffice to set the “Left” value to the negative “Right” value, or also to a higher negative *value, otherwise it does not work. It is not consequential. Second, In an older version of LibreOffice it was possible that the Distance value (here “Right”) can be greater than the size of the element, to insert a small space right of the shape. From Version ~6.4 this was no more possible, unfortunately. That should be small questions to the LibreOffice community.*


Graphic Styles: ofpAggrRight

Organizer | Line | Area | Shadowing | Transparency
Text | Text Animation | Dimensioning | Connector

Drawing Object Text
- ☐ Fit width to text
- ☐ Fit height to text
- ☐ Fit to frame
- ☐ Adjust to contour

Custom Shape Text
- ☐ Word wrap text in shape
- ☐ Resize shape to fit text

Spacing to Borders

Left:	-0.40 cm
Right:	0.40 cm
Top:	0.00 cm
Bottom:	0.00 cm

Text Anchor
- ☐ Full width

Figure 4: Style_ofpAggrRight_TextProp.png

The pin for connection to the class or FBlock is shown as this small shape or figure. However, it is not the shape itself that marks the shape as pin for code generation, the associated style sheet is the essential one. The look of the figure can be changed if desired, it is for human. But **the style sheet marks the semantic of the figure, the kind of the element.** The settings in the style sheet, especially the size of the text, can be overridden by direct formatting. This is for larger fonts explained in the chapter before and shown in page . Also the settings in the style sheet can be changed for centralized approach. The name of the style sheet is the important one.

Style sheets are a proven concept for text writing. The direct formatting approach can be also used to a style sheet formatting approach, and both can be combined. A style sheet allows change a formatting style for all designated elements (paragraphs, parts of text etc.) to achieve a uniform presentation. It is an advantage that is often not enough known. That's for 3.3

Pinscommon explanations.

3.3 Pins

An input or output of a Function Block (FBlock) is named ***Pin of the FBlock*** in the UFBgl. Hence on the pins connections between the FBlocks are connected, using connectors in LibreOffice, see next chapter.

But some connections are connected also to the whole FBlock, for example as destination for an aggregation. But this builds also a pin in the internal data map.

The pins are either simple small figures with a fixed size, known from UML as the diamond (filled / non filled) for Composition and Aggregation, or adequate forms for events and data, or they are simple text fields. The pin appearance does not play any role for the interpretation and converting of the graphic, but may be proper for manual view. For interpretation the associated style (indirect formatting) is essential. The style determines the kind of the pin.

The first idea for UFBgl was, using a common pin style which is proper for appearance, and defining several styles for the connection kinds between pins (aggregation, composition, data or event flow etc). This idea comes, because the end point of connectors can define in a UML-conform and interesting way, not only with an arrow left or right. Then the connector style would determine the pin kind. But this idea is worse, because pins should be well defined also in non connected states, for example for association of event and data pins. They should show the capability of a FBlock. Hence it is better to have different styles which determine the kind of the pin. The connector style (see next chapter, and on page

Hence, the sometimes existing `ofRef…` or `ofc…` styles should not be used for content semantic, only for appearance. All connection styles (except a few special cases) are the same for functionality, only different in appearance.

For the pins the simplest variant is, have a text field with the associated style.

- texxt

3.4 Connectors of LibreOffice for References between classe

The connectors as known from LibreOffice are the proper possibility to connect FBlocks or classes. The connection can be done between pins of the FBlock, or also from/to the FBlock itself.

You can use connectors with orthogonal lines, or straight or curve connectors as if you want.

LibreOffice assigns four connection points ("glue points") to each element by itself. This is sufficient for the pins. It is very simple to connect for example the end point of a diamond of an aggregation with the mid of a port as destination of the aggregation, or also with any other class if the whole class is referenced.

For the larger class block with maybe more connections on different positions you can add some more glue points.

Using connectors between elements in your graphic, the connection remains stable if you move some blocks. You may adjust the inflection points (more precise the mid points between inflection). Some commercial tools such as Simulink try to adjust connections between blocks by itself by sophisticated algorithm, which should avoid lines crossing blocks, and make instead mad ways around all blocks only to avoid crossing a free but reserved area for a name of a block. LibreOffice is here more friendly, it does nothing by itself, only move the connection as necessary, and let the user decide about the outfit of the connection path.

A connector as reference between blocks should have also a Style. If the connected elements are well dedicated by Style Sheets, you can use the `ofRef` style for all connectors. It produces a small arrow on the end, and a line width of 0.2 mm, no more.

But there is also a possibility using connectors as in UML. The connectors have especially the start arrow outfit as in UML necessary (diamond for aggregation). Then you can use for the connected elements the common style `ofPinLeft` or `ofPinRight` which does not specify the kind of the element. The connector specifies it. That is the originally approach of UML, also possible here (but not recommended). Both are supported by code generation.

3.5 Connect Points for more complex reference

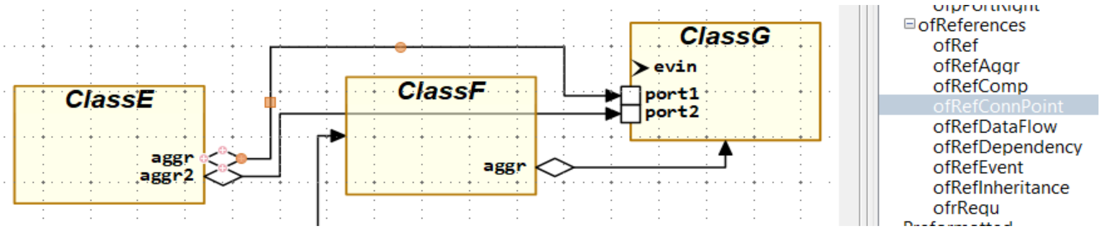


Figure 5: ReferenceLineCrossesBlock.png

LibreOffice seems to be have the disadvantage that additional inflection points on orthogonal connectors are not possible. Look on the example left side. The connection from `aggr2` to `port2` through ClassF is not nice.

The solution is shown also in this mage. From `aggr1` to `port1` two connection lines are concatenated. The first line is of type (style) `ofrConnPoint`, its without arrow on end. Both lines together appears as one line, with proper inflection points.

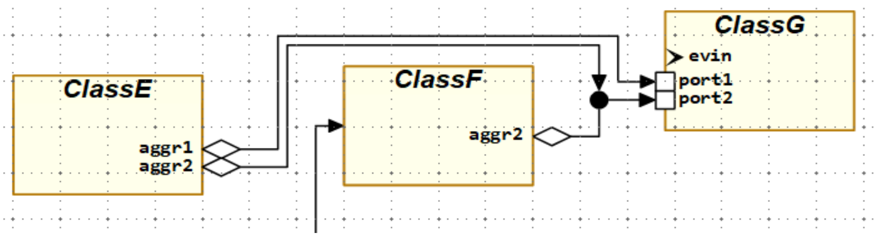


Figure 6: OFB/ConnPoint.png

Another question is: Having aggregations or other references with one destination and more sources. In UML often there are drawn parallel. But it is more consequently to use a connection point as it is known from any electrical circuit scheme and also from Function Block Diagrams for data flow. The difference is only: Data flow and electrical schemes has one source and more destination. An aggregation has one destination and can have more sources. The reference line to the connection point is either a simple `ofRef` which has an arrow on its end, or it is the same as in the image above for concatenation of reference lines, with style or type `ofrConnPoint`.

3.6 Diagrams with cross reference Xref

The cross reference or usual nominated as Xref is an often used symbol to replace too much lines in one graphic, or also to make connections to several sheets of a graphic. The last one should not be in focus here, because on graphic sheet presents one aspect, spread one diagram over several sheets is not familiar for UML or also Function Block Diagrams.

You may use a Xref for signals and connections, which are well known from name, and which have basically connection meanings (such as "reset") and may be connected to more as one block.

- The figure for the Xref can have any form, but should use the given form (copy it from template). The Style Sheet should be either `ofbXrefLeft` or `ofbXrefRight`, whereby the difference is only the text alignment to left or right.

- The name in the Xref symbol should be a mnemonic name for the functionality, valid for this diagram. Here it is a combination of the type of the port and part of name, maybe proper.

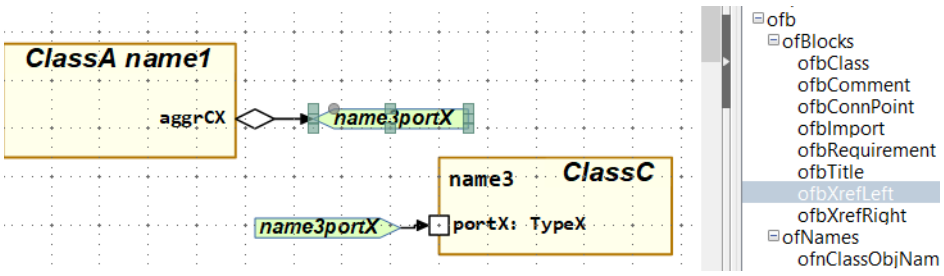


Figure 7: UMLdiagramXrefExample.png Cross Reference usage

- The line from a block to the Xref should be the same type (here a simple `ofRef`) as without Xref.

- The line from the Xref to the block should have usual the same type, but this is not evaluated. Because the type of connection can be also composition or association here, the type for the association is used here, it is not specificated to the aggregation or composition with the filled or non filled diamond.

You can use Xref connections for all line types. The evaluation of the graphic builds a list for all Xref by name per sheet, and checks the connections.

4 Capabilities and concepts of OFB diagrams

Table of Contents

4.1 Graphic Blocks, pins and text fields inside a GBlock

The diagram contains primary Graphic Blocks (GBlock) which are associated to one of the style `ofb…`. This GBlocks should not overlap, should have a well distance each other.

Secondly the graphic consists of ***pins***, which are part of a GBlock. Pins are associated with a style `ofp…` or only `ofPin`. The pins should be associated to a GBlock. This is done via its positions. At least a pin should have one coordiante (left, right, top, bottom) inside the GBlock area, then it is associated to the GBlock. The pins can jut out a little from the GBlock so that the connection points are properly visible.

Third, the GBlock can contain text fields, also possible a little bit jut out, but usual inside the GBlock, with a style `ofn…`. It is for the name and type of a `ofbFBlock` or also for some attributes and operations as known in UML.

See ***Error: Reference source not found Error: Reference source not found*** page ***Error: Reference source not found*** and ***Error: Reference source not found Error: Reference source not found*** page ***Error: Reference source not found***

4.2 Show same FBlocks multiple times in different perspective

There is an interesting and important principle using in UML class diagrams. A class can be presented in more as one perspective in several diagrams, and also more as one time in one diagram. The class is presented by its name, it is also able to find it in the repository of the UML data. The diagrams plays only the role of presentation of the class with its properties just in several perspective.

In opposite, traditional Function Block Diagrams shows one FBlock as one instance. Often the FBlock does not need a specific name, then it is automatically named

The ***UFBgl*** approach uses the principle, showing also a FBlock in several perspectives, in opposite to traditional FBlock diagrams, but similar as UML. It means, a FBlock as one instance can be shown more as one time in the same diagram or in several pages of the same module also in several files. The FBlock is dedicated by its instance name with a type or by its type name. Drawing a second FBlock with the same name is the same instance. All FBlocks with the same type describes this type in sum.

This principle enables showing complex large FBlocks in several perspectives. Different connections are shown on different places, also the same connection can be shown more as one. For example inputs of one functionality of a FBlock are shown on one page with focus of that input signals, other input signals are shown on a second page, and the output connections and processing are shown on a third one. Also the connections are unique dedicated by its pin name on the named FBlock with the named type. This offers more overview. The dispersion of one FBlock connectivity in several views may be seen as disadvantage, it becomes confusing. But notice, there are search operations and evaluations of the graphic which gives an overview to find all locations of the same FBlock instance. The idea is newly for FBlock diagrams, look for its advantage.

Now this idea is also usable for the class description idea: Any FBlock instance is dedicated by its type. The type is the class type. All occurrences of the same type of Flocks are properties of its class. Also FBlock with only the type name, without instance name presents the class properties. The sum of all is the property. This is true for the type of a c FBlock which is a class as also for the connectivity of an instance of a FBlock in several graphic presentations.

Look for example to . The FBlock with name `h3p` is assigned to the type `BpParam`, left bottom. But this block is drawn twice, the second is magenta, has not the type identification because the name is unique, and shows the instance with another event input `ctorObj` and some other data. This is another functionality associated to this same instance, and also to the same class.

4.3 More as one page for the FBlock or class diagram

The chapter above ***4.2 Show same FBlocks multiple times in different perspective*** allows simple to disperse a diagram over a lot of pages (as necessary) because the same FBlock instance can be shown for example with its input signal wiring, and on another page with its output signals, or group of signals. This allows formally descriptions more near to explanations. One Image (one side) should present one aspect. Which – this is document- or explanation oriented. Data flow connections can also be joined by Xref blocks.

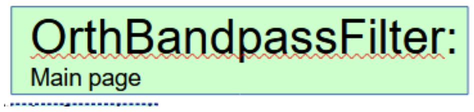


Figure 8: ofbTitle-1.png

Any page need have a title block, of style `ofbTitle`. It contains the name of the module and a short text what it contains.

The pages can contain several modules. The association of module diagrams to files.odg is an important topic. If you have related modules, you can store all it in one file. On the other hand it is possible to have more as one file for one module. This should only be regarded while translation the module.

4.4 Function Block and class diagram thinking in one diagram

One of the basic ideas of the UFGgl approach is just, join UML thinking and FBlock thinking. UML presents in class diagrams relations between classes. A class is an abstraction of implementation. The implementation uses instances (of classes).

In opposite, ordinary Function Block Diagrams only work with instances. A "class" is an unused word in this way of thinking. But in fact, using a Function Block type from a Library is *“instantiation of a class”*, the library block type is the class.

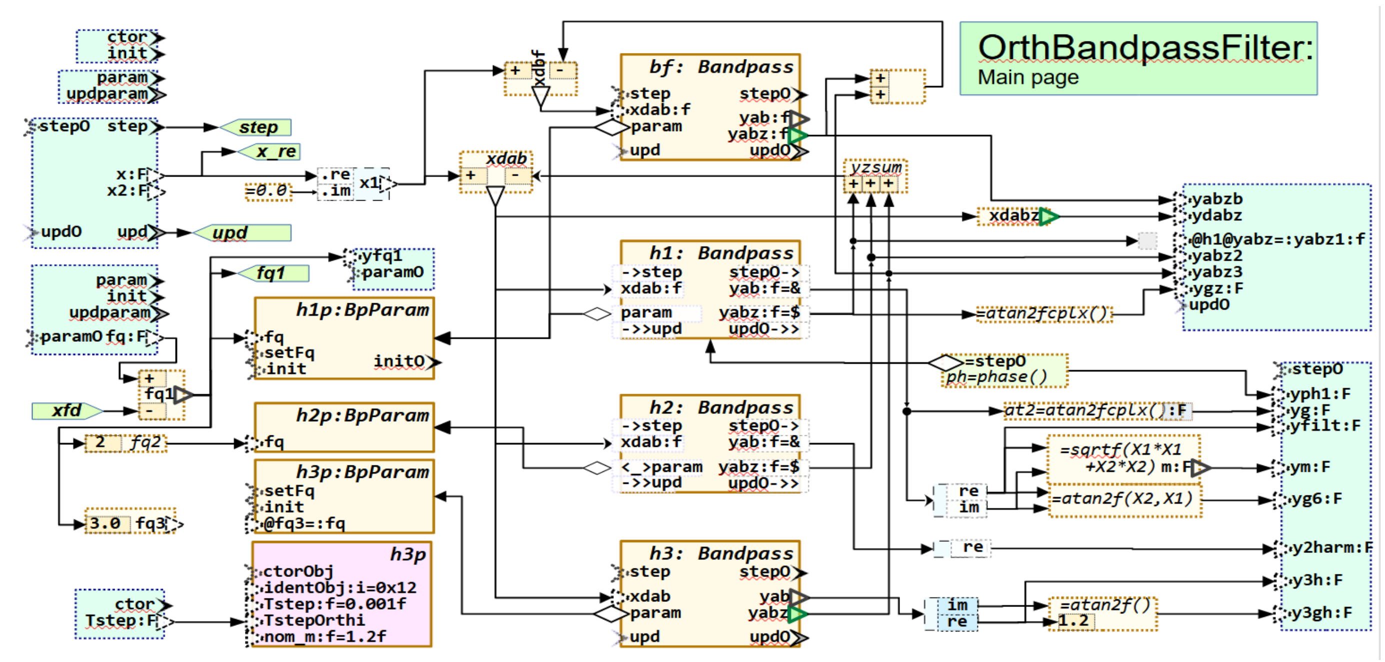


Figure 9: OrthBandpassFilter.odg.png

Error: Reference source not found shows primary a Function Block Diagram (FBlock diagram). The green parts are the input and output pins of the module. Some FBlocks presents expressions, these are with dashed lines. The other FBlocks presents instances (each three from the same type) which are connected with data flow.

But from the `Bandpass` FBlocks to the `BpParam` FBlocks there are aggregations. That shows two things:

a) There is an aggregation from the type (class) `Bandpass` to the class `BpParam`. This is a relation of a class diagram.

b) The aggregation from `bf` and `h1` is initialized to refer `h1p`, as also `h2` refers `h2p` and `h3` refers `h3p`. This is a property of the FBlock instances.

The relation shown with the aggregation can be seen also as data flow, but in the opposite direction. Initially the address of the `h1p` FBlock is provided to the `bf` and `h1` FBlock, to refer it, adequate for `h2` and `h3`. Hence, the diagram contains information about class (or type) relations as class diagram and information about instance relations as Function Block Diagram with data flow.

The combination in thinking of FBlock instances, its type (the class) and several operations, here presented by the several events is a kind of ObjectOriented thinking. The “Object” is the instance of a well defined type, the type (class) has some properties valid for all Objects of this type, and it has operations.

The last one aspect, given operations, is also shown in the green block right mid with `phase():F`. This is a shape of style `ofbExpression` but with an aggregation. It means the expression aggregates a FBlock instance, which are the data for the given operation in the expression, and hence the operation is associated to the data type, it is an Object Orientated operation (or method as often named). The second specifity is, this operation should not have side effects, it does not change data in the aggregated object, because it is designated as expression term. This is an important feature of ***Functional Programming***, and unfortunately not so much considered in Object Orientation, but important. In C++ implementation this is an operation ending with `const` after the closing parenthesis if the function definition line:

```
float Bandpass::phase() const {...}
```

but for example in Java it has not a proper counterpart, Java does not know a designation for const operations, unfortunately. (It is not the final keyword!).

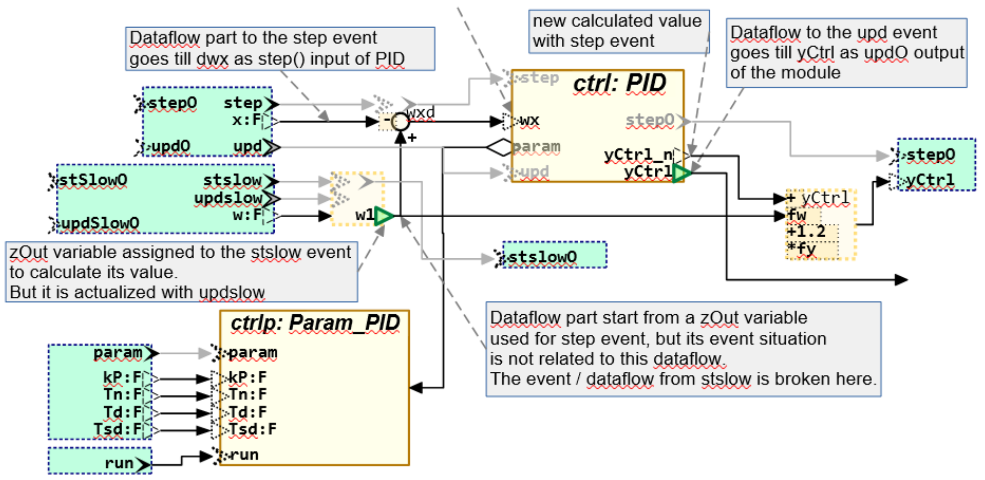
In opposite, operations which change data should be present as FBlock with the adequate event. The event characters the operation, as shown on all FBlocks, especially the three different operations shown in two FBlocks `h3p` left bottom. Note that `setFq(float fq)` and `init(float fq)` are defined in the same FBlock, only possible in combination with init.

4.5 Using events instead sample times in FBlock diagrams

Usual for FBlock diagrams sample times are familiar. It follows from the basic approach that the FBlock connections are executed cyclically. That is so in some applications, for example industrial automation control. But sometimes events also play a role. In ordinary automation control often this is regarded by polling (quest of input signals) in a cyclically kind, because their basic operation system supports firstly cycles. The importance of events was often not the focus when such systems were created, although events were common and well-known in other areas of software technology. For example Simulink works basically with *“sample times”* but has specific opportunities (*“triggered subsystem”*) to deal with events.

Well, the assignment of signals and FBlocks to events ***includes working with sampling times***, but also triggered operations. More as that, the ***event flow presents*** better as a data flow the ***execution order of FBlocks***. Only using the data flow sometimes it is not well as necessary predicted. If the execution order is internal information (the user does not see it unless you study the generated source code), then uncertainties remain.

The UFBgl tool allows the automatic derivation of the event flow from the data connections (data flow). The event flow is shown in the textual representation of the graphic and can be viewed or analyzed. It is also possible to determine a specific event connection in the graphic by the user.


Dataflow part to the step event goes till dwx as step() input of PID

new calculated value with step event

Dataflow to the upd event goes till yCtrl as updO output of the module

zOut variable assigned to the stslow event to calculate its value. But it is actualized with updslow

Dataflow part start from a zOut variable used for step event, but its event situation is not related to this dataflow. The event / dataflow from stslow is broken here.

Figure 10: OFB/DataFlowPID4.png

The ***Error: Reference source not found Error: Reference source not found*** is an example primary as Function Block diagram with a ***data flow***. The ***event flow*** shown in gray is not necessary to be drawn. Here it is only shown in gray what is automatically generated. But the ***event pins*** should be determined as shown (***drawn black***). With the given event pins the data are related to the events, instead to *“sample times”*. Here the `x` ist related to `step`, and the `w` to `stepslow`. The *reference value* `w` comes from another sample time or just with another event. The data flow from `x` to the output `yCtrl` is given, hence `yCtrl` is related to the step ***event chain*** and it is delivered with the `stepO` output event. The value stored in the `w1` variable is a *“state value”* set with the `stepSlow`event and only used, similar as after a *“Rate Transition”* in Simulink.

But this image has also an ***Aggregation*** from the `PID` controller FBlock to its Parameter FBlock. This is ***UML***. In Runtime, the address of the parameter instance is delivered to the `ctrl: PID` one time on initializing the system. It means that is a ***data flow*** from `ctrlp_Param_PID` to `ctrl: PID` revers to the aggregation line.

The green blocks of style `ofbMdlPins` are responsible to determine the module pins from/to outer or just the type of the module. Each `ofbMdlPins` block is responsible to associate event-data relations (as also familiar in IEC61499 diagrams), but additionally the update pin is also associated here:

It means that the input variable `x` is bind to the input event `step`. It presents the `step()` operation (should be called cyclically in the step or sample time). Because the `x` is forwarded by data flow to the `ctrl: PID`, also the event `step` is forwarded. Due to the interface definition of the `PID` type the input `dwx` is associated to the `PID` event input `step`. Hence the data flow `x` → `ctrl.dwx` determines also an event flow from `step` → `ctrl.step`.

The role of “*update*” comes from the mealy and moore automate thinking for logic and it is also familiar in numeric solutions for control: All values are first prepared. Preparation uses always the values from the step time before (or in binary logic preparation of D inputs of Flipflops uses only values of the Q outputs of the clock cycle before). That is the ordinary role of the step event.

The update event now realizes the switch of all state values (or clock for Q in Flipflop logic) from the old to the current step to use for the next step. In a sample or step time of a controlling logic first all modules executes the prepare event which is here named `step`. If all parts have been prepared, then the update comes. This assures exactly working for solutions of differential equations and typically for controller theory, it is the Euler principle for numerical integration.

A FBlock can also propagate output values with the prepare event, it depends from the functionality. In Simulink as similar solution an input of an S-Function can be designated as `ssSetInputPortDirectFeedThrough(port,1)` if it influences an output or not (set to 0, default).

In this example shown the output `y.ctrl` is set newly with the `ctrl.upd` event. Hence an event connection between `ctrl.upd` and `upd` of the module accompanies the data flow from `ctrl.y` to the modules `yCtrl` output. The relation between `step`, `stepO`, `upd`, `updO`in the PID FBlock type is clarified by the class definition of `PID`.

Next you see a code snippet of the textual representation of this module in IEC61499, see next chapter:

```
FUNCTION_BLOCK CtrlExample
EVENT_INPUT
  param WITH Td, Tn, Tsd, kP;
  run;

  stslow WITH w;
  ...
END_EVENT
EVENT_OUTPUT
  stepO WITH yCtrl;
  ...
VAR_INPUT
  Td : REAL;
  Tn : REAL;
  ...
VAR_OUTPUT
  yCtrl : REAL;
END_VAR
FBS
  ctrl : PIDf_Ctrl_emC;
  ctrlp : Param_PID;
  w1 : Expr_FBUMLgl( expr:='+;;' );
  wxd : Expr_FBUMLgl( expr:='-+;;' );
  yCtrl : Expr_FBUMLgl( expr:='+; ...
END_FBS
EVENT_CONNECTIONS
  run TO ctrlp.run;
  stslow TO w1.prep;
  updslow TO w1.upd;
  step TO wxd.prep;
END_CONNECTIONS
DATA_CONNECTIONS
  Td TO ctrlp.Td; (*dtype: F *)
  Tn TO ctrlp.Tn; (*dtype: F *)
```

4.6 Storing the textual representation of UFBgl in IEC61499

It is interesting and promising that the widely proven FBlock programming in the IEC61131 standard for industrial automation control (tools such as Siemens Simatic *FBD in TIA-Portal* or Beckhoff *Codesys*) has been further developed to the IEC61499 standard. This development was started in ~2006, Also Siemens was one of the driver in that time. The IEC61131 is used since many years for automation programming. The IEC61499 is standardized and used, but not from the global meaningful players, they only monitors this development. The reason (in my mind and experience) is not disadvantages of IEC61499, it is more a too widely usage, supporting and maintenance of the long term existing IEC61131.

The IEC61499 has introduced an event coupling between function blocks. This determines the stepping order better than the ordinary net lists in IEC61131, but it allows also to distribute the implementation of one Function Block Diagram over several automation stations. Event connections between distant stations forces automatically network communication implementation and assures the correct order of execution in the dispersed station, without additional effort. That's the advantage for automation programming. But the more universal character of event coupling inclusively state machine thinking can also basically used for embedded control programming.

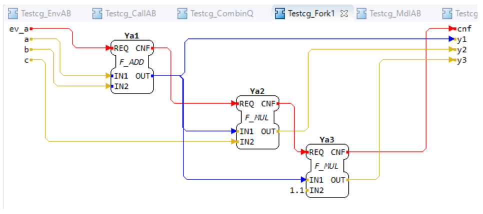


Figure 11: 4diac/Testcg_Fork1.png

A chain of events in the same implementation platform (same thread in a CPU) defines a statement order. Different event chains are related to operations, which can be called either cyclically (for step time driven thinks) of also from the state behavior or independent for example on user accesses.

But the drawing of the event connections in a IEC61499 diagram is an additional effort. The image shows an example with event coupling for simple data relations with the graphical edition tool 4diac. In most cases an event flow (chain) is also determined by the data flow. Evaluation of the data flow results in an event connection, which should not be drawn manually. It is automatically detected during the evaluation of the graphic, and stored in the data model. Only if dedicated event relations are necessary, the events should be drawn in graphic.

The IEC61499 standard is used to store the content of UFBgl diagrams in textual form. This allows also a proper comparability if details in the diagrams are changed. That is a high importance to use this tooling in the development of software, a proper traceability of changes is necessary. With pure graphics, this is often not properly supported, one of the reasons for the still widespread use of textual programming.

It is also possible to read this stored IEC61499 textual files for processing for sub modules, and for code generations, as well as reading IEC61499 fbd files from other tools to merge here.

4.7 Source code generation from the graphic

As is usual with some FBlock graphics, code generation from the graphic is a prerequisite for being able to work productively with it. This chapter should only give an overview. Refer for more opportunities in chapter ToDO

The evaluation of the graphic is done with a Java command line process as (shortened)

```
java -cp tools/vishiaBase.jar;
 … tools/vishiaFBcL.jar
 … org.vishia.fbcl.Ufbconv
 … -dirGenSrc:src/UFBglExmpl/cpp/genSrc
 … src/UFBglExmpl/odg/OrthBandpassFilter.odg
```

This reads the graphic, writes anyway a IEC61499 fbd file, and writes here C-language header and implementing code.

The graphic is shown (as part, one page) in ***Error: Reference source not found Error: Reference source not found***. The generated code looks like (shortened)

```
/**Generated by org.vishia.fbcl.
made by ...
#ifndef HGUARD_OrthBandpassFilter
#define HGUARD_OrthBandpassFilter
#include <emC/Ctrl/OrthBandpass_Ctrl_emC.h>

typedef struct OrthBandpassFilter_T {
  struct { // Locale struct for all din
    float x; // OrthBandpassFilter.x
    float x2;
    float fq;
  } din;

  struct { // Locale struct for all dout
    bool initOk;
    ...
  } dout;

  float_complex xdab; // Expression xdab

  OrthBandpassF_Ctrl_emC_s h1; // h1
  Param_OrthBandpassF_Ctrl_emC_s h1p; // h1p
  OrthBandpassF_Ctrl_emC_s h2; // h2
  ...

} OrthBandpassFilter_s;
```

```
void step_OrthBandpassFilter ( );

void upd_OrthBandpassFilter ( );
...
#endif
```

The implementation file is generated as:

```
/**Operation step(...)
 */

void step_OrthBandpassFilter
( OrthBandpassFilter_s* thiz
                             , float x, float
x2 ) {
  // --> x1.prep otx:evChainExprSetvar
  float_complex x1;
  x1.re = x; // Y D otx:evChainExprSetvar
  x1.im = 0; // Y D otx:evChainExprSetvar
  ...
  thiz->xdab.re = ( x1.re - ( thiz->h1.ya ...
  thiz->xdab.im = ( x1.im - ( thiz-
>h1.yabz.im
   + thiz->h3.yabz.im));
  step_OrthBandpassF_Ctrl_emC(&thiz->h1,
                             thiz->xdab);
  ...
```

There are some stuff which is regarded beside the event flow and hence the execution order. The types of all elements are forward and backward propagated. For the here used complex data types the operations are duplicated respectively specific functions are created, and so on.

The code generation is controlled by textual template files using the java class OutTextPreparer, see

Any user can proved its own templates for code generation, can copy the originals and modify, or can write its own template for other languages or only specific style guides. For pure C language an object oriented style is used of course to represent the instances of classes. classes are presented by struct { } with its associated operations with a thiz reference to the own struct. This can be encapsulated also by C++.

4.8 Run and Test and Versioning

Only yet minutes:

- Compilation in a PC platform (Visual Studio, Eclipse CDT, ...
- Environment for running in C/++ as given (familiar for C development)
- Physical simulations cannot be done, maybe as future development.
- But coupling with another Simulation tool for physics is very recommended, use your own tool. Can bei Simulink, Modelica, or what ever.
- The coupling should be always possible with shared memory on the same PC. For Simulink such an SharedMem Sfunction block, configurable due to a header file on the counterpart, is existing since ~2021, aks me. Should be documented also here.

Versioning:

- Store the odg graphic
- Store the IEC61499 textual representation for compare which changes.
- Store the generated sources in the target language “Secondary Sources”.

One of the important capabilities is the generation of code in a proper target language. The other approach is: storing the graphic in a unique proper readable textual representation. The advantage of that is: The content of the graphic is comparable between progress of development (versions). Whereby not the graphic appearance is in focus (better seen in original graphic), but the content for functionality and code generation.

To have an overview look on the following image:

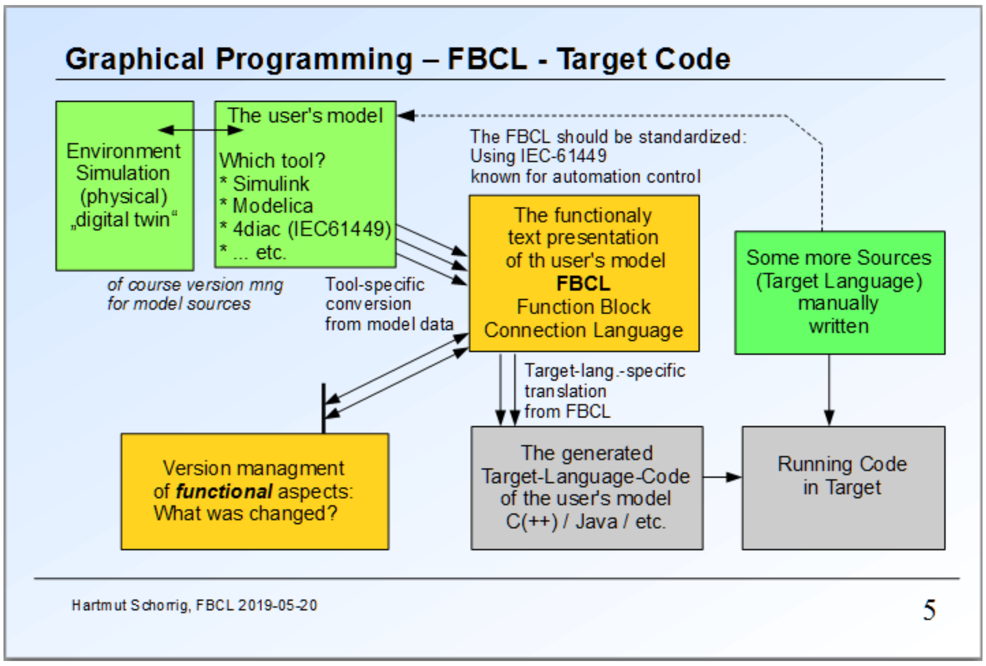

Graphical Programming – FBCL - Target Code
Environment Simulation (physical) „digital twin“
The user's model
Which tool?
* Simulink
* Modelica
* 4diac (IEC61449)
* ... etc.
The FBCL should be standardized: Using IEC-61449 known for automation control
of course version mng for model sources
Tool-specific conversion from model data
The functionaly text presentation of th user's model FBCL Function Block Connection Language
Some more Sources (Target Language) manually written
Target-lang.-specific translation from FBCL
Version managment of functional aspects: What was changed?
The generated Target-Language-Code of the user's model C(++) / Java / etc.
Running Code in Target
Hartmut Schorrig, FBCL 2019-05-20
5

Figure 12: Fbcl/FBCL-TranslationTargetSlide.png

This is an older image from 2019, but it shows the whole truth. The so named FBCL (Function Block connection language) is here shown as textual representation of the graphic, whereby here the usage of Open/LibreOffice for the graphic was not yet present. But the using of IEC61499 was already found as coding standard for the textual graphic representation.

This figure shows also the topics of simulation of the functionality shown in the graphic, also including usage of manual written (core) sources in the target language.

Docu file: Approaches-OFB_VishiaDiagrams