Last page backward

UFBgl – Unified Function Block graphic language
-
Diagrams for UML and Function Blocks drawn with OpenOffice or LibreOffice

Dr. Hartmut Schorrig
www.vishia.org

2024-04-15

Table of Contents

Table of Figures

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 20 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 [1]. Another contribution to UML comes from David Harel [2] 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 [3] or simulation tools. Especially Simulink from Mathworks [4] 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 [5] for automation devices as a basically feature. Also in Simulink events are designated and used for "*triggered subsystems*" as well as for state machines. But events are familiar also in UML for *State Charts*, and should be familiar in Object Orientation.

3 Approaches for the graphic, basic considerations

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

3.1 Question of sizes and grid snapping in diagrams

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) Documentation
  b) 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.

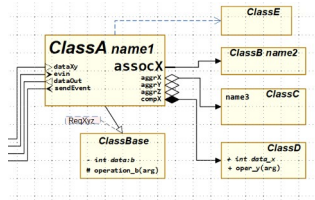


Figure 1: View 40%

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.

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.

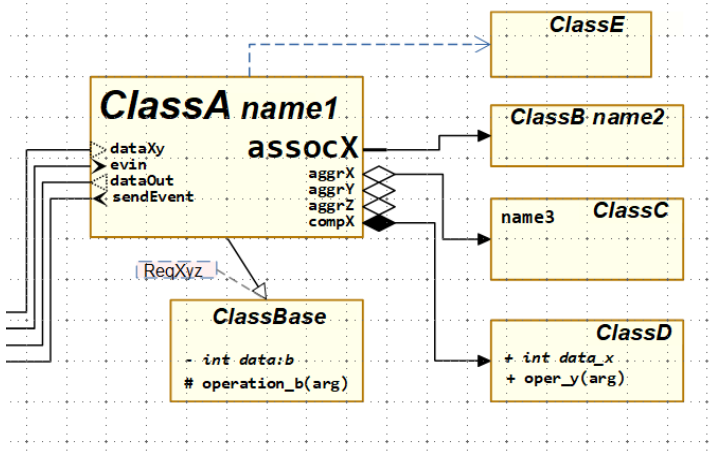


Figure 2: View 100%

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:

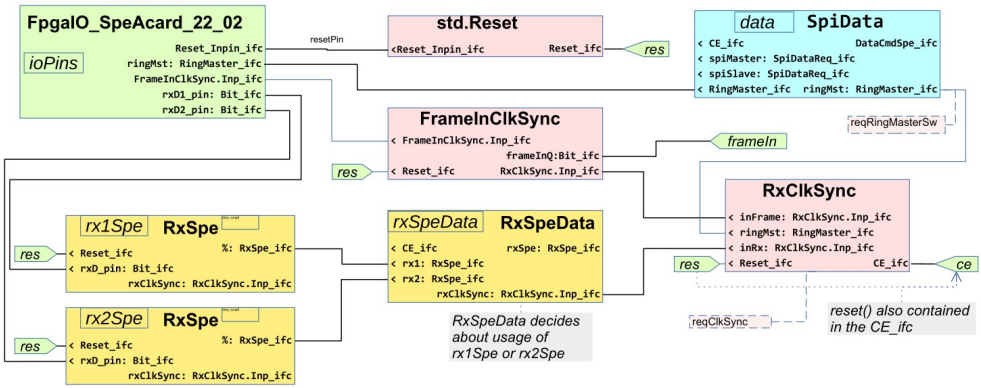


Figure 3: Example for a Module Diagram

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 style sheets for elements

The first used is a rectangle shape which presents a class or Function Block (FBlock). The rectangle should be marked with the style sheet `ofbClass` or also `ofbFBlock`. This style sheet (indirect formatting) associates the semantic to the shape.

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 elements for connections are named ***pin.*** This is similar as in Function Block Diagrams where the data connections are presented also as pins.

In UML a **port** is known. This is also a pin, see Style_ofpAggrRight_TextProp.png Figure1: on next page 15.

The pins are simple small figures with a fixed size, known from UML as the diamond (filled / non filled) for Composition and Aggregation, or they are simple rectangles. 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.

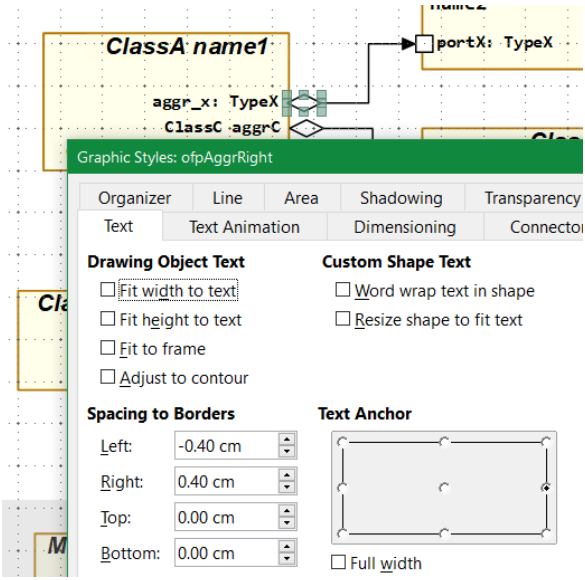


Figure1: Style_ofpAggrRight_TextProp.png

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 Figure 1: View 40% page 11. 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 common explanations.

3.3 Connectors of LibreOffice for References between classes

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, nor 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.

3.4 Connect Points for more complex references

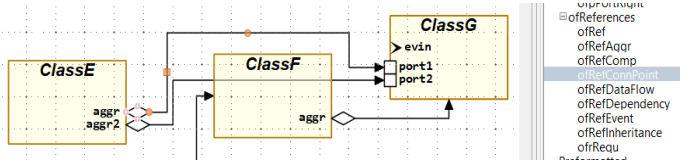


Figure2: ReferenceLineCrossesBlock.png

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. LibreOffice seems to be have the disadvantage that additional inflection points on orthogonal connectors are not possible. Look on the example right side.

The solution is shown also image. 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.

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**.

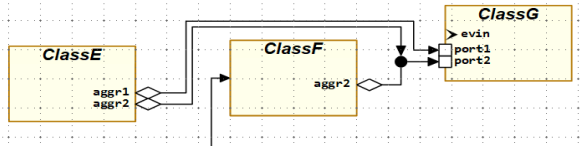


Figure3: OFB/ConnPoint.png

3.5 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.

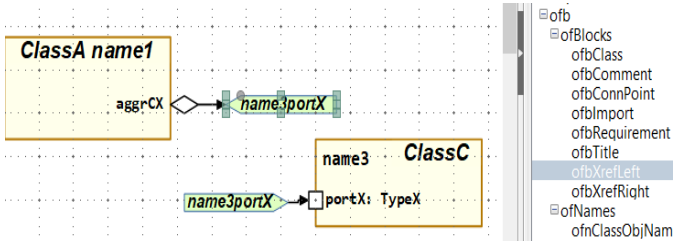


Figure4: UMLdiagramXrefExample.png Cross Reference usage

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.
- 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 Working flow creating your own diagrams

First you should load and open the template file from

https://www.vishia.org/SwEng/oofb.wrk/src/UML_FB_DiagramTemplates/odg/UML_FB_DiagramsTemplate.otg

To create a new empty UML class or Function Block diagram you should save this template file under your specific location/name.odg. You should delete the content, the style sheets are not deleted.

Reopen the template file, you need it to copy figures and elements from.

If you have your own file with content but maybe an older version of style sheets, you may copy the style sheets immediately with zip: The odg or otg file is a zip file format. Add the extension .zip and unzip it (simple us the Total Commander). It contains a `styles.xml`. Replace the `styles.xml` in your own file (with zip extension). Remove the zip extension and reopen it in Libre Office. It should work. Do not forget to make a backup copy. This is a non documented way, but it seems to be stable since many

years. It works also for OpenOffice in different versions.

Look for Grid and Snap

- Open *"Tools - Options"*, select *"Libre Office Draw"* and then first *"General"*. Look for the measurement unit, it should be "cm".
- Then open "Libre Office Draw" and "Grid", look for the proper grid settings (recommended 0.5 cm and 5 Subdivisions because the natural cursor step width is 1 mm. Select "Snap to grid". This is strong recommended, because you have a lot of work for unsnapped blocks and some small inflection points in orthogonal reference lines.

If you have copied from the template, it should be proper.

Create a class or function block:

- Create a simple rectangle in your diagram and assign the style sheet `ofnClass`. The it gets the yellow color with brown border. Alternatively you can copy a class block from the template.
- Create a simple rectangle and write first your class name into it (press F2 to write text in a selected rectangle). The assign the style sheet `ofnClassTypeName` to it. Now move the rectangle into the class box, usual (not necessary, but recommended) to the top right border. You should not place the name exact in the mid, it makes a little bit trouble by selecting the correct glue point for the class rectangle.

  Alternatively you can copy the rectangle from the template.
- Maybe write some data or operations into your class block in the same kind, either by copying from the template, or also by creating simple rectangles and assign the style.

Copy connection elements

- Then you may copy connections (aggregations etc.). For this you should use the template, copy the correct element in your diagram. On paste it lands on exact the same position as in the template, its on the top spread of the page. You can use the cursor keys to shift it to your destination firstly, so long it is selected. Sometimes the landing position is inside any other stuff, this is a little bit confusing. Unfortunately Libre Office does not paste a figure on the cursor position (as other tools do). It would be more proper.
- You can copy more connection points from the same type also from other ones in your diagram of course, it is usual faster.
- On copying and moving the figures the landing position should be any time in the 1 mm-Grid. Sometimes it may be wrong, you see it on small inflection points and obviously misplaced positions. Then you can press F4, correct the position to even mm. If you have activating snapping, all will be proper after such an adjustment (till a next non obviously positioning which may be also caused by accidentally size changes).

Group and ungroup

Meanwhile the detection of content does not need grouping. You can group associated elements (FBlocks with their pins) but you do not need.

The association between elements, FBlocks and pins, is detected by its position. The FBlock frame is responsible, derermines the area. The pins should be inside or at least touch this area with at least on coordinate. In the moment it is a problem if two FBlock frames are too near, should just have a distance of 1 mm.

Small problems with movement

The elements have a height of 2 mm and often only a size of 2 x 2 mm. If you select it, Libre Offce shows drag points to change the size, but because the size is not changeable, also a "non possible" symbol. The space for movement is small in the mid of this points. 2 x 2 mm is the smallest size where movement is possible on a 1920-pixel screen with full size width. This is a little bit stupid.

But you can also move with cursor keys.

Using a higher zoom factor (200 % is recommended) ameliorates this situation. Usual you don't need to see your page margins.

Hint: Bring to Back / bring to Front

The rectangle of a class should have a transparency. Then you see also elements which are arranged below (in the back) in relation to the class rectangle. But to work with, the inner elements should be in front and the class rectancle should be in back. Use the menu entry "*Shape - Arrange*" or the context menu with "*Arrange*" to adjust it.

Using layer

This is not tested yet. Maybe interesting in future.

5 Overview capabilities and concepts of the UFBgl

What do the diagram contents mean?

This chapter should discuss some presentations in the ObjectOriented Function Block diagrams in relation to the UML standards and some quasi Standards used for Function Block. Function block representation and UML should not be a contradiction. It should be thought together for the future.

5.1 All Kind of Elements with there Style Sheets

The next image shows all given template elements. It is the content of the file

https://vishia.org/fbg/deploy/UFBgl_DiagramTemplate.odg

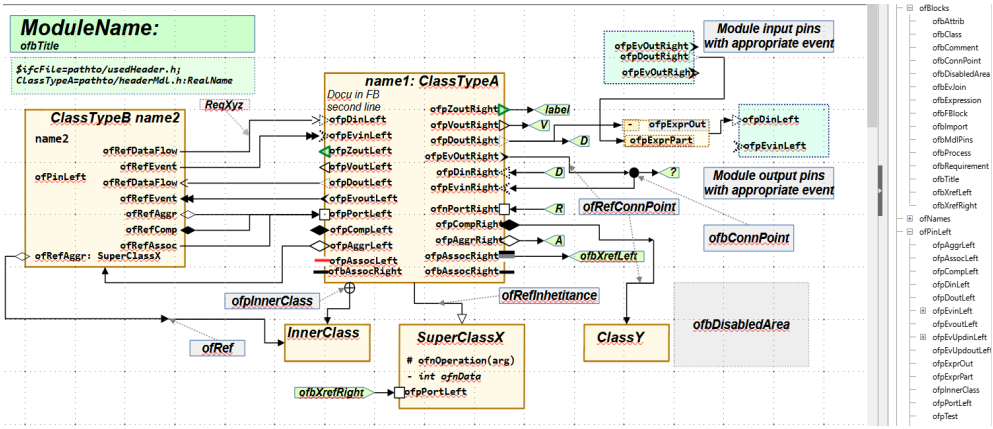


Figure5: odg/UFBgl_DiagramTemplate.png

Right side you see some style sheets. You can use this image (given in the file `UML_FB_DiagramsTemplate.odg`) to pick an element, copy it to clipboard and insert it in your graphic. The style sheets are copied by opening this file and save it with your name. Unfortunately LibreOffice does not allow loading style sheets from another given odg document, only by copying the original one (see also https://ask.libreoffice.org/t/how-can-i-import-styles-from-other-draw-documents/8834).

But you can copy the internal `style.xml` file from the `UML_FB_DiagramsTemplate.odg` zip archive. This is a simple, proven workflow that has not been recommended as often, but it works:

* Copy the original `UML_FB_DiagramsTemplate.odg` file to `UML_FB_DiagramsTemplate.odg.zip`
* Open the zip file by a unzip tool.
* Copy the internal `styles.xml` for your own.
* Make a backup from your own `*.odg` file only to have it for trouble.
* Rename your own `*.odg` file to `*.odg.zip` and open it with a zip tool.
* Replace the internal `styles.xml` with the `styles.xml` from the template.
* Rename your own `*.odg.zip` file back to `*.odg`
* Check if all is proper. It should be.

For dealing with zip content, using the Total Commander is a good decision.

The class in the mid with `name: ClassTypeA` contains all connection elements for the concept described in 3.2 Using figures with style sheets for elements page 14. The identifier of the style sheet is here used also as name, only for documentation.

The class left `ClassType name` contains simple connection elements of the base style `ofPinRight` and `ofPinLeft`, but using connections with the specific type. Their style names are shown here as pin names. This was a first concept, maybe in future not recommended. Here the connection styles determines the kind of the pin.

The figure outfit is proper for view, but not necessary for content. It is also possible to use simple rectangles with the proper style. Then it is not so good recognizable which kind of pin it is. But handling of content (the text) is more proper. It may be recommended to use this simple rectangle forms for the amount of data pins, and use the specific form with the triangle shape for the events to see what's happen. This is in the moment growing experience. The evaluation of the graphic works with both variants, because for evaluation only the associated style is essential, not the form.

The internal data of a class can be shown, as usual in UML, with the style `ofnData`. The designation about private, public, protected should be written with a first character `-` `+` `#` as usual in UML. Writing the type of the data is recommended. The operations can be written with their argument names, if it is more informational. The operation itself, its body, should be define anyway in a programming code and not with a diagram. The association between the shown operation in a diagram and the real operation is only for documentation, should not be formalistic.

For the documentation blocks the style `ofbComment` should be assigned. A requirement is presented also usual in UML with a short identifier. It is written in a `ofbRequirement` rectangle block. The connector between `ofbComment` and `ofbRequirement` has the style `ofRefDocu`. If you copy this connectors from the template, you get also the style reference.

This diagram contains also data and event flow. This is described in chapter Error: Reference source not found.

5.2 FBlock or class, name and type

The name of a FBlock and the type can be written in the text of the rectangle shape for `ofbFBlock` which is used for the FBlock, and also for a class in UML thinking. The original style of `ofbFBlock` expects the text in the right top corner, see Figure6: But sometimes this works not properly, then either "*Format – Clear direct Formatting*" on the shape helps, or Menu "*Format – Text Attributes*" and adjust it. You can use also the direct formatting to put the name and the type in the mid, to another corner, or at a desired position. But right top is often a good decision because the FBlocks have often more inputs (left side) then outputs.

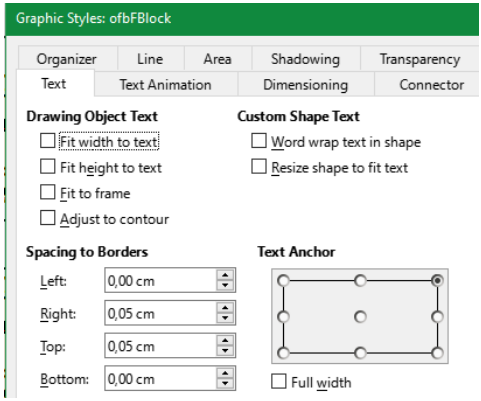


Figure6: odg/ofbFBlock-TextStyle.png

- By the way, inputs do not need positioned left, can be also right or rotated on top or bottom, same as outputs. The drawing style have more possibilities than some commercial tools, you can use it for your own.

The other possibility for name: type is a text field marked with the style `ofnClassTypeName`. This text field can be positioned anywhere inside or touching your FBlock shape.

If you want to describe only the class (type), then you need to write `:typeIdent` with the colon. This is not UML-conform, but unique.

If you omit the type name, but the classification of the named instance is done in another FBlock with the same name, it is admissible. It may simplify the diagrams. If the type is never associated, an error message is given on translation.

5.3 Show same FBlocks multiple times in different perspectives

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 able to found 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

This approach uses the principle, showing also a FBlock in several perspectives, in opposite to traditional FBlock diagrams, but similar as UML. It means, on FBlock as one instance can be shown more as one time in the same diagram or in several diagrams related to the same module. The FBlock is dedicated by its name. Drawing a second FBlock with the same name is the same instance.

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 Figure7:in the next chapter. 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 instance, and also to the same class.

5.4 Function Block and class diagram thinking in one diagram and the ObjectOrientation and also Functional aspect

One of the basic ideas of this 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.

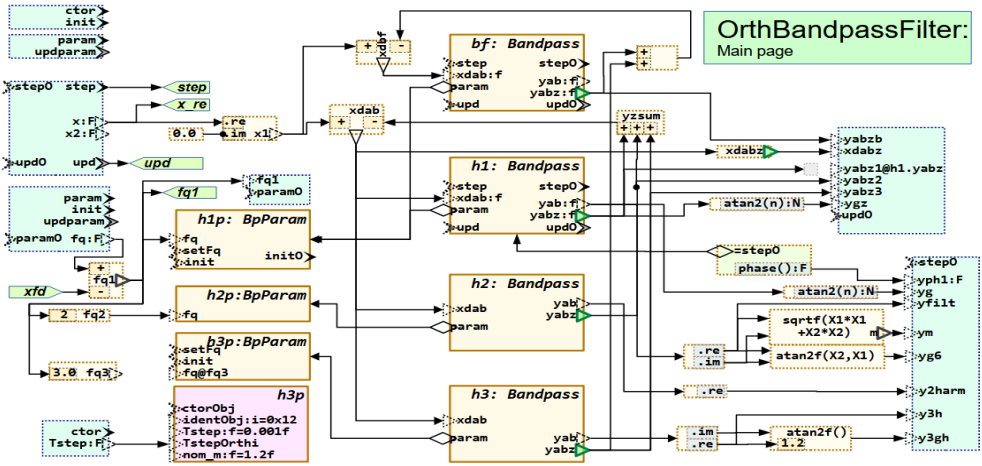


Figure7: odg/OrthBandpassFilter.odg.png

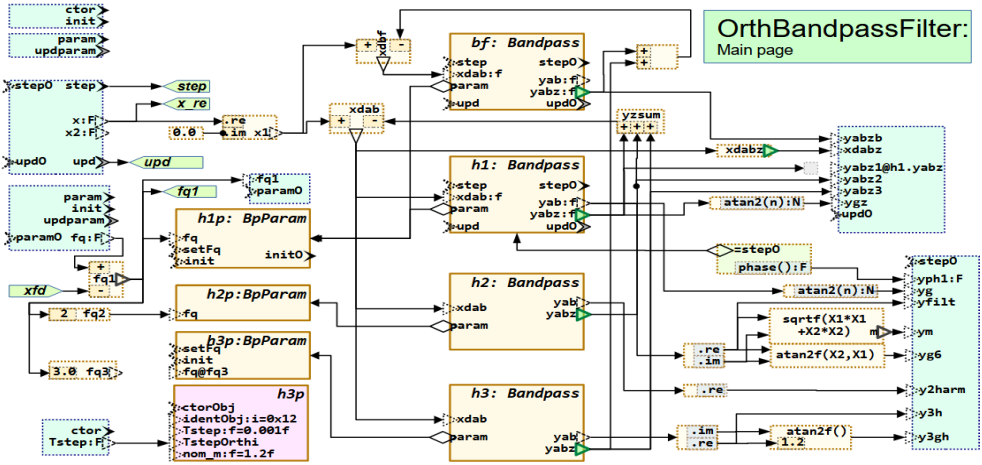


Figure8: odg/OrthBandpassFilter.odg.png

Figure8:odg/OrthBandpassFilter.odg.png 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.

5.5 More as one page for the FBlock or class diagram

The chapter above 5.4 Function Block and class diagram thinking in one diagram and the ObjectOrientation and also Functional aspect 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.

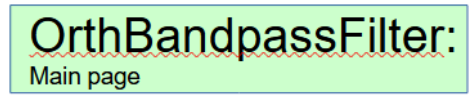


Figure9: ofg/ofbTitle-1.png

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.

5.6 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.

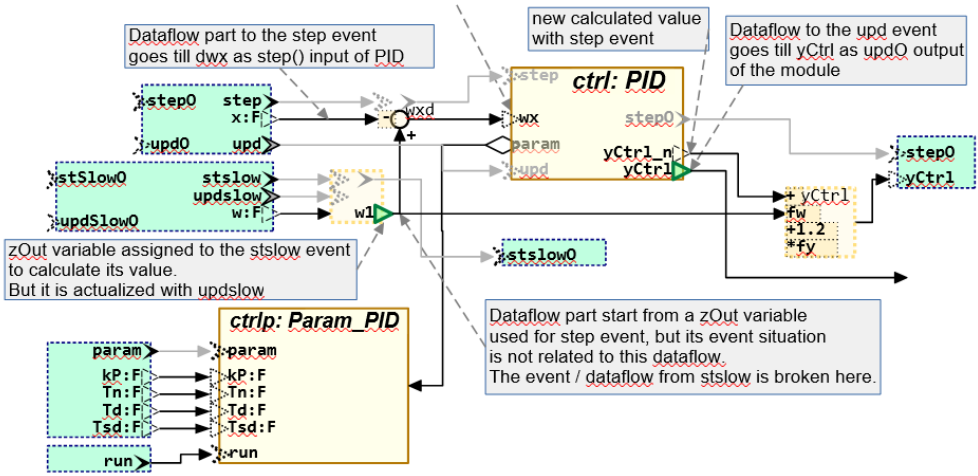


Figure10: OFB/DataFlowPID4.png

The Figure10:OFB/DataFlowPID4.png 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 *)
```

5.7 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.

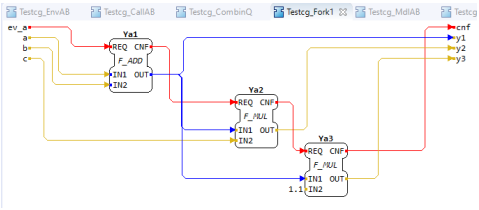


Figure11: 4diac\Testcg_Fork1.png

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.

5.8 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 Figure7:odg/OrthBandpassFilter.odg.png.
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++.

5.9 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:

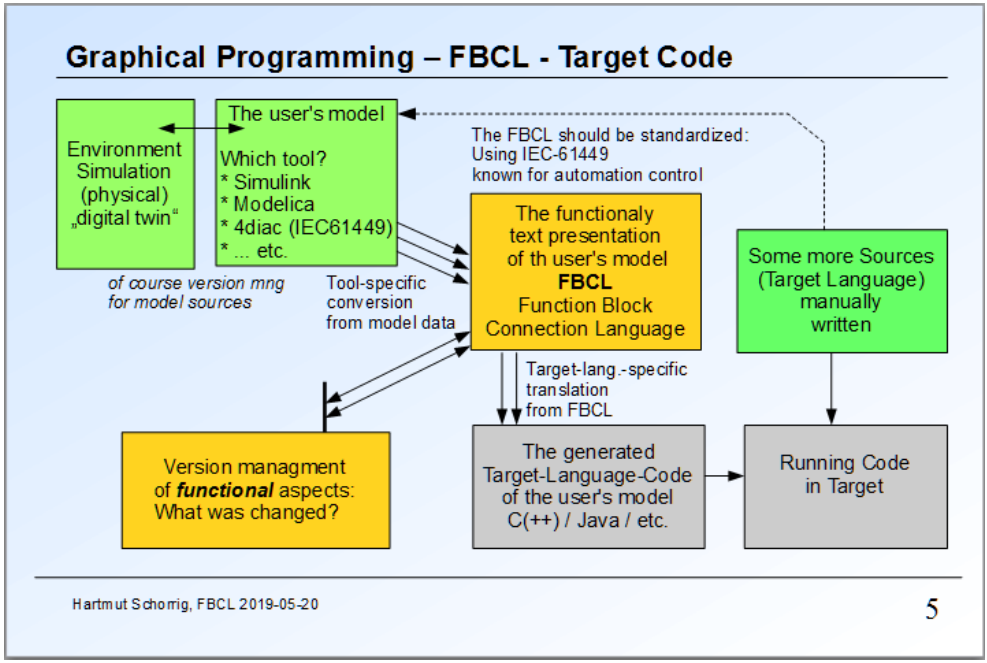


Figure12: Fbcl/FBCL-TranslationTargetSlide.png

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.

6. Details

Table of Contents

6.1 Data types

Table of Contents

In the Figure10:OFB/DataFlowPID4.png the input `x:F` is designated as float input with the letter `F`. This is very space-saving but still obvious. Other tools sometimes have only a *“Pin dialog”* where the type can be selected and can optional show the type in the graphic, but then all types destroying the overview. The idea only using one character should be seen as proper, the number of types used are not too much. This is for the standard usual numeric types. The type of aggregations are determined by the destination class. A type name can be given additionally if necessary.

The problem on numeric and basic types is: There are a lot of designations in different programming languages and usages, but they are similar. A second approach is: Also regard non full deterministic types.

```
ANY                                 A
+-ANY_DERIVED                       L
+-ANY_ELEMENTARY                    E
  +-ANY_MAGNITUDE                   M
  | +-ANY_NUM                       N
  | | +-ANY_REAL                    G
  | | |   LREAL                     F
  | | |   REAL                      D
  | | +-ANY_INT                     K
  | |     LINT, DINT, INT, SINT     J I S B
  | |     ULINT, UDINT, UINT, USINT Q U W V
  | +-TIME                          T
  +-ANY_BIT                         b
  | +-LWORD, DWORD, WORD, BYTE      q u w v
  | +-BOOL                          Z
  | CHAR                            C
  +-ANY_STRING
  |   STRING                        c
  |   WSTRING     (not specified)
  +-ANY_DATE                        H
      DATE_AND_TIME                 t
      DATE, TIME_OF_DAY             a h
```

6.1.1 One letter for the base type:

IEC61499 and also the automation system programming language IEC61131 knows the following definition of types, See *IEC 61131-3 Second edition 2003-01, Reference number IEC 61131-3:2003(E)*, page 32. The type `CHAR C` was later defined in IEC61131.

Complex types, not defined in IEC61499

```
  ANY_MAGNITUDE                     M
  +-ANY_CNUM                        n
    +-ANY_CREAL                     g
    |   CLREAL                      f
    |   CREAL                       d
    +-ANY_CINT                      k
        CLINT, CDINT, CINT          j i s
```

The shown character for this types (green) are used for UFBgl, based on this basic types:

- **D F J I S B** that are the standard numeric types which are also known with this same char in Java as return value of java.lang.Class.getName() for the primitive types **double**, **float**, **long** (64 bit), **int** (32 bit), **short** (16 bit) and **byte** (8 bit). They have its adequate in C/++ with **int64_t**, **int32_t**, **int16_t** and **int8_t** for the integers. In IEC61499 they are named **LREAL**, **REAL**, **LINT**, **DINT**, **INT**, **SINT**.

- **Q U W V** are the unsigned typs in C++ **uint64_t**, **uint32_t**, **uint16_t** and **uint8_t**. In IEC61499 they are named **ULINT**, **UDINT**, **UINT**, **USINT**. In Java there is not a counterpart, the larger signed types should be used. The used characters should have their mnemonic in *"Quad word"*, *"Unsigned"* instead **I**=**int32**, *"Word"* usual in some systems for 16 bit and **V**, it is near **W**.

- **q u w v** are the counterparts of unsigned, designated as *"Bit types"* as also in IEC61499 as **LWORD**, **DWORD**, **WORD**, **BYTE**. Distinguish between *"unsigned"* and *"bit value"* is not familiar in C/++ language, both is **uint…,** but it may be proper to distinguish it on user level of an application. In IEC61499 and IEC61131 (sometimes designated as *"safe language"*) it is distinguished. The difference for the UFBgl usage is: The bit types are not compatible with the common numeric type **N**.

- **Z** is for boolean, the same as in Java **Class.getName()**. What is a boolean, it should be clarified. How is a boolean presented in machine level: This is not a problem of the graphic, depends on implementing stuff. A boolean may be also possible to represent only by one bit in a bitfield. In IEC61499 it is named **BOOL**.

- **d f j i s** That are the complex types as counterpart to the real types. Complex types are fundamentally for numeric solutions, but they are not standardized in any language. General this types are structured types. For IEC61499 code generation they are named **CLREAL**, **CREAL**, **CLINT**, **CDINT**, **CINT**.

- **C c** is for one character and a String. Unfortunately the letter s or S is already used for *"short"* and T or t for *"Time"*. Whether a character has 8 or 16 bit (ASCII, UTF8, UTF16) is clarified on implementing level.

- **T** is for a current time (relative) due to the usage in IEC61499 and IEC61131 as **TIME**. How many milli or nanoseconds is represented by one step, it should be clarified by the implementation. It should be the same for all time values for the whole application.

- **t** is an absolute time stamp adequate to **DATE_AND_TIME** in IEC61499 / 61131. The format of the absolute time stamp should be clarified for the implementation. Often it is the seconds after *Jan 1th, 1970* (as in UNIX), or better seconds and nanoseconds after a dedicated base year. It is important that it is a continues value of seconds.

- **a h** is a value of the date only, the d**a**y, and the time of day or the question which **h**our. As mnemonic. It is also implementing specific how is it presented in machine code. It is supported also as continues value. For the human interface it is always processable as human readable format, which can also regard time zones etc or country specific presentations. This stuff should not be mixed in a core application.

6.1.2 Unspecified types:

Some FBtype uses unspecified types, because they are available for more or all numeric types, or the type is checked and used really on runtime. In C/++ this is often designated as **void*** also as pointer to basic numeric types. In Java there is the **Object** class as common representation of all types. But the main approach is: The type should be specified by forward or backward declaration in the graphic model by data connections.

- `N` presents any numeric type. This is formally also an unsigned type, whereby using unsigned for numerics is sometimes a prone of error. It is compatible to `D F J I S B Q U W V`
- `n` presents a complex numeric type, compatible to `d f j i s`
- `M` is any numeric presentation, not complex one and not bit values. It is `N T`
- `E` is a non referenced type.
- `L` is a referenced type. In IEC61499 and 61131 it is named `ANY_DERIVED` and distinguished from the `ANY_ELEMENTARY` . It does mean a structured type or also an enumeration defined there with `TYPE … END TYPE`. All of them can be present by an aggregation to a FBlock which contains the appropriate values. The `L` follows the `Class.getName()` in Java for the `Object` type. It is especially any reference type to a class type (a pointer) similar as the `void*` in C/++.
- `A` is a really unspecified type. This is also if the type specifier is not given.

6.1.3 Array data type specification

Arrays with one dimension and a determines length are defined by a simple number after the one-char-type, such as `F3` for a `float[3]` array. This is a concise simple style which needs less space in the graphic.

Using simple one dimensional arrays is often necessary in FBlock graphics, because several values are calculated with the same procedures. It depends from the implementation whether a FBtype can really process a vector, or whether more as one FBlock is instantiated and called for the vectorized calculation. The graphic should not deal with this implementation detail. For example a FBtype to calculate the complex representation from a 3-phase voltage in a grid has of course an input `F:3` for the three phase values, and hence an output `f` as complex, and also an output `F` for the so named zero sequence value which is often `0.0`.

For expressions there is a simple way to build vectorized values and access to elements:

TODO

6.1.4 Container type specification

A container is known in higher programming languages, for example in Java as `java.util.List` or as sorted container as `java.util.Map`. Also an array with a non limited size is a container.

In UML the `*` is familiar to designate an aggregation with more possible destinations. This is also a quest of container: The aggregation (or also association and composition) has a multiplicity. Whereby the possibility to select exactly between 1.. or 0.. or 0..2 members or such is not supported in this granularity. It is possible also to have an array of a dedicated size also for aggregations. But whether this elements are set or they are nil, this should be checked by the implementation.

- Write a `*` after the type specifier or also on place of the type specifier (`name:*`) it is designated: Any container. The implementing level decides about the implementation of a container. A container refers or contains any number of elements, sorted in order of input. Such a linear container can also implemented by an array in a free size.
- `**` after the type designates a sorted container. The sorting key is implementation specific or specific from the creating and using FBlocks. Often the name of an element is the sorting key (it's a `String`).
- `[99]` after the type designates an array with variable size but possible with a given maximal size. `[]` is a free variable size.
- `[1..4]` after the type designates an array with this possible range of size. It is similar the number of associations in UML

What about more dimensional arrays … should be clarfied in future. Writing style dimensions separated by comma such as `[9,3]` or `F2,3` for an array of 2 element which each 3 elements. All rows and columns have an equal length. It should also be possible to use `[][]`, then the rows and columns or more dimensions can have each any different length, such as arrays in Java language.

6.1.5 Structured type on data flow

A structured type for data inputs and outputs is an instance of a FBtype. This instance comes from the data output provided to the data input. The difference to an aggregation is: The aggregation is a stable connection from one instance to another one, the using FBlock can access the currently data from the aggregated FBlock. For that also problems of data consistence (mutual exclusion on access changed data) should be considerate as known in Object Orientation and UML.

The data flow with instances of FBtype presume constant instances, which are not changed after delivering on the data input. This approach comes from the IEC61499. It is often also used in ordinary programming, but not so obviously. The common solution is: The data are binding to the event instance. Or, the event instance contains the data.

Often, for such approaches, dynamic allocated memory is used. This is the simplest form. But for frequently used dynamic memory the problem of defragmentation exists. In Java Runtime Systems this problem is solved by using the Garbage Collector. Another possible solution is: Using only memory blocks with equal sizes.

The other often simple solution is: Using a pool of event data. The event flow is usual deterministic in amount. It doesn't make sense to shoot around with events. An event should be created (using a member of the pool) only if it can also be processed, and if the pool is empty, there are obviously too much events in queues, not processed, and more events are only disturbing. Hence, the pool of event data is often a possible and proper solution for implementation.

Designation of the data type:

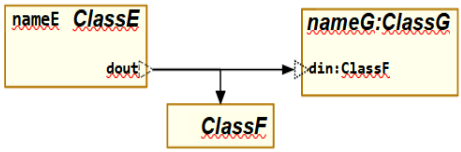


Figure 4: OFB/DflowStructData1.png

The Figure 4 shows two possibilities to dedicate the type of the data flow:

- If you have a connection from a dout or din pin to a class frame of style `ofbClass` or to a FBlock frame, style `ofbFBlock` without instance name, then this defines the type of the data pin.
- The second possibility is, use the type name after colon.

You can define the data pin type also in an extra diagram:

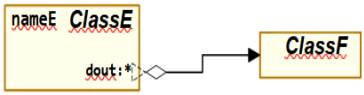


Figure 5: OFB/DflowStructData1.png

Here the connection is used as Style `ofRefAggr` which shows the non filled diamond as in UML. Additional for the type an `*` is written. This means, as also for other types, The type is a container. Also an array size can be used there, or the `**` for a sorted container or `[]` for an array of not variable size. This is also possible of course for a immediatelly type specification as in Figure 4 on `ClassG`.

6.1.6 Data type forward and backward propagation

The input variables of the PID controller do not need this type declaration here, because the type is forwarded. But it is shown nevertheless, gets more clarity for usage. The type of the output variable `y:F` do also not need to be shown if or because the module is well defined in its interface for explicitly types or for type forwarding.

More step times or calculation events: In this example automatically an event chain is generated from `stslow` (means a slower step time) to the expression block with the `w1` variable, and forward to the event output `stslowO` (not shown here). Because `w1` of of style `ofpZout…` it needs updated with the correspond `updslow` event on the module's input block. If the value of the `ofpZout` variable is connected to outputs of the module with also the `updslow` event, the appropriate data flow will be assigned to this event chain till `updslowO`.

Data consistence: If the value of the `ofpZout` variable is used in another event chain, as shown here for built `dwx`, the stored value of the last calculation (after update) is used. In this case the value comes from another step time or calculation event, just the `stslow`, and hence consistently data all from this update event can be used. The consistence of the data should be guaranteed by a proper implementation. For example a slower step time can prepare values in with higher calculation effort, but the update of this values is done in a high priority interrupt which cannot be interrupted by another. The update needs only copying of values, or as better solution switch only a pointer to a double buffer system, if the update event is registered for the interrupt. Then the values are always consistent.

old:

You can show data and event pins on classes, but the connections are only sensible between the instances. This is familiar for FBlock diagrams. The t**ype of data pins** can be given immediately on the pin (after colon), but can be also forward propagated by a data flow. Simple arithmetic operations do not change the type of source pins and forward the type to the destination pins. Specific operations (for example access to the real and imaginary part of a complex value or to an array element) does not change the numeric type but influences the real/complex or array property of the type. Specific FBlocks can forward the type of inputs to the type of outputs. A backward propagation (as in Simulink) is not designed, because sometimes a mix of forward and backward propagation is more confused by the user. An important property of FBlock diagrams is, that the numeric type of pins in library FBlocks are not determined, instead a type dedication as `ANY_NUMBER` (in IEC61499) or such can be used. In Simulink it is determined as "*inherit*" type. It means that the types in the usage of the FBlocks depends from its using environment. For code generation either any template should be used (C++) or the FBlock should be existing as variant with all necessary types, or the FBlock implementation is a macro (C language) where the compiler associates the type.

6.2 One Module, Inputs and Outputs, file and page layout

Table of Contents

6.2.1 Module in file organized in pages

On odg file can or should contain one module, but can contain also more as one module. It should be possible to distribute one module to more as one odg file (do in future). But then all these files must be processed with one translation step.

Any page must have a shape with style `ofbTitel`:

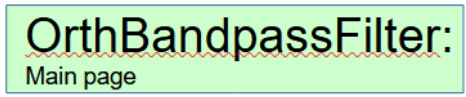


Figure13: odg/ofbTitle-1.png

The first word separated with colon is the name of the module, should be an identifier. The following text is only comment in the graphic. It is not used for code generation or other content evaluation.

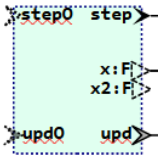
If you write a sharp as first character `#Modulename:...`, then this page is commented out, not used for evaluation.

You can have more as one page in one file with the same `Modulename`. Or just more as one file. The pages are count in order of the files and in the file.

6.2.2 Module pins

Module pins should be contained in a shape or graphical block (***GBlock***) with the style `ofbMdlPins`

Figure14: odg/ofbMdlPins-1.png



This ***GBlock*** should contain data input and output pins, whereby for practical reason the output pins (usual right on side, left in the block frame) are separated from the input pins (usual left on side, right in the ***GBlock*** as shown here).

But also associated events should be given. The events are important for association to the data.

The module's data input pins are of style `ofpDout…`, usual `ofpDoutRight`. Why dout: because they are data outputs to the inner connection of the module, they are data inputs seen from outside, from usage of the module. Figure14: shows module's data inputs. Adequate, the module's data outputs are of style `ofpDin…`, usual `ofpDinLeft`.

The module's event input are `ofpEvout…` and `ofpUpdEvout…`. Both are shown in Error: Reference source not found right side.

With the association of data to events the data are associated to this event, or in other words, it builds the arguments to the event

operation in the order given from top to down. Whereby, data to update events does not exists, the data are associated to the prepare event (**ofpEvout…)**

The given update event is associated for the update operation proper to the prepare operation.

It means for this Figure14:, the module has one operation

```
step_OrthBandpassfilter(…, float x, flost x2);
```

and one operation

```
upd_OrthBandpassfilter(…);
```

without data arguments. For prepare and update see chapter 7.4. Prepare and update actions page 74. The association of the prepare event (here **step**) with the update event (here **upd**) in the module's pin block is essential for build the event flow due to the data flow. The event flow is first build for the prepare event, but all reached FBlocks are associated then also to the given update event, if they have an update operation.

The presentation of the module's event out pins for prepare and update, style **ofpEvin…** or **ofbEvUpdin…** (optional) means, that the module's input event are not ending in a state machine, which has specific output events, instead this are operations with immediately output data and a created output event if they are calculated. From outside, without knowledge of the inner module functionality, this module can be seen as a black box Standard FBlock with a simple regular state machine. It means, each event reacts with an output event, and does not really change the state, or it has defacto no states.

A module with FBlocks with state machines are not realized in the version of UFBgl. But then the module's output events are not given.

To complete this description, have a look to page one of this module as a whole. The same image is used more times in this documentation, because it shows some important concepts on one example:

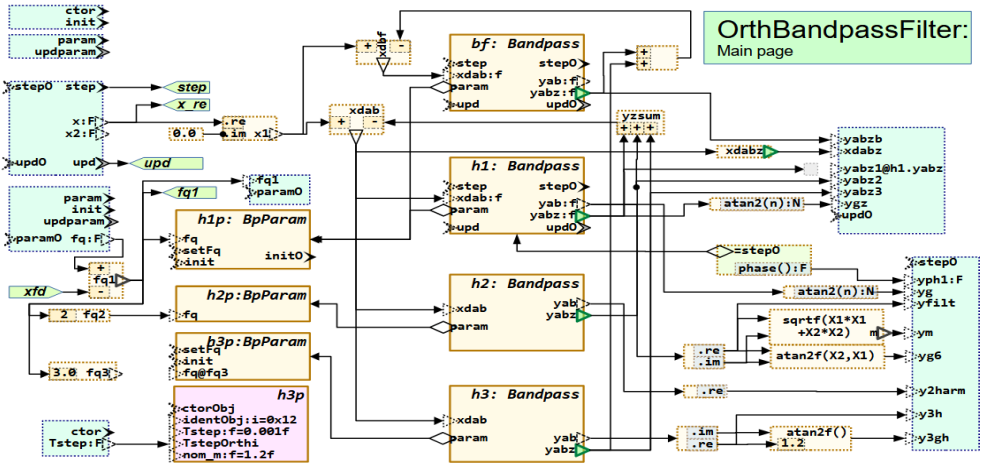


Figure15: odg/OrthBandpassFilter.odg.png

In Figure15: left top are some input events and data, and proper output pins for the prepare event are right side for **stepO**, and also right side above with **updO**.

The code generation can offer an operation prototype with these output values which should be implemented outside, but it is called inside the module if the module's

output event is activated. That is the pure event driven implementation. This output event operation can be implemented either to send the events via communication in an event queue, or via inter-process-communication to any other device, or it can be implemented to organize the call of an operation of another module.

The other more simple more manual programming approach is, only offer the calculated values in data struct.

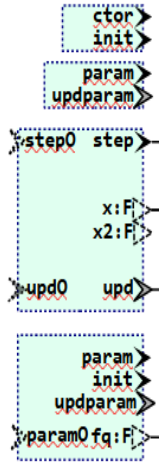
6.2.3 Order of pins

The order of the pins is important both for the generate fbd file (IEC61499 presentation) as also as argument order in the operations, and as order in the generated code. If you think on reproducible build, then it is important that a repeated generation of code should create the same source code if the determining conditions are not changed. For example if a graphic position of a FBlock was moved to a slightly other position, or one connection is new routed in graphic, but is unchanged in functionality, then the generated code should be unchanged. But any where the order of the pins should be determined. It may be sensible to sort the pins by its name (alphabetically), but it is better to sort the pins by its graphic position of first usage. If the pins are used furthermore, in other pages or in the same page twice, it is not essential. The first detection in graphic determines.

To have an overview this part of Figure15: is repeated here: in a part as Figure16: For the approach of using the graphic position, the graphic here contains left top first the both events for `ctor` and `init`. It means the first event (left, top) is `ctor`. Then `init` comes. This is the order of the event in the IEC61499 fbd file and also in generated code. First the `ctor_…()` operation comes in the implementation source, then the `init_…()`. But the data for `ctor` and `init` are not designated here, it is in another `ofbMdlPins` block.

The order is first the order of the `ofbMdlPins` ***GBlocks***, and then the order of the pins inside each ***GBlock***.

Figure16: odg/ofbMdlPins-2.png



For the GBlock order, internally a number is build consist of the page number on a high position (bit 22), the x position from bit 11 and the y position. The positions have a resolution of 1 mm, hence 2047 mm or 2 m * 2 m area can be used for the graphic, and ~ 1000 pages. But the x position is filtered to columns: When two GBlocks are almost under each other, but not exact, they should be related together in one column. For that a distance of +-9 mm is accepted as the same x column. Whereby not the first found shape determines the common x position, but the mid value of all. Look on Figure16:. You see that the GBlocks are on the same x position rights side but not left side. But all are accepted to be in one column. It means the order is as you see.

A GBlock more right comes in order after the last GBlock on bottom more left. But the distance of +- 9 mm of the column width should be proper to a normal size of a GBlock (10..20 mm width) and a proper column association.

The pin order in a ***GBlock*** is first left from top to bottom with x1 left of or exact on the border of the GBlock area, then on top (y1 less or equal the GBlock area), then right side with x2 right or equal to the GBlock border, and then bottom side from left to right. At last also Pins which are only inside the GBlock are regarded. in order of first left to right, then (the fine order) top to down, in 1 mm rounded positions.

For this example it is very easy. First comes `ctor` and `init` from the first GBlock, then `param` and `updparam` from the second GBlock, then `step0`, `upd0`, `step`, `x`, `x2` and `upd` in this order from the third GBlock, and last `param0` and the rest from the forth GBlock.

The same is done also for ***FBlocks***, which can have more as one GBlock for one FBlock. Also here the order of the same FBlock instance (same name) is used as first order, from page, x-column +- 9 mm and then y-position. Then the pin order inside each of this FBlock is build with the same rule.

Also the same is valid for ***FBexpr***, the expression GBlocks. Whereas FBexpr are always present by only on GBlock. The order of arguments of the expression is left side from top to bottom etc.

6.2 4 The module's output

It may be possible to adapt the code generation that instead access to output variable any time an operation call for a *"getter"* is generated in the code, and hence executed with the core sources. This is if for example in C++ all instance variables are encapsulated as private. But often especially for generated code which follows stronger rules as manual written one, the immediate access to the variable in a data struct is desired. Then the special solution to call a function, not only a getter, really to execute a functionality may be desired. Such a function may have also input arguments and may have output values called by reference if more as one output is necessary. One output value is usual returned by value.

6.3 Possibilities of FBlocks

This chapter should show all possibilities for Function block shapes (FBlocks).

Table of Contents

6.3.1 Difference between class, type and instance

In ordinary Function Block Diagrams usual any FBlock is an instance. The term *“class”* is not usual. If a FBlock is derived from a FBlock in a library, the FBlock in the library can be seen as *“type”*.or just *“class”*. The library FBlock contains the inner functionality, the own diagram *“uses”* it and builds an instance with own inner data..

In UML (Unified Modeling Language) the term *“class”* as synonym for a *type* is usual, and instances (incarnation of the class type), sometimes denoted also as *“object”* are more rarely used in diagrams.

The UFBgl (*Unified FBlock graphic language*) uses any FBlock as presentation of the type (*class*). If the FBlock have an instance name, it is also an Object or ***FBlock***. The type is presented by all FBlocks with the same type name, also if they are several instances. But also the same FBlock (same instance, same instance name) can be presented more as one time with several graphic shapes (GBlocks). It means a class or a FBlock can be shown in different contexts, see also 5.3 Show same FBlocks multiple times in different perspectives page 22

The Figure17:shows an example which contains 3 FBlocks which define the type or class `Bandpass`. Two of them are only for type definition, here the association of data inputs and outputs to events are defined, and also the aggregation `param` associated to the `init` event. The `h3:Bandpass` is an instance definition which contains constant values for two inputs and connections for two other ones. Similar, this is a type definition because here the inputs for `kA`, `kB` etc. are defined as associated to the `ctorObj` event. It is for construction. The type `WaveMng` is defined with also 3 FBlocks, but all with the instance `wf1mng`. One of these FBlocks has no type definition, but the type assignment to the instance is given on two FBlocks with `wf1mng:WaveMng`, one association would also be unique, both associations should be congruently. The more as one FBlocks are necessary because the event and data association should be clarified each on one graphic FBlock instance.

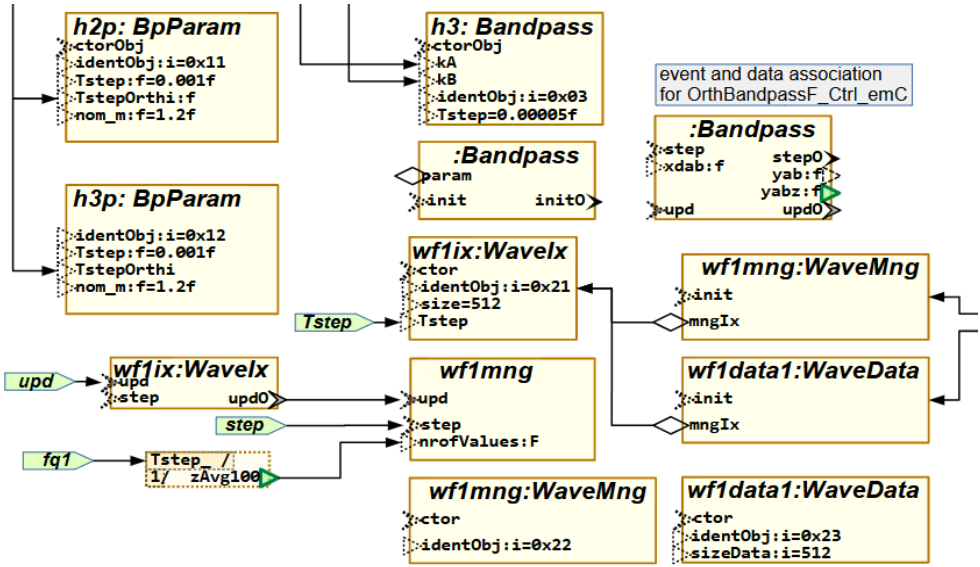


Figure17: odg(ExmplFBlocksTypes.png

6.3.2 FBlocks for each one function, data – event association

In this chapter and also following the following terms are used:

- *Association* between data and events. Also in IEC61499 the term *association* is used in the same manner. The meaning of *association* in UML kind is not related to this.

- *Aggregation* is here the term of UML, used for aggregations shown in the graphic. In implementation these are usual references (containing addresses of the aggregated data with determined type or just pointer).

- *corresponding* events for input and output and for prepare and update (see also 7.4. Prepare and update actions

- The terms *"operation" "method"* and *"function"* means all the same. *Method* is the first used term for Object Orientation. *"0peration"* of a class means the same, the implementation in C language is named *"function"* (may / should have a reference to the data for Object Orientation) and *"function"* is also a common understanding what is done (execution of any functionality).

In ordinary Function Block Diagrams one graphic FBlock presents one instance of a FBlock, and each FBlock has often only one function internally, maybe completed with corresponding construction and init functions. No more. But usual programming in C language (object oriented), more as one function or ***operation*** can be used with one data `struct`, and in object oriented languages (C++, and more) any class has of course more as one *"method"*, *operation* or just *function*.

The non-consideration of the object-oriented concept with several operations per class may be one of the reason of the divergence between graphical programming (often used, non object oriented, specific user-bubble, specific tools with code generation) and the frequently object orientated text coding (other bubble of engineers).

One of the goal of UFBgl is: bringing it together.

But first, discuss about the event thinking:

The idea of event driven thinking of the here used IEC61499 textual presentation of the graphic is not in contradiction to the object oriented thinking with operations, as explained following.

If you look in Figure17:left side, you see some FBlocks with the `ctorObj` or the `ctor` event. That calls the `ctor…` operation for this instances with the given constant or wired input data. It is adequate with the other events. `step` with `xdab` in `:Bandpass` defines the `xdab` data input associated to the `step` event, or just as input argument for the step_… operation. The other `stepO`, `upd` and `updO` events are also corresponding to `step`, as its output (which operation follows) and as corresponding update event.

It means, the events describes the order of execution of some FBlocks, and it describes also one operation of the FBlock.

If you are thinking to the Sequence Diagrams in UML, the origin idea of this sequence diagrams may be really the event communication. But as concession to code generation, which does not regard event thinking, it was broken down to *“operation sequences”*.

The following rule is used:

- If a graphic FBlock has exact one prepare event input (style `ofpEvin…`), then it defines all data input associated to this prepare event.
- The only one update event input (style `ofpEvUpdin…`) is then the correspond update event input.
- The only one `ofpEvout...` is the corresponding output event to the `ofpEvin`.
- All data outputs are associated to the `ofpEvout`.
- The only one `ofpEvUpdout…` corresponding to the only one `ofpEvUpdin`.
- If more as one `ofpEvin…` is given in the graphic FBlock, or more as one `ofpEvout…` or neither an `ofpEvin…` nor an `ofpEvout...`, then this graphic FBlock does not define associations between data and events. The FBlock can be used instead as overview over more as one events, over all or parts of non formal event- associated data but showing commonly relationships of data etc.
- If more as one update events are given, it is shown as error, only the first update event is used (`ofpEvUpdin…` or `ofpEvUpdout...`).
- The data associated to the events and the corresponding events may not be complete. data -event-associations and corresponding events can be dispersed over more as one graphic FBlock. It means the conclusion *“that’s all”* cannot be done. But it should be recommended to show things as complete.

It means, ***a graphic FBlock instance represents*** (a part of) o***ne function, operation or method*** of the assigned instance with its type. In this manner the term *“Function block”* for one function (*operation*, *method*) of a type is proper. The association to one type is given with the type designation, and the assignment to the same instance data are designated by the instance name.

Thinking in these FBlock approaches is related to Object Oriented thinking.

6.3.3 Aggregations are corresponding to ctor or init events

If aggregations are merged in a graphic FBlock instance between data and events, the aggregations are ignored for correspond event-data assignments. But if the `ofpEvin…` event **starts with** `ctor` or with `init`, t**hen the aggregations are associated to this event**. It means aggregations can be connected only in such operations which

names starts with `ctor` or `init`. That are usual used for the constructors and the `init` operation. See also chapter 6.4.2 Life cycle of programs in embedded control: ctor, init, step and update.

It means, the opportunity is given to show aggregation ordinary in diagrams for understanding of relations between FBlocks (instances or classes) between important data connections with there event – data associations (in IEC61499 terms). The data connections regarding its events are used for code generation as arguments of the operation, the aggregations are also regarded as connection between instances, but not related to the shown events.

If the aggregations are never shown together with an `ctor-` or `init-` event, then they are automatically associated to an event with name `init`, or just to the `init_Type(…)` operation. This simplifies drawing diagrams.

This rule is effective for code generation. The generation scripts can be indeed adapted to call any specialized operation, for example to use the identifier part after `init…` as name for the function, but it may be more simple to adapt the called code for example by a macro or inline operation named `init_…(…)` which calls then the original one.

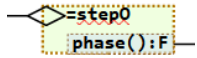
6.3.4 FBlocks for operation access in line in an expression - FBoper

This is a contribution to the Object Orientation. In ordinary FBlock diagrams one FBlock instance presents an instance (of a class) but only with one operation, or some only specific operations. For example, in Simulink S.Functions, *sample time* associations to pins are mapped to several operations). But the object-oriented world has more than one specific operation in addition to simple getter accesses as operations in one instance (class).

This approach, more as one operation for one FBlock, is settled by different events given in more as one FBlock presentation, as described in 6.3.2 FBlocks for each one function, data – event association. The specific event maps to the operation, the associated data are the arguments of this operation. But an operation with return value, usable in line in an expression is not settled with that. Also outputs of an operation *“called by reference”* to given variables are not settled.

For that a specific expression presentation is used, the FBoper (Function Block operation):

Figure18: odg/ FBoperGetter.png



The right figure shows a simpe getter possible as part of an expression. The aggregation refers the proper FBlock, see also Figure15:. The `=stepO` means, that the operation (getter) can be called only after the `stepO` output event of the referenced FBlock. It means the data to get are prepared after finishing the correspond step event. In ordinary textual languages such things are given by the line sequence (calling order). For graphical programming the events determines the order.

This getter ***FBoper*** can be used more as one time in the graphic. It is not an only repeated graphic presentation (due to 5.3 Show same FBlocks multiple times in different perspectives), it is really each an operation call for each graphic presentation.

That fact is more able to explain with the following example:

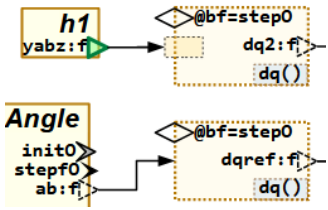


Figure19: odg/FBoperInOut

Here two times the same operation of the same instance is called, but with different input values. The instance is in both cases the **bf** instance, textual given with the **@** connector (see chapter 6.5 Connection possibilities page 58).

It means, the same operation for the same instance is used twice, but with different input values. That's why it is important that the operation itself do not change internal data in the aggregated FBlock with name **bf**, given in the aggregation as connection.

The called function should be designated in C language as

```
void dq_Bandpass(Bandpass const* thiz
  , float_complex x, float_complex* y1);
```

or just in C++

```
void Bandpass::dq(
  float_complex x, float_complex* y1) const;
```

The reference to the type (to the data) `Bandpass*` is `const`. , also in C++ language given with the `const` on end of the operation declaration, regarding to the implicit `this` pointer. In Java language unfortunately an adequate designation does not exist (**`final`** does others). This `const` designation can be seen as contribution to the ***Functional Programming Approach***. It means, the output is only determined by the input (also the referenced data of input pointers, means the data of the instance), but no side effects occurs. This is also the approach for this ***FBoper*** constructs in UFBgl.

Also here, **`=step0`** on the aggregation means, that the FBoper can be executed only after valid **`step0`**, it means after **`step`** was executed. In source code programming this should be regarded by the line order, call `dq..()` only after `step..()`. Here for graphical programming it is deterministic in this kind. After the evaluation of the graphic it is really a ***event-Join-FBlock*** with one input of the **`fb.step0`** to the expression prep input. The other input to Join comes from the data input before. But because the first FBoper is feed by a **`ofpZout`** pin which has valid data outside the event flow, here only the **`fb.step0`** is connected to the FBoper. This can be seen in the produced fbd file, for this example:

```
EVENT_CONNECTIONS
bf.step0 TO dq2_X.prep;

bf.step0 TO JOIN_dqref_X_prep.J1;
gref.stepf0 TO JOIN_dqref_X_prep.J2;
JOIN_dqref_X_prep.J TO dqref_X.prep;
```

6.4 Expressions inside the data flow

Table of Contents

The general difference between Expressions (FBexpr) and FBlocks is: FBexpr have no state. There are always calculations from input to output.

Expressions for data flow are presented by a figure (here a circle, but usual also a rectangle) of the style `ofbExpression`. This figure can immediately connected by `ofRef` connectors or simple `Default Drawing Style` for input and output, whereby the input connector can have a text for the expression.

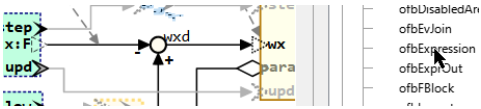


Figure20: odg\ExpressionExmp1.png

The name `wxd` is the text on the circle itself. It should be placed proper using the Dialog in LibreOffice: "*Format – Text Attributes*".

This is the simple form. Note, writing a text to a line with some inflection point is a little bit sophisticated in currently LibreOffice versions.

6.4.1 Expression parts as input

The other possibility is using a rectangle box with the style `ofbExpression`, in the following text referred to as ***FBexpr***: (*"Function Block as expression"*). The original outfit of the style is a dashed line as border. Small inner rectangle shapes with style `ofbExprPart` can be used for the expression inputs. The internal type of this elements is `DinExpr_FBcl` and hence ***DinExpr*** is written for that in the following text.

They can contain operators and also a factor as constant or as variable. The basic form to add and sub is:

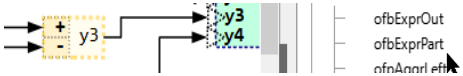


Figure21: odg\ExpressionExmp2.png

In opposite to the circle with lines, here is enough place and clarity to write a text associated to the expression input. This can be one of the operations known from

mathematics and logic in the following groups:

- `+ -` numeric ADD FBexpr with unary operator `-` possible
- `* / %` numeric MULT (DIV, Modulo) FBexpr with unary operator `-`: numeric `%` is modulo.
  The inputs can be multiplied (also without designation if at least one of this operators are given) or divide or calculating the module. A FBexpr with only a `/` operator builds the reciprocal from the input; only a `%` operator builds the modulo. The operators are generated in the order of inputs from top to down.
- `&` boolean or bit wise AND, with unary operator `~` possible for negate. At least one input (recommended the first) should have the `&`
- `| v` boolean or bit wise OR, with unary operator `~` possible for negate. The `v` may be better readable as `|`, hence recommended.
- `^` boolean or bit wise XOR, with unary operator `~` possible for negate. Note that also `==` and `<>` can be used for boolean and bits for an exclusively OR.
- `== != <> < <= > >=` For numeric, boolean or bit wise comparison, with unary operator `~` or `-` possible for bit wise negate or numeric negate. More as one inputs can be used. `<>` is defined for 'not equal' in IEC61499 and also Structure Text, which is translated to `!=` in C/++. If more as one input is used with `==`, all should be equal. Also `<>` means, all are not equal together. Elsewhere the relations are valid in comparison to the input before, or in comparison to the first input. The first input should have either the `==` operator or given without operator.

Mixing faulty operators cause an error while evaluation the graphic.

Look on the following examples:

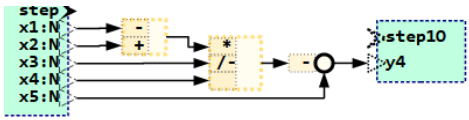


Figure22: odg\ExpressionExmpCombi.png

The Figure22: shows a combinatorics, the expression is

```
y4 = -((-x1 + x2) / (-x3) * x4 ) + x5;
```

The last expression block has the `-` as *DinExpr* immediately near the circle which is an `ofbExpression`. This is an alternative instead write the `-` on the line. But of course in the translated source expression line the `-` appears before the representing `(…)` of the expression before.

In the middle FBexpr the `*` on the 3th input is omitted because it is default, the expression is detected as multiply expression. Also the `*` on the first input can be omitted because the `/` is enough concise to determine this FBexpr as Multiply expression with one operand to divide. The `-` after `/-` is the unary `-` for the `X2` input. All of this should be intuitive understandable.

But to reinforce it look on a boolean example:

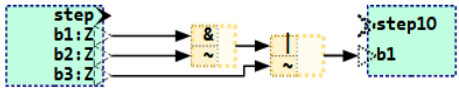


Figure23: odg\ ExpressionExmpCombiBoolean.png

This is

```
yb1 = (b1 & !b2) | !b3;
```

In C/++ Syntax. Because the data types are boolean in C/++ the `!` should be used for negation (NOT). If the data types would be `u w v` then the ~ will be proper. The code Input generation designates it automatically.

6.4.2 More possibilities of DinExpr

But there are more possibilities:

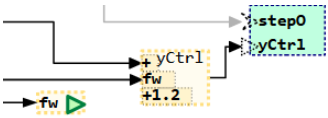


Figure24: odg\ExpressionExmpK2const.png

This figure shows an add expression, but the second input is also multiplied with the variable `fw` and the 3th input is a constant with the given value be added.

The variable `fw` should be able to find in the state variables of the model. It is wired as the `K2` input in the IEC61499 textual presentation. The constant value of the 3th Input is a constant on the `X3` input.

6.4.2.1 Variables in the DinExpr

There is also a possibility to write two variables in the expression input, but only if the input is not connected:

Figure25: odg/ExprExmp2Vars.png

The Figure25:shows right side a numeric integrator or += operation in C thinking, the input X1 is added and before multiplied with the factor `fd_f`. This may be done in a fast cycle, means should need only less calculation time. The factor may be variable, it is a time factor calculated as shown with the left FBexpr. That uses the `Tstep`, which is the cycle time, and divide it by the real integration time constant `Tfd`. This is done in another, a slower cycle because the `Tfd` value does not change so fast (possibility) and the division needs more calculation time (necessity to calculate not in the fast cycle). It is stored in a state variable `fd_f`, which is accessed by identifier in the right FBexpr.

The connection between the output `fd_f` and the input for multiplying in the right FBexpr can be drawn here with connections. But, the calculation of the factor may be placed on another page, the factor may be used more as one time, it may be more obvious if both are separated.

It is adequate also for the values of `Tstep` and `Tfd`. That are variables, should be known and locate on other pages on the graphic, a wiring is not necessary, it is more confusing as helpful. Where to find this variables? Of course either as input values of the module or as output of a parameterize FBlock.

6.4.2.3 Syntax/semantic of DinExpr

A constant or a variable in the DinExpr plays often the role of a multiplier, but can also be used to divide, to add and subtract or to mask for bit operations. That's why the syntax of the DinExpr should be exactly presented:

```
DinExpr::=[\.<$?componentAccess>
| \[ [<$?arrayIndexVar>|<#?arrayIndex>] \]
|[<$?variableX>|=<#?number>|='<*'?string>'|]
 [<opK> [<unaryOpK>]]
 [<$?variableK>|<#?numberK>]
 [[<unaryOpX>]<opX> ]
].
```

The syntax is given using ZBNF-Syntax: The meta morphemes are written in `<morpheme>` or `<..?semantic>` whereby `$` as morpheme means: any identifier, `#` is any number, `*'` means any String till the end character `'`. The semantic helps to explain. Plain text is written immediately without quotations. Special symbols `<>[]{}.` are used for syntax expressions. If they are necessary in the plain text, a `\` is written before. After \ `[…]` is an option. `[…|…]` is an alternative. `[…|…|]` is an alternative option.

- The *DinExpr* can be empty.
- If the text in a `ofpExprPart` shape starts with a dot as `.name`, it is the name of a component of the variable on output of this expression. See 6.4.5 Set components to a variable
- Similar as dot, if the text starts with a `[` then it is an array store input. The text

designates the index either numeric **[0]** or via a variable **[ixVar]** or also via the second input if only **[]** is given.

For the next three possibilities the following is valid:

If the pin has an input connected, the constant is the multiplier and assigned to the **K..** input. Then continue on **variableK**. If the pin has no connection, the constant or also a variable is wired to the **X..** input as **variableX**. or **number** or **string**. It means one FBexpr supports also multiply its inputs with numeric state variables, which is often proper usable. Also for comparison constant values are proper usable.

- **variableX**: An identifier on first position can be the replacement of the non connected input. But if the input is connected it is the **variableK** after the omitted **opK**.

- **number**: The same is with a given number. If the input is not connected, it is a constant on the X-input. If the input is connected, then it is the **numberK**. The number can be given hexadecimal. A numeric given number is converted in the proper form due the type for code generation. For example writing `13.0f` instead **13.0** for a float operation.

- **string**: A String in apostrophes is notated as String as given in the IEC61499 representation. For code generation, it is used as is. That makes it possible to write for example **'M_PI'** to address a `#define-Makro` given number. Without apostrophes it would search a variable named M_PI, not found, produce a warning but let this identifier in the code. That is dirty. Also a complex expression can be written for code generation uses as is.

- **opK**: The second operand which is connected to the input K… can be operate with this operators with the input.

operatorK::=+|-|*|/|%|&|\|^|

The compare operators are not admissible, because for this comprehensive expression form they change the type to boolean.

- If the **opK** is omitted, the default is *****. **factor+** or only **factor** means, the input is first multiplied with the factor, then added. Also in a MULT term **factor*** means, the input is multiplied with factor, then both are multiplied with the rest of the expression term. Whereas **+factor*** means, the factor is first added with the input, then both are a multiply input in a MULT term.

unaryOpK::=-|/|~.

- **unaryOpK**: Also the second operand can have an unary operator after the given operator.

- **variableK**: The second operand can be either a variable of the module given as identifier which is connected to the K… input in the IEC61499 presentation.

- **numberK**: The second operand can be a number which may be converted by code generation to a necessary form. Also **0x1234**, a hexa number is accepted, but not converted.

- **stringK**: If the second input is given in apostrophes, it is designated as character string literal on the K… input as constant used as is for code generation. If the expression is a string expression (concatenation) then the code generation writes this **"string"**.

- **unaryOpX::=-|/|~.** The unary operator is regarded to the whole input for the expression term after a possible K input. For using an unary operator the **<operatorX>** should be written after. For example a simple **/-** means, that the input is subtract in an ADD expression, but before subtract the reciprocal is built as unary operation with the whole input. **var/-** means the input is multiplied by var, then the reciprocal of both is built, and the result is subtract.

- **opX**: Operator for the input:

opX::=+|-|*|/|%|&|v|^|>|\<|>=|\<=|=|==|\<>.

The operator for this expression is written at least right side. The syntax presents all possible operators. But as shown in 6.4.1 Expression parts as input only determined combinations are admissible. Note that a `\<` in ZBNF presents a single `<`.

The operation with X and the second input is always done with more precedence, it is in parenthesis for the generated code.

(see `FBexpr_FBcl#setOperatorToPins()`)

6.4.2.3 Some examples for DinExpr

TODO

6.4.3 Any expression in FBexpr

The `ofpExprOut` shape or also the text of the `ofbExpression` can contain both a function ***written with parenthesis***, for example `atan2()` or any expression written in the target language using `X1`, `X2` etc. for the inputs. The source code generation inserts this function or expression either as written or with an adequate derived code, see next. Some functions should be well known for graphical level. Specific maybe complicated functions can be written in the implementation level and called here immediately.

Look on a first basically example:

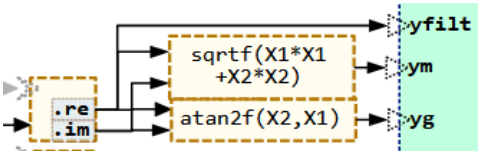


Figure26: odg/ExprAnyX1X2.png

The `ofbExpression` shape or block has not any `ofpExprPart` or `ofpOut` pins, it is not necessary. Input and outputs are immediately bonded to the expression block. The inputs are counted from top to down, and then right side from top to down, or also from left to right first top, and at last on bottom side, if necessary. The input pins has in this order the names `X1` .. `X99` so much as given.

While code generation, the identifier `X1` … etc. are replaced by the values which are connected on the inputs using the .code template scripts, see chapter 6.4.9 FBoper, operation for a FBlock.

Because often target languages such as Java or C/++ are very similar in expression writing, the expression notation in the graphic is compatible with some languages. With an adaption table function names can be replaced for a specific destination language. For example the here shown `sqrtf()` is known for C/++ language, for float calculation. For Java source code it can be adapted with `(float)Math.sqrt()`. This is done as part of the translation template.

Also for this possibility input `ofpExprPart` can be used to influence the inputs also with factors, or using constants or negate the input values.

6.4 4 Output possibilities

All shown expression examples till now have its outputs on the expression shape. In this kind the expression is not represented with a variable, it is an inline expression. The value is stored or used from the input pin after.

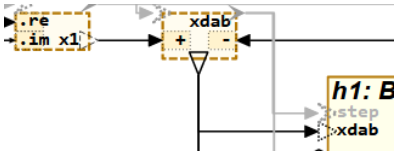


Figure27: odg\ExprOutpin.png

This example Figure27: shows two expressions with a pin symbol on output. A pin symbol forces creation of a variable in the generated code. Especially on forking the data flow (using for more as one input) as here for `xdab` it is sensible. The left output has the style `ofpDoutRight` which is a normal data output. This forces a stack local variable in the code. But here the variable is necessary to collect the both parts of the

complex value. If the expression is only used in one event chain, it is always ok.

The second expression `xdab` uses a style `ofbVoutLeft`, here the shape is rotated to 90°. This forces an instance variable in the `struct` or `class` of the module. The advantage is, it can be better visited in debugging. The variable can be used also in more as one event chains, which are more as one operations, but the data consistence is not guaranteed then, as usual in such situations.

The name of the output pin determine the name of the expression. If the output pin has not a name as for `xdab`, the name of the expression is the text in the `ofbExpression` shape box.

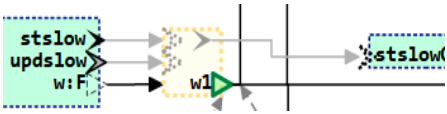


Figure28: odg\ExprOutStateUpd.png

The Figure28: uses an output with style `ofpZoutRight`. The letter `z` is derived from the https://en.wikipedia.org/wiki/Z-transform which is used for calculation, z is the stored (state) value. Hence it is set with the *update event*, here `updSlow`. The image shows the prepare and update events in gray, because there are automatically calculated. The input of the expression is here only one value `w`, the expression can have more inputs as shown in the chapter before 6.4.1 Expression parts as input. The expression is calculated with the prepare event, here `stslow`, due to the data flow. But the output of this prepared value, setting of the variable is done with the associated update event, it means after (or before the next) preparation calculation. It means all Zout variable have the state of the last step for the next preparation. In Simulink those are 1/z Blocks, so named *"Unit Delay"*, or also so named *"Rate transition"* FBlocks, from view of another event chain (means another sample time, or another operation in implementation. If the update operations are atomic, non interruptable, then all Zout data are consistent.

6.4.5 Set components to a variable

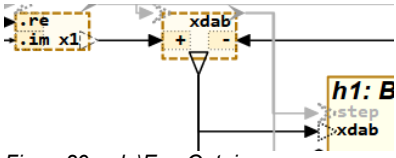


Figure29: odg\ExprOutpin.png

Input is .re or such or also [1] or [index].

The output must be a variable. The type should proper to the input descriptions. Simplest case: complex, type given with `:f`, `:d` or also an array given with `:F3` as float array or also `:f3` as complex array. More possibility use a structured type whereby the structure should be defined in the target language (in C in header file). `:structType` see 6.4.8 Type specification in expressions

Generally variables as expression output can be drawn more as time. If the expression has no input, then this variable can be accessed, not set. If the expression is this kind of set a component, different components can be set to the same variable, on different positions (also pages) in the graphic. The variable is only existing one time. The type need to be given only one time. If the type is given more as one time, it must be equal.

6.4.6 Output with ofpExprOut

The graphic style `ofpExprOut` can be used to define an output for an inline expression, but with a called function. This results in the same as shown in 6.4.3 Any expression in FBexpr, this text can be also notated as text in the `ofpExpression` shape. The difference is better handling in graphic.

In this case the name of the FBexpr FBlock in the IEC61499 presentation can be given as identifier in the expression FBlock.

The function designation can also contain a type for the output and also specific types for the inputs, writing after `:`, see next chapter

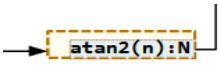


Figure30: odg/ExprAtan2n.png

The Figure30: shows an atan2() operation which takes a complex value as input and outputs a scalar number. To translate it, firstly the type letters for maybe non full specified values are replaced by the forward propagate types, for example results in `atan2(f)=F`. With this text the source code generation searches a proper translation, exact this String is used as identifier for a OutTextPreparer sub script which is then used for code generation. This sub script can be

```
<:otx: atan2(f)=F : fbx, cacc>
<:set:dinVar=genValueDin(fbx.din[1],'')><: >
atan2f(<&dinVar>.im, <&dinVar>.re)<.otx>
```

which results in generated code for example to `atan2f( cvar.im, cvar.re);` which calls the atan2() as given in C/++ destination language.

The designation of the output (here `N` as any numeric) is important, elsewhere the type propagation forwards the input type to the output. It does not know that the atan2() operation outputs a scalar.

6.4.7 FBexpr as data access

If you look at Figure27: the you see on input `.re` and `.im`. This expression needs an output variable, which collects the real and imagine part and delivers a complex value.

The opposite expression is

Figure31: odg\ExprOutReIm.png

Here the outputs are drawn in graphic style `ofpExprOut` with internal text starting with the dot. On access (without output variable) from the input the adequate part, here from the complex value, is accessed.

The same as for `.re` and `.im` can be done for elements of an array. The collect (on the `ofpExprPart`) and the access (on the `exprOut`) should be written in form `[2]` where as the `2` is the immediately constant index to the array. But also a variable index is possible, write `[X2]` where `X2` is the value on the second `K` input of the expression. The size of the array variable on a collect expression should be dedicated, given with the type specifier, see next chapter.

6.4.8 Type specification in expressions

In the texts of the expression inputs and outputs (`ofpExprPart`, `exprOut` and also the pins on output `ofpDout`..., `ofpVout`... `ofpZout`... the text on the pin can contain a `:...:Type` as suffix. This can be written after a variable name (for the out pins) as also for all other possibilities for the expression part and output. The type designation follows chapter 6.1 Data types. The types should be semantically sensible. In this kind the size of an array can be defined, see example:

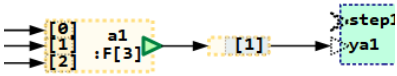


Figure32: odg/ExprArray.png

Here the text to the output is wrapped, this is not important. But it ends with `:F[3]`, means it is a `float[3]` array in C/++ or also Java language. The right expression then accesses the element 1.

6.4.9 FBoper, operation for a FBlock

The FBoper as shown in the following Figure can be seen also as part of the expression flow, hence it is here mentioned. But such an FBlock is intrinsically an concept of the FBlock and classes.

See chapter 6.3.4 FBlocks for operation access in line in an expression - FBoper on page 47

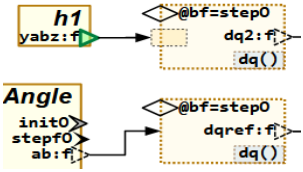


Figure33: odg/FboperInOut.png

6.4.10 FBexpr capabilities compared to other FBlock graphic tools

Compared for example with the known IEC61131 FBD diagrams for industiral automation programming the last one contains usual a lot of FBlocks for specific operations, for example ADD3, ADD3, SUB2, AND with two inputs which can be cascade etc. In comparison to the possibilities of UFBgl it needs some more FBlocks in the diagram, the diagrams will be more voluminous but not more clearly. It is a entanglement in details. Often a textual written expression is more proper understandable then a lot of wiring.

Expressions in the FBexpr blocks are related to the target language. This is an advantage for programming, it's clear what's happen. The expressions in a familiar target language are quite easy to understand from a customer level (with focus on mathematics). In opposite using a specific formula writing style of any specific tool needs also the understanding of this tool, sometimes it is more specialized as the familiar used programming languages.

Also a lot of specific numeric function blocks for sin, cos and whatever are lesser helpful as a simple written `sin()` in the graphic box.

Some graphic tools have also some parameters for expression blocks, which are hidden (not shown) in the graphic. They are editable in a ***"parameter dialog"***. Often this is for the data types. Here also the types are shown with its simple short designation.

6.5 Connection possibilities

6.6 Drawing and Source code generation rules

Table of Contents

C/++ is only one example for a target language but it is the most familiar, hence it is used her for description.

6.4.1 Writing rules in the target language used from generated code from UFBgl

Often some core functions are offered, or they are anyway existing in the target language. Follow the idea of system levels, modules and black boxes, such functions are independently tested and documented (independent of an application) and can be really seen from the graphic level as *“black box”*, understandable what they do, but the inner operations are not topic of study, they are presumed as well.

Of course the provided functions in the target language should be proper to the source code generation of the ***UFBgl*** with whose event-data and the Object oriented concepts. That is usual possible with some wrappers around legacy software or, for Object Orientated C language, this concept is anyway proper.

Details of the following rules can be adapted in the templates for Code generation, see chapter 8.10 Code generation due the to event flow page 118. For the standard given templates for ***emC*** (*embedded multiplatform C/++*) it means:

- Data associated of one module with name `MyModule` should be assembled in a `struct` with the name `MyModule_s`. The leading `_s` is used to differ the module’s identifier with the class name without `_s` if C and C++ are mixed (may be recommended). Note: Use the typedef style

```
typedef struct MyModule_T {
  int32 myVariables;
} MyModule_s;
```

- The usable type is then only `MyModule_s`, and not `struct MyModule…` as often seen. It is more simple and obviously.

- You can have a class encapsulating the `struct` definition:

```
class MyModule : MyModule_s {
  inline void step (...) {....}
};
```

The class wraps the:

- C-language Object-Oriented Operations which should be written as:

```
void step_MyModule(MyModule_s* thiz, ....) {
  .... }
```

- It means there are operations in C which are strongly related to the data with the data pointer named `thiz`. It is similar the C++ `this`, but written with `z` to allow mix with C++ and use a C++ Compiler for C files (which may be seen as recommended).

- The names should be `step_`, `upd_`, `init_`, `ctor_` following with the Module name, as default. That are the default

names for the events automatically created and used, or spedific names determined by the evin of the FBlock.

6.4.2 Life cycle of programs in embedded control: ctor, init, step and update

The UFBgl is first for embedded control programming with graphical support. For that speak about the life cycle.

Usual in embedded control programs does not use frequently allocated memory because of the possibility of fragmented memory, and also there is no process management which can free the whole memory if an application is closed. Normally an application is never closed. That's why allocation of memory is only usual on startup. All instances are prepared, and then the program runs till power off or reset. In rare cases specific applications are added on demand and also removed if there are no more necessary, with a may be specific memory allocation handling.

This is other than in PC programming, where a running program is a job, used on demand, finished and removed if it is no more necessary – or it hangs. An embedded application must never hang, it should run without restart also some years.

The UFBgl supports that thinking and regards three phases:

- `ctor`: This is an event or operation call to construct one FBlock either independently or with knowledge of values (data inputs) and other FBlocks (as aggregation) which are already constructed before. This means that the knowledge of data is consistently tree-like.

Because of specific handling of construction the operations for the constructions must start with `ctor` and other operations must not start with `ctor`. To fulfill this necessity for legacy code you can write simple wrappers (maybe as `#define` or as `inline`) which does not cause additional code.

```
#define ctor_MyModule(THIZ) \
  legacyConstructionRoutine(...)
```

The often seen rule to write macro names only in upper case is of course not recommended here. Or better use the `inline` possibility available since C99 also for C language.

- **`init`**: A specific initial phase is necessary if there are circular dependencies between FBlocks. To fulfill a correct initialization one FBlock should be deliver proper initializing data, but this FBlock may depend also from other FBlocks. Then the initializing can be done only step by step. A proper example is: Aggregation between two FBlocks each other, maybe also to inner instances of these FBlocks (ports).

That's why the `init_MyModule(...)` operations are executed in a loop till all is ready. The basic form for that is:

```
ctor_FB1(&dataFB1, args);
ctor_FB2(&dataFB2, args, ... dataFB1);
//
bool bInitOk;
int ctAbortInit = 10;
do {
  bool bOkPart;
  bOkPart = init_FB1(&dataFB1, ... &FB2);
  bInitOk &= bOkPart;
  bOkPart = init_FB1(&dataFB1, ...&FB1);
  bInitOk &= bOkPart;
} while(!bInitOk && --ctAbortInit >=0);
```

As you see here (example) the `ctor_FB2` can use the `FB1` because it is always constructed, but not vice versa. But the `init_FBx` can use the (already existing, constructed) other FBlocks. The `init_` operation checks whether it has all necessities gotten from the other FBlocks, then it returns true. Else it returns false. The `init_` operations are all called one after another, in a proper but, not strong order. They are called repeatedly in this loop. But the loop is aborted if it needs too much iterations, which are intrinsically a result of a software error (any FBlock is not satisfied with the other ones). It means on

`ctAbortInit <0` an emergency handling (search the cause) is necessary. The maximum number of necessary `init_` loops should not greater then the number of `init_FBlocks(...)` in the loop. Then also in a revers sensitive order called `init_FBlocks(...)` delivers the data from the last called to the first one.

Because of this specific handling, the operations for initialization must start with `init_` and other operations must not start with `init_`, or basically, the `init` event should be used for `init` in the graphic. To fulfill this necessity for legacy code you can write simple wrappers (maybe as `#define` or as `inline`) which does not cause additional code.

```
inline init_MyModule(MyModule_s* thiz, ...) {
  legacyInitialization_Staterments(...)
}
```

- **`prep`** or **`step`**: This is the often cyclical called step routine for the sampling time. Such operations are often called immediately in interrupts. It is also possible to call lesser prior routines in a back loop of a simple controller organization without a specific RTOS (*RealTime Operations System*), or just also in a specific RTOS. *prep* comes from *prepare* in opposite to *update*.

- **`upd`** operation for *update*: In controller algorithm with often solves differential equations it is necessary first calculate the new state of all inner variables using the previous (old) state, and then update all states at ones.

If new and old variables are sometimes used confused, the results are often not entirely correct. With sensitive algorithms (e.g. filters) they are completely wrong. This is often not properly taken into account. The code generation of UFBgl respects this. The basic form of this is:

```
interrupt opeationOneStep (...) {
  prep_FB1(&dataFB1, ... &FB1, &FB2)
  prep_FB2(&dataFB1, ...  &FB1, &FB2)
  upd_FB1(&dataFB1, ...)
  upd_FB2(&dataFB2, ...)
```

As you see, first all ***preparations*** are done for new states, using the current ones. Then ***update*** the new states to the current ones comes for the next step. This is similar also of D and Q on Flipflops in digital logic.

The **`upd`** operations helps also for data consistence. If a whole update operation (consist of calling some **`upd`** operations for the inner FBlocks) are executed in a locked state (with mutex) or just in *disable interrupt* state for a simple non RTOS controller software, then interruptive routines gets always consistent data from its interrupted operations (tasks). The update operations usual should not need longer calculation times, because the do only copy data.

The **`ctor`**, **`init`**, **`prep`** or sometimes **`step`** and the **`upd`** are the basically existing events for execution. Regarded in the models by the user, regarded by source code generation.

6.4.3 Using events in the module pins and FBlocks, meaning in C/++

See chapter 5.6 Using events instead sample times in FBlock diagrams page 26

The events in an UFBgl diagram replaces on the one hand the often used “*sampling times*”, on the other hand they are really events in an event controlled execution. But for code generation the execution of an event in a FBlock is one operation. That’s the important rule.

But the events should not be elaborately shown and wired in the diagrams. Similar as associating sample times to data in other FBlock graphic tools, the events need primary only be given in the module’s pin definition (style **`ofbMdlPins`**). Not only the wiring of events in the diagram (event connections) can be omitted, also events in FBlocks can be omitted, if the association with the data is unique.

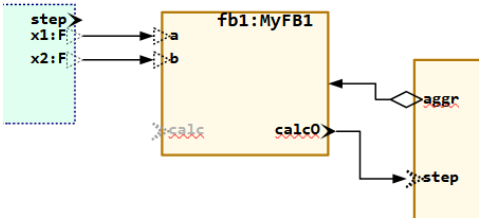


Figure34: ExmplEvDeflt_calcOstep.png

Look for a not simple but should be obvious example in Figure34:

- The both input values **x1** and **x2** are associated to a module input event **step**, usual the module gets a `step_..(..., float x1, float x2)` operation.

- The **fb1** has a named output event **calc0**. Hence for the input variables the input event, here drawn in gray as not active, is **calc**. The called operation is `calc_MyFB1(…)`. If the FBlock would not have any event designation, a **prep** event will be created as default.

- But notice, that an event – data association can also be drawn on another position of the graphic, proper to the rule "*Any element of the functionality can be shown more as one time in different contexts*" described in chapter 5.3 Show same FBlocks multiple times in different perspectives page 22. If the data inputs are associated to another event there, this is valid. Then the here shown **calc0** does not influence the input data association between **calc0** is an output event.

- For this example it is shown in the graphic that a called `calc_...(fb1...)` operation is followed by a `step_..(fb2...)` operation of the next FBlock because this is dedicated by the here shown event connection. In this special case the **fb1** has no data output which should elsewhere determine the calculation order (or just event connection). Hence it should be dedicated by the drawn event connection.

- The aggregation from the second **fb2** to the **fb1** needs an initialization. For that both FBlocks gets an **init** → **init0** event pair per default (as nowhere other it is dedicated in another way, just as default). The own address of the **fb1** as "port" output is related to the **init0** event, and the aggregation is related to the **init** event of the right FBlock.

- And also for construction a **ctor** and a **ctor0** event is associated to all FBlocks which are not expressions.

With this simple rules the code generation from UFBgl to C language in the default version (can be adapted, see TODO) is compatible with your basic function blocks in C language.

Then you don't need specific extra definitions outside of the Libre/Open Office graphic.

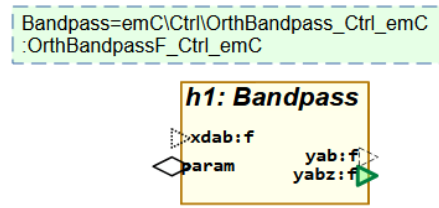


Figure35: FBlockSimpleUsage.png

This is the only necessity in the graphic to use it together with the existing code in C/++ language:

- The green box is of style **ofbImport** and declares the alias `Bandpass` in the graphic as full Module type `OrthBandpass_Ctrl_emC` which is the module's name in C language (see emc/.../Ctrl/OrthBandpass.html (www).

- The input events **step**, **init**, **ctor** and the output events **step0** and **init0**, are automatically created because here events are not defined.

- Because at least one output with the graphic style `ofpZout...` is given, also the input event `upd` and the output event `updO` is automatically defined.
- All data inputs are associated to the `step`, all data outputs which are not `ofpZout` are associated to `stepO`. All `ofpZout` outputs are associated to `updO`.
- All data inputs and outputs should be marked with the used types, here `F` for `float` and `f` for `complex_float`. This designation is only necessary ones if the FBlock is more as one time used.
- All aggregations, also associations are associated to the `init` event. They are inputs for the `init` event or just the `init_Module(thiz, param)` generated C operation though the direction of the connection is to the referenced class, to initialize the reference.
- All Ports (not in example) with graphic style `ofpPort...` are associated to the `initO` event. They are outputs usable for other `init` inputs due to there reference connections.

6.4.4 More possibilities, definition of special events

If your target language module has more operations then the `ctor_…`, `init_…` and `step_…`, or you want to use another name instead for `step_…` then you can define your own events.

- TODO event with data in one block: It is for the data, an aggregation is not associated, it is associated to init.
- event in one block only with aggregation: It is instead init
- You can have more as one graphic block to show specific data and event relations.

TODO figures, program, test.

6.5 Converting the graphic – source code generation

As fast mentioned also in chapter5.8 Source code generation from the graphic page 29, 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, especially for versioning.

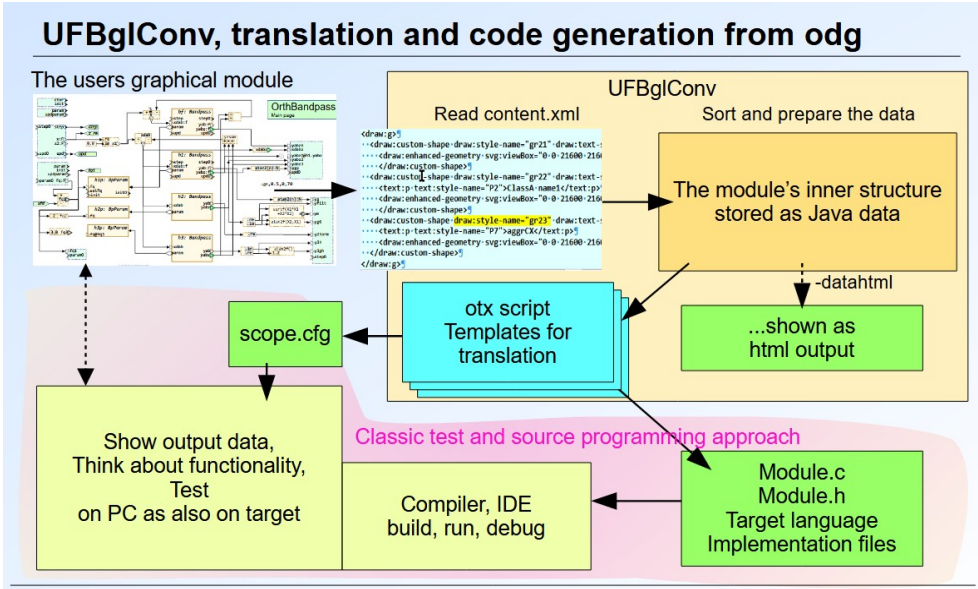


Figure36: Fbcl/UFBglConvAndTestSlide.png.

The slide above shows the working flow with UFBglConv code generation. The classic approach is the magenta area on bottom side: Manually written code, test and compare with an only-documented module architecture and design. That is also valid, but supplemented with an automatically code generation from the graphical module, as shown on upper side in the slight. For code generation proper readable and adaptable templates are used as otx scripts.

This otx scripts have a syntax described in:

vishia/Java/pdf/OutTextPreparer.pdf (www)

6.5.1 calling conversion with code generation

The code generation from Open/LibreOffice odg files can be performed with:

```
@REM This file is the batch file to call java and similar the argument file.
cls
if not exist ..\cpp\genSrc mkdir ..\cpp\genSrc
if not exist ..\fbcl mkdir ..\fbcl

@REM use --@file:label, the file is this file itself as %0
java -cp ../../../tools/vishiaBase.jar;../../../tools/vishiaUFBgl.jar org.vishia.fbcl.UFBglConv --@%0:args
@echo off
REM the arguments are written in lines which are comments for the batch processing ::
REM characters before the label args are identification for the arg lines, but not part of the argument,
REM one space and ## after the args label defines remove trailing spaces and remove comments after this ##
REM --- is a commented argument for the java main routine
```

```
::args ##
::---dirStdFB:src/libModules_fbd/fbd
::---codeTpl:d:\vishia\fbg\source.wrk\src\srcJava_vishiaFBcl\java\org\vishia\fbcl\translate\cHeader.txt
::-dirGenSrc:../cpp/genSrc
::---dirCmpGenSrc:src/ExmplGenSrc/cmpGen
::-dirFBcl:../fbcl
::---dirCmpFBcl:src/ExmplGenSrc/cmpGen
::-dirDbg:../fbcl/dbg                  ## output directory for some log files for data debugging
::---ifbd:path/to/file.fbd          ## for a inner module
::---ifbd:path/to/othermodule.fbd   ## can be given more as one
::-odg                              ## writes an file.odg as inner data presentation
::-oxmltest                        ## possibility to write back the read content.xml
::---oxmldatahtml
::---datahtml                       ## possibility to write the internal data in html
::-i:../odg/MyExample.odg             ## The input odg file to translate
pause
```

This is the whole content of the batch file **src/MyExampleComponent/makeScripts/genSrc_odg.bat** in the example download, inclusively some explanations.

The input file is the last argument after **-i:.** More as one such argument, hence more input files are possible. A Module can have some pages in more input files, all they are summarized before code generation of the module. Also other modules can be read.

For used modules the rule is: First name the used module, then the using module in the **-i:** argument. Then the using module can participate on the existing definition of the used module. Elsewhere some default mechanism are effective, if the used module is not full specified while using, and this will be seen in a not proper code generation.

Especially files which are present in the target language, not graphically drawn, can be inputted by an interface description in IEC61499 syntax (textual). This interface description may be simple proper to hand-written, but also an automatic translation from C-header files can/should be used, see TODO later.

The extension of the **-i:** file determines how to read it. **.odg** is OpenLibreOffice, **.fbd** is a IEC61499 file. **.slx** should be for Simulink (yet TODO), all other graphic sources should/can be translated adequate.

The **-dirStdFB:** is used to look for files, which are used as modules but not given as **-i:** argument. In this (may be more as one) directories proper module files are searched.

The three **-dirGenSrc:** **-dirFBcl:** **-dirDbg:** describe where the output files should be stored. The name of the output files are name of the module in the **ofbTitle** shape in the graphic, with the proper extension.

The three directories **-dirCmpGenSrc:** **-dirCmpFBcl:** **-dirCmpDbg:** are only for internal test to compare results with given files after code changes (test evaluation).

The **-codeTpl:** option (possible more as one) describes paths to otx files (OutTextpreparer) for code generation. If this argument is not given, internally files for C code generation are used. See next chapter.

- The option **-odg** forces output of a textual file which documents the internal graphic structure as text (not in IEC61499 syntax). In the necessary given **-dirDbg:** directory. The advantage in opposite to an fbd file is: If a FBlock is more as one time drawn, all draw instances are reported. But the summary of the FBlocks for its functionality is not contained there, it is in the fbd file.

- An fbd file is output always if the **-dirFBcl:** directory is given.

- **-log** writes a log file for example with the execution order of data type propagation and event propagation in the given **-dirDbg:** directory.

- **-oxmltest** forces the output of the read **content.xml** file after reading (check of the correctness of **XmlReader**, or also look for details in the graphic file).

- `-oxmldatahtml` writes the read XML data (Java internals) in a readable html file.
- `-datahtml` writes the prepared module data (see chapter 8.1 Data Model data classes page 91 (Java internals) in a readable html file.

6.5.2 Templates for code generation

The code generation is controlled by templates. Hence the adaption to any programming language and also to any rule set for a given programming language is possible.

The templates can be contained in more as one file. Any file contains the rule for some parts of code.

7 Discussion about graphic presentation approaches and implementations

A graphical Function Block Diagram (***FBD*** or also FBlock diagram) builds the content and interface of a Function Block type (FBlock type). The top level FBlock diagram is also intrinsically a FBlock type.

The content and interface of a FBlock type can also be described with the textual FBlock syntax given in IEC61499 see [IEC 61499-1/Ed.2] chapter B.2.1 Function block type specification.

This document is related to embedded software more than to automation control software. The difference to automation control is mentioned in some notes.

For embedded software the code generation (C/++) is an important topic. This is the focus of the documentation.

7.1. Data and event flow

The graphical presentation shows the data flow and due to IEC61499 also the event flow. The event flow determines the execution order.

Pure data flow with Sample time designation versus event flow

In

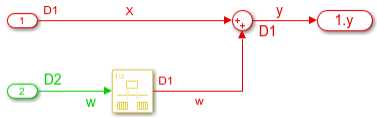


Figure 6: Smlk / Exmpl_SimpleStepTimes.png

comparison to other FBlock diagrams for example from Simulink, usual only the data flow is shown there. It determines the execution order, whereby different step times are used. Each sample time has its data flow. The Figure 6 shows that, the step times are shown here with colors and also with “D1”, “D2”.

This

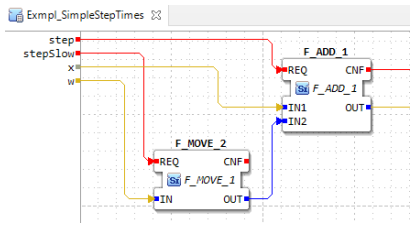


Figure 7: 4diac / Exmpl_SimpleStepTimes.png

system can be mapped to the system of event flow, whereby each event flow is associated to one sample time in Simulink.

Event flow on the same device → it is a simple execution order of FBlock operations

If all FBlocks or a block of FBlocks with a given event flow are arranged on the same device, one event flow can be code generated to an execution order of one operation of the module per module’s input event, which calls the operation of the FBlocks in the given event order. The operation of the FBlocks are that operations which are associated to one state entry caused by the input event. For that there are some variants, see next chapter FBtype Kinds and their usage

Event queue for execution, also for distributed devices

The other general possibility is using an event queue. The execution in the module (and also in sub modules) is determined only by the queueing and dequeuing of events regarding a first-in first out approach: Any execution of a FBlock's functionality puts the emitted event in the queue, which determines further execution. This event queue approach is necessary and possible, if the FBlocks of the diagram are distributed on several devices. The originally approach for IEC61499 is oriented to several dispersion automation devices, whereby the whole functionality over more as one device is shown in only one diagram (or more diagrams, but not sorted to the devices, sorted to software function module's functionality).

Of course, the event queue is combined with event or message transfer between the devices.

The combination of both is sensible. Often in embedded control one FBlock diagram is really associated to only one device. Then the event queue is not necessary. Code generation can be regard the execution order due to the events. But the possibility to disperse the execution to several devices may be also interesting for embedded software solutions as well as used for automation device software. Emitted events are then put in a transmission queue, the transmission is done via field buses or such, and received events via transmission are also put in the queue. While dequeuing they are processed. Of course this needs some milliseconds time, not for very fast control parts, but proper for set values, monitoring values, parameter changes and all these stuff.

Automatic detection of event flow

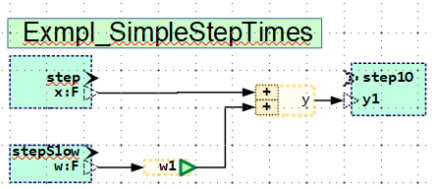


Figure 8: odg / Exmpl_SimpleStepTimes.png

For the Libre-Office Solution The events should be given on the input and output blocks (green), adequate to the given step times in Simulink on the ports. But the connection is done automatically due to the detected data flow. The event flow is written in the textual fbd file with IEC61499 norm due to the here shown graphic. The green triangle, style `ofpZoutRight`, is adequate to the rate transition. it is an output of the `stepSlow` used in the `step` event chain.

7.2. FBtype kinds and their usage (due to IEC61499)

In IEC61499 there are different types of FBlocks:

a) Simple FBlock with one operation: It contains only one function or operation, one input event, one output event. The output data are produced in combinatoric due to the inputs. Examples for such simple FBlocks are mathematic functions, expressions etc. The term "***Simple FBlock***" is also a term in the IEC 61499 norm.

b) Standard FBlocks with more simple operations, as Object Orientation with more events, but with simple association between the input event and output event. The term "***Standard FBlock***" is used in IEC 61499 for FBlocks which have a state machine, named ***ECC*** = "***Execution Control Chart***". Any state can have one or more associated operations, which are executed on state entry, and one or more associated output events, which are activated after the entry operation execution also on state entry.

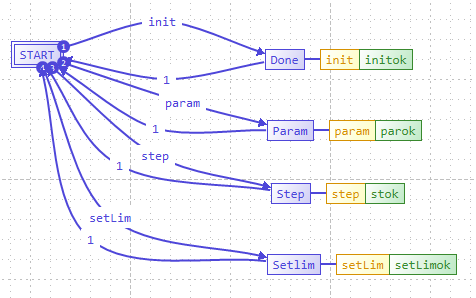


Figure37: SimpleRegularStmn.png

If this state machine is simple regular, then any input event is associated to exact one immediately coming output event. ***simple regular*** means, the `IDLE` or `START` state is triggered by each of the existing events, forces entry to exact one destination state, which unconditionally goes back to `IDLE`. Then each event is associated to one (or more) operations. Each operation may set outputs, or change the internal numeric state (not the ECC state) of the FBlock.

After execution of each operation exact one output event is created. This is shown in the example ECC right side.

This kind of Standard FBlocks are similar as the behavior of a class instance in Object Orientation. The input event is adequate an operation call. Really, the operation for the input event driven state change is executed and outputs are set. The only one dedicated output event comes immediately, hence the processing in the thread can be done straightforward. The events simple determine the execution order if the next FBlocks are associated to the same device.

- **c) Standard FBlocks with State machine** with more events, with a real maybe complex state machine. For that FBlocks the coming output events after an input event depends of the inner state of the machine. It is not simple predictable. Hence, the code generation can not participate from. If the following FBlocks in the event queue are arranged on the same device and hence should work in the same thread, the following operations the module should be called in execution order controlled by a bit mask value, which presents one or more coming output events. This bit mask value should be delivered from the FBlock, proper to the set output events. The other variant is: Using an event queue and processing it either in the same device or also possible send the event to the other destination device where the following content of the module is arranged.
- **d) Composition FBlocks with dispersed operations**. A ***Composite FBlock*** in terms of IEC61499 contains the composition of some FBlocks which are wired together with a data and an event flow. The member FBlocks can have any type of this list. It means the execution order or operations of a composite FBlock can be complicated, not well obviously. For usage (view from outside) the code generation offers some operations associated first to the input

events, but builds so named meta events which presents the inner event flow. If the inner FBlocks are on the same device, then with the meta events a proper order of execution can be found, working without event queue.

- If the inner FBlocks are not on the same device or an event queue is used for other reasons, then the inner execution uses the centralized event queue (one per thread), the processing of the event queue is then the execution.

7.3. Construction, init, run with several step times or events and shutdown

Coming from source code programming (C/++) the life cycle of a running software application can be differ to general three phases:

- **Construction:** Getting memory to run, set initial values. The construction phase is related to the ***constructor*** (ctor) in some programming languages or also with the initializing of memory before entry in `main()` in C language applications. It is the first phase of startup.

- **Initialization:** The initialization should be separated from the construction, because setting the correct initial values to run needs communication between several parts of the application, it presumes the construction. The initialization of one part can depend on finished initialization of another part, which delivers the values for the own initialization. Also a mutual initialization is sometimes necessary, also aggregations of modules each other. For that initialization needs loops. The initialization should be finished in a less number of loops. Any module should check its state of initialization and signal the finished state. If all modules have finished, then the initialization phase can be finished.

Often this initialization phase is not proper provided in some platforms. It should be cared about.

- **Run:** This is the working phase till the device is down. It is determined by physical events (timer, signal input) and often organized in fix sample or step times, and also event driven actions. This is also for simple devices with poor controllers and powerful devices.

In simple platforms often cyclically triggered interrupts does the work of time steps. Additional the back loop can handle event based actions one after another, whereby sometimes an event queue organization is not used, instead setting some bits etc. controls the actions.

In powerful platforms also cyclically interrupts may do the work for very fast step times (microseconds). Interrupts are not only responsible to handle hardware event sources. But the bulk of work may be done in a ***Real time operation system*** or a specific proper framework.

- **Shutdown:** For embedded applications shutdown is done by cutting the power supply. But hardware outputs should go in a save state. If this is not guaranteed by cutting power supply, or some actions should be done in communication etc, this can be seen as a part of the **Run** phase, after that the readiness to shut down is signaled.

Sometimes actions should be done to save values. This should be organized with interrupts or events during the phase of detecting cutting power supply from outside and the really loss of voltage for work, where the capacitors of the power supply loose its charge.

The phase of shutdown should not be confused with the ***destructor*** which is known for example in C++ language or the topic of ***garbage collection*** and ***finalze*** in Java. This mechanism are proper for temporary 'shut down' of some modules while the system is running.

7.4. Prepare and update actions

In some situations of calculation especially on resolve differential equations first all new values should be calculated starting from the current values (the state). In a second step all new calculated values are set to the current ones, the new state.

In mathematics this is the standard Euler method (from the mathematics Leonhard Euler, 1707 - 1783): https://en. wikipedia-.org/wiki/Euler_method.

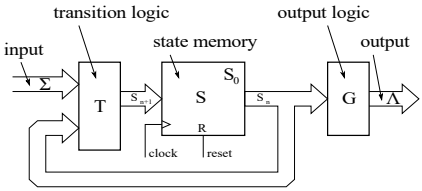


Figure 9: Moore automat

To calculate the new values, exclusively the old values should be used in all parts of the whole system of equations. Only then the solution is mathematically exact. This is the ***prepare*** phase. After them, or before the next step, the new values should be declared as current state, that is the ***update***

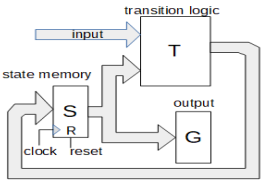


Figure 10: Moore automat 2

Also the theory of digital machines from Moore and Mealy based on this approach. Look on *Figure 9* and *Figure 10*. The *Figure 9* is from https://en.wikipedia.org/wiki/ Moore_machine. It shows the ***prepare - update*** concept in a proper kind for the Moore state machine. Here the Block T for Transitions is the ***preparation***, calculates the new state for D-Inputs of the FlipFlops, and the Block S is for ***update***, saves the prepared state as current one. This is classic.

Exact the same is drawn in *Figure 10* right side.. Only the positions are a little bit changed. But compare it with the next image:

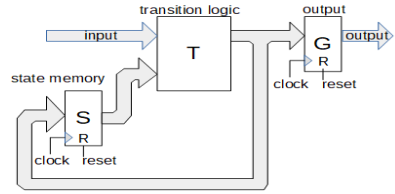


Figure 11: data flow with qout

In difference to the image Moore automat 2 above only the FlipFlops which presents the output state are separated from the other FlipFlops for the inner state. But also output logic is removed, the output functionality is built immediately from the logic block. This is a special more simple case of the Moore automat (sometimes named as '*Medwedew-Automat*'). If states are necessary also for output as also as inner state, the FlipFlops are twice.

The *Figure 11* opens up an understanding of what happens during signal processing in control technology for analog variables or for automation processing. It is primarily the same

But the output registers are formed by the physical output, the digital-to-analog converter, also the transfer of information to another device that outputs or processes it, or setting a new pulse width for electrical converters, etc. These outputs are assigned to the next step time, just as the outputs of the flip-flops in the digital automaton are the state of the next clock period.

This is a general approach, separating between ***prepare*** and ***update***. This general approach can be subverted for certain solutions.

- All operations to calculate a new state from the old state are done in ***prepare***.
- The ***update*** refreshes all current values of all FBlocks to the before calculated prepared values.

For that it is to difference between FBlocks, which are only combinatory and state less. That FBlocks are used in ***prepare*** chains, or also in calculations for the update. FBlocks with a state can have also a prepare event input, but have also an update event input which updates the new prepared state to the outputs.

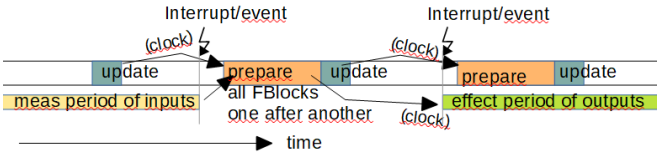


Figure 12: Timing prepare, update and hardware access

7.4.1. Example prepare and update for boolean logic

Exact the same approach is also used for boolean logic with D-Flip-Flops: The next value (as booleans) is ***prepared*** by logic on the D-inputs of Flipflops, and then all together on the same time are ***updated*** to the Q-output with a clock edge.The image above shows any processing signals (with the AND) which uses a value from the previous step time.

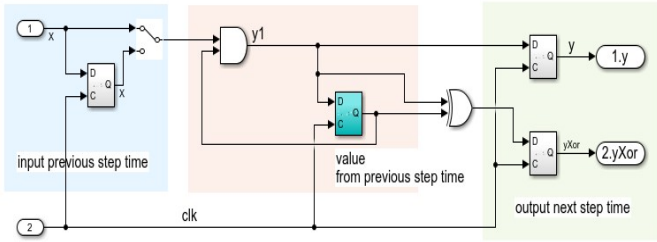


Figure 13: Example binary logic prep & update

One result of preparation is the signal `y1` which is output as `y` valid for the next step time. For that the outputs on an IC (for example FPGA) have DFF on the pins. The signal is '***clocked***', it comes time synchronous to a central clock. But the same signal is also used in a module after, where it is compared with the previous state of the same signal. It means the difference of the output in time can be built, here evaluated with a XOR to detect changes.

7.4.2. State of the art, ignoring prepare and update concept

Outside of boolean logic and FPGA usual a proper order of calculation is often found to regard the correct relations between the current (old) and new values for solving differential equations. This is often so in ordinary C/++ development, as also for example in the event driven 4diac tool for IEC61499. Because the execution sequence can be determined with tricky precision of the event connections, an appropriate solution will usually be found for the modeling approaches.

See the next examples.

7.4.3. Example prepare and update in source text languages (C/++)

What about ***update*** and the state variables: Usual, in C++ language programming and also in automaton programming the output of the prepared values are stored in variables anyway. If this variables are just used as current values for the next step then the ***update*** process is already done with store values and used for the next step. Look at the simple solution in C programming for an integrate:

Cpp: Simple integrate

```
  yIntg += fIntg * x;
```

All is done with one statement, maybe with one machine code instruction. The old value is used, the difference is added as expression here from input and multiply the integrate factor, and the result is stored back to the only one integrate variable.

Because the proper solution is usual solved individually inside a module, regarding data dependencies and the correct calculation order, the ***prepare - update*** concept is not usually in focus. But sometimes small errors occurs which are not so obviously.

The simple form above is only possible if the old integrate value is no more necessary for any other operation later, after this operation the previous current value in no more existing. That's why look on a little bit more complex integrate process, the solving of a differential equation for a bandpass filter. As example you can visit www.vishia.org/emc/html/Ctrl/OrthBandpass.html, chapter equations This is a filter algorithm. The equations in C are programmed firstly as:

Filter algorithm in C integrates dependent two values

```
1: static inline void step_OrthBandpassF_Ctrl_emC
       (OrthBandpassF_Ctrl_emC_s* thiz, float xAdiff, float xBdiff)
2: {
3:   Param_OrthBandpassF_Ctrl_emC_s* par = thiz->par;
4:   float a = thiz->yab.re;             // store the current value of component yab.re
5:   thiz->yab.re = par->fI_own * thiz->yab.re;
6:                + par->fI_oth * ( thiz->kA * xAdiff - thiz->yab.im);    // integrate .re
7:   thiz->yab.im = par->fI_own * thiz->yab.im;
8:                + par->fI_oth * ( thiz->kB * xBdiff + a);               // integrate .im
9: }
```

The `yab.re` and `yab.im` are the both the current and also the new values after solving the differential equations. For an exact result it is very important to use the previous value `a` in line 4 instead the already new calculated value `yab.re` for calculation of `yab.im`. This is a simple solution. ***prepare and update*** are done also in one step, but the current value for the second equation is stored immediately in an individually variable.

But what is happen for this solution if the current values of the integrate variables are need for more operations, in this example

for a more complex filter for harmonics. Then it is better to have a systematic solution, which looks like:

Filter algorithm in C consequently with prepare and update

```
static inline void step_OrthBandpassF_Ctrl_emC(OrthBandpassF_Ctrl_emC_s* thiz
  , float xAdiff, float xBdiff
) {
  Param_OrthBandpassF_Ctrl_emC_s* par = thiz->par;
  thiz->xadiff = xAdiff; //store for evaluating (phase) and debug view
  thiz->yab.re = par->fown * thiz->yabz.re + par->foth * ( thiz->kA * xAdiff - thiz->yabz.im);
  thiz->yab.im = par->fown * thiz->yabz.im + par->foth * ( thiz->kB * xBdiff + thiz->yabz.re);
}

static inline void upd_OrthBandpassF_Ctrl_emC(OrthBandpassF_Ctrl_emC_s* thiz) {
  thiz->yabz = thiz->yab;    // update the current state z
}
```

For that two calls are necessary, first `step_…` to prepare the new values whereby the new values are stored here in `thiz→yab`. Right side in all equations this `thiz→yab` should never be used to build `thiz→yab` itself, don't mix old and new values, access always `thiz→yabz`. But for further operation the `thiz→yab` is accessible if necessary (as also the D-inputs of FlipFlops can be used to calculate further preparation phase D-values).

The `upd…` is the ***update*** operation. It stores the new state as current state for the next step. This assignment is intrinsically a fast `memcpy` from view of machine code.

The ***prepare - update approach*** needs two variables more, more memory, and the second update call is necessary. But the solution is more obviously and better able to review.

It is to decide which is more important, a very fast algorithm or obviously sources. Unfortunately the compiler optimization does not solve here this problem.

7.4.4. Example prepare and update in 4diac with MOVE-FBlock

The example of the simple integrate is also solvable by the simple calculation order controlled by the event flow:

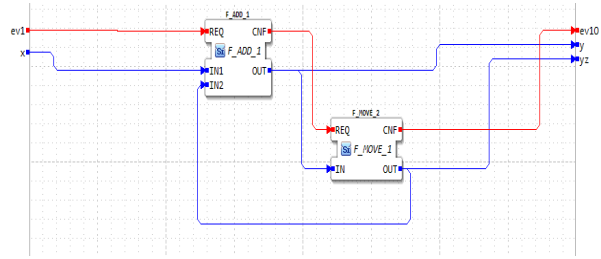


Figure 14: Example 4diac prep & update

Here the MOVE block is executed immediately after ADD and stores the output from the ADD FBlock for the next event occurrence which is the next step time. The previous value after integrate is no more existing after the event flow.

an extra event chain with `ùpd` and `updO`. The prepared result of ADD is available for further preparation which can also use the current (previous) value of the ADD, present in `y` and `yz`, for example to build a difference, the growth of the integrate between two step times, similar as the XOR in the boolean logic image Example binary logic prep & update

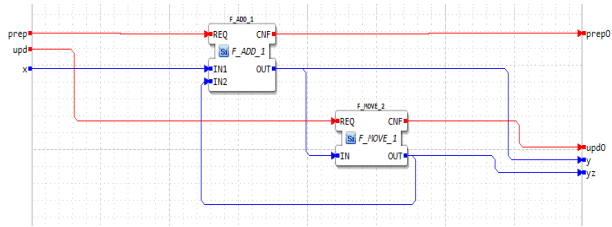


Figure 15: Example 4diac prep & update

This is almost the same as image Example 4diac prep & update. But here the update is

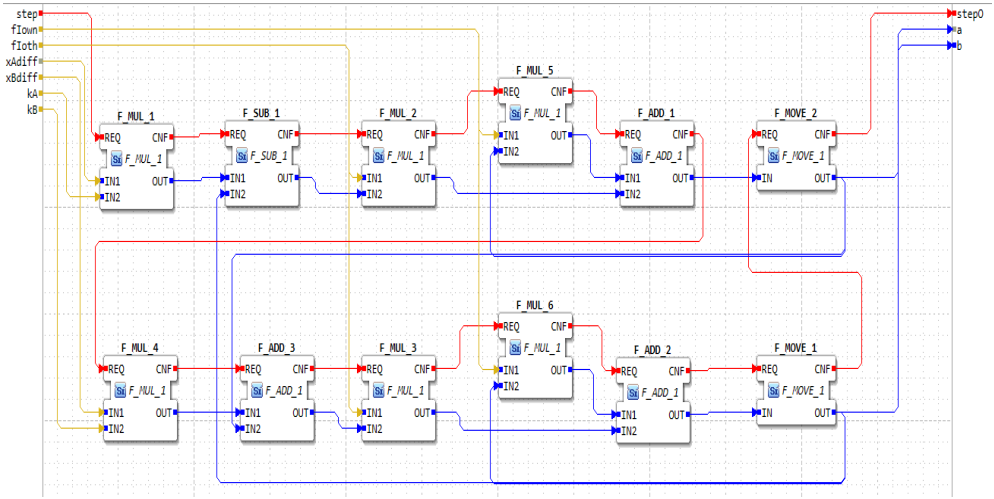


Figure 16: OrthBandpass without update event

This image shows the bandpass filter algorithm in 4diac similar as in Filter algorithm in C integrates dependent two values. The current previous values for integrate are used from the `F_MOVE_1` FBlock right side, but after calculate the filter the bothe `F_MOVE_1` FBlocks are also updated immediately in the same event chain. This works exact for the filter algorithm for one filter, but it gives slightly wrong results if more than one filter is used, for example for harmonics. Look for this usage of the image Example binary logic prep & update:

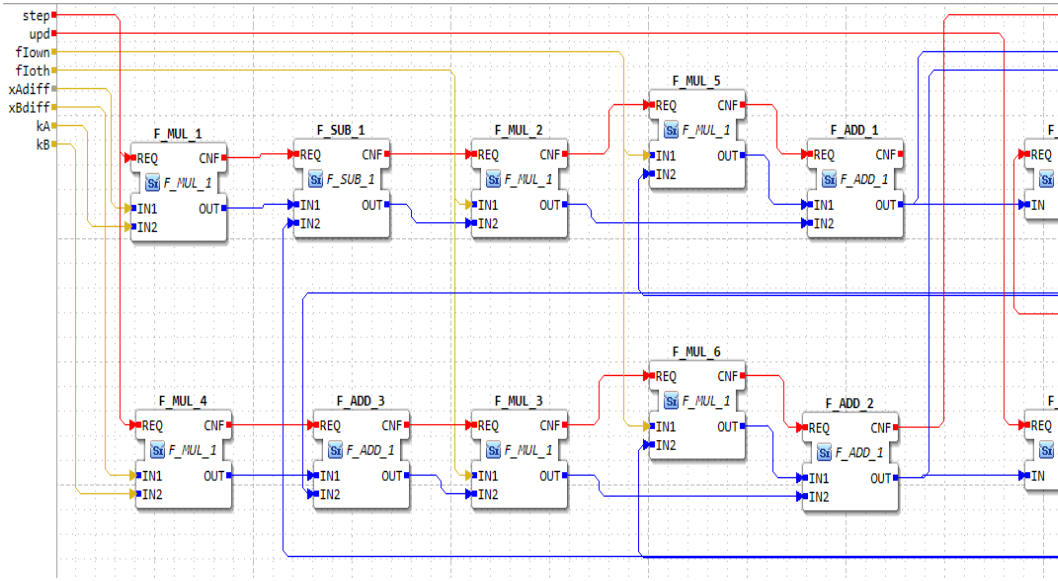


Figure 10. OrthBandpass with update event

There are two differences, first is the `upd` and `updO` event for update, but also a `ya` and `yb` is given which presents the calculated new outputs. This may be important because if the outputs are used as process outputs, they become active in the next step time because of course, it should be first give to the output device. If only the `yaz` and `ybz` are given, then they are the old values, one time back, which causes an additional dead time for control.

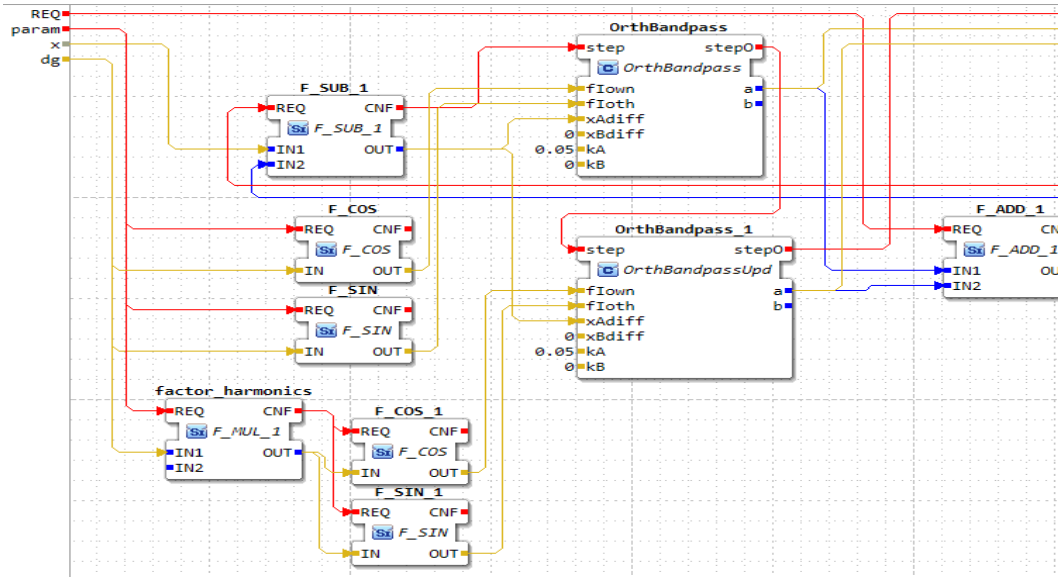


Figure 11. OrthBandpass in a filter application

The image above just shows an application where two OrthBandpass without update event are used, one for the fundamental oscillation, and one for an harmonic. Both values are output, `yfilt` is the filtered output of `x` and `y2harm` is the detected harmonic. That is the mission and possibility of this filter stuff. Also more as one harmonic is possible to filter. The principle is, all detected waves are added and compared with the input. The difference input for all ***OrthBandpass*** is equal, but each ***OrthBandpass*** has the resonance for its own frequency. If all frequencies are summarized and this is sufficient then the difference is 0 and the signals are stable.

But back to the event topics. The events are connected in that kind, that the resulting signals from the filter are presented in the outputs. The `F_ADD_1` is calculated firstly, takes the ***old*** current values from the step time before, put it in the feedback, and last the both ***OrthBandpass*** FBlocks are calculated. This is tricky. But what about if for more harmonic parts or other evaluations outside of this module the ***old*** current values are necessary. Then the logic becomes more complex.

Using the ***prepare and update*** concept is more obviously.

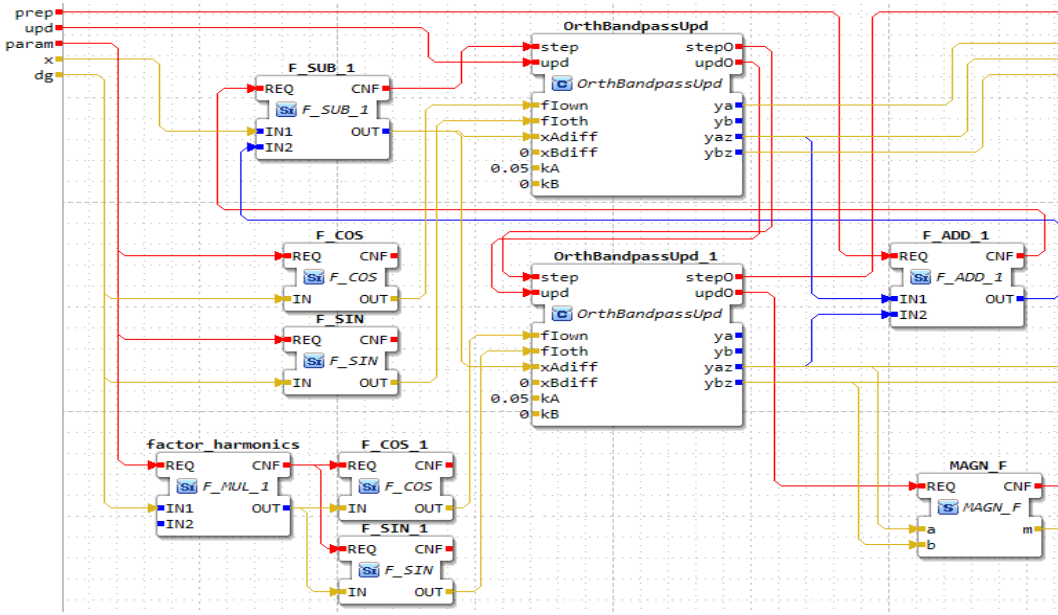


Figure 12. OrthBandpass in a filter application

Using the base variant of the filter with update, now also an filter application is possible and simple understandable, which outputs the filtered signal as new one for output on physic, and delivers also signals for further evaluation, here both components of fundamental and harmonic oscillation and the magnitude of the harmonics. The last one is calculated in the `upd` event chain.

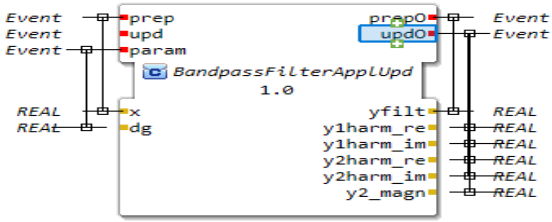


Figure 13. OrthBandpass in a filter application

The interface shows the assignment of `yfilt` to the `prepO` output event, and the other signals to the `updO` event. The `prep` event queue is for ordinary evaluation of calculations, the end signal may be output to hardware or transmit, and the `upd` event queue delivers ***signals as state of another event updO*** to use it in the `prep` calculation (in the comprehensive superior module). But of course both event chains are related, not formally, but semantically. The event source should organize the proper order of `prep` and `update`.

7.4.5. Example prepare and update in Simulink

in Simulink (© Mathworks) also an ***prepare - update*** concept is used. Simulink knows S-Functions, so named System-Functions which are not programmed graphically, instead textual. This S-Functions can be written in C language. The S-Functions can be used to understand the calculation principles of Simulink, it is obviously. The Standard FBlocks should have (expectable) the same principles. See especially the ***unit delay*** in this chapter below.

In the SFunction implementation two different operations should be called: `mdlUpdate(…)` and `mdlOutputs(…)`. The original text from the Mathworks help is

https://www.mathworks.com/help/simulink/sfg/mdloutputs.html: *The Simulink® engine invokes this required method at each simulation time step. The method should compute the S-function's outputs at the current time step and store the results in the S-function's output signal arrays.*

https://www.mathworks.com/help/simulink/sfg/mdlupdate.html: *The Simulink® engine invokes this optional method at each major simulation time step. The method should compute the S-function's states at the current time step and store the states in the S-function's state vector.*

The `mdlOutputs(…)` operation can process inputs of the FBlock, and sets of course the outputs of the FBlock. If the FBlock is only combinatoric (an expression), then this is the only need operation, `mdlUpdate(…)` has no sense.

If the FBlock has states, then the output can be calculated from states and inputs. These input pins should be marked as `ssSetInputPortDirectFeedThrough(…)`. Then the engine of Simulink detects loops in the data flow with these pins which is shown normally as error. It means these input pins should be used only straight forward with the outputs for combinatoric. Note: A Moore automaton would not process inputs for the outputs, uses only the states. But this is not a Mealy-automaton, because due to figure data flow with qout the outputs are further used in prepare-calculation or are the inputs for the physical output. The view of Mealy and Moore is inappropriate here. It is in mid of the transition or just prepare logic.

The `mdlUpdate(…)` operation can have inputs of the FBlock to calculate the new state from input and the state before, or it can also used internal variables calculate on the `mdlOutputs(…)` to set the state. It does not change outputs of the FBlock.

In the graphical Simulink model first all `mdlOutputs(…)` operations of all FBlocks are called. It means the current states (of the step time before) are presented on the outputs and the data flow for combinatorics are calculated, offer to inputs for further processing.

If all `mdlOutputs(…)` are called and the combinatoric data flow is done, then all `mdlUpdate(…)` are called. They may use values on inputs, but do not change outputs, and calculate the internal state for the next step time.

It means the `mdlOutputs(…)` with the combinatoric calculation is exact the ***prepare*** phase, and the `mdlUpdate(…)` is the ***update***. For ***update*** a few combinatorics inside the FBlock can be also calculated. That makes it a little bit more powerful for some special desires, but

also more complicated. The state can also be set only from internal variables calculated on `mdlOutputs(…)` due to the image data flow with qout.

Because the programming of user - S-Functions in C/++ language can be done in any kind in responsibility to the user, it is also possible to omit the `mdlUpdate(…)`, do all in `mdlOutputs(…)` and consider the order of statements. The result of one FBlock can then be exactly, but the mix of ***prepare*** and ***update*** both done in one operation `mdlOutputs(…)` can cause small mathematically errors in differential equation solving over more FBlocks. Note that the order of calculation is other, `mdlUpdate(…)` of **all** FBlocks is called **after all** `mdlOutputs(…)` are processed.

The *unit delay* FBlock

Now look on the working example for the bandpass filter above with pure Simulink graphic.

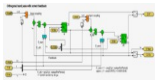


Figure 14. Bandpass filter base FBlock in Simulink

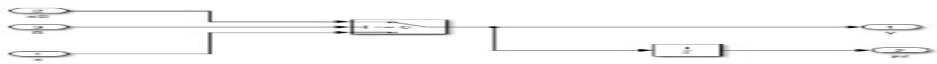


Figure 15. setable unit delay in Simulink

The FBlocks A and B are a simple store FBlocks able to set as shown right. The important one FBlock here inside is the ***unit delay*** marked with `1/z`. It stores the value on input as current value for the next step. It means the first called `mdlOutputs(…)` outputs the current value, also for the own integrate, and also for use for further calculations with the current state (set from the previous step time). The later called `mdlUpdate(…)` then stores the input inside, to output it in the next step time.

If you look now to the whole module Bandpass filter base FBlock in Simulink then you see the `Yz` outputs of the both storage FBlocks as `Yz` or `Yaz` for this module. This is the current state from the previous step time whereas `Y` is the new state also usable for example for immediately output, which becomes currently in the next step because of physical device properties. But also for example the difference between `Yz` and `Y` can be built to get the growth (differential) of the outputs.

If you look on a usage of this module, you see that the `Yz` is used for a feedback to compare the input value with the current state, not the `Y`. Because both FBlocks have the ***unit delay*** inside with exact usage of `mdlOutputs(…)` and `mdlUpdate(…)` the solution is correct. This is a bandpass filter with high resolution, so small errors are seen in a bigger abbreviation of phases or resonance frequencies.

7.4.6. Example prepare and update for odg Graphic code generation (Libre Office)

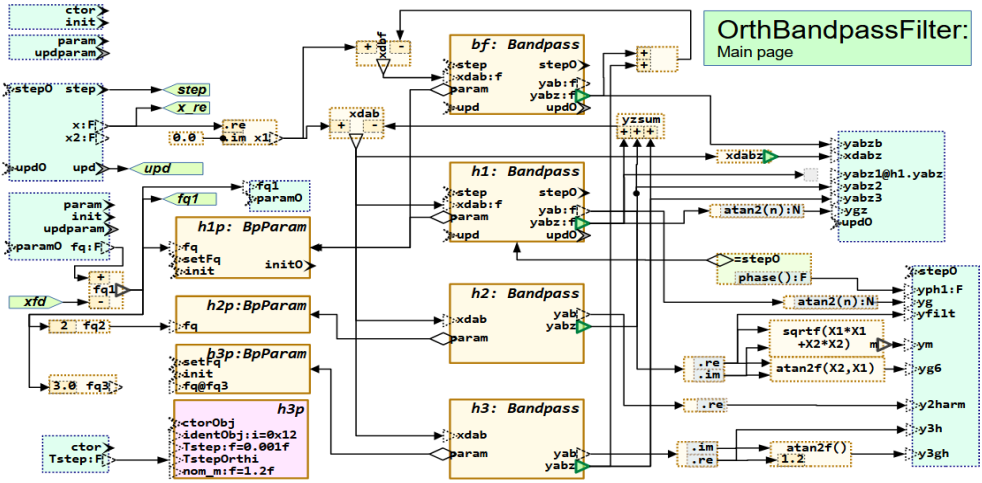


Figure38: OrthBandpassFilter.odg.png

The image above shows the application of a bandpass filter, the same as shown also in C, 4diac and Simulink, drawn in LibreOffice graphic. This is the approach of ../pdf/UML-FBCL-Diagrams-Libreoffice-2023-09-23.pdf. From this graphic both a IEC61499 module should be generated as well as also execution code in C (this is in progress, not ready yet). The event connections are all gray, because they don't need to be drawn, they are established by the data flow exploration. Only the data flow connections should be drawn. But the event pins and the event to data associations should be known. For that the green dashed blocks shows input and outputs of the module, whereby always one prepare event pin is contained in the module's pin block, and also the associated update event pin and the associated output pins. With this information and with the adequate information in the used FBlocks the event connections can be determined.

The image contains also an ***aggregation*** `param`, to a `BpParam` FBlock which is filled with the `param` event.

The used modules are given as C language routines with a wrapper in IEC61499 as textual.fbd The wrapper for the `OrthBandpassF_Ctrl_emC` is given as following (manually written following the C operations):

Wrapper for OrthBandpassF_Ctrl_emC in IEC61499 to adapt to C

```
FUNCTION_BLOCK OrthBandpassF_Ctrl_emC
EVENT_INPUT
  ctor WITH OTHIS, Tstep;
  init WITH param;
  step WITH xab;
  upd WITH step;  (* Note: Association of upd to the step dataflow *)
END_EVENT
EVENT_OUTPUT
  initO WITH initOk;
```

```
  stepO WITH yab;
  updO WITH upd, yabz;  (*Note: Assoc upd input event)
END_EVENT
VAR_INPUT
  OTHIS: OrthBandpassF_Ctrl_emC__REF;
  xab : CREAL;     (* Difference to adjust *)
  param: Param_OrthBandpassF_Ctrl_emC__REF; (* reference to parameter *)
  Tstep: REAL;   (* Step time for calculations *)
END_VAR
VAR_OUTPUT
  yab: CREAL;     (* new calculated value *)
  yabz : CREAL;   (* state value from last update *)
  initOk: BOOLEAN;
END_VAR
```

Wrapper for OrthBandpassF_Ctrl_emC in IEC61499 to adapt to C

```
VAR
  THIS: OrthBandpassF_Ctrl_emC__REF;
END_VAR
EC_STATES
  IDLE;        (* EC idle state *)
  CTOR: CTOR;  (* Constructor *)
  INIT:INIT -> initO; (* EC State with Algorithm and EC Action *)
  STEP: STEP -> stepO, ->step2;
  UPD: UPDATE -> updO;
END_STATES
EC_TRANSITIONS
  IDLE TO CTOR:= ctor; (* constructor call *)
  IDLE TO INIT:= init; (* An EC Transition with event*)
  IDLE TO STEP:= step;
  IDLE TO UPD:= upd;
  CTOR TO IDLE:= 1;
  INIT TO IDLE:= 1;
  STEP TO IDLE:= 1;
  UPD TO IDLE:= 1;
END_TRANSITIONS
ALGORITHM CTOR IN ST:
  THIS := ctor_OrthBandpassF_Ctrl_emC(othiz:=OTHIS, Tstep:=Tstep);
END_ALGORITHM
ALGORITHM INIT IN ST:
  initOk := init_OrthBandpassF_Ctrl_emC(thiz:=THIS, param:=param);
END_ALGORITHM
ALGORITHM STEP IN ST:
  step_OrthBandpassF_Ctrl_emC(thiz:=THIS, xAdiff:=xab.real, xBdiff:=xab.imag);
  yab := THIS.yab;
END_ALGORITHM
ALGORITHM UPDATE IN ST:
  upd_OrthBandpassF_Ctrl_emC(thiz:=THIS);
  yabz := THIS.yabz;
END_ALGORITHM

END_FUNCTION_BLOCK
```

In words of Simulink, this is a S-Function.

In the graphic you see outputs green with dark borders for `yabz`. This outputs have a graphic style of `ofpZoutRight`. This identifies it as an output of a value from the last steptime as current state, similar as a ***unit delay*** in Simulink or as an output without `ssSetInputPortDirectFeedThrough(…)` for a Simulink S-Function. This output is related to the `upd` event in the FBlock.

For the data flow it means that this outputs are given, can be used without preparation.

The data flow goes forward to the adder, then to the subtraction, and to the inputs of the ***Bandpass*** modules. Also the input of the module is processed. Due to this data flow the `prep` event is calculated starting from the module's input, first through the adder, then to the ***Bandpass*** FBlocks, whereby all three can be calculated parallel. Any Bandpass `yab` output is then taken through the ***complex to real*** access and put to the step output, related to the output `stepO` event. That is the preparation.

The ***update*** of the ***Bandpass*** FBlocks is necessary because they have an update event input `upd` which is related to the `step` event input. Hence they need connected to that event from the module, which is related to the same prepare event. This is the `step` event chain, and the `upd` of the module is associated.

The outputs `yabz1` and `yabz2` of the modules are designated again with the graphic style `ofpZoutLeft`, but it needs to be related to an update event which renews the value. This is explored due to the event-data relation `yabz` to `updO` in the `OrthBandpassF_Ctrl_emC` module and the data flow.

7.5. How to associate the prepare to the update event

`prepare` (in the example `step`) und `update` are related. If the events are given manually in the graphic, then it is not a quest. But in the graphic above Wrapper for OrthBandpassF_Ctrl_emC in IEC61499 to adapt to C only the data flow is given. The event flow, here drawn in gray, can be missed, should be supplement automatically. This is as usual for FBlock diagrams, where often only the data flow is drawn.

To determine the correct event connections as shown here in gray, the data should determine which update event is associated to a step event. Also it should be known from all used FBlock types, which data in- and outputs are related to the events. In the image and in this way in LibreOffice FBlock diagrams the relation between prepare and update event is given in the input box (style `ofbMdlPins`). Such an module pin box contains exact one prepare event, the associated update event, associated prepare and update output events (left side) and the data associated to the prepare event. The module pin box right side with `yCtrl` associates this pin with the `updO` event.

The FBlock PID itself is given as ready to used SFunction in C language with all these events regarded in implementation. The interface of this FBlock type is given as fbd file in the textual notation of IEC61499:

Step and update association in FBD

```
EVENT_INPUT
  ctor WITH OTHIS, Tstep;
  init WITH param;
  step WITH xab;
  upd WITH step;  (* Note: Association of upd to the step
dataflow *)
END_EVENT
EVENT_OUTPUT
  initO WITH initOk;
  stepO WITH yab;
```

```
  updO WITH upd, yabz;  (*Note: Assoc upd input event)
END_EVENT
VAR_INPUT
  .....
```

Here in line 5 the `upd` event is declared using another event `WITH step`. Normally for IEC61499 textual notation only a data association to events should be noted here. But the syntax is not changed by this approach, only the semantic. On evaluation of the source it is detect: `upd` is related `WITH step`, `step` is an event, and hence `upd` is an update event related to the `step`. This is the only one enhancement of IEC61499 textual notation, without syntax change.

With this information, and the information in the state machine (ECC) about associated output events to inputs (see link TODO) the necessary event connections can be determined. See chapter TODO other html document to write

8 Inner Functionality of the Converter Software

This main chaper

This first level chapter should show the inner functionality of the converting software to read Open/LibreOffice diagrams, translate to and read IEC61499, generate source code, and also organizes co-working with Simulink and Modelica.

It may be not only for deep experts; Also if you see this inner stuff you can better understand the concepts.

Generally all this converter software is written in Java. Only a standard Java is used (based on Java-8), without additional libraries, but the own vishia basic library vishiaBase.jar is elaborately used. This library contains also all basic functinality for example to read XML.

You find some information about the vishiaBase.jar also in https://vishia.org/Java/index.html

The sources for the vishiaBase.jar and for the vishiaUFBgl.jar are able to download as zip beside the jar files itself (https://vishia.org/Java/deploy/, the version archives are hosted on https://github.com/JzHartmut

You can translated and executed the sources for example in an Eclipse environment, in debug mode.

This first level chapter should contain enough hints to navigate in this sources. Some javadoc links are contained here. Also the sources with its generated javadoc contains explanations of the classes and operations.

8.1 Data Model data classes

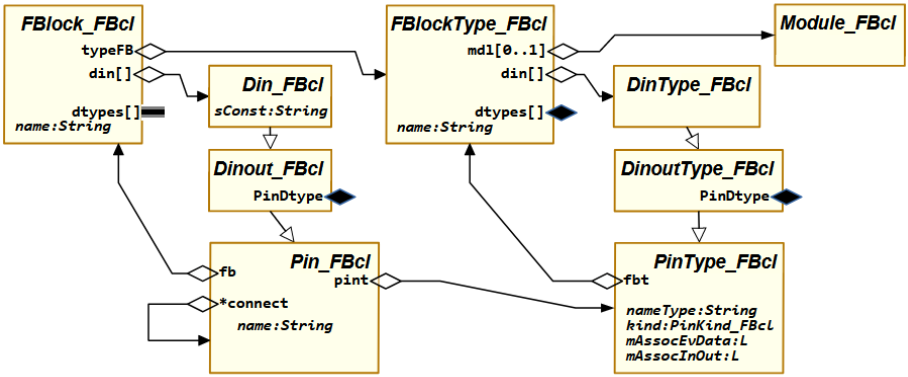


Figure 17: FBlock_FBtype_Pin.png

The diagram Figure 17 shows the relation between Instances of FBlocks and its Types and Pins. The FBlocks are the base elements of the Function Block Diagrams. and also for UML class diagrams. A class is presented also with a FBlock_FBcl (www) because of its interconnections. To distinguish between pure classes and instances of classes in the diagram the FBlock presentation of classes has an internal name starting with "$".

This diagram Figure 17 shows the pins of Blocks and their types only on the example of data input pins. The other pin kinds are similar.

8.1.1 FBtype_FBcl

FBtype_FBcl (www) presents a FBlock Type. There are some standard types such as for expressions, event join or rendezvous of events (E_REND in IEC61499) and variable storage (F_MOVE in IEC61499) and for compatibility all known standardized FBlocks of the IEC61131 norm (automation control, PLC = Programmable Logic Control).

All functionality which is immediately given in C/++ language for embedded control can be wrapped with a FBlock_Type_FBcl to embed it in a graphic. The FBlock type definition can be written manually in textual form using the IEC61499 coding (fbd file), or also designed in a LibreOffice graphic (odg, FBUMLgl).

Specific FBlock_Type_FBcl on user level can be defined graphically with FBUMLgl. It can be stored as fbd file due to the IEC61499 standard.

The Java class FBlock_Type_FBcl presents the interface data of this FBlock types. The content (inner functionality) is either given immediately in C/++ or the appropriate destination language, or it is contained in a Module_FBcl, see Java class Module_FBcl (www).

The FBlock_Type_FBcl has 6 arrays which describes the Pins:

- **dinPin**: DinType_FBcl (www):
- **doutPin**: DoutType_FBcl (=>www)
- **evinPin**: EvinType_FBcl (www)
- **evoutPin**: EvoutType_FBcl www)
- **refPin**: PinTypeRef_FBcl (www)

- **portPin**: **PinTypeRef_FBcl**

The data and event pins are also defined in IEC61499. The **refPin** is an aggregation to another FBlock, as source pin (as in UML). The counterpart is the **portPin**, which is a destination pin. In Uml either it is a really port (any inner instance reference in a FBlock), or it is **THIS**, which presents the whole referenced FBlock.

For IEC61499 presentation (fbg, FBcl source file) the **refPin** **is mapped to** a **dinPin**, arranged after the other **dinPin** as input. On runtime the reference value will be set in the initialize phase with the init event. The data flow is reverse to the UML presentation as reference to the other instance or type. Adequate it is with the **portPin** **is mapped to** a **doutPin** because it delivers as output the reference. The type of this dinPin and doutPin are always designated in the IEC61499 files as ***name*__REF** whereby ***name*** is the name of the **FBtype_FBcl**.

8.1.2 FBlock_FBcl

FBlock_FBcl (www) presents an instance of a Function Block, It refers its FBtype_FBcl (www) and it has an instance **name**. The pins of a FBlock instance are then different from the type pins, if multiple pins are existing. Then the type has only one pin which name ends with “0999” or “1999”, and the instance pins counts from 0 or 1, for example X1..X3 for three inputs. Also not all type pins may be existing for the FBlock, if there are unsused.

The data types of a FBlock can differ from the data types in the type pins, it can be specialized.

8.1.3 Pin_FBcl and PinType_FBcl

The pins of a FBlock_FBcl (www) are based on Pin_FBcl (www) with the specifications:

- **din**: Din_FBcl (www):
- **dout**: Dout_FBcl (=>www)
- **evin**: Evin_FBcl (www)
- **evout**: Evout_FBcl www)
- **reference**: PinRef_FBcl (www)
- **port**: PinPort_FBcl (www)

The Pin_FBcl contains the connection to other pins to other FBlocks whereas the referenced PinType_FBcl (www) contains some common information, see next chapter.

8.1.4 PinType_FBcl

PinType_FBcl (www) contains Information to any pins. It is the base / super class for all pin types. It contains:

- `fbt`: The FBtype where the pin is member of.
- `namePin`: String: It is the pin name same as in the instance or ..1999 or ..0999 for a **multiple pin**.
- `ixPin`: int: The index in the array, and also the bit number in some mask bits.
- `kind`: PinKind_FBcl (www): an enum describes the function.
- `mAssocEvData: long`: up to 64 event or data associations. This is in IEC61499 the designation

```
EVENT_INPUT
  step WITH x;
  ……
VAR_INPUT
  x : REAL;
```

  But also the back association, which data uses which event, is stored here. `evinPin` is associated to `dinPin` and vice versa, and `doutPin` to `evoutPin`.

- `mAssocInOut: long`: up to 64 input and output associations. This is not immediately shown in IEC61499 but can be determined. See also 7.2. FBtype kinds and their usage (due to IEC61499) . For ***Standard FBlocks*** the output event depends on the state machine. Any output event which may be occure on an input event because of a state entry is contained in the mask for the input event. For the Standard FBlocks with a simple regular state machine the input and the output events are well associated, it is simple. Due to the event association also the data association are marked.

  For a Composite FBlock consisting of an usual graphical interconnection of FBlocks the input – output -association are an result of the connections.

  Note that detail informations about event and data input output mapping are contained in the EccAction_FBcl to the states. This informations are used for evaluation of the inner content of a module.

- The DinoutType_FBcl (www) contains also a data type information for the `dinPin` and `doutPin` as well as also for `refPin` and `portPin`:, see Data Types
- The EvinoutType_FBcl (www) contains te association between prepare and update event as number assocEvPrepUpd related to the ixPin, see 7.4. Prepare and update actions
- The EvinoutType_FBcl (www) contains also references to EccAction_FBcl (www) for immediately execution of actions to events, see also
- The PinTypeRef_FBcl (www) refers with `FBtype_FBcl fbRef` the type of the reference.

Operations or Actions assigned to the Pins, code generation

The `EvinType_FBcl` has usual an assigned `Ecc_Action_FBcl`. On inner Pins of a module the input event is related to a pin of type Evout_FBcl, and also a data inputs are offered with a Dout_FBcl, an output to the inner FBlocks of the module, the actions are assigned to the common class PinType_FBcl. TODO it's better it is dedicated.

The `DoutType_FBcl` has an assigned `Ecc_Action_FBcl` if the inner logic of a `FBtype_FBcl` comes from a ***Composite FBlock*** (a FBlock with graphical content). Then this action describes the access operation to this output pin or also to more as one related output pins, depending on code generation rules.

For ***Standard FBlocks*** the outputs are immediately the output variables which are set by the actions on the `EvinType_FBcl`, or depending on the code generation, they are simple access operations (“*getter*”).

Simple FBlocks has only one Action which may be stateless. If it is stateless then it is an expression. For that the `EvoutType_FBcl` has assigned an `Ecc_Action_FBcl` which calculates the expression tracked backward. The action or just ***operation*** of a stateless Simple FBlock with one output can be written in an expression line.

If a *Simple FBlock* (also an expression) has more as one output, the outputs are presented by inner variables. It means the calculation of such an expression is broken.

Association between Event and Data Pins

The Pins in FBlock_Type_FBcl are contained in adequate arrays. The position in the arrays are used for bit masks mAssociatedInOut and
Java class: org.vishia.fbcl.fblock.PinType_FBcl.html#mAssociatedEvData =>www.

Associaton between Input and Output pins

This should be contained in EccAction_FBcl

Association between prepare and update events.

The element Java class: org.vishia.fbcl.fblock.EvinoutType_FBcl.html#assocEvPrepUpd =>www.
contains the index of the prepare event in a given update event.

Multiple pins

A multiple pin is pin definition in a PinType_FBcl which can be represented by more as one pin on the FBlock_FBcl instance. This is typically used for expressions, adders or such. In IEC61131 and also IEC61499 this is not intended because the implementation languages cannot deal with it. But this idea is similar “variable number of arguments” in programming languages such as C or Java.

For input via FBUMLgl it is desired and for code generation from FBcl this is not a problem. There is a tricky possibility to store a pin in the FBtype_FBcl which presents multiple inputs:

The name of the pin should end with ...0999 or ...1999, for example “X1999”.

The “999” suggests “many”. The number should not be necessary as normal pin Name.

If the FBtype_FBcl has such a pin, any pin number from ...0 or just ...1 is available and refers the same pin “...0999” in the type. The pins has all the same properties, but of course different data connections, or different constants, or also different data types just as pins of

instances have in comparison to the type pins. The code generation can deal with this situation.

If such an design should be implemented in original IEC61499 environment (for example fortis), a proper type should be present. Or just, fortis can also be enhanced to deal with this situation.

Data Types

8.2 Module with FBlocks

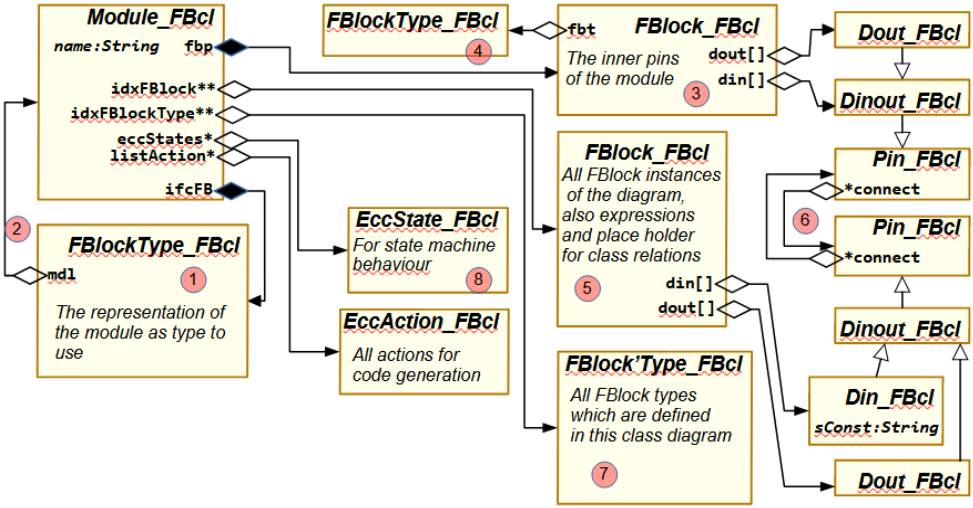


Figure 18: Module_FBcl

Any Graphic with Libre Office builds a Module_FBcl (www). The module can be presented any time as ***Composite FBlock type*** in IEC61499.

The image shows the important ones:

(1) The representation of the module to outside with the FBtype_FBcl (www) is referenced as **ifcFB** (interface FBlock), and is referenced as **mdl** from there **(2)**. This back reference can be removed if the module is code generated and the inner data are no more necessary. The interface **FBtype_FBcl** remains then as library module.

(3) The pins of the module to outer counterpart to the pins in the **ifcFB**) are contained in the referenced FBlock via **fbp** (FBlock for pins). Whereby the input pins are here output pins to the inner wiring inside the module and vice versa. The aggregated FBtype_Fbcl **(4)** is only internally necessary, it is also mirrored in respect to the pin direction to **(1)**.

(5) The module consists of many FBlocks, which are referenced all sorted by name via **idxFBlock**. Also expressions are FBlocks

(6) Right side it is shown that these FBlocks are wired together with its pins, and also wired to the module's I/O-pins.

(7) Only that **FBtype_FBcl** are indexed via **idxBlockType** which are defined in this module. Used **FBtype_FBcl** from FBlocks as given are not contained in this index.

(8) Also States and actions are referenced, see chapter TODO

8.3 Write instances for FBlock_FBcl, FBtype_Fbcl, Module_FBcl

TODO

8.4 DType_FBcl and DTypeBase_FBcl

8.4.1 Using DType_FBcl

Instances of DType_FBcl (www) are referenced from data pins, see chapter *8.1.3 Pin_FBcl and PinType_FBcl*. They contains

- **dt**: The reference to the basic data type: DTypeBase_FBcl (www)
- **sizeArray**:
  - 0 for scalar,
  - 1.. for a one dimensional array.
  - -1 **arrayUndef** not yet defined
  - -2 **arrayFree** Array with a variable size but given on runtime
  - -3 **arrayList** A container as list
  - -4 **arrayKeyList** A container as sorted list.

The same instance of **DType_FBcl** is often used by several pins of the same **FBtype_FBcl** or **FBlock_FBcl** and also shared between some or many pins inside a module, whenever the same data type is used. Generally connected pins refer the same instance of **DType_FBcl** on both ends. For a Module_FBcl (www) and also inside FBlock_FBcl (www) and FBtype_FBcl (www) there is a container **dtypes**, which refers all non full specified **DType_FBcl** instances used in the pins of the FBlocks. Changing this only few instances of **Dtype_Fbcl** can manipulate all data types using it. For example a module can code generated as scalar functionality or alternatively as vector, or for float arithmetic, and alternatively for double or integer.

There are a few *"fixed"* **Dtype_Fbcl** instances. That are these which refers the basic types without array or container designations. Often this instances are used, and then it is the same in the pins of **FBtype_FBcl** and **Fblock_Fbcl**.

Instances of **DType_FBcl** which are not full dedicated in a used **FBtype_FBcl** are never copied to the **FBlock_FBcl**, because they should be adapted (changed). That is especially if the **DTypeBase_FBcl** is a non full specified data type such as "**ANY_NUM**" instead **float**, **int** etc. That is typical for some expressions or mathematically operations. This is done first by creating a clone of the **DType_FBcl** instance for the pins of a **FBlock_FBcl** from the pins of the **FBtype_FBcl**. The clone is necessary because afterwards the **DType_FBcl** can be changed, independent of the **DType_FBcl** instances in the **FBtype_FBcl**. This changes are done to get more deterministic types. Either the **dt** reference in a **DType_FBcl** can be changed, or by replacing the instance of **DType_FBcl** in all appropriate pins.

But while forward and backward propagation the number of different instances of **DType_FBcl** is reduced.

For the last action all **DType_FBcl** instances contains a reference:

- **usingPins**: Reference to all pins using this type. It is **null** (not existing) if all **DType_FBcl** refers a deterministic type. This reference is used to change a changed **DType_FBcl** on one pin in all other appropriate pins.
- **deps**: This container references all **DType_FBcl** which are not the same but depending in some characteristic. If for example one **DType_FBcl** is complex, another is real, or one is scalar and the other is an array, but both should have the same numeric type. then changing the type in the one **DType_FBcl** should be done also in all depending **DType_FBcl** instances.

8.4.2 Using DTypeBase_FBcl

All types in DTypeBase_FBcl (www) are designated by a `public final char typeChar`. One char is enough and concisely

The basic types without container and array specifications are either standard types, contained in `DTypeBase_Fbcl.stdTypes`.

Or they are the reference type to used `FBtype_FBcl`. In IEC61499 these are “ANY_DERIVED” types, applied to “TYPE END” language constructs. In the UFBgl these should be able to map to specific `FBtype_FBcl`. The `DTypeBase_FBcl` contains a field `typeRef` for this reference. The `DTypeBase_FBcl` instance for a specific `FBtype_FBcl`.reference is always created for the `FBtype_FBcl` itself referenced their with `dtypeTHIS`.

8.5 Read data from LibreOffice odg files

8.5.1 The file format of odg – content.xml

Let's have first a look to the file format from Libre Office. The odg format is a zip archive. You can add the extension zip, and then look into with a zip utility.

Right side you see a screen shot from the opened zip file (with Total Commander). The zip file contains three important xml files.

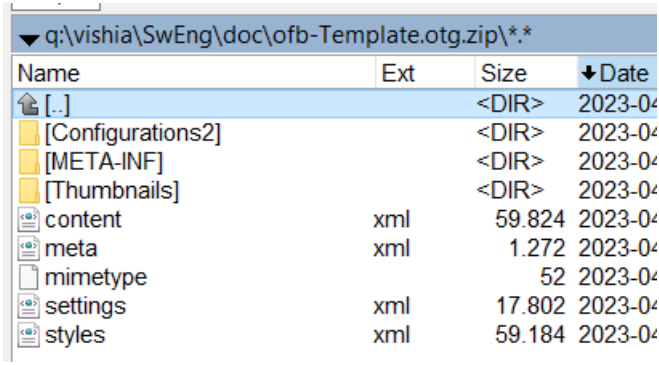


Figure 19: ContentOfodg.zip.png

- content.xml contains the graphic itself
- styles.xml contains the style sheet settings. If you want to copy your settings between some files, you can copy this styles.xml inside the two zip file. It seems to be safe.
- settings.xml is not relevant for the content itself, also the other files are helper for the Office tool.

Now have a look inside the content.xml (pressing F3 in Total Commander to view to pure textual content:

It is one very long line without structure not well human readable, but it is well formed XML.

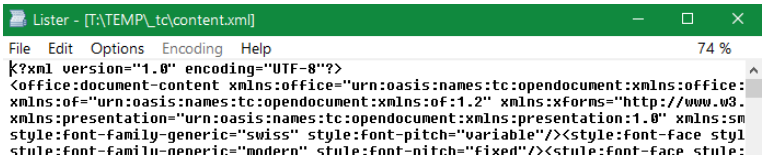


Figure 20: ContentOfodg-content-xmlPure.png

After beautification it looks like

```
<draw:g>
  <draw:custom-shape draw:style-name="gr21" draw:text-style-name="P1" draw:layer="layout"
    <draw:enhanced-geometry svg:viewBox="0 0 21600 21600" draw:type="rectangle" draw:enhan
    </draw:custom-shape>
  <draw:custom-shape draw:style-name="gr22" draw:text-style-name="P2" draw:layer="layout"
    <text:p text:style-name="P2">ClassA name1</text:p>
    <draw:enhanced-geometry svg:viewBox="0 0 21600 21600" draw:type="rectangle" draw:enhan
    </draw:custom-shape>
  <draw:custom-shape draw:style-name="gr23" draw:text-style-name="P7" xml:id="id18" draw:i
    <text:p text:style-name="P7">aggrCX</text:p>
    <draw:enhanced-geometry svg:viewBox="0 0 21600 21600" draw:glue-points="10800 0 0 1080
  </draw:custom-shape>
</draw:g>
```

This is right side truncated, it shows the graphical "group" with the "**ClassA name1**" as shown in Figure 18: Module_FBcl page 96. You can see here also the aggregation **aggrCX**. The style names are not written immediately plain here, instead a referencing is done, the **draw:style-name="gr23"** describes some possible direct formatting properties and the references to the knwon style "**ofpAggrRight**" as you see in the content.xml in the **<style...>** part.

```
<style:style style:name="gr23" style:family="graphic" style:parent-style-name="ofpAggrRight">
  <style:graphic-properties draw:marker-start-width="0.24cm" draw:marker-end-width="0.24cm" f
  <style:paragraph-properties style:writing-mode="lr-tb"/>
  </style:style>
```

This is all understandable and comprehensible. Hence read out of data is only a problem of sorting.

8.5.2 Read content.xml to internal data

The class readOdg\xml\XmlForOdg (www) presents the access to the read XML data. This class was automatically created by calling the tool suite on vishia.org/Java/html/RWTrans/XmlJzReader.html (www) but adapted afterwards. The base class which should not be adapted is readOdg\xml\XmlForOdg_Base (www), this class contains the data read from XML. The data structure in this class follows the structure of the src/.../odgxmlcfg.xml (www) which controls interpreting of the XML data. The class to read the XML file is vishia.org/Java/docuSrcJava_vishiaBase/org/vishia/xmlReader/XmlJzReader.html (www). It is called in readOdg\xml\XmlForOdg_Base (www).

The following code snippet shows how the **XmlJzReader** is invoked:

```
/**Reads completely the content.xml from the
 * and stores the data in the returned ins...
 * @param fInOdg The file to read
 * @return the read data from XML
 * @throws IOException On file read problems
 */
private XmlForOdg readXml (File fInOdg) th...
  String sFileOdg = fInOdg.getName();
  XmlJzReader xmlReader = new XmlJzReader();
  xmlReader.setNamespaceEntry("xml", "XML");
  xmlReader.readCfgFromJar(XmlForOdg.class,
                           "odgxmlcfg.xml");
  XmlForOdg_Zbnf data = new XmlForOdg_Zbnf();
  xmlReader.setDebugStopTag("text:span");
  xmlReader.openXmlTestOut( new File(this....
  xmlReader.readZipXml(fInOdg, "content.xml",
  return data.dataXmlForOdg;
}
```

The following text is a data snippet, gotten from the Variable View in Eclipse. odg is the returned instance. The text after a name is the **toString()** output, which contains sometimes only TODO (not used till now) but you can for example see the content of

a draw_page, and hence the XML structure. - It is only an illustration.

```
xOdg: XmlForOdg @unknown:0 XmlForOdg 251 1
 idxStyle: Map<String,String> null null
 office_document_content: XmlForOdg$Office_doc
  office_automatic_styles: XmlForOdg$Office_au
  office_body: XmlForOdg$Office_body TODO toSt
   office_drawing: XmlForOdg$Office_drawing TO
    draw_page: List<Draw_page> [TODO toString,
     [0]: Object TODO toString XmlForOdg$Draw_
      draw_connector: List<Draw_connector> [TO
      draw_custom_shape: List<Draw_custom_shap
      draw_frame: List<Draw_frame> null null
      draw_g: List<Draw_g> [(9.5cm, 4.1cm) + 2
      draw_master_page_name: String Default St
      draw_name: String page1 String 287 16552
      draw_polygon: List<Draw_polygon> [4.2cm,
      draw_polyline: <unknown type> null null
      draw_style_name: String dp1 String 289 1
     [1]: Object TODO toString XmlForOdg$Draw_
     [2]: Object TODO toString XmlForOdg$Draw_
     [3]: Object TODO toString XmlForOdg$Draw_
 office_font_face_decls: XmlForOdg$Office_fon
 office_scripts: String null null  16552
 office_version: String null null  16552
 office_version: String 1.3 String 264 16552
```

As you see, the data structure follows the XML content. The data are mapped from XML to this internally Java data. The mapping depends from the content of the `odgxmlcfg.xml` file, which controls the `XmlJzReader`, but this `cfg.xml` is so completely as necessary.

8.5.3 Sorting data from XML mapping to UFBgl data

A box which presents a FBlock is a shape in a page in XML data. An FBlock in the module is an instance of `FBlock_FBcl` in the structure as shown in chapter 8.2 Module with FBlocks.

The approach is, reading all shapes and associating to the semantic units of the module due to their graphic style and also due to there relative position. A primary idea for associating pins to blocks was building a group in LibreOffice graphic. But grouping should be seen only as graphic possibility, not for semantic. The position, the pin is inside the shape which represents the block, is the intrinsically possibiity of association.

The graphic from `xOdg` is evaluated page by page.

As first step in the graphic a shape with the style `ofbTitle` is searched in the page. The textual content till `:` is the module name. If it starts with `#` the page is disabled (not to evaluate). The module name is searched in `idxOdgMdl`. It is found or new created and stored there as `OdgModule` instance. A module can be read from some pages in one file or also dispersed to several odg files.

Secondly all shapes are evaluated which are block shapes, means they build the frame for blocks. This is checked in `gatherFBlockShape(...shape...)`. This is done first outside and then inside of groups.

The graphic styles which build block shapes are `ofbFBlock`, `ofbClass`, `ofbMldPins` and `ofbExpression`.

8.5.5 Preparation of Expressions from odg

The internal Handling of expressions needs a little bit explanation. Refer to chapter 6.4 Expressions inside the data flow page 50 to see the capabilities of expressions.

`createExprPins(…)`:

In `createExprPin(…)` all pins from a OdgFBlockGraphicInstance (www?) are evaluated. This are the drawn pins in the graphic, type is OdgPinInstance (www?). The kind of the pin due to the graphic style is stored in …

createDoutExpr(…) handles all pins with style ofpDout, ofpVout, ofpZout, ofpExprOut. The first name of one of these pins but not a ofpExprOut determines the name of the FBexpr instance.

8.6 Read data from Simulink

8.7 Read data from IEC61499 text files (fbd)

8.8 Forward and backward declaration of data types

This is a topic of the data flow. The forward declaration is done by the operation WriteModule_FBcl#propgDtypes() (www)

8.8.1 Forward/backward propagation of dedicated pins

The data type propagation starts by adding all pins to an internal `List<Dout_FBcl> listDout` with dedicated DType_FBcl (www) on `dout` pins of all FBlocks and dedicated pins of the ***module's inputs*** which are formally an `dout` (to the inner of the module). From this pins the connection is traced to connection `Din_FBcl` pins to following FBlocks, which then have the same data type. This is set, or checked. Conflicting data types are reported.

Then, in the reached `FBlock_FBcl`, depending pins which are yet not full dedicated are set, see next sub chapter.

After this forward propagation adequate is done with an internal `List<Din_FBcl> listDin` with dedicated `DType_FBcl` on `din` pins of all FBlocks and dedicated pins of the module's outputs, but only with pins which are not reached by forward propagation with an own data flow connection. Pins which are already reached by forward propagation do not need to handle again. But this pins, not reached by own forward data flow may be (are) also specified by the forward propagated flow to other pins of the same FBlock, if they are depending. All input pins of a standard expression have the same data type, specific expressions have depending data types.

TODO a proper figure is necessary

reached pins The remaining `Din_FBcl` pins which are dedicated but not already forward propagated are then backward tracked to the connected `Dout_FBcl`.Hence this pins are now also dedicated. Also tracking through the FBlocks is done as described in *8.8.3 Forward declaration for depending pins of a FBtype* also for backward traciing.

As result of this forward and backward propagation the most of pins in FBocks in the module, especially in expressions, are set to its fix data types whenever it is possible. If different fix data types are clashing in connections or depending pins, this is report as an error of propagation. It should be fixed in the module.

As result of its propagation all pins with dedicated types are clarified.

8.8.2 Forward and backward propagation of non dedicated pins

If pins remains which are not full dedicated in its data type, then the module itself is not full qualified. Code generation from only the module alone is not possible. The module can be used inside another module, and then this superior module should determine the data types of all to generate code.

But to can do so, the same instances of not full qualified `DType_FBcl` is necessary on the inputs or outputs of a module (favored: inputs) which are also used in the inner of the module, or just depending `DType_FBcl` are necessary to build as described in the follwing chapter for this module.

To do so, the same algorithm of propagation is done with the non full qualified module input and module output pins. As result, concise but not full qualified `DType_FBcl` instances are built with its `Depenency`

8.8.3 Forward declaration for depending pins of a FBtype

If pins are not full qualified then some pins depends from another. If the data type of one pin is dedicated, also all or some other pins should be dedicated with the same data

type. A simple expression can only have the same dedicated data type on all its pins.

But specific mathematics expressions have depending dedications. Simple, look on the expression which combines real and imagine part to a complex value. It is drawn in graphic as Figure 21.

- The yellow part above shows the presentation of the expression instance itself. The expression is presented in the data model in Java by a `FBlock_FBcl` with its pins, which are here one `Dout_FBcl` (a variable in generated code) and the both `Din_FBcl` as inputs, derived to `DinExpr_FBcl`.

- The types and dependencies are contained in the type specification in the middle part. For both real inputs only one representing `DinoutType_FBcl` exists because the specific pin functionality is contained in `DinExpr_FBcl.sExprTerm`. But the data type is referred there as `dtype` aggregation to `DType_FBcl`

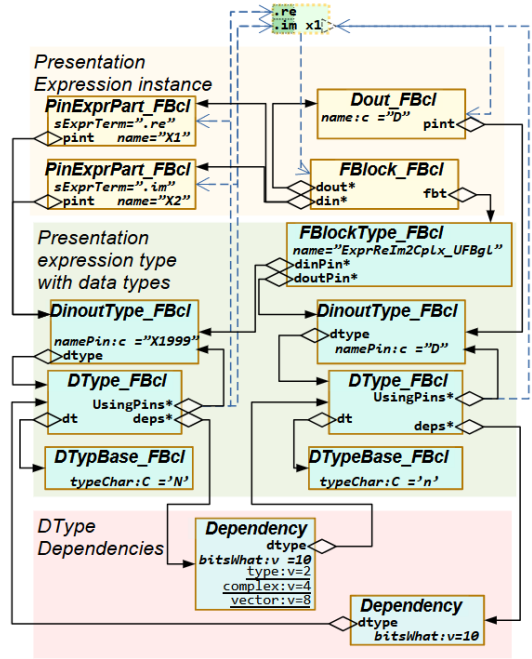


Figure 21: ExprReIm2Cplx_DTypeDeps.png

- The type of the inputs in the FBtype definition is "`ANY_NUMERIC`" with the short char `N`. The type of the outputs in the FBtype definiton is `ANY_CNUMERIC`. It is `n`, a complex numeric value. The dedicated numeric type is not defined by the expression type definition.

- The pin instances of the expression do not get a type per default, because the type is not dedicated.

- On forward propagation of the data types in the module, a DType `F` may be used on inputs, which is `float`. Firstly it is checked whether this type is complatible to the given DType `N` in the FBtype. It is compatible. A type `u` won't be compatible, because the `DWORD` is not `ANY_NUMERIC`, it's a bit type value `ANY_BIT`.

- If it is compatible, then the forward propagated `DType_FBcl` instance is used immediately for the `din` of the expression.

- But now, the **DType_FBcl** in the pin type of the expression type FBtype is tested, it has some references in **usingPins**, here only one. The reference goes to the FBtype pins. The **FBlock_FBcl.din** pins of the expression itself, and also all **FBlock_FBcl#dout** pins are checked whether they reference the **DinoutType_FBcl** instance referenced by **usingPins**. This is polling, an immediately aggregation or association does not exists, because from the FBtype no associations to an instance are possible. And the depending pins are only mananged by the FBtype.

This is also done if no extra Dependency exists as shown in Figure 21.

- But in this case also an aggregation **Dtype_FBcl#deps** exists with one member. The instance of DType_FBcl.Dependency (www) contains one reference **dtype** to the depending **DType_FBcl** which refers via **usingPins** the appropriate type pins. The instance pins, it is the output of the expression, is find again by polling, test which pin has the **DinoutType_FBcl** as **Pin_Fbcl#pint**

- The **Dependency** has a second value **bitsWhat**. This is a bit mask with a few bits with shown meaning. In this case the type and the vector should be the same in the two used **Dtype_FBcl** of the expression pins, but the complexity is different. The complexity (real or complex) is defined by the **Dtype_FBcl** of the FBtype here with *“complex”* with the lower case ‘**n**’.

- With this information the **DType_FBcl** data type for the instance pins can be select or created newly with the information from both **Dtype_FBcl**, the forward propagated one on the inputs and the given in **Dinout_FBcl** which determines *“complex”*. This is done with the operation DType_FBcl. getDependingType (Dependency) (www)

For this algorithm the distinction between DType_FBcl instances in the FBtype defionition and **DType_FBcl** instances used in the module for the **FBlock_FBcl** pins seems to be a little bit sophisticated. To prevent errors in the algorithm for some cases, the affiliation of a **DType_FBcl** instance to the instance or type usage of FBlock is given with the variable **Dtype_Fbcl#fixTypePin**. It is an enum with the value **eType** or **ePin** for affiliation either to the type or to the FBLock instance, and also the value **eFix**. The last value is one of the few instances which are standardized given for scalars.

8.9 Identification of the event flow due to data flow

In IEC61499 diagrams and language the ***event flow*** is an integral part of the model, planned by the architect of the solution. The *data flow* should match to the given event flow. Some special options are possible: Using data before they are newly calculated. It means that is a possibility, but also also a prone of error if mistakes are done.

In opposite, ordinary Function Block Diagrams uses only the ***data flow*** to calculate the processing order paired with dedicated sample time designation.

For the UFBgl diagrams, the internal processing uses the event flow as in IEC61499, but it is not necessary to dedicate it in all details from the graphic model. It is automatically generated due to the data flow.

8.9.1 UFBgl: Binding event to data on in/outputs

Other than for reading for example Simulink diagrams, the UFBgl need a dedicated association between data in- and outputs and the associated event pins. With the given event pins the data are related to the events, instead to *“sample times”*.

TODO adequate image as for simulink

8.9.2 Resulting evout because of evin of a FBlock

This is the question of track the event chain(s).

In chapter 7.2. FBtype kinds and their usage (due to IEC61499) page 72 Simple and Basic FBlocks are mentioned. Simple FBlocks have only one event input (evin) and one event output (evout) following the evin. Basic FBlocks can have more events. The special case of basic FBlocks with a simple regular state machine results in a non state-depending correlation between input and output events. This is regarded in building and executing the event chain. Such FBlocks are similar as classes (instances) of a class with more operations. The evin forces execution the operation, and on success the evout given with resulting data ready to get. But it is also similar to FBlocks in other Function Block Diagrams (such as Simulink) for each one sample time per event.

If a FBlock with a state machine is inside the module, it may build independent event outputs which builds an own event chain, as mentioned in the introduction to the chapter above.

8.9.3 Some Contemplation to bind data to events, event cluster

In Simulink events for that usage are unknown. Instead each data input should have a dedicated sample (step-) time association. The step time replaces the event association, if all functionality (all data pins of one step time) should be associated to one event flow. But this is also for optimization of code generation often not a good decision. It is better to have a fine division in primary independent function groups:

For UFBgl and IEC61499 you can have this fine division by manually planning of data and event associations, whereby you have more events as step times. Lets look on an example:

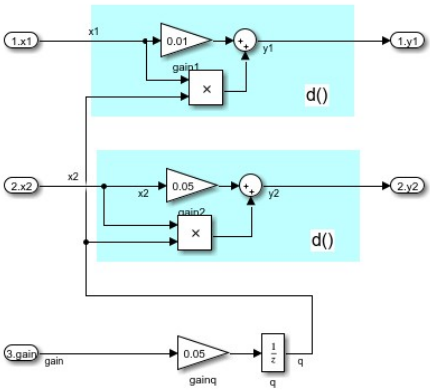


Figure 22: smlk/Testcg_MdlTstepSmlk.png

All data have the same sample time here. But maybe it is not necessary to calculate the outputs of y2. Then it is better to have two event chains, one for y1 and a second for y2. A third event chain is given, because the q variable is a *“unit delay”*, a stored value from the sample time before calculated with the third event.

The associations of the din and dout with same sample times to different events is done with first back tracking from the data, detection which input data are necessary for one or a group of output data. Doing that also branches are detected: Some data should be calculated before, as common data for then independent branches. For that look to a more sophisticated example:

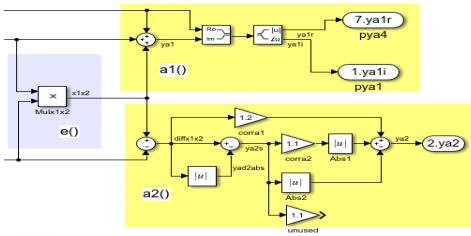


Figure 23: smlk/ParallelSimple_smlk_-EvChainBack.png

Both yellow blocks a1) and a2) are independent and hence controlled by different event chains with own event inputs for the module. But to execute this blocks, it is necessary to calculate block e) before. This is the first event to call.

TODO more simple smlk model

TODO Test with UFBgl, manual drawn evin and also a manual EvJoin FBlock.

8.9.4 Temporary info in pins for data→event processing

The *Dinout_FBcl* (www) contains two elements which are set temporary while built the event chain:

- `Evout_FBcl#idxRepresentingEvents`: This is a `HashMap` of `<Evout_Fbcl>` which contains all events, which drives this event. This is essential to detect the situation which is shown in Figure39: in the following chapter 8.9.5.5 Connect the events if all doutSrcOther are driven by events to prevent to much effort for unnecessary JOIN of events.

- `Evinout_FBcl#mEvinClusterEnd`: One bit for each evin in the module's inner evoutMdl array and in the array of inner evin for state machines, corresponding to the `PinType_Fbcl#ixPin`. Any event pin of FBlocks is marked to designate the association to an event cluster per end event. This is used for backward event to data flow algorithm (currently in version 2024-03 not used, but it was used in 2019, todo: do not remove the idea).

- `Evinout_FBcl#mEvoutClusterStart`: One bit for each evout in the module's inner evinMdl array and in the array of inner evout for state machines, corresponding to the `PinType_Fbcl#ixPin`. Any event pin of FBlocks is marked to designate the association to an event cluster per start event. This is used for forward event to data flow algorithm.

The Type Evout_FBcl (www) contains two elements which are set temporary while built the event chain:

- `Evinout_FBcl#idEvent`: This is a unique identification for each event for all modules while translating. It is used as key in Dataflow2Eventchain_FBrd#mapEvPrepUpdInQueue (www) which contains the unique instance of the triple of three representative events to process, see todo

8.9.5 UFBgl: Build the event chain

One event chain is the order of calculation starting with a dedicated, often module input event. Or adequate, if the event processing is organized with event queues on each or a group of FBlocks, it is the resulting order of execution the events for any FBlock. If the data flow is split with a variable of style `ofpVout...` (which results in an instance variable) then the event chain is also split into more than one event chains. More event chains are joined together with the specific `Join_UFBgl` FBtype if more as one event chain is necessary for data inputs.

it is Presumed that all input and output data of the module are assigned to events. The event connections in the module are not necessary and are just automatically propagated. But it is also possible to have some manual made event connections and also `Join_UFBgl` FBlocks for a more sophisticated event flow, or if the event flow should be explicitly presented in the graphic. The fine wiring of events can then be carried out automatically on the basis of the data flow.

8.9.5.1 Start on module's evin

This is organized by the operation Dataflow2Eventchain_FBrd#connectEventsForward() (www).

This operation puts firstly ***all input events of the module*** in a container (`LinkedList <EvPrepUpdInQueue> queueEvout`) to process it one after another. Whereby the update input events (see Error: Reference source not found7.4. Prepare and update actions) are combined with their prepare events due it is given in a UFBgl module input block (style `ofbMdlPins`). Both pins, prepare and

update, are associated in a class EvPrepUpdInQueue (www). This `queueEvout` is the filled by furthermore by more detected events in the chain. If the list is empty, all is done.

An adequate list `LinkedList <EvPrepUpdInQueue> queueEvUpd`) remains yet empty, it is filled on found update events for the update event chain.

The `doutSrc` pins of the module are marked with `doutSrc. bEvDataPropg = true` because they are driven by default by the module's event.

8.9.5.2 propagate one step forward

The operation `propgEvent(evoutSrc, …)` does the work for one event from the `queueEvout`. Each `evoutSrc` pin (first the evin of the module, it is a `Evout_Fbcl`) is tracked by tracking the associated `doutSrc` pins (firstly the module din pins, it is `Dout_Fbcl`) forward. This is a two-stage loop because there may be more as one `doutSrc` pins associated to one `evoutSrc`, and there may be more connections for each `doutSrc` to the `dinDst`.

It is asserted that `doutSrc.bEvDataPropg == true` because elsewhere the event should not be propagated. But the connected `dinDst` is tested `if(!dinDst.bEvDataPropg) {...`. If an `dinDst` is already marked, then it was already tracked and should not be handled again. On start it is not marked.

The log writes

```
- ^step: xa==>y0.X2
```

for tracking the `evoutSrc` `step` with the `doutSrc` `xa` and the `dinDst` `y0.x2`. For this input the associated `evinDst` input(s) of the FBlock are picked. More as one is possible but usual only one `evinDst` is existing.

8.9.5.3 Check all other dinDst

With the information about one data connection with the associated event the operation

checkDinOtherAndConnectEv(...) (www).

is called. This operation checks also all other `din` pins which are associated to this `evinDst`, because, the quest is not the data connection, it is the event connection. With that it is detected which `evoutSrc` are altogether necessary driving the `evinDst`. Often this is only the one given `evoutSrc` tracked with the `doutSrc`, but it is possible that other pins are driven by `doutSrc` with other events associated. With these all other `evoutSrc` respectively the whole information about its event chain the list `listEvoutSrc` is filled and offered to the `connectEvent…(…)` operation, see next chapter.

This operation `checkDinOtherAndCon…(…)` works in the following kind: While testing all `dinDstOther` to appropriate `doutSrcOther` the following cases are possible, the output to the log is shown in console font in `“ ”`:

- `constant`: `“#fb.din”` The `dinDstOther` is driven by a constant value, no event necessary, it is ok.
- Not connected, `constant`: `“#0=>fb.dindst”`. A not connected pin is set to the constant value “0”. It is ok. The code generation should deal correctly with it.
- `zout`: `“fbsrc.dout%=>fb.pindst”` The driving output is a state variable, style `ofpZout…` in the graphical model. The output value can be taken without an event. It is ok. But for tracking the update event chain, this output is handled as an event relevant input, see next.
- `bEvDataPropg`: `“^fb.evSrc:doutSrc+=>dinDst”`: The `dinDst` is driven by an `doutSrc` which is already driven by an event in a propagated chain. This `evSrc` is taken as one input for the `evinDst` firstly stored in a `listEvoutSrc`. This list is temporary built for the `dinDst` of the

checked FBlock inside the `checkDinOtherAndConnectEv(...)` operation.

The storing of `evoutSrc` is done by calling `addEvoutSrc(evoutSrc, list..)`. This operation checks the `evoutSrc` whether it is already stored in the list, but also whether another `evoutSrcGiven` is stored in the list with its relation to the `evoutSrc`. If the `evoutSrcGiven` is driven by the new coming `evoutSrc`, then the `evoutSrc` does not need to be stored, because the `doutSrc` comes from an FBlock which is before in the event chain. It can be used without regarding its `evoutSrc`, because this event forces the `evoutSrcGiven`. But vice versa if the `evoutSrcGiven` drives the new coming `evoutSrc`, then this `evoutSrcGiven` is no more necessary. It is replaced by the `evoutSrc`. The `evoutSrcGiven` is then removed from the list and the `evoutSrc` is added instead, also responsible for the newly regarded `dinSrc`.

8.9.5.4 Discard the step if not all `doutSrcOther` are driven by events yet.

The result of this check is the true/false decision whether the found event sources of all inputs in the list `listEvoutSrc` can be connected to the `evinDst` of the checked FBlock. It not all `dinDstOther` are driven, because its `doutSrcOther` are not yet all registered in an always built event connection, the `listEvoutSrc` will be discarded. The same check will be repeated later, but then with more registered `doutDstOther` in event chains.

8.9.5.5 Connect the events if all `doutSrcOther` are driven by events

In the positive case the event connection can be done.

The operation connectEventMaybeJoin(...) (www). does the work. It gets the `listEvoutSrc` from chapter 8.9.5.3 Check all other dinDst and the `evinDst` to connect.

For that some situations are possible:

- ***only one:*** If the `listEvoutSrc` contains only one `evoutSrc`, and the `evinDst` has not a given connection, then it is very simple, both should be connected.

For example if you have the following situation:

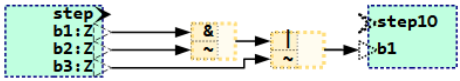


Figure39: ExpressionExmpCombiBoolean.png

then the right boolean expression (v) is driven intrinsic by two events, the `evoutSrc` of the left boolean expression (&) and the input step event. But the input step event is contained already in the `evoutSrc` driven from the left expression (&), hence not in the `listEvoutSrc`.

- ***OR:*** If the `listEvoutSrc` contains more events, the `evinDst` has not a given connection, then it is already clarified in `checkDinOther…(…)` that all events comes from different event chains. But if all the data inputs are provided by all this events, means any event provides all data, then the events are simple wired all to the `evinDst`, it is a OR relation. Any event in the `listEvoutSrc` can drive the evinDst independenlty.

- ***JOIN:*** If If the `listEvoutSrc` contains more events, and the data comes from different event chains which presents usual a parallel structure, then both event chains should be reached the point where the data are ready.

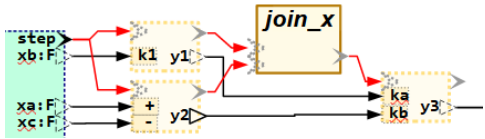


Figure40: EventParallelJoin.png

This situation is shown in the image above. y1 and y2 are the necessary data, which are calculated parallel in the graphic, parallel if the program is executed with parallelization (using multi core technology or such) or just

calculated one after another. If both are ready, then the event for y3 should come. This is done by the JOIN_UFB FBlock which is inserted automatically. The `listEvoutSrc` contains the event from y1 and y2.

- `init` ***and*** `ctor` ***handling:*** If a data flow is used both for any other event and for `init` and / or for `ctor`, then the init driven event chain is only connected to the `init` event as `evinDst`, same as for `ctor`. And an `init` or `ctor` driven event is not connected to another `evinDst`, if the FBlock has a `ctor` respectively an `init` event.

It asserts that the construction does only call the `ctor` operation, if it is existing. And also `init` calls only `init`. Look on the small examples:

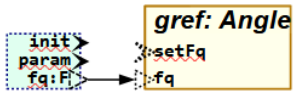


Figure41: EventNoInitConn.png

In this simple case the data `fq` are provided with `init` and also with `param`. But the connected FBlock uses the data input `fq` only with any other event, here `setFq`. The FBlock has an `init` event, but just not related to `fq`. That's why this data connection forces only connect `param`→`setFq` and not `init`→`setFq`. If the same FBlock would not have an init event, then `init`→`setFq` will be connected, as specific handling of any FBlock with no init routine in the initialization phase.

Either the `listEvoutSrc` contains only one event, that one which was originally tracked. Then this only one event is connected. It is the simple case.

If more as one `evoutSrc` is in the list, then the following decision is necessary:

...Then a `Join_UFBgl` FBlock is necessary to firstly join this more `evoutSrc`, and the output of the Join FBlock is connected then with the `evinDst`.

- If these events come all from the same source for all `dinDstOther`, then both events drives the data. The events are independent. Both are connected to the `evinDst`. It is an OR connection of events.
- If the events are independent, one drives a part of doutSrcOther, another drives other data sources, an AND connection os necessary for the events. In other words, all these events are necessary to deliver the data (`dinDstOther`) for the given functionality, the `evinDst`. The AND of the events are organized by an ***JOIN_UFBgl*** FBlock which is inserted in the event flow. The function of that *JOIN_UFBgl* is similar with the E_REND FBlock in IEC61499 (REND = rendezvous of the events), but the *JOIN_UFBgl* have a variable number of inputs.
- The last of this cases is, if an event connection is also existing (from the graphic) independent from this data driven event connection. Then it means the event connection determined from graphic is necessary because of the intention of the graphic (not questinoned), and the other event(s) are necessary because of delivering data. It means also a ***JOIN_UFBgl*** FBlock is necessary to fulfill this situaltion.

8.9.3.6 Put evoutDst in the queue to continue

Last not least the event outputs from the FBlock associated to the `evinDst` are determined. If the FBlock is simple, this is exact one event. It is possible to have more events. This is for ***Composite FBlocks*** in IEC61499 terms or also for ***Simple FBlocks*** with only one operation. It is also valid for Standard FBlocks with a simple reguar state machine, see chapter 7.2. FBtype kinds and their usage (due to IEC61499). This output evoutDst are put in the `queueEvout` to find

more data driven event connections.

If a FBlock has a more complex state machine (ECC = *Execution Control Chart*), then its output events are driven due to the execution of its ECC, hence builds new event chains which are connected from there. This **evout** are put in the **queueEvout** from beginning to build the independent event chain. The quest whether and when an event is created is not related with this event chain algorithm.

Note: For code generation it builds callback operations from the ECC execution.

xxxxxxxxxxx rest weg

It is important that a FBlock's event input **evinDst** can be added to the event chain if all **doutSrcOther** are provided with data from currently end points of clarified event chains. One of this end point evout is anyway the event which has determined the data source. Usual only this only one evout may be necessary, then it is simple.

If more as one event chain delivers the data necessary for the event inputs

It means either the other event associated din of the FBlock are provided with const values, or values from other events (from a **ofpZout…** dout pin), similar as a *"rate transition"* or *"unit delay"* in Simulink, or just they are already reached by the own event chain marked with the number of the event pin.

If the din is provided with a dout which is associated to another event chain and which is not a state value (**ofpZout…**), this is an error in the graphical model and shown as that. A mix of data from different event chain without dedicated designation as state value is a prone of error in functionality. That's why it is rejected. The algorithm itself may be ignore that fact.

If any din is just not provided with an already event driven dout, then it is assumed that this FBlock should be inserted in thís event chain before, should be calculate first. For that checkDinOtherAndConnectEv(...) is called recursively, but with this depending evin on the depending FBlock. This is a necessary data branch which may be also detected first in another tracking flow, or it is never detected first because it depends only on const or **ofpZout…** pins. Then it is the only one possibility to include it.

The event chain is then built from the starting evout of the first recursion to this operation to the evin of the last found proper FBlock in recursions. Going back after recursions

8.10 Code generation due the to event flow

As written in 8.9 Identification of the event flow due to data flow the event flow results vital from the data flow, inclusively some manual given event connections. The code generation can now use the event flow.

For the following presented kind of code generation it is presumed that all FBlocks are arranged in the same memory area. Dispersed FBlocks are specific designated, they break the built event chains. It is also possible, but not explained here, that the event flow combines several hardware devices, with communication.

Each `evin` of a module results in one operation of this module which contains the content of all FBlocks in one event chain.

It is possible that also intermediate `evin` inside a module are built. These builds also operations, but these operations should be called only internally due to the event sources. Especially FBlocks with state machines (ECC in IEC61499 words) are candidates for event emitting. This is regarded later (TODO for further versions).

Each `doutMdl` can have an access operation. It is a getter (Object orientated). Either the gotten value is immediately accessible, so the getter can be removed by code optimization (only to hide the access to a private output variable), or this operation can execute an expression using more as one states in the FBlock. If the FBlock or this part of a FBlock has no states, it is combinatorial, then the access operation to the `doutMdl` can immediately access the inputs of the module. Then the operation to the `evinMdl` is not given and not necessary. But this feature is in the moment (2014-03) also shift to a further version.

Following the script for C-source generation is shown and discussed:

6.6.3 Using a templates for code generation with OutTextPreparer

This is the general approach: All generated codes are controlled by templates, see *www.vishia.org\Java\pdf\RWTrans\OutTextPreparer.pdf* (www). Hence it is possible to adapt the code generation due to also specific approaches and styles.

The templates for code generation can be controlled by the option `-tplCode:path/to/templatefile`, whereby more as one file is possible (use the option more as one). If this path is not given, the internal templates for standard C code generation are used. This templates are stored in the jar file in the internal path `org/vishia/fbcl/writeFBcl/cHeader.otx` and `.../cImp.otx`. This files can be adapted if the tool is adapted, but only in consent with maintainer of the sources. The recommended way for user experience is: Copy this files to your own location and use the `-tplCode:` option.

The template files should set a variable which allows the association to determined file types. For C/++ generation this is `.c` or just `.cpp` and `.h` for the header files:

```
<:set:GenCode1=".h">
<:set:GenCode2=".c">
```

The name of this variable should be start with `GenCode` following by a number starting with `1`, as shown. Then the generation scripts with `<otx::GenCode1:` etc. are used to generate a file with the name of the module (in the `ofbTitle` style box in the graphic) and the here given extension. It means you can also generate some information files with any data representation from the internal given data.

The directory of the output files is the argument `-dirCode:path/to/dir`. The file name is the module name, which is written The extension, added to the module name as full file name, is that text, which is defined in the template with

```
<:set:GenCode2=".c">
```

adequate to each `GenCode…` start script.

It means you can have more as one file code generated with any content controled by the template. You can for example also generate reports from the data content, xml files or csv, and more.

The main script for the whole file should get internal data structure of a module as argument, hence should start with

```
<:otx:GenCode2:mdl>
<:type:mdl:org.vishia.fbcl.fblock.Module_FBcl>
```

as also shown in the following Snippet 42: .

Snippet 42: Start of the script for C code generation in the code generation template example

```
<:set:GenCode2=".c">                          ## extension .c for the c-File

<:otx:GenCode2:mdl>
<:type:mdl:org.vishia.fbcl.fblock.Module_FBcl><: >
/**Generated by org.vishia.fbcl. made by Hartmut Schorrig, vishia.org script 2024-03-23*/
#include "<&mdl.name>.h"
<:for:header:mdl.iterImport()>#include <:><&header.getValue()><:>><:n><.for>


<:for:evinMdl:mdl.fbp.evout>                              ##all input events of the module
<:type:evinMdl:org.vishia.fbcl.fblock.Evout_FBcl>
<:if:evinMdl.name.equals('init')>
 .....
<:else>
/**Operation <&evinMdl.name>(...)                  ## for each input event generate an operation
 */
void <&evinMdl.name>_<&mdl.name> ( <&mdl.name>_s* thiz<: >      ## name_TypeName( TypeName* thiz
  <:for:refMdl:evinMdl.iterPort()>
  , <&refMdl.dType().dt().typeRef.name> const* <&refMdl.name><.for><: >          ##argument list
  <:for:dinMdl:evinMdl.iterDout()>
  , <&dinMdl.dtypeCpp()> <&dinMdl.name><.for>
) {
  <:exec:prcEvchainOperation(evinMdl, OUT)>                     ## whole body of the operation
} // <&evinMdl.name>_<&mdl.name><.if>
<.for>

<.otx>
```

This is the whole script for the .c-File, only the `init` event is fade-out to increase overview. It is similar.

The type of the argument `mdl` is tested in the script in the second line. The test itself is an assertion (necessary?) but more an asserted documentation. You see here which class is really used as container of the data. Look to the Javadoc for fbcl/fblock/Module_FBcl.html (www).

The `<: >` on end of the line after `<:type:…` prevents output of a newline, the script text continues with the next given text (`<: >` means, skip all white spaces in the script).

With `<&mld.name>` the value of the field **`name`** (in Java) is output in the **`mdl`** data.

Last not least this script iterates over all `mdl.fbp.evout` , and prod

8.10.5 Tracking the event chain for a module's operation

6.6.2 Access operation to dout, arguments

If a dout access operation uses values from din of the FBlock, this values should be delivered from the back connected outputs. This is typical for expression FBlocks, but also for some other ones.

The `DoutType_FBcl#mUsedInputs` contains the bit mask for the `din` due to the `dout`. The inputs with their types builds the arguments, the argument order is the order of the inputs in the type. If the instantiation has more inputs due to one type din (multiple pins) then all inputs in order are used.

8.10.6 Code generation for one FBlock, one line or statement in the chain

For one evin `prcEvin(…)` is then called. It checks the conditions of the FBlock in the order of the following sub chapters.

8.10.6.1 Generation with a FBlock specific script

First with the typeName of the FBlock a proper type specific otx script is searched. If it is found, it is called with the arguments

- fb: The FBlock instance
- dout: null or the first dout of the fb, this helps for some typical FBtype
- din: null or first input of the fb, same
- doutype: DoutType_Fbcl of the dout or null if not given.
- evin: Evin_FBcl the evin of the FBlock which is triggering
- evSrc: Evout_Fbcl the event before.

The following example shows the snippet to generate a `ofpZout…` variable ***TODO because of new Expr approach this example is nor more proper*** setting on output of an expression. Here in the script it is clarified that this variable should only set with an update event, in the update routine. This is a FBlock-specific condition and hence tested only here. The preparation event is indeed connected to the FBlock that presents the variable, but it should not be effective.

Such an FBlock is contained in the fbd file with a line (example):

```
xdabz : VarV_UFB;
```

In the otx script for example the comments can be changed. The `thiz->` is a part of the translation script and can be replaced, etc.

Snippet 43: Example script for C code generation for a specific FBlock

```
##Set of the value(s) of a VarZ_UFB FBlock (output variable of an expression in an instanc ...
#
<:otx: VarV_UFB: evSrc, fb, evin, din, dout, doutype>  ##dout is the expression output

  // <&evSrc.nameFBpin()> --> <&fb.name>.<&evin.name>  otx: VarV_UFB (<&fb.typeName()>)<: >
<:if:din.isComplexDType()>
  thiz-><&fb.name()>.re = <&genExprTermDin(din,'.re', OUT, 0)>;  // <&dout.nameFBpin()>
  thiz-><&fb.name()>.im = <&genExprTermDin(din,'.im', OUT, 0)>;  // type is complex, otx:<: >
```

```
<:else>
  thiz-><&fb.name()> = <&genExprTermDin(din,'', OUT, 0)>;  // <&dout.nameFBpin()><: >
<.if><: >
<.otx>
```

Exact this script is used to set an expression output variable. The output variable itself is the FBlock VarV_UFB and the expresssion which determines it is immediately connected before.

Generally the `FBexpr_FBcl` which does not have an output variable are skipped by the 8.10.5 Tracking the event chain for a module's operation. This expressions are evaluated by tracking backward input values as described in 8.10.12 Code generation for Fbexpr.

TODO an proper FBType for complex multiplication expression should be created in the Java data and hence should have a proper otx Script. Without that special handling: If a variable is not scalar, especially complex as here shown, or an array (TODO), the code generation works component wise. It means it does not automatically a cross product for complex values, instead multiply the components. But this is faulty, because a complex multiplication makes also a cross product as

```
y.re = x1.re * x2.re - x1.im * x2.im;
y.im = x1.im * x2.re + x1.re * x2.im;
```

That's just an important TODO solve in the next time. How to do: The type and operators of the expression should be detected, and with this string the proper otx script should be gotten and used. Hint: The output of such an expression for cross multiplication of complex should anytime a variable. Elsewhere it is not possible to generate code because it cannot be back tracked through such complicated stuff if more as one cross multiplications are in the term. The intermediate results

8.10.6.2 Expression to set an element in a variable

TODO

8.10.6.x Set the module output

8.10.6.x create code for ctor

8.10.6.x create code for init

8.10.6.x call any FBlock content

8.10.12 Code generation for Fbexpr

The possibility of expressions in FBexpr_FBcl (www) is flexible, see using description in chapter 6.4 Expressions inside the data flow on page 50. General four kinds of generation are to be distinguished:

- **“.”**: Set components of an output variable. That is **.re**, **.im**, elements of an array, elements of a used defined structure. The expression should have exact one variable on output, see 6.4.5 Set components to a variable page 55

- **“:”**: Access to components of the connected only one input variable. See 6.4.7 FBexpr as data access page 56.

- **“=$”**: Generate the expression as statement with assignments to the given variables on the expression outputs: This is done if all outputs (often only one output) is a variable, not a **ofpExprOut**, or also if the one **ofpExprOut** is not connected (but other outputs as variables exists).

- **“~&@%”**: Generate in line as expression term. This is done if one or the only one output is an **ofpExprOut** and it is connected to another input.

The characters in **“…”** are the output of the FBexpr_FBcl.getAccess() (www) or just the first character in the **expr** constant input able to see in the **.fbcl** file (IEC61499). All variants are implement by the same operation FBexpr_FBcl.genExprOut(...) (www). Exact this operation is called for the variable on the expression main output.

For **cAccess** = one of **“~&@%”** there are

This is for scalar values or for one component for component wise values.

The expression with an output variable to assign is described in 8.10.6.1 Generation with a FBlock specific script shown with the `otx:VarV_UFB` script.

The end point or just start point for back tracking of an expression term is always an input of a FBlock. This is for data for any FBlock, but especially here the input of the **VarV_UFB** FBlock to set the variable value. As seen in the script Snippet 43: Example script for C code generation for a specific FBlock, the otx-element
`<&genExprTermDin(din,'', OUT, 0)>`
is inserted for the input(s). If the variable consists of more components, here the complex parts **.re** and **.im**, then the expression term is calculate independent for both components. Then the component access is given and added on each variable access in the expression term. For example the generated code for a longer complex subtract term is

```
thiz->xdab.re =  (x1.re-  (thiz->h1.yabz.re+
          thiz->h3.yabz.re) ) ;  // xdab.V V
thiz->xdab.im =  (x1.im-  (thiz->h1.yabz.im+
          thiz->h3.yabz.im) ) ;  // type is
          complex, otx: VarV_UFB (VarV_UFB)
```

This is due to the graphic Figure38: OrthBandpassFilter.odg.png page 85. For both component of the complex summation one line with an expression is created, due to two `<&genExprTermDin(din,'.re', OUT, 0)>`. and `<&genExprTermDin(din,'.im', OUT, 0)>`

The **genExprTerm(din, …)** is an operation in fbcl/writeFBcl/WriterCodegen.html#genExprTermDin(...). (www). It is programmed in Java and primary not adaptable by a comprehensive generation script, but details are adaptable:

- First is is tested whether the input is not connected. Then either the constant value stored in the input (**Din_Fbcl#getConstant()**) is called. The numeric constant value written in a simple form due to IEC61499 is converted in a proper presentation for the programming language. This is controlled by (...TODO yet without conversion). If a constant is not given a **0** is replaced.

- If the din is connected, then fbcl/writeFBcl/WriterCodegen.html#genValueDout(...). (www) is called from the source of connection. See there for further explanation.

- If the output is an expression output without such specifications, then the inputs of this FBexpr are summarized with its operators and also factors on the `K..` inputs and constants. For that the operation fbcl/writeFBcl/WriterCodegen.html#genValueExprDin(...). (www) is called.

8.10.12.1 What does genExprTerm(...)

- If more as one input exists, then first a `(` is added, and last a `)` . It means the expressions with more operands are always in parenthesis, because anytime the operators can have a different precedence. The arrangement of the FBexpr in the graphic is determining.

- The operator for the input is prepared in fbcl/fblock/FBexpr_FBcl#setOperatorToPins(...) (www). This operator per `din` is output to the generated code if the `din` has either a connection, a constant or which is either connected direct to an output (then the expression term is simple, one state, only the output variable or operation), or the input has a constant value, or just this is connected to an `ofpExprOut` pin of an expression.

This otx-Element calls the `ofpExprPart` refers a variable. The `setOperatorToPins()` checks the admissibility of operators (do not mix multiply, add, boolean) and removes a left side unnecessary operator because in expressions in all programming languages the binary operators are between the operands. Unary operators can follow the binary ones.

- The operator stored in fbcl/fblock/FBexpr_FBcl#setOperatorToPins(...) (www).

***** End of document *****