Diagrams for UML and Function Blocks drawn with Libre Office

Dr. Hartmut Schorrig
www.vishia.org

2023-09-24

Table of Contents

Table of Figures

1 Basic idea using Libre Office for Graphical programming

One advantage 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** was tested to draw the diagrams, and a translator to evaluate the content was written. 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 "*UML 3.0 first*" - Join FBlock Diagrams with 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.

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 *FBcD*. The "*c*" means either "*class*" or may be also "*connection*" for the UML like connections between FBlocks as supplement to the known data flow for ordinary FBlock Diagrams. 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, basic considerations

This chapter shows how capabilities of **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 11 pt, as written also here. A smaller font (9 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.

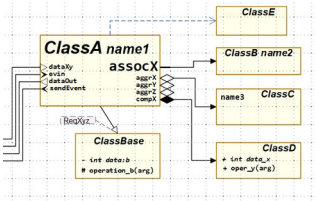

Figure 1: View 40%

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.

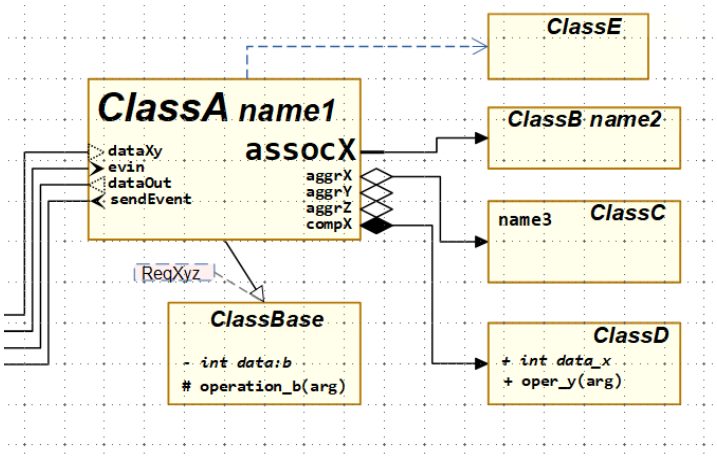

Figure 2: View 100%

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:

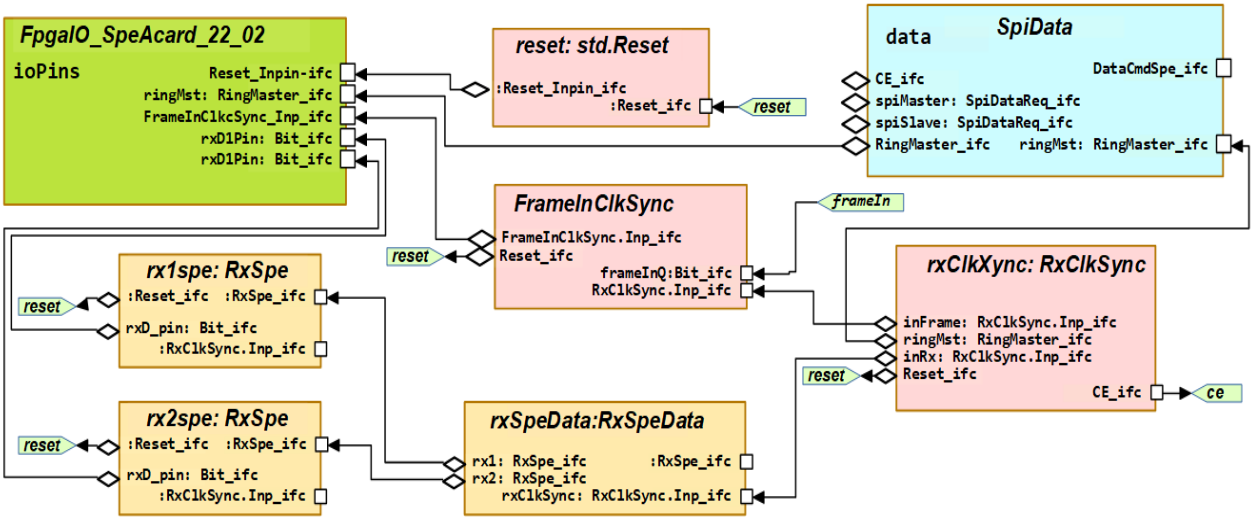

Figure 3: Example for a Module Diagram

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 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 this class, not a property of relations between this classes. It is exactly the same as for data. A data element has a type, and a reference has also a type, the type of the referenced class. It least the name of a reference is only 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 shown as 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 5.5 UML Class Diagram with ports page 24.

This 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 elements contains a text, which is the identifier for the element or also the type specification. The text is written outside on the element itself by using the Libre Office 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.

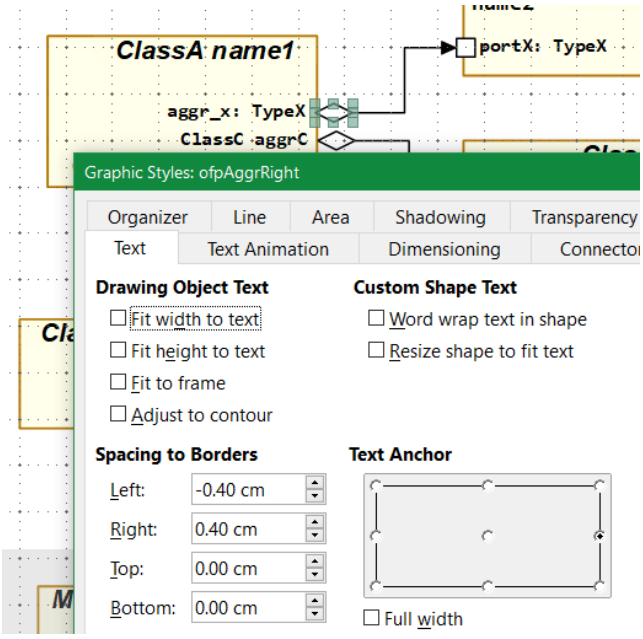

Figure 4: Text alignment in connection elements

Now, an element or more precise a connection to the class is shown as this small figure. But not the figure itself characteristics the element for code generation, instead the associated style sheet. The look of the figure can be changed, should not be changed, 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 7. The style sheet should not be changed by the user. It is defined for this kind of diagrams.

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 are common statements.

3.3 Connectors of LibreOffice for References between classes

The connectors as known from LibreOffice are the proper possibility to connect blocks, which are classes or objects. The connection can be done for the class itself, or for one one of the elements.

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 elements of the class. 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 Sheet. 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

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

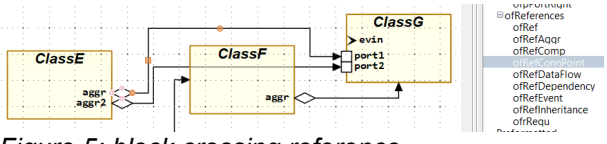

Figure 5: block crossing reference

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

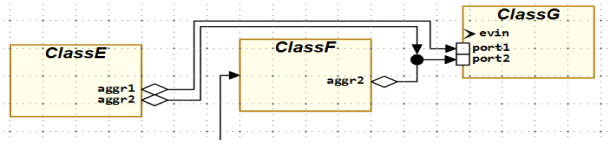

Figure 6: Using a connection point for aggregations

also from Function Block Diagrams for data flow. The difference is only: Data flow and electrical schemes has one source and more destination. An aggregation has one destination and can have more sources. The reference line to the connection point is either a simple `ofRef` which has an arrow on its end, or it is the same as in the image above for concatenation of reference lines, with style or type `ofrConnPoint`.

3.5 Diagrams with cross reference Xref

Image Cross Reference usage

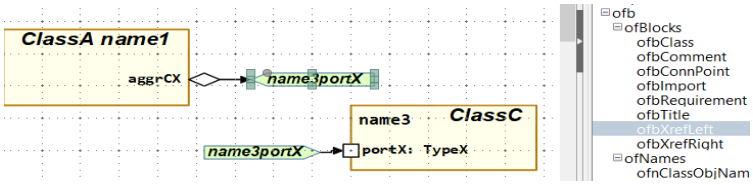

Figure 7: UMLdiagramXrefExample.png
Cross Reference usage

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

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

- The figure for the Xref can have any form, but should use the given form (copy it from template). The Style Sheet should be either `ofbXrefLeft` or `ofbXrefRight`, whereby the difference is only the text alignment to left or right.
- The name in the Xref symbol should be a mnemonic name for the functionality, valid for this diagram. Here it is a combination of the type of the port and part of name, maybe proper.
- 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.

3.6 All Kind of Elements with there Style Sheets

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

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

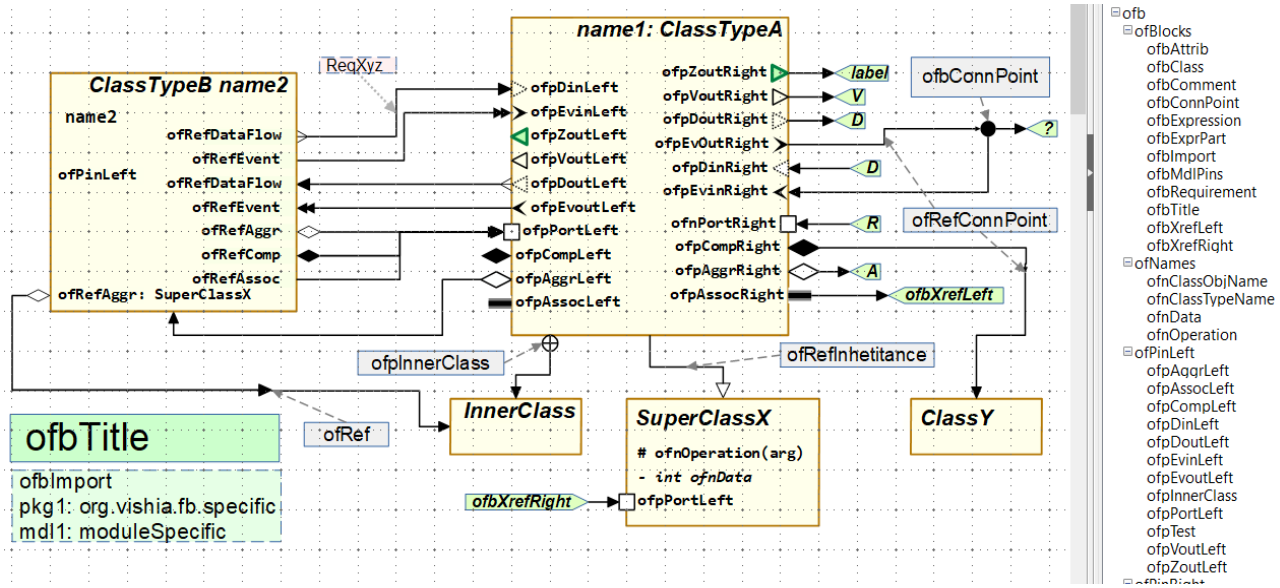

Figure 8: UMLallConnections.png

Right side you see some style sheets. You can use this image (given in the file `UML_FB_DiagramsTemplate.otg`) 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.otg` zip archive. This is a simple, proven workflow that has not been recommended as often, but it works:

* Copy the original `UML_FB_DiagramsTemplate.otg` file to `UML_FB_DiagramsTemplate.otg.zip`
* Open the zip file by a unzip tool.
* Copy the internal `style.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 `style.xml` with the `style.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: ClassType` contains all connection elements for the concept described in 3.2 Using figures with style sheets for elements page 8. The identifier of the style sheet is here used also as name.

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.

The type name of a class is marked with the style `ofnClassTypeName`. A class can have also an instance name, then it is an Object or a *Function Block*. A super class cannot have a name, also not a composited class. Because there are defined by its relation to the using class. A *Function Block* presents a class which is instantiated by a here usual not shown main class of a module as composite. Then its name is written either in the element `ofnClassTypeName` as first, then with colon, or after the class type name. This writing style is also possible for all names of the elements. The type can be written their also, shown only for the aggregation left bottom `ofRefAggr: SuperClassX`. The type of a connection (a reference) may be sometime interesting. In UML a reference should refer always the class which is the reference type. But in this here used combined class-object-Diagram (objects are *Function Blocks*) sometimes the reference goes to the used instance with a derived type, and the reference type should be shown.

The style `ofnClassObjName` is the other possibility to show the name of a Function Block or just class object. Then only the name is written here, parallel to the `ofnClassTypeName` which may/should only contain the class type name. In the left `ClassTypeB name2` both is used.

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 5.7 Data and event relations between FBlocks.

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

This is basically and your daily work.

- You can catch all content of a class block inclusively the connection elements with selecting with left pressed mouse. Then you should select "Shape-Group-Group" from the menu. It is ctrl-G per default.
- After grouping you can move your class block simple with the mouse or by cursor keys as a whole. All Elements are moved together and are stable in the class block.

  The grouping is also recommended and necessary for content evaluation. You should anyway have a grouped situation.
- But in the grouping state you cannot connect the elements of the class with a connector. Only the glue points of the group itself are accessible, and that is not desired.

It means before connecting, and also before shifting some elements you should ungroup. This is in menu "Shape-Group-Group". You should assign the hot key strg-sh-G to ungroup to have a fast work flow. Sometimes also strg-U may be used. Originally this is not assigned. Use menu "tools - customize" and then keyboard. Search "ungroup" and assign the desired key.

You need very often Group and Ungroup.

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.

5 Semantic discussion

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 What is a class, package, component, module and a FBlock

A class is well defined in programming languages such as Java or C++ as language feature. But for usage, a class can present only a simple functionality, or it can contain a lot of sub instances of classes with a comprehensive functionality.

A package is well defined in Java language. In other languages (C++) it can be seen as association of some classes to just a package, for example with the same name space. In UML a package is also defined in this manner, without exactly rules for any implementation languages.

In UML a component is a functional unit. It is not related to the class thinking, and has also its own diagram kind, the *Component Diagram*. A component is usual a greater unit, for example the whole operation system of a hardware component in a device or a specific part of a runtime system of a device. Also a hardware part may be designated as component. In this manner the component is not in focus for the FBc Diagrams.

The terminus *module* has different meaning in different usages, it is commonly a part of a whole. Sometimes also module and component are confused. In Software programming it is often used as term for *Modular Programming* [6]. But a module can be usual mapped as class, if Object Oriented Programming is used, as presumed for the FBc Diagrams. Hence it is synonymous: The term module is used here if the functionality of cooperation of different class instances are in focus. But one module is also a class seen from outer to build a more comprehensive module. Then last not least the whole functionality of a modular or Object Oriented design is a component of the system presented in the UMLs *component diagram*.

A unit has familiar the same meaning as module or instance of a class. *Unit test* is the separated test of one module or also a separated test of one instance of a (comprehensive) class.

The wording "*Function Block*" (FBlock) comes from the *Function Block Diagram* which is not related to UML but overall used in the Function Block oriented graphical programming or modeling. A FBlock is a unit with a dedicated function and its input and output pins. In ObjectOrientation it is an instantiation of a class. Whereby sometimes (in some Function Block Tools) a FBlock contains only one function, as well as a class can contain more. But a FBlock can contain different parts associated to more as one sample or step time (also in Simulink), or here associated to some events to processes. Hence they can run in different threads or also in synchronous sample times.

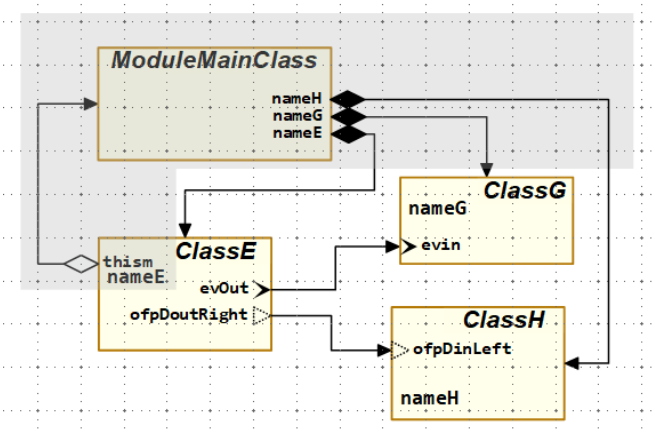

Figure 9: UMLFBlockdiagramDataflowExample.png

For the FBc Diagram, the essence is, that a FBlock has a name. This name is not related to an (often used in the past) static instances which are not recommended and not a goal of this development. As also shown above, the names are defined in the module's main class as composition names. The `ModuleMainClass` does not need to be drawn for an ordinary FBlock

diagram. It is self-evident and part of the code generation.

5.2 Data and event flow and class relations in one diagram

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

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 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.

5.3 Event, data and reference relations

In UML the classes have only reference relations, the references between the classes. The references can be associations, aggregations and compositions, as known. But also inheritance and usage can be seen and dedicated as “reference relation”. Also a documentation block, an annotation for example to associate a requirement is a reference.

Whereas in traditional FBlock diagrams there are only data relations as data flow. It is important and familiar that the data flow is assigned to dedicated sample or step times. A more complex FBlock can have also more as one step time, then it has more parts for different data flows.

The step time binding is a specialized one, an event binding is more universal. An event can occur cyclically, then it is adequate to a step time binding. But an event can also occur on demand, if an out giving condition applies, or also depending from a state (state machine using).

It is interesting and promising that the widely proven FBlock programming in the IEC61131 standard 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 also from Simatic as also other companies (Codesys and more). Currently 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 connection 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. 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 right images 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.

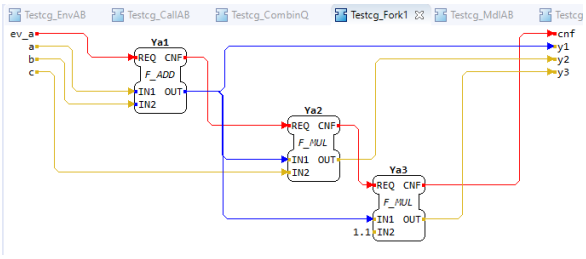

Figure 10: 4diac\Testcg_Fork1.png

The conclusion using the event driven principle is: To show the content of a graphical diagram in evaluated textual form the syntax of IEC61499 as textual representation can be used. This is done in this approach. Because of that a standardized textual representation of the graphical content is given.

5.4 Content of one or some diagram/s is a package or a module

One LibreOffice Diagram contains either a package in UML thinking, or a module in FBlock thinking. Both are suitable for each other because often classes from a specific module implementation are organized in an appropriate package. In FBlock diagrams presentation of one module in one diagram is familiar.

A Diagram should have a title, both for the module which presents it and for the package. This is shown right in the Box with style `ofbTitle`. `pkg`: and `mdl`: are the keywords. Furthermore, other packages can be designate with an alias. That is in the second green box with style `ofbImport` (marked here). The alias is left from colon, right side an unique identifying string for the package respectively for the module should be written, which can be evaluated.

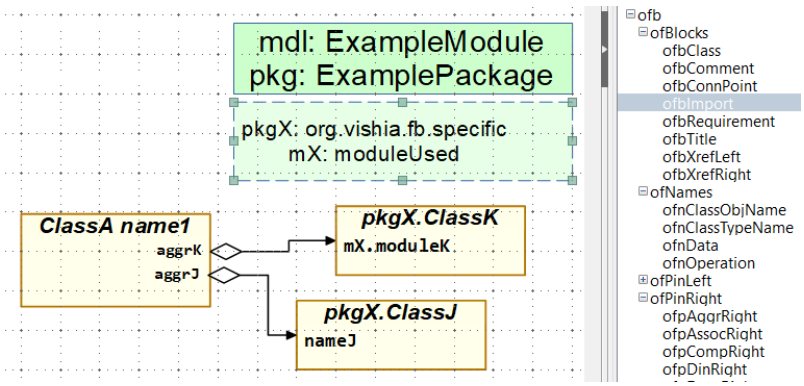

Figure 11: omdImportPkgMdl.png

Just, The `ClassA` is member of the own package "`ExamplePackage`". The instance of this class named "`name1`", a Function Block, is part of the module "`ExampleModule`". The FBlock `mX.ModuleX` can be found in another module with alias `mX`, which is located in "`moduleUsed`", described with other diagrams. It is only used here.

The second FBlock nameJ is member of this ExampleModule, but its type is locate in another package, here the `pkgX`, able to find in the package path "`org.vishia.fb.specific`".

Hint: The package path is Java-affine, for other languages for example a name space or the name of the header file can be written here.

The possibility to define different packages in one UML Class Diagram is not supported here. To show a package structure an enhancement should be done for this diagram writing approach. But usual, in UML there are often small problems in showing used classes outside of the own package because of sophisticated connections (the other package is separated in the diagram). Here using the alias approach classes from another package can be located closed to the using classes.

One package or one module can have more as one diagram! The content is merged as one content for the module, or for the package.

As well as other presentations (especially written source code) can be also used to generate the data content of one module or package. This is especially interesting for modules with the FBlock approach. Other tools, for example Simulink, does not support more diagrams for one module. But Simulink supports nested content in a Function Block with graphical content. On the other side Simulink has not a distinctive module structure. On code generation all modules (except "*Atomic Subsystem*" are flattend. This specificas may be known and regarding on planning of packages and modules.

If a module is viewed from outside as a class, of course its pins should be presented in the module's diagram. In Simulink there are specific so named ports as interface to outside. Here the denotation "*pin*" is used. If an aggregation from outside is necessary, it should

5.5 Black (gray) box thinking for sub modules

A used FBlock or class block in a diagram is first a black box. It is used per interface. The internal functionality should be known by documentation, but not seen in the same diagram.

But of course, if a FBlock is also a module described with LibreOffice graphic, this diagram file can be opened, viewed, also adapted. But note, it's a different module. This is first a test and documentation approach. A module should always be tested by module (unit-) test with own test beds, own test sequences. If you change a used module during use, you break the module boundaries, the principle of modularization. That is possible of course, working on a complete solution. But the boundaries of modules, with own tests, should be anytime retained. Test a changed module in the whole context of application, but test it also with specified test cases.

Especially, a used FBlock can be realized immediately in the target language code, for embedded control it is often C or C++ language. Then for tests, this should be run as “implementation code in the loop” – either with the real hardware (specific processor card) or also as running C/++ code on a PC platform with emulation of the hardware interfaces. See also chapter TODO test of the graphic modules.

In opposite, for example the tool “Simulink” (Mathworks) can open a sub module (named there “Subsystem”) immediately in the graphic editor, with nesting editing or also in a separated tab or window. The content of a sub module can be a library module which can be changed an later updated, or it can be a nested part of the one module. In Simulink this is necessary because one module can be presented only on one page. And this page is filled with not too much content, the whole functionality of a module needs more space. Hence parts are placed nested. This nested “Subsystems” are then not a black or gray box itself, it is really a part of the whole, should not be tested with an extra test environment. The problem for this work flow is: Where are the boundaries for a real own sub module. It is, of course, a more as one used library module. But from view of the structure of one system, this boundaries are blurring. That is the risk.

With this approach with LibreOffice graphic a comprehensive module can be presented with more as one page, in several parts in one file or also in more as one file with several parts, which should be summarized of course while translating (evaluating) the graphic content. A module can consist of different large content. But the risk for that is also given: Where are sensible boundaries for sub modularity? This is answered by the the architecture of the system, often also determined by responsibilities of more as one developer (-teams).

5.6 Interface definition of used sub modules

Primary a used module can be simple presented by its frame of style “ofbFBlock” with the adequate elements in a group. Primary no type definition is necessary. If the elements are related to existing elements, it is proper.

But for translation or evaluation of the graphic some information for the type of an used FBlock are necessary for correct translation. That is especially event and data associations, maybe the type of references if derived classes are referenced, and also of course names of the operations and the code in the target language to generate.

For that the interface of a used FBlock type is described with a IEC61499 fbd file (textual). This file is automatically created by evaluation of a graphical sub module and can be manually written for FBlocks which are used as target language FBlocks. For the syntax of this file see TODO IEC61499 fbd syntax.

5.6 Classes or FBlocks in a diagram and its relations

A UML-class is present by a rectangle determined as `ofbFBlock` as well as a FBlock. A FBlock is a named class, hence it is an instance of a class or just a Function Block in a Function Block Diagram. The class and instance name are either separated by colon before the class name or by a space after the class name. But also an extra name field of style "`ofnClassObjName`" can be used to determine the name of a FBlock as instance.

If you have relations between classes, then this are really class relations of references between the class types. Other than in UML 2.0 standard the reference relation is not only determined by the ***connection between*** the class blocks as association, aggregation, or composition. In all implementation codes there are ***properties from one class to another***. An aggregation etc. is presented by a reference variable in the source class, and the destination class in the diagram determines the type of the reference. Bilateral references are not given, present it in the diagram by two references, from both sides. Bilateral references are sometimes self evident in a first system design, but they are usual not persistent for the implementation.

Hence, an association, aggregation or composition is shown as pin on the source class and a reference connection to the target (destination) class. The right image shows the pin with name `aggrC` (marked) of style `ofpAggrRight`. The reference of style `ofRef` starts on the pin and goes to the here named FBlock of type `ClassC`.

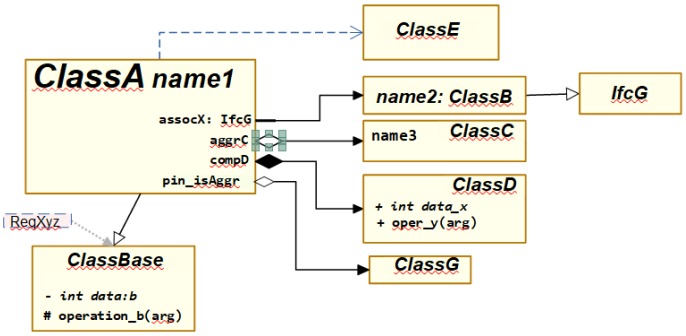

Figure 12: OFB/UMLclasObjRelation.png

The pin `pin_isAggr` is a simple pin of style `ofPinRight`. The connected aggregation is of style `ofRefAggr` which is marked with the diamond on the start side to recognize it as aggregation. This is the alternative, similar as UML (the style of the reference as aggregation determines it as aggregation, and not the pin style). But the name of the reference is also written here not to the reference (as in UML), but on the pin (as in ordinary programming).

In this Figure 12: OFB/UMLclasObjRelation.png the `aggrC` references not the pure class but the instance with name `name3` and class type `ClassC`. If this is the only one found relation of this ClassA for this aggregation, it gets the type `ClassC`. Other than on the `assocX`. It references the instance of type `ClassB`, but it should not be of type `ClassB`, instead of the interface type `IfcG`. That cannot be automatically detected. Either anywhere other in one of the diagrams for `ClassA`, the `assocX` is connected to only a class FBlock (without instance name) of this type `IfcG`. Then the relation to the class Block determines the type prior to the relation to a named FBlock. Alternatively as shown here, the type can also be given on the pin itself, as usual for programming languages. Then the type is not shown graphically, it is given textually, which is also or maybe better obviously.

But the value of the reference is determined by the referred instance, it means `aggrC` of type `ClassC` refers the instance `name3` for the whole run time of the application (in its property as aggregation). `assocX` of type IfcG refers initially the instance `name2`, but the instance reference may be changed during runtime (because it is not an aggregation, it is an association).

For compositions (style `ofpComp...` never a named FBlock should be used, it is an error in the graphic. Why: The composite FBlock is created by its parent and has the name of the composition pin in the parent. If it is created in the module, and referenced only from the parent, it is an aggregation.

5.7 Data and event relations between FBlocks

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 type of 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 ist 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.

The right image shows an example for data relations. The event relations shown in gray color need not to be drawn in the graphic, there are automatically created in the model. The image shows in gray what is automatically generated. But the event pins should be determined as shown (drawn black).

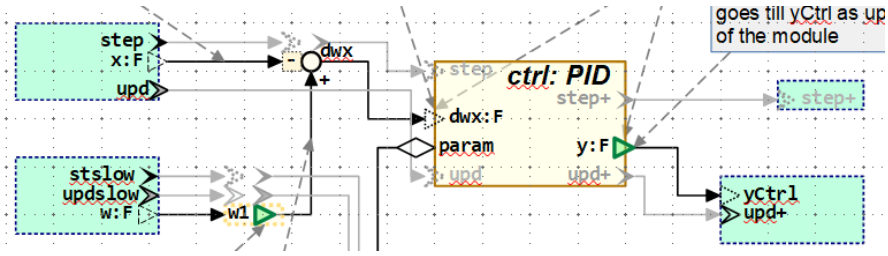

Figure 13: OFB/DataFlowPID4.png

The type determination for numerics is done by one character. It is concise. The letters `B S I J F D Z` are used for `byte`, `short`, `int` (32 bit), `int64`, `float`, `double` and `boolean`. That are the same letters as used internally for Java types. Adequate lower case letters `s i j f d` are used for complex types. Arrays with a determines length are defined by a simple number after the one-char-type, such as `F3` for an `float[3]` array. This is a concise simple style which needs less space in the graphic.

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.

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`. See TODO chapter events and state machines.

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 for input and output, whereby the input connector can have a text for the expression, and the output connector can have a name for a variable presenting the expression output. 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. The other possibility is: Using rectangles with style `ofbExprPart` and a data output with style `ofpDout…` or `ofpVout...` or `ofpZout…`. The last one is shown for the variable `w1`. This is a variable presenting the last state value or just a "*unit delay*" in Simulink thinking.

Variables of style `ofpVout...` are stored in the module's inner data and can be accessed by more chains from events of the primary same event source, it is a branching point in the data flow. Whereas variable of style `ofpDout…` can be established only as stack local variables in the implementation to find it for code generation, also usable for branching point but only in the same event chain (used in the same operation for implementation level). The distinction between this variable types may be interesting and relevant also for the graphic design, not only a detail of implementation. For example one can access all variables which are stored as state in a class with a debugging tool in runtime.

Expression terms, Multiplying input values: Rectangle figures of style `ofbExprPart` joint in a group with the expression figure style `ofbExpression` are expression inputs which can be added, subtracted, multiplied or divided in the whole expression term, determined by the first operation symbol. And it can be multiplied with a factor for the input value by following expression terms. The factor can access also connections from `ofbXref`... blocks (cross references) by its name. This allows a compacted graphic style also for comprehensive expressions.

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.

5.5 UML Class Diagram with ports

Ports are introduced in UML with version 2.0. They are firstly used in *Component Diagrams* as interaction points. The meaning of ports in class diagrams differences from destination languages for UML and for semantic approaches. Any where it is an access from outer to inner functionality without necessary knowledge of the environment class.

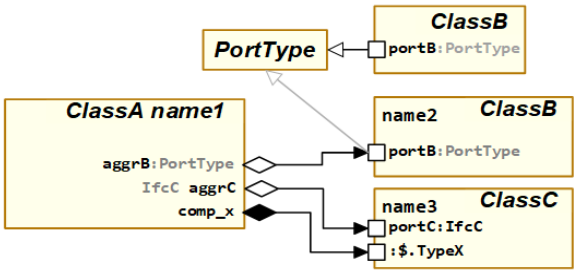

Figure 14: UMLdiagramPortExample.png: Class diagram with ports

An association line does not end on a class box, instead it ends on the symbol of a port (small square) which is designated with the style "`ofpPortLeft`" or "`ofpPortRight`". The difference between "`ofpPortLeft`" or "...`Right`" is only the text alignment, no more.

A port has a name inside the class, as shown here, but has also a type. The type can also be written here, but also in another class diagram or also only in the implementation software. The type should fulfill of course the type of the using aggregations, maybe inherit. The type can be written after the name separated with colon (shown here as `portB: PortType`) or also before the name separated with space.

As also shown in this diagram, the aggregation itself has also a name and can have a type designation, written in the same way: After colon to the aggregation text, here `aggrB: PortType` or also in this example `IfcC aggrC`. As known in ObjectOrientation, a reference of a base or super type can refer an instance of a derived type. If the diagram is checked and correct, the `IfcC` is a super type of the type of the port.

The types are detected also by the connection itself. If the type is given in graphic for the destination of an aggregation etc (a reference), the most abstract seen type is used automatically, it it is not determined by the software itself.

The next paragraphs should give a guideline how ports can be practically used.

In Java language inner non static classes are known, inclusively the anonymous class or interface implementation. This is a proper example for using the port technology. The inner class instance is an access point or port with an own type to the inner of the environment class. Adequate solutions are also possible in C++:

```
class ClassB {

  PortType portB = new PortType() {     // an anonymous class extension or interface implement.
    // ... implementation of interface
    // ... access to the Environment
  };  //Note: syntactically the semicolon is necessary here, it is an element definition.

  class TypeX implements IfcC {         // a port class which has access to the ClassB
    // ... implementation of interface
    //     with access to the Environment
  }
  TypeX portC = new TypeX();            // note that the TypeX is in namespace of ClassB
}
```

The `TypeX` is here defined as inner class type in name space of the `ClassB` (int the diagram above in `ClassC`) as environment class. It becomes a port because of its instantiation as `portC`, which is a composition in respect to the instance of the environment class. The access from the inner `TypeX` to the environment class is done by an implicitly reference which can be used explicitly as `ClassB.this` (shown explicitly as `this$1` in debugger).

Now the port `portC` is accessed from another class (`ClassA.aggrC` in the diagram) via the known interface of this inner class `IfcC`. For this application an anonymous interface implementation will

be also sufficient. The diagram shows only the type `IfcC` for the port, it means the implementation type is hidden, it is clarified in the class definition, not on usage level. The using classes do not need knowledge about the environment class of the port on run time, only to create the aggregation the environment `classC` should be known with the public element `portX`.

But, the `ClassA.comp_x` uses and should know the inner `TypeX` because it is a composition of `ClassA`. It means this instance is only used in focus of `ClassA`, but has access to inner things of its environment `ClassC` or `ClassB` in the code lines above. It is really an agent, owned by the creator but with access to another. Such constellations are sometimes necessary. In C++ for such the `friend` relations (C++ keyword) are used, in Java it is clarified with this inner class access mechanism. To create this agent inner class, the using `ClassA` should know the instance where the composite is created in, and should know the public constructor of the Port. The constructor should be called in Java in form:

```
ClassB.TypeX comp_X = name3.new TypeX();
```

The `portB` is instantiated only one time in its environment class and can be accessed by the name. The Diagram shows some possibilities how the type of the port can be declared. It can be done also by an inheritance relation from the port to an interface, or give the type on the port access.

Adequate implementations for C++ or also in Object Oriented C are possible, it needs an explicitly reference to the environment class, and an explicitly but maybe private inner class definition for the anonymous class in Java.

This concept of inner classes (non static) was created with the first version of Java in the first half of the 1990th. It is an interesting concept because for example the same interface can be used to implement more as one time in a class, each with an inner class implementation, or for example the similar functionality as for multi inheritance can be done, using specific inner classes for each inheritance bough. The port idea in UML 2.0 comes approximately 10 years later, and it seems to be non related to the Java inner classes, as well as the Java inner class concept was never adapted to C++ as standard. But the presentation of this concept with the port symbol as access port from outer to the whole class seems to be proper and matching. Hence it is used here. The port symbol is **not drawn** with the ***provided*** and ***required interfaces***. The one provided interface can be given by the type of the port, no more is necessary. If the port implements more as one interface (which is a special case), then the inheritance of the interfaces can be shown with the port as in Figure 14: UMLdiagramPortExample.png: Class diagram with ports above, also more as one. This is a provided interface. But it is more simple and pragmatic to use the port in the same kind as a class for connection, connect the aggregation etc. immediately.

The required interface is not presented by the Java inner class concept. A required interface is a property of a class as a whole, by its aggregations. That is a practical aspect. If ports of a standard UML 2.0 diagram should be implemented in a simple form in a class (C++ or Java), you can replace the required interface from the ports with each one aggregation. Or, if the port is used on a component, the port may be presented by a class itself with its aggregations, which should be satisfied by interfaces. The provided interfaces of a class itself or its ports in this drawing style are the the provided interfaces of the ports for UML 2.0.

6 Evaluation of the content for code generation and / or textual documentation

Working with Libre Office for drawing diagrams will be really non sensible if the content is not able to evaluate. Because:

An only documentation oriented documentation is dead. If the code is changed, sometimes the documentation cannot be updated, it is too much effort to find out all positions where the documentation should be updated. It is an old style to write documentation independent of the code.

But we know usual from other tools, that working immediately with text oriented sources in programming languages with proper IDEs (Integrated Development Environments) is often proper and effective. The combination with graphical programming is often sophisticated and lesser helpful. Precisely for this reason, this work, using Libre Office with evaluation of the content, was carried out as an alternative solution to the well-known and (advertised powerful) professional tools.

The goal for evaluation the content of this Libre Office diagrams is not: "*Just have another (more powerful?) code generation*". The goal is: present the content in a textual readable and comparable form. Then it can manually used, it can be used for text based comparison with older or other content. It can be used to compare with textual given source code, maybe with preprocessing the source code to get an adequate presentation (parsing the source code, output an intermediate format). In that kind the code can be tuned to the graphic so that the graphic matches to it.

The generation of the textual form from the graphic should be able to control by textual templates. So a specific code generation or a specific comparison format can be aimed.

That is shown in the next sub chapters.

6.1 The file format of odg

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.

- 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.

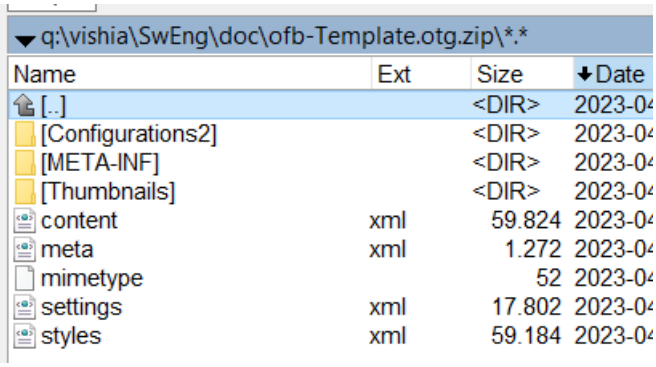

Figure 15: ContentOfodg.zip.png

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.

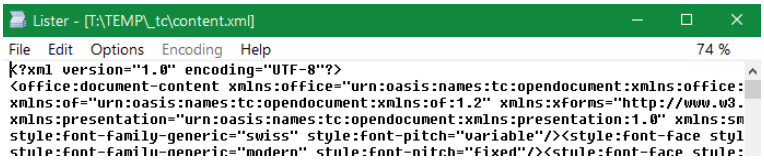

Figure 16: ContentOfodg-content-xmlPure.png

After beautification it looks like

```
<draw:g>
  <draw:custom-shape draw:style-name="gr21" draw:text-style-name="P1" draw:layer="layout"
    <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="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 7: UMLdiagramXrefExample.png Cross Reference usage page 10. 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.

6.2 Software to evaluate the graphic information from Libre Office

This software is given completely in two jar archives. You should start per command line:

```
java -cp tools/vishiaBase.jar;tools/vishiaFBcL_odg.jar org.vishia.odg.ReadFBcL_odg
```

If you start the software without arguments, you get an argument description:

```
...Reader content from odg for FunctionBlockGrafic
obligate args: -o:... ...input
-o:path/to/output.file
-datahtml:path/to/data.html
-oxmldatahtml:path/to/xmldatahrml.html
-oxmltest:path/to/outTest.xml if given writes the red xml input back
-obeauty:path/to/outputBeautificated.xml if given writes the beautificated input
path/to/input.odg
```

Given proper arguments produces the outputs.

For all example diagrams which are contained in the `src/UML_FB_DiagramExamples/odg/ofb-UML-Examples.odg` you get a text file which contains for example for the ClassA the following read out content:

```
/**A type definition ^= class in this package.
 */
class ClassA {  //
    ClassB assocX; //association
    ifcC aggrC; //aggregation
    TypeX aggrCX; //aggregation
    ClassC aggrX; //aggregation
    ifcX aggr_x; //aggregation
    ClassD compX; //compositon
    ClassC.PortType comp_x; //compositon
    EvTypeZ evin; //event_in
    EvTypeZ sendEvent; //event_out
    (float) dataXy; //data_in
    (float) dataOut; //data_out
} // end class
```

This is the summarization of all drawn content for the ClassA.

For the instance name1 which is type of ClassA the summarization is:

```
/**An object (FBlock) in this module.
 */
FBlock name1 : ClassA {  //
  association assocX  =: name2.@fbSrc ;
  aggregation aggrC  =: name3.@fbSrc ;
  aggregation aggrCX  =: name3.portX ;
  aggregation aggrX  =: name3.@fbSrc ;
  aggregation aggr_x  =: name2.portX ;
  compositon compX  =: @ClassD.@fbSrc ;
  compositon comp_x  =: name3. ;
  event_out sendEvent  =: name1.evin ;
  data_out dataOut  =: name1.dataXy ;

  // back tracking info:
  event_in evin  =: name1.sendEvent ;
  data_in dataXy  =: name1.dataOut ;
  <??null??> <??null??> ;

} // end class
```

Here you see the connections from and to the pins of this FBlock.

To produce the results in this form there is a template file with the following whole content:

```
=== odgPin
    <&pin.sType> <&pin.name>; //<&pin.kind.sKind>

=== odgPinObj
  <&pin.pinClazz.kind.sKind> <&pin.pinClazz.name> <: >
    <:if:pin.idxPinConnected><:for:cpin:pin.idxPinConnected> <: >
      =: <&cpin.fb.name>.<&cpin.pinClazz.name> <:if:cpin_next>
          ~ <.if><.for><.if>;



=== odgClass
/**A type definition ^= class in this package.
 */
class <&fb.sType> {  //
<:for:assoc: fb.idxAssoc><:call:odgPin:pin=assoc><.for><: >
<:for:aggr: fb.idxAggr><:call:odgPin:pin=aggr><.for><: >
<:for:comp: fb.idxComp><:call:odgPin:pin=comp><.for><: >
<:for:port: fb.idxPort><:call:odgPin:pin=port><.for><: >
<:for:evin: fb.idxEvin><:call:odgPin:pin=evin><.for><: >
<:for:evout: fb.idxEvout><:call:odgPin:pin=evout><.for><: >
<:for:din: fb.idxDin><:call:odgPin:pin=din><.for><: >
<:for:dout: fb.idxDout><:call:odgPin:pin=dout><.for><: >
<:for:pin: fb.idxUnspecConn><:call:odgPin:pin=pin><.for><: >
} // end class



=== odgObj
/**An object (FBlock) in this module.
 */
FBlock <&fb.name> : <&fb.fbClazz.sType> {  //
<:for:assoc: fb.idxAssoc><:call:odgPinObj:pin=assoc><.for><: >
<:for:aggr: fb.idxAggr><:call:odgPinObj:pin=aggr><.for><: >
<:for:comp: fb.idxComp><:call:odgPinObj:pin=comp><.for><: >
<:for:evout: fb.idxEvout><:call:odgPinObj:pin=evout><.for><: >
<:for:dout: fb.idxDout><:call:odgPinObj:pin=dout><.for><: >
<:for:pin: fb.idxUnspecConn><:call:odgPinObj:pin=pin><.for><: >
<:if:fb.fbPinDst><:call:odgPinObj:pin=fb.fbPinDst><.if>
  // back tracking info:
<:for:port: fb.idxPort><:call:odgPinObj:pin=port><.for><: >
<:for:evin: fb.idxEvin><:call:odgPinObj:pin=evin><.for><: >
<:for:din: fb.idxDin><:call:odgPinObj:pin=din><.for><: >
<:if:fb.fbPinSrc><:call:odgPinObj:pin=fb.fbPinDst><.if>
} // end class

===
```

This file is used with the

https://www.vishia.org/Java/html/RWTrans/RWTrans.html#_outtextpreparer

or see also its Javadoc:

https://www.vishia.org/Java/docuSrcJava_vishiaBase/org/vishia/util/OutTextPreparer.html

contained in the `vishiaBase.jar` file als general solution. It accesses immediately to the converted data.

7 Links and bibliographie

[1] https://en.wikipedia.org/wiki/Unified_Modeling_Language with hint to the founder Grady Booch, Ivar Jacobson and James Rumbaugh

[2] https://en.wikipedia.org/wiki/I-Logix David Harel, Statemate, State charts in UML

[3] https://en.wikipedia.org/wiki/LabVIEW The tool LabVIEW from National Instruments

[4] https://en.wikipedia.org/wiki/Simulink Simulink from Mathworks

[5] https://en.wikipedia.org/wiki/IEC_61499 Automation control programming system using event connections, see also https://www.eclipse.org/4diac/

[6] https://en.wikipedia.org/wiki/Modular_programming The term *module*, usage in software.

[7] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/ gives a proper overview about UML

8 Requirements

The Identifier of the requirements can be found in the documentation and in the tool software. The numbers are unrelated, it's only an id without sorting approaches. The requirements are agil. It means they describe the current state. The Requirements are not sorted here by number, they are sorted by content association. Use ctr-F to find a specific number.

Req740 The outputs of FBlocks can be current values (from the same step time) `ofpVout` or the values from the step time before (state value, adequate unit delay in Simulink) `ofpZout`.

Req741 The output of Expression blocks can be state values (adequate unit delay in Simulink): `ofpZout` or also current values stored in class variables of the module `ofpVout`, or current values stored in stack variables `ofpDout` for more as one usage, or the expression is used only temporary with connection from the expression block.

Req742 The outputs from the step time before `ofpZout` can be used in any other step time or event chain. Problems of data consistence of more as one output should be solved by the implementation. Possibilities are for example double buffering. It should not be a part of the immediately modeling. It is possible to support some consistent variables by a structure as text input for code generation.

Req743 The outputs from the class variables can be used also in an other step time or event queue. The difference between `ofpVout` and `ofpZout` is meaningful only for the current step time or event chain.

2023-09-24 by Dr. Hartmut Schorrig www.vishia.org