(left empty side before page 1)
Note: Most of Browser pdf presentations does not support the book mode.
Remove this page if you want to have a PDF file in book mode.

OFB – Implementation – chapter 7

(empty first page)

Dr. Hartmut Schorrig
www.vishia.org 2024-10-09

7 Inner Functionality of the Converter Software

This first level chapter should show the inner functionality of the converting software to read Open/LibreOffice diagrams, translate to and read IEC61499, and generate source code in the target language.

Table of Contents

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, or Open-Jdk), with is standard libraries, without additional libraries. The vishia basic library `vishiaBase.jar` is used. This library contains

all basic functionality for example to read XML.

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

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.

7.1 Data Model data classes

Table of Contents

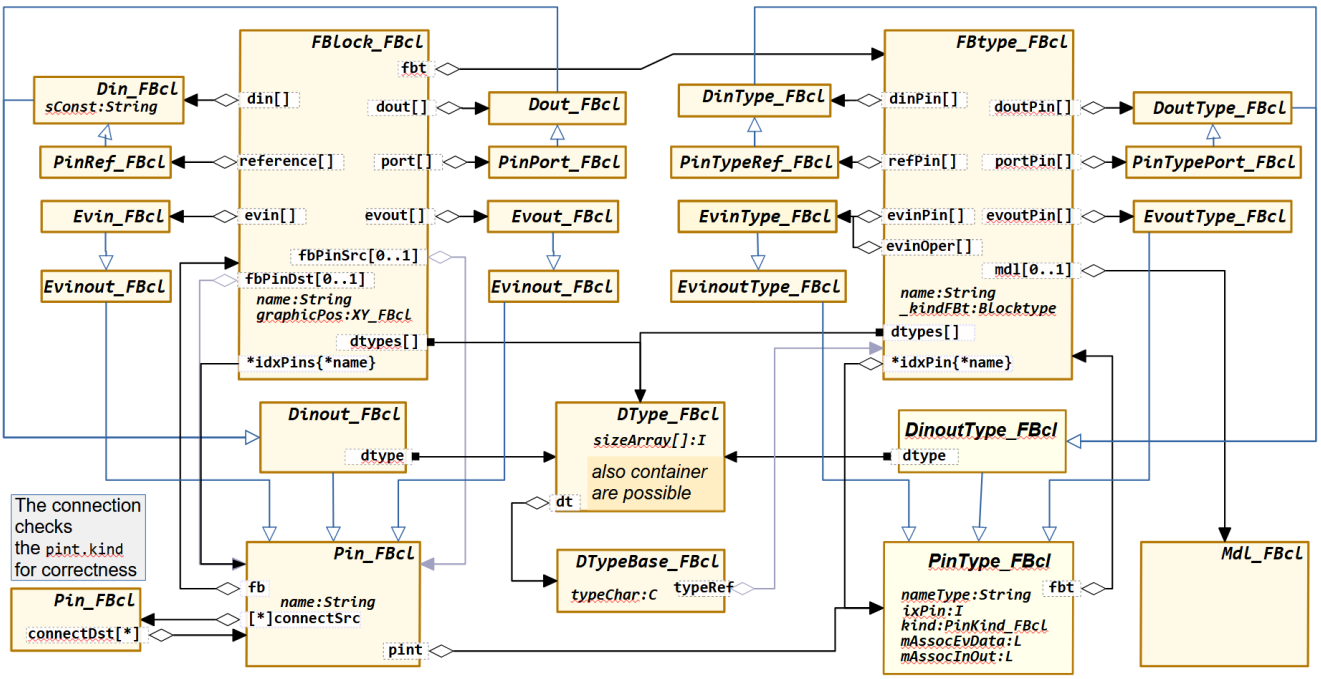

Figure 1: FBcl/FBlock_FBlockType_Pin_omd.png Overview class diagramm FBlock FBtype, Pin_FBcl

The diagram Figure 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 org.vishia.fbcl.fblock.FBlock_FBcl (www) because of its interconnections.

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

7.1.1 FBtype_FBcl

The class .../fblock/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, OFB diagram).

Specific `FBlock_Type_FBcl` on user level can be defined graphically with OFB graphic. 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 ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Module_FBcl.html (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.html (www)
- `portPin`: PinTypePort_FBcl.html (www)

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 `<:i:name.>__REF` whereby `<:i:name.>` is the name of the `FBtype_FBcl`.

The constructor FBtype_FBcl(kind, name, module, OFB-project) (www) gets only the name and the module where the FBtype_FBcl is member on. The module is adequate the package in UML. The kind is only used the distinguish between some basically functionality. The OFB-project is used only for creation a proper DType_FBcl as data type for the instance itself, stored in FBtype_FBcl#dtypeTHIS (www). It is intrinsically not necessary.

7.1.2 FBlock_FBcl

./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/FBlock_FBcl.html (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.

7.1.3 Pin_FBcl and PinType_FBcl

The pins of a FBlock_FBcl(http://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/FBlock_FBcl.html) are based on Pin_FBcl(http://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/Pin_FBcl.html) with the specificatio ns:

- `din`: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Din_FBcl.html (www):
- `dout`: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Dout_FBcl.html (=>www)
- `evin`: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Evin_FBcl.html (https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/Evin_FBcl.html)
- `evout`: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Evout_FBcl.html https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/Evout_FBcl.html)
- `reference`: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinRef_FBcl.html (https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinRef_FBcl.html)
- `port`: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinPort_FBcl.html (https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinPort_FBcl.html)

7.1.3.1 PinType_FBcl

The Pin_FBcl contains the connection to other pins to other FBlocks whereas the referenced PinType_FBcl(www) contains common information to any pins of instances. It is the base / super class for all pin types. It contains:

- `fbt`: The FBtype where the pin is member of.
- `namePin`: https://vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/DinType_FBcl.htmltring: 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`: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinKind_FBcl.html (https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinKind_FBcl.html): 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 *Error: Reference source not found* . 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 fblock/DinoutType_FBcl (www) contains also a data type information for the `dinPin` and `doutPin` as well as also for `refPin` and `portPin`:, see *Error: Reference source not found*
- The EvinoutType_FBcl(www) contains te association between prepare and update event as number assocEvPrepUpd related to the ixPin, see *Error: Reference source not found*
- The ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/EvinoutType_FBcl.html(https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/DinoutType_FBcl.html) contains also references to ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/EccAction_FBcl.html(https://vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/DinoutType_FBcl.html) for immediately execution of actions to events, see also
- The ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinTypeRef_FBcl.html(https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/PinTypeRef_FBcl.html) refers with `FBtype_FBcl fbRef` the type of the reference.

7.1.3.2 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

PinType_FBcl#mAssociatedEvData www.

7.1.3.3 Associaton between Input and Output pins

This should be contained in EccAction_FBcl

7.1.3.4 Association between prepare and update events

The element EvinoutType_FBcl#assocEvPrepUpd (www). contains the index of the prepare event in a given update event.

7.1.3.5 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.

7.1.3.6 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 dedica ted.

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.

7.1.4 Write instances for FBlock_FBcl, FBtype_Fbcl, Module_FBcl

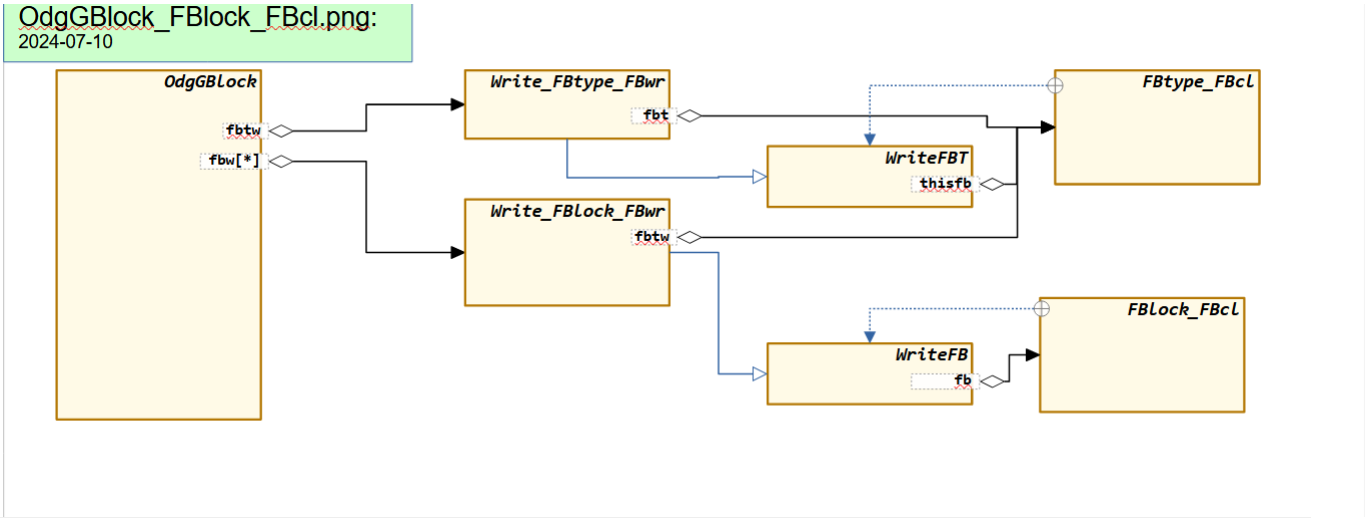

Figure 0: FBcl/OdgGBlock_FBlock_FBcl_omd.png Overview class diagramm form Graphic Block to FBlock_FBcl

This image shows the writing access to `FBlock_FBcl` or `FBtype_FBcl` via Writing wrappers, here from an Graphic blocks (GBlock) as read from the odg-graphic.

A box in the graphic presents a so named GBlock (graphic block), which is one of the graphic presentation of a `FBlock_FBcl` or a `FBtype_FBcl` . But a `FBlock_FBcl` or `FBtype_FBcl` can be presented with more as one GBlock. That's why it is important to associated the `OdgGBlock` in an early step to the appropriate `FBlock_FBcl` and its `FBtype_FBcl` or only to its `FBtype_FBcl` if a class (type) is presented. But this FBcl blocks are accessed via the org.vishia.fbcl.fblockwr.Write_FBlock_FBwr (www) .and Write_FBtype_FBwr (www). This is a pattern to prevent writing (creating) operations in the `FBlock_FBcl` and `FBtype_FBcl` class, which is usual used in hence normally offered only for read only usage. The write operations with access also to private data of `FBlock_FBcl` and `FBtype_FBcl` is done with an inner class in `FBlock_FBcl` and `FBtype_FBcl` WriteFB and WriteFBT, which can also access private elements. This inner class is now the super class of the `Write_FBlock_FBwr` and `Write_FBtype_FBwr` . Hence the inner changing operations for `FBlock_FBcl` and `FBtype_FBcl` are reached via `protected` access. With this pattern no public changing operations are able to reach outside of this wrapping classes `Write_FBlock_FBwr` and `Write_FBtype_FBwr`.

But after all XML data are gathered, on end of gatherGraphic(xOdg) (www), neither the `Write_FBlock_FBwr` and `Write_FBtype_FBwr` nor the `FBlock_FBcl` and `FBtype_FBcl` are filled with data. The data are only contained in `OdgModule` and its `OdgGBlock` and `OdgGPin` instances due to the graphic.

7.1.5 FBexpr_FBcl: FBlock for expressions, presentation in FBlock_FBcl

Figure 1: FBcl/FBexpr_omd.png class diagramm presentation of FBexpr with FBlock

An FBexpr_FBcl (www) is generally also a FBlock. The difference is: The Code generation does not base on a background functionality. The `FBtype_FBcl` also referenced from an `FBexpr_FBcl` has never a model referred (the aggregation `FBtype_FBcl#mdl` is `null`) nor a able to found existing target language code. The code generation is done exclusively from the information given in the graphic by this both objects and their pin objects.

7.1.6 Module with FBlocks

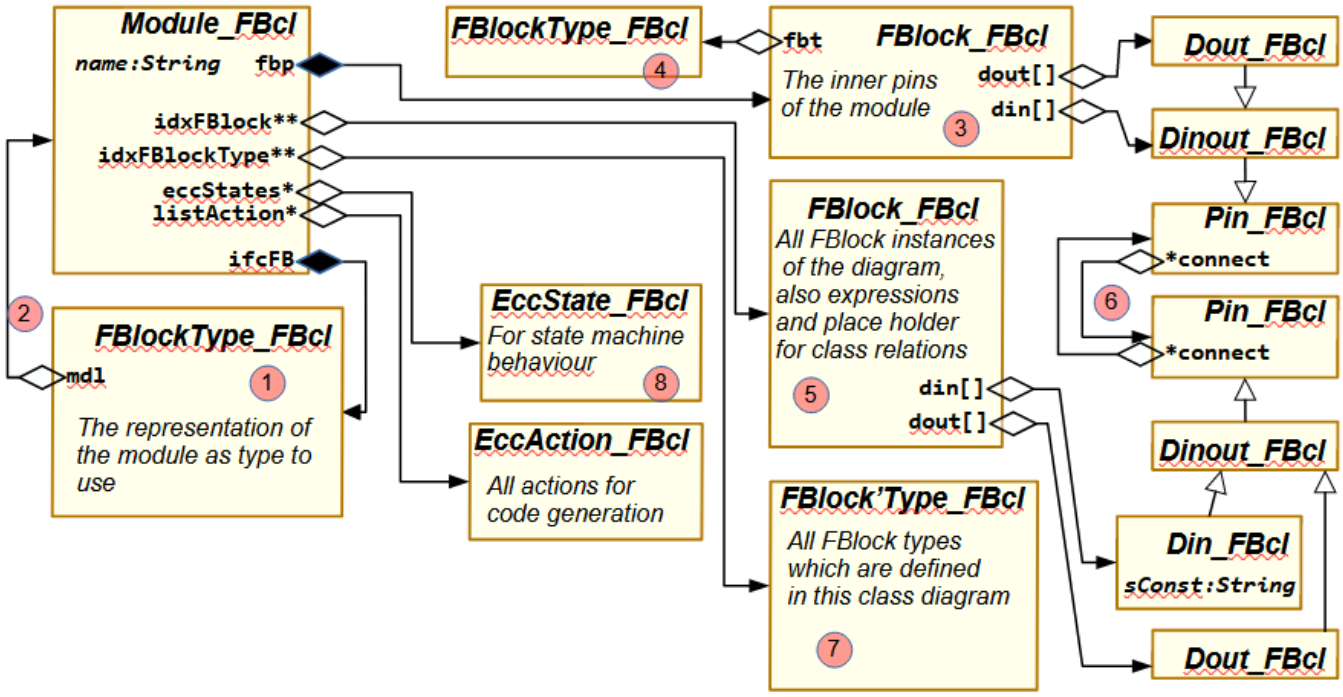

Figure 2: FBcl/Module_FBcl.png: Module and its inner FBlocks

Some pages in Libre/OpenOffice-draw (or from another input file) builds a ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Module_FBcl.html (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 th e ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/FBlockType_FBcl.html(https://vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/FBlockType_FBcl.html) 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

7.1.7 DType_FBcl and DTypeBase_FBcl

7.1.7.1 Using DType_FBcl

Instances of ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/DataType_FBcl.html (www) are referenced from data pins, see chapter *<:@ref:#173:8.1.3 Pin_FBcl and PinType_FBcl.>*. They contains

- `dt`: The reference to the basic data type: ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/DataTypeBase_FBcl.html(http://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/DataTypeBase_FBcl.html)
- `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 ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Module_FBcl.html (www) and also inside ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/FBlock_FBcl.html (www) and ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/FBlockType_FBcl.html (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 integ er.

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.

7.1.7.2 Using DTypeBase_FBcl

All types in fblock/DataTypeBase_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`.

7.1.8 Event tree node

The event connections in a module builds firstly a tree of event nodes. But this tree is also characteristics by some cross event connections.

For code generation the tree of event nodes is determining the order of execution. Primary the order of execution is only the event flow. But this is too detailed, not able to overview. Also the algorithm of roll out the event flow is not very obviously.

That's why after build all event connections a tree of event node is built. This tree structure has an `Iterator` which offers the member of this event tree node in a defined order which is used also for code generation as also in the event connection part of the fbd file (IEC61499). But for the cross connections in the event flow there are some special cases.

Look on the following example:

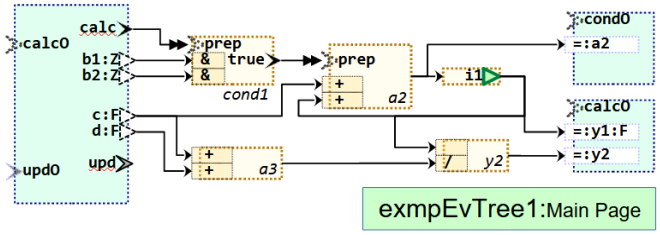

Figure 3: odg/exmpEvTree1.png

The image shows a simple expression relation, but with a boolean event expression: The FBexpr `a2` and follow does only act if the expression evaluates to true. But this is not primary important for the following considerations, only for code generation.

The shown data connections and the drawn event connection results in the following event flow, see *7.6.2 Identification of the event flow due to data flow* page *41*:

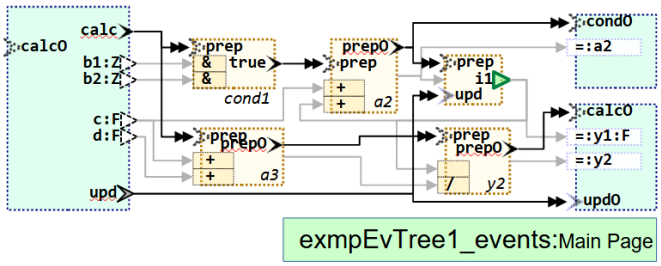

Figure 4: odg/exmpEvTree1_events.png

The data connections are shown in gray.

The fbd output file (IEC61499) shows the following event connections:

```
EVENT_CONNECTIONS
calc TO a3.prep
  a3.prepO TO y2.prep
    y2.prepO TO calcO
calc TO cond1.prep
  cond1.true TO a2.prep
    a2.prepO TO condO
    a2.prepO TO i1_X.prep
      i1_X.prepO TO i1.prep
upd TO i1.upd
upd TO updO
END_CONNECTIONS
```

The indentation allows detect the tree structure of the event connection. Sorting of connections are always alphabetically, hence first `a3.prep` is shown. The event connection chain continues with `y2.prep` in this line till `calcO`. The second connection from `calc` goes to `cond1.prep`, continues with its true event output, forces `condO`, and also the preparation of the state variable i1, which is updated with `i1.upd`.

This is also exact the order of generated code.

How the event connections are sorted? Using an fblock.EvoutTreeNode_FBcl (www):

The next image right side shows the complete instance view of the OFB module in the images *Figure 3: odg/exmpEvTree1.png* and *Figure 4: odg/exmpEvTree1_events.png*.

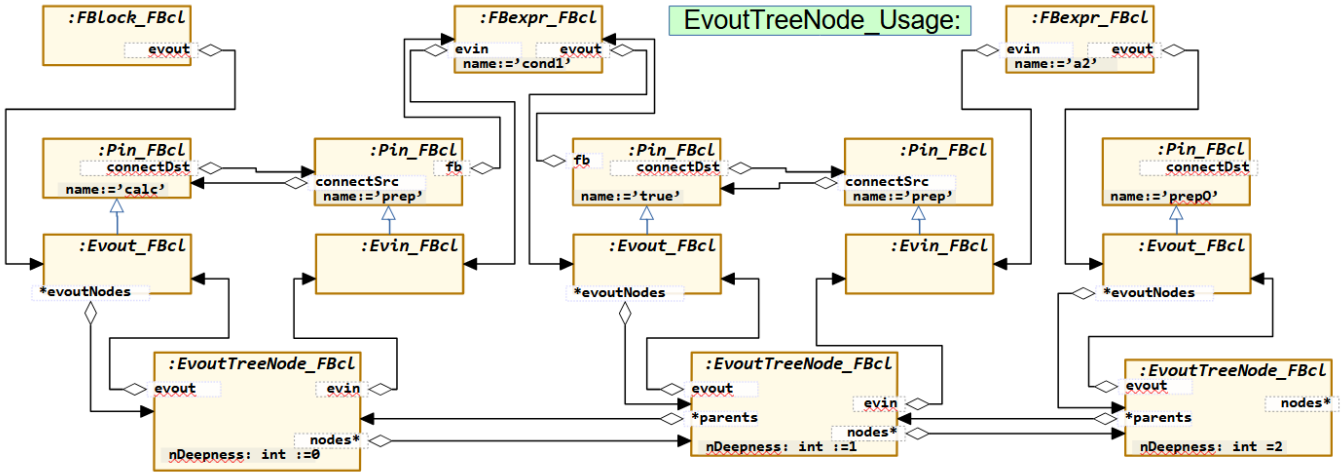

Figure 5: FBcl/EvoutTreeNode_Usage.png

The image shows as class/instance diagram the first two event connections of *Figure 3: odg/exmpEvTree1.png* from `calc` to the FBexpr `cond1` and then from its output `true` to the next `a2.prep`.

The information for connecting events are repeated here. The connection is a property of the pins already, and twice given by designation of the `evout` - `evin` pair in the fblock.EvoutTreeNode_FBcl (www). Also the relation between `evin` and `evout` of the same `FBlock_FBcl` can be traced by the operation fblock.Evin_FBcl#iterCorrespondEvout(...) (www). Just this is done to built this event tree.

The intrinsic information is just the order of event connections which is also the order of execution in the code generation, see *7.7.2 Tracking the event chain for a module's operation* page *54*. This gets important also for Join of events:

7.2 Reading graphic files from different inputs, UFBglConv

The org.vishia.fbcl.UFBglConv (www) is the main class to convert several input files, and outputs IEC61499 files and code generation for the target system.

This class has a `main(String[] args)` entry to invoke with java command line. Additional

- UFBglConv#smain(...) (www) and
- UFBglConv#amain(CmdArgs...) (www)

are given to call it from a Java application inside, and to call it from a JztxtCmd script.

After parsing the commands `execute(...)` is called. This operation does the following:

- First, all input files in CmdArgs#listFileIn from the cmdline arg `-i:path/to/input.file` were processed, the type (extension) of the input file determines the reading conversion. odg-files are read, see chapter .
- The data of this first step are stored in the Write… instances of TODO
- all parsed modules are stored in `Prj_FBCLrd#idxWriteModules`.

Whereby the type of the conv

7.2.1 Complete a module

Completion of a module is done whenever the module is read completely, before the next module is read. The operation `Write_Module_Fbwr#completeMdl()`. does the following actions:

- `completeMissingEventPinsInTypes()`:
- `postPrepareFBlocks()`
- `postPrepareDtypesToIfc`
- `completeFBoper()` for all `listFBoper`
- `propgDtypes(…)`: This is the essential operation to equalize the used Dtype_Fbcl instances via all connections and depending pins. Non full determined data types in a FBtype will be determined for the FBlock instances due its connections.
- `Dataflow2Eventchain_Fbrd#prc()`: This is the essential operation to create the correct execution order of all internal operations of the module, described by the event flow. The event flow follows the data flow. See chapter TODO
- `completeAllFBlockTypePinsInModule()`: All FBtype should have the pins, adequate read library modules.
- `completeAllMdlPins(null)`: establishes the pins for the FBtype of the module
- `createAllMdlIfcPins()`: The `FBtype_FBcl` instance for the module's interface is the mirror of the internal `Module_FBcl#fbp` pin - `FBlock` and its associated `FBtype_FBcl`. The last ones are created with gathering the pins of the module in the module's implementation graphic. The pins are mirrored because for example an evin of the module seen from outside is used internally in the module as evout as source for the internal wiring.

  This operation builds the pins for the module's interface `FBtype_FBcl` instance from the given internal pins of the module. For an existing `EvoutType_FBcl` as input of the module or output for internal using a `EvinType_FBcl` with the same name is created, etc.
- `setExprOperatorToPins()`;
- `adjustFBexrFnDtypes()`:

7.3 Read data from LibreOffice odg files

Table of Contents

7.3.1 The file format of odg – content.xml

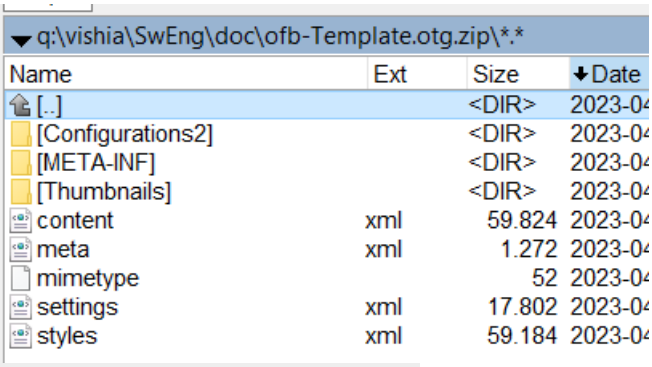

Figure 6: OFB/ContentOfodg.zip.png

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.

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.

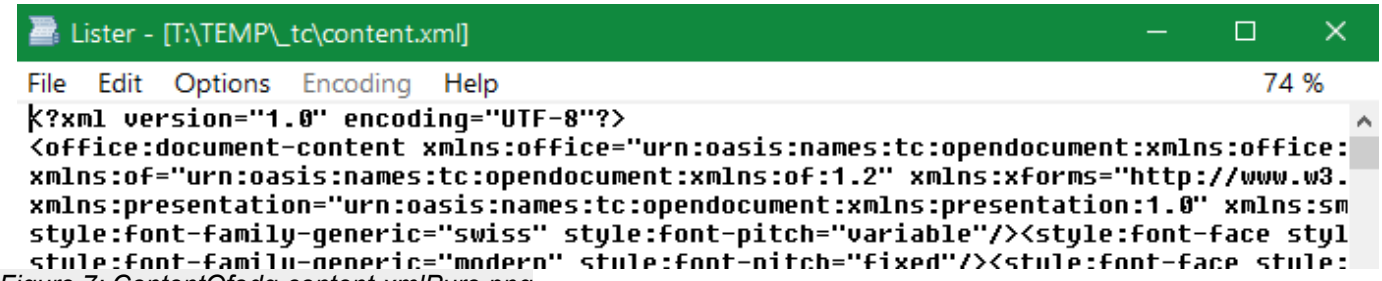

Figure 7: ContentOfodg-content-xmlPure.png

But there is an option in LibreOffice: Menu *"Tools -> Options"*, Select *"LibreOffice -> Advanced"*, press Button [*Open Expert Configuration*], then search *"PrettyPrinting"* or select in the configuration tree *"org.openoffice.Office.Common -> Save -> Document"*, select the property *"PrettyPrinting"* and set it to *true*. See also https://wiki.documentfoundation.org/Documentation/ODF_Markup/en#Pretty_Printing

Then the content.xml will be stored with some more lines. It is able to overview. But regard, a newline inside a `<text:p>` may create at least one space. Hence the `<text:p>` till `</text:p>` is still written in a long line (but not too long).

After beautification it looks like (as textual snippet):

```
<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 page . 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.

7.3.2 Read content.xml from the odg graphic file to internal data

The org.vishia.xmlReader.XmlJzReader (www) contained in the `vishiaBase.jar` is used to read the XML data. This class can select determined parts from the XML file, read not all. Therefore a configuration file is used. The data are stored in a common or a specific data class. A possible common class is org.vishia.xmlReader.XmlDataNode (www). But this class is not used here (it is used for the LibreOffice to VML converter).

Instead a specific destination class for the XML data was created: org.vishia.fbcl.readOdg.xml.XmlForOdg (www). The given instance of this class is filled from the `XmlJzReader` by reading the `content.xml`. The data are organized proper to the internal XML tree structure. Reading the `content.xml` is controlled by the config file fbcl/readOdg/xml/odgxmlcfg.xml (www) as part of the jar file `vishiaUFBgl.jar`.

presents the access to the read XML data. This class was automatically created by calling the tool suite on .../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 fbcl.readOdg.xml.odgxmlcfg.xml (www) which controls interpreting of the XML data. The class to read the XML file is org.vishia.xmlReader.XmlJzReader (www) in the `vishiaBase.jar` file. It is called in fbcl.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) { ...
  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$Office_d
  office_automatic_styles: XmlForOdg$Office_
  office_body: XmlForOdg$Office_body TODO to
   office_drawing: XmlForOdg$Office_drawing
    draw_page: List<Draw_page> [TODO toStrin
     [0]: Object TODO toString XmlForOdg$Dra
      draw_connector: List<Draw_connector> [
      draw_custom_shape: List<Draw_custom_sh
      draw_frame: List<Draw_frame> null null
      draw_g: List<Draw_g> [(9.5cm, 4.1cm) +
      draw_master_page_name: String Default
      draw_name: String page1 String 287 165
      draw_polygon: List<Draw_polygon> [4.2c
      draw_polyline: <unknown type> null nul
      draw_style_name: String dp1 String 289
     [1]: Object TODO toString XmlForOdg$Dra
     [2]: Object TODO toString XmlForOdg$Dra
     [3]: Object TODO toString XmlForOdg$Dra
  office_font_face_decls: XmlForOdg$Office_f
  office_scripts: String null null 16552
  office_version: String null null 16552
  office_version: String 1.3 String 264 1655
```

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.

7.3.3 Sorting XML data to Shapes for each page

After reading the XML file the operation OdgReader#gatherGraphic(xOdg) (www) is called. Any word ***This evaluates*** the `whole XML content`, but sorted to pages, because in XML the data are sorted also in page nodes:

7.3.3.1 Gather Pages and the title

```
<office:body>
    <office:drawing>
      <draw:page draw:name="page1" ...
        <draw:custom-shape ...
```

For each page, a `new OdgPage(nr, xpage, xOdg)` is created and first, `page.gatherTitleDisabledArea(xpage, xOdg);` is called. The title is the box with `ofbTitle`. It contains the module name for this page and hence it is essential to associate the page to the module. If the module name is commented with leading `#`, then this page is commented and hence skipped. The module name is essential. Maybe only determined modules are read from XML, controlled by the command line option `-im:ModuleName`. Pages for each one module should be one after another. If another module starts with a new page, the module is OdgReader#completeModule (www)

The disabled areas in the graphic are important to prevent evaluation of disabled parts.

7.3.3.2 Gather all shapes per page

If the page is not skipped, then first all shapes are gathered and sorted by its style information, with its texts and coordinates. This is a formally operation, done by OdgPage#gatherShapes(xpage, xOdg) (www) It evaluates all `<draw:custom-shape…`, `<draw:polygon…` and `<draw:frame`... from the page, and also `<draw:g` to evaluate this elements in a group.

From all this elements first the x-y-coordinates are detected. If they are in a range of a disabled area (which is a shape with `ofbDisabledArea` style), then the shape is ignored. This is the functionality described in html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.2.1 GBlock styles, ofb* page *4* and html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.5.1 Module in file organized in pages* page *16*.

Then all other properties of the shape are gathered from XML and stored in instances of OdgShape (www). Any of a shape in the graphic is formally mapped to an `OdgShape` instance. This can be a pin, a fblock etc. The `OdgShape` instances are stored sorted to its kind in some lists inside OdgPage (www).

For the shapes the draw style is evaluated (in XML the attribute `draw:style-name="..."` in the shape). Via the OdgReader#idxFBkindFromStyle (www) and also OdgReader#idxPinkindFromStyle (www) the textual given style name is transferred to the internal used `enum`.

7.3.3.3 Evaluate the shapes

The approach is, reading all shapes and associating to the internal data 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 possibility 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).

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`.

After gathering and sorting the shapes there are evaluated. First this is done for the non-pin shapes. Especially first the GBlock shapes with style `ofbFBlock`, `ofbClass`, and then `ofnNameTypeFBlock` and `ofnClassTypeName` (deprecated). After this operation you have instances of OdgGBlock which referes to FBlock_FBcl and FBtype_FBcl in your Write_Module_FBwr. This, first the OdgGBlock are necessary to associate the pins.

The pin shapes which are all stored in OdgPage#shPin are evaluated after detecting alls FBlock shapes. So their OdgGBlock is already stored with its coordinates able to find in ReadOdg.listBlocks_x and _y. the operation ReadOdg#searchFbg(...) searches the associated GBlock by coordinates to each evaluated pin. The pin- FBlock association is done with the pin graphic coordinates. At least one rectangle corner point should be inside the rectangle box of the GBlock.

Pins which have not a proper GBlock found can be either a free variable in the module, as `ofpVout...` or `ofpZout...`, or also as simple `ofPin...` with a text designation `...=$` or `...=&` on end. Or an error messages is shown: *"Warning ...Pin @...position outside FBlock shape"*..

- `ofbAccess` as style of the found GBlock forces adding the pin with OdgReader#gatherAccessPin(...).
- `ofbExpression` as style of the found GBlock forces adding via OdgReader#gatherExprPin(...)
- All other pins are added via OdgReader#gatherFBlockPin(...) to the found OdgGBlock.
- But pins of style `ofpExprPart` are sorted as extra shape kind in OdgShape#shExprPart. They are also added to the found GBlock with OdgReader#gatherExprPin(...) without selection of the GBlock kind, but with test of the GBlock kind.
- Pins of style `ofpExprOut` are sorted as extra shape kind in OdgShape#shExprOut. They can be applied either to a GBlock if style ofbExpression or also ofbAccess as their output. Hence, the kind of the found GBlock is tested, and either ... here is a little bit TODO cleanup in sw.

7.3.3.4 Evaluating Pin texts

The text inside a pin shape is evaluated on creation of the OdgGPin instance while setting the element OdgGPin#nameType. Then all information from the graphic are contained in a prepared form in the OdgGpin instance, ready to use to later create the adequate Pin_FBcl instances with the correct derived types.

The Evaluation of the pin texts is done in the constructor of the type OdgNameTypeArray, which contains all semantic relevant information given in the text of the graphic shape for OFB usage. The syntax and meaning es explained in html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.3 Texts in graphic blocks and pins* page *8*.

TODO explain short meaning of the info.And how it is parsed.

7.3.4 Gather data for OdgModule page by page

The shapes or boxes with style `ofbFBlock` or also `ofbClass` are contained in the current `odgPage.`shFBlock (www). This container is evaluated before the inner shapes of a GBlock as for example `odgPage.shPin` are evaluated. First a OdgGBlock is created, and then via calling OdgReader.html#assignFBlockName TypeIdCreateFBlock(...) (www) the appropriate `FBlock_FBcl` or `FBtype_FBcl` is

searched in the project or it is created. But it is referenced via the Writer classes, see *7.1.4 Write instances for FBlock_FBcl, FBtype_Fbcl, Module_FBcl*

7.3.4.1 Associate the page to a module

After the Shapes are built for each one page inside the operation OdgReader#gatherGraphic(xOdg) (www), and after first the `ofbTitle` is scanned, the page is associated to the module, which name is contained in the `ofbTitle` box.

The module name is searched in OdgReader#idxOdgMdl (www) `idxOdgMdl`. It is found or new created and stored there as OdgModule (www) `OdgModule` instance. Each page completes the module internal data.

The next image should give an overview over that data:

7.3.4.2 Aggregation to FBcl blocks via Writer

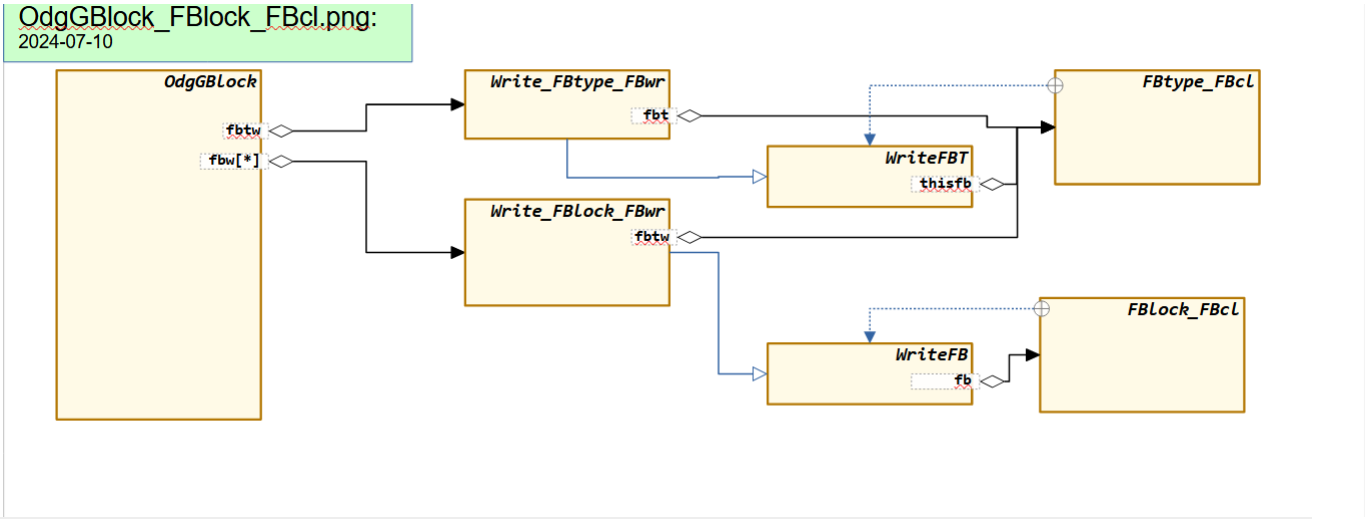

Figure 8: FBcl/OdgGBlock_FBlock_FBcl_omd.png Overview class diagramm form Graphic Block to FBlock_FBcl

This image shows the existing Graphic blocks (GBlock) as read from the odg-graphic and its relationships to the FBcl classes. The FBcl classes are created here, but not completed.

A box in the graphic presents a so named GBlock (graphic block), which is one of the graphic presentation of a `FBlock_FBcl` or a `FBtype_FBcl` . But a `FBlock_FBcl` or `FBtype_FBcl` can be presented with more as one GBlock. That's why it is important to associated the `OdgGBlock` in an early step to the appropriate `FBlock_FBcl` and its `FBtype_FBcl` or only to its `FBtype_FBcl` if a class (type) is presented. But this FBcl blocks are accessed via the org.vishia.fbcl.fblockwr.Write_FBlock_FBwr (www) .and Write_FBtype_FBwr (www). This is a pattern to prevent writing (creating) operations in the `FBlock_FBcl` and `FBtype_FBcl` class, which is usual used in hence normally offered only for read only usage. The write operations with access also to private data of `FBlock_FBcl` and `FBtype_FBcl` is done with an inner class in `FBlock_FBcl` and `FBtype_FBcl` WriteFB and WriteFBT, which can also access private elements. This inner class is now the super class of the `Write_FBlock_FBwr` and `Write_FBtype_FBwr` . Hence the inner changing operations for `FBlock_FBcl` and `FBtype_FBcl` are reached via `protected` access. With this pattern no public changing operations are able to reach outside of this wrapping classes `Write_FBlock_FBwr` and `Write_FBtype_FBwr`.

But after all XML data are gathered, on end of gatherGraphic(xOdg) (www), neither the `Write_FBlock_FBwr` and `Write_FBtype_FBwr` nor the `FBlock_FBcl` and `FBtype_FBcl` are filled with data. The data are only contained in `OdgModule` and its `OdgGBlock` and `OdgGPin` instances due to the graphic.

(empty page)

7.3.5 Build the data in FBcl data

The data in `FBlock_FBcl` and `FBtype_FBcl` as well as also in `Pin_FBcl` and `PinType_FBcl` and its derived classed and also in `Module_FBcl` are designated as FBcl data. FBcl is the *Function Block connection language* which is independent of the Libre/OpenOffice odg graphic and the base for the IEC61499 presentation and also for code generation.

The Module_FBcl and also FBlock_FBcl, FBtype_FBcl are created on creation of the Graphic representations OdgGBlock and OdgGPin. But they are not complelely filled.

The completion of the FBcl data from the graphic representation is done in readOdg.OdgModule#buildFbgData(...) (www). This does the following:

- create all FBlock pins: OdgModule#createFBPins_duetoGBlockPins() (www).
- createFBPins_duetoGBlockPins()
- createFBclConnectionsFromGraphic()
- buildAssociatedMdlEventDataPins()
- buildAssociatedFBtypeEventDataPins()
- Write_Module_FBwr#completeMdl()

empty

7.3.6 Connect all FBcl pins due to connection of graphic pins

The graphic connections are done page per page of the odg file after reading the shapes, see *7.3.4 Gather data for OdgModule page by page* page *22*. The graphic connections are not the same as the real FBcl connections:

- The graphic connections are oriented general to one graphic page. But the graphic of one module can use more as one page. Connections over pages are done by
    - Xref Cross references
    - Using free variables
    - Use the same FBlock instance (or Type) more as one time with another Graphic GBlock.

It means, after gathering the graphic, on finishing reading one module, the graphic connections should be translated to FBcl connections in the module without the graphic page orientation.

That is done for all graphic pins

- (readOdg.OdgGPin) in all graphic
- readOdg.OdgGBlock of the current
- readOdg.OdgModule.

For that the operation OdgModule#createFBcl ConnectionsFromGraphic(...) (www) is called. This gets GBlocks from:

- OdgModule#idxModuleIfc (www): All module pins
- OdgModule#idxGBlock (www): All GBlocks of the module
- OdgModule#idxGExprByName (www): All expression pins
- OdgModule#idxGAccessByName (www): All Access FBlock pins

From the graphic connection first a textual connection is searched. That is the denomination of `@fb@pin...=:` in the pin description. The named FBlock_FBcl and its pin is searched and connected.

Furthermore, all outgoing connections are tracked. This is not bound to output pins, all pins are evaluated. Because an outgoing connection can also start from an input pin, which is connected to any output.

For tracking output the operation OdgModule# processFBlockPinGraphicConnection(…) (www) is called. It calls after a short preparation (create necessary pins which are only given in the type) the operation OdgModule#connectViaDemuxXref(...) (www). This operaton is recursively called inside if the connection has more paths.

For the graphic connection it should be regarded:

- Using Xref: Then the graphic connection goes to any Xref GBlock. There are more as one Xref GBlocks with the same Label, especially on several pages. All this GBlocks with the same label references to only one readOdg.OdgXrefGraphic which is derived from readOdg.OdgGBllock. It means any Xref graphic appereance does not have its own OdgGBlock, it has a common one valid for the whole OdgModule. It is referenced sorted by name in the container odgReader.OdgModule#idxXrefGrpahic ByName (www). But for the connection this is not important. Important is the fact. That all incomming connections (usual one, onle one source) is contained in the OdgGBlock#fbPinDst, and all outgoing connections are staring from OdgGBlock#fbPinSrc. With that, a connection via Xref can simple tracked by regarding all outgoing connections if the incoming connection hits the Xref.

- Using Demux FBlocks. They have the graphic style `OfbDemux`, respectively the pins have `ofpDemux`.

The Demux FBlocks are a little bit more complicated, hence explained following:

- If a graphic connection from any source pin ends on a `odpDemux` pin, then this Demux pin contains a selection String, named in the software as `pinDemuxSel`. Writing with or without […] does not play a role, the string inside the `[...]` is used or the given string without leading and trailing spaces. Then the connection is continued with the mulitplex pin, which is in OdgGBlock#fbPinSrc of this Demux Gblock. But the selection String named `pinDemuxSel` in the Java source is transported as argument of OdgModule#connectViaDemuxXref(...) (www)

*\t

7.3.7 Preparation of Expressions from odg

See also the data description of FBexpr in chapter *7.1.5 FBexpr_FBcl: FBlock for expressions, presentation in FBlock_FBcl* page *10*

The internal Handling of expressions needs a little bit explanation. Refer to chapter html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.7 Expressions inside the data flow* page to see the capabilities of expressions.

7.3.7.1 createExprPins(...) `createExprPins(…)`

In `createExprPin(…)` (www?) all pins from a OdgFBlockGraphicInstance 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 …

First the kind of all pins is checked, to check which FBtype_FBcl of the FBexpr should be taken. The kind depends

- how...

An internal String array `sExpr[3]` accumulates the information about the operators of all pins. It is created empty first. The content contains elements, separated by comma `,` which are stored in this form as constant value for the `expr` input of the FBexpr seen in the FBcl file (IEC61499).

- The `sExpr[0]` holds the operators of the Din pins.
- Whereby the first element contains in two characters the access type of the FBexpr and the basic operation. The access type is `~` `&` `=` for an inline expression (without data output), as an inline expression but with one or some outputs beside (possible only if an textual given operation is given) and an expression which as a variable on its output, hence called as statement to set it. This last variant can also have some outputs beside (more outputs, all have variables associated). The same is for `@` `%` `$`, in this order, but with the property that the expression has a THIS aggregation to its FBlock, it is a FBoper, see html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.7.9 FBoper, operation for a FBlock* page *46*
- The basic operation on `sExpr[0].charAt(1)` is `+` `*` `&` `v` `^` `=` `h` for the operation types ADD, MULT, AND, OR, XOR, CMP and SHIFT. For all Din pins the operator should be given. The graphic may omit the operator if it is the default operator for the operation. But here it needs to be specified anyway.
- Also an unary operator is written here, at end of the element. Unary operator are only `-` `~` `/`, last for reciprocal.
- The `sExpr[1]` holds the operator for the operators of the K pin. The first element is empty, the `sExpr[1]` starts with `",..."`. Not existing K pins are presented with an empty element. It means `sExpr[1]=",,*"` if only the K2 exists. The K1 position is empty. The K operator should be given anyway, also it is omitted in graphic for the default, often `/`.

xxxx

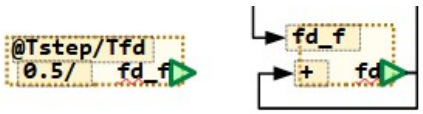

Figure 9: odg/ExprExmp2Vars.png

For this image the `sExpr[0]` contains `"=*,*,/"` because first char `=` means, it is an assignment to a variable. The next `*` determines it as a MULT expression. Then the input operators follow with `,*,/"` for `*` and `/`. The `sExpr[1]` contains `",/,"` because the K1 is used with division, and the K2 is not existing.

For all pins `createExprPinAndKpin` is called. This completes the `sExpr[]` succesive for each pin.

At least, if all pins are created, the String for `sExpr[0]` is adjusted. All din pins should have its correct operator, But in the graphic the default operator can be omitted. It means it should be completed. Note that on code generation, the operator on the first pin may be remove again in an expression, for example +a +b is translated to (a+b). This is done in code generation. Here the operators are completed. For all CMP operation `=` is written as default, not `==`, for the Din pin which is compared to. `=` is the default operator for the input to compare. Note that in opposite, in IEC61499 as also in IEC61131 the compare operator for 'equal' is `=`, and not `==` as in C/++, Java and other programming languages. Whereas for an assignment `:=` is used instead `=` in current familiar languages. That was the writing style also in the first structured language Algol, as also in Pascal, which were present in the time of foundation of the IEC61131 automation programming language (1980th. In opposite, the founder for the C language have introduced the shorter `=` for the assignment, but then used `==` for the equal operator. This contradiction still reaches us here today.

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.

The syntax of an ofpExpPart ist evaluated in OdgModule#checkPrepareExprOperator(...) (www)

7.3.7.2 createExprPinAndKpin

In `createExprPinAndKpin(…)` for one gpin OdgGpin (www?) the appropriate . 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 …

(emptyline)

Tip for debug: first lines there, the complete text from graphic can be tested, to set a debug.stop

7.4 Read data from Simulink

TODO

7.5 Read data from IEC61499 text files (fbd)

TODO

7.6 Complete preparation of the module

Table of Contents

7.6.1 Forward and backward propagation of data types

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

7.6.1.1 How data types are stored in Java FBcl data

First lets clarify how the data types to pins are stored:

Right side there is an *Figure 10: OFB/Pin-DTypeFix.png*. It shows on `(1)` a FBlock with two pins in an OFB graphic. Below it is presented how this graphic is mapped to internal Java data beginning on the `ofcDependency` connections.

`(2)` The FBlock in the OFB graphic is presented by an `FBlock_FBcl` instance in the Java data. The pins are referred from this `FBlock_FBcl` instance via Din and Dout as instance of `Din_FBcl` and `Dout_FBcl` which have its common super class `Dinout_FBcl` and then the super class for all pins `Pin_FBcl`, which contains the pin name.

- This super class relation is shortened for the `Dout_FBcl`, here the `Pin_FBcl` is not shown and the element '`name`' is drawn in the `Dinout_FBcl` instead. This is admissible in OFB diagrams, it is a better manageable kind of class diagrams. Elements which are anywhere associated to a super class can be drawn on other parts of the OFB class diagram also in the derived (inheriting) class.

`(3)` The Pin_FBcl#pint (www) refers the `PinType_FBcl`, as well as the FBlock_FBcl#fbt (www) refers the type of the FBlock in the OFB graphic. The Type of the FBlock presents the user type in the OFB diagram, here named with "`Type`".

(5) The data type of the pins are generally bounded to the DinoutType_FBcl#dType (www), data types are relevant only for Din and Dout pins, not for others. Because the DinoutType_FBcl#ixDtype (www) is `=-1`, this is the valid data type also for the pin itself via `Dinout_FBcl`.

(6) The `DType_FBcl` contains the array size, here 3 elements for X, and scalar for Y. Scalar is designated with a one dimensional array, but with number of dimensions `==0`. Whereas if `sizeArray ==null`, it is not defined.

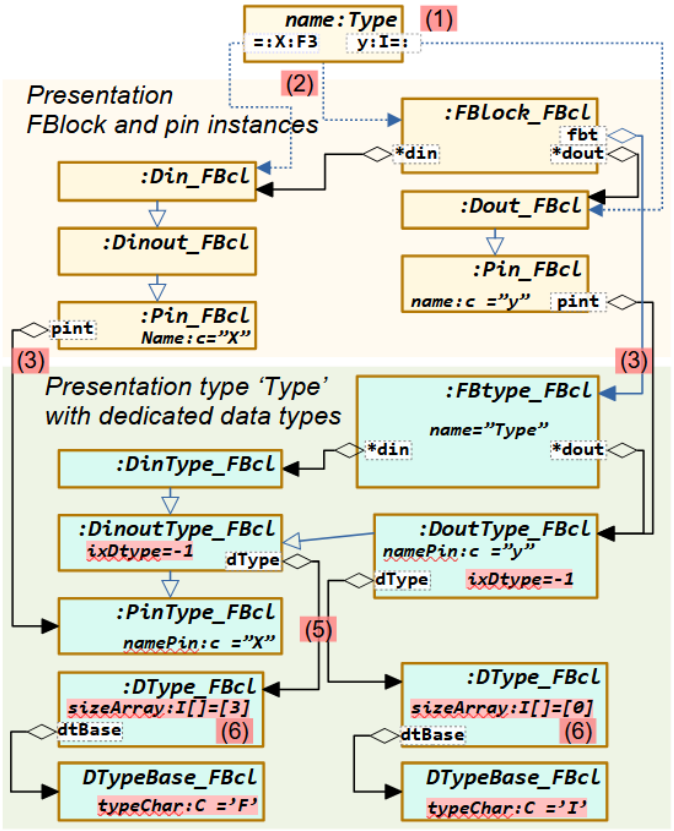

Figure 10: OFB/Pin-DTypeFix.png

General:

- If a pin has not a data type set, then `DinoutType_FBcl.dType == null`. This is a non full qualified OFB Type, which is qualified then by the data type propagation.
- If a pin has a data type set, but the data type is non full qualified, then the non full qualified `DType_FBcl` is referred from `DinoutType_FBcl#dType` as well as from the `DinoutType_FBcl#ixDtype` as index to the data type array in FBtype_Fbcl#dTypes (www). Then the OFB FBtype is deterministic, but not full qualified in data types. The real used data type in a FBlock is set in the array FBlock_Fbcl#dTypes (www), see *7.6.1.3 Forward and backward propagation of non dedicated pins*
- If a pin is full qualified in its data type, the DinoutType_FBcl#dType (www) refers the complete defined `DType_FBcl` which is often a standard type, and in lesser cases (for array pins) a `DType_FBcl` with a specific `sizeArray` which refers in DType_FBcl#dtBase (www) a standard basic data type.
- It is clarified, that the `DType_FBcl` with the same properties in a module are presented also by the same instances. If the module's pins are non full qualified, then the same mechanism is valid for the module's interface FBtype_FBcl via Module_FBcl#ifcFB (www) and the inner pins in Module_FBcl#fbp (www). So usage of the module in a FBlock in the superior using OFB diagram can qualify the pins.

7.6.1.2 Forward/backward propagation of dedicated pins

The data type propagation starts by adding all `dout` pins of all FBlocks and dedicated pins of the ***module's inputs*** which are formally an `Dout_FBcl` (to the inner of the module) which have a dedicated fbcl/fblock/DType_FBcl (www) to an internal `List<Dout_FBcl> listDout`.

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. The `dout` pins, which are yet dedicated, are added to the `List<Dout_FBcl> listDout` to use on further forward propagation

Also from begin, a `HashMap<Din_Fbcl, Din_FBcl> idxDinBackward` is filled with all outputs of the module which are `Din_FBcl`

`din` pins, which are dedicated. This `HashMap` is filled also with all found `din` pins of all FBlocks which are now propagated, because of backward propagation if necessary. This `HashMap idxDinBackward` is also used to check whether a pin reached on forward propagation is already propagated, hence by another forward propagation and the related propagation inside the FBlock. If it is so, only the data type is tested whether it is matching. The propagation stops here.

After all forward propagation are done, all remaining `Din_FBcl din` pins in the `HashMap<> idxDinBackward` are handled for a backward propagation. This `din` are either full dedicated from beginning, or they are dedicated now because of the forward propagation, and are ***not reached*** by the forward propagation (because if they are reached, they are removed from the `HashMap<> idxDinBackward`. It mains that are the remaining necessary backward propagation candidats.

The following (non animated) graphic should show this process. Note that the order is the order of the colors magenta, red, orange, green, cyan and blue. blue is the last backward propagation.

TODO image.

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.

7.6.1.3 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 non 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 following 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`

7.6.1.4 Forward declaration for depending pins of a FBtype

If pins are not full qualified then the data type of some pins depends from another. If the data type of one pin is qualified, then all depending pins can be qualified too.

There are two scenarios of qualifying the pins:

a) Some pins have the same data type. If one is qualified, the others with the same data type are automatic qualified because they refer the same `Dtype_FBcl` instance. This is typical for simple expressions, where all pins have the same type.

b) Some pins have the same, but other pins have derived data types with specific changed properties. For example one pin is scalar, and another pin is complex with the same `DtypeBase_FBcl`. Or one pin is an array, and the other pin is the element type of the array.

To manage this, the `FBtype_FBcl` contains an element FBtype_Fbcl#dTypes (www) and the instances FBlock_FBcl contains also a FBlock_Fbcl#dTypes (www). This is shown in the *Figure 10: OFB/ExprReIm2Cplx_DTypeDep.png* right side with the red aggregations `(1)`. Some pins uses exact the same `DType_FBcl` from this aggregation, via the DType_FBcl#usingPins(www) , (5) and some other pins uses another `DType_FBcl` but with the same DType_FBcl#depId(www) (6)

But first let explain the class diagram right side. This image is similar the *Figure 10: OFB/Pin-DTypeFix.png* one side before, also the marker-numbers are the same. This image shows more context, and have simplified some inheritance contexts instead:

`(1)` on top is the snippet from a OFB diagram, here an `ofbAccess` FBlock.

`(2)` are shown as dependency connection to that instances in the internal Java data which presents this FBlock in the OFB diagram. It is a `FBaccess_FBcl` extending a `FBlock_FBcl` for the FBlock itself and `Din_FBcl`, `Dout_FBcl` for the pins.

`(3)` are the references from the instance (yellow part) to the type description of this OFB user FBlock (green part). The `FBlock_FBcl` instance refers the instance pins `Din_FBcl`, `Dout_FBcl`, and the `FBtype_FBcl` contains the type information of the user FBlock with the pin types `DinType_FBcl` and `DoutType_FBcl`. Whereas the class diagram shows that the both data inputs instances `Din_FBcl` refers the same type instance `DinType_FBcl`.

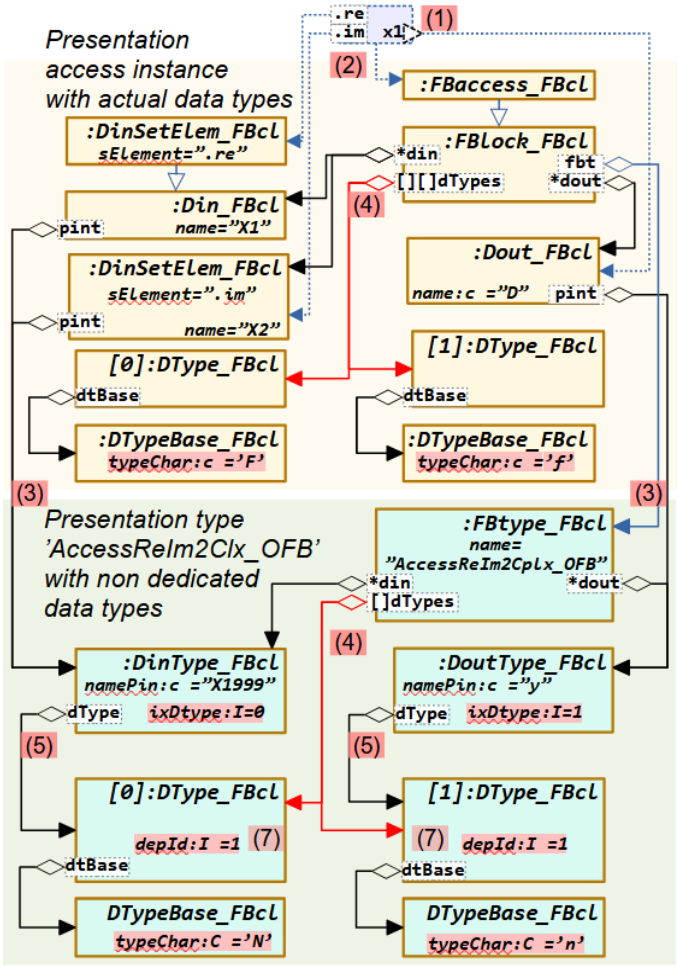

Figure 11: OFB/AccessReIm2Cplx_DTypeDep.png

`(4)` Both, FBlock_FBcl and also FBtype_FBcl have references to data types `DType_FBcl`. FBlock_Fbcl#dTypes (www), FBtype_Fbcl#dTypes (www). That are the non deterministic DTypes in FBtype, and their determination for the instance in FBlock. Determined DTypes of the pins are not contained here, see `(5)`.

`(5)` Anyway, a `DinoutType_FBcl` refers the data type of the pin via

DinoutType_FBcl#dType (www). If the type is fix (determined, full qualified), then `dType` refers either one of the static defined standard `DType_FBcl` in

Because the `Din_FBcl` X1 and X2 are from the same `DinType_FBcl`, there `DType_FBcl` refers to the same instances.

Below there are some internal information about data associations as class diagram.

- The yellow part shows the presentation of the expression instance itself. The expression is presented in the data model in Java by a fbcl/fblock/FBexpr_FBcl (www) which is inherited fro `FBlock_FBcl`, here shown. The `FBlock_FBcl` refers the pins, shown as used types fbcl/fblock/PinExprPart_FBcl (www), and `Dout_Fbcl`, but derived from `Pin_FBcl`, which contains the aggregation `pint` to the `PinType_Fbcl`.

- This aggregations go to the green part, which is the FBtype situation. The `FBtype_FBcl` is the instance with the name "`ExprReIm2Cplx_UFB`". It has independent from the particular usage the data pins fbcl/fblock/DinType_FBcl (www) and fbcl/fblock/DoutType_FBcl (www). Not shown here: Both are derived from `DinoutType_FBcl`, and then from `PinType_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 aggregation `dType` shown in the graphic is contained in the inherited class `DinoutType_FBcl`. But for this type pins it is null, not set. Because the ***data type is not dedicated***.

- Instead, the possible non dedicated data type(s) is/are referenced from the `FBtype_FBcl` in the array aggregation `dTypes`. That are the red aggregations. The both `DinoutType_Fbcl` objects contain the index in this aggregation array, and hence the aggregation via index `ixDtype` between the data pins and the data types of this pins.

- On creation of the ExprReIm2Clpx_UFB instance first the corresponding `dTypes` aggregation array in fbcl/fblock/FBlock_FBcl (www) remains empty, not determined in the FBlock itself, if a data type is not notated on this pins, as usual.

- But on data type propagation this instances are set from the real used data types. Then they are full dedicated for this using instance of the expression.

- The indices in `Fblock_FBcl#dTypes` are determined by the indices DinoutType_FBcl.html#ixDtype , as also the Dtype_Fbcl instances there determines the kind of dependency between the both types. One is real, determined by 'N', the other is complex, determined by 'n', but both are depending. It means if one of the input is given as real type `float`, `typeChar 'F'`, then the output is dedicated by the correspond complex float type with `typeChar = 'f'`.

- And hence the input determines the output type, also if it is not the same, only depending, and the data type propagation can be forwarded from the outputs of the expression to the next FBlocks.

- The `usingPins` aggregation in `Dtype_FBcl` helps to find out the appropriate pins

7.6.2 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.

7.6.2.1 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

7.6.2.2 Resulting evout because of evin of a FBlock

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

In chapter *Error: Reference source not found* page 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.

7.6.2.3 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:

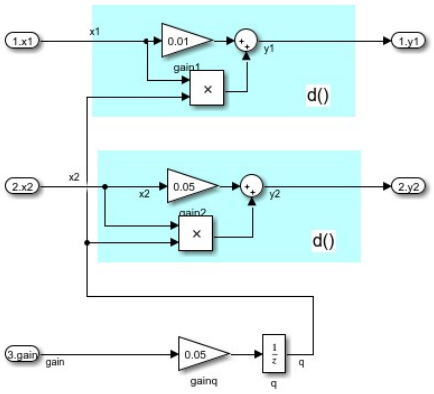

Figure 12: smlk/Testcg_MdlTstepSmlk.png

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:

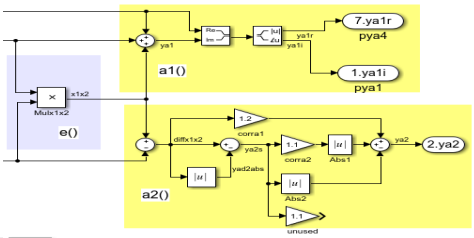

Figure 13: smlk/ParallelSimple_smlk_EvChainBack.png

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.

7.6.2.4 Info in pins for data to event processing

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

- `Evinout_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 in the following chapter *7.6.3 OFB: Build the event chain* to prevent to much effort for unnecessary JOIN of events.

The fblock.Pin_FBcl (www) contains

- `Pin_FBcl#mEvMdlChain`: Here the bit for the driving `evinMdl` (possible more as one) are set, also on Din, Dout, Evin, Evout pins. This is used to prevent twice handling for the event chain built.

The fblock.Evinout_FBcl (www) contains some elements which are set temporary while built the event chain:

- `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.

This bits are the same as `Pin_FBcl#mEvMdlChain`: (?) todo remove one

- `Evinout_FBcl#idEvent`: This is a unique identification for each event for all modules while translating. It is used as key in

fbcl/readSource/Dataflow2Eventchain_FBrd.html#mapEvPrepUpdIn_Queue (www) which contains the unique instance of the triple of three representative events to process, see todo

(empty page)

7.6.3 OFB: 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_UFB` FBtype if more as one event chain is necessary for data inpu ts.

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_UFB`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.

7.6.3.1 Start on module's evin

This is organized by the operation Dataflow2Eventchain_FBrd#connectEvents Forward() (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 foundError: Reference source not found*) 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.

7.6.3.2 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 `doutSrcxa` 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.

7.6.3.3 Check all other dinDst, build `listEvoutSrc`

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.
- If connected, all `doutSrcOther` are checked:
- `doutSrcOther.sConstant()`: `“#const: fbsrc.dout=>fb.pindst”` The `dout` pin is marked as delivering only a constant, then it can be ignored as contribution for the event chain. The constant value is evaluated either with the `init` event or (better) with a constant calculation before or with `ctor`.
- `zout`: and not `bCheckEvUpdoutMdl`: `“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.
- All other checks needs the `Write_FBlock_FBwr fbwSrcPrev` to evaluate all pins.
- `zout`: and `bCheckEvUpdoutMdl`: The update event bits should be added (TODO)
- `doutSrcOther.isEventChainDriven(evStart)`: The `doutSrcOther` is already driven by the same event from this chain. Then the associated `evoutSrc` are necessary to connect as immediately event in the chain, hence first added to the `listEvoutSrc`. But it is tested whether the event is already connected. This occurs on manually (graphic) connection. Then nothing is added.

TODO text more clear

`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`.

- If the `doutSrcOther` comes from an FBexpr and the other conditions were not met, then the inputs of the expression maybe more in the queue can also be constant, driven by Zout etc. There inputs should be evaluated in the same kind. Hence this operation `checkDinOtherAndConnectEv(...)` is called recursively to check it. The return value true indicates, all is found. On false, TODO

7.6.3.4 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.

7.6.3.5 Connect the events if all `dinDstOther` are driven by events using `listEvoutSrc`

In the positive case the event connection can be done.

The operation Dataflow2Eventchain_FBrd#connectEventMaybeJoin(...) (www) does the work. It gets the `listEvoutSrc` from chapter *Error: Reference source not found* 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:

<:@image:./img/odg/ExpressionExmpCombiBoolean.png :: title=ExpressionExmpCombiBoolean. :: size=6.3cm*1.66cm :: px=408*108 ::DPI = 164. >

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.

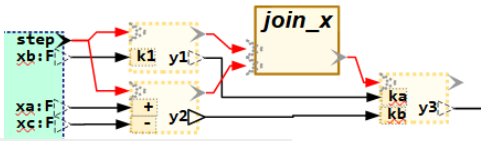

Figure 14: EventParallelJoin.png

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:

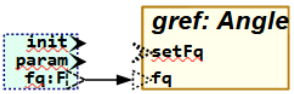

Figure 15: any image

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_UFB` 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_UFB*** 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_UFB* 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_UFB*** FBlock is necessary to fulfill this situaltion.

7.6.3.6 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 *Error: Reference source not found*. 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

7.6.4 Completion of condition events

Condition events are used for conditional execution of FBlocks, see main chapter “Handling of OFB” html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.6.8 Conditional execution with boolean FBexpr*. The completion is necessary for following FBlocks which are not immediately in one chain of the ‘true’ or ‘false’ output.

7.7 Code generation due the to event flow

Table of Contents

As written in *Error: Reference source not found* 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:

7.7.1 Using a templates for code generation with OutTextPreparer

This is the general approach: All generated codes are controlled by templates, see ./../../../../../Java/pdf/RWTrans/OutTextPreparer.pdf (https://vishia.org/Java/pdf/RWTrans/OutTextPreparer.pdf). 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:

<Code:Otx.><:set:GenCode1=".h">
<.Code>

<: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

<Code:Otx.><:set:GenCode2=”.c”>
<.Code>

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

<Code:Otx.><:otx:GenCode2:mdl>
<.Code>

<:type:mdl:org.vishia.fbcl.fblock.Module_FBcl>

as also shown in the following .

<:style:Code-Description:Label:????>Start of the script for C code generation in the code generation template example

```
<Code:Otx.><:set:GenCode2=".c"> ## extension .c for the c-File <.Code>
<: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 ./../docuSrcJava_FBcl/org/vishia/fbcl/fblock/Module_FBcl.html (https://www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/Module_FBcl.html).

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

7.7.2 Tracking the event chain for a module's operation

7.7.2.1 What are module's operations, `prcEvChainOperation(...)`

The event chain for an operation starts either on an Evout_FBcl of the module's event input, or on an Evout_FBcl of a FBlock which contains a state machine. The last one operation is called after entry in the appropriate state for the one-device code generation. The first one operations are start from outside, if an input event comes, respectively if this operation is called from outside, either manually or from code generation from the using ODB module.

The sensibility of an operation from an event output of a state machine FBlock depends on the implementation. It is only done if in the generated code a callback for this event operations is done, or if in the generated code an event queue for such events is used. A simple and often recommended kind if code generation uses polling for bits set by output events. This is described in chapter *7.7.4 Conditional events in the operation* page *56*

The operation which generates this operations
is .WriteCodegen#prcEvchainOperation(…)

7.7.3 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.

7.7.4 Conditional events in the operation

There are two reasons for conditional events:

a) FBexpr which produces a boolean output and which have a `true` or `false` event output are specific FBexpr of FBtype `ExprEv_OFB`. This both events comes after the input prep event, if the boolean result of the expression is either true or just false. This is the view to the event flow.

b) FBlocks which an internal StateMachine can either have a callback or a given event queue for its internal events (see *Error: Reference source not found* page *Error: Reference source not found*). Or it can have similar as a) bits for each events which are outside polled to execute to proper statements.

For the code generation in one execution order it means, with the boolean output of this FBexpr an `if(FBexprOut) { … }` should be produced: Or just also with the bits of a FBlock output with StateMachine. The FBlocks in the following chain are only executed if the boolean FBexpr output is either true or false, or the bit is 1. This is valid for all FBlocks in this chain.

But sometimes, this chain is more complex. It means, not immediately the `if(...){` can be code generated. Instead the condition should be first only immediately generated and stored in a module local bit variable one bit per conditional event, or just in an array of bit variables. The condition itself is valid already on emitting the event, and tested on consuming. The output event and all successive events are marked with the Bit index of the appropriate event. Check the event bit variable with the mask results in the necessary `if(evBits[1] & mEvxy){` as example for an array of event bits. In the same kind also an adequate bit mask inside a used FBlock is tested whether the event is emitted: `if(fbxy.evBits & mEvxy_FBxyType){` . For code generation the visible name is built from the event's name. for events from internal a FBlock, or just from the name of the `ExprEv_OFB`..The code generation also generates the bit mask of the events due to its names as constant definition, or in C als `#define mEvxy 0x0002` (example).

```
bool nameExprEv_OFB = b1 & b2;
  // ...
further code
if(nameExprEv_OFB) {
  // ...
this is the conditional evant chain
}
```

Note that the `ExprEv_OFB` instance should be named for example as “`evCondxy`”. Then the code es readable.

Successive following FBlocks with the same condition are all part of the same if branch.

A code as shown above is optimized by familiar compilers: The `bool` condition variable is usual held in a CPU-register or on a simple accessible stack location because it is immediately near it's usage is jmp condition. Hence it produces the same optimal code as

```
if(b1 & b2) {
  // ...
this is the conditional evant chain
```

But this more complex writing style is necessary because if some other events are joined in the following chain, the condition variables of all JOIN inputs should be regarded:

```
if(evCond1 & evCond2) {
  // ...
this is the conditional evant chain
}
```

… check the evOutSrc for Join or for true, false on code generation of

It means an event in the `queueEvSrc` inside the WriteCodegen#prcEvchainOperation(…) are optional marked with the ExprEv_OFB instance of FBexpr, and Also the class WriterCodegen.EvJoin contains a List of FBexpr_FBcl references of these (0, one or more as one) `ExprEv_OFB` for the condtion.

7.7.5 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.

7.7.5.1 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.

<:style:Code-Description:Label:????>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 *Error: Reference source not found*. This expressions are evaluated by tracking backward input values as described in *Error: Reference source not found*.

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

7.7.6 Expression to set elements in a variable

The variable to set can be an array variable or also a structured variable. It should be given either immediately on the output of the expression. Then it is a FBlock of Type VarL_UFB

7.7.7 Set the module output

The module output ***doutMdl*** in the graphic is a `Din_FBcl` as part of the inner pins of the module FBlock referenced via `Module_FBcl#fbp`. It should be presented by a data element in any target language. To support getter (encapsulation of data) of course proper operations can be generated, depending from the given module outputs, maybe also depending from their output event associations, but that's another question, see chaper TODO

Now it is important to distinguish the data type of the output and also the kind of the feeding input. The output data type can be a reference, a structure type, an array or a simple scalar variable. Note, a reference is a `Port_FBcl`.

For feeding inputs three situations should be considerate. The input can be a specific FBexpr which sets the elements of a following variable, which is here the module dout. The input can be a variable with the same data type and also `sizeArray`. Or the input can be any expression generated inline, where for all elements of the output the data are separated built. To do all this variants, some otx scripts exists. This scripts are called all in fbcl/writeFBcl/WriterCodegen#prcEvinFBlock(...) (www) under condition `if(evinDst.pint.kind.isModulePin())`. It is checked in this given order:

- `otx: setMdlOutScalar`: This otx script in `cImpl.otx` is called, if the modules output is scalar. This is the usual case. The script calls via WriterCodegen#genDinAccess(...) (www) to build the feeding input expression in the standard way.
- `otx: setVarArrayElem`: This otx script is the same as called on internal variables, if the ***doutMdl*** is an array type and the connection comes from one or more FBexpr of type `ExprSetArray_UFB`. This expression(s) set(s) the array elements of this module output variable.
- `otx: setVarStruct`: This otx script is the same as called on internal variables, if the ***doutMdl*** is a structured type and the connection comes from one or more FBexpr of type `ExprSetStruct_UFB`. This expression(s) set(s) the components of this module output variable.
- `otx: setMdlOutArrayExpr`: This otx script is called if the doutMdl is an array type and the input does not come from an `ExprSetArray_UFB`, it comes from an ordinary expression. It is the same script as for an array variable with this condition.
- `otx: setMdlOutArrayCpy`: This otx script is called if the doutMdl is an array, but the input is not an expression. Then expected, the input comes from a variable or an output of any FBlock which has the same type. Hence the array data needs to copy, or also if the implementation is an array pointer, the pointer is copied.
- `otx: setMdlOutArrayCpy`: This otx script is called similar as `otx: setMdlOutArrayCpy`, if the ***doutMdl*** is not an array, but also not a scalar (first tested), hence it is a structured data. Following it should copy the data or pass the reference to the data.

7.7.7.1 create code for ctor

7.7.7.2 create code for init

7.7.7.3 call any FBlock content

7.7.8 Code generation for FBexpr

The possibility of expressions in ...fblock.FBexpr_FBcl (www) is flexible, see using description in chapter html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.7 Expressions inside the data flow* on page . General four kinds of generation are to be distinguished:

- `'.'`: Set components of an output variable. That is `.re`, `.im`, or elements of a used defined structure. The FBtype of the expression is `ExprSetStruct_UFB`. The expression should have exact one variable on output, see html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.7.5 Set components to a variable* page. The input names are the names of the component, it means “re” or “im” for set complex components or the name of elements in a struct variable. For generation the otx-Script `SetVarCmpn` is used

- `“]”`: Set array elements of an output variable. The FBtype of the expression is `ExprSetArray_UFB`. The expression should have exact one variable on output, see html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.7.5 Set components to a variable* page. The names of the input should contain the array indices with the schema `“X0”` to access `[0]`, `“X2_3”` to access `[2,3]` of a two-dimension array, `“Xix_ix2”` to access a two dimensional array with the index variables `ix` and `ix2`. This index variables should be accessible as variables inside the module, adequate factors on expressions, it are wired on the `K` inputs of the expression.

- `“:”`: Access to components of the connected only one input variable. See html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.7.7 FBexpr as data access* page .

- `'['`: Access to array elements of the connected only one input variable. See html (www) / Handling-OFB_VishiaDiagrams.pdf (www): *5.7.7 FBexpr as data access* page .

- `“=$”`: 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).

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 *Error: Reference source not found* 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 *Error: Reference source not found*, the otx-element n `<&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) ) ;
```

This is due to the graphic *Error: Reference source not found* page . 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 WriterCodegen.#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 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 oper ation fbcl/writeFBcl/WriterCodegen#genValueExprDin(...) (www) is called.

7.7.8.1 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 dete rmining.

- The operator for the input is prepar ed in FBexpr_FBcl#setOperatorToPins(prj) (www). This operator per `din` is output to the generated code if the `din` has either a connection, a constant or 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 FBexpr_FBcl#setOperatorToPins(prj) (www). 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