UFBgl
Unified Function Block
modeling graphic language
-
Internal software description

Dr. Hartmut Schorrig
www.vishia.org
2024-01-04
Table of Contents

1 Data model for FBcl and conversions................................................................. 6

2 Software to evaluate the graphic information from Libre Office...................... 7
   1.1.1 Main class and arguments........................................................................ 7

3 Data Model......................................................................................................... 8
   3.1 FBlockType_FBcl, PinType_FBcl................................................................. 8
      3.1.1 FBlockType_FBcl.................................................................................. 8
      3.1.2 FBlock_FBcl......................................................................................... 9
      3.1.3 Pin_FBcl and PinType_FBcl................................................................... 9
      3.1.4 PinType_FBcl....................................................................................... 10
   Operations or Actions assigned to the Pins, code generation............................. 10
   Association between Event and Data Pins.......................................................... 11
   Association between Input and Output pins....................................................... 11
   Association between prepare and update events................................................ 11
   Multiple pins..................................................................................................... 11
   Data Types......................................................................................................... 12

3.2 Module with FBloks....................................................................................... 14
   3.3 Write instances for FBlock(Type)_FBcl, Module_FBcl................................. 15
   3.4 DType_FBcl and DTypeBase_FBcl............................................................... 16
      3.4.1 Using DType_FBcl.................................................................................. 16
      3.4.2 Using DTypeBase_FBcl......................................................................... 17

4 Preparation of the data....................................................................................... 18
   4.1 Read data from LibreOffice odg files............................................................. 18
   4.2 Read data from Simulink............................................................................... 18
   4.3 Read data from IEC61499 text files (fbd)....................................................... 18
   4.4 Data and event flow...................................................................................... 18
   4.5 Forward and backward declaration of data types........................................... 20
      4.5.1 Forward/backward propagation of dedicated pins................................. 20
      4.5.2 Forward and backward propagation of non dedicated pins.................... 20
      4.5.3 Forward declaration for depending pins of a FBloclType.......................... 20

5. Execution (code generation) in an instance of a FBlocl type............................ 24
   4.1. Data and event flow.................................................................................... 24
4.2. FBlockType Kinds and their usage.................................................................26
4.3. Construction, init, run with several step times or events and shutdown........27
4.4. Prepare and update actions...........................................................................29

4.4.1. Example prepare and update for boolean logic........................................30
4.4.2. State of the art, ignoring prepare and update concept.............................31
4.4.3. Example prepare and update in source text languages (C/++)....................31
4.4.4. Example prepare and update in 4diac with MOVE-FBlock.......................33
4.4.5. Example prepare and update in Simulink..................................................38
4.4.6. Example prepare and update for odg Graphic code generation (Libre Office).........................40
4.5. How to associate the prepare to the update event........................................42

5 Internal data coding in a Libre-Office file........................................................45
5.1 The file format of odg.....................................................................................45

1 Substantiation.....................................................................................................47
3.1 Sub chapter......................................................................................................48

1.1.1 SubSub chapter.........................................................................................48
1.1.1.1 4th Chapter..........................................................................................48

1. Code-Description..............................................................................................48

7 Links and bibliographie......................................................................................48
8 Requirements.....................................................................................................48

Table of Figures

Figure 1: Conversion to and from FBcl..............................................................6
Figure 2: FBlock_FBlockType_Pin.png..............................................................8
Figure 3: Module_FBcl......................................................................................14
Figure 4: ExprRelm2Cplx_DTypeDeps.png........................................................21
Figure 5: Smlk / Exmpl_SimpleStepTimes.png.................................................24
Figure 6: 4diac / Exmpl_SimpleStepTimes.png...............................................24
Figure 7: odg / Exmpl_SimpleStepTimes.png....................................................26
Figure 8: simple regular state machine.............................................................26
Figure 9: Moore automat....................................................................................29
Figure 10: Moore automat 2..............................................................................29
Figure 11: data flow with qout...........................................................................29
1 Data model for FBcl and conversions

This figure 1 should show the possible sources and results of FBcl conversion. FBcl means “Function Block connection language”, it is the kind of presentation of FBlocks and their connections. The IEC61499 is a language which is used for automation programming with FBlocks, and in the “composite FBlock” diagram exact this FBlock connections are used. Simulink is as known a FBlock presentation with their connections as well as some other graphic programming presentations.

The LibreOffice Graphic works with some style templates and can also converted to the inner data model. Only this presentation supports also UML elements (aggregations, inheritance…) see [1].

The output is now also the IEC61499 Composite diagram presentation both in textual and xml form, but also code generation to some destination languages.

The data model in Java runtime presents the content in some classes. The package www.vishia.org/fbg/docuSrcJava_FBcl/org/vishia/fbcl/fblock/package-tree.html contains the classes of the Java runtime data.


2 Software to evaluate the graphic information from Libre Office

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

```
java -cp tools/vishiaBase.jar; tools/vishiaFBcl_odg.jar org.vishia.fbcl.Ufbconv
```

This document is related to the processing algorithm with understanding software and data structures. Hence it may be recommended to look also in the source files written in Java. Some links from this document to the javadoc generated source documentation are given. You should refer to the users manual for the Libre-Office drawing and conversion

1.1.1 Main class and arguments

The main class for odg reading is org.vishia.fbcl.Ufbconv.
3 Data Model

3.1 FBtype_FBcl, PinType_FBcl

The diagram Figure 2 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 **FBBlock_FBcl** because of its interconnections. To distinguish between pure classes and instances of classes in the diagram the FBlock presentation of classes has an internal name starting with “$”.

This diagram Figure 2 shows the pins of Blocks and their types only on the example of data input pins. The other pin kinds are similar.

### 3.1.1 FBtype_FBcl

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

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

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

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

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

The Java class FBlock_Type_FBcl on user level can be defined graphically with FBUMLgl. It can be stored as fbd file due to the IEC61499 standard.
3 Data Model

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

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

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

### 3.1.3 Pin_FBcl and PinType_FBcl

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

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

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

### 3.1.2 FBlock_FBcl

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

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

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

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

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

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

- **mAssocInOut**: long: up to 64 input and output associations. This is not immediately shown in IEC61499 but can be determined. See also 4.2. FBType Kinds and their usage. For Standard FBBlocks 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 FBBlocks 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 FBBlock consisting of an usual graphical interconnection of FBBlocks the input–output-association are an result of the connections.

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

- The DinoutType_FBcl (www) contains also a data type information for the `dinPin` and `doutPin` as well as also for `refPin` and `portPin`., see Data Types
- The EvinoutType_FBcl (www) contains also references to EccAction_FBcl (www) for immediately execution of actions to events, see also
- The PinTypeRef_FBcl (www) refers with FBtype_FBcl fbRef the type of the reference.

Operations or Actions assigned to the Pins, code generation

The EvinType_FBcl has usual an assigned Ecc_Action_FBcl. On inner Pins of a module the input event is related to a pin of type Evout_FBcl, and also a data inputs are offered with a Dout_FBcl, an output to the inner FBBlocks of the module, the actions are assigned to the common class PinType_FBcl. TODO it’s better it is dedicated.
The **DoutType_FBcl** has an assigned **Ecc_Action_FBcl** if the inner logic of a **FBtype_FBcl** comes from a **Composite FBlock** (a FBlock with graphical content). Then this action describes the access operation to this output pin or also to more as one related output pins, depending on code generation rules.

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

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

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

**Association between Event and Data Pins**

The Pins in **FBlock_Type_FBcl** are contained in adequate arrays. The position in the arrays are used for bit masks **mAssociatedInOut** and

Java class: org.vishia.fbc.fbblock.PinType_FBcl.html#mAssociatedEvData =>www.

**Association between Input and Output pins**

This should be contained in EccAction_FBcl

**Association between prepare and update events.**


contains the index of the prepare event in a given update event.

**Multiple pins**

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

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

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

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

If the FBtype_FBcl has such a pin, any pin number from ...0 or just ...1 is available and refers the same pin “...0999” in the type. The pins has all the same properties, but of course different data connections, or different constants, or also different data types just as pins of
instances have in comparison to the type pins. The code generation can deal with this situation.

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

**Data Types**
3.2 Module with FBlocks

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

The image shows the important ones:

1. The representation of the module to outside with the FBtype_FBcl (www) is referenced as ifcFB (interface FBlock), and is referenced as mdl from there. 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.

2. 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 is only internally necessary, it is also mirrored in respect to the pin direction to (1).

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 is only internally necessary, it is also mirrored in respect to the pin direction to (1).

4. The module consists of many FBlocks, which are referenced all sorted by name via idxFBlock. Also expressions are FBlocks.

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

6. 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.

7. Also States and actions are referenced, see chapter TODO.

Figure 3: Module_FBcl
3.3 Write instances for FBBlock_FBcl, FBtype_Fbcl, Module_FBcl
TODO
3.4 DType_FBcl and DTypeBase_FBcl

3.4.1 Using DType_FBcl

Instances of DType_FBcl (www) are referenced from data pins, see chapter 3.1.3 Pin_FBcl and PinType_FBcl. They contain:

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

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

There are a few “fixed” Dtype_FBcl instances. That are those 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.
3.4.2 Using DTypeBase_FBcl

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

The basic types without container and array specifications are either standard types, contained in DTypeBase_Fbc1.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.
4 Preparation of the data

4.1 Read data from LibreOffice odg files

4.1.1 The file format of odg – content.xml

4.1.2 Read content.xml to internal data

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

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

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

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

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

The internal Handling of expressions needs a little bit explanation. Refer to *UFBgl-Diagrams-OpenLibreoffice.pdf* chapter *Expressions inside the data flow* to see the capabilities of expressions.
4.2 Read data from Simulink

4.3 Read data from IEC61499 text files (fbd)
4.4 Forward and backward declaration of data types

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

4.4.1 Forward/backward propagation of dedicated pins

The data type propagation starts by adding all pins to an internal List<Dout_FBcl> listDout with dedicated DType_FBcl (www) on dout pins of all FBlocks and dedicated pins of the module’s inputs. From this pins the connection is traced to connection Din_FBcl pins to following FBlocks. Then, in the reached FBlock_FBcl, depending pins which are yet not fully dedicated are set, see next sub chapter.

The same is done with an internal List<Din_FBcl> listDin with dedicated DType_FBcl on dout pins of all FBlocks and dedicated pins of the module’s outputs, but only with pins which are not reached by forward propagation. Pins which are already reached by forward propagation, do not need to handle again. The remaining Din_FBcl pins which are dedicated but not already forward propagated are then backward traced to the connected Dout_FBcl. Hence this pins are now also dedicated. Also tracing through the FBlocks is done as described in 4.4.3 Forward declaration for depending pins of a FBtype also for backward tracing.

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

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

4.4.2 Forward and backward propagation of non dedicated pins

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

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

4.4.3 Forward declaration for depending pins of a FBtype

If pins are not full qualified then some pins depends from another. If the data type of one pin is dedicated, also all or some other pins should be dedicated with the same data type. A simple expression can only have the same dedicated data type on all its pins.

But specific mathematic expressions have depending dedications. Simple, look on the expression which combines real and imagin part to a complex value. It is drawn in graphic as Figure 4.

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

PinExprPart_FBcl.

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

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

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

This is also done if no extra Dependency exists as shown in Figure 4.

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

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

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

For this algorithm the distinction between \texttt{DType_FBcl} instances in the FBtype definition and \texttt{DType_FBcl} instances used in the module for the \texttt{FBlock_FBcl} pins seems to be a little bit sophisticated. To prevent errors in the algorithm for some cases, the affiliation of a \texttt{DType_FBcl} instance to the instance or type usage of \texttt{FBlock} is given with the variable \texttt{Dtype_FBcl#fixTypePin}. It is an enum with the value \texttt{eType} or \texttt{ePin} for affiliation either to the type or to the \texttt{FBlock} instance, and also the value \texttt{eFix}. The last value is one of the few instances which are standardized given for scalars.
4.5 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 from the graphic model. It is automatically generated due to the data flow.

### 4.5.1 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

### 4.5.2 Simulink: Binding data to sample times, 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:

![Simulink Diagram](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:
4 Preparation of the data

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 FBBlock.

4.5.3 Temporary info in pins

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

- Dinout_FBcl#bEvDataProp: If this is true, then the appropriate pin is reached by an event chain, the doutSrc pin is set with the evoutSrc in the chain respectively the dinDst was tracked already for build the event chain.

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

- Evout_FBcl#idxRepresentingEvents: This is a HashMap which contains all evoutSrc which are existing in the appropriate event chain as members before. It means, all fb.evout of the chain. This information is important to detect events, which are already handled in the chain which seems to be newly inputs. But they are not newly inputs.

4.5.3 UFBgl: Build the event chain

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

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

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 4.4. Prepare and update actions) are combined with their prepare events due it is given in a UFBgl module input block (style ofbMdlPins). These pins are associated in a class EvPrepUpdInQueue (www). The doutSrc pins of the module are marked with doutSrc. bEvDataProp = true because they are driven by default by the event.

The operation prcEventPrep(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 \texttt{doutSrc} pins (firstly the module \texttt{din} pins, it is \texttt{Dout_Fbl}) forward. This is a two-stage loop because there may be more as one \texttt{doutSrc} pins associated to an \texttt{evoutSrc}, and there may be more connections for each \texttt{doutSrc} to the \texttt{dinDst}.

It is asserted that \texttt{doutSrc.bEvDazaPropg} because elsewhere the event should not be propagated. But the connected \texttt{dinDst} are tested if(!\texttt{dinDst.bEvDataPropg}) {...} If an \texttt{evinDst} 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 \texttt{evoutSrc} step with the \texttt{doutSrc} \texttt{xa} and the \texttt{dinDst} \texttt{y0.x2}. For this input the associated \texttt{evinDst} input(s) of the FBBlock are picked. More as one is possible but usual only one \texttt{evinDst} is existing. With this information the operation checkDinOtherAndConnectEv(...) (www).

is called. This operation checks also all other \texttt{din} pins which are associated to this \texttt{evinDst}. Because, the quest is not the data connection, it is the event connection. With that it is detected which \texttt{evoutSrc} are altogether necessary driving the \texttt{evinDst}. Often this is only the one given \texttt{evoutSrc} tracked with the \texttt{doutSrc}, but it is possible that other pins are driven by \texttt{doutSrc} form other events. Then a Join_UFBgl FBBlock is necessary to firstly join this more \texttt{evoutSrc}, and the output of the Join FBBlock is connected then with the \texttt{evinDst}.

This operation tests all other \texttt{dinDstOther} to appropriate \texttt{doutSrcOther} respectively the connection state of the \texttt{dinDstOther}

There are the following cases possible, in console font in “ ” the output to the log is shown:

- \texttt{constant:} “#fb.din” The \texttt{dinDstOther} is driven by a constant value, no event necessary, it is ok.
- \texttt{zout:} “\texttt{fbsrc.dout%=>fb.pindst}” The driving output is a state variable, style ofp2Out... in the graphical model. The output value can be taken without an event. It is ok. But for an \texttt{evinUpd} this property is just not relevant, the update event chain works with this output.
- Not connected, \texttt{constant:} “#\texttt{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.
- \texttt{bEvDataPropg:} “\texttt{fb.evSrc:doutSrc+=>dinDst}”: The \texttt{dinDst} is driven by an \texttt{doutSrc} which is already driven by an event in a propagated chain. This \texttt{evSrc} is taken as one input for the \texttt{evinDst} firstly stored in a \texttt{listEvoutSrc}. This list is temporary built for the \texttt{dinDst} of the checked FBBlock inside the checkDinOtherAndConnectEv(...) operation.

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

It is important that a FBBlock’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 FBBlock 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 FBBlock should be inserted in this event chain before, should be calculate first. For that checkDinOtherAndConnectEv(...) is called recursively, but with this dependingevin on the depending FBBlock. 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 FBBlock in recursions. Going back after recursions
4.6 Code generation due to event flow

As written in 4.5 Identification of the event flow due to data flow the event flow results from the data flow. The code generation can now use the event flow.

For this 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.

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 intermediate **evin** inside a module were built, they are present in additional **evin** of the FBlock used which presents the module. These additional **evin** are depending from the originally plant **evin** of the module. These builds also operations, but these operations should be called only after calling the depending operation. That should be regarded either manually if the code to use the module is hand-written, or it is regarded by the code generation of the using module translated with UFBgl. These additionally **evin** are regarded while building the event chains, hence clarified for code generation. See chapter TODO eventchain.

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.

### 6.6.1 Resulting evout because of evin of an FBlock

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

### 6.6.2 Access operation to dout, arguments

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

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

### 6.6.3 Using a Template with OutTextPreparer

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

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

The template files should contain parts:

```xml
<:otx:GenCode1:mdl, out>
```
**4 Preparation of the data**

This is the main script for header files. The main script for implementation c file starts with

```c
/**Generated by org.vishia.fbcl. made by...*/
#include HGUARD_<&mdl.name>
...
```

The scripts (part of the template `<:otx:...`) with names GenCode... without number or numbered started with 1 are necessary for one or more output files.

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

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

adequate to each GenCode... start script.

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

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

### 1. Script for C code generation in the code generation template example

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

<:otx:GenCode2:mdl, out>
/**Generated by org.vishia.fbcl. made by Hartmut Schorrig, script 2023-12-21*/
#include "&<&mdl.name>.h"
<:for:header:mdl.iterImport()>
#include "<:<><&header.getValue()>"<:>><:n></for>

<:for:evinMdl:mdl.fbp.evout>
## all input events as chain
/**Operation `<&evinMdl.name>(...) */
void `<&evinMdl.name>_<&mdl.name> ( ) {
} // `<&evinMdl.name>_<&mdl.name>
<:for>

#endif //HGUARD_<&mdl.name>
<:otx>
```

The arguments for the main scripts for code generation are always `mdl` and `out`. `mdl` is the

#### 4.6.5 Tracking the event chain for an operation
4.6 Code generation due to event flow
5. **Execution (code generation) in an instance of a FBlock type**

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

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

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

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

4.1. **Data and event flow**

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

**Pure data flow with Sample time designation versus event flow**

In comparison to other FBlock diagrams for example from Simulink, usual only the data flow is shown there. It determines the execution order, whereby different step times are used. Each sample time has its data flow. The Figure 7 shows that, the step times are shown here with colors and also with “D1”, “D2”.

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

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

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

**Event queue for execution, also for distributed devices**

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

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

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

Automatic detection of event flow

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

![Exmpl_SimpleStepTimes.png](image)

4.2. FBtype Kinds and their usage

In IEC61499 there are different types of FBBlocks:

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

b) Standard FBBlocks with more simple operations, as Object Orientation with more events, but with simple association between the input event and output event. The term "Standard FBBlock" is used in IEC 61499 for FBBlocks which have a state machine, named \texttt{ECC = "Execution Control Chart"}. Any state can have one or more associated operations, which are executed on state entry, and one or more associated output events, which are activated after the entry operation execution also on state entry.
If this state machine is simple regular, then any input event is associated to exact one immediately coming output event. simple regular means, the IDLE or START state is triggered by each of the existing events, forces entry to exact one destination state, which unconditionally goes back to IDLE. Then each event is associated to one (or more) operations. Each operation may set outputs, or change the internal numeric state (not the ECC state) of the FBlock. After execution of each operation exact one output event is created. This is shown in the example ECC right side.

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

• c) Standard FBLOcks with State machine with more events, with a real maybe complex state machine. For that FBLOcks the coming output events after an input event depends of the inner state of the machine. It is not simple predictable. Hence, the code generation can not participate from. If the following FBLOcks in the event queue are arranged on the same device and hence should work in the same thread, the following operations the module should be called in execution order controlled by a bit mask value, which presents one or more coming output events. This bit mask value should be delivered from the FBlock, proper to the set output events. The other variant is: Using an event queue and processing it either in the same device or also possible send the event to the other destination device where the following content of the module is arranged.

• d) Composition FBLOcks with dispersed operations. A Composite FBLOck in terms of IEC61499 contains the composition of some FBLOcks which are wired together with a data and an event flow. The member FBLOcks can have any type of this list. It means the execution order or operations of a composite FBLOck can be complicated, not well obviously. For usage (view from outside) the code generation offers some operations associated first to the input events, but builds so named meta events which presents the inner event flow. If the inner FBLOcks are on the same device, then with the meta events a proper order of execution can be found, working without event queue.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

To calculate the new values, exclusively the old values should be used in all parts of the whole system of equations. Only then the solution is mathematically exact. This is the prepare phase. After them, or before the next step, the new values should be declared as current state, that is the update concept in a proper kind for the Moore state machine. Here the Block T for Transitions is the preparation, calculates the new state for D-Inputs of the FlipFlops, and the Block S is for update, saves the prepared state as current one. This is classic.

Exact the same is drawn in Figure 12 right side.. Only the positions are a little bit changed. But compare it with the next image:

Figure 11: Moore automat

To calculate the new values, exclusively the old values should be used in all parts of the whole system of equations. Only then the solution is mathematically exact. This is the prepare phase. After them, or before the next step, the new values should be declared as current state, that is the update concept in a proper kind for the Moore state machine. Here the Block T for Transitions is the preparation, calculates the new state for D-Inputs of the FlipFlops, and the Block S is for update, saves the prepared state as current one. This is classic.

Figure 12: Moore automat 2

Also the theory of digital machines from Moore and Mealy based on this approach. Look on Figure 11 and Figure 12. The Figure 11 is from https://en.wikipedia.org/wiki/Moore_machine. It shows the prepare - update concept in a proper kind for the Moore state machine. Here the Block T for Transitions is the preparation, calculates the new state for D-Inputs of the FlipFlops, and the Block S is for update, saves the prepared state as current one. This is classic.

Figure 13: data flow with qout

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

The Figure 13 opens up an understanding of what happens during signal processing in control technology for analog variables or for automation processing. It is primarily the same

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

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

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

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

![Diagram of Timing prepare, update and hardware access](image)

**Figure 14: Timing prepare, update and hardware access**

### 4.4.1. Example prepare and update for boolean logic

Exact the same approach is also used for boolean logic with D-Flip-Flops: The next value (as booleans) is prepared by logic on the D-inputs of Flipflops, and then all together on the same time are updated to the Q-output with a clock edge. The image above shows any processing signals (with the AND) which uses a value from the previous step time. One result of preparation is the signal $y_1$ which is output as $y$ valid for the next step time. For that the outputs on an IC (for example FPGA) have DFF on the pins. The signal is 'clocked', it comes time synchronous to a central clock. But the same signal is also used in a module after, where it is compared with the previous state of the same signal. It means the difference of the output in time can be built, here evaluated with a XOR to detect changes.

![Diagram of Example binary logic prep & update](image)

**Figure 15: Example binary logic prep & update**
4.4.2. State of the art, ignoring prepare and update concept

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

See the next examples.

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

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

```
Cpp: Simple integrate

```

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

Filter algorithm in C integrates dependent two values

```
Filter algorithm in C integrates dependent two values

1: static inline void step_OrthBandpassF_Ctrl_emC
2: (OrthBandpassF_Ctrl_emC_s* thiz, float xAdiff, float xBdiff)
3: {
4:   Param_OrthBandpassF_Ctrl_emC_s* par = thiz->par;
5:   float a = thiz->yab.re;  // store the current value of component yab.re
6:   thiz->yab.re = par->fI_own * thiz->yab.re;
7:       + par->fI_oth * ( thiz->kA * xAdiff - thiz->yab.im);  // integrate .re
8:   thiz->yab.im = par->fI_own * thiz->yab.im;
9:       + par->fI_oth * ( thiz->kB * xBdiff + a);  // integrate .im
10: }
```

The yab.re and yab.im are the both the current and also the new values after solving the differential equations. For an exact result it is very important to use the previous value a in line 4 instead the already new calculated value yab.re for calculation of yab.im. This is a simple solution. prepare and update are done also in one step, but the current value for the second equation is stored immediately in an individually variable.
But what is happen for this solution if the current values of the integrate variables are need for more operations, in this example for a more complex filter for harmonics. Then it is better to have a systematic solution, which looks like:

Filter algorithm in C consequently with prepare and update

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

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

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

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

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

It is to decide which is more important, a very fast algorithm or obviously sources. Unfortunately the compiler optimization does not solve here this problem.
4.4.4. Example prepare and update in 4diac with MOVE-FBlock

The example of the simple integrate is also solvable by the simple calculation order controlled by the event flow: 4diac prep & update. But here the update is an extra event chain with \( \text{upd} \) and \( \text{updO} \). The prepared result of ADD is available for further preparation which can also use the current (previous) value of the ADD, present in \( y \) and \( yz \), for example to build a difference, the growth of the integrate between two step times, similar as the XOR in the boolean logic image Example binary.

![Figure 16: Example 4diac prep & update](image1)

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

![Figure 17: Example 4diac prep & update](image2)

This is almost the same as image Example logic prep & update.
This image shows the bandpass filter algorithm in 4diac similar as in Filter algorithm in C integrates dependent two values. The current previous values for integrate are used from the _F_MOVE_1 FBlock right side, but after calculate the filter the both _F_MOVE_1 FBlocks are also updated immediately in the same event chain. This works exact for the filter algorithm for one filter, but it gives slightly wrong results if more than one filter is used, for example for harmonics. Look for this usage of the image Example binary logic prep & update:
There are two differences, first is the upd and updO event for update, but also a ya and yb is given which presents the calculated new outputs. This may be important because if the outputs are used as process outputs, they become active in the next step time because of course, it should be first give to the output device. If only the yaz and ybz are given, then they are the old values, one time back, which causes an additional dead time for control.
4.4. Prepare and update actions

Figure 11. OrthBandpass in a filter application

The image above just shows an application where two OrthBandpass without update event are used, one for the fundamental oscillation, and one for an harmonic. Both values are output, $y_{filt}$ is the filtered output of $x$ and $y_{2harm}$ is the detected harmonic. That is the mission and possibility of this filter stuff. Also more as one harmonic is possible to filter. The principle is, all detected waves are added and compared with the input. The difference input for all OrthBandpass is equal, but each OrthBandpass has the resonance for its own frequency. If all frequencies are summarized and this is sufficient then the difference is 0 and the signals are stable.

But back to the event topics. The events are connected in that kind, that the resulting signals from the filter are presented in the outputs. The $F_{ADD\_1}$ is calculated firstly, takes the old current values from the step time before, put it in the feedback, and last the both OrthBandpass FBlocks are calculated. This is tricky. But what about if for more harmonic parts or other evaluations outside of this module the old current values are necessary. Then the logic becomes more complex.

Using the prepare and update concept is more obviously.

Figure 12. OrthBandpass in a filter application

Using the base variant of the filter with update, now also an filter application is possible and simple understandable, which outputs the filtered signal as new one for output on physic, and delivers also signals for further evaluation, here both components of fundamental and harmonic oscillation and the magnitude of the harmonics. The last one is calculated in the upd event chain.
The interface shows the assignment of $y_{fil}t$ to the prepO output event, and the other signals to the updO event. The prep event queue is for ordinary evaluation of calculations, the end signal may be output to hardware or transmit, and the upd event queue delivers signals as state of another event updO to use it in the prep calculation (in the comprehensive superior module). But of course both event chains are related, not formally, but semantically. The event source should organize the proper order of prep and update.
4.4.5. Example prepare and update in Simulink

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

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

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

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

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

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

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

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

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

It means the mdlOutputs(…) with the combinatoric calculation is exact the prepare phase, and the mdlUpdate(…) is the update. For update a few combinatorics inside the FBBlock can be also calculated. That makes it a little bit more powerful for some special desires, but also more complicated. The state can also be set only from internal variables calculated on
mdlOutputs(...) due to the image data flow with qout.

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

The unit delay FBlock

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

Figure 14. Bandpass filter base FBlock in Simulink

Figure 15. setable unit delay in Simulink

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

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

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

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

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

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

Wrapper for OrthBandpassF_Ctrl_emC in IEC61499 to adapt to C

```c
FUNCTION_BLOCK OrthBandpassF_Ctrl_emC
  EVENT_INPUT
    step0 WITH OTHIS, Tstep;
    init WITH param;
    step WITH xab;
```
Execution (code generation) in an instance of a FB\textit{lock} type

```
upd WITH step;  (* Note: Association of upd to the step dataflow *)
END EVENT
EVENT_OUTPUT
  initO WITH initOk;
  stepO WITH yab;
  updO WITH upd, yabz;  (*Note: Assoc upd input event)
END EVENT
VAR_INPUT
  OTHIS: OrthBandpassF_Ctrl_emC\textunderscore REF;
  xab : CREAL;     (* Difference to adjust *)
  param: Param_OrthBandpassF_Ctrl_emC\textunderscore REF; (* reference to parameter *)
  Tstep: REAL;   (* Step time for calculations *)
END_VAR
VAR_OUTPUT
  yab: CREAL;     (* new calculated value *)
  yabz : CREAL;   (* state value from last update *)
  initOk: BOOLEAN;
END_VAR
Wrapper for OrthBandpassF\textunderscore Ctrl\textunderscore emC in IEC61499 to adapt to C
```

```
VAR
  THIS: OrthBandpassF\textunderscore Ctrl\textunderscore emC\textunderscore REF;
END_VAR
EC STATES
  IDLE;        (* EC idle state *)
  CTOR: CTOR;  (* Constructor *)
  INIT:INIT -> initO; (* EC State with Algorithm and EC Action *)
  STEP: STEP -> stepO, ->step2;
  UPD: UPDATE -> updO;
END_STATES
EC_TRANSITIONS
  IDLE TO CTOR:= ctor; (* constructor call *)
  IDLE TO INIT:= init; (* An EC Transition with event*)
  IDLE TO STEP:= step;
  IDLE TO UPD:= upd;
  CTOR TO IDLE:= 1;
  INIT TO IDLE:= 1;
  STEP TO IDLE:= 1;
  UPD TO IDLE:= 1;
END_TRANSITIONS
ALGORITHM CTOR IN ST:
  \text{THIS} := ctor_OrthBandpassF\textunderscore Ctrl\textunderscore emC(othiz:=OTHIS, Tstep:=Tstep);
END_ALGORITHM
ALGORITHM INIT IN ST:
  initOk := init_OrthBandpassF\textunderscore Ctrl\textunderscore emC(thiz:=THIS, param:=param);
END_ALGORITHM
ALGORITHM STEP IN ST:
  step_OrthBandpassF\textunderscore Ctrl\textunderscore emC(thiz:=THIS, xAdiff:=xab\textunderscore real, xBdiff:=xab\textunderscore imag);
  yab := THIS.yab;
END_ALGORITHM
ALGORITHM UPDATE IN ST:
  upd_OrthBandpassF\textunderscore Ctrl\textunderscore emC(thiz:=THIS);
  yabz := THIS.yabz;
END_ALGORITHM
END_FUNCTION_BLOCK
```

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

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

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

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

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

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

### 4.5. How to associate the prepare to the update event

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

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

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

Step and update association in FBD

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

With this information, and the information in the state machine (ECC) about associated output events to inputs (see link TODO) the necessary event connections can be determined. See chapter TODO other html document to write
5 Internal data coding in a Libre-Office file

5.1 The file format of odg

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

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

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

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 21: ContentOfodg-content-xmlPure.png
After beautification it looks like

This is right side truncated, it shows the graphical "group" with the "ClassA name1" as shown in Figure 12: Moore automaton 2 page 36. 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 known style "ofpAggrRight" as you see in the content.xml in the `<style...>` part.

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

static void testNumeric (TestOrg parent) {
    TestOrg test = new TestOrg("testNumeric formatted", 2, parent);  // The pattern.
    OutTextPreparer otxNumLine = new OutTextPreparer("otxNumLine"  //arguments need and used.
        , "ix, f1, f2"
        , "One line with values at ix=<&ix>: <&f1:%1.3f>, <&f2:%1.3f> \n")
    OutTextPreparer.DataTextPreparer otxData = otxNumLine.createArgumentDataObj();
    StringBuilder sOut = new StringBuilder();
    try {
        for(int ix = 0; ix < 6; ++ix) {
            double f1 = ((float) ix)/2.0;
            double f2 = Math.sin(f1);
            otxData.setArgument("ix", ix);
            otxData.setArgument("f1", f1);
            otxData.setArgument(2, f2);
            otxNumLine.exec(sOut, otxData);
        }
    } catch(IOException exc) {
        test.exception(exc);
    }
    test.expect(sOut, cmpNumeric, 5, "");
    test.finish();
}

This operation is part of class Test_OutputTextPreparer (=) www
The working instance of the OutText-Preparer is created here locally.
Recommended as final instance variable.

- List1_24 with more indent
- List with manual bullets
  - List2

3.1 Sub chapter

1.1.1 SubSub chapter

1.1.1.1 4th Chapter

To show enough content in a diagram you may use an A3 paper in As living example look on the following Class-Object-diagram:

Figure 22: View 40%
7 Links and bibliographie


8 Requirements

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

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