PLCopen®

for efficiency in automation

PLCopen - Promotional Committee 2

–

Training

Guidelines for usage of Object-Oriented Programming

Version 1.0 – Official Release

Date: November 18, 2021

Total number of pages: 27

PLCopen Guidelines for usage of Object-Oriented Programming

The following paper is a document created within the PLCopen Promotional Committee 2 – Training.

It summarizes the results of the PLCopen Promotional Committee meetings, containing contributions of its members as well as external sources:

Name	***Company***
Rene Simon	Hochschule Harz
Wolfgang Doll	Codesys
Yves de la Broise	IntervalZero
Anders Lekve Brandseth	Framo
Daniel Wall	Eaton
Filippo Venturi	SACMI
Georg Rempfler	Wyon
John Dixon	ABB
Dominik Franz	ABB
Ralf Dreesen	Beckhoff
Saele Beltrani	SACMI
Yo Takahashi	Mitsubishi Electric
Juliane Fischer	TUM
Eelco van der Wal	PLCopen

Change Status List:

Version number	*Date*	*Change comment*
V 0.1	May 28, 2019	Document created by EvdW as input for the webmeeting
V 0.2	June 19, 2019	As result of the webmeeting June 11
V 0.3	July 2, 2019	As result of the webmeeting on July 2
V 0.4	July 30,2019	As result of the webmeeting on July 30
V 0.5	Oct. 30, 2019	As a result of the discussions upfront and webmeeting
V 0.6	Feb. 2, 2021	Conversion to boiler demo example
V0.6c	March 5, 2021	Added examples and text in Ch. 3
V0.7	April 22, 2021	After feedback and webmeeting
V0.8	May 6, 2021	After feedback and webmeeting
V0.9	May 20, 2021	After additional feedback and decisions at webmeeting
V0.99	May 27, 2021	Published as Release for Comments to the public
V1.0	Nov. 18, 2021	As a result of the feedback and webmeeting on Nov. 17

Contents

List of figures

List of abbreviations:

FB	Function Block conform IEC 61131-3
FC	Flow Controller
FD	Feedwater Drum
FT	Flow Transfer
LY	Level Yield
LC	Level Controller
OOP	Object-Oriented Programming
OT	Operation Technology
PID	Proportional, Integral Derivative control algorithm
PLC	Programmable Logic Controller

1 Introduction to this document

The 3rd edition of the IEC 61131-3 standard makes the usage of Object-Oriented Programming, OOP, possible. Parallel to this, PLCopen has defined the concept of function block libraries including OOP and the PLCopen Common Behaviour Model (see on the PLCopen Website "Creating PLCopen Compliant Libraries V1_0" dated May 4, 2017), as well as sets of function blocks for motion control, safety, and communication.

Starting with OOP, several choices need to be made right from the beginning: are all function blocks in one Class? Think about the PLCopen Motion Control function blocks as an example, do we need then the AxisRef as the reference to the axis or even the MC_ as part of the FB name? How do they contain the methods? And will we use only methods, or also direct access to variables? Is the state machine for the axis controlled by the methods? Are all axes' objects with methods, and we access them only via these methods? How about interfaces? And do we prefer composition above inheritance?

All these choices give a different look & feel to the users across the different systems, different training guidelines and differences in maintenance. And this is where PLCopen wants to help and give guidance to create a more homogeneous programming methodology.

1.1. Goals of this working group

Overall, there is little information on how to use OOP for industrial control or the operation technology (OT). The goal of this group is to help here with the following recommendations:

- Guidance to using OOP in addition to the "classical" way.
- Provide the same look & feel in using OOP across the different platforms and implementations.
- Create generic design patterns for industrial control programming.
- The classical programming way should be possible to be used in addition to the OOP way (e.g. this can mean that we have to extend the classical FBs with interfaces, methods, properties, and maybe input and outputs).

As example, a boiler demo will be used to represent the different forms of programming.

2 Introduction to OOP features of the 3rd edition of the IEC 61131-3 standard.

2.1. The 3rd edition of IEC 61131-3

Modern programming environments refer in many cases to OOP which is included in Python, C++, Objective-C, Smalltalk, Delphi, Java, Swift, C#, Perl, Ruby and PHP.

Important aspects of the 3rd Edition of the IEC 61131-3 standard are the integration of object-oriented features. This includes Classes, including Methods and Interfaces, OOP features for Function blocks, and Namespaces.

The 3rd edition of standard IEC 61131-3 – Programming Languages is approved as International Standard. This means that this edition is now official and is available as International Standard at www.IEC.ch . This standard is fully upwards compatible to IEC 61131-3, 2003 (2nd edition).

The inclusion of these extensions allows to implement PLC code with the same proven concepts and best practices as the ones used in (object oriented) software development using higher level languages since decades. These concepts yield easier readable, more modular, and finally more maintainable code. Moreover, by becoming more modern and similar to the mentioned highly distributed languages, a whole pool of software engineers may be attracted into the field of PLC development. And therewith, PLC code may become a first-class citizen in the software portfolio of enterprises.

Note that references are made to the IEC 61131-3 standard itself as is available at www.IEC.ch. These used references are for explanation only and for completeness the standard itself should be bought.

2.2. Definitions in the IEC 61131-3 standard

With these OOP extensions, there are several changes or new definitions in comparison to the 2nd edition. For instance, a program organization unit, originally defined as function, function block, or program, is extended to include a category "class".

Function blocks and classes may contain methods. Moreover, they are inheritable what leads to the concepts of base types, derived function blocks and derived classes, respectively.

The most important new definitions are listed here:

Name	Description
base type	data type, function block type or class from which further types are inherited/derived.
call	language construct causing the execution of a function, function block, or method.
class	program organization unit consisting of: • the definition of a data structure, • a set of methods (like subroutines) to be performed upon the data structure A class is an implementation— a concrete data structure and collection of subroutines— while a type is an interface.
derived class	class created by inheritance from another class. Note 1 to entry: Derived class is also named extended class or child class. Note 2: this is of course very much in line with derived data type and derived function block type as defined in the IEC 61131-3-2003.

dynamic binding	situation in which the instance of a method call is retrieved during runtime according to the actual type of an instance or interface.
inheritance	creation of a new class, function block type or interface based on an existing class, function block type or interface, respectively.
input variable	variable which is used to supply a value to a program organization unit except for **class.**
instance	individual, named copy of the data structure associated with a function block type, class, or program type, which keeps its values from one call of the associated operations to the next.
interface	language element in the context of OOP containing a set of method prototypes. Example: it is similar to a motor flange: it describes holes diameter, distance, shaft size, but it is not a motor.
method	language element similar to a function that can only be defined in the scope of a function block or class type and with implicit access to instance variables of the function block instance or class instance. Example: a boiler can have a Fill method, a HeatUp method, each one performing a specific task.
override	keyword used with a method in a derived class or function block type for a method with the same signature as a method of the base class or function block type using a new method body
output variable	variable which is used to return a value from the program organization unit except for **classes.**
program organization unit	function, function block**, class,** or program.
signature	set of information defining unambiguously the identity of the parameter interface of a METHOD consisting of its name and the names, types, and order of all its parameters (i.e., inputs, outputs, in-out variables, and result type).

2.3. Differences in OOP languages

There are other OOP languages, and the table hereunder gives a short overview of the commonalities and differences.

Language Properties	*IEC 61131-3 2nd edition*	*IEC 61131-3 3rd Edition*	*C++*	*Java*	*C#*
Multi languages	+	+	-	-	-
OOP/procedural mixed	-	+	+	-	-
Classes	~ (FB)	+	+	+	+
Methods	~ (Actions)	+	+	+	+
Interfaces	-	+	-	+	+
Polymorphism	-	+	+/-	+	+
Semantic Reference	-	+ (Interfaces)	-	+	+
Constructor/Destructor	-/+	-/+	+	+	+
Properties	-	-	-	-	+
Dyn. Memory („new“)	-	-	+	+	+
Access control	~ (Variables)	~ (Variables)	+	+	+

3 Generic Approach

3.1. General

The PLCopen document "Guidelines for usage of Object Orientation" defines a set of currently generally accepted rules for good OOP design in the context of the IEC61131-3 3rd Edition.

It is advised to read the document PLCopen Software Creation Guidelines: Creating PLCopen Compliant Libraries, Version 1.0 of May 4, 2017, which is downloadable from the PLCopen website. As part of this, we will use the SOLID principles in the automation technology environment. SOLID stands for:

- **S**RP — Single Responsibility Principle
- **O**CP — Open/Closed Principle
- **L**SP — Liskov Substitution Principle
- **I**PS — Interface Segregation Principle
- **D**IP — Dependency Inversion Principle

3.2. Modules and Commands as Objects

When designing applications and libraries in an OOP way, we differentiate between modules and commands.

- A *module* represents a part of an automation application including their software function and can also be nested hierarchically as required. For example, actors, sensors, or component assemblies of machines may be represented in code by "modules".
- A *command* represents a concrete action of a module. It is assigned to one module or to a group of modules that implement the same interface.

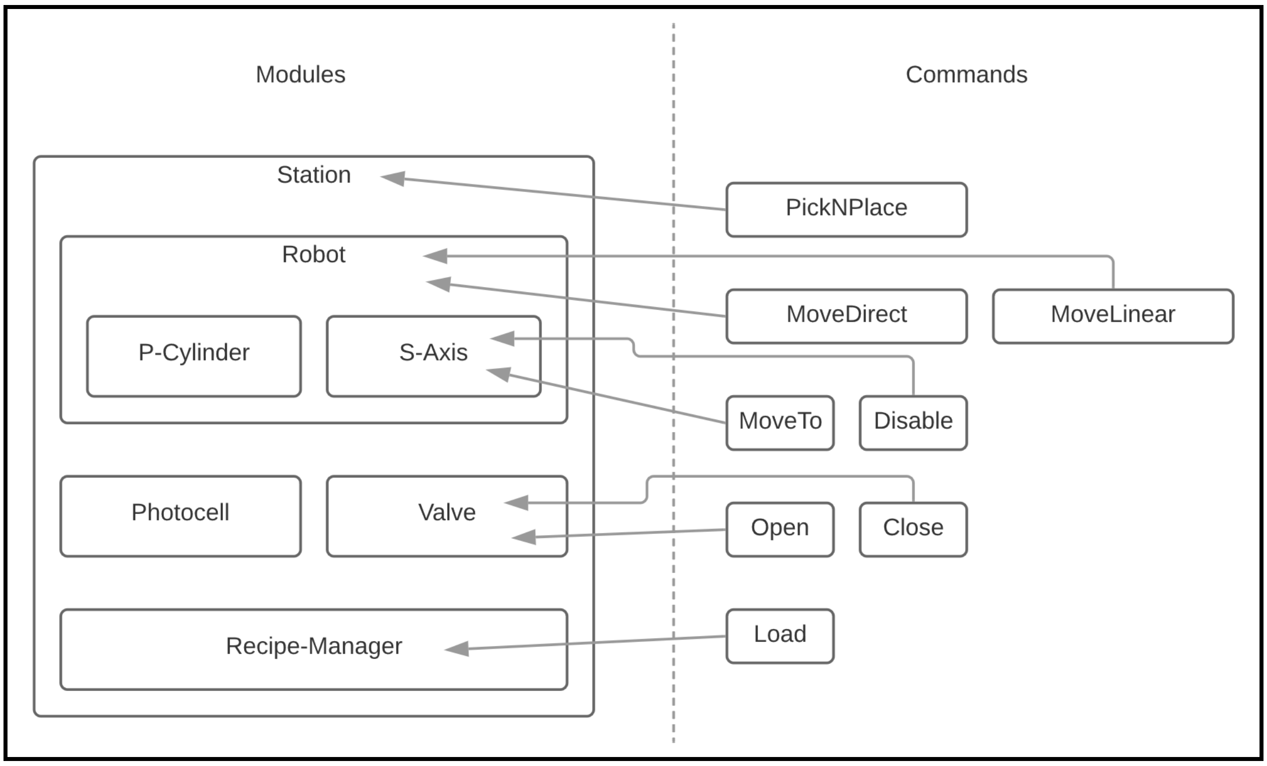


Figure 1: Representation of IModules and ICommands

Every module is implementing a generic *IModule* interface.
Some modules implement additionally a more specialized interface. For example, the modules that provide the functionality of an axis, are implementing an *IAxis* interface.

Every command will implement a generic *ICommand* interface and will follow the design guidelines for PLCopen conforming function blocks.
The axis related commands provide an input of type *IAxis* to get the reference to a related axis instance they will act upon.

With this construction it is possible to bring two worlds reconcilable together.
For example, the classic PLCopen Motion function blocks can be reused in their traditional form. At the same time, these function blocks can also be controlled via their modern interface. This opens new possibilities for innovative solutions such as reconfiguring applications at runtime.

4 The basic example: boiler demo

4.1. Overview of the different phases

The original example was used for a demonstration at the Hanover Fair in cooperation with the OPC Foundation.

This original code is transferred into a format that is compliant with the PLCopen Guidelines, so based on the harmonized function block interfaces.

The next step is to convert this to Object Orientation, which is followed by the addition of an error handler and alarm management.

The program is written in Structured Text and the code can be found on the PLCopen website as a Codesys project.

4.2. Application description

The application of the example “Boiler Demo” consists of a simulated boiler as shown in Figure 2: The boiler demo.

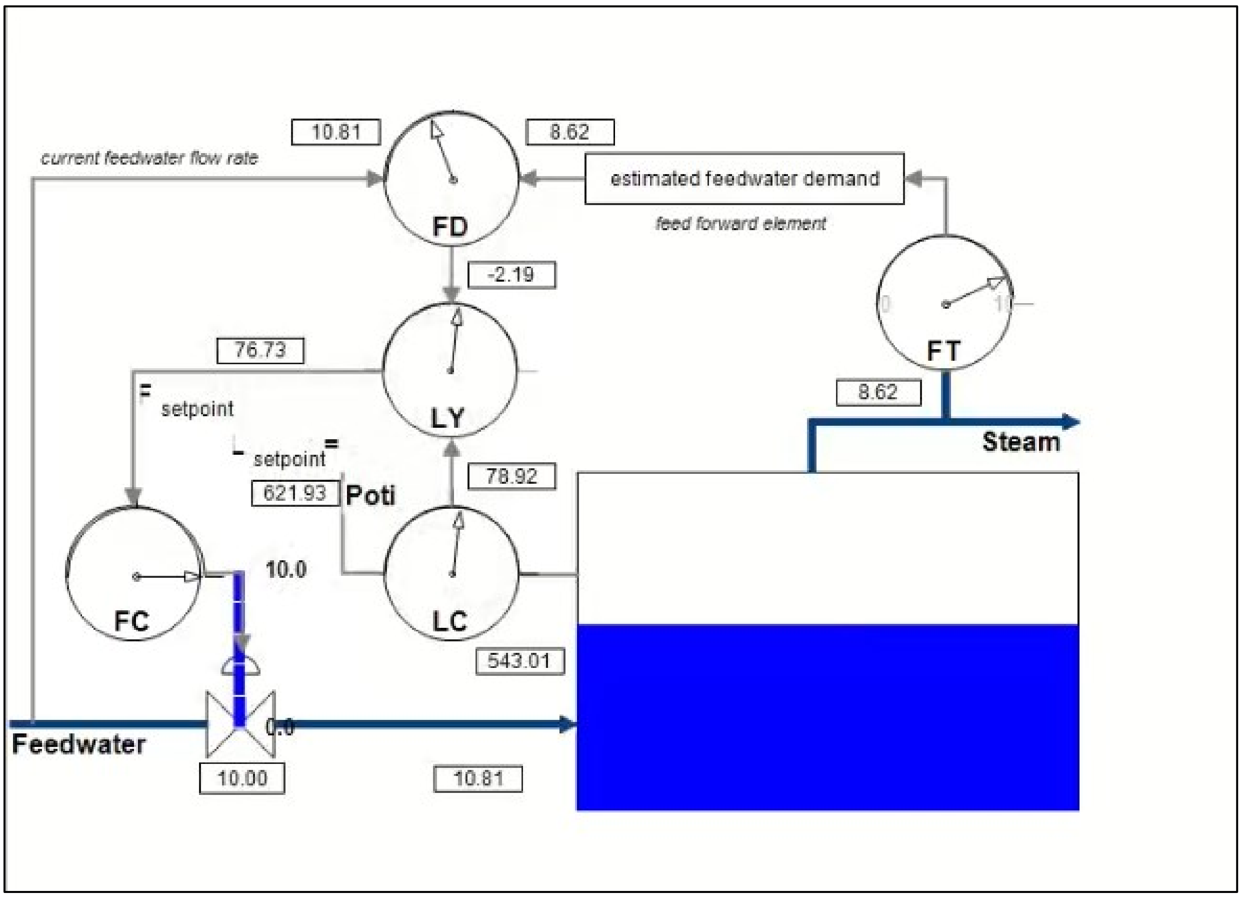


Figure 2: The boiler demo.

4.3. The example in a classical program style

The application simulates a boiler process with a basic PID controller supported by flow control (FC) and level control (LC) including a random generator for disturbances in the feeds.

There are two programs, 4 user derived functions and 17 user derived function blocks.

PROGRAM - MAIN & P_Initialisation

Defined FUNCTIONS

F_FeedwaterEstimator
F_AlignValues
F_IntToReal
F_RealToInt

Defined FUNCTION BLOCKS

FB_AutomaticMode
FB_InputParameter
FB_DRAND
FB_BasicPIDCtrl
FB_Boiler
FB_OutputPipe
FB_SteamFlow
FB_InputPipe
FB_Valve
FB_FeedwaterRate
FB_Drum
FB_FillLevel
FB_Potentiometer
FB_Motor
FB_LevelController
FB_FlowController
FB_ControlComputation

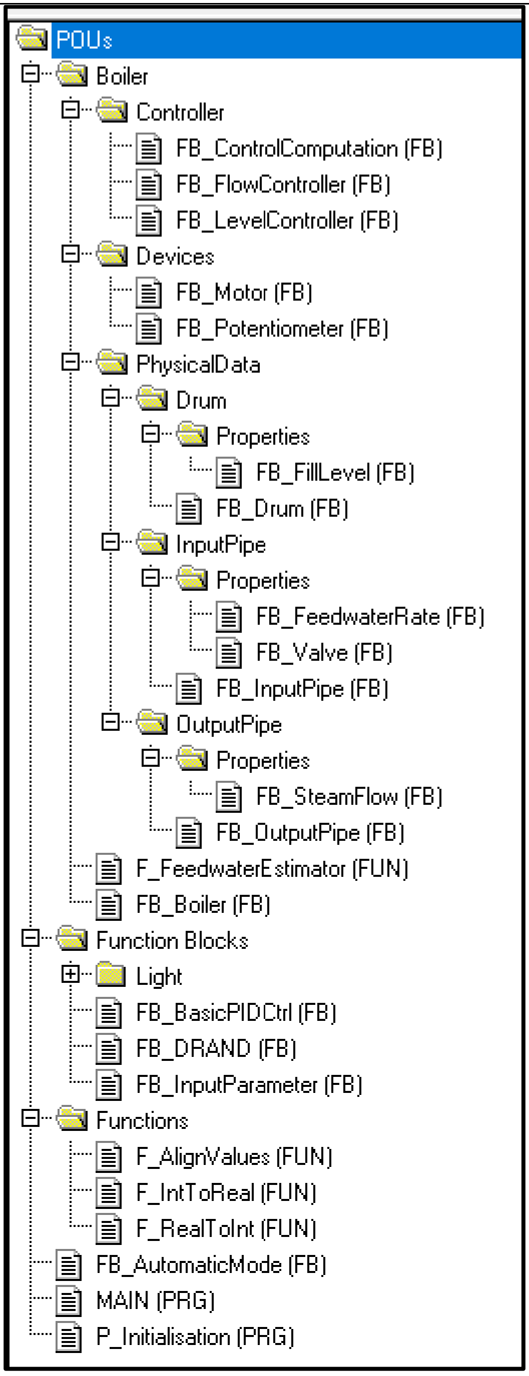

```
POUs
  Boiler
    Controller
      FB_ControlComputation (FB)
      FB_FlowController (FB)
      FB_LevelController (FB)
    Devices
      FB_Motor (FB)
      FB_Potentiometer (FB)
    PhysicalData
      Drum
        Properties
          FB_FillLevel (FB)
        FB_Drum (FB)
      InputPipe
        Properties
          FB_FeedwaterRate (FB)
          FB_Valve (FB)
        FB_InputPipe (FB)
      OutputPipe
        Properties
          FB_SteamFlow (FB)
        FB_OutputPipe (FB)
    F_FeedwaterEstimator (FUN)
    FB_Boiler (FB)
  Function Blocks
    Light
    FB_BasicPIDCtrl (FB)
    FB_DRAND (FB)
    FB_InputParameter (FB)
  Functions
    F_AlignValues (FUN)
    F_IntToReal (FUN)
    F_RealToInt (FUN)
  FB_AutomaticMode (FB)
  MAIN (PRG)
  P_Initialisation (PRG)
```

Figure 3: Layout of the POUs

4.4. Converting the program to a PLCopen compliant version

The design is already well structured with regards to the first SOLID rule "Single Responsibility Principle". There is a separate FB for every hardware element, as well as for every new task.

Unfortunately, this application has not yet been designed according to an OOP approach, so the other rules are not adhered to.

There is no interface and there is no abstraction. All FBs were programmed and used directly on the detail level.

There are two possible approaches to adapting this application:

1. A complete redesign of the application according to OOP approaches and rules. An exceptionally good architecture can be achieved in this way.

Unfortunately, this solution is very time-consuming and expensive, which makes it difficult for a machine builder to get a completely new software generation approved by the management.

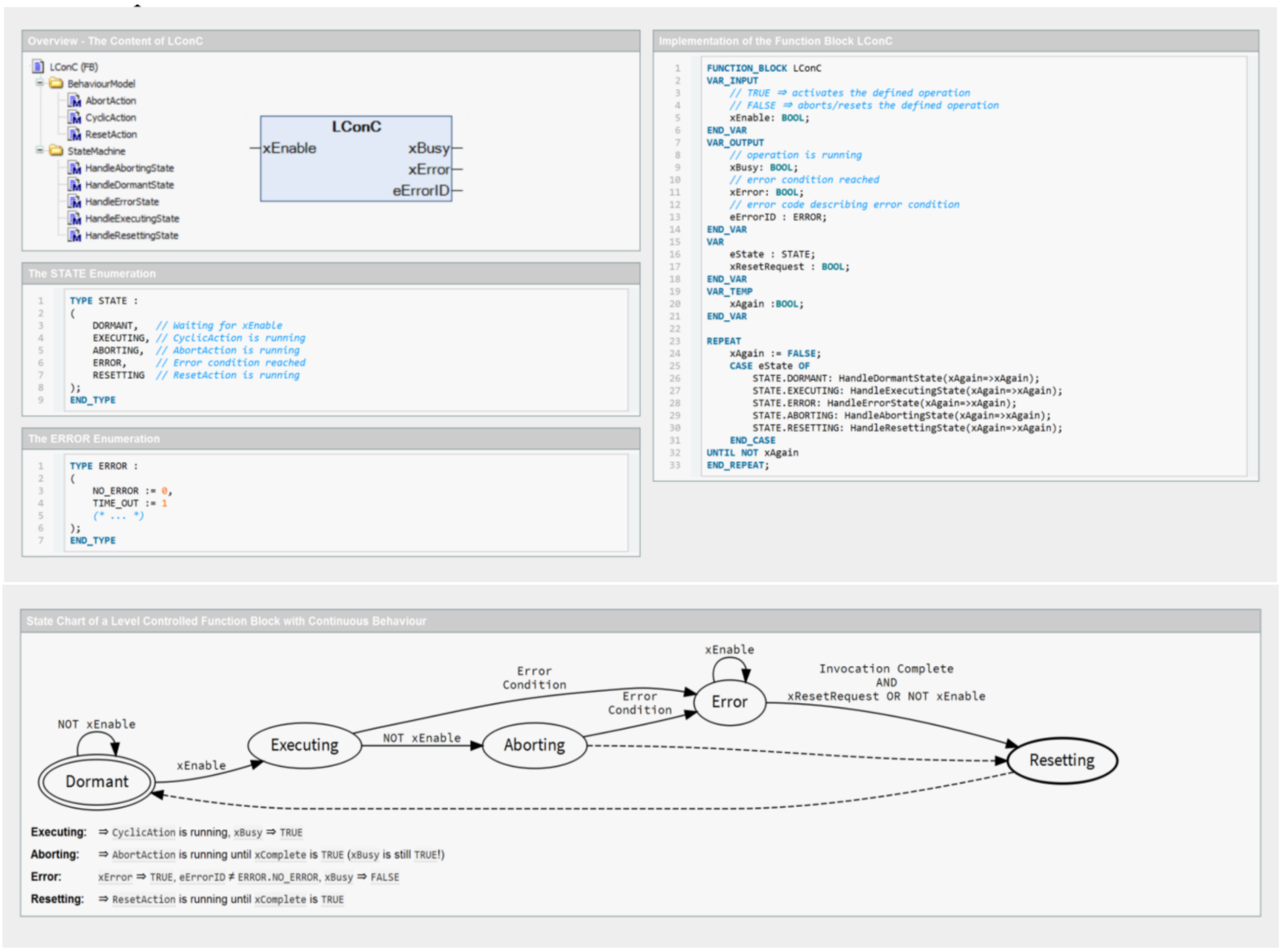
2. Therefore, we try to present a second possibility to partially restore the application. Due to the good starting point, this step can be considered. For (brownfield) applications that have grown over many years and are too nested, it is better to rely on a new generation.

The original building blocks were programmed on the lowest level of detail. The first step is to achieve standardization and abstraction of the building blocks. The PLCopen guidelines for function block libraries offer a clear interface and standardized behaviour for this purpose. The essence of the guidelines is the so-called "behaviour model". So, the first step was to convert the original program including the defined function and function blocks into the applicable behaviour model, including the state machine and the error behaviour. The OOP structure of the PLCopen Common Behaviour Model offers us excellent opportunities to implement this in just a few steps and changes.

Also, we try to use the “Open/Closed Principle” and “Dependency Inversion Principle”. In addition, a first interface is defined which can be used for future extensions.

Because the building blocks can be extended by the defined behaviour models, in this case the LConC is used, they get an abstract higher level that does not depend on the details of the lower level. The LConC also defines an interface which must now be implemented by the old FB’s. In the first step, however, only the code from the body of the old FB’s has to be copied into the Cyclic Action method.

As a result, the “Open/Closed Principle” is adhered to, the block retains its functionality, and the existing software code does not have to be changed. This also gives classic style software developers the opportunity to slowly get used to the changeover. Since the FB’s are still 1-to-1 identical, and the implementation has not been changed significantly either. But the modules are now open for expansion. Also, incorrect user entries and parameterization can now be monitored in the StartAction (as used in the edge triggered function blocks as defined in the PLCopen library document), in order to increase the stability. In addition, the modules now have an abstract and uniform interface with which further extensions can be made that follow the “Interface Segregation Principle”.



```
TYPE STATE :
(
    DORMANT,   // Waiting for xEnable
    EXECUTING, // CyclicAction is running
    ABORTING,  // AbortAction is running
    ERROR,     // Error condition reached
    RESETTING  // ResetAction is running
);
END_TYPE
```

```
TYPE ERROR :
(
    NO_ERROR := 0,
    TIME_OUT := 1
    (* ... *)
);
END_TYPE
```

```
FUNCTION_BLOCK LConC
VAR_INPUT
    // TRUE ⇒ activates the defined operation
    // FALSE ⇒ aborts/resets the defined operation
    xEnable: BOOL;
END_VAR
VAR_OUTPUT
    // operation is running
    xBusy: BOOL;
    // error condition reached
    xError: BOOL;
    // error code describing error condition
    eErrorID : ERROR;
END_VAR
VAR
    eState : STATE;
    xResetRequest : BOOL;
END_VAR
VAR_TEMP
    xAgain :BOOL;
END_VAR

REPEAT
    xAgain := FALSE;
    CASE eState OF
        STATE.DORMANT: HandleDormantState(xAgain=>xAgain);
        STATE.EXECUTING: HandleExecutingState(xAgain=>xAgain);
        STATE.ERROR: HandleErrorState(xAgain=>xAgain);
        STATE.ABORTING: HandleAbortingState(xAgain=>xAgain);
        STATE.RESETTING: HandleResettingState(xAgain=>xAgain);
    END_CASE
UNTIL NOT xAgain
END_REPEAT;
```

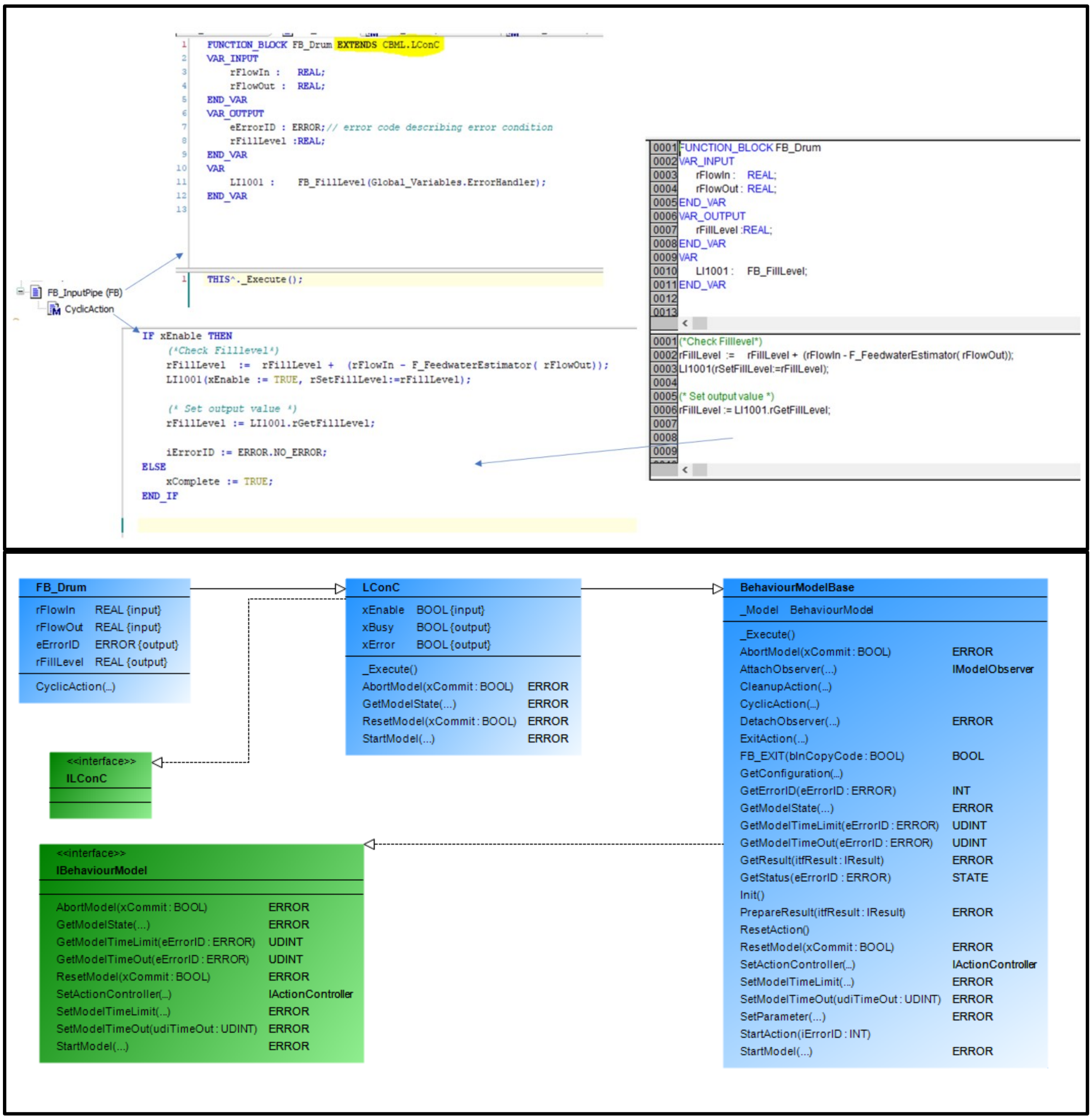
Executing: ⇒ `CyclicAction` is running, `xBusy` ⇒ `TRUE`

Aborting: ⇒ `AbortAction` is running until `xComplete` is `TRUE` (`xBusy` is still `TRUE`!)

Error: `xError` ⇒ `TRUE`, `eErrorID` ≠ `ERROR.NO_ERROR`, `xBusy` ⇒ `FALSE`

Resetting: ⇒ `ResetAction` is running until `xComplete` is `TRUE`

Figure 4: Function block interface and state diagram of LConC



```
FUNCTION_BLOCK FB_Drum EXTENDS CBML.LConC
VAR_INPUT
    rFlowIn :   REAL;
    rFlowOut :  REAL;
END_VAR
VAR_OUTPUT
    eErrorID : ERROR;// error code describing error condition
    rFillLevel :REAL;
END_VAR
VAR
    LI1001 :    FB_FillLevel(Global_Variables.ErrorHandler);
END_VAR
```

```
THIS^._Execute();
```

```
IF xEnable THEN
    (*Check Filllevel*)
    rFillLevel  :=  rFillLevel +  (rFlowIn - F_FeedwaterEstimator( rFlowOut));
    LI1001(xEnable := TRUE, rSetFillLevel:=rFillLevel);

    (* Set output value *)
    rFillLevel := LI1001.rGetFillLevel;

    iErrorID := ERROR.NO_ERROR;
ELSE
    xComplete := TRUE;
END_IF
```

```
FUNCTION_BLOCK FB_Drum
VAR_INPUT
    rFlowIn :   REAL;
    rFlowOut :  REAL;
END_VAR
VAR_OUTPUT
    rFillLevel :REAL;
END_VAR
VAR
    LI1001 :    FB_FillLevel;
END_VAR
```

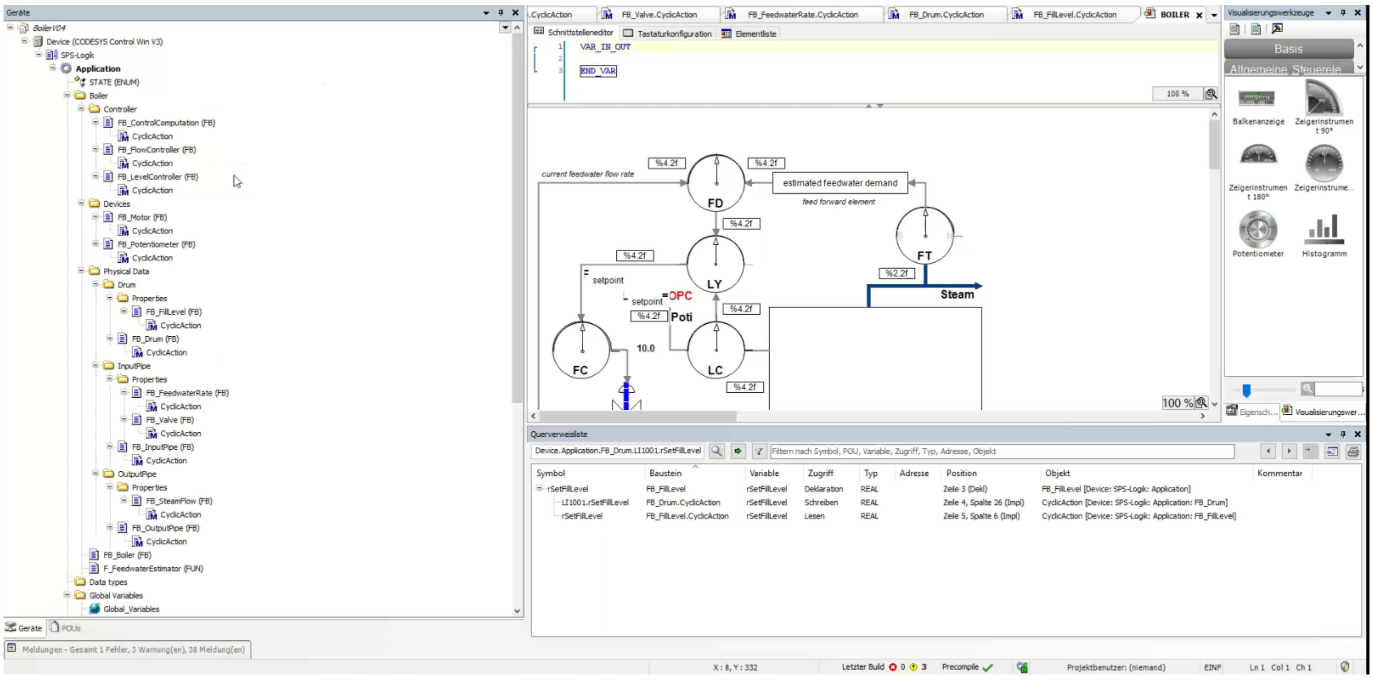
```
(*Check Filllevel*)
rFillLevel  :=    rFillLevel +  (rFlowIn - F_FeedwaterEstimator( rFlowOut));
LI1001(rSetFillLevel:=rFillLevel);

(* Set output value *)
rFillLevel := LI1001.rGetFillLevel;
```

FB_InputPipe (FB)
CyclicAction
FB_Drum
rFlowIn REAL {input}
rFlowOut REAL {input}
eErrorID ERROR {output}
rFillLevel REAL {output}
CyclicAction(...)
LConC
xEnable BOOL {input}
xBusy BOOL {output}
xError BOOL {output}
_Execute()
AbortModel(xCommit : BOOL) ERROR
GetModelState(...) ERROR
ResetModel(xCommit : BOOL) ERROR
StartModel(...) ERROR
BehaviourModelBase
_Model BehaviourModel
_Execute()
AbortModel(xCommit : BOOL) ERROR
AttachObserver(...) IModelObserver
CleanupAction(...)
CyclicAction(...)
DetachObserver(...) ERROR
ExitAction(...)
FB_EXIT(bInCopyCode : BOOL) BOOL
GetConfiguration(...)
GetErrorID(eErrorID : ERROR) INT
GetModelState(...) ERROR
GetModelTimeLimit(eErrorID : ERROR) UDINT
GetModelTimeOut(eErrorID : ERROR) UDINT
GetResult(itfResult : IResult) ERROR
GetStatus(eErrorID : ERROR) STATE
Init()
PrepareResult(itfResult : IResult) ERROR
ResetAction()
ResetModel(xCommit : BOOL) ERROR
SetActionController(...) IActionController
SetModelTimeLimit(...) ERROR
SetModelTimeOut(udiTimeOut : UDINT) ERROR
SetParameter(...) ERROR
StartAction(iErrorID : INT)
StartModel(...) ERROR
<<interface>>
ILConC
<<interface>>
IBehaviourModel
AbortModel(xCommit : BOOL) ERROR
GetModelState(...) ERROR
GetModelTimeLimit(eErrorID : ERROR) UDINT
GetModelTimeOut(eErrorID : ERROR) UDINT
ResetModel(xCommit : BOOL) ERROR
SetActionController(...) IActionController
SetModelTimeLimit(...) ERROR
SetModelTimeOut(udiTimeOut : UDINT) ERROR
StartModel(...) ERROR

Figure 5: Same interfaces for different devices

The consistent design of the Common Behaviour Model enables us to create an abstraction that is even independent of the state machine chosen. As all state machines, it is derived from an abstract class with a general defined interface.

This now offers the possibility to manage the FBs in a uniform way at a higher level. For example, to implement an error handler that can manage all FBs regardless of their implementation (Interface Segregation Principle).



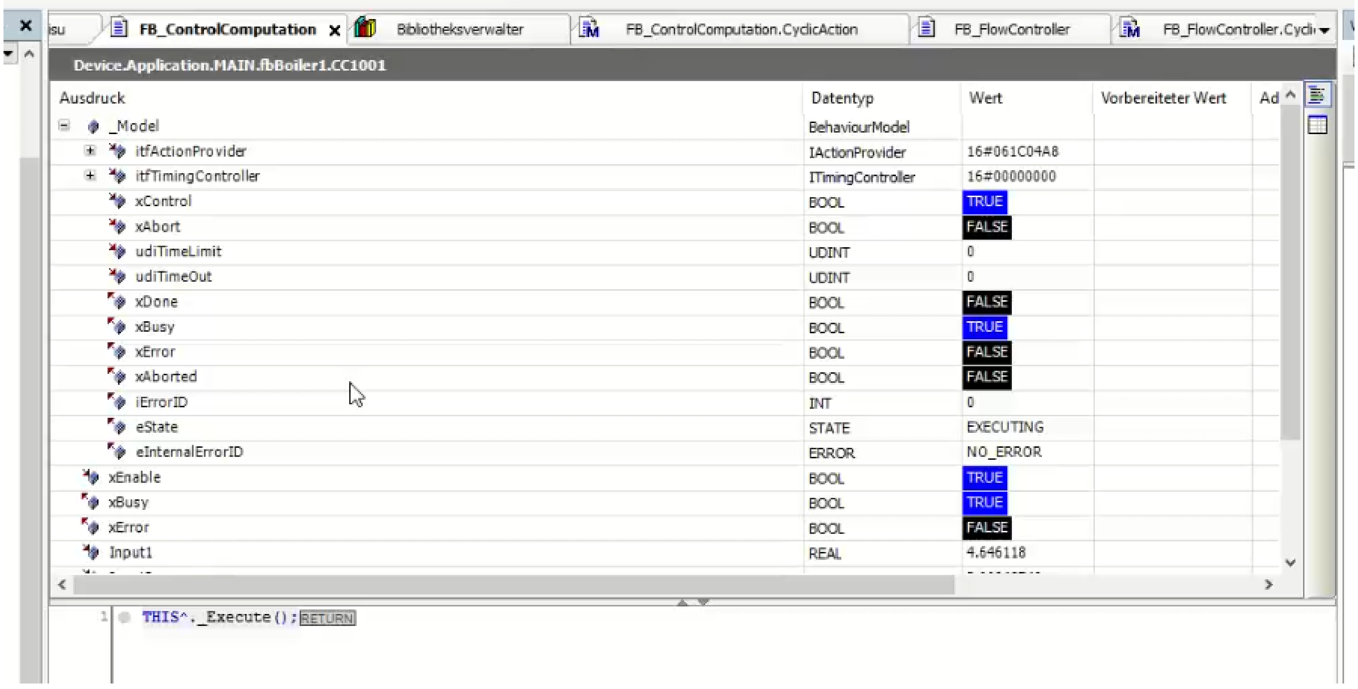
```
Boiler04
  Device (CODESYS Control Win V3)
    SPS-Logik
      Application
        STATE (ENUM)
        Boiler
          Controller
            FB_ControlComputation (FB)
              CyclicAction
            FB_FlowController (FB)
              CyclicAction
            FB_LevelController (FB)
              CyclicAction
          Devices
            FB_Motor (FB)
              CyclicAction
            FB_Potentiometer (FB)
              CyclicAction
          Physical Data
            Drum
              Properties
                FB_FillLevel (FB)
                  CyclicAction
              FB_Drum (FB)
                CyclicAction
            InputPipe
              Properties
                FB_FeedwaterRate (FB)
                  CyclicAction
                FB_Valve (FB)
                  CyclicAction
              FB_InputPipe (FB)
                CyclicAction
            OutputPipe
              Properties
                FB_SteamFlow (FB)
                  CyclicAction
              FB_OutputPipe (FB)
                CyclicAction
          FB_Boiler (FB)
          F_FeedwaterEstimator (FUN)
        Data types
        Global Variables
          Global_Variables
```

```
VAR_IN_OUT
END_VAR
```

Querverweisliste

Device.Application.FB_Drum.LI1001.rSetFillLevel

Symbol	Baustein	Variable	Zugriff	Typ	Adresse	Position	Objekt	Kommentar
rSetFillLevel	FB_FillLevel	rSetFillLevel	Deklaration	REAL		Zeile 3 (Dekl)	FB_FillLevel [Device: SPS-Logik: Application]	
LI1001.rSetFillLevel	FB_Drum.CyclicAction	rSetFillLevel	Schreiben	REAL		Zeile 4, Spalte 26 (Impl)	CyclicAction [Device: SPS-Logik: Application: FB_Drum]	
rSetFillLevel	FB_FillLevel.CyclicAction	rSetFillLevel	Lesen	REAL		Zeile 5, Spalte 6 (Impl)	CyclicAction [Device: SPS-Logik: Application: FB_FillLevel]	



Device.Application.MAIN.fbBoiler1.CC1001

Ausdruck	Datentyp	Wert	Vorbereiteter Wert
_Model	BehaviourModel		
itfActionProvider	IActionProvider	16#061C04A8	
itfTimingController	ITimingController	16#00000000	
xControl	BOOL	TRUE	
xAbort	BOOL	FALSE	
udiTimeLimit	UDINT	0	
udiTimeOut	UDINT	0	
xDone	BOOL	FALSE	
xBusy	BOOL	TRUE	
xError	BOOL	FALSE	
xAborted	BOOL	FALSE	
iErrorID	INT	0	
eState	STATE	EXECUTING	
eInternalErrorID	ERROR	NO_ERROR	
xEnable	BOOL	TRUE	
xBusy	BOOL	TRUE	
xError	BOOL	FALSE	
Input1	REAL	4.646118	

```
THIS^._Execute();RETURN
```

Figure 6: Overview of the states in a simulation

Be aware that the main program here is not usable in real applications as it is missing a start-up sequence, a shutdown sequence, and error handling. This is the part that is customized for every application. These additions will show further benefits of the OOP due to the re-use.

4.5. Adding Error Behaviour

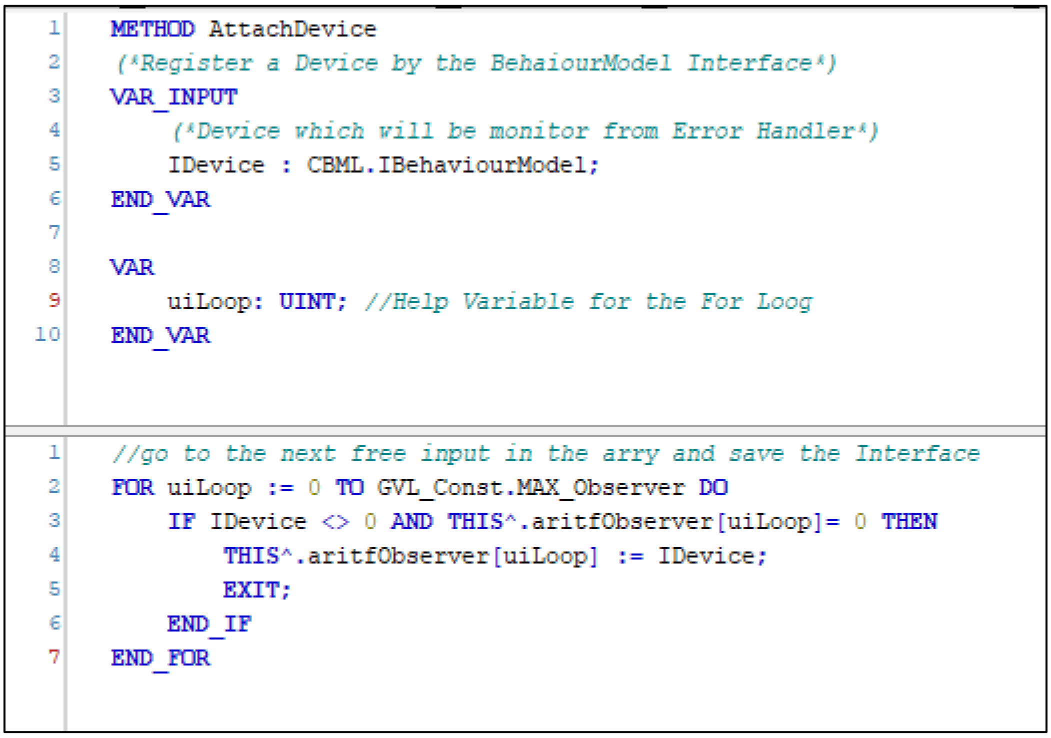
As an example of this, the error behaviour is added as a central error handling function block, which traces all the used function blocks. This is shown in 2 ways: via *inheritance* and via *composition*.

We will not use the different detailed FBs directly, but we will use our abstract Interface IBehaviourModel (see Figure 5:Same interfaces for different devices). This interface gives access to the Outputs from the function blocks which include xDone, xBusy, xError, xAborted, iErrorID. This is the idea from the Interface Segregation Principle, and with this, all the FBs from the Common Model Library can now be connected. This is not corresponding to only one FB-type, nor to one Behaviour Model. All described Behaviours and types can be connected without any adaptation to this ErrorHandler.

The idea is to map all the function blocks to an array which is included in the error handler, to enable an iteration through all the function blocks. In this Array we will not have a Pointer to the different FBs on detail level, but we will have the basic interface IBehaviour Model only. With the Method GetModelState we can monitor the required information from all FB. See Figure 8:The Error Handler FB with the list of the devices (FBs).

The Error Handler has two methods to link it to the devices (FBs): AttachDevice and DetachDevice. The error handler has these two methods for attach and detach, reflecting it in the device list. The error handler references to this list (see Figure 10:The structure for Error Handler).

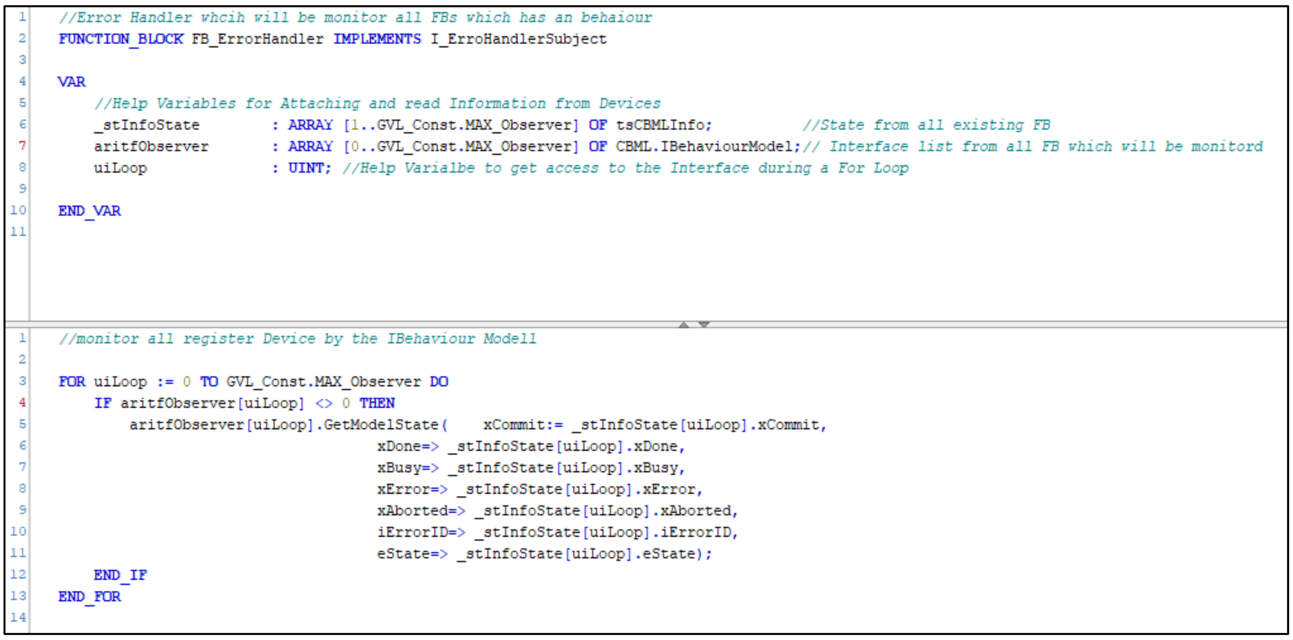
We can now monitor all FBs with the same Method GetModelState and build a structure with Status Information so that one can see the status from all the function blocks.



```
METHOD AttachDevice
(*Register a Device by the BehaiourModel Interface*)
VAR_INPUT
    (*Device which will be monitor from Error Handler*)
    IDevice : CBML.IBehaviourModel;
END_VAR

VAR
    uiLoop: UINT; //Help Variable for the For Loog
END_VAR
```

```
//go to the next free input in the arry and save the Interface
FOR uiLoop := 0 TO GVL_Const.MAX_Observer DO
    IF IDevice <> 0 AND THIS^.aritfObserver[uiLoop]= 0 THEN
        THIS^.aritfObserver[uiLoop] := IDevice;
        EXIT;
    END_IF
END_FOR
```

Figure 7: The AttachDevice Method of the Error handler

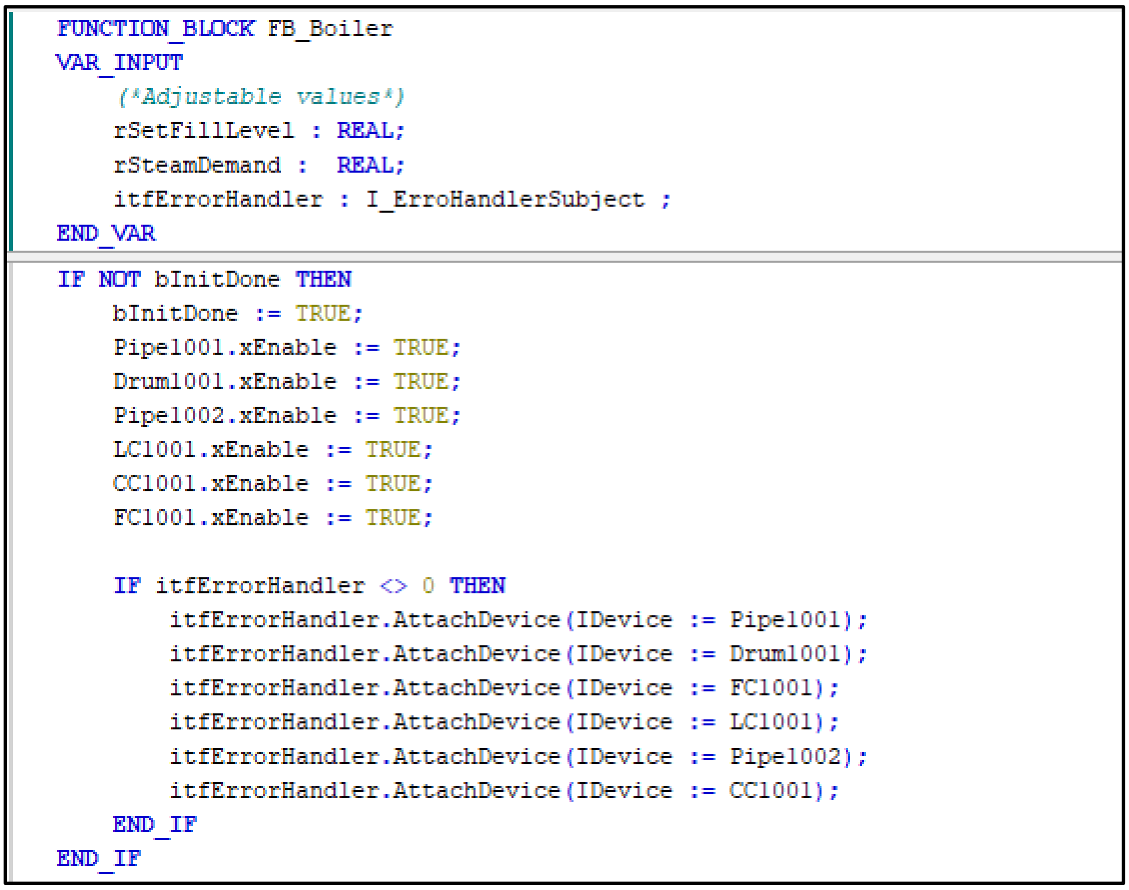


```
//Error Handler whcih will be monitor all FBs which has an behaiour
FUNCTION_BLOCK FB_ErrorHandler IMPLEMENTS I_ErroHandlerSubject

VAR
    //Help Variables for Attaching and read Information from Devices
    _stInfoState        : ARRAY [1..GVL_Const.MAX_Observer] OF tsCBMLInfo;          //State from all existing FB
    aritfObserver       : ARRAY [0..GVL_Const.MAX_Observer] OF CBML.IBehaviourModel;// Interface list from all FB which will be monitord
    uiLoop              : UINT; //Help Varialbe to get access to the Interface during a For Loop

END_VAR
```

```
//monitor all register Device by the IBehaviour Modell

FOR uiLoop := 0 TO GVL_Const.MAX_Observer DO
    IF aritfObserver[uiLoop] <> 0 THEN
        aritfObserver[uiLoop].GetModelState(    xCommit:= _stInfoState[uiLoop].xCommit,
                                    xDone=> _stInfoState[uiLoop].xDone,
                                    xBusy=> _stInfoState[uiLoop].xBusy,
                                    xError=> _stInfoState[uiLoop].xError,
                                    xAborted=> _stInfoState[uiLoop].xAborted,
                                    iErrorID=> _stInfoState[uiLoop].iErrorID,
                                    eState=> _stInfoState[uiLoop].eState);
    END_IF
END_FOR
```

Figure 8: The Error Handler FB with the list of the devices (FBs)

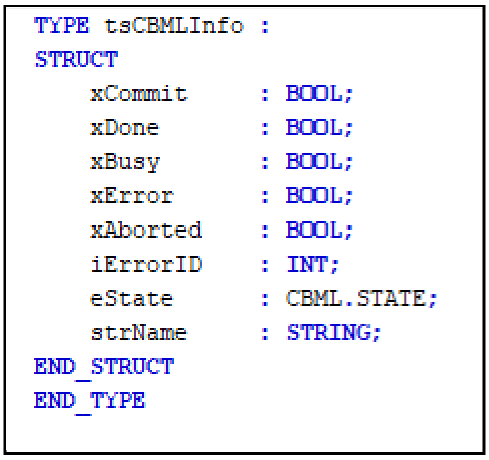
The definition of the Attach and Detach Methods is done in an Interface which is implemented from the Error Handler. This interface can now be used by the Boiler to attach all the instances to the Error Handler.



```
FUNCTION_BLOCK FB_Boiler
VAR_INPUT
    (*Adjustable values*)
    rSetFillLevel : REAL;
    rSteamDemand :  REAL;
    itfErrorHandler : I_ErroHandlerSubject ;
END_VAR
```

```
IF NOT bInitDone THEN
    bInitDone := TRUE;
    Pipe1001.xEnable := TRUE;
    Drum1001.xEnable := TRUE;
    Pipe1002.xEnable := TRUE;
    LC1001.xEnable := TRUE;
    CC1001.xEnable := TRUE;
    FC1001.xEnable := TRUE;

    IF itfErrorHandler <> 0 THEN
        itfErrorHandler.AttachDevice(IDevice := Pipe1001);
        itfErrorHandler.AttachDevice(IDevice := Drum1001);
        itfErrorHandler.AttachDevice(IDevice := FC1001);
        itfErrorHandler.AttachDevice(IDevice := LC1001);
        itfErrorHandler.AttachDevice(IDevice := Pipe1002);
        itfErrorHandler.AttachDevice(IDevice := CC1001);
    END_IF
END_IF
```

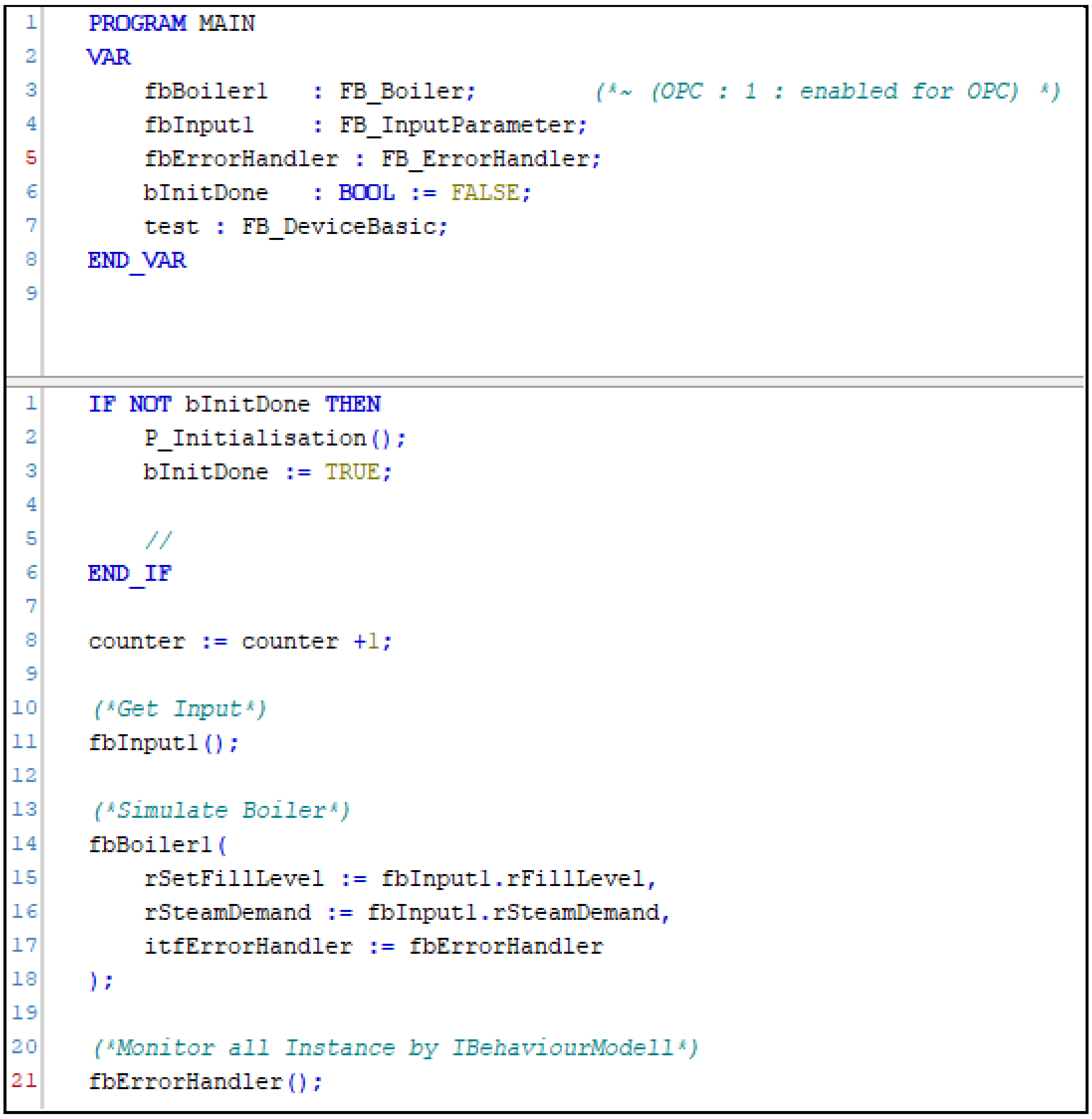
Figure 9: Initialization from the Error Handler in FB_Boiler



```
TYPE tsCBMLInfo :
STRUCT
    xCommit      : BOOL;
    xDone        : BOOL;
    xBusy        : BOOL;
    xError       : BOOL;
    xAborted     : BOOL;
    iErrorID     : INT;
    eState       : CBML.STATE;
    strName      : STRING;
END_STRUCT
END_TYPE
```

Figure 10: The structure for Error Handler

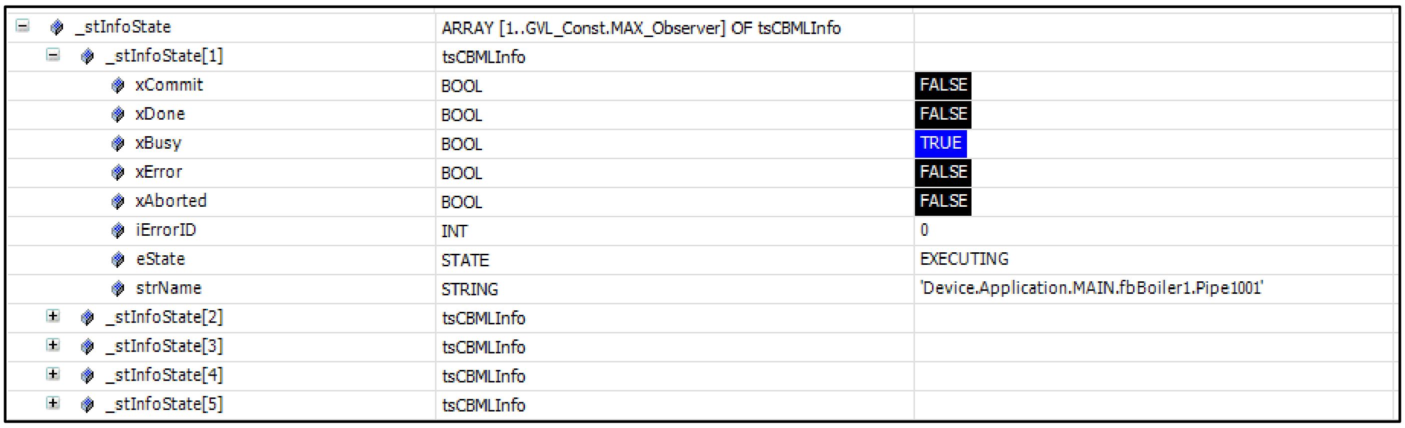
In the main program we can now build an instance from the Error Handler and connect the Error Handler to the Boiler instance:



```
PROGRAM MAIN
VAR
    fbBoiler1    : FB_Boiler;          (*~ (OPC : 1 : enabled for OPC) *)
    fbInput1     : FB_InputParameter;
    fbErrorHandler : FB_ErrorHandler;
    bInitDone    : BOOL := FALSE;
    test : FB_DeviceBasic;
END_VAR
```

```
IF NOT bInitDone THEN
    P_Initialisation();
    bInitDone := TRUE;

    //
END_IF

counter := counter +1;

(*Get Input*)
fbInput1();

(*Simulate Boiler*)
fbBoiler1(
    rSetFillLevel := fbInput1.rFillLevel,
    rSteamDemand := fbInput1.rSteamDemand,
    itfErrorHandler := fbErrorHandler
);

(*Monitor all Instance by IBehaviourModell*)
fbErrorHandler();
```

Figure 11: Main program with error handling



_stInfoState	ARRAY [1..GVL_Const.MAX_Observer] OF tsCBMLInfo	
_stInfoState[1]	tsCBMLInfo	
xCommit	BOOL	FALSE
xDone	BOOL	FALSE
xBusy	BOOL	TRUE
xError	BOOL	FALSE
xAborted	BOOL	FALSE
iErrorID	INT	0
eState	STATE	EXECUTING
strName	STRING	'Device.Application.MAIN.fbBoiler1.Pipe1001'
_stInfoState[2]	tsCBMLInfo	
_stInfoState[3]	tsCBMLInfo	
_stInfoState[4]	tsCBMLInfo	
_stInfoState[5]	tsCBMLInfo	

Figure 12: Example of the error message for one FB

The error handler is now a standard component of all FBs. It handles all the devices centrally without separately calling all the instances of the used FBs, while being able to centrally read all the relevant information of the FBs.

This concept makes it easier to control all the function blocks with one list only. In an alternative way, one must connect to all the FBs and point to all the different available function blocks. Now you have only one interface with one list from which you can control all listed function blocks.

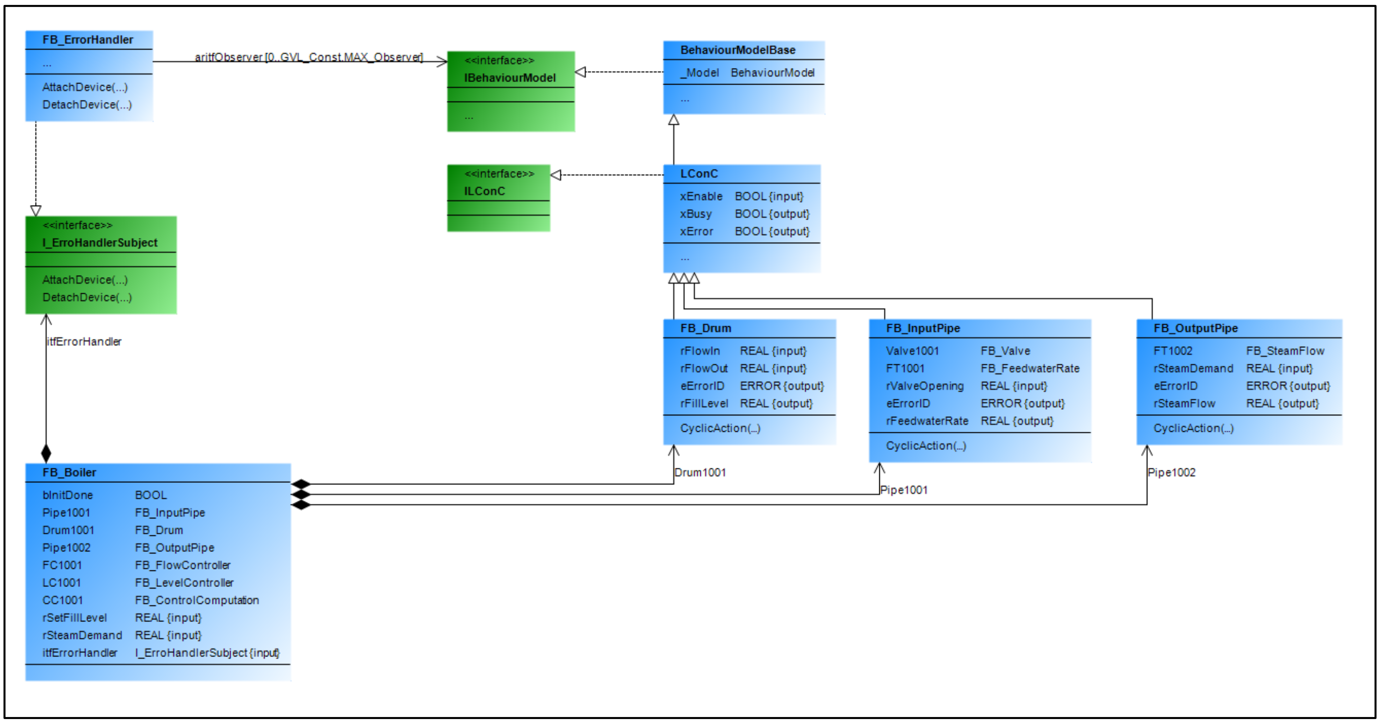


Figure 13: Overview of the architecture

The model in Figure 13: Overview of the architecture shows that LConC (and its interface) inherits from the BehaviourModelIBase and so the interface IBehaviourModel.

The I_ErrorHandler interface is linked to all the FBs which are used in the demo as well as to the FB_ErrorHandler. In this way, the ErrorHandler can connect to all applicable FBs.

Using inheritance as a base design for FBs gives a nice abstract control option but shows some limitations. We are no longer able to extend the detailed device FB on an abstract Level. The reason for this limitation is that one cannot use inheritance on the abstract Level BehaviourModelBase as it uses inheritance already.

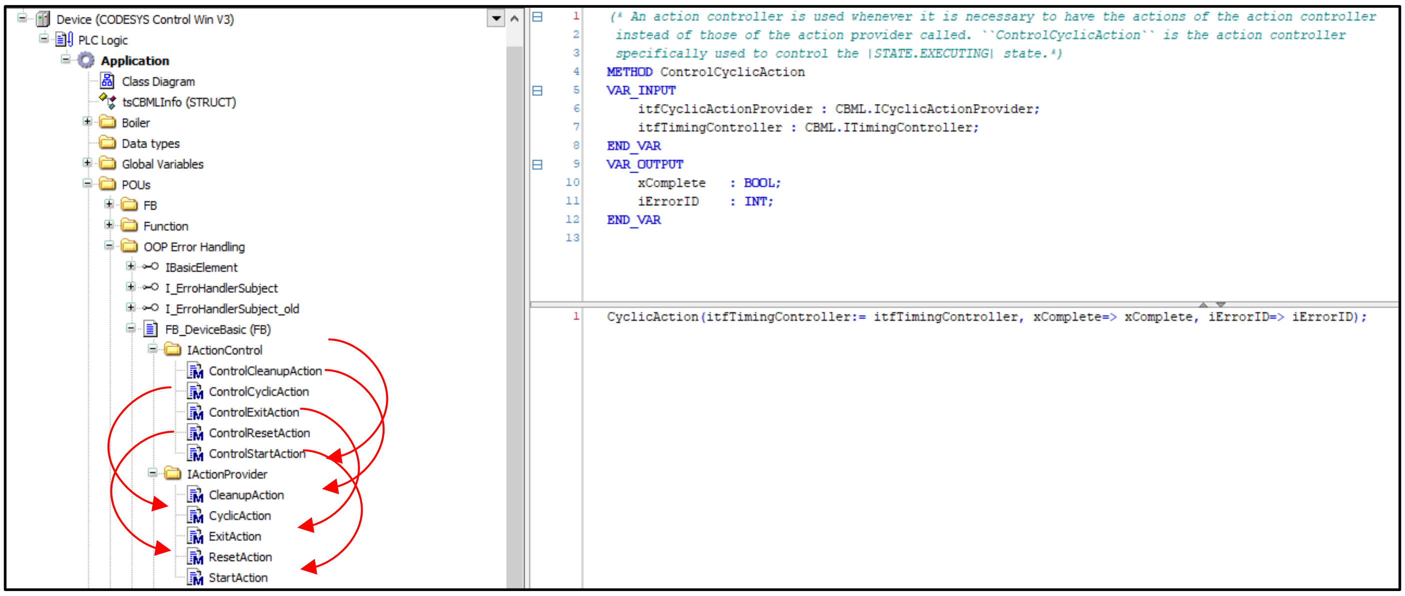
This is the point to check, if we can optimize this design by using ***composition*** instead of ***inheritance*** for the Behaviour Model (see 5.1 Fundamentals for OOP Guidelines – considerations). For this, we must show how we can define the Behaviour Modell as a member of our FBs but define the Method for the different states in the same way as before. (Open/Closed Principle).

A feature from the Common Behaviour Model can help us to implement this design. The FBs in the Library support the connection of an Action Controller - see red lines in figure 16 - which allows a FB which implements the IActionControl to do a Method call instead of the methods which come from the Behaviour Model.

The idea is now to create a new Abstract Device Base Class, which has as a new feature 'Name' from the FB as a complete Path. To support now the Composition from Behaviour Model, our Base Class implements the IActionControl. To follow the "Open/Closed Principle", it is recommended to implement the IActionProvider Interface also. This allows the detailed Device FB to work with the behaviour implementation on the same way as before.

In the Method from the Action Controller, one calls a Method, which comes from the Action Provider.

The following example shows this situation for the method ControlCyclicAction:

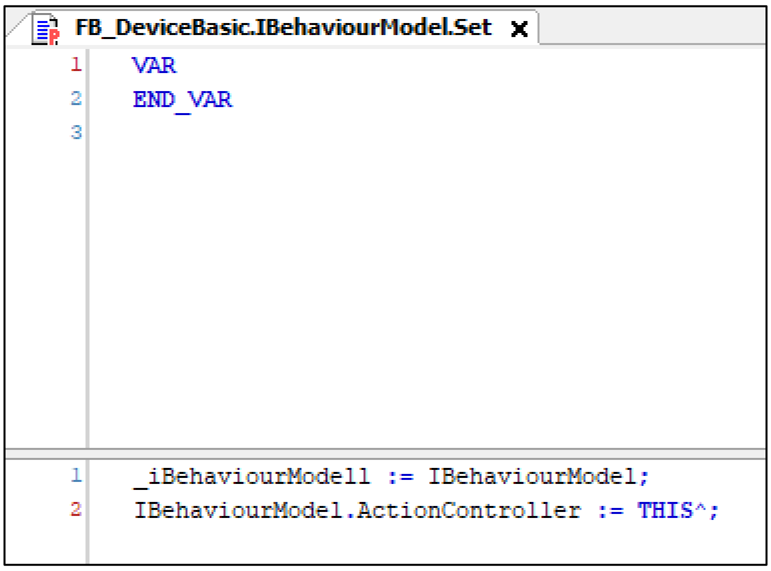


```
(* An action controller is used whenever it is necessary to have the actions of the action controller
 instead of those of the action provider called. ``ControlCyclicAction`` is the action controller
 specifically used to control the |STATE.EXECUTING| state.*)
METHOD ControlCyclicAction
VAR_INPUT
    itfCyclicActionProvider : CBML.ICyclicActionProvider;
    itfTimingController : CBML.ITimingController;
END_VAR
VAR_OUTPUT
    xComplete   : BOOL;
    iErrorID    : INT;
END_VAR
```

```
CyclicAction(itfTimingController:= itfTimingController, xComplete=> xComplete, iErrorID=> iErrorID);
```

Figure 14: Example of a cyclic action linked to the IActionControl

The Base Class will also get a Property to 'get' and 'set' the IBehaviourModel. In the Setter the IActionControl will directly link to the instance of the created Behaviour Model.


FB_DeviceBasic.IBehaviourModel.Set

```
VAR
END_VAR
```

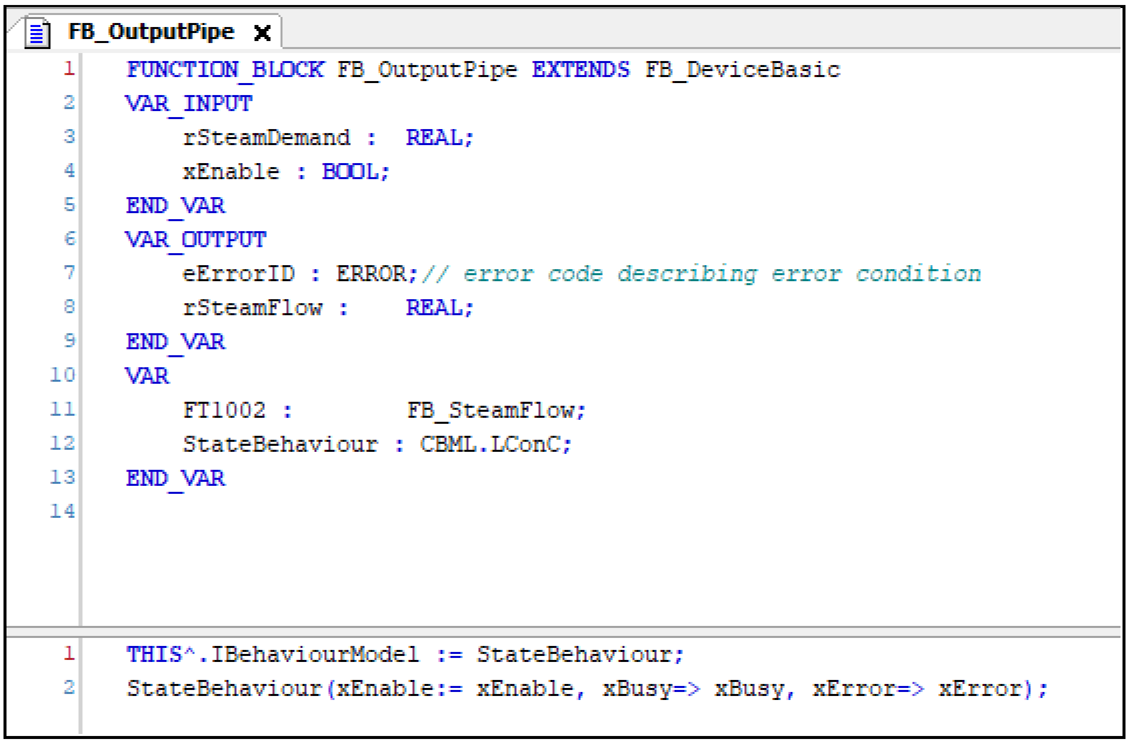
```
_iBehaviourModell := IBehaviourModel;
IBehaviourModel.ActionController := THIS^;
```

Figure 15: Implementation of the Set action of the Behaviour Model

The detailed FB can now decide, which Behaviour Model it will use. The only applicable changes are the following:

- Change the inheritance from the LConC to our own Base Class.
- Create an Input variable for xEnable or xExecute based on your decided Behaviour.
- Create an instance from the Behaviour you want to use.
- Change the Execution from the Model in the Body and give the member Behaviour instance to the Property.

See the following example:

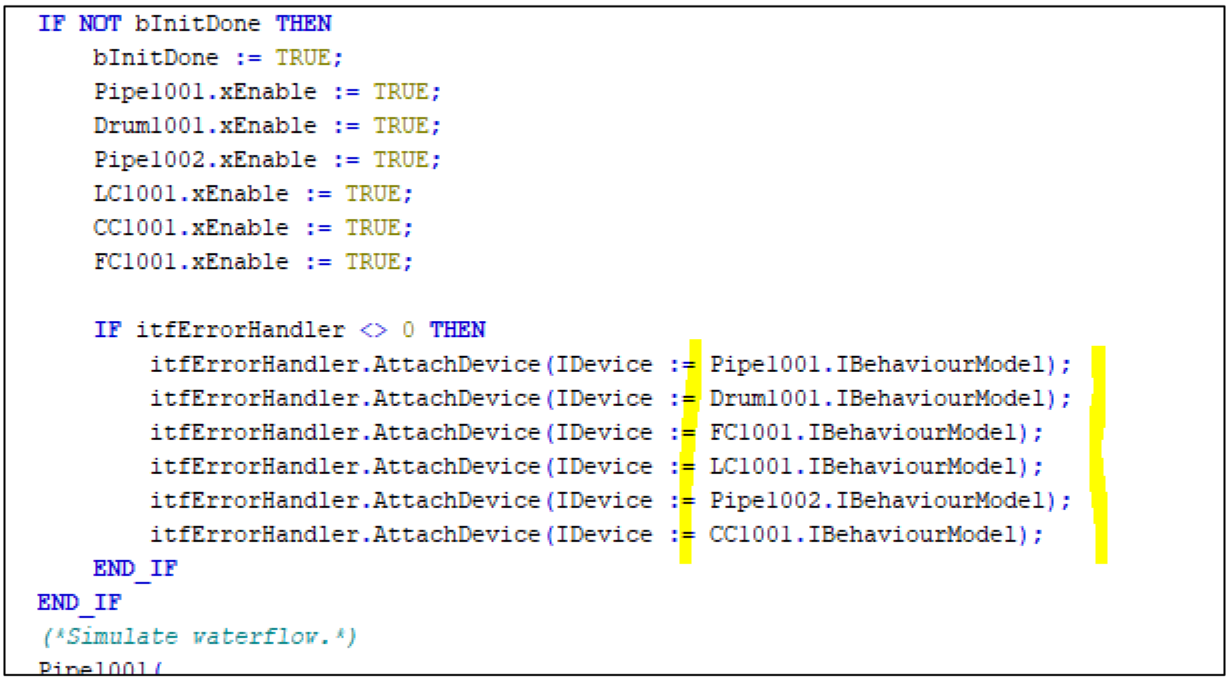

FB_OutputPipe

```
FUNCTION_BLOCK FB_OutputPipe EXTENDS FB_DeviceBasic
VAR_INPUT
    rSteamDemand :  REAL;
    xEnable : BOOL;
END_VAR
VAR_OUTPUT
    eErrorID : ERROR;// error code describing error condition
    rSteamFlow :    REAL;
END_VAR
VAR
    FT1002 :        FB_SteamFlow;
    StateBehaviour : CBML.LConC;
END_VAR
```

```
THIS^.IBehaviourModel := StateBehaviour;
StateBehaviour(xEnable:= xEnable, xBusy=> xBusy, xError=> xError);
```

Figure 16: The needed changes in the Device FB

To attach our FBs to the Error Handler we use our property so that the interface to the Error Handler can still be the same.



```
IF NOT bInitDone THEN
    bInitDone := TRUE;
    Pipe1001.xEnable := TRUE;
    Drum1001.xEnable := TRUE;
    Pipe1002.xEnable := TRUE;
    LC1001.xEnable := TRUE;
    CC1001.xEnable := TRUE;
    FC1001.xEnable := TRUE;

    IF itfErrorHandler <> 0 THEN
        itfErrorHandler.AttachDevice(IDevice := Pipe1001.IBehaviourModel);
        itfErrorHandler.AttachDevice(IDevice := Drum1001.IBehaviourModel);
        itfErrorHandler.AttachDevice(IDevice := FC1001.IBehaviourModel);
        itfErrorHandler.AttachDevice(IDevice := LC1001.IBehaviourModel);
        itfErrorHandler.AttachDevice(IDevice := Pipe1002.IBehaviourModel);
        itfErrorHandler.AttachDevice(IDevice := CC1001.IBehaviourModel);
    END_IF
END_IF
(*Simulate waterflow.*)
Pipe1001(
```

Figure 17: Corresponding changes to the attached devices

The architecture is shown in Figure 18:UML Diagram of the new design. It has now the advantage, that we can use the new abstract FB to add new features which are related to all FBs in a consistent way. Code redundance is also avoided in every FB. Furthermore, it provides the possibility to inject the Behaviour Model from outside with the Property BehaviourModel.

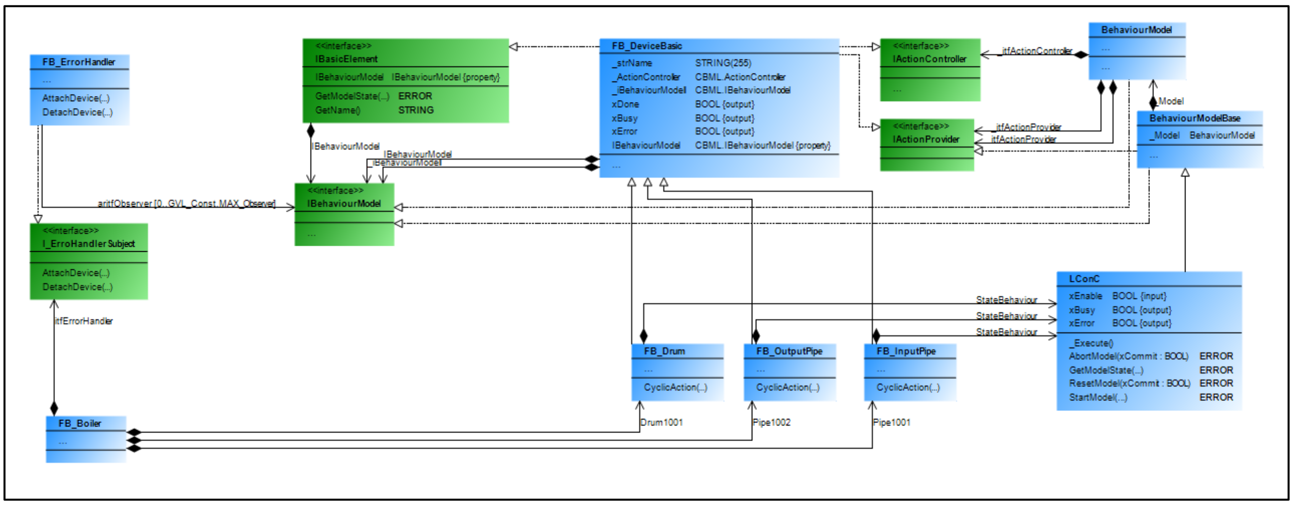


Figure 18: UML Diagram of the new design

Based on this, it is easy to create additional functionalities, like an alarm list, as done in 4.6 Adding a system alarm list.

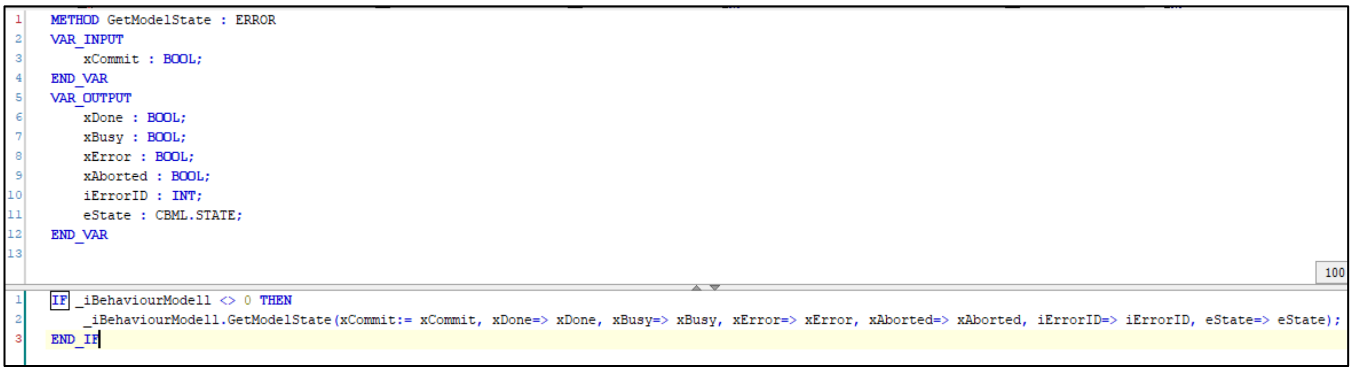
4.6. Adding a system alarm list

One can differentiate between process alarms and system alarms, to filter them for different users (e.g. tank operator and system administrator). This example deals with system alarms, but the process alarms can be created in a similar way. One can even extend the FBs with an output like ProcessAlarm and handle these separately.

To create an error list it may be useful, to know in which FB the Error is happening or even the complete instance path.

For that, we have implemented the name and a Method to have access to the Name in our Abstract Base class. The name is generated here by a feature of the Codesys Development System, but it can also be set via a Property, for example during the Init Routine in the Boiler FB. Using an Interface which describes the Method directly follows the Interface Segregation Principle.

To get access to this method, we need a new or extended Interface for the Error Handler. Changing the interface is normally not the best style but, in this case, we can directly implement the interface in such a way, so that it includes only the necessary information. In this sample these are the GetModuleState and the GetName Methods. So, it makes sense to adapt the Interface in the Attach / Detach Method and the Array of Interfaces in a way that they do not use the IBehaviourModel but the new Interface IBasicElement. The Method GetModelState will delegate the request to the internal set State Machine, as shown hereunder (Note: ERROR type is defined in Figure 4: Function block interface and state diagram of LConC):

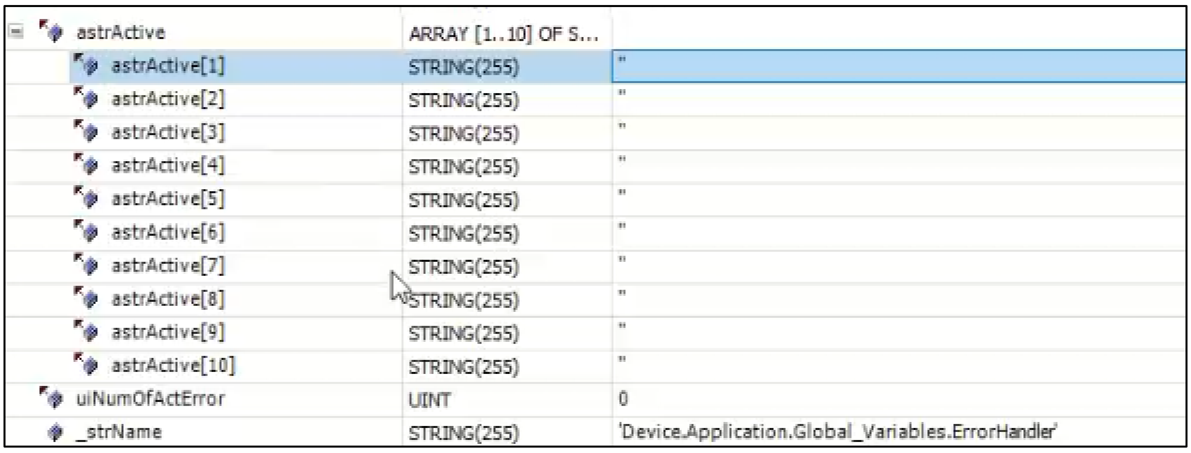


```
METHOD GetModelState : ERROR
VAR_INPUT
    xCommit : BOOL;
END_VAR
VAR_OUTPUT
    xDone : BOOL;
    xBusy : BOOL;
    xError : BOOL;
    xAborted : BOOL;
    iErrorID : INT;
    eState : CBML.STATE;
END_VAR
```

```
IF _iBehaviourModell <> 0 THEN
    _iBehaviourModell.GetModelState(xCommit:= xCommit, xDone=> xDone, xBusy=> xBusy, xError=> xError, xAborted=> xAborted, iErrorID=> iErrorID, eState=> eState);
END_IF
```

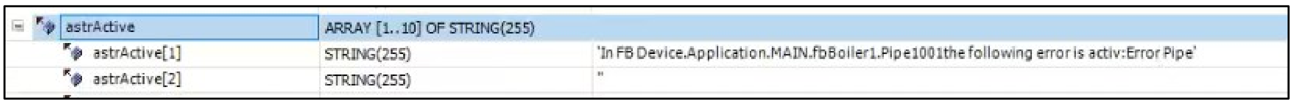
Figure 19: Implementation of the delegation of the GetModelState

This is implemented in a way that all errors which happen are presented in a string array, which users can read.



astrActive	ARRAY [1..10] OF S...	
astrActive[1]	STRING(255)	''
astrActive[2]	STRING(255)	''
astrActive[3]	STRING(255)	''
astrActive[4]	STRING(255)	''
astrActive[5]	STRING(255)	''
astrActive[6]	STRING(255)	''
astrActive[7]	STRING(255)	''
astrActive[8]	STRING(255)	''
astrActive[9]	STRING(255)	''
astrActive[10]	STRING(255)	''
uiNumOfActError	UINT	0
_strName	STRING(255)	'Device.Application.Global_Variables.ErrorHandler'

Figure 20: String array for the error handler

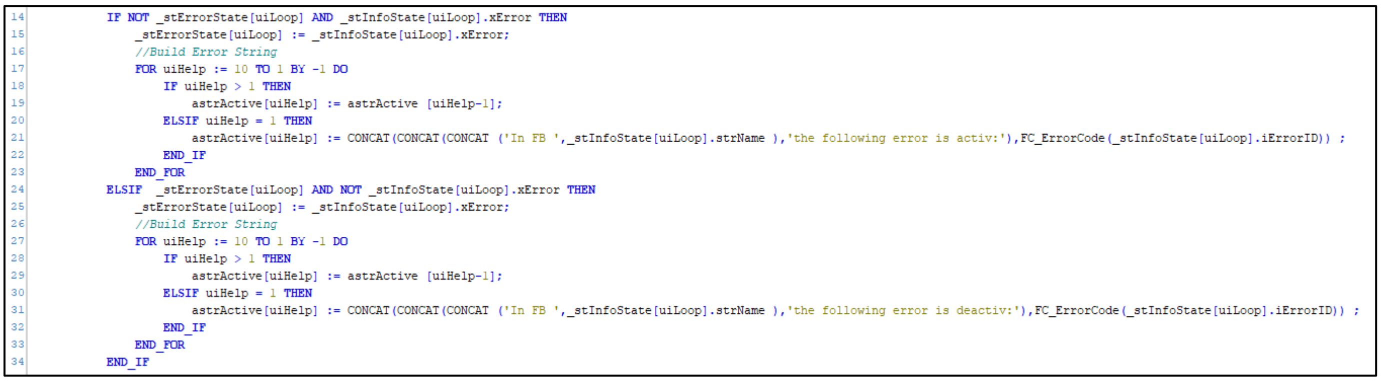


astrActive	ARRAY [1..10] OF STRING(255)	
astrActive[1]	STRING(255)	'In FB Device.Application.MAIN.fbBoiler1.Pipe1001the following error is activ:Error Pipe'
astrActive[2]	STRING(255)	''

Figure 21: Example of an error string

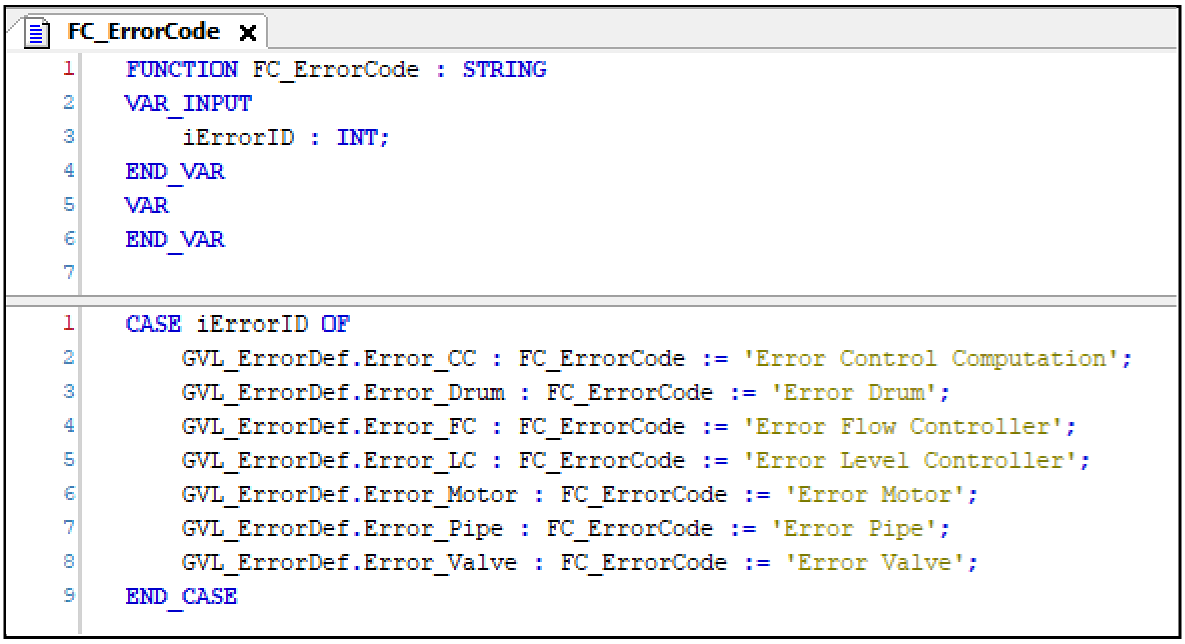
Figure 22:Example of creating the string in the error message, shows a small function as part of the error handling which converts the Error into a clear string, only on the rising and falling edge of the xError Output.

Combined with a function FC_ErrorCode which converts the iErrorID to a string (see Figure 23:The function FC_ErrorCode).



```
        IF NOT _stErrorState[uiLoop] AND _stInfoState[uiLoop].xError THEN
            _stErrorState[uiLoop] := _stInfoState[uiLoop].xError;
            //Build Error String
            FOR uiHelp := 10 TO 1 BY -1 DO
                IF uiHelp > 1 THEN
                    astrActive[uiHelp] := astrActive [uiHelp-1];
                ELSIF uiHelp = 1 THEN
                    astrActive[uiHelp] := CONCAT(CONCAT(CONCAT ('In FB ',_stInfoState[uiLoop].strName ),'the following error is activ:'),FC_ErrorCode(_stInfoState[uiLoop].iErrorID)) ;
                END_IF
            END_FOR
        ELSIF  _stErrorState[uiLoop] AND NOT _stInfoState[uiLoop].xError THEN
            _stErrorState[uiLoop] := _stInfoState[uiLoop].xError;
            //Build Error String
            FOR uiHelp := 10 TO 1 BY -1 DO
                IF uiHelp > 1 THEN
                    astrActive[uiHelp] := astrActive [uiHelp-1];
                ELSIF uiHelp = 1 THEN
                    astrActive[uiHelp] := CONCAT(CONCAT(CONCAT ('In FB ',_stInfoState[uiLoop].strName ),'the following error is deactiv:'),FC_ErrorCode(_stInfoState[uiLoop].iErrorID)) ;
                END_IF
            END_FOR
        END_IF
```

Figure 22: Example of creating the string in the error message



FC_ErrorCode

```
FUNCTION FC_ErrorCode : STRING
VAR_INPUT
    iErrorID : INT;
END_VAR
VAR
END_VAR

```

```
CASE iErrorID OF
    GVL_ErrorDef.Error_CC : FC_ErrorCode := 'Error Control Computation';
    GVL_ErrorDef.Error_Drum : FC_ErrorCode := 'Error Drum';
    GVL_ErrorDef.Error_FC : FC_ErrorCode := 'Error Flow Controller';
    GVL_ErrorDef.Error_LC : FC_ErrorCode := 'Error Level Controller';
    GVL_ErrorDef.Error_Motor : FC_ErrorCode := 'Error Motor';
    GVL_ErrorDef.Error_Pipe : FC_ErrorCode := 'Error Pipe';
    GVL_ErrorDef.Error_Valve : FC_ErrorCode := 'Error Valve';
END_CASE
```

Figure 23: The function FC_ErrorCode

A possible improvement in the design can be to change the way the ErrorHandler checks all the FBs in a cyclic way. One idea can be to use here the Observer Pattern in this way that the ErrorHandler is connected to the Devices as an Observer and the Devices will send out a notification if the Status is changed.

5 Additional References

5.1. Fundamentals for OOP Guidelines – considerations

Within this document, some key aspects shall be taken into consideration:

- Good design but bad performance -- Which details should be considered in order not to influence the performance too much?

- Inheritance and composition -- How inheritance weakens encapsulation
  Many respected OO designers[1] have stated that composition should be given preference over inheritance whenever possible and that inheritance should only be used when necessary.
  However, this statement is somewhat shortened. The call that composition should be used whenever possible obscures the actual problem.
  Experience shows that composition is in most cases more appropriate than the use of inheritance.
  The fact that composition seems more appropriate in most cases does not mean that the use of inheritance is to be avoided.
  Both methods, inheritance, and composition, have their justification, but in the right context.

- Avoidance of dependencies
  The introduction of dependencies at runtime (Dependency injection) plays a central role in the discussion whether the composition should be given a priority over inheritance.
  It is very important to note that this is only a discussion.
  The purpose of this discussion is not necessarily to find the "optimal" way to design a class, but to provide food for thought on how to deal adequately with the decision for or against the use of inheritance or composition.

5.2. Notes on performance considerations

With the change in the architecture, it is very important not to lose sight of the special situation of a PLC compared to a PC/server system.

Overall, there are three actors who could still do a little more for the application of OOP:

- Hardware design
  Currently we can see a big difference in the measurements of the runtime. Depending on the hardware, a virtual method call takes drastically more time or is of little importance compared to a normal subroutine call. Of course, a small runtime extension in a cyclic executed PLC operation is often of great importance. The decisive factor here is the connection of the data/code memory to address and data buses and the possibilities of proper cache mechanisms. Especially the cache can help us to save time in cyclic operation. When designing a controller today, it is no longer just a question of how much time 1024 IL instructions require!
- Compiler design
  The code generator for a PLC must ensure that the code parts of an application are in relation to the cache structure in the neighborhood. So, code and data which are used cyclically can be processed very fast without a new transfer from the main memory. The implementation of interfaces, methods, and properties but also references must be adapted to the needs of a PLC. So, use cases like OnlineChange and

[1] Like: *Design Patterns: Elements of Reusable Object-Oriented Software* by ErichGamma, RichardHelm, RalphJohnson, and JohnVlissides (the GangOfFour).

Debugging must be considered early and well. The allocation of memory on the stack (local variables) must not create a problem when using methods. The initialization of extensive data structures can become a problem (e.g., when cyclically called) and should be accordingly controllable by the user.

- Software design
  The keywords ABSTRACT, FINAL, PRIVATE, ... must be used consciously.
  This allows the developer to influence essential aspects of the runtime of his software at a very early stage. The developer must keep an eye on whether and when the initialization of his local variables must take place. The PLC development environment should provide a way to prevent the initialization of a set of specific variables so that "much" time can be saved here.

5.3. Notes for further investigations

- Synchronous versus a-synchronous communication / run-time environments
- Should we introduce dynamic memory allocation to the classical control world?
- Show no preferences for already existing implementations and programming styles.
- It can be that a programming paradigm shift is unavoidable.