ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV014

CONTROL SYSTEM OF A PORTABLE PUMPING STATION FOR ULTRA-HIGH VACUUM

M.Trevi[†], L.Rumiz, E.Mazzucco, D.Vittor, Elettra Sincrotrone Trieste, Trieste, Italy

Abstract

Particle accelerators operate in Ultra High Vacuum conditions, which have to be restored after a maintenance activity requiring venting the vacuum chamber. A compact, independent and portable pumping station has been developed at Elettra to pump the vacuum chamber and to restore the correct local pressure. The system automatically achieves a good vacuum level and can detect and manage vacuum leaks. It has been designed and manufactured in-house, including the mechanical, electrical and control parts. By means of a touch screen an operator can start all the manual and automatic operations, and monitor the relevant variables and alarms. The system archives the operating data and displays trends, alarms and logged events; these data are downloadable on a removable USB stick. Controlled devices include two turbomolecular pumps, one primary pump, vacuum gauges and one residual gas analyser. The control system has been implemented with a Beckhoff PLC (Programmable Logic Controller) with RS-485 and Profibus interfaces. This paper focuses in particular on the events management and object-oriented approach adopted to achieve a good modularity and scalability of the system.

INTRODUCTION

Sometimes sectors of the accelerator vacuum chamber need maintenance or updates requiring venting, but when they have been carried out it is necessary to create ultra-level vacuum conditions to go back to normal operations.

At Elettra, ultra-level vacuum is usually created locally using a pumping station with on electromechanical logics on board (relays, mechanical timers, etc.). The main disadvantage is that it is not programmable and in case of failure it can stop without any information about the reason (for example a mains interruption due to a thunderstorm). Another disadvantage is that, due to its dimensions, the system is not easily moveable inside the accelerator tunnel. These issues led us to design an automatic and autonomous system managing entire parts of the system: alarm management, operator interface, archiving of variables and events like commands, value changed, etc. An important requirement was the compactness of the entire system that has been achieved by suitable choices of mechanical, electrical and automation components. Moreover another feature that we wanted to reach was the possibility to record log and trend.

PLC, I/O boards and wirings are contained in a 3-unit rack (~180 mm height). The controller based on a Beckhoff CX5120 PLC is compact with very good heat dissipation. The Beckhoff development environment feature an IDE and OOP (Object Oriented Programming) capabilities. An Exor panel has been chosen as HMI.

For the development of the software we have taken inspiration from a template on the Beckhoff site based on OOP [1]. We chose to create OOP syntax in order to be independent on that of the manufacturer. In this way it should be possible to move the software to other controllers. We have assumed that these controllers implement structured text and that a function block can be divided in actions.

The OOP paradigm is mainly based on three principles:

- *Encapsulation*: the internal state of an object can be modified by a public method.
- *Inheritance*: a child class can derive and redefine methods from a super class. The inheritance defines a hierarchy between classes; the mechanism is static and defined at compilation time.
- *Polymorphism*: it is a dynamic mechanism used at run time. If a class has a method *m()* and one subclass (child) redefines *m()*, the polymorphism allows executing an *m()* version according to which kind of subclass object is calling it.

In the PLCs software classes are implemented by Function Blocks (FB), as defined in the IEC 61131-3 standard [2]. The instance of a FB is equivalent to the object of a class. The standard mentions typical keywords of OOP like *methods*, *extends*, *implements*, etc. We have not used these elements in order to comply as much as possible with other kinds of controllers that do not implement these keywords. *Methods* are replaced by dividing the FB code in several independent parts called *actions*. *Extends* are replaced by writing FB I/O interfaces in order to implement the same concepts.

Another important aspect of the design was alarms and commands management. An alarm can be configured in order to allow the following actions: auto-reset, acknowledge and recording. A command can be recorded by means of the PLC logging system or by the *audit trail*, which is a logging mechanism provided by the HMI. In this log the HMI writes setpoint changes, commands sent to the PLC and it tracks who did what. This feature is widely used in industrial systems [3]; adopting it is particularly easy to save and move log files outside the system for data analysis.

All the system configurations can be easily changed many times during the design and commissioning of the software using a form which we have designed in C# VS2015. The result of this form is written in an XML code to be imported in the file system of the HMI project.

[†] massimo.trevi@elettra.eu

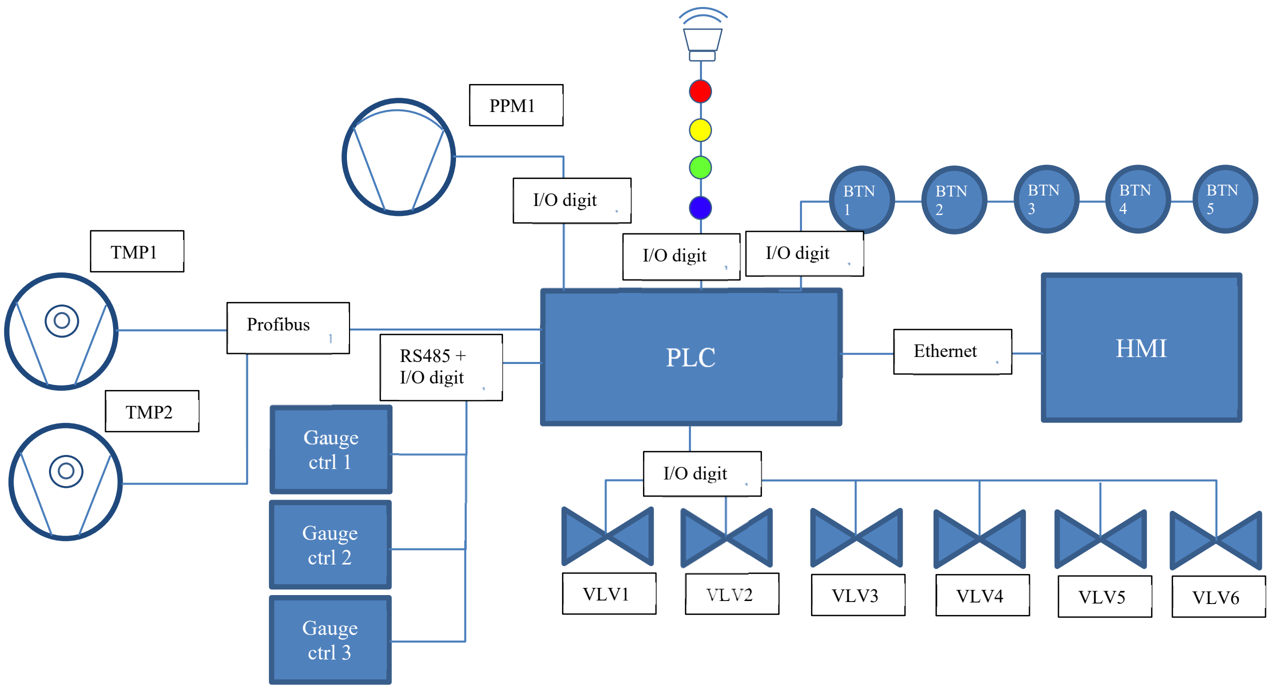


Figure 1: Architecture.

BRIEF DESCRIPTION OF PROCESS

When a vacuum chamber is installed in the accelerator it has to be brought to ultra-high vacuum conditions (~ 10^{-9} mbar). This process is divided into three steps, corresponding to three different vacuum levels and three different kinds of vacuum pumps to be used:

- *Primary* pump (Primary Pump Motor - PPM): starts creating flow vacuum;
- *Turbomolecular* pump (Turbomolecular Pump Motor - TPM): can create high vacuum, in the range 10^{-6}-10^{-8} mbar;
- *Ionic* pump (SIP): allows reaching the target of ~10^{-9} mbar.

The system described in this paper uses the first two kinds of pumps. We have implemented following two features:

- *Leak test*: the test to search for leaks in the vacuum chambers. It is performed by means of a leak detector connected to the vacuum chamber and using helium as a tracer gas; the helium gas has a very small monatomic molecule that can easily penetrate even the smallest holes that the external atmosphere put in contact with the internal vacuum;
- *Residual Gas Analyser* (RGA*)*: instrument that analyses the residual gases inside the vacuum chamber once it has been pumped to a given level. It provides information on the presence of any vacuum leak or contaminants, and on the quality of the vacuum chamber cleaning. Moreover, during some laboratory or industrial processes, it can identify and follow the trend of the partial pressures of the gases introduced into the system (pure gases or mixtures).

The system can be switched in manual and automatic mode. In manual through to the HMI it is mainly possible to command valves and pumps and to set the device configurations. The system will normally be used in automatic mode: this means that when an operator pushes the automatic start button, the PLC automatically sends commands to valves and pumps in order to reach the target pressure.

ARCHITECTURE

Figure 1 shows complete architecture of system. Hardware and software details following.

Hardware

The PLC communicates via Profibus with the turbopump controllers (TMP) and via RS485 with the controllers of the vacuum gauges. The HMI screen is interfaced via Ethernet and the other devices (primary pump, valves, buttons and traffic light) are connected via digital I/Os. In addition, the gauge controllers provide threshold contacts to warn the PLC of any anomaly on the pressure with faster response time.

The traffic light has the following components and meanings:

- *Buzzer*: activated when at least one alarm has to be acknowledged;
- *Red light*: activated when there is at least one alarm;
- *Yellow light*: activated when there is at least one warning;
- *Green light*: if continuous it means that the machine is in automatic mode, if blinking it means that the cycle has started;
- *Blue light* - means that the machine is operating in manual mode.

PLC Software

The main idea has been to divide each task in a number of smaller subtasks. This principle is widely adopted for example for FBs and commands that a device has to execute.

The pumping system is divided into zones which have a logical and process meaning. As shown in Figure 2 the system is divided in 5 zones: *A*, *B*, *C*, *D* and *E*. A zone is defined as the part of the pumping system that can be sectioned by valves. Only the main PLC program can interact with the zone I/O interfaces, which can include other devices interface of FBs like sensors and pumps. In this way this implementation allows to comply with encapsulation.

The main code of this kind of FBs is empty because they are divided in actions. Every action has a task which can:

- be executed by an external command; in this case we call it a *method* or
- expose FB object states to the program or
- interact between internal actions.

The action that implements a command has three features compliant with the method principle:

- call derived FB (class);
- can add code after calling derived class if necessary;
- does not call the derived class in case of rewriting action (override).

These features allow the implementation of *inheritance* and *methods override*.

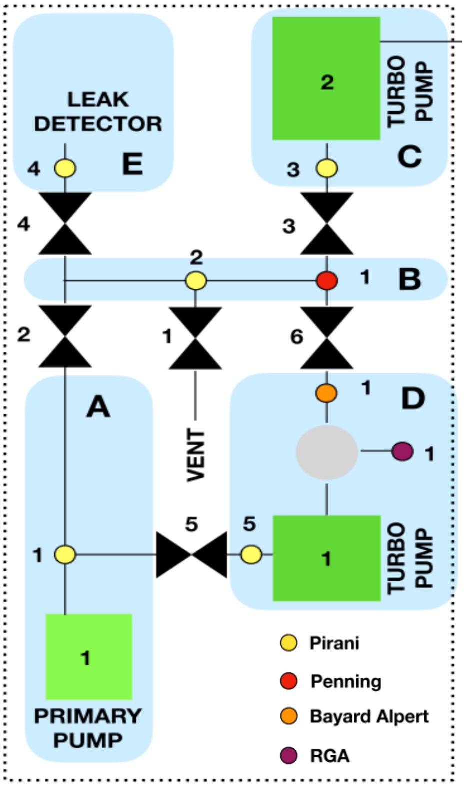


Figure 2: P&Id Process and Instrumentation Diagram.

OBJECT MODELING

The UML-Class Diagram has been used to model the machine devices.

UML (Unified Modelling Language)

UML [4, 5] is a general-purpose, modelling language in the field of software engineering, which is intended to provide a standard way to visualize the design of a system.

A class diagram is a type of static structure diagram that describes the structure of a system by showing:

- classes;
- attributes;
- operations (or methods);
- relationships between objects.

In the Figure 3 we can see that the boxes (FB/class) are connected by different kinds of arrows and lines:

- dashed line with arrow means that there is an association like a dependency between classes; the depend-

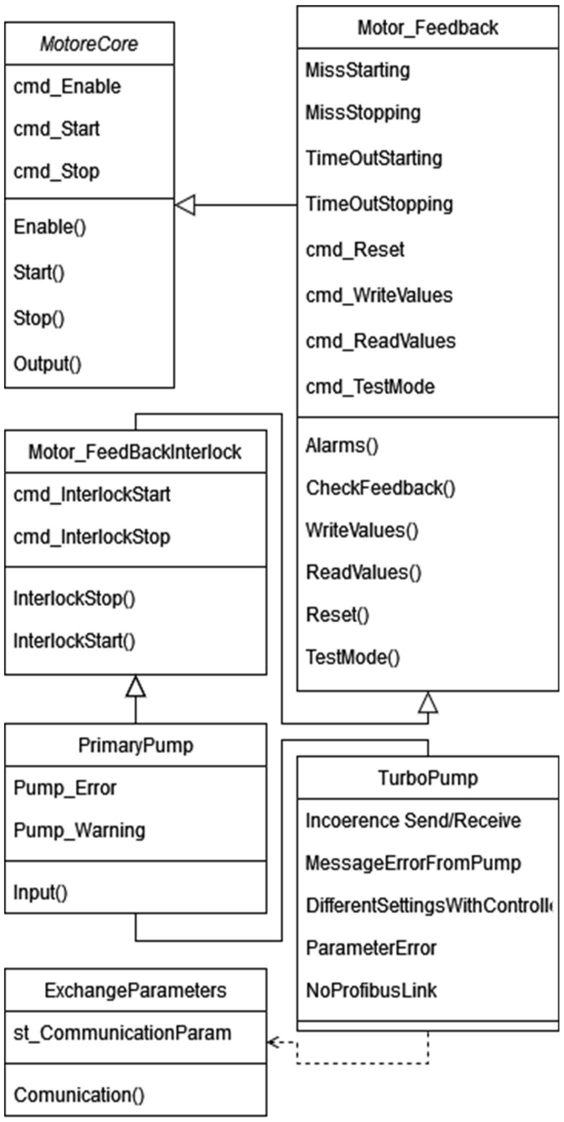


Figure 3: UML - From MotorCore To Pump.

ent class object might use an object of another class in the code of an action/method;

- solid line with arrow means that there is a generalization between FB/class, so there is an inheritance.

The boxes are divided into three parts: on the top we have the name of FB/class, in the middle we have the attributes managed by FB/class, at the bottom we have the actions/methods.

Devices Modelling Example

In order to respect the OOP paradigm, the design of machine device FBs starts from a generic FB (superclass) which is then adapted according to the requirements.

Figure 4 describes the machine model where we can observe the OOP paradigm application in a logical class.

Figure 3 shows the pumps description and the derivation from a generic FB (superclass) of two derived classes: primary and turbomolecular pump FB. They are derived from *Motor_FeedBackInterlock*, which means that primary and turbomolecular pumps have interlock features to start and stop, but also *Motor_FeedBack* and *MotorCore* features.

We would like to underline that a simple push button can be modelled by an OOP paradigm. In our case we have a button with a LED frame and therefore we start from a button that has the electric input (*ButtonCore*) and we derive the *ButtonLight* that has a feedback too.

Figure 4 shows how the machine has been modelled: the machine is divided in zones. A zone can be enabled by a command as well as a device. Every zone has at least one *Pirani* sensor and so *Zone_Pirani* has been derived from *Zone_Core*. From this point there are other three derived FB/classes in order to respect their composition. As explained in the UML chapter different kinds of lines and arrows show different relationships between classes.

Note that, for example, *Zone_Pirani_TP* is derived by *Zone_Pirani* but depends on *Turbopump* class that is modelled in Figure 3.

We can also see this modelling from a P&Id (Process and Instrumentation Diagram) schematic as in Figure 2. In this way we can check matching between a mechanic based scheme and a UML scheme.

FB I/O INTERFACE

Every FB interface that describes something to be displayed on HMI has been designed in order to have the same interface both for a logical or physical type. In addition, it was decided to have a minimum common set of commands and states to be managed in the same way at HMI level.

Having the same interface is useful to organize information to and from a FB. We have divided them in:

Inputs:

- Physical input, connected to PLC boards;
- General input: Boolean, numbers and strings;
- General input: Boolean, numbers and strings used in test mode;

Output:

- Physical output, connected to PLC boards;
- Logical status: represents the status of objects described by FB;

In/Out:

- Alarm;
- Command: corresponds to executing an action/method;
- Data: every kind of number that represents for example thresholds, timers, parameters, configuration, etc.

In/Out variables have been necessary to allow FB/classes to change variables value both inside and outside the code. Note that an Input variable has a part dedicated to TestMode which means that this variable is used when a simulation is running and the object is in TestMode. Simulation is very useful before the commissioning to test automatic cycles, behaviour in case of alarms, etc. Another point to take into consideration in this kind of interface organization is that in the code you can manage devices or logical parts in the same way: they have enable commands, read/write value actions, alarms

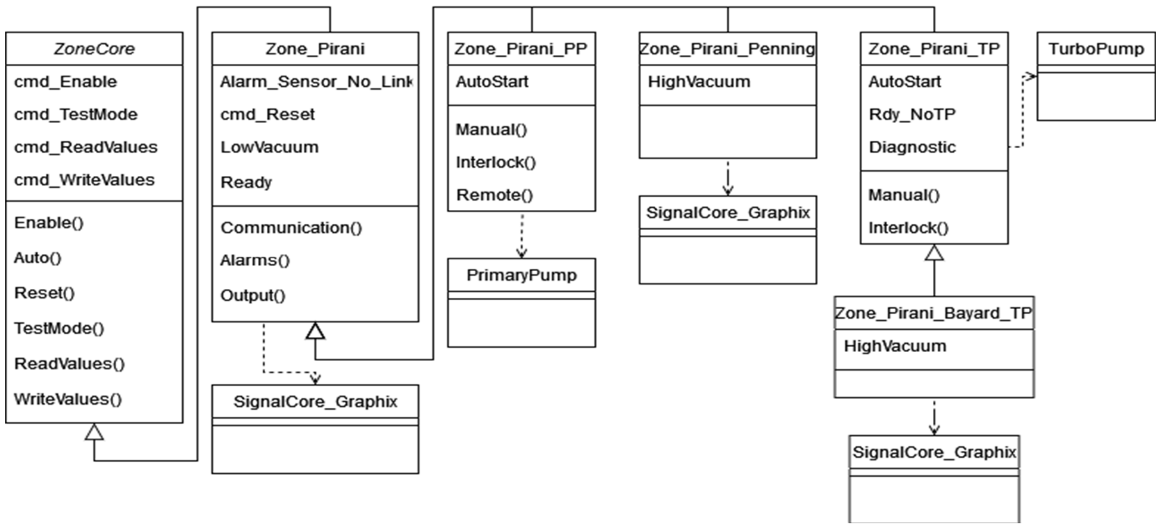


Figure 4: UML Machine Model.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV014

and warning statuses.

Regarding the second choice, in order to have a minimum common set of states and commands, let's consider for example that we want to reuse the management of an object visualization and therefore, according to the status of an object, display the following states common with different colours:

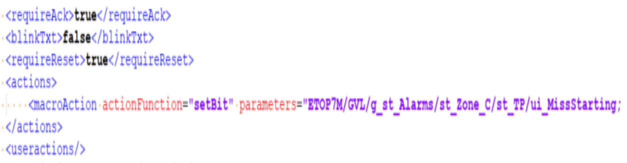
- *Alarm* in red;
- *Warning* in yellow;
- *Run* in green;
- *Testmode* in blinking blue;
- *Disabled* in light grey;
- *No link* in blinking yellow.-

ALARM AND COMMAND SETTINGS

Another important design choice was how to manage the setting of alarms and commands. In fact, every alarm and command is represented by a 16-bit word to define the configuration. One of the mandatory requirements of the project was the possibility to log and record alarms and actions. Also, anomalous events must be recorded, for example a short black out.

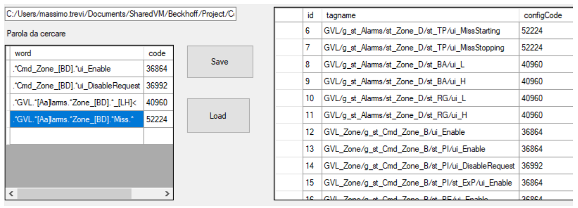
The idea is to have a central logging function in the PLC that is called each time an event happens; examples are a button pushed by an operator or a device that changes state to alarm. An alarm can require an acknowledgement by an operator or be reset automatically. Moreover, an alarm or a command can be logged/recorded. A state associated to each alarm defines if it is active and if an acknowledgement is required. For these reasons the 16-bit word representing an alarm or a command is divided into two parts: 12 bit for the configuration, 4 bit for the state. In this way each alarm and command is represented by a number from 0 to 65535.

While logging is managed by the PLC for example with a circular buffer, the recording action is more complex. To do that we have used the audit trail feature of the HMI, while another feature of the HMI has been employed to acknowledge alarms. For this purpose, a graphical interface using a form written in C# VS2015 has been developed. The form analyses the dictionary created by the HMI. The dictionary is a XML list where tags are described by name and address information. There is a table where it is possible to put the tag name to be found and its setting number. There are some buttons to create tag and alarm files in XML format. According to this number the code configures alarms and tags with the appropriate XML setting based on the specific IDE language. Figure 5 and Figure 6 show an example of setting values contained in a Windows form and the effects on the XML code; the code 52224 is used to configure an alarm to be resettable and acknowledgeable.



```
<requireAck>true</requireAck>
<blinkTxt>false</blinkTxt>
<requireReset>true</requireReset>
<actions>
    <macroAction actionFunction="setBit" parameters="ETOP7M/GVL/g_st_Alarms/st_Zone_C/st_TP/ui_MissStarting;
</actions>
<useractions/>
```

Figure 5: XML extract.



C:/Users/massimo.trevi/Documents/SharedVM/Beckhoff/Project/C

Parola da cercare

word	code
.*Cmd_Zone_[BD].*ui_Enable	36864
.*Cmd_Zone_[BD].*ui_DisableRequest	36992
.*GVL.*[Aa]larms.*Zone_[BD].*_[LH]<	40960
.*GVL.*[Aa]larms.*Zone_[BD].*Miss.*	52224

id	tagname	configCode
6	GVL/g_st_Alarms/st_Zone_D/st_TP/ui_MissStarting	52224
7	GVL/g_st_Alarms/st_Zone_D/st_TP/ui_MissStopping	52224
8	GVL/g_st_Alarms/st_Zone_D/st_BA/ui_L	40960
9	GVL/g_st_Alarms/st_Zone_D/st_BA/ui_H	40960
10	GVL/g_st_Alarms/st_Zone_D/st_RG/ui_L	40960
11	GVL/g_st_Alarms/st_Zone_D/st_RG/ui_H	40960
12	GVL_Zone/g_st_Cmd_Zone_B/ui_Enable	36864
13	GVL_Zone/g_st_Cmd_Zone_B/st_PI/ui_Enable	36864
14	GVL_Zone/g_st_Cmd_Zone_B/st_PI/ui_DisableRequest	36992
15	GVL_Zone/g_st_Cmd_Zone_B/st_PI/st_ExP/ui_Enable	36864
16	GVL_Zone/g_st_Cmd_Zone_B/st_RF/ui_Enable	36864

Figure 6: Part of Windows Form.

FUTURE IMPROVEMENTS

There are a number of features that have not been implemented yet because this system is a prototype, but will be taken into account in future implementations.

For example, it would be useful to provide a TCP/IP interface to receive commands and send data via *Tango* for remote panels. Furthermore, there would be the possibility of a *VNC* connection by a remote host with a server on the HMI.

Other features that we want to implement are automatic interlock messages: for example, if we want to open manually a valve that cannot be opened or if the start button is pressed and the cycle does not start, a message should appear to show the reason to the operator. In this way it would be possible to create a troubleshooting system to help the operator to make the right operation to fix the problem and start the system faster.

Moreover, we would like to consider converting the present PLC program in order to use Beckhoff OOP keywords and eventually evaluate improvements in terms of complexity and performance in the two solutions.

Finally, it would be useful to design a superclass for generic serial communications to allow deriving from it the Profibus and RS485 classes.

REFERENCES

[1] Beckoff, https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2527303819.html&id=

[2] Wikipedia, https://en.wikipedia.org/wiki/IEC_61131-3

[3] Food and Drug Administration - CFR 21 - part 11.

[4] Wikipedia, https://en.wikipedia.org/wiki/Unified_Modeling_Language

[5] Available: https://www.visual-paradigm.com