2nd IFAC Workshop on Automatic Control in Offshore Oil and Gas Production,
May 27-29, 2015, Florianópolis, Brazil

BR-Sensor: An On-line Data-driven Soft Sensor of Downhole Pressure ⋆

Edson F. A. Rezende *, **Alex F. Teixeira** **
Eduardo M. A. M. Mendes *,

* *Department of Electronic Engineering, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, M.G., Brazil*
e-mail: edsonfrederico@yahoo.com.br, emmendes@cpdee.ufmg.br.
** *Research and Development Center (CENPES), Petroleo Brasileiro S.A. , Rio de Janeiro, R.J., Brazil,*
alex.teixeira@petrobras.com.br

Abstract: In this work an object-oriented approach for implementing soft sensor components, encapsulated as dynamic-link-library (DLL), is proposed. A variety of models used for the estimation of a specific unmeasured process variable and implemented as DLL components is integrated to the soft sensor main component by simply following the interface described in this paper. In order to check the performance of the proposed approach a computational analysis is carried out. Although the main objective is to deploy the soft sensors into the supervisory the main ideas laid out here can be extended to other Windows-based solutions.

Keywords: Softsensors, object-oriented, software-design, Kalman-Filters, dynamic-link-library.

1. INTRODUCTION

In the process industry, on-line soft sensors usually run Windows-based solutions on a PC microcomputer. Supplier companies of advanced control systems such as Emerson, Honeywell, Aspen and Yokogawa deliver Windows-based solutions. Alternatively, there is also hardware implementations of soft sensors (Garcia et al., 2008). In (Teixeira et al., 2014), a data-driven soft sensor for the downhole pressure of a gas-lift well was developed and, despite successful offline implementation, an on-line version was not derived.

There are many reports of the good performance of on-line soft-sensors in the literature. Salvatore et al. (2009) show the result of a neural-network based soft sensor implemented in the PI System™ (a Plant Information Management System - PIMS - provided by OSISoft®) to infer the sulfur content in a diesel hydrotreating unit. The soft sensor was shown to be very low computationally demanding with execution times lower than one second. Li et al. (2013) have developed and commissioned an expert control system that works based on the on-line estimates of important process variables calculated by a soft sensor in an aluminium production factory.

An interesting scheme for implementing soft sensors was proposed in (Zhou et al., 2013). The authors used a VBA (Visual Basic Application) module supported by the SCADA system, RSView32™, to provide an integration between the supervisory system and MATLAB® by means of the DDE protocol for data exchange. The authors mentioned that MATLAB®was used to perform some necessary calculations, mainly the most complex matrix calculations of a typical soft sensor. The proposed soft sensor was implemented in a grinding plant.

Although the performance of soft sensors has been reported in the literature, very few works discuss the implementation procedure in details let alone the technology and systems they are used for. This work aims to provide a contribution in this context by showing details of the development of a Windows-based on-line version of a soft sensor, coined *BR-Sensor*, that estimates the downhole pressure on the gas-lift well presented by Teixeira et al. (2014). *BR-Sensor* is currently under validation at CENPES-Petrobras. The methodology proposed here can be used to integrate a variety of mathematical models to a production environment in oil platforms.

This paper is organized as follows. In Section 2, an brief introduction to *BR-Suite* and *BR-Optimus* is given. In Section 3, a basic soft sensor that can be integrated to the *BR-Optimus* environment is developed and as a by-product a useful methodology for implementing different representations of estimated models is given. The final solution, a system that not only uses models to predict the downhole pressure in gas-lift oil wells in an entire oil platform but it can also manage different models and representations, the *BR-Sensor*, is presented in details in Section 4. The conclusions are given in Section 5.

2. BACKGROUND: BR-SUITE AND BR-OPTIMUS

The soft sensor implemented in this work, *BR-Sensor*, was developed on the top of a base system available at Petrobras. This system consists of a SCADA (Supervisory Control And Data Acquisition) system called *BR-Suite* and a module called *BR-Optimus*. The *BR-Optimus*

⋆ Final support from Petrobas S.A. is acknowledged.

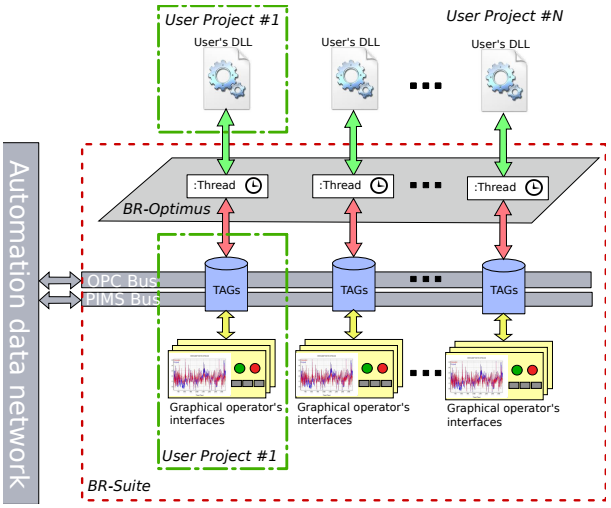

Fig. 1. *BR-Suite* environment: Relationship between *BR-Suite*, *BR-Optimus* and user's project (UP) modules. The red-dashed box represents the *BR-Suite* environment that contains the *BR-Optimus* layer, two option of data bus for updating the user's project's tags (OPC and/or PIMS) and a set of UP. The UP elements are inside the green-dash-dot box.

module runs under the *BR-Suite* environment. The *BR-Optimus* project is basically a multi-thread task responsible for the management of user projects (*UP*). Each UP encompasses three elements: (1) a set of data tags; (2) a set of graphical operator interface; and (3) a dynamic-link-library (DLL) for performing any task such as calculation or user functions. Items (1) and (2) are implemented inside the *BR-Suite* environment and item (3) is not. In fact, *BR-Optimus* acts as a layer between *BR-Suite* and the user's DLLs. *BR-Suite* also comes with two data exchanging interfaces for gathering data from the process on an OPC DA Client and a driver for communicating with a PIMS (Plant Information Management System), a PI System™. With these two options of data exchanging, read and write data from or to the process is available to the user.

The tag database is updated with an information traffic going through three directions: (1) the OPC/PIMS data channels, (2) the operator graphical interfaces and (3) the user DLL. To exchange information with the user DLL, *BR-Optimus* has a dedicated thread instance that works as a bridge. Figure 1 shows the *BR-Suite* environment.

In order to have a dedicated thread running on *BR-Optimus* and to keep a communication with the user DLL, an interface, *IDataAndControl*, is needed. Figure 2 shows the interface and its three methods.

The main objective of each method is listed bellow:

(1) **StartupAlgorithm()**: this method sends the information on how the user tag database is organized to the user's DLL. The data types should be known when creating the user DLL;
(2) **ExecuteAlgorithm()**: this method runs periodically, shares the current values of the user tag database with the user DLL and expects that the user DLL, as a response to some data processing, make changes in the current values of the user tag database.
(3) **FinalizeAlgorithm()**: the last tasks are executed before the DLL is unloaded in this method;

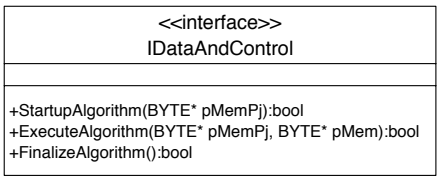

Fig. 2. IDataAndControl interface specification. An user's DLL must have the interface (three methods) in order to be integrated to *BR-Optimus*.

Figure 3 shows a diagram with the task sequence execution to clarify how the methods are called and organized.

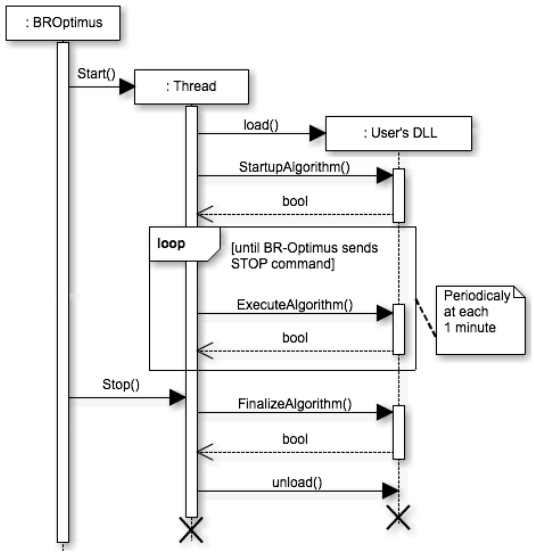

Fig. 3. Task sequence execution diagram of *IDataAndControl*'s methods invoked by a dedicated thread in the *BR-Optimus* layer.

3. DESIGN OF A BASIC SOFT SENSOR

In order to accomplish the task of developing a soft sensor to estimate a specific variable of oil wells in a petroleum oil platform the development of a basic soft sensor is described to serve as the first step towards a complete implementation of mathematical models into soft sensors. This section provides some information on this procedure. A typical soft sensor flowchart is presented in Section 3.1. The computational solution for the implementation of the soft sensor is described in Section 3.2. Section 3.4 shows an example of a code. Section 3.5 shows how a model can be modified or changed.

3.1 The estimation flowchart

A general soft sensor that runs a closed-loop estimation scheme as proposed in Teixeira et al. (2014) and reformulated here as shows Figure 4 will be the starting point.

The soft sensor consists of

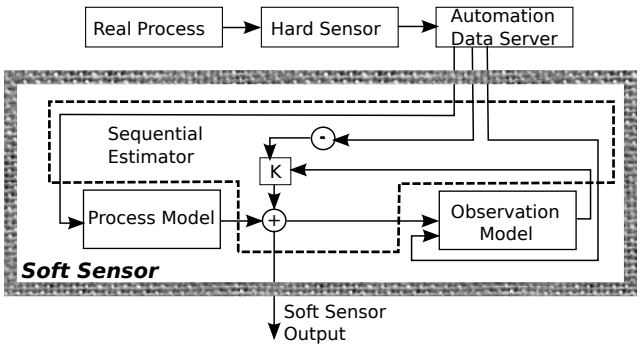

Fig. 4. Estimation flowchart of a basic soft sensor running a closed-loop estimation scheme.

(1) a **process model**: that gives an estimate of the desired variable from the measurements of other variables;
(2) an **observation model**: that explains other variable from the desired variable prior estimation and is used to correct the process model output;
(3) and a **sequential estimator**: responsible for feeding the model's inputs, reading the model output and calculating the final estimate or prediction of the desired variable at each iteration;

There are several sequential estimators schemes available in the literature (Simon, 2006; Daum, 2005). In this work, the *Unscented* Kalman Filter (Julier and Uhlmann, 2004) was chosen for being an effective estimator to deal with nonlinear models. Within the filter, there are many mathematical representations that can be used to write the models but only two possible representations were considered here: polynomial (a special case of Nonlinear Autoregressive Moving Average models with eXogenous inputs) and neural networks (NN) (e.g. MLP) (Rafiq et al., 2001). These two representations have been proven to be universal approximators (Chen and Billings, 1989; Cybenko, 1989) .

The soft sensor, that resides in the virtual world, has an interface with the real world through the automation data server that can be, for example, an OPC Data Server, a PIMS system or even a tag database of a SCADA system. In the next section an alternative to implement the soft sensor flowchart in the virtual world is presented.

3.2 From the estimation flowchart to the component diagram

One of the software requisites for developing on-line soft sensors is to provide a model maintenance capability since a gradual deterioration of its performance can be observed in most cases as stated in Kadlec et al. (2009). As far as software modelling is concerned, a way to achieve this capability is treating the process and observation models of a soft sensor as object components. This strategy can be seen in the work of Herbst and Pate (1999) where they developed a soft sensor called *ComMINSens* to be used in milling circuits.

Since the *DLLs*, which are object components, are the core in the technology of integration adopted in the *BR-Optimus* architecture, they are extensively used here to implement the proposed soft sensor. The estimation flux flowchart exhibited in Figure 4 can be now seen as the component diagram in Figure 5.

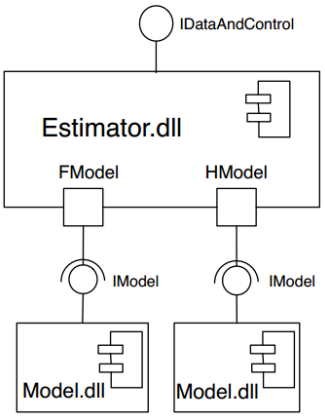

Fig. 5. Component diagram: How the estimation flowchart can be converted into software components.

3.3 A linear algebra library

The implementation of sequential Kalman-Filter-based estimator's algorithms requires the computation of matrix operations such as inversion and element-wise multiplications. One of prerequisites when choosing a linear algebra library is the support to complex matrix calculations and the possibility to declare matrix data types within the user's code so as to improve code quality.

In this work, the *Eigen* matrix library (Bates and Eddelbuettel (2013)), which is a free-cost and high-quality C++ library for linear algebra computation, was used. The Eigen library introduces the class *MatrixXd* that allows the user to define matrix variables with double data-type. This class was extensively used during the development of the proposed soft sensor.

3.4 Describing the IModel *interface*

The process or observation model in the component diagram shown in Figure 5 is implemented using the *IModel* interface exhibited in Figure 6 and then compiled as a *Dynamic-Link-Library*. In C++ programming, for a function to work as an interface method the function declaration statement should be preceded by *"extern "C" __declspec(dllexport)"*. Note that all the attributes must be declared as global variables in this project. The methods that starts with *"Get"* are important only to show extra information to the user of the *IModel* interface.

The methods that are of crucial importance are:

- **Initialize()** is responsible for loading the model parameters into the memory.
- **Calculate()** calculates the one-step-ahead model prediction.
- **Update()** reads a text-file with model parameters and loads them into the memory.

In order to show how *Calculate()* can be used, the following nonlinear autoregressive model with exogenous inputs (NARX) is considered

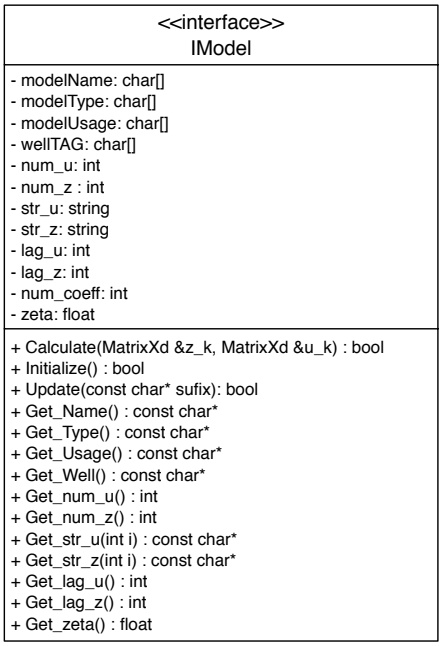

Fig. 6. Interface IModel: all model components must implement this interface. The method *Calculates()* returns the one-step-ahead prediction

$$\begin{aligned} z(k) = \; & +0.15529\times10^{+1}\, z(k-1) - 0.43213\, y(k-2) \\ & -0.68464\times10^{-3}\, z(k-2)^2 \\ & -0.25743\times10^{-2}\, u_2(k-1)z(k-3) \\ & +0.20553\times10^{-2}\, u_2(k-2)z(k-3) \\ & +0.22339\times10^{-3}\, u_2(k-7)u_1(k-1) \\ & -0.22169\times10^{-4}\, u_1(k-4)u_1(k-1), \end{aligned} \tag{1}$$

where z is the model output and u_1 and u_2 are two different inputs. For details on how to model in Equation 1 was obtained, please refer to the Methodology Section in (Teixeira et al., 2014).

The model in Equation 1 is implemented as

```
extern "C" __declspec(dllexport) bool
    Calculate(MatrixXd &z_k, MatrixXd &u_k)
{

//Build the regression-vector
v_reg<< z_k(0,1),
z_k(0,2),
z_k(0,2)*z_k(0,2),
u_k(1,1)*z_k(0,3),
u_k(1,2)*z_k(0,3),
u_k(1,7)*u_k(0,1),
u_k(0,4)*u_k(0,1);

//Calculate the model output
MatrixXd aux(1,1);
aux = v_reg *v_theta;

//Function returning
z_k(0,0) = aux(0,0);

return true;
};

extern "C" __declspec(dllexport) bool Initialize()
{
//Loads parameters
v_theta << 1.552927565454389e+00,
-4.321297647086121e-01,
-6.846362847500344e-04,
-2.574346641459825e-03,
 2.055252691826919e-03,
 2.233863904075120e-04,
-2.216917620149358e-05;


return true;
};
```

3.5 Model maintenance strategy

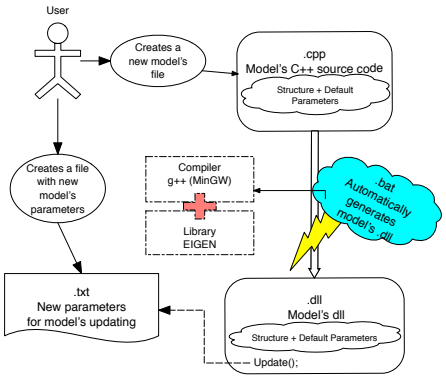

Fig. 7. Options for model maintenance.

The model maintenance can be performed in two different ways depending on what part needs to be modified: the model parameters or the model structure. In order to update the model parameters, a text-file with a proper format containing the new parameter values must be created. After invoking *Update()*, which receives as argument the file path, the model will be updated in the DLL currently running. In situations where a complete different model was determined using some modelling technique, the new model structure and parameters must be coded using a new DLL. Similarly to the previous case, a text-file must be created. However this new file should contain the implementation of the interface *IModel* with the new *Calculate()* and *Initialize()* functions in C++ code formatting. An auxiliary batch file that uses a compiler that accepts command-line inputs such as *MinGW* can be used to create the DLL for the new model. The compilation process can be triggered by the user to generate the new model dll that can be loaded into the environment using *Estimator.dll*. Once the new model is loaded it is ready to be used. Figure 7 illustrates these two situations.

4. DESIGN OF THE BR-SENSOR

As shown in the previous sections, a simple on-line soft sensor can be easily implemented by just considering the component diagram presented in Figure 5. In this section a real and more complex case is considered: the problem of the estimation of the downhole pressure for an entire petroleum oil platform, the real motivation for the creation of *BR-Sensor*.

Typically a platform is connected to many, say N, oil wells and each of them was installed with a hard sensor to measure the downhole pressure (PDG-P). Due to the

necessity of maintaining and optimizing production, there is a great interest in always having *PDG-P* information even if the hard sensor fails. In order to achieve several estimates of the *PDG-P* pressure, M different estimators for each oil well were implemented. One possible solution would be to combine those estimates using some function. The idea is to provide the best possible estimate for the downhole pressure. Note that each estimator has a process model and a observation model as can be seen in Figure 9.

By taking advantage of the *IModel* interface described in Section 3.4, the object component "*Estimator.dll*" that appears in the component diagram in Figure 5 will be extended to a new one, called "*BR-Sensor.dll*". This new DLL can deal with a multi-estimator and multi-well scenario (See Figure 9). For this specific task, the object-oriented programming is the perfect tool and Section 4.2 contains the description of the class used for modelling.

4.1 The BR-Sensor's class diagram

To design a class diagram from the hierarchy depicted in Figure 9 is a straightforward process. Basically, each level becomes a class and the result can be seen in Figure 8 on page 6.

Some important comments regarding the class diagram are listed below:

- The classes "*C_Platform*", "*C_Well*" and "*C_Estimator*" have, as attribute, a pointer to the variables of the user-defined data structures called "*Platform*", "*Well*" and "*Estimator*" respectively. These variables are instantiated as global variables in the *BR-Sensor*'s DLL and are a mirror of the tag database of the User's Project (UP) *BR-Sensor* that exist in the *BR-Suite* (See Section 2).
- The class "*C_Estimator*" is an abstract class which allows the designer to write several implementations of sequential estimators algorithms of his/her choice (EKF, UKF, UKF-SR and etc.). For simplification, Figure 8 shows just one derived class, the "*C_UKFEstimator*" class, that implements the *UKF* algorithm. Other derived classes are omitted.

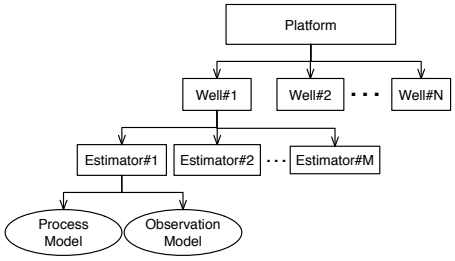

Fig. 9. BR-Sensor's object structure.

4.2 The BR-Sensor's user interface

To configure the execution of *BR-Sensor* it is necessary to define a scenario:

- which are oil wells whose pressure downhole are estimated;
- number and type of estimators for each one of the oil wells;
- which are the actual models (process and observation) used for each estimator.

A user interface module was developed in the *BR-Suite* environment and the screen used by the operator to register, configure and see the on-line results is shown in Figure 10.

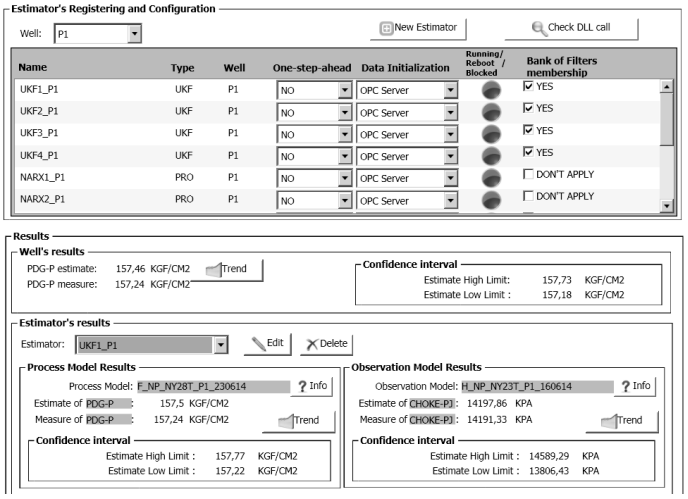

Fig. 10. An example of the *BR-Sensor*'s user graphical interfaces. Here the operator can register, configure and keep updated with the estimator results.

4.3 The BR-Sensor's computational performance

The computational performance of *BR-Sensor* was measured in terms of time consumption. The idea is to measure the mean time taken by *BR-Sensor* when executing the following tasks for each well: execution of all of the implemented estimators, consolidation of their outputs and file access time when saving the results. In addition the mean time needed by two different class of estimators was measured. One estimator, Type-1, is very simple and gives the one-step-ahead prediction of process models, the other one, estimator Type-2, runs the *UKF* algorithm. For Type-1 estimators it has been also noticed a difference in time consumption between NARX and NN models. For the simple performance analysis conducted here, *BR-Sensor* was configured with 15 wells and each of them with 4 Type-2 estimators using NARX models. *BR-Sensor* ran 1000 estimation iterations and the results are shown in Table 1.

Table 1. Computational performance measured in mean time consumption per execution unit over 1000 iterations.

Platform	Well	Estim. Type-1		Estim. Type-2
		NARX model	NN model	NARX Model
117.8ms	7.8ms	0.02ms	0.25ms	0.70ms

Since the typical period required by the operator to receive a new pressure estimate is one minute, the results in Table 1 shows that *BR-Sensor* meets the requirements. Note that the time period should be sufficient for *BR-Suite*, *BR-Optimus* and *BR-Sensor* to perform their cyclic tasks and the results in Table 1 shows that *BR-Sensor* had a low-level time consumption. The higher complexity of estimator Type-2 will certainly reflect in the time consumption.

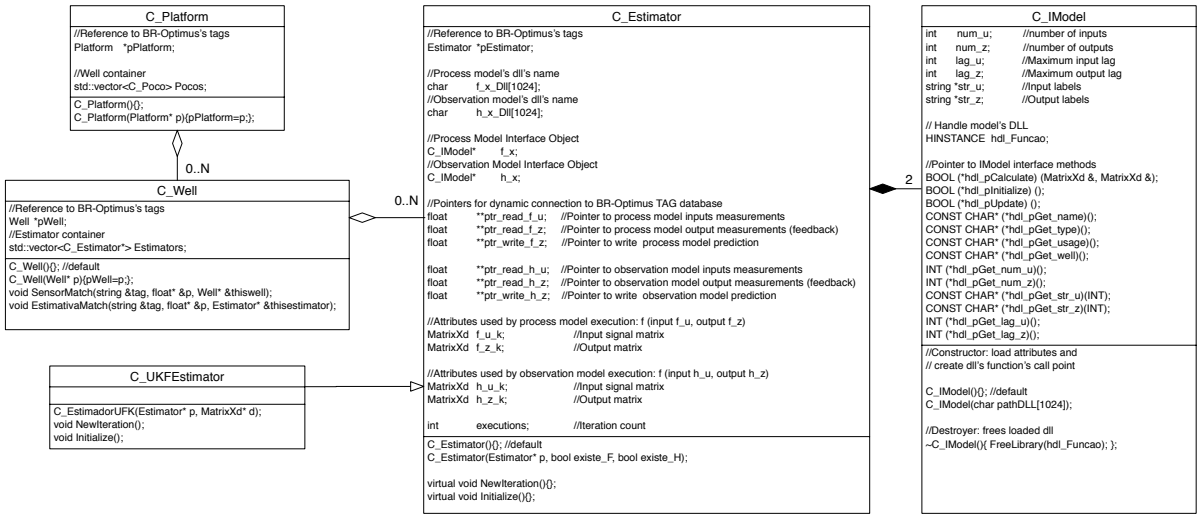

Fig. 8. *BR-Sensor*'s class diagram.

For instance Neural Networks based models, specifically Multilayer Perceptrons type, requires about 10 times the time required by a NARMAX model. Although this is not a fixed relation since it depends on the number of neurons and layers, it shows that complex representations will be more time consuming.

5. CONCLUSION

In this work an object-oriented approach for designing soft sensor components, encapsulated as dynamic-link-library (DLL), was proposed. To accomplish this, the *DLL*-based system integration technology combined with object-oriented programming approach was used to implement a soft sensor for the estimation of downhole pressure for an entire oil platform. A systematic methodology for implementing mathematical models as *DLLs* was presented and shown to be easily modified depending upon the user demands.

A simple but interesting performance analysis of the implemented soft sensor was made. It was shown that the time consumption of *BR-Optimus* meets the requirements and as consequence the solution proposed here can be deployed in a real production environment.

Another advantage of this methodology was to show that a platform operator can easily mastered the *IDataAndControl* interface to integrate a user DLL to the environment *BR-Optimus*.

ACKNOWLEDGEMENTS

The authors acknowledge the financial support from Petrobras.

REFERENCES

Bates, D. and Eddelbuettel, D. (2013). Fast and elegant numerical linear algebra using the RcppEngine package. *Journal of Statistical Software*, 52(5).

Chen, S. and Billings, S.A. (1989). Representations of nonlinear systems: the NARMAX model. *International Journal of Control*, 49(3), 1013–1032.

Cybenko, G. (1989). Aproximation by superpositions of a sigmoidal function. *Mathematics of controls, signals and systems*, 2(4), 303–314.

Daum, F. (2005). Nonlinear filters: Beyond the Kalman Filter. *IEEE A&E Systems Magazine*, 20(8), 57–69.

Garcia, C., , Berni, C.C., and Oliveira, C.E.N. (2008). Hardware/firmware implementation of a soft sensor using an improved version of a fuzzy identification algorithm. *ISA Transactions*, 47, 157–170.

Herbst, J.A. and Pate, W.T. (1999). Object components for comminution system softsensor design. *Powder Technology*, 105, 424–429.

Julier, S.J. and Uhlmann, J.K. (2004). Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3), 401–422.

Kadlec, P., Gabrys, B., and Strandt, S. (2009). Data-driven soft sensor in the process industry. *Computers and Chemical Engineering*, 33, 795–814.

Li, Y.G., Gui, W.H., Yang, C.H., and Xie, Y.F. (2013). Soft sensor and expert control for blending and digestion process in alumina metallurgical industry. *Journal of Process Control*, 23, 1012–1021.

Rafiq, M.Y., Bugmann, G., and Easterbrook, D.J. (2001). Neural networks for engineering applications. *Computers and Structures*, 79, 1541–1552.

Salvatore, L., Souza, M., and Campos, M. (2009). Design and implementation of a neural network based soft-sensor to infer sulfur content in a brazilian diesel hydrotreating unit. *Chemical Engineering Transactions*, 17, 1389–1394. doi:10.3303/CET0917232.

Simon, D. (2006). *Optimal state estimation*. John Wiley & Sons.

Teixeira, B.O.S., Castro, W.S., Teixeira, A.F., and Aguirre, L.A. (2014). Data-driven soft sensor of downhole pressure for a gas-lift oil well. *Control Engineering Practice*, 22, 34–43.

Zhou, P., Chai, T., and Sun, J. (2013). Intelligence-based supervisory control for optimal operation of a DCS-controlled grinding system. *IEEE Trans. on Control Systems Technology*, 21(1), 162–175.