Software development of components for complex signal analysis on the example of adaptive recursive estimation methods.

SIMON BOYMANN, RALPH MASCHOTTA, SILKE LEHMANN, DUNJA STEUER
Institute of Biomedical Engineering and Medical Informatics
Ilmenau Technical University
POB 100565, D-98684 Ilmenau
GERMANY

Abstract: - One natural way to develop new complex signal processing strategies is to combine basic algorithms. Therefore we need these algorithms as components. Using object-oriented technologies, it is easy to build components from algorithms. This can be demonstrated by the example of adaptive recursive estimation methods. Components must be able to dock to each other and to communicate. Furthermore, the components must detect some major sources of errors and react effectively. With the help of these components we will show how complex adaptive recursive estimation methods can be created and how they can be implemented.

Key-Words: - software development, signal analysis strategies, components, adaptive recursive estimation method, object-oriented technologies,

1 Introduction

Recent developments of powerful computer equipment combine with modern information processing methods such as object-oriented technologies to stimulate new opportunities for the design of algorithms addressing complex signal and image processing problems. Also, there is great interest from partners in the industry in new algorithms that fit specific signal processing problems. However, the technology transfer from academia to industry was hampered by the lack of a common interface. Both sides require algorithms that come with an interface both of these partners can utilize. One very natural approach to building complex signal processing strategies is to concatenate basic algorithmic components into a network that meets the set demands. Using adaptive recursive estimation methods as an example, we describe how such components may be realized as object-oriented structures, and how they could be combined.

2 Problem Formulation

2.1 Adaptive recursive estimation methods[1]

Adaptive recursive estimation methodes have a common structure.

$$S_0 = s_0$$
$$S_{n+1} = S_n + c_n \cdot K(S_n, x_{n+1}) \qquad (1)$$

From these basics we can construct simple adaptive procedures.

Adaptive Mean

$$M_{n+1} = M_n + c \cdot (x_{n+1} - M_n) \qquad (2)$$

Adaptive second moment

$$E_{n+1} = E_n + c \cdot (x_{m+1}^2 - E_n) \qquad (3)$$

Adaptive Quantil

$$Q_{n+1} = \begin{cases} Q_n + c \cdot \alpha & if \quad x_{n+1} > Q_n \\ Q_{n+c} \cdot (1-\alpha) & if \quad x_{n+1} < Q_n \end{cases} \qquad (4)$$

Their structure makes them suitable for object-oriented programming.

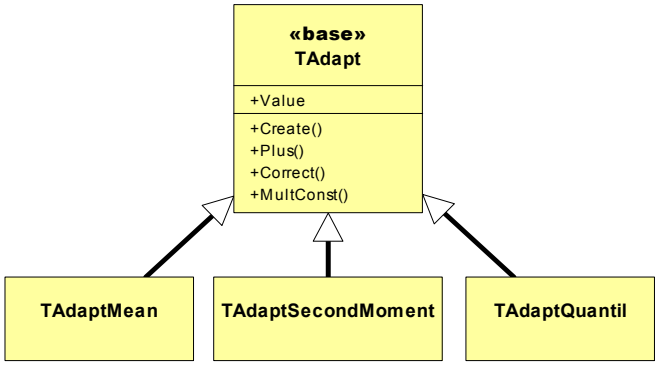

Fig. 1: Class diagramm of basic adaptic recursive estimation methods

2.2 Complex signal processing strategy

Prerequisite to building signal processing strategies are primitive algorithms implemented as components. These components must be able to dock to each other and to communitate with each other.

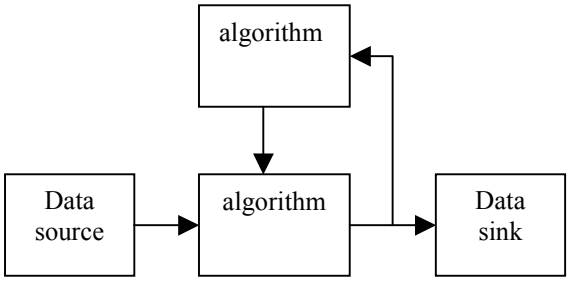

Fig. 2: Example pipeline

In figure 2 some simple components are docked to each other. We will call the combination of components a "pipeline"[2]. A component can have entrances and gates. It is obvious that the components also have to implement mechanisms to control the flow of the data.

Although the flow of the data is always from the data source to the sink, the request to perform a calculation could appear on both ends of the pipeline. The first situation is an example for an offline analysis. The data source is a file with megabytes of EEG-data for example. The data sink is a viewer simple displaying the result of the strategy. It is impractical to perform the calculation on the entire data source. It is more effective to perform the calculation only on the data currently displayed by the data sink. Here, calculation is triggered by the sink. The second situation is an example for an online analysis. The data source produces a new data item from time to time and the sink has to display the new result. In this case the calculation is triggered by the source.

The remarks above indicates that the amount of data transported through the pipeline changes. It could be an single data item, a data block with a special size to perform a FFT for example, full pictures or even series of pictures.

Another fact we have to consider is that splitting of the data flow and loops are definitely allowed and wanted. Therefore, we have to provide mechanisms to prevent deadlocks.

So far we discussed only components with one gate. But imagine a data source representing again a file of EEG data. In this file, more important information is contained than just the readings, for example, the sample rate. Thus we must design components with multiple entrances and gates.

3 Problem Solution

3.1 Docking and communication

First we concentrate on the docking of the components and the communication. We have already identified two different requests to the pipeline causing it to perform the calculation. The first one we will name a "call" request (fig. 3). It starts from the source and causes all components in the pipeline to perform their calculation, including the data sink, and ends by returning to the source. The result of one component is sent to the next component as a parameter of their "call" method. The second request we will name a "recall" request. Starting from the end of the pipeline, this request will cause the preceding component in the pipeline to deliver the result of their calculation based on a block of data with a special size and position within the available data. The result of the preceding component is returned to the requesting component by the preceding components "recall" method.

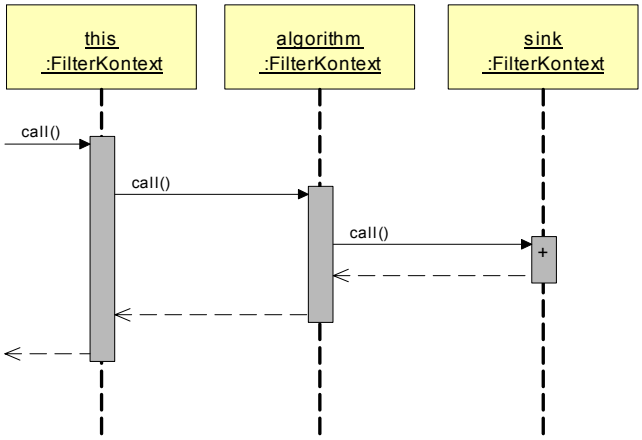

Fig. 3: Sequence diagramm of a "call" request

Now think of a split of the data flow. For example there could be two different data sinks. Both request the result of one preceding component based on the same data. This will cause double calculations although the same result will be displayed. We need to manage the results of each component better. This is realized by adding a component called "pipe"[2] between the components performing a calculation. These components we will name "filter"[2]. One other big advantage of these pipes is that the filter whose major responsibility is the calculation, freed from managing the data. The filters calculate their results based on the data provided by their entrance-pipes and submit the results to their gate-pipes.

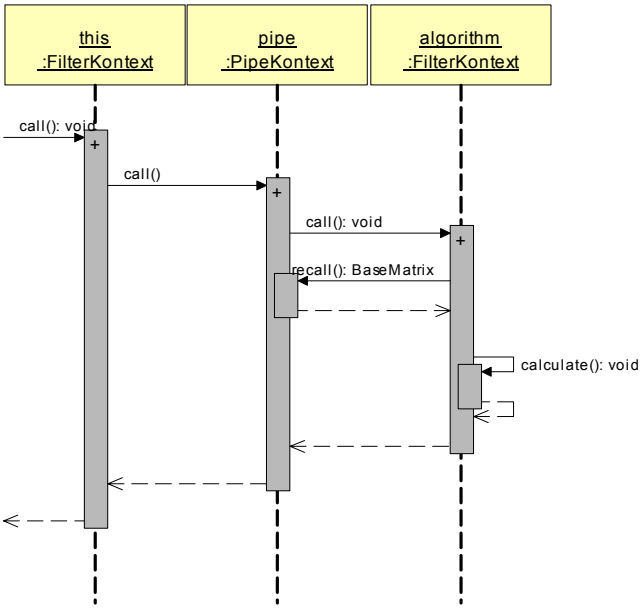

Fig. 4: Sequence diagramm "call" request with pipes and filters

The connection between the components is a simple reference to each other. Every filter has a list of references to its entrance-pipes and gates and every pipe owns a list of its gate-filters and a reference to the entrance-filter. The interface to the components is realized by an abstract class. This result is in the class diagram provided in fig. 4.

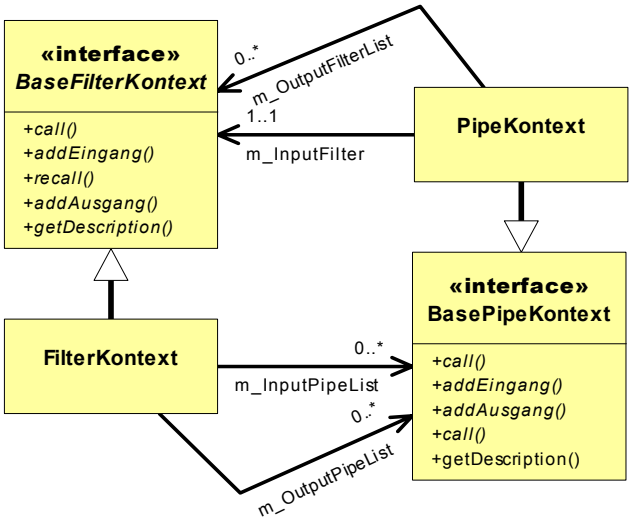

Fig. 5: Class diagram of the context

When a filter finishs its calculation, it will sequentially call the "call" method of every referenced gate-pipe in its list. So do the pipes. The "recall" request works similar.

3.2 Pipeline control

To control the pipeline we add two more classes to our class structure, a class to control the pipe and one class to control the filter. By controling these components we can control the whole pipeline. The control class of the filter is also responsible for supplying the algorithm with the correct data from the entrance-pipes. This is done by a mapping from a description of the algorithm to the list of entrance-pipes. But this topic is not discussed in this paper.

A first error we would like to address is introduced by backcoupling. This may cause a filter to request data from itself. In the case of the first calculation requested of the filter, it maybe sensible to use default data in place of the data not yet available. Otherwise, the filter will have to produce an error. This can be implemented by defining two flags, "isrecalling" and "firsttime". The very first time a filter asks an entrance pipe for data, both flags will be set to "true". The flag of "firsttime" is set to "false" after the first calculation and "isrecalling" is set to "false" evere time a request returned.

The next problem appears if a filter calls itself. We define a flag "iscalling" which is set to "true" when the filter calls its gate pipes, a reset if the calls returned successfully. If the filter is called while "iscalling" is "true", the filter produces an error.

The following control mechanism is implemented in the pipes. Think of a data source with two gates. One delivers the readings and the other one the sample rate of data acquisition. The next filter performs a FFT on the data and also needs the sample rate. The data source will call its first gate and the connected pipe the filter to do the calculation. The filter requests through the pipes the information from both gates of the source and performs. In the next step the source will call the other gate. The same happens again. The FFT-filter performs the calculation two times with the same information. This must be prevent by the pipes. A "call" request can only pass a pipe if the data saved by the pipe is not requested already by a special filter. Otherwise, the pipe would not call this special filter, but all other filters connected to it as gate-filter. Therefore, the pipe has to identify the filter which is calling its "request" method and mark the gate the filter is connected to as "allreadyrecalled". So we need a flag for every filter connected to the pipe. These flags are reset every time new data are written to the pipe.

The pipe also checks if requested data match the data it has saved already. In this case the pipe return the data without causing a new calculation of its entrance-filter.

3.3 Structure of components

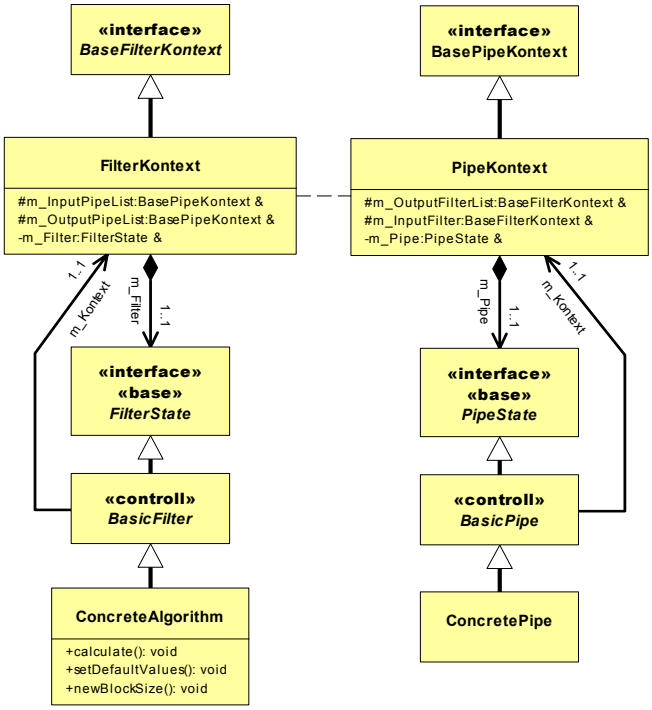

Fig. 6: Class diagram of the components

The class structure in figure 5 shows a component to consists of three parts:

- The context responsible for the communication and connection with the neigbour components.
- The base classes doing the controlling the pipeline, and
- the concrete classes implementing the algorithm.

To implement an algorithm only the calculate method must be implemented. If default values for the special case discussed above shall be set, the method "setDefaulfValues" must be implemented and if the algorithm needs a special amount of data the "newBlockSize" method will have to be invoken.

4 Conclusion

From the current state of development we can easily create the basic adaptive recursive estimation methods shown in figure 1 as components. The class "TAdapt" will just have to inherit from "BasicFilter" and implement the "calculate" method. Thus we build a small library of basic adaptive recursive methods.

4.1 Signal processing strategies by aggregation

A simple example for a combination of basic components is the adaptive variance.

$$V_{[C_1,C_2]}(X) = E^{C_2}(X) - \left(M^{C_1}(X)\right)^2 \qquad (5)$$

This arithmetic expression can be transformed in a pipeline as shown in figure 6.

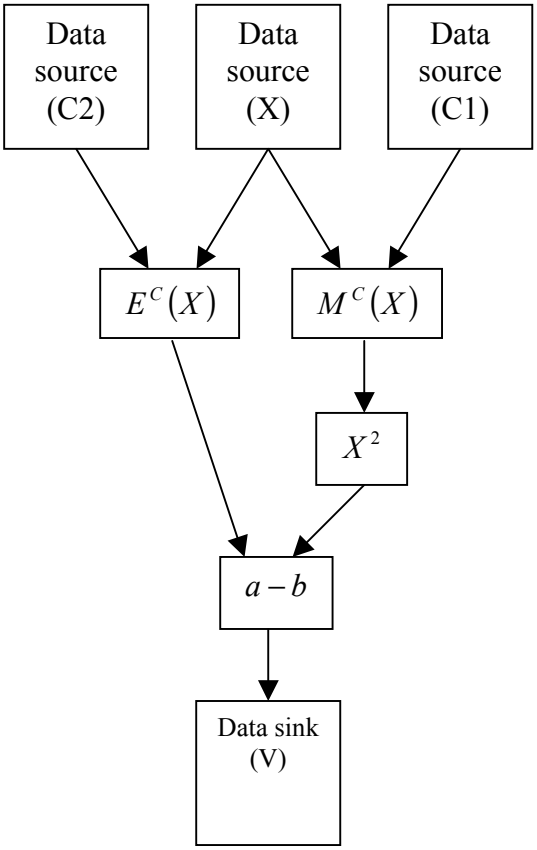

Fig. 7: adaptive variance pipeline

Building pipelines from basic components is especially useful during the initial stages of development of a new signal processing strategy. This approach allows frequent changes to components and subsequent observation of the resulting performance of the strategy.

Obviously, the control mechanisms within the components cause some performance drawbacks. At runtime, a single module executing the job of the pipeline will always yield superior performance. Therefore, after satisfactory evaluation of the strategy, you may want to reimplement it for optimized execution.

4.2 Implemention of complex signal processing methods

The use of object-oriented techniques encourages the implementation of complex algorithms. Two main parts of the adaptive variance are implemented already, the

adaptive mean and the adaptive second moment. To implement the adaptive variance the new class must inherit from the class “BasicFilter” and implement the “calculate” methode using the adaptive mean and the adaptive second moment. The adaptive variance can in turn be used as a component in other signal processing strategies.

In the future we plan to implement librarys with various signal processing algorithms these supplemented by an application to manage and combine these components. Based on the structure of the components, we hope that this application can fullfill the demands of a wide variety of today’s and tomorrow’s signal processing problems. Because the interfaces of the components were designed in close contact with our partners from industry we hope to improve ourcooperation.

Supported by TMWFK: B699-00011

References:

[1] D. Steuer, G. Grießbach; Object-oriented realization of complex adaptivee recursive estimation methods in biosignal analysis, *Proceedings of 3rd International Workshop on Biosignal Interpretation (BSI99)*, Chicago 1999, 245-248

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal ; *A System of Patterns: Pattern-Oriented Software Architecture*, West Sussex, England, John Wiley & Sons. 1996