COMPUTER METHODS AND PROGRAMS IN BIOMEDICINE 85 (2007) 129–137



journal homepage: www.intl.elsevierhealth.com/journals/cmpb



Java-based framework for processing and displaying short-echo-time magnetic resonance spectroscopy signals

B. De Neuter, J. Luts, L. Vanhamme, P. Lemmerling, S. Van Huffel*

Katholieke Universiteit Leuven, Department Electrical Engineering, ESAT-SCD, Belgium

ARTICLE INFO

Article history:
Received 2 December 2004
Received in revised form
21 March 2006
Accepted 1 September 2006

Keywords:
Java
Application framework
Magnetic resonance spectroscopy (MRS)
Quantification

ABSTRACT

Magnetic resonance spectroscopy (MRS) can be used to determine in a non-invasive way the concentrations of certain chemical substances, also called metabolites. The spectra of MRS signals contain peaks that correspond to the metabolites of interest. Short-echo-time signals are characterized by heavily overlapping metabolite peaks and require sophisticated processing methods. To be useful in a clinical environment tools are needed that can process those signals in an accurate and fast way. Therefore, we developed novel processing methods and we designed a freely available and open-source framework (http://www.esat.kuleuven.ac.be/sista/members/biomed) in which the processing methods can be integrated. The framework has a set of abstract classes, called hot spots, and its goal is to provide a general structure and determine the control flow of the program. It provides building blocks or components in order to help developers with integrating their methods in the framework via a plug-in system. The framework is designed with the unified modeling language (UML) and implemented in Java. When a developer implements the framework he gets an application that acts like a simple and user-friendly graphical user interface (GUI) for processing MRS data. This article describes in detail the structure and implementation of the framework and the integration of our processing methods in it.

© 2006 Published by Elsevier Ireland Ltd.

1. Introduction

Concentrations of certain chemical substances can be determined in a non-invasive way by using magnetic resonance spectroscopy (MRS). These chemical substances are also called metabolites. A metabolite corresponds to one or more peaks in the spectrum from a MRS signal. The area under a peak or a set of peaks is related to the concentration of the corresponding metabolite. By quantifying the areas under the peaks one gets estimated values that are correlated with the concentrations of the metabolites of interest. A short-echo-time MRS signal contains more information about more metabolites than a long-echo-time MRS signal. The drawbacks of short-echo-time MRS signals are the lower signal-to-noise ratio (SNR), heavily overlapping peaks and the presence of a broad baseline. These three factors make it difficult to make an accurate quantification. Prior knowledge is extensively used to tackle these problems [1].

LCModel [2,3] and magnetic resonance user interface (MRUI) [4,5] are two widely used software packages for quantifying short-echo-time MRS data. MRUI contains the time-domain methods AMARES (advanced method for accurate, robust and efficient spectral fitting) [6] and QUEST (quantitation based on quantum estimation) [7]. The method AMARES belongs to the class of interactive time-domain methods and fits a parameterized model function to the data by minimizing a cost function. LCModel uses a method that works in the frequency-domain. Both LCModel and QUEST fit

* *Corresponding author*. Tel.: +32 16 321065; fax: +32 16 321970.
E-mail address: jan.luts@esat.kuleuven.be (J. Luts).
0169-2607/$ – see front matter © 2006 Published by Elsevier Ireland Ltd.
doi:10.1016/j.cmpb.2006.09.005

a signal with a sum of adjusted spectra coming from a set of in vitro measured or simulated metabolites.

It takes time to develop software tools and design a graphical user interface (GUI). It would be convenient if big parts of the code can be reused in an easy and straightforward way. For this reason a framework [8,9] was created in Java in which one can integrate various methods. A unique definition for a framework does not exist. The definition one can find in Ref. [10] is probably the most referenced one: 'A framework is a set of classes that embodies an abstract design for solutions to a family of related problems'. A framework can be seen as a software architecture that has its influence on both design and code. It provides a set of hot spots, which are just abstract classes from the implementation point of view. In this way hot spots are flexible points in the framework that are used to connect code to the framework. Every instantiation of the hot spots creates a new application. All the code that is contained in the framework can be easily reused.

A modular design is used to make the framework more flexible. Central in this design is the plug-in system. Although this plug-in system is inspired by and influenced by the plug-in architecture of the eclipse platform [11], no code of eclipse is used in the plug-in system. In Java it is possible to package and distribute code in a jar file. A jar file is an archive file that uses the ZIP file format. It can be used for data compression, archiving and decompression. In addition a jar file has some advanced functionalities including electronic signing, version control and package sealing. This is possible because a jar file has a manifest file that contains information about the files packaged in it. A plug-in in the framework is a piece of software that is stored in a jar file and is placed in the directory plug-in, which one can find in the installation directory. Every time the application is started, it will first search in this directory for all available plug-ins. In this way it is easy to add or remove plug-ins. Because the framework does not know until run-time which plug-ins are available, it will use the manifest file from the plug-ins to identify the functionalities and to locate certain classes in it. For example, the plug-in dspGUI has an entry 'InitGUI: dspGUI.Init' in its manifest file. When this manifest file is investigated at run-time, the framework will know that the class dspGUI.Init will initialize the GUI of the plug-in. When there is more than one value for an entry, they will be separated with a semicolon. From now on when referring to the entries of a plug-in, the entries in the manifest file from a plug-in are meant.

Section 2 discusses the general design and the ideas behind the framework. In Section 3 the specific design for the MRS part of the framework is discussed in more detail and Section 4 shows an illustration from the framework by creating an application for processing short-echo-time MRS data.

2. Design and ideas behind the framework

The goal of the proposed software package is to provide a graphical user interface for doing quantification of short-echo-time MRS signals under an open-source licence, this means that the code is freely available on our website[1] and easy to adapt to the specific needs of the user. Although the authors are inspired by the MRUI project in which they also participated, there are some clear differences with the new approach proposed in this work. First, in contrast to LCModel and MRUI the proposed framework and software package is freely available and open-source. Users can add GUI components and customize existing components. As such the framework wants to offer an easy way to add functionality in the shape of plug-ins and this leads to a very fast improvement of the software. Moreover, this allows usage in scientific as well as clinical environments. Second, the framework comes with a new time-domain method, called AQSES (accurate quantitation of short-echo-time-domain signals), for doing short-echo-time quantification [12,13]. Third, the framework provides a FIR filtering technique to remove undesired resonances from specific frequency ranges [14]. From an implementation point of view the main goals of the framework are:

- to enable reuse of code;
- easy to adjust I/O, preprocessing and processing methods;
- easy to add or remove I/O, preprocessing and processing methods;
- easy to maintain the framework.

The different components communicate through specific interfaces. The interfaces have to be flexible and well-defined. As one can see in Fig. 1 the framework is built up with different layers that are contained in a plug-in system. The plug-in system is loaded and initialized as soon as the framework is started by a little piece of code and allows us to write all the code from the framework as a set of different plug-ins. The bottom layer provides general functionalities like a plug-in manager and a general graphical user interface (GUI), which is nothing more than a frame with a menu bar and a tool bar. It is in fact a sub-framework because it provides hot spots to create applications with a simple GUI and a plug-in system. The other layers are designed and built as plug-ins. A plug-in is a little piece of code that will be discovered and loaded at run-time. The plug-in system makes the framework very flexible, allows the reuse of the code and makes it easy to extend the framework with new functionalities. On top of this general layer one can find the second layer containing the plug-ins dspSystem and dspGUI, for processing and displaying MRS data. Both are sub-frameworks and have hot spots that can be instantiated. The sub-framework dspSystem provides hot spots to load, preprocess and process MRS data. The sub-framework dspGUI makes it possible to present the MRS data to the screen and manipulate the sub-framework dspSystem layer. The plug-ins philips and siemens load the MRS data as acquired from the MRS scanner made, respectively, by the manufacturers Philips and Siemens. The plug-in Matlab loads the MRS data in Matlab format. The plug-in ecc contains code to perform eddy current correction and the plug-in hlsvdpro contains code to perform HLSVD filtering [15]. The plug-in aqses contains a method to quantify short-echo-time MRS signals [12,13]. The framework library comprises a set of plug-ins to perform specific tasks.

[1] http://www.esat.kuleuven.ac.be/sista/members/biomed.

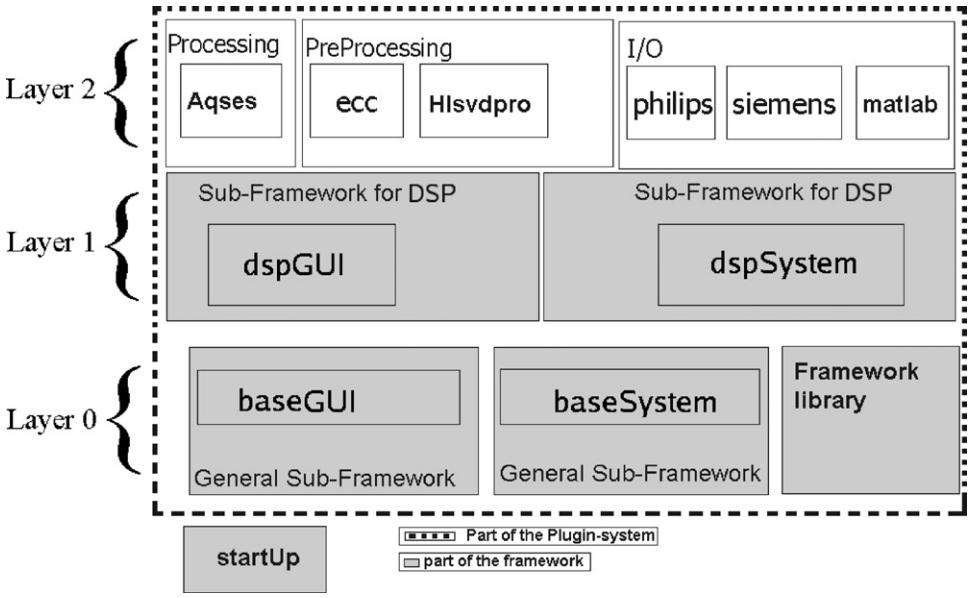


Fig. 1 – Structure of the framework. The general sub-framework baseSystem provides a plug-in system that is used to build the other components from the framework. The components in dark grey form the framework. The white components instantiate the different hot spots and the result is our application for processing short-echo-time MRS data.

3. System description

3.1. *Used technologies*

The unified modeling language (UML) [16] and design patterns [17] are used to make a blueprint of the framework so one can keep a clear overview over it. A design pattern is a model solution for a certain problem in a certain context. The implementation of the framework is done in Java [18–20]. Java is an object-oriented (OO) programming language that is well suited to create a framework because it has the concepts polymorphism, data abstraction and inheritance. Polymorphism means one can assign to a single variable or parameter values from different types. This concept is used to get a loose coupling between the different components of the framework. Abstract classes and interfaces are used to define the hot spots and allow the framework to specify methods that the user has to implement. Inheritance makes it easy to create new components from existing ones. The framework is tested on Linux and Windows, but it should work on any platform for which a Java virtual machine (JVM) exists.

3.2. *Design of a signal*

MRS signals are acquired as free induction decay (FID) signals. Fig. 2 shows the UML diagram of such a signal. Every instance from the class Signal has an association with an instance from the class Header. A Signal instance contains the time domain signal, the name of the signal and additional information. In the Header instance one can find the protocol that is used to acquire the signal. It has the begin time, field strength, sampling interval, length of the signal, transmitter frequency and the nucleus. A Header instance is used by the framework to determine if two signals are acquired with the same protocol. The interface SignalFactory is used to create instances from the class Header and the class dspSystem.Signal.

3.3. *Design of a MRS project*

To process MRS data in the framework the user has to start a new MRS project. A project contains zero or more signals that are acquired under the same protocol. In Fig. 3 one can see that an instance of the class dspProject has associations with an instance of ProjectSignal. The class ProjectSignal is a subclass from the class dspSystem.Signal and has associations with a template and a result. The processing methods in the framework are template-based. A template is a text file that describes how to preprocess and process the data. When the data is processed, it stores for each Signal the result in the corresponding object from the class Result. Working with templates has the great advantage that one can store the settings so they can be used later. This allows the user to load a template and start processing immediately.

3.4. *Processing MRS data*

In Fig. 4 one can see the different steps to process MRS data. First the data are loaded and visualized by applying the fast Fourier transformation to the time domain signal. The data are preprocessed and processed and finally the results are displayed to the screen.

3.4.1. *Loading from MRS data files*

The plug-in dspSystem (see Fig. 1) contains code to read and load data files from the different manufacturers. The framework can read files coming from the Philips and Siemens NMR scanner. It can also read files stored in Matlab format. The framework provides a hot spot FileFormat (see Fig. 5) for load-

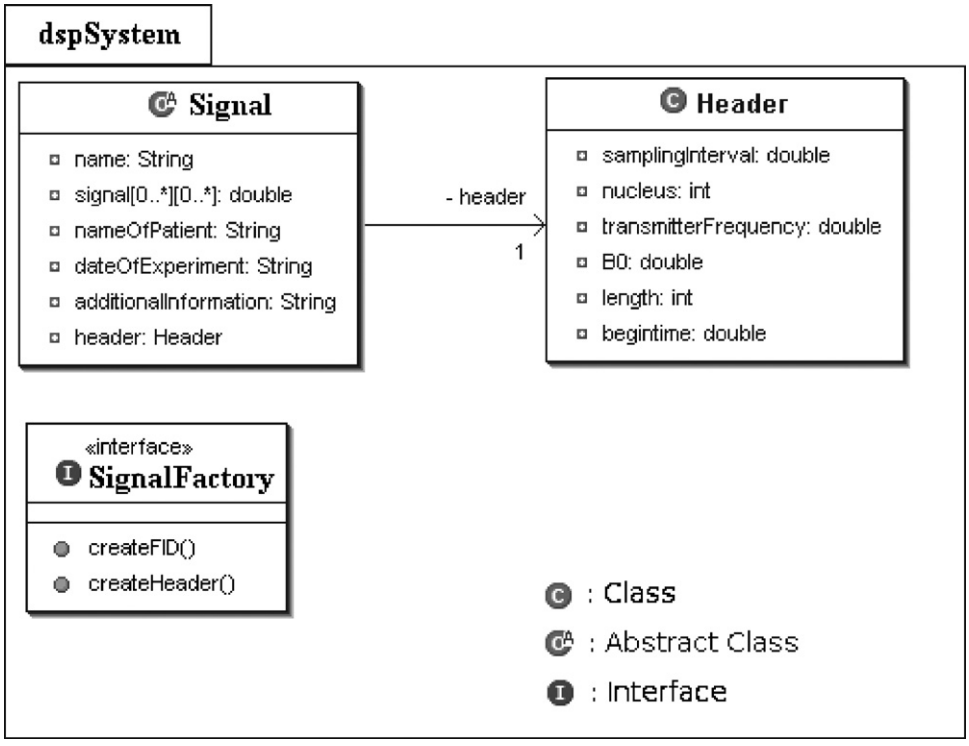


Fig. 2 – Design of a Signal. Every instance of the class FID has an association with an instance of the class Header.

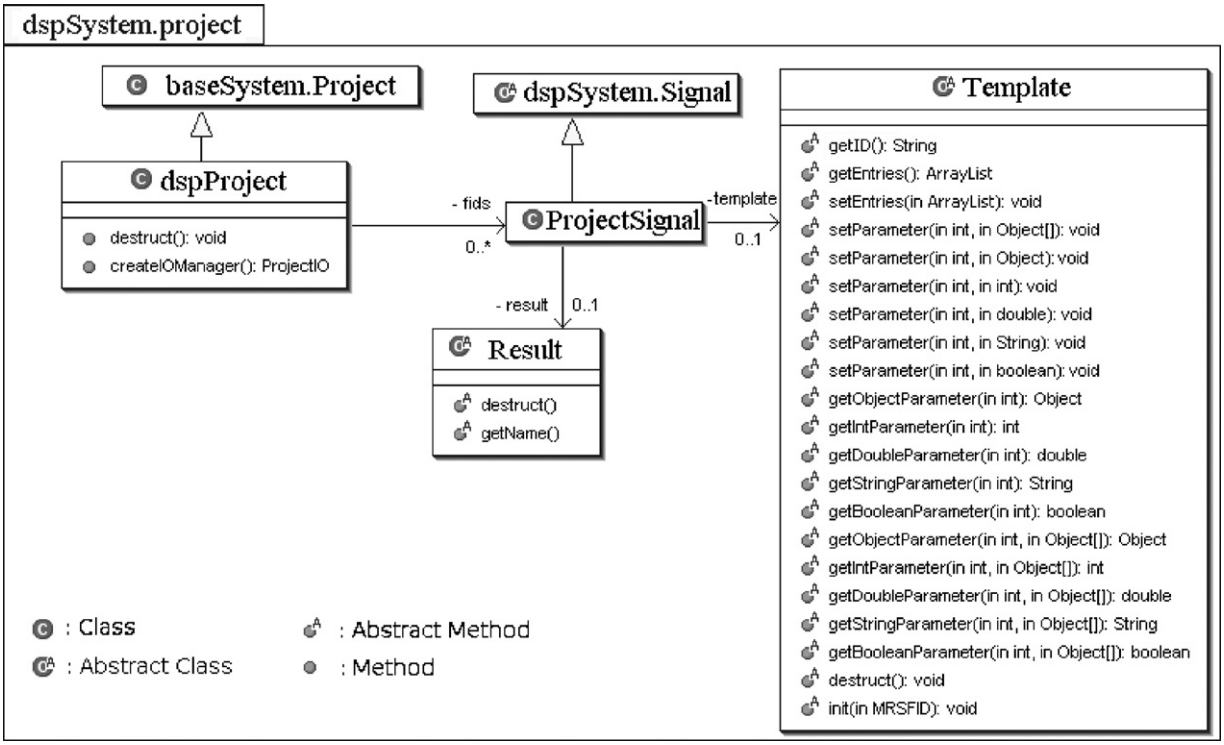


Fig. 3 – The class diagram from a MRS project. A MRS project is template based. Every Signal in the project is associated with a template. This template contains the settings for the current processing method.

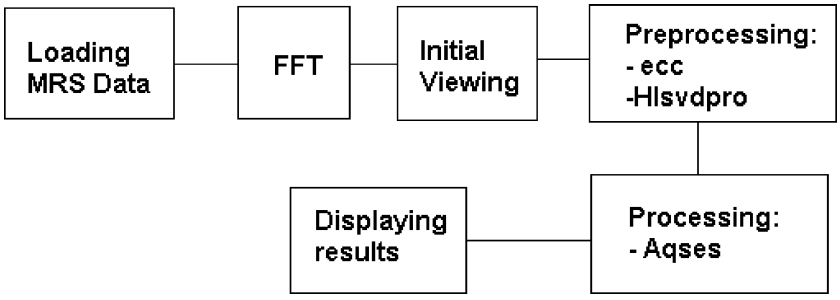


Fig. 4 – Flow chart for processing MRS data.

ing MRS data from different manufacturers. The class IOManager will manage the instantiations of this hot spot and is implemented using the singleton pattern. When the singleton instance of this class is created, it will check the entries from the plug-ins for an entry dspIO. This entry will specify the instantiations of the hot spot. An example is the plug-in philips. This plug-in has the class Philips, which is a subclass from FileFormat, and is used to read data coming from a Philips NMR scanner. This plug-in has the entry 'dspIO: philips.io.Philips' in its manifest file. The singleton instance of the class IOManager will use the chain of responsibility pattern to organize the different instantiations of the hot spot FileFormat. This pattern creates a chain of objects through which a request can be passed. This is an easy way to decouple the sender of a request from the different receivers. In this case the instance from IOManager is the sender and the different instantiations of the hot spot are the receivers. When a file needs to be loaded it will pass this file through the chain until one of the instantiations can load it. If the file reaches the end of the chain an exception will be thrown. In this way one can add dynamically new plug-ins to load data from manufacturers.

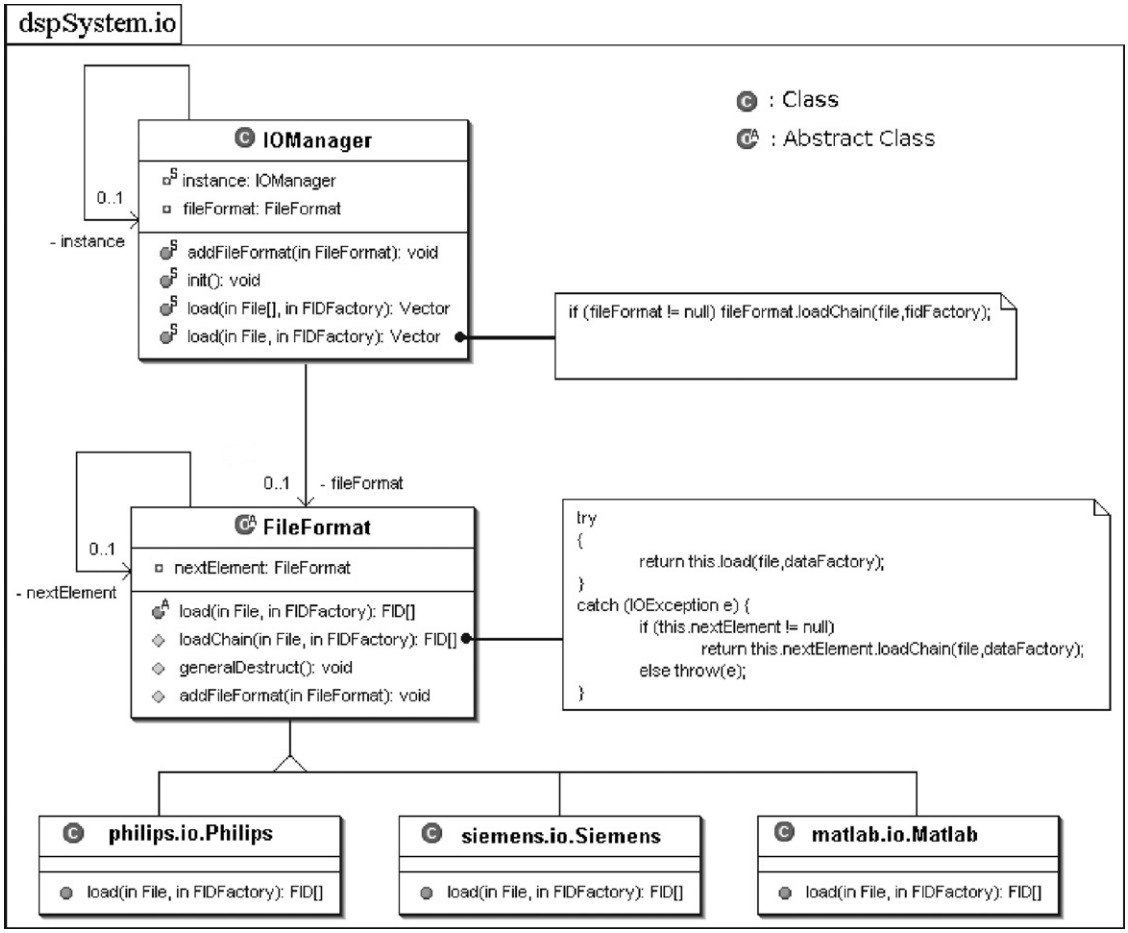


Fig. 5 – Loading of MRS data files. The chain of responsibility pattern is used to design and implement the classes that are responsible for reading the data files from the different manufacturers.

3.4.2. *Preprocessing*

There are two hot spots for preprocessing. One is the abstract class PreprocessingMethod that has three methods for preprocessing the data. The first method will preprocess only one Signal, the second one will preprocess an array of Signal objects and the third one will preprocess all Signal objects available in the project and return new objects from the class Signal:

- public abstract void preprocess(Signal signal, Object[] param);
- public abstract void preprocess(Signal[] signal, Object[] param);
- public abstract Signal[] preprocessAll(Signal[] signal, Object[] param);

The other hot spot is the interface PreprocessFactory and creates new instances of the class PreprocessingMethod. This interface is designed with the abstract factory pattern. A plug-in that instantiates these hot spots should have an entry dspPreprocessing to specify the factory class. The class PreprocessingManager will keep track of the preprocessing algorithms. Every preprocessing algorithm should have a unique name. To preprocess the data in a project, one passes the name of the selected preprocessing method and the parameters for this method to the project. The project will try to locate the correct factory class, create a new instance of the class PreprocessingMethod and start the preprocessing. The parameters passed to the method are stored in an array from type Object. The class Object is the root of the class hierarchy and all other classes are subclasses from this class. In this way the framework does not know which parameters are passed and it will be loosely coupled to the preprocessing methods. It is up to the user to decide what parameters will be passed. Preprocessing will be performed immediately on the data.

3.4.3. *Processing*

The class ProcessingMethod is a hot spot and will be used by the project to start the processing. The user can use this hot spot to integrate his processing method in the framework. The settings for a processing method are in an instance of the class Template. Every instance of ProjectSignal has a reference to an instance of Template. In that way the user can tune the settings from the processing method for each Signal individual. A template can be stored in a text file that the user can edit with a regular text editor. This makes it easier to reuse the settings. The class Template has the following set of methods to set and to get the settings of a template:

- public abstract void setParameter(int index, Object[] param);
- public abstract void setParameter(int index, Object param);
- public abstract void setParameter(int index, int param);
- public abstract void setParameter(int index, double param);
- public abstract void setParameter(int index, String param);
- public abstract void setParameter(int index, boolean param);
- public abstract Object getObjectParameter(int index);
- public abstract int getIntParameter(int index);
- public abstract double getDoubleParameter(int index);
- public abstract String getStringParameter(int index);
- public abstract boolean getBooleanParameter(int index);
- public abstract int getIntParameter(int index);
- public abstract Object getObjectParameter(int index,Object[] param);
- public abstract int getIntParameter(int index,Object[] param);
- public abstract double getDoubleParameter(int index,Object[] param);
- public abstract String getStringParameter(int index,Object[] param);
- public abstract boolean getBooleanParameter(int index,Object[] param);
- public abstract int getIntParameter(int index,Object[] param);

The methods accept an integer index noting which setting has to be changed or returned. These indices are defined by the processing method itself. Because it is the user who decides to change a setting at run-time, the plug-in dspSystem does not need to know the exact meaning of the indices. It will just pass the messages between the user and the template. After processing the data, the results are stored in instances of Result. The class ProcessingMethodFactory is a factory class and is used to create the instances from the classes ProcessingMethod and Template. To find the available factory classes, an entry dspProcessing is introduced. In the current application a method for the quantification of short-echo-time MRS data that is packaged in the plug-in aqses was developed. But because the hot spots are flexible, it is easy to create new processing methods that are completely different.

4. Application for processing short-echo-time MRS data

In this section the use of the framework and the effort that must be expected to extend it is demonstrated. Users can for instance extend the framework to load new file formats, implement new preprocessing methods or implement new processing methods. This means that they should provide a plug-in with an implementation in Java or code that can be connected to Java for the hot spots of the framework as well as a single manifest file for each plug-in. To illustrate this an application with a new method for processing short-echo-time MRS signals in the time domain [12,13] is presented. The method imposes prior knowledge by using a database containing in vitro measured model solution signals. The MRS time-domain signal is fitted by a sum of adjusted in vitro metabolite time domain signals (acquired under the same experimental conditions) and a baseline constructed with splines. The adjustments of the in vitro metabolite signals are modeled by frequency, damping, phase and amplitude parameters. Prior knowledge between these parameters can be imposed. In a mathematical formulation the in vivo measured MRS time-domain signal y of length m satisfies the semiparametric model:

$$y(t) = \hat{y}(t) + \varepsilon_t := \sum_{k=1}^{K} \alpha_k (\zeta_k)^t v_k(t) + b(t) + w(t) + \varepsilon_t,$$
$$t = t_0, \ldots, t_{m-1}, \quad (1)$$

where $\alpha_k, \zeta_k \in \mathbb{C}$ are unknown parameters that account for the amplitudes of the metabolites in the database and the corrections to the database signals $\{v_k, \text{ for } k = 1, \ldots, K\}$. The nonparametric term $b(t)$ represents the baseline, this is the effect of unknown macromolecular components that are not present in the database of metabolite profiles. The term $w(t)$ refers to the residual water component and the ε_t term characterizes the noise. The stiffness of the nonparametric baseline is governed by a regularization parameter λ. The parameters of the resulting model are determined by means of a non-linear least squares algorithm (Levenberg–Marquardt) for several values of λ. A generalized cross-validation test is used to determine the λ yielding the best bias-variance trade-off. Furthermore, the problem of residual water is dealt with by the inclusion of a maximum-phase FIR filter in the optimization stage [14]. The method will calculate for each metabolite a correction of the phase, amplitude, damping and frequency. These corrections are shown on the result panel (see Fig. 6). Further, Cramer–Rao bounds, adapted for semiparametric nonlinear regression [12], are used to obtain a degree of uncertainty of the quantified parameters. The metabolite time domain signals are stored in a database which can be created and manipulated with a database editor that is included in the application. The method is packaged in the plug-in aqses. The actual code to quantify the data is written in Fortran 77 and stored in a shared library. This code is platform dependent and has been compiled for Windows and Linux, making the application platform dependent. Fortran is an older language and therefore extensive and well tested mathematical libraries exist. For this reason and also for reasons of computational efficiency the method is implemented in Fortran 77. Through the Java native interface (JNI) and an interface written in C, the Fortran library is connected with the Java code in the plug-in. The effort needed for implementation is demonstrated in the next subsections.

4.1. *Instantiation of the sub-framework dspSystem*

To integrate the AQSES method in the sub-framework dspSystem one has to instantiate the following five hot spots:

- dspSystem.project.Template,
- dspSystem.project.Result,
- dspSystem.processing.ProcessingMethod,
- dspSystem.processing.ProcessingMethodFactory,
- dspSystem.io.DSPResultIO.

The corresponding implementations are:

- aqses.system.AqsesTemplate,
- aqses.system.result.AqsesResult,

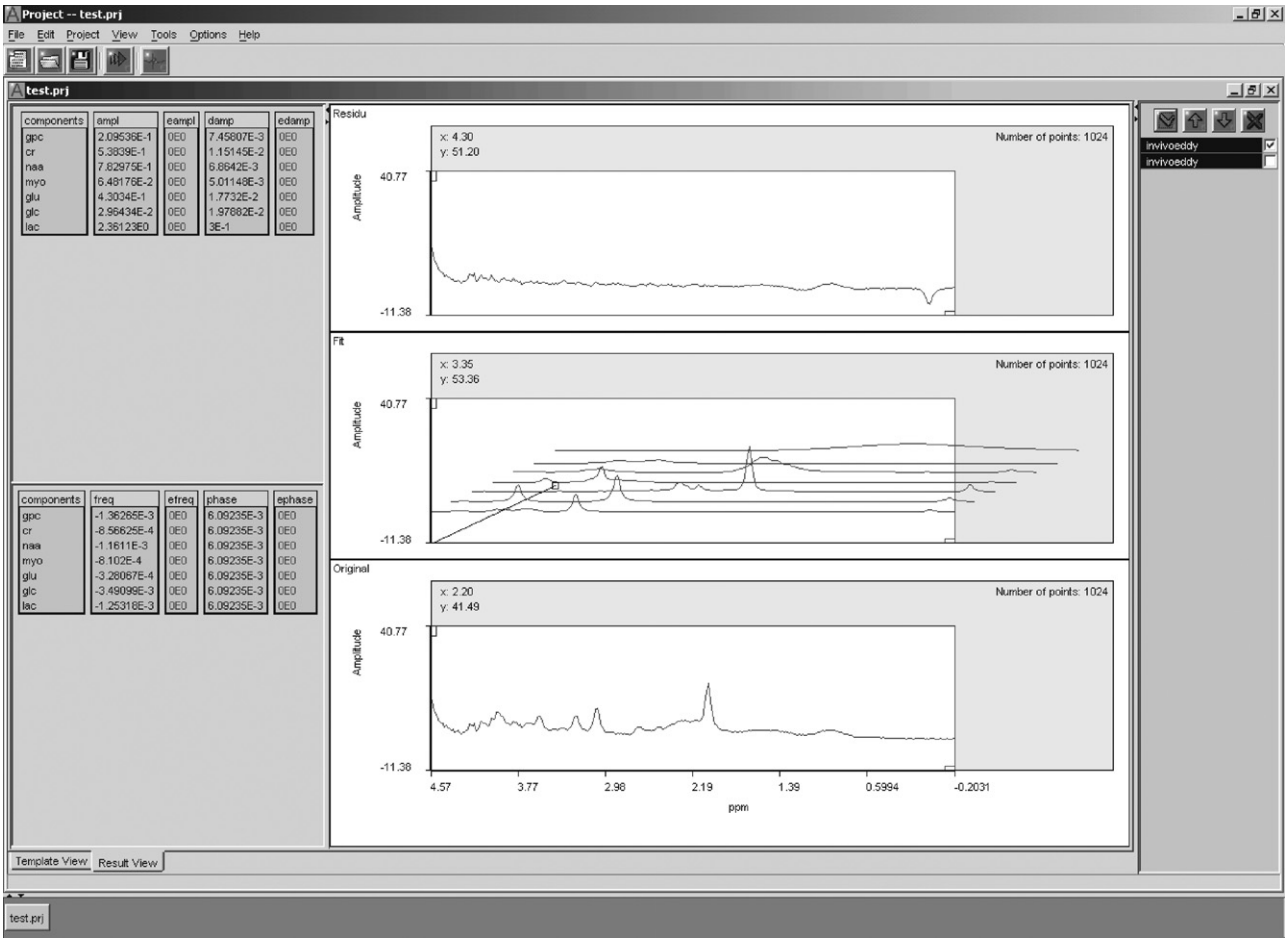


Fig. 6 – The result pane of the GUI. This panel becomes visible when there are results available for a MRS project. The contents of the panel is determined by the processing method. In our case that is the method AQSES.

- aqses.system.AqsesMethod,
- aqses.system.AqsesFactory,
- aqses.system.io.AqsesResultIO.

To the manifest file of the plug-in aqses the following entries are added:

- InitSystem: aqses.system.Init,
- dspProcessing: aqses.system.AqsesFactory.

The first entry specifies the class to initialize the system layer of the plug-in aqses. During initialization the application will determine the platform it runs on and copy the correct shared library to a working directory. In this way one can keep the application platform independent for Linux and Windows. The second entry defines the factory class that is necessary to integrate the method in the dspSystem sub-framework.

4.2. *Implementation of the sub-framework dspGUI*

To integrate the method completely in the framework one also needs to implement the hot spots from the framework dspGUI. If these hot spots are not implemented, the processing method will not be available through the GUI. This may happen in some cases where the method is only accessible through another plug-in. The method will implement the following hot spots:

- dspGUI.processing.ProcessingGUIMethod,
- dspGUI.processing.ProcessingMethodGUIFactory,
- dspGUI.result.resultPanel.

The corresponding implementations are:

- aqses.gui.AqsesGUIMethod,
- aqses.gui.AqsesGUIFactory,
- aqses.gui.result.AqsesResultPanel.

The following entries are added to the manifest file of the aqses plug-in:

- InitGUI: aqses.gui.Init,
- dspProcessingGUI: aqses.gui.AqsesGUIFactory.

The sub-framework dspGUI can create the necessary classes through an instance of the factory class aqses.gui.AqsesGUIFactory. When the user selects a processing method, the framework will locate the correct factory class, call the method to initialize it and get then the components it needs for integrating the method in the GUI.

5. Implementation

5.1. *Hardware and operating systems*

For developing and testing the framework normal multimedia PCs are used. The operating systems used are Windows 98, Windows 2000, Windows XP and the Linux distributions Redhat 9.0, Fedora Core 2, Mandrake 9.2, Mandrake 10 and Gentoo. The following configuration of one of the PCs is given as an example:

- System model: Dell OptiPlex GX150;
- Operating system: Windows 2000;
- CPU: Pentium III 1 GHz;
- GPU: NVIDIA RIVA TNT2 Pro;
- Hard disk: 20 GB;
- Memory: 512 MB.

5.2. *Software*

All the software tools are freely available on the web. The following tools are used:

- The Java 2 Platform, Standard Edition (J2SE). This is provided by Sun Microsystems and provides the Java environment to run and develop Java applications. To use the framework one needs to use at least version 1.4. At the moment version 1.4.2 [18] is used.
- Eclipse SDK, this is an Open-Source Java IDE. At the moment version 2.1.3 [21] is used.
- Eclipse UML plug-in Omondo. This plug-in makes it possible to work simultaneously with UML diagrams and Java code. The current version used is 1.2.1 [22].
- g77 and gcc. These are the GNU Fortran compiler and the GNU C compiler. These compilers are used to create shared libraries. There are versions for Windows and Linux. On Linux version 3.3.3 and on Windows version 3.3.1 [23] is used.

6. Conclusions

A free open-source framework is presented in this paper and it allows developers and researchers to focus more on the design and the implementation of their methods and package it in a software tool. It also allows to reuse a big amount of the code for implementing new applications. In the future the number of available plug-ins will grow and the framework will get more complete and powerful. By using a plug-in system there is a small over-head when an application is started. All the different plug-ins have to be located, initialized and loaded. But it allows developers to change the framework dynamically and makes it more flexible. In the framework templates can be used to store settings and to automate the processing of the data. By using templates one can make a minimalistic GUI with almost no settings without loosing any functionality or one can make a GUI bloated with settings. In this way the GUI can be adapted to the user of the framework. The framework will be available under an OSI approved open-source license [24]. This will allow others to create their own methods and plug-ins. The framework itself is completely written in Java and is platform independent. As shown in the application the user can still choose to embed code written in another programming language. It will make the application platform dependent. However, by checking the platform and loading the correct shared library one can maintain platform independency between Windows and Linux for the application.

Acknowledgements

Dr. Sabine Van Huffel is a full professor at the Katholieke Universiteit Leuven, Belgium. Research supported by Research Council KUL: GOA-AMBioRICS, several PhD/postdoc & fellow grants; Philippe Lemmerling was a postdoctoral

fellow of the Fund for Scientific Research, Flanders (Belgium) (F.W.O.-Vlaanderen). Flemish Government: FWO: PhD/postdoc grants, projects, G.0269.02 (magnetic resonance spectroscopic imaging), research communities (ICCoS, ANMMM). PhD grants provided by IWT. Belgian Federal Government: DWTC (IUAP IV-02 (1996–2001) and IUAP V-22 (2002–2006)—dynamical systems and control: computation, identification & modelling); EU: BIOPATTERN (contract no. FP6-2002-IST 508803), eTUMOUR (contract no. FP6-2002-LIFESCIHEALTH 503094), and HealthAgents (contract no. FP6-2005-IST 027213).

REFERENCES

[1] J.A. Stanley, D.J. Drost, P.C. Williamson, R.T. Thompson, The use of a priori knowledge to quantify short echo in vivo ^{1}H MR spectra, Magn. Reson. Med. 34 (1) (1995) 17–24.
[2] S.W. Provencher, A constrained regularization method for inverting data represented by linear algebraic or integral equations, Comput. Phys. Commun. 27 (1982) 213–227.
[3] LCModel Software Package. http://www.s-provencher.com/pages/lcm-manual.shtml.
[4] A. Naressi, C. Couturier, J.M. Devos, M. Janssen, C. Mangeat, R. de Beer, D. Graveron-Demilly, Java-based graphical user interface for the MRUI quantitation package, MAGMA 12 (2001) 141–152.
[5] MRUI software package. http://carbon.uab.es/mrui.
[6] L. Vanhamme, A. van den Boogaart, S. Van Huffel, Improved method for accurate and efficient quantification of MRS data with use of prior-knowledge, J. Magn. Reson. 129 (1999) 35–43.
[7] H. Ratiney, M. Sdika, Y. Coenradie, S. Cavassila, D. van Ormondt, D. Graveron-Demilly, Time-domain semi-parametric estimation based on a metabolite basis set, NMR Biomed. 18 (1) (2005) 1–13.
[8] M.E. Fayad, D.C. Schmidt, R.E. Johnson, Building Application Frameworks, Wiley, 1999.
[9] M.E. Fayad, D.C. Schmidt, R.E. Johnson, Implementing Application Frameworks, Wiley, 1999.
[10] R.E. Johnson, B. Foote, Designing reusable classes, J. Object-Oriented Program. 1 (2) (1988) 22–35.
[11] Eclipse Plug-in Architecture. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.
[12] D. Sima, S. Van Huffel, Regularized semiparametric model identification with application to NMR signal quantification with unknown macromolecular baseline, J. R. Stat. Soc. Ser. B 68 (3) (2006) 383–409.
[13] J.B. Poullet, D.M. Sima, A.W. Simonetti, B. De Neuter, L. Vanhamme, P. Lemmerling, S. Van Huffel, An automated quantitation of short echo time MRS spectra in an open source software environment: AQSES, Internal Report 05-168, ESAT-SISTA, K.U. Leuven, Leuven, Belgium, 2005. NMR Biomed., in press.
[14] T. Sundin, L. Vanhamme, P. Van Hecke, I. Dologlou, S. Van Huffel, Accurate quantification of ^{1}H spectra: from finite impulse response filter design for solvent suppression to parameter estimation, J. Magn. Reson. 139 (2) (1999) 189–204.
[15] T. Laudadio, N. Mastronardi, L. Vanhamme, P. Van Hecke, S. Van Huffel, Improved lanczos algorithms for blackbox MRS data quantitation, J. Magn. Reson. 157 (2002) 292–297.
[16] C. Larman, Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design, Prentice-Hall, PTR, 1998.
[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, An Introduction to Object-Oriented Analysis and Design, Addison-Wesley, 1995.
[18] The Java 2 Platform, Standard Edition. http://java.sun.com/j2se.
[19] K. Walrath, M. Campione, The JFC Swing Tutorial, A Guide to Constructing GUIs, Addison-Wesley, 1998.
[20] J. Shirazi, Java Performance Tuning, 2nd ed., O'Reilly, 2003.
[21] Eclipse SDK. http://www.eclipse.org.
[22] Eclipse UML Plug-in Omondo. http://www.omondo.com.
[23] The GNU Fortran Compiler and GNU C Compiler. http://gcc.gnu.org.
[24] The Open Source Initiative (OSI). http://www.opensource.org/licenses/.