Lorenzo Cioni

The data-flow model as a design tool for the development of DSP software

(revised version of a paper accepted at the "V Simposio Internacional de Comunicación Social", Santiago de Cuba, January 22-24 1997)

Abstract

The present paper describes an approach to the design of application-oriented software for Digital Signal Processing (DSP) that has been inspired by the data flow model. We propose a style of applications design guided by the flow of data among the modules that represent the functional elements of the application under development. Such a style makes use of flow graphs, that are typical of the data flow model, as tools for both the design and the analysis.

The proposed approach lends itself to either bottom-up, top-down or middle-out design methodologies according to the granularity of the modules from which the designer begins the specification of the application under development. The resulting model, tough of abstract type, can be easily implemented even on distributed or multiprocessor architecture with hardware specialised for I/O. The paper presents an informal description of the model and some simple examples of its use for the design of applications for DSP.

Introduction

The data flow model is usually considered an alternative model to the more traditional control flow model. According to the control flow model the ordering of the operations fully defines the evolution of a computation whereas in a data flow model such an evolution is governed by the so called data dependencies where with the term "data dependency" we mean the fact that only the availability of the needed data determine the execution of one of the possible operations. Our approach to data flow model exploits this potentially high parallelism and sees it as a design tool for the development of application-oriented software.

This fact allows us to take into consideration only the more abstract aspects of the model and disregard any computationally hard features such as non determinism, evaluation criteria and recursion. The main aspects we are interested in are modularity, high parallelism and the use of directed, mainly acyclic graphs for the description of data flow programs.

Modularity and parallelism make the development of applications easier ([De73a] and [De73b]) whereas the use of graphs in which nodes are linked by arcs that represent the flow of data among modules (data dependencies) allows us to have the design or the analysis always under control.

The adoption of modularity press the designer to define simple and self-contained functions that are easy to implement, check and maintain whereas parallelism encourages the use of distributed systems and the development of classes of modules that can be used repeatedly.

We note, moreover, that in order to develop software ([AnGre87]) three approaches are available. They are top-down, bottom-up and middle-out, depending on the granularity of the modules from which the design starts.

In the first case an application is subdivided in simpler and simpler modules till the implementation level, in the second case the design starts from the statements and library routines of a programming language so to define more and more complex modules whereas in the last case the design starts from non primitive modules that are both composed to form a complex application and subdivided so to be implemented in a given programming environment.

Anyway the design benefits from the facts that the modules are characterised by minimal interactions and functional independence. Modules represent, in the aforesaid design approaches, the elementary units of a program representation and can be grouped in classes or composed in more complex modules within which they maintain their integrity and their independence from the context.

The applications' domain: digital signal processing

The basic elements of any system devoted to DSP are shown in figure 1 (from [Cio96]) and can be grouped as follows ([Cio94] and [NaIns94]):

- data I/O,
- data storage,
- data analysis and
- data presentation.

Data I/O includes data recording and playback together with some filtering functions over speech data, usually embedded on the I/O boards but that can be performed even within DSP programs. Moreover there must be some conversion functions since many currently available DSP programs work with non-standard file formats.

Data presentation includes techniques for the visualisation of data both from acquisition, analysis and storage. Moreover there usually there are included also classical editing operations such as *copy*, *cut*, *paste* and *delete* and some functions to perform rendering operations together with the availability of a signal generator that allows the generation of standard parametric functions.

Data analysis involves frequency and time domain techniques that include pitch extraction, calculation of spectrogram and energy, evaluation of zero crossing, definition of voiced and unvoiced portions of speech, evaluation of the formants and so on.

Data storage allows the storage of data in permanent structures such as files or data bases: the availability of a data base structure allows the easy maintenance of related data in complex but flexible structures.

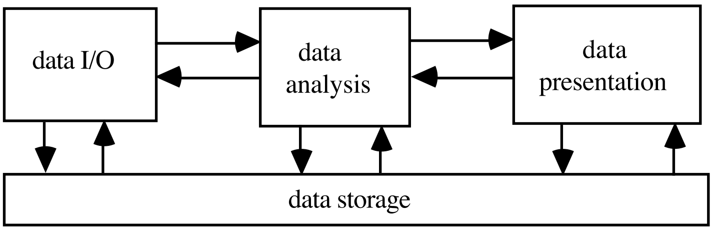

Figure 1

This short analysis shows how the domain is composed by a lot of functions that can be grouped together and that may be defined in terms of modules so that the availability of techniques for the composition of modules should be of great help in the development of large applications.

We therefore have a domain characterised by "context dependant" operations that can be grouped to form modules that, in their turn, are naturally composed in macro-modules or applications. Modules perform, for instance, operations such as recording, playback, filtering, spectral analysis, LPC analysis and many others and can be grouped to form more complex applications that perform I/O, editing, visualisation, rendering, labelling, analysis an do on. Going on with the abstraction process we can define an application for pitch analysis that is composed by the following modules: I/O, analysis and visualisation with an optional editing module.

This process defines a bottom-up approach but it is easy to understand how define the corresponding top-down and middle-out approaches.

In the first case we establish the need of an application with a certain set of operations and then recursively partition it till the lowest level modules.

In the latter case me define coarse grain modules and both plan their interactions within a complex application (i. e. recursively a module) and implement them with already available modules or suitably designed modules.

From an abstract syntax point of view we may sketch the following definition

$$\mathbf{A{:}{=}{=}\ m \mid A \rightarrow m \mid m \rightarrow A \mid m\ /\!/\ A \mid A\ /\!/\ m}$$

i.e. an application (A) is a module (m) or is an application that passes data to a module or receives data from it or evolves in parallel with a module.

We note that an application defined in this way is in itself a module and so can be accessed only through its interface and that the modules belong to a set that depends on the abstraction level of the design.

Moreover we have rules like the followings that define the data dependencies among the modules, where $\equiv$ stands for an equivalence relation:

$$\mathbf{m \rightarrow m_1\ /\!/\ m_2 \quad \equiv \quad m \rightarrow m_1\ /\!/\ m \rightarrow m_2}$$

$$\mathbf{m_1\ /\!/\ m_2 \rightarrow m \quad \equiv \quad m_1 \rightarrow m\ /\!/\ m_2 \rightarrow m}$$

In such rules the real evolution depends on the evaluation criteria for each module. If we have modules with a lazy evaluation criterion they can produce their output values just when only the needed values are present.

So if we have a "lazy module" that calculates **f(x, y) = x * y** and we know that **x = 0** it can perform the evaluation without waiting the availability of the value of **y**. We note that **x** and **y** are not names of variables but names of tokens whereas if the same operation is performed by a "strict module" the calculation is performed only when the tokens with the values of both **x** and **y** are available.

An application is therefore composed by a set of co-operating modules connected to form a recurrent, directed graph: some of the modules are hidden since they do not interact with the user whereas some others have a (graphic) interface towards the user and are said to be interactive. An interactive module is, of course, a lazy and asynchoronous module since it operates depending on user's timings and needs.

From the foregoing considerations it is easy to understand that the development of an application coincides with the definition of a graph of modules and that therefore in order to make the development of applications easier a software developer should have a dedicated environment based on a graphic user interface. Such an environment should include at least the following meta-applications:

- a Graph Manager that allows the definition of the graph of modules depending on the data dependencies
- an Module Editor that allows an easy customisation of each module and
- a Class Handler that allows the storage of modules within classes so that a user can define classes of modules according to an object-oriented approach and then create an instance of a class and customise its parameters according to particular needs.

The flow of data as a design tool

The use of modules for software design and implementation is therefore a common approach and is well founded on famous and stable paradigms such as object oriented programming.

The main drawback of this undoubtedly fruitful approach is that whereas usually the design and implementation of simple modules can be carried out easily and effectively (since the set of operations that we want to include in each module is well known as well as the representation domains on which it will act) it may happen that both their composition in and their definition within a complex application are a little bit more complex affair: often it is not clear which is the best design approach and, at the same time, if we can get the same result by composing already available modules. This fact in many cases confines the use of modules to the top-down approach.

We propose therefore a design philosophy that is driven by the flow of data among the modules that compose the functional elements of the application under development. From this point of view we use flow graphs as tools for the analysis and design of relations (one-to-one, one-to-many and many-to-one) producer/consumer that allow, through the definition of a data exchange among modules, the definition of complex applications.

The analysis of the flow of data among modules forces the designer to think in terms of modules and module composition during both analysis and design.

In this way we aim at the definition of a descriptive tool for the definition of simple applications that can be easily co-operate for the execution of complex tasks. Such a tool should encourage software reuse and both the integration of computing systems and the use of distributed and heterogeneous environments.

As shown in figures 1 and 2, the designer defines macro modules according to the first step of a top-down approach by defining the flow of data among these or, as a last step of a bottom-up process, defines an application resulting from the composition of modules.

The basic data flow model with some extensions

The data flow model is characterised by a so called basic model and some derived models that share the basic properties. We need to make some extensions to such a framework so to account for some characteristics of our domain of application.

According to the data flow model ([BaToVa87] and [Tre87]) a computation can be described in terms of a network or a graph in which the nodes are the functional modules whereas the arcs represent both the channels through which the modules exchange data and the dependencies among the modules.

The streams of data are represented with tokens running along the arcs and the evolution of a computation is determined by flowing of the tokens on the arcs.

The modules (the nodes of either the network or the graph) are characterised by a functional input/output relation and there can be nodes with only input or output arcs. In such a model the computation is controlled by the availability of the data on the input arcs of each module (data driven) and we can have two evaluation criteria: strict or lazy evaluation. In the first case a module cannot produce its output tokens unless there is a token on each input arc whereas in the second case only a subset of the input arcs need to have a token in order for a module to produce its values on the output arcs.

According to the model the computation is in itself completely asynchronous and is characterised by an high parallelism among the modules: the first characteristic requires the presence of queues of tokens on the arcs whereas the parallelism is limited by logical dependencies among the modules.

Since, however, we are going to use the model as a descriptive tool the characteristics that are useful for our purposes are its "asynchronicity" and its high parallelism together with the heavy use of modules and of network/graph for the definition of programs.

The model we have seen is characterised by active elements linked by arcs and the computation is driven by the flow of data. For our purposes we need some extensions that infringe the purity of the model but allow us to better describe our applications.

Such extensions include the possibility to have configuration files (such those labelled as # in figure 2) switches (see D in figure 2) and the possibility to define modules that are characterised by input/output arcs and by asynchronous interactions with the user (for instance the module labelled as presentation in figure 2): we require the possibility to define modules whose evolution is governed not only by the availability of data on the input arcs but also by asynchronous user's requests.

A simple example

Figure 2 (from [Cio96] with modifications) shows a simple application of the proposed approach to the design of an application for pitch extraction. This simple example should show how the analysis of the flow of data among the modules helps either in the analysis and or in design. We note how the arcs define flows of data and are not related in any way with a control switching among the modules.

If we consider the figure from the top, starting from the module labelled (A), we assume a top-down approach (it is obvious that in figure 2 the approach does not reach the primitive statements of a programming environment!!) whereas if we start from the bottom and go upwards and downwards we assume either a middle-out approach (if the modules below the labels Analysis and Presentation are not available) or a bottom-up approach if we already implemented them.

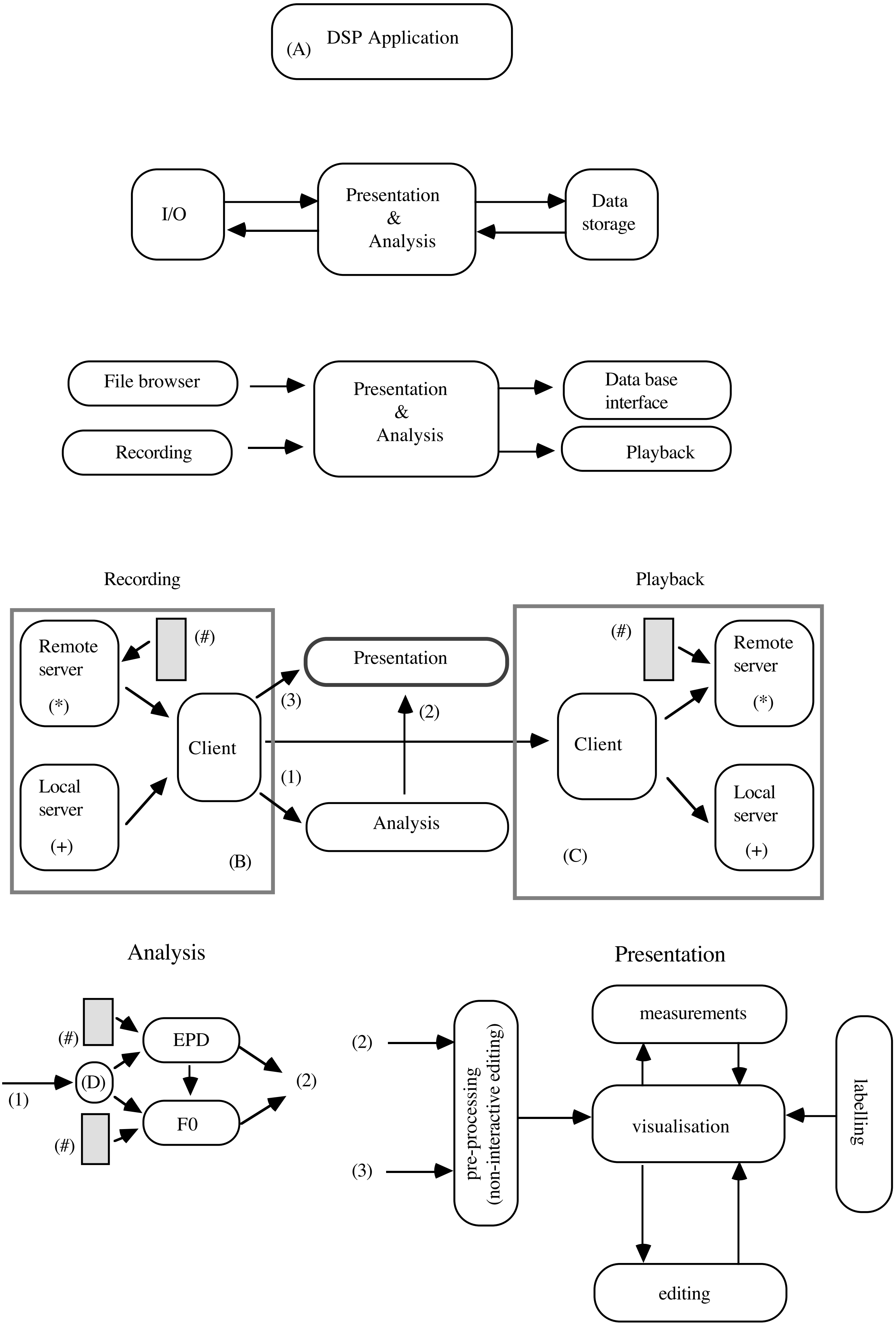
(A)
DSP Application
I/O
Presentation & Analysis
Data storage
File browser
Recording
Presentation & Analysis
Data base interface
Playback
Recording
Playback
Remote server
(#)
(*)
Local server
(+)
Client
(B)
Presentation
(3)
(2)
(1)
Analysis
(#)
Remote server
(*)
Client
Local server
(+)
(C)
Analysis
(#)
EPD
(D)
(1)
F0
(#)
(2)
Presentation
measurements
(2)
pre-processing (non-interactive editing)
(3)
visualisation
labelling
editing

Figure 2

We note that modules labelled as (B) and (C) are not primitive modules whereas modules labelled as F_0 and EPD can be seen as primitive in a given domain.

According to the top-down approach, if we are designing a complex application for pitch analysis, we can start by decomposing it in some main modules such those shown in the upper part of figure 2: I/O, Presentation&Analysis and Data storage.

We note that the we have not an univocal decomposition but we think the one we are suggesting is a good one since it allows us to encapsulate in distinct modules the interactions with I/O devices, with the Operating System and the computational part of the application. From now on the further decomposition are almost obvious.

Two interesting and really portable modules are the file browser for the search of files and the data base interface that, in its simpler form, interacts with the native fail system for the storage of new files.

We note that the I/O module is split in two modules (Recording and Playback) and each module is, in its turn, decomposed as shown in the boxes labelled (B) and (C).

Such boxes are characterised by a client module and both a remote and a local server: the client module provides a user interface so to allow recording or playback of speech whereas the server modules interact with a physical device that can be plugged in the local host or can be driven by a remote host.

For instance, we can have two local servers for playback and recording with low quality (8kHz 8bit/sample) and two remote servers with higher quality and programmable parameters: sampling frequency (8/10/12/16 Hz) and bit/sample ratio (8 or 16).

The module labelled Presentation&Analysis is, in its turn, decomposed in two modules (Analysis and Presentation) so that once more we tend to isolate the computationally hard aspects of the application from the more soft ones.

Since in the present example the analysis is oriented to pitch extraction we can decompose the analysis module as shown so to define two simpler modules that are characteristic of Sift algorithm: EPD and F0: the first allows to extract information that can be used by the second that performs pitch extraction from the speech file.

Both require the presence of configuration files (labelled as # in figure 2) and their interaction is governed by a programmable switch (D): configuration files contain the values of parameters that are critical for both modules and that the user can set by using a further module not shown in figure 2.

On the other hand, the Presentation module is characterised by more classical and general elements such as pre-processing, measurements, visualisation, labelling and editing.

Such modules allows the user to perform operations such as filtering and rendering or cut/copy/paste operations (editing and pre-processing) and to display the content of one or more files (and each file can contain either speech or a spectrogram or a set of values of F0) by interacting with a display manager and a window manager.

They also allow the user to obtain numerical values (measurements) or to fix string values (labelling).

It is obvious how such a decomposition can be carried on till the definition of modules that can be either subroutines or library functions, where each subroutine is, in its turn, written in a given programming language by sing a particular programming environment.

Conclusive remarks

The proposed approach represents an abstract framework that uses classical concepts but in a somewhat new perspective and that provides a graphical way for supporting application-oriented software development.

Following this path we have a tool that supports software development that can be used with bottom-up, middle-out and top-down approaches so to encourage software reuse and exploit at the most both modularity and parallelism.

For further details see [Cio95], here we note that, as stated in [Cio96], the design of application-oriented software should induce the definition of applications that are useful by themselves and that can be easily composed thanks to the analysis of the flow of data among the modules.

The proposed approach is well suited for an implementation on distributed multiprocessor and/or multitasking architecture is there is a correspondence among modules and processes or group of processes and the designer makes use of native tools.

Bibliography

[AnGre87] D. Andrews & M. Greenhalg, *Computing for Non-Scientific Applications*, Leicester University Press, Leicester, 1987.

[BaToVa87] F. Baiardi, A. Tomasi & M. Vanneschi, *Architettura dei Sistemi di Elaborazione*, Vol. 1, Franco Angeli, Milano, 1987.

[Cio94] L. Cioni, "A data-base for speech signal processing", proceedings of SST-94, 6-8 December 1994, Perth, Western Australia, Australia.

[Cio95] L. Cioni, "Co-operative principles in applications design", the 4th International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, Pisa, 3-8 April 1995.

[Cio96] L. Cioni, "Il modello data-flow come design tool per lo sviluppo di applicazioni per il Digital Signal Processing", XXIV Convegno Nazionale dell'Associazione Italiana di Acustica, Trento, 12-14 June 1996.

[De73a] J. B. Dennis, "The design and construction of software systems", *Software Engineering. An advanced course*, eds. G. Goos and J. Hartmanis, Lecture Notes in Compter Science, Springer Verlag, Munich, 1973.

[De73b] J. B. Dennis, "Modularity", *Software Engineering. An advanced course,* eds. G. Goos and J. Hartmanis, Lecture Notes in Compter Science, Springer Verlag, Munich, 1973.

[NaIns94] National Instruments, *IEEE 488 and VXIbus Control, data Acquisition and Analysis*, Products Cathalog, 1994.

[Tre87] P.C. Treleaven, "The data-flow approach for MIMD multiprocessor systems", *Parallel Processing Systems. An advanced course*, ed. D. J. Evans, University Press, Cambridge, 1987.