FP7-ICT-2013-C TWO!EARS Project 618075

Deliverable 2.1

WP2 Software Architecture

WP2 *

May 29, 2014

* The TWO!EARS project (http://www.twoears.eu) has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 618075.

Project acronym:	TWO!EARS
Project full title:	Reading the world with TWO!EARS

Work packages:	WP2
Document number:	D2.1
Document title:	WP2 software architecture
Version:	1

Delivery date:	29 May 2014
Actual publication date:	29 May 2014
Dissemination level:	Restricted
Nature:	Report

Editor:	Guy Brown
Author(s):	Guy Brown, Remi Decorsière, Dorothea Kolossa, Ning Ma, Tobias May, Christopher Schymura, Ivo Trowitzsch
Reviewer(s):	Jonas Braasch, Dorothea Kolossa, Bruno Gas, Klaus Obermayer

Contents

1 Executive summary **1**

2 Overview of the Two!Ears software architecture **3**
2.1 Background . . . 3
2.2 Software architecture . . . 4
2.3 Overview of the report . . . 6

3 Bottom-up auditory signal processing **7**
3.1 Software design . . . 7
3.1.1 Processors . . . 7
3.1.2 Manager . . . 9
3.1.3 Data organization . . . 10
3.1.4 General overview . . . 12
3.2 Handling user requests . . . 13
3.2.1 Dependencies . . . 13
3.2.2 Feedback . . . 14
3.3 Available processors . . . 16
3.3.1 Signals . . . 16
3.3.2 Cues . . . 18
3.4 Planned extensions to the software . . . 19

4 Reference **21**
4.1 WP2 reference . . . 21

Acronyms **23**

Bibliography **25**

1 Executive summary

The goal of the Two!Ears project is to develop an intelligent, active computational model of auditory perception and experience in a multi-modal context. At the heart of the project is a software architecture that optimally fuses prior knowledge with the currently available sensor input, in order to find the best explanation of all available information. Top-down feedback plays a crucial role in this process. The software architecture will be implemented on a mobile robot endowed with a binaural head and stereo cameras, allowing for active exploration and understanding of audiovisual scenes.

This deliverable sets out the design of the software architecture, with an emphasis on communication between the components of the system. An object-oriented approach is used throughout, giving benefits of reusability, encapsulation and extensibility.

The first stage of the system architecture concerns bottom-up auditory signal processing, which transforms the signals arriving at the binaural head into auditory cues. Bottom-up signal processing is implemented as a collection of processor modules, which are instantiated and routed by a manager object. This affords great flexibility, and allows real-time modification of bottom-up processing in response to feedback from higher levels of the system. Processor modules are provided to compute cues such as rate maps, interaural time and level differences, interaural coherence, onsets and offsets.

This deliverable describes the aspects of the software architecture developed in work package two. A more complete account of the software architecture is given in deliverable D3.1, including a description of the blackboard system and a proof of concept study.

2 Overview of the Two!Ears software architecture

This report documents the design of the Two!Ears software architecture and describes the motivation for the approach taken. Our approach is to first describe the software architecture in general terms. A specific example of applying the architecture to a computational hearing problem is then given; specifically, the problem of localising and identifying a single sound source under conditions in which front/back confusions must be resolved.

2.1 Background

The goal of the Two!Ears project is to develop an intelligent, active computational model of auditory perception and experience in a multi-modal context. In order to do so, the system must be able to recognise acoustic sources and optical objects, and achieve the perceptual organisation of sound in the same manner that human listeners do. Bregman (1990) has referred to the latter phenomenon as auditory scene analysis (ASA), and to reproduce this ability in a machine system a number of factors must be considered:

- ASA involves diverse sources of knowledge, including both primitive (innate) grouping heuristics and schema-driven (learned) grouping principles;
- Solving the ASA problem requires the close interaction of top-down and bottom-up processes through feedback loops;
- Auditory processing is flexible, adaptive, opportunistic and context-dependent.

The characteristics of ASA are well-matched to those of *blackboard* problem-solving architectures. A blackboard system consists of a group of independent experts ('knowledge sources') that communicate by reading and writing data on a globally-accessible data structure ('blackboard'). The blackboard is typically divided into layers, corresponding to data, hypotheses and partial solutions at different levels of abstraction. Given the contents of the blackboard, each knowledge source indicates the actions that it would like to perform; these actions are then coordinated by a scheduler, which determines the order in which

actions will be carried out.

Blackboard systems were introduced by Erman *et al.* (1980) as an architecture for speech understanding, in their Hearsay-II system. In the 1990s, a number of authors described blackboard-based systems for machine hearing (Cooke *et al.*, 1993, Lesser *et al.*, 1995, Ellis, 1996, Godsmark and Brown, 1999). All of these systems were in most respects 'conventional' blackboard architectures, in which the knowledge sources consisted of rule-based heuristics. In contrast, the TWO!EARS architecture aims to exploit recent developments in machine learning, by combining the flexibility of a blackboard architecture with powerful learning algorithms afforded by probabilistic graphical models.

2.2 Software architecture

The diagram of the general software architecture is shown in Figure 2.1. The acoustic input, which consists of the left and the right-ear time domain signals captured by the robotic platform, is processed by a peripheral processing module that simulates the effective signal processing in the auditory system. The significance of this task lies in the extraction of meaningful signals and cues that capture important aspects of the acoustic scene, which will enable the higher processing stages of the architecture to interpret the acoustic scene. Therefore, the two time domain signals are processed independently by a monaural pathway, which consists of a middle ear and a cochlear module. In addition, a binaural processor compares the two monaural signal streams in order to evaluate interaural differences between the left and the right ear signal representations. Based on these monaural and binaural signal representations, a number of monaural and binaural cues are extracted. These cues describe and summarize relevant characteristics of the monaural and binaural signal representations over short time frames. In contrast to purely signal-driven (bottom-up), and therefore *static* approaches, the TWO!EARS software architecture explicitly incorporates task-dependent feedback.

In addition, video signals are captured by cameras on the robotic platform. The output from the first stage of processing is then a multi-dimensional, audiovisual representation of the environment which provides the input to subsequent stages of the architecture.

Later stages of the TWO!EARS architecture are broadly based on the HEARSAY-II system (Erman *et al.*, 1980). A number of **knowledge sources** (KS) collaborate via the blackboard, by triggering when relevant data is available and depositing new data. The architecture is event-driven; a change in the state of the blackboard (such as the arrival of new data, or the emergence of a new hypothesis) causes an event to be broadcast. A **blackboard monitor** is responsible for monitoring and handling these events; it maintains an **event register** that indicates which KS can respond to a certain event. The possible actions that

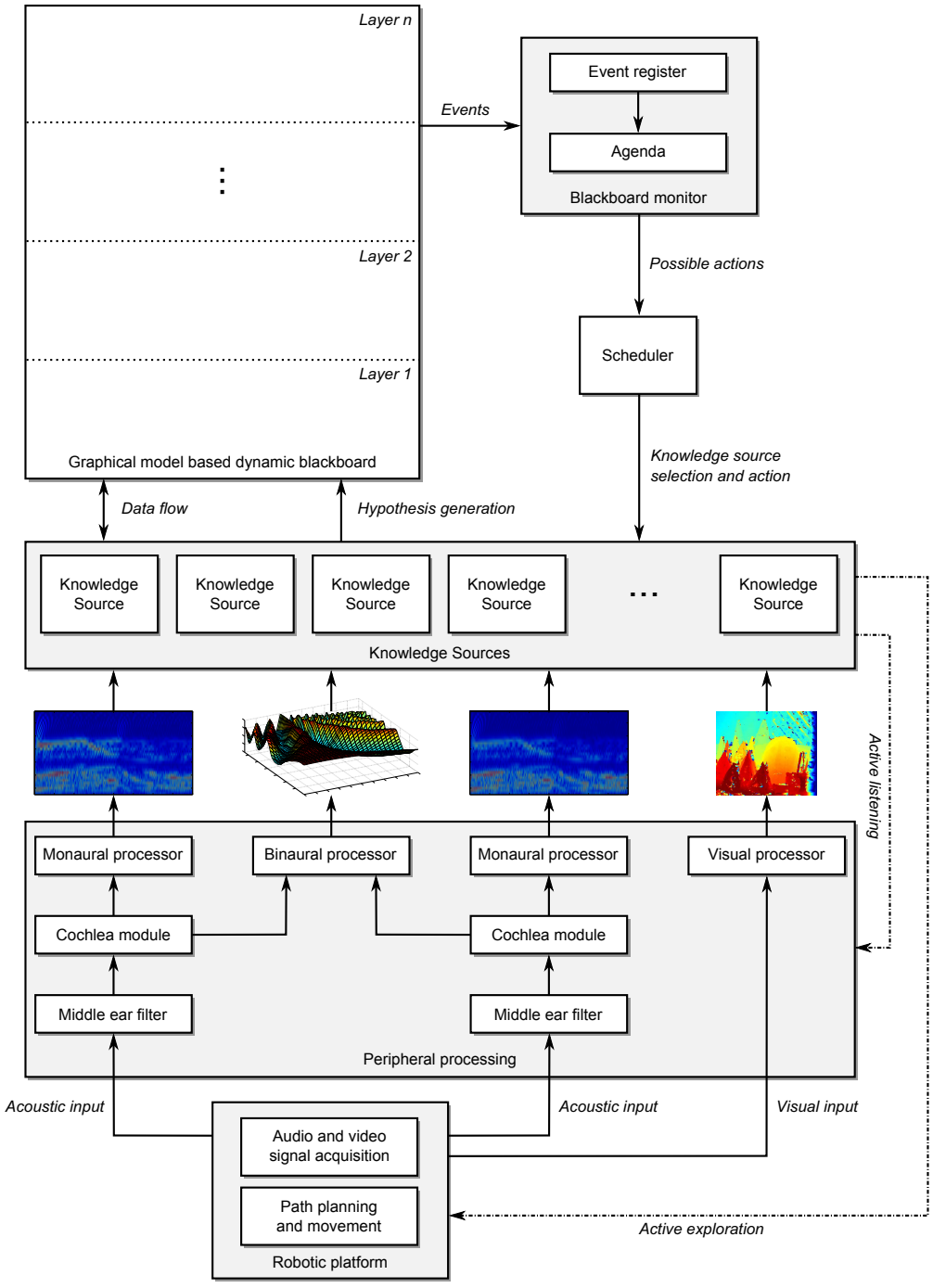

Figure 2.1: System diagram of the general software architecture

can be performed, given the current state of the blackboard, are listed in an **agenda**. A **scheduler** is then responsible for ranking the possible actions and selecting one to perform. When the action is performed, this will most likely result in a further change in the state of the blackboard leading to the broadcast of further events.

Graphical models form a key part of the TWO!EARS architecture, either as structures on the blackboard or as the basis for knowledge sources. The architecture therefore has the flexibility to combine rule-based and statistics-based information processing. The blackboard is divided into layers of abstraction, such that an hypothesis at level n is supported by evidence at level $n-1$. At the highest level of the blackboard, the layers constitute a 'world model' which describes the acoustic sources in the environment in terms of their relationships, properties and meaning.

The TWO!EARS architecture will be implemented on a robotic platform in due course, allowing for **active exploration** of the environment. For example, hypotheses on the blackboard about the location of a sound source of interest may trigger a path planning action, which results in the robot moving closer to the source's predicted location. Similarly, planning actions may dictate that it is necessary for the robot to rotate its head in order to gather more information.

Similarly, the TWO!EARS architecture allows for **active listening**. Properties of the bottom-up processing, such as the tuning characteristics of cochlear filters, can be modified by top-down feedback from higher stages of the blackboard. Such feedback may occur at multiple levels, including the interaction of binaural hearing and mobility at the sensorimotor level. Reflexive movements of the robot, which occur without hypothesis-driven feedback from the blackboard, may also occur.

MATLAB has been chosen as the implementation language for the software architecture, because it is widely available within TWO!EARS partner laboratories, it supports object-oriented programming, and can be run directly on the robot platform.

2.3 Overview of the report

In the remainder of this report, Section 3 describes the bottom-up auditory signal processing techniques developed within work package two (WP2). Finally, Section 4 provides reference material that will be helpful in using our MATLAB implementation.

3 Bottom-up auditory signal processing

The task of WP2 is to transform the listener's ear signals, that are supplied by work package one (WP1), into a multi-dimensional signal and cue representation. In the following the general software design is described in detail.

3.1 Software design

The processing stages of the WP2 software package, as well as the types of outputs it provides, are essentially dependent on requests made by the software user. They are subject to change not only between calls to the package (e.g., switching from scenario A to scenario B), but also, and more importantly, during execution of the software (e.g., when feedback from higher stages is received). Hence there is a strong incentive for the software to be modular and able to adapt to potentially very different scenarios. This naturally suggests an object-oriented approach in the implementation.

3.1.1 Processors

An object-oriented approach allows each processing stage (e.g., the computation of one cue from a given signal) to be assigned to an independent "processor" object. The following two fundamental properties of object-oriented programming can then benefit the modularity of the implementation. *Encapsulation* allows these processors to be self-managed, and most importantly independent of each other and of other existing objects. *Inheritance* of individual processors from a parent processor class allows new processing stages to be added and implemented with only a little new code writing (which is less likely to introduce errors).

Parent and children processor classes

In practice, an abstract processor class is implemented, which will be the parent class of all processors. It should therefore encapsulate all properties and methods that are common to all processors. Figure 3.1 presents a class diagram for the `Processor` parent

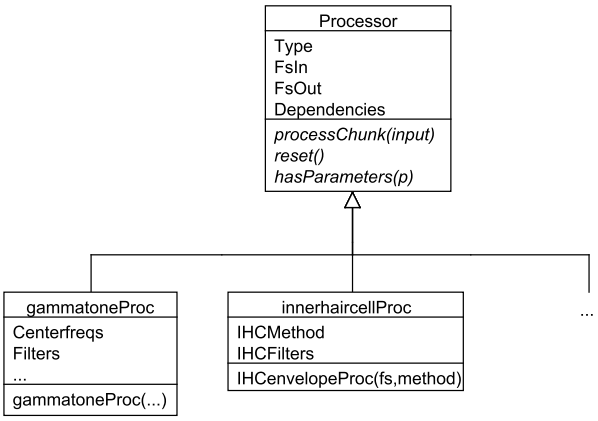

Figure 3.1: Class diagram of the processor parent and children classes.

class as well as two example child classes. Properties that are common to all processors include:

- a label (`Type`) to identify the action of the processor
- the sampling frequencies of the input (`FsIn`) and output (`FsOut`) that the processor manipulates.

The additional property `Dependencies` is not required for the functioning of the individual processor, but its use simplifies the management of several processors and will be described in Section 3.2. All child processors should then implement the following abstract methods:

- `processChunk` which, given an input, computes and returns the corresponding output
- `reset` which resets the processor to a clean state, e.g., for processing a new signal

Again, as for the `Dependencies` property, the method `hasParameters` is not necessary but will simplify some processes described later.

The bottom-up signal processing of WP2 involves many processing stages. Each stage can then be implemented as a child of the `Processor` class. Figure 3.1 shows two example children that inherit the `Processor` class. Inheritance is indicated in the class diagram by a closed-head arrow. Each child can have additional properties that are relevant to the processing it performs. For example the `gammatoneProc` which is responsible for performing filtering by a Gammatone filterbank, in addition to the `Processor` properties, has to keep track of the center frequencies of its channels (in `Centerfreqs`). Its processing also involves a number of filters (see subsection below), which are stored as a property (`Filters`).

Other child processors will involve different properties of their own (e.g., the inner hair-cell envelope extractor `innerhaircellProc` has an additional name tag `IHCMethod` for the method employed). A detailed list of currently available (child) processors is given later in this document (section 3.3).

Additional methods for child processors are essentially specific constructors. Different processors need different information to be created, hence each child has its own constructor which takes specific inputs.

Filter objects and real-time compatibility

Among the processors that are implemented, many involve some filtering operations. For example the `gammatoneProc` and `innerhaircellProc` pictured in Fig.3.1 both involve filtering and have "filters" stored as properties. These filters are also implemented as objects, i.e., in a similar fashion as the processors, with a parent filter class and specialized child filter classes. This approach can benefit significantly from *encapsulation* by storing a filter's internal state as one of its properties. By restricting the access of this property to the filter object only (i.e., have it being a *private* property), the filter can self-manage its internal state without any risk of being "contaminated" by any outside event. This means that successive calls for filtering will take into account the filter's states relative to the previous call. Note that this makes the approach fully compatible with real-time processing (i.e., the ability to process an incoming signal in a sequence of short blocks), without additional effort. The filter object also includes a `reset()` method that will clear its internal states, e.g., to initialize the processing for a new signal. When filter objects are instantiated in a processor object, the `reset()` method of the processor essentially calls the `reset()` method of all the filter instances it contains.

3.1.2 Manager

A given configuration of the WP2 software will involve several processing stages, hence multiple processors. The processors have to be instantiated, and their inputs/outputs routed according to which processor needs or generates which signal (or cue). This is not done manually by the user but is instead handled by a dedicated object, the "manager".

The manager class (see Fig.3.2) contains instances of the processors needed for the overall processing as the property `Processors` (e.g., as an array of individual `Processor` objects). To know where to fetch the inputs for each processors, `InputAddress` contains a list of pointers to the inputs of each processor. Similarly, `OutputAddress` indicates where to place the output of a given processor. Because some processors take as input the output of other

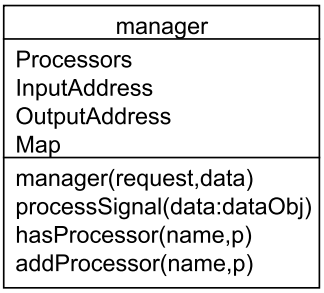

Figure 3.2: Manager class diagram.

processors, the processing has to be ordered (we will return to this dependency issue in section 3.2). The order in which processors are called is stored in `Map`.

Processing and routing of inputs/outputs is then carried on through the method `processSignal`. This method loops over the total number of processors, calling the `processChunk` of each of them, but one at a time. Assuming the signals are contained in `data`, for a given index `i`, this breaks down to one line of code (here split on 4 lines for readability):

```
j = Map(i)
in = Manager.InputAddress(j)
out = Manager.OutputAddress(j)
data(out) = Manager.Processors(j).ProcessChunk(data(in))
```

The last line shows how processing and routing of inputs/outputs are performed all at once. This operation is repeated for all the processors (i.e., all the indexes `i`).

So far, we described how the manager performs the processing in a "steady-state" execution. The critical task of the manager is then to take into consideration user requests at the initialization of the program as well as while the program is running (i.e., in that case, when feedback is provided). These tasks are at the core of the "managing" tasks of the `manager` object, and are described in the following (section 3.2).

3.1.3 Data organization

The last building block in the WP2 software concerns actual data. An object oriented approach is also used for storing all the signals and cues that were extracted in the various processing stages. In a similar way as the processor class described in section 3.1.1, a general parent "signal" class is implemented. All the signals and cues resulting from WP2

processing are then implemented as children of this class. All existing signals are then grouped in a single "data" object.

Signal class

Many signals of different nature are generated by the processing performed by the WP2 software. Although different signals have different properties (e.g., different dimensions, different scaling, different sampling frequency/time-frame,...), they share common properties. These common properties, as well as methods that all signal objects should have, are presented in Fig. 3.3. Signal objects allways contain:

- a `Label`, which formally describe a given *instance* of the object (e.g., "Left ear signal" or "Interaural level difference"). It is this label which is used, for instance, as a plot title when plotting the signal.
- a `Name`, which is a name tag associated to this signal type (e.g., "time", "innerhaircell", or "ILD")
- a description of its `Dimensions` (e.g., "m channels x n samples") to prevent inconsistencies
- a sampling frequency `FsHz`
- the actual `Data`, stored as an array

Two methods are then common to all signals:

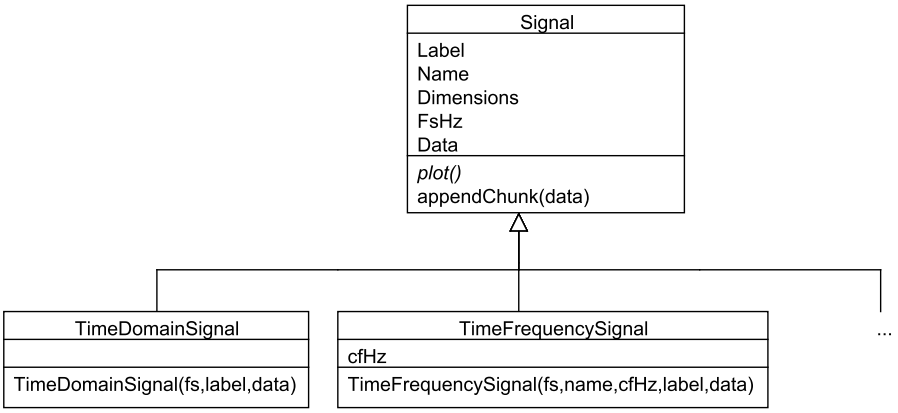

Figure 3.3: Signal parent and children classes.

- `plot()` which plots the signal. Because signals are of different dimensions, the method is abstract at this point and needs to be implemented by each child class.
- `appendChunk(data)` which adds the new signal chunk `data` (e.g., a recently computed output) to the already existing data.

Child signal classes inherit these properties and methods. They are implemented according to their dimensionality. For example, Fig. 3.3 shows diagrams for the time domain signal child class (e.g., used for a signal recorded at the ear) as well the time-frequency signal child class (e.g., used for a gammatone filterbank output, a inner hair-cell envelope,...). Specific child classes are then added for signals of a different nature.

Data object

Several signals of different nature are instantiated during the process of WP2. They are all collected in a single instance of a `dataObject`. The manager class responsible for WP2 processing then interacts with this data object. Each property of the `dataObject` is a `Signal` object. The name of the property is set by the signal property `Signal.Name`. For example if an inner hair-cell representation is requested for a single signal, there will be three properties in the `dataObject`:

- `dataObject.time`
- `dataObject.gammatone`
- `dataObject.innerhaircell`

If several signals with the same name exist they are collected in the same property, as an array of objects. Apart from its constructor, the `dataObject` class only has one method, `addSignal(sig)`, which adds the signal object `sig` to its properties.

3.1.4 General overview

Figure 3.4 summarises the WP2 software design. A standard arrow denotes an interaction (e.g., between the manager and the data object). An arrow ending with a filled diamond shows a composite aggregation, i.e., the object touching the diamond embeds one or several instances of the object at the other end of the arrow (e.g., the manager instantiates one or more processors). The numbers by the arrows ends indicate the possible number of instances, with $*$ being any integer (e.g., there can be one or more signal objects in the data object, zero or more filters in a processor, but there is only one manager).

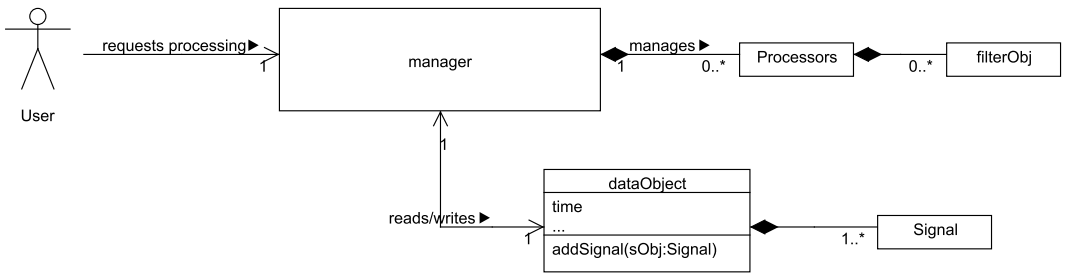

Figure 3.4: Overall class diagram for WP2 software

3.2 Handling user requests

The manager class is responsible for instantiating processors, ordering the processing, and routing inputs and outputs between processing stages. But it is not stand-alone, in the sense that it will be prompted by a "user" to extract some internal representations. In most scenarios, the "user" is of course not a physical person, but software from one of the other work packages. Two prerequisites concerning the way a user can request a given representation are crucial. First, the user should be allowed to request one single representation without explicitly requesting the other representations necessary to compute the original request. Second, requests are not only made at start-up, but should occur at any time during processing. Practical solutions to these two problems are presented in the following subsections.

3.2.1 Dependencies

As described above, a single processor object is responsible for only a single processing stage. However a given signal will likely be derived from another signal, itself deriving from yet another one. In other terms, there is a chain of dependencies between the existing signals, and multiple processing stages are required to derive only a single signal. The manager needs to know of these dependencies, and instantiate not only the processor responsible of a requested signal, but also the processors needed for the signals it depends on. It should also be aware of which order to call in the processors for generating an output.

In practice, the instantiation of the processors occurs in the constructor of the `manager` class. The constructor is called with a list of requested signals (`manager(request,...)` in Fig. 3.2). This list does not explicitly state the dependent signals. Instead, the manager calls an external function (`getDependencies`) that returns the full list of dependencies

for a given signal and instantiates the processors needed for each dependent signal. The list returned by `getDependencies` can be ordered in decreasing order of dependency (i.e., increasing processing order), such that the mapping `manager.Map` can be initialized to $(1, 2, ..., n_{proc})$ where n_{proc} is the total number of processors. As an example, say the user requests the computation of level differences (ILDs). The ILDs depend on the inner hair-cell envelope of the output of a Gammatone filterbank. The list of dependencies (as returned by `getDependencies`) therefore looks like:

$$\texttt{ild} \rightarrow \texttt{innerhaircell} \rightarrow \texttt{gammatone} \rightarrow \texttt{time}$$

The processing order is given by the decreasing dependency order:

$$\texttt{manager.Processors} = \{\texttt{timeProc}, \texttt{gammatoneProc}, \texttt{innerhaircellProc}, \texttt{ildProc}\}$$

and the mapping `Map` is initialized to

$$\texttt{manager.Map} = \{1, 2, 3, 4\}$$

Additionally, when instantiating a processor, the manager will populate its `processor.Dependencies` property with a pointer to the processor(s) that are one level below in terms of dependency. For example, for the case above, `innerhaircellProc.Dependencies` will point to `gammatoneProc`. This will help in dealing with feedback as is described in the following subsection.

3.2.2 Feedback

A crucial point in the philosophy behind the Two!Ears framework is that the bottom-up auditory processing should take into account decisions taken by higher-stage models. The WP2 framework must therefore be designed to include such top-down feedback, and evolve according to it. In practice, higher-stage feedback will be initiated by requesting a change in parameters for one or more processing stages, or requesting a completely new processing stage (e.g., extracting a new auditory feature). But this has to be done "on the fly", i.e., during execution of the processing and when the manager has already been instantiated.

When a new processing stage is requested, the manager needs to assess whether or not a processor corresponding to this request already exists. It should not only compare the tasks of the processor, but also the particular parameters under which the processing is carried. For instance, say the feedback is a request for an inner hair-cell representation using the model `'dau'`. If an inner hair-cell processor already exists but uses the method `'hilbert'`, the manager has to be "aware" of this discrepancy and instantiate a new inner hair-cell processor that would use the correct method.

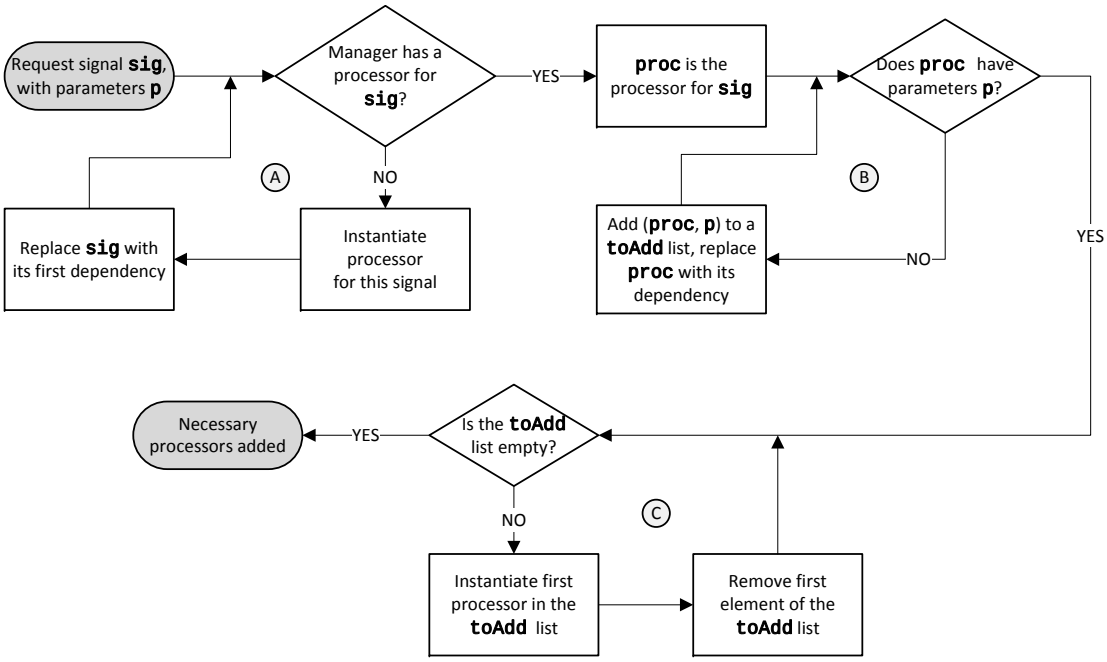

Figure 3.5: Flowchart picturing the operations performed when feedback is received in order to create adequate new processors. The feedback consists in a request for a signal `sig` with a set of parameters `p`.

Further, if a processor corresponding to the request already exists, then the manager needs to investigate if its dependencies also use the adequate parameters and if not instantiate all the "missing" processors. In practice, the process is illustrated in Fig. 3.5. Three recursive loops are present in the diagram of Fig. 3.5 marked as A, B, and C. The "user" request consists of a signal name (`sig`) that should be computed using a set of parameters `p`. The list of parameters `p` contains all the parameters for all the processing stages needed to obtain `sig`. The first loop A resembles the process described in section 3.2.1, where a given processor and its dependencies are instantiated. However it will stop when one of the dependent processors already exists and moves on to loop B. Loop B verifies that the already existing processor `proc` returned by loop A has the suitable parameters `p`. If not, it needs to find the first of its dependent processors that does have the correct parameters while keeping track of future processors to instantiate in a `toAdd` list. Loop C then instantiates all the necessary processors.

In practice the operations carried out in the three loops A, B and C from Fig. 3.5 are facilitated by the manager class methods `hasProcessor` and `addProcessor`, as well as the `hasParameters` method from the processor class. Along with the instantiation of the processors, the input/output addresses stored in the manager are updated (though not shown on Fig. 3.5). New signals in the data structure are created for every new processor

that is instantiated, even if the new processor only computes an already existing signal, only with a different parameter. This design limits confusion between processing stage and signals in the data structure. However, as the framework develops and is used in more complicated scenarios, solutions will have to be devised to avoid memory leaks by deleting obsolete signals and processors.

3.3 Available processors

In the following a list of currently available processors is presented, together with their corresponding MATLAB function names. A distinction is made between the processors that are used to extract signals and cues. The time, gammatone and inner hair cell (IHC) signals are sample-based, whereas the correlation-based signals and all cues are computed on the basis of short time frames. The frame size and the frame shift are general parameters and can be controlled by the flags `wSizeSec` and `hSizeSec`, respectively. As discussed in Sec. 3.2.1, the computation of a particular signal or cue will depend on the extraction of other signals and cues. Therefore, an overview of the corresponding dependencies for all supported signals and cues is given in Fig. 3.6.

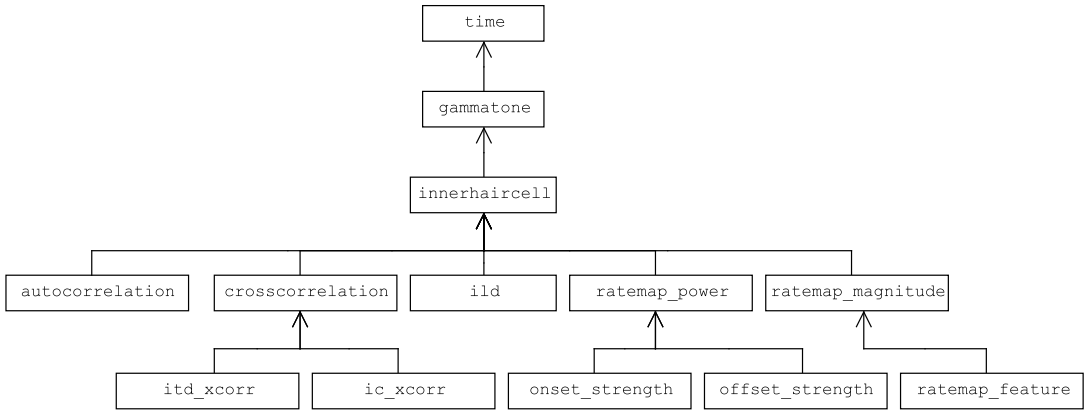

Figure 3.6: Diagram showing the dependencies of individual signals and cues.

3.3.1 Signals

Time (`timeProc.m`)

The left and the right ear time domain signals can be pre-processed by resampling the input to a new sampling frequency `fsHz`. In addition, the flag `bRemoveDC` can be used to

activate a DC removal filter, which applies a 4th order Butterworth high-pass filter with a cut-off frequency of 50 Hz. Finally, the flag `bNormRMS` can be used to normalize the time domain signal according to its root mean square (RMS) value. In case the input signal is binaural, the larger RMS value will be used for normalization.

Gammatone (`gammatoneProc.m`)

The time domain signal is processed by a bank of gammatone filters that simulates the frequency selective properties of the human basilar membrane (BM). The corresponding MATLAB function is adopted from the AMToolbox. An overview about the functionality of the toolbox can be found in Søndergaard and Majdak (2013). The gammatone filters cover a frequency range between `flow` and `fhigh` and are linearly spaced on the equivalent rectangular bandwidth (ERB) scale (Glasberg and Moore, 1990). In addition, the distance between adjacent filter center frequencies on the ERB scale can be specified by `nERBs`, which effectively allows to control the frequency resolution of the gammatone filterbank. The filter order, which determines the slope of the filter skirts, is set to `n = 4` by default.

Inner hair cell (`innerhaircellProc.m`)

The IHC functionality is simulated by extracting the envelope of the output of individual gammatone filters. The corresponding IHC function is adopted from the AMToolbox. Typically, the envelope is extracted by combining half-wave rectification and low-pass filtering. The cut-off frequency and the order of the corresponding low-pass filter vary across methods and the following flags for `ihcMethod` are supported: Hilbert transform `'hilbert'`, half-wave rectification `'halfwave'`, low-pass filtering `'dau'` (Dau *et al.*, 1996), and low-pass filtering, compression and expansion `'bernstein'` (Bernstein *et al.*, 1999).

Auto-correlation (`autocorrelationProc.m`)

Autocorrelation is an important computational concept that has been extensively studied in the context of predicting human pitch perception (Licklider, 1951, Meddis and Hewitt, 1991). To measure the amount of periodicity that is present in individual frequency channels, the normalized auto-correlation function (ACF) is computed based on the IHC representation of the left and the right-ear signals.

For the purpose of pitch estimation, it has been suggested to modify the signal prior to correlation analysis in order to reduce the influence of the formant structure on the resulting

ACF (Rabiner, 1977). This pre-processing can be activated by the flag `bCenterClip` and the following nonlinear operations can be selected for `ccMethod`: center clip and compress `'clc'`, center clip `'cc'`, and combined center and peak clip `'sgn'`. The percentage of center clipping is controlled by the flag `ccAlpha`, which sets the clipping level to a fixed percentage of the frame-based maximum signal level.

Cross-correlation (`crosscorrelationProc.m`)

The IHC representations of the left and the right ear signals are used to compute the normalized cross-correlation function (CCF) for short time frames. The normalized CCF is evaluated for time lags within `maxDelaySec` (e.g., $[-1\,\text{ms}, 1\,\text{ms}]$) and is thus a three-dimensional function of lag time, time frame and frequency channel.

3.3.2 Cues

Interaural level difference (`ildProc.m`)

The interaural level difference (ILD) is estimated for individual frequency channels by comparing the frame-based energy of the left and the right-ear IHC representations. The ILD is expressed in dB and negative values indicate a sound source positioned at the left-hand side, whereas a positive ILD reflects a source lateralized to the right-hand side.

Interaural time difference (`itdProc.m`)

The interaural time difference (ITD) between the left and the right ear signal is estimated by locating the time lag that corresponds to the most prominent peak in the normalized CCF. This estimation is further refined by a parabolic interpolation stage (Jacovitti and Scarano, 1993, May *et al.*, 2011).

Interaural coherence (`icProc.m`)

The interaural coherence (IC) is estimated by determining the maximum value of the normalized CCF. It has been suggested that the IC can be used to select time and frequency instances where the binaural cues (ITDs and ILDs) are dominated by the direct sound of an individual sound source, and thus, the corresponding binaural cues are likely to reflect the true location of one of the active sources (Faller and Merimaa, 2004).

Ratemap (`ratemapProc.m`)

The ratemap represents a map of auditory nerve firing rate (Brown and Cooke, 1994) and is frequently employed in computational auditory scene analysis (CASA) systems as a spectral feature. The ratemap is computed for individual frequency channels by smoothing the IHC signal representation with a leaky integrator that has a time constant of `decaySec`. Then, the energy is integrated across time frames and thus the ratemap can be interpreted as an auditory spectrogram.

Onset (`onsetProc.m`)

According to Bregman (1990), common onsets and offsets are important grouping principles that are utilized by the human auditory system to organize and integrate sounds originating from the same source across frequency. Onset are detected by measuring the frame-based increase in energy. This detection is performed based on the logarithmically-scaled energy, as suggested by Klapuri (1999).

Offset (`offsetProc.m`)

Similarly to onsets, offsets are detected by measuring the frame-based decrease in logarithmically-scaled energy.

3.4 Planned extensions to the software

The flexibility offered by the object oriented approach allows the WP2 framework to be easily extended. The main extensions will likely consist of adding new types of signals or cues that are requested by other work packages. New processor, signal, and possibly filter child classes will be added accordingly. Encapsulation then ensures that the addition of new components will not affect existing content.

In addition to updates based on the requests from other work packages, the following extensions are planned:

- As the number of available processors increases and the number of processing stages multiplies, there is an increasing risk of memory leaks (particularly when using the software in a scenario that includes feedback). As a precaution, a "garbage collector" should be implemented for the manager, that finds and removes processors and signals that are no longer in use.

- Similarly, having more processor types implies more parameters. Extensions to facilitate parameter handling (both for the software users and developers) will be designed.
- Extensions in which motor commands and/or proprioception are directly integrated with signal processing functions, in order to model reflexive processing.

4 Reference

4.1 WP2 reference

Processor	Parameters (type)	Options
`timeProc.m`	`fsHz`(int) `bRemoveDC`(boolean) `bNormRMS`(boolean)	
`gammatoneProc.m`	`fsHz`(int) `flow`(double) `fhigh`(double) `nERBs`(double) `n`(double)	
`innerhaircellProc.m`	`fsHz`(int) `ihcMethod`(char)	`'hilbert'`, `'halfwave'`, `'bernstein'` or `'dau'`
`autocorrelationProc.m`	`fsHz`(int) `bBandpass`(boolean) `bCenterClip`(boolean) `ccMethod`(char) `ccAlpha`(double)	`'clc'`, `'cc'` or `'sgn'`
`crosscorrelationProc.m`	`fsHz`(int) `maxDelaySec`(double)	

Table 4.1: List of available signal processors. A detailed description of the individual processors can be found in Section 3.3.

Acronyms

ACF auto-correlation function

ASA auditory scene analysis

BM basilar membrane

CCF cross-correlation function

CASA computational auditory scene analysis

ERB equivalent rectangular bandwidth

ILD interaural level difference

ITD interaural time difference

IC interaural coherence

IHC inner hair cell

RMS root mean square

WP1 work package one

WP2 work package two

Bibliography

Bernstein, L. R., van de Par, S., and Trahiotis, C. (**1999**), "The normalized interaural correlation: Accounting for N_0S_π thresholds obtained with Gaussian and "low-noise" masking noise," *Journal of the Acoustical Society of America* **106**(2), pp. 870–876. (Cited on page 17)

Bregman, A. S. (**1990**), *Auditory scene analysis: the perceptual organization of sound*, MIT Press. (Cited on pages 3 and 19)

Brown, G. J. and Cooke, M. P. (**1994**), "Computational auditory scene analysis," *Computer Speech and Language* **8**(4), pp. 297–336. (Cited on page 19)

Cooke, M., Brown, G. J., Crawford, M., and Green, P. (**1993**), "Computational auditory scene analysis: listening to several things at once," *Endeavour* **17**(4), pp. 186–190. (Cited on page 4)

Dau, T., Püschel, D., and Kohlrausch, A. (**1996**), "A quantitative model of the "effective" signal processing in the auditory system. I. Model structure," *Journal of the Acoustical Society of America* **99**(6), pp. 3615–3622. (Cited on page 17)

Ellis, D. P. W. (**1996**), "Prediction-driven computational auditory scene analysis," Ph.D. thesis, Massachusetts Institute of Technology. (Cited on page 4)

Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. (**1980**), "The Hearsay-II speech understanding system: integrating knowledge to resolve uncertainty," *Computing Surveys* **12**(2), pp. 213–253. (Cited on page 4)

Faller, C. and Merimaa, J. (**2004**), "Source localization in complex listening situations: Selection of binaural cues based on interaural coherence," *Journal of the Acoustical Society of America* **116**(5), pp. 3075–3089. (Cited on page 18)

Glasberg, B. R. and Moore, B. C. J. (**1990**), "Derivation of auditory filter shapes from notched-noise data," *Hearing Research* **47**(1-2), pp. 103–138. (Cited on page 17)

Godsmark, D. and Brown, G. J. (**1999**), "A Blackboard Architecture for Computational Auditory Scene Analysis," *Speech Commun.* **27**(3-4), pp. 351–366, URL `http://dx.doi.org/10.1016/S0167-6393(98)00082-X`. (Cited on page 4)

Jacovitti, G. and Scarano, G. (**1993**), "Discrete time techniques for time delay estimation," *IEEE Transactions on Signal Processing* **41**(2), pp. 525–533. (Cited on page 18)

Klapuri, A. (**1999**), "Sound onset detection by applying psychoacoustic knowledge," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3089–3092. (Cited on page 19)

Lesser, V. R., Nawab, S. H., and Klassner, F. I. (**1995**), "IPUS: An architecture for the integrated processing and understanding of signals," *Artificial Intelligence* **77**, pp. 129–171. (Cited on page 4)

Licklider, J. C. R. (**1951**), "A duplex theory of pitch perception," *Experientia* **7**(4), pp. 128–134. (Cited on page 17)

May, T., van de Par, S., and Kohlrausch, A. (**2011**), "A probabilistic model for robust localization based on a binaural auditory front-end," *IEEE Transactions on Audio, Speech, and Language Processing* **19**(1), pp. 1–13. (Cited on page 18)

Meddis, R. and Hewitt, M. J. (**1991**), "Virtual pitch and phase sensitivity of a computer model of the auditory periphery. I: Pitch identification," *Journal of the Acoustical Society of America* **89**(6), pp. 2866–2882. (Cited on page 17)

Rabiner, L. R. (**1977**), "On the use of autocorrelation analysis for pitch detection," *IEEE Transactions on Audio, Speech, and Language Processing* **25**(1), pp. 24–33. (Cited on page 18)

Søndergaard, P. L. and Majdak, P. (**2013**), "The auditory modeling toolbox," in *The Technology of Binaural Listening*, edited by J. Blauert, Springer, Heidelberg–New York NY–Dordrecht–London, chap. 2, pp. 33–56. (Cited on page 17)