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

Deliverable 3.1

Software Architecture

WP3 *

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:	WP3
Document number:	D3.1
Document title:	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

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.

Bottom-up cues are provided as input to a blackboard system, which consists of a collection of independent knowledge sources (KS) that communicate by reading and writing data on a globally-accessible data structure (the blackboard). The blackboard is divided into layers, which describe hypotheses at different levels of abstraction. Our blackboard system uses an event-driven design for efficiency; when data is placed on the blackboard, an event is broadcast which can be responded to by a KS with a matching precondition. Graphical models play an important role in the blackboard. KS are able to access a general graphical model maintained by the blackboard, and can perform inference on certain nodes in order to generate new evidence. A KS can also contain its own graphical model.

As a proof of concept, a specific instantiation of the software architecture is described which localises and identifies a single sound source. It is shown that top-down feedback in the system plays a crucial role when front/back confusions occur, prompting head movements that allow the confusions to be resolved.

It should be noted that the current document includes deliverable D2.1 in order to give a complete overview of the Two!Ears software architecture. This document should therefore be considered the most comprehensive account of the software architecture as of 31st May 2014.

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

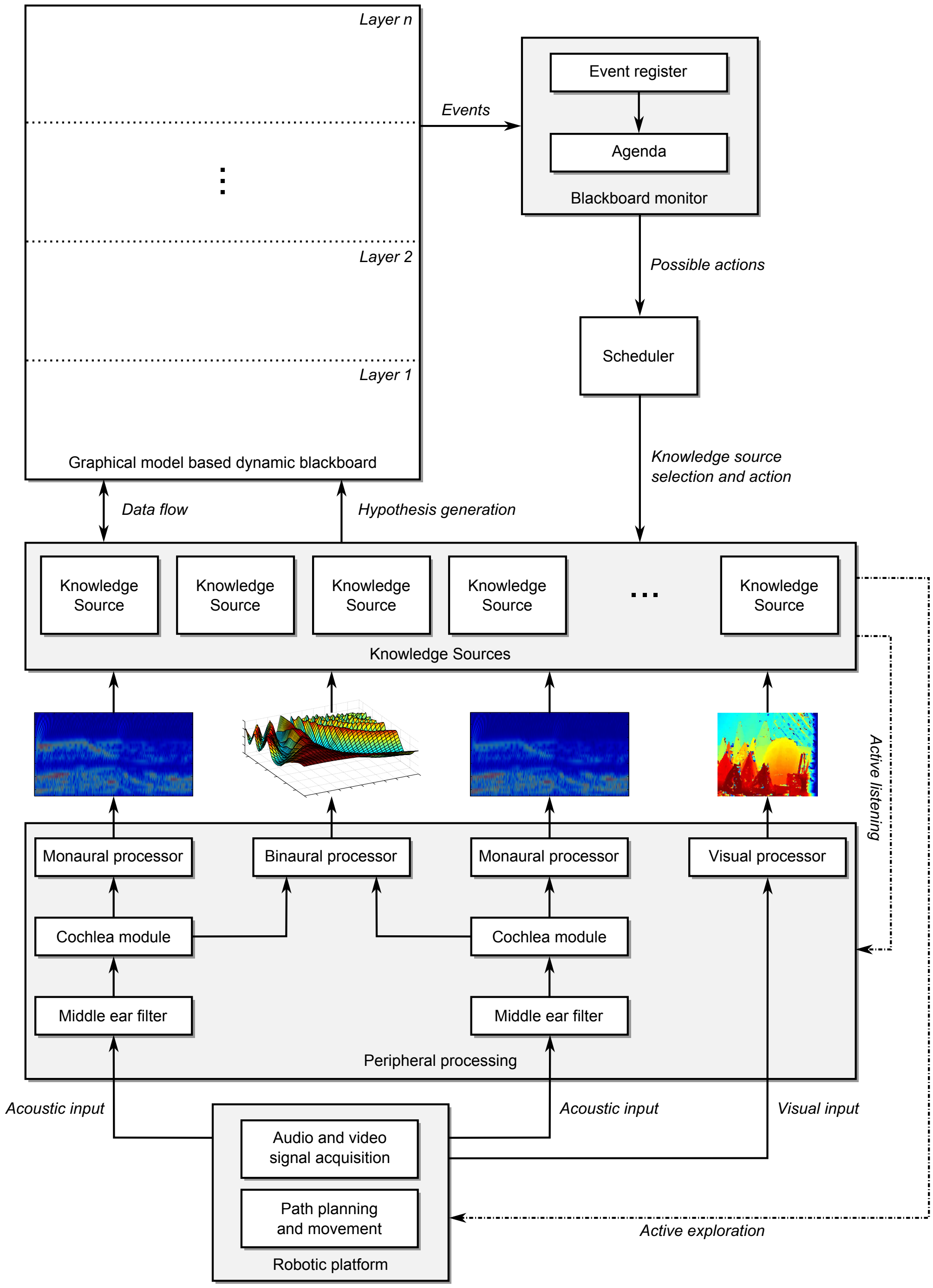

Figure 2.1: System diagram of the general software architecture

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. An example of such an approach is given in Section 5.

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

The remainder of the report is organised as follows. Section 3 describes the bottom-up auditory signal processing techniques developed within work package two (WP2). Section 4 then explains the design of the blackboard system in detail, corresponding to the output of work package three (WP3). Up to this point, the system is described in general terms. As a proof of concept, a particular instantiation of the architecture is then described in Section 5, which solves the problem of localizing and identifying a single sound source. It is shown that top-down and bottom-up interactions in the blackboard allow front/back confusions to be resolved. Finally, Section 6 provides reference material that will be helpful in using our Matlab implementation of the Two!Ears architecture.

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

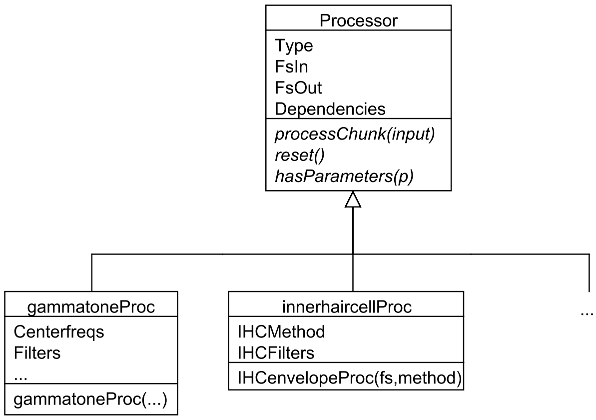

Figure 3.1: Class diagram of the processor parent and children classes.

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

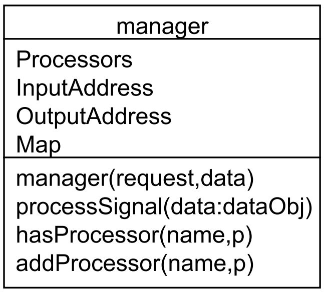

Figure 3.2: Manager class diagram.

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:

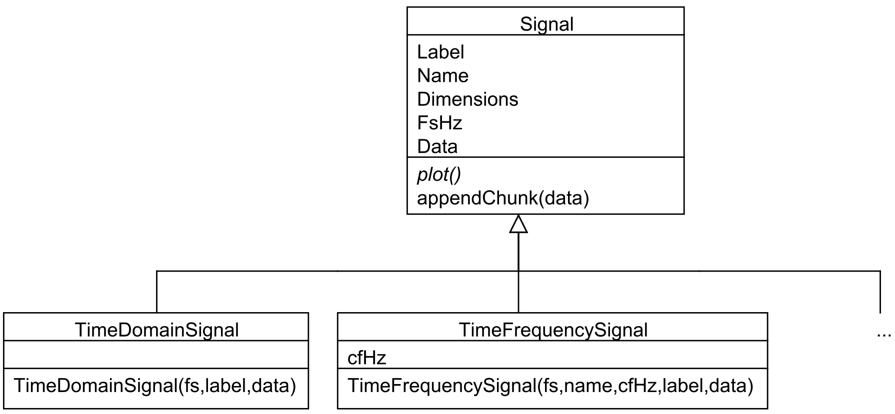

Figure 3.3: Signal parent and children classes.

- `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).

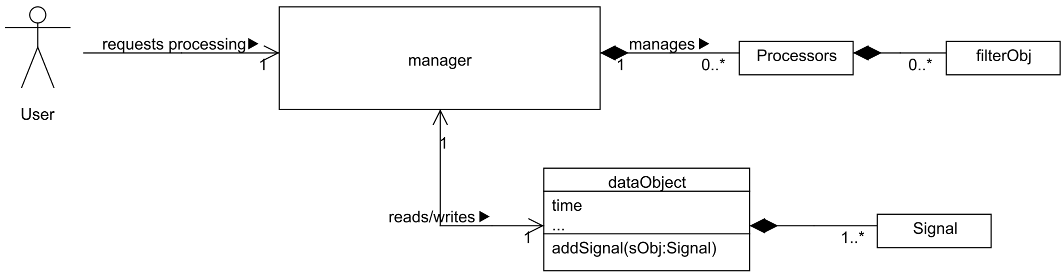

Figure 3.4: Overall class diagram for WP2 software

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.

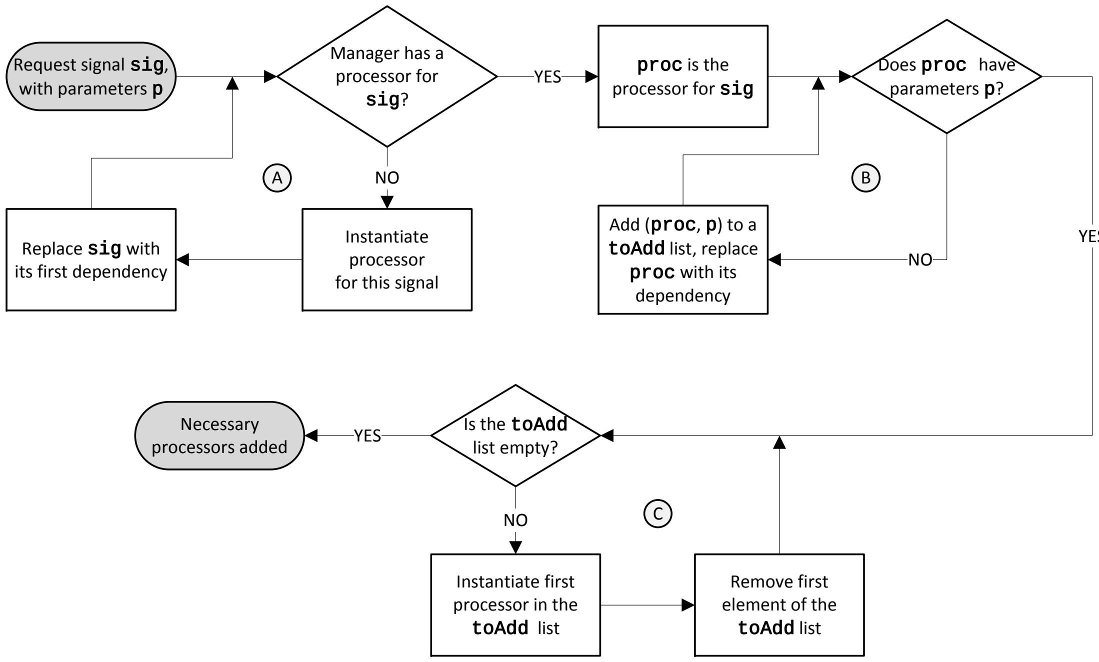

Figure 3.5: Flowchart picturing the operations performed when feedback is received in order to create adequate new processors. The feedback consists in a request for a signal `sig` with a set of parameters `p`.

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.

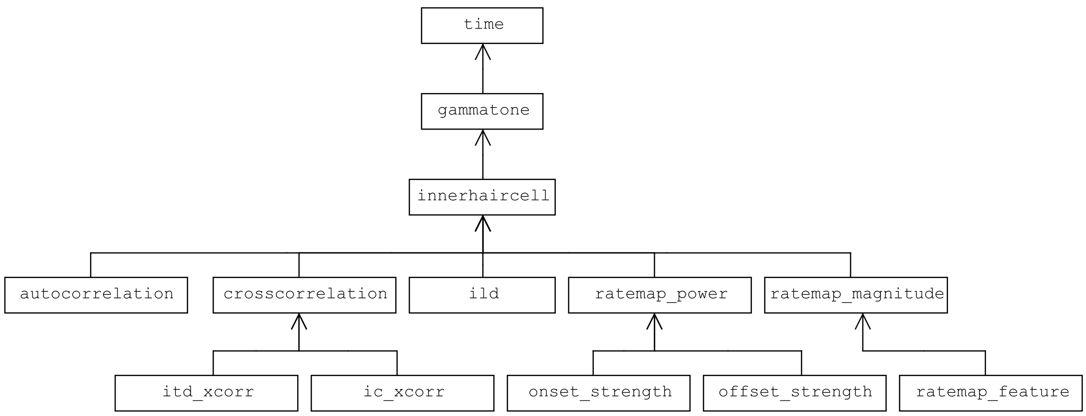

Figure 3.6: Diagram showing the dependencies of individual signals and cues.

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., $[-1\,\mathrm{ms}, 1\,\mathrm{ms}]$) 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 Blackboard system

Within the Two!Ears architecture, the blackboard system provides the substrate for modelling acoustic and audiovisual scenes, using prior knowledge in conjunction with current observations to build and maintain a belief state that should be as representative of the current environment as the state of information allows.

For this purpose, deterministic (or *rule-based*) knowledge can be integrated with statistical knowledge sources and with sensory information.

In the following, we will describe the software architecture, starting from design considerations and overall architecture in Section 4.1, moving on to the communication architecture in Section 4.2, more specific questions on the implementation of core computations in Section 4.3, and finally, we will describe planned extensions in Section 4.4.

4.1 Software design

The basic framework of the Blackboard System is implemented in Matlab. As in WP2, in order to provide a modular structure at an appropriate level of abstraction, the implementation is based on an object-oriented approach. Each blackboard component is encapsulated in a separate class. An overview of the main classes of the blackboard software architecture is shown in Fig. 4.1. This section will provide a basic explanation of the main classes that can be used to design a blackboard system using the current framework. A more detailed description of a specific instantiation of the blackboard will be given in Chapter 5.

4.1.1 Blackboard components

Blackboard base class (`Blackboard.m`)

The `Blackboard` class provides the core capabilities to manage communication with different knowledge sources (KSs) and keeps track of hypotheses that are generated during runtime. Initially, an instance of the `Blackboard` class has to be provided with a list of KSs, that

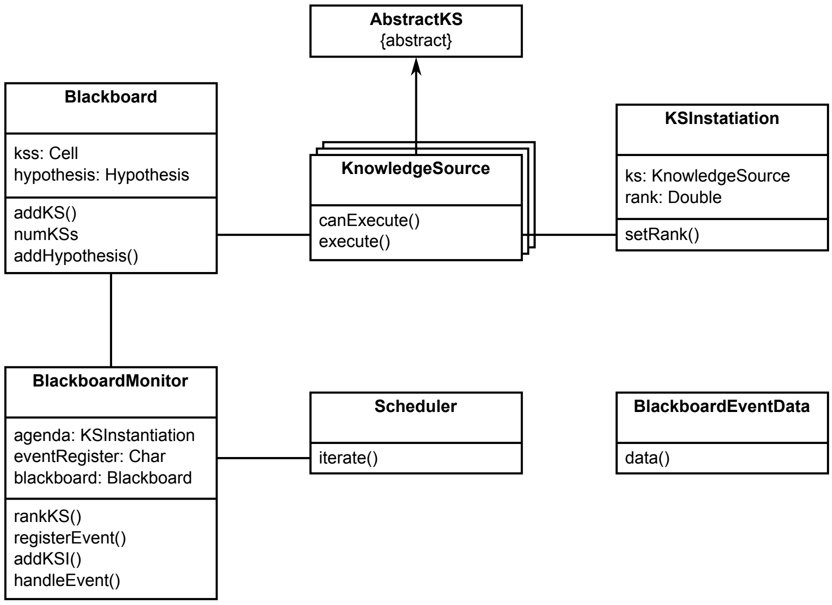

Figure 4.1: Class diagram, showing the main classes in the general software architecture.

will be internally stored within a MATLAB cell array. To add KSs to the blackboard, the method `addKS` can be used with an instance of a `KnowledgeSource` object as argument. In a similar way, hypotheses, that were generated by certain KSs, can also be added to the blackboard using the `addHypothesis` method during runtime. The blackboard stores and manages hypotheses internally and makes them accessible to the KSs. Furthermore, the blackboard is based on an event-driven approach, which allows the attachment of events to the generation of certain hypotheses. A detailed description of this approach will be given later Section 4.2.

Managing events (`BlackboardMonitor.m`, `BlackboardEventData.m`)

The event-based approach that is used here makes it necessary to manage and keep track of events that are generated on the blackboard. In the current system, this is realized within the `BlackboardMonitor` class, which contains a register of possible events and maintains an agenda that serves as a basis for decision regarding the scheduling of actions within the Blackboard System. Additionally, the `BlackboardEventData` class allows the attachment of data to a certain event. Section 4.2.1 gives a more detailed overview of the event-based processing and the role of this class.

Scheduling actions (`Scheduler.m`)

The role of the scheduler is to decide which knowledge source (KS) should be activated at a certain time. The `Scheduler` class provides functions to take this decision based on the agenda that is provided by the Blackboard Monitor. Currently, a simple ranking algorithm is implemented, where a specific weight is attached to each KS, so that the scheduler picks the KS with the highest weight from the current agenda. Future developments will focus on more advanced scheduling techniques that incorporate additional information from the blackboard state into the scheduling process.

4.1.2 Knowledge Sources

Base classes (`AbstractKS.m`, `KnowledgeSource.m`)

KSs have two basic properties: a precondition and a description of the actions that can be performed by a specific KS if the precondition is met. The `AbstractKS` class provides an abstract instantiation of those preconditions, that are described by the methods `canExecute` and `execute`. For the instantiation of a specific `KnowledgeSource` object, the corresponding class inherits the properties and methods of the `AbstractKS` class. The `KnowledgeSource` class serves as a flexible template for the construction of KSs. Preconditions have to be defined within the `canExecute` method, whereas the actions that are performed by the KS are described within the `execute` method. Currently, possible actions are

- acquisition of data from the Blackboard, the auditory preprocessing or the (simulated) robotic platform,
- the generation of new hypotheses on the blackboard and
- triggering feedback by sending specific commands and data to lower processing stages.

Some examples of possible KS actions are illustrated in Fig. 4.2.

Ranking (`KSInstantiation.m`)

As mentioned previously, the current scheduling algorithm is based on a simple ranking algorithm. The method `setRank` of the `KSInstantiation.m` class attaches a rank to a specific KS which is currently modeled as an integer in the range $[0, 100]$, where a higher number corresponds to higher importance in the scheduling process.

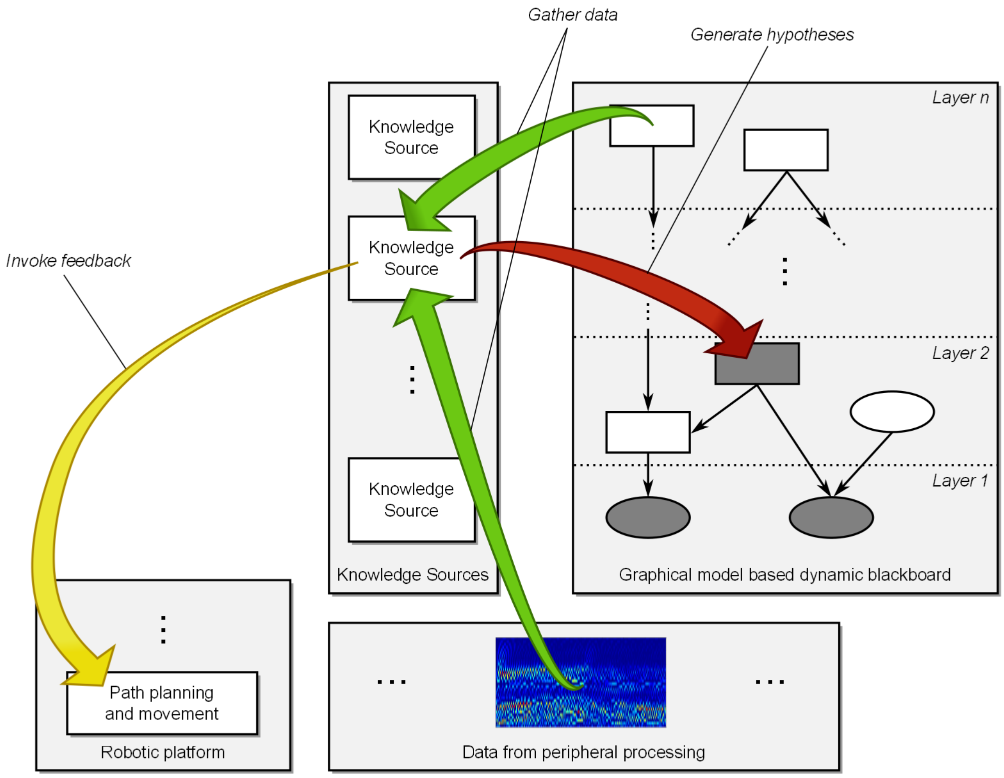

Figure 4.2: Examples of possible actions of KSs with other system components: data acquisition, hypothesis generation and feedback.

4.2 Communication within the blackboard system

Communication within the blackboard system is based on an event-driven architecture. Events are notices that the blackboard broadcasts in response to something that happens within the blackboard system, such as a knowledge source placing some new data on the blackboard space, or a feedback from a top-level knowledge source.

Fig. 4.3 shows the communication architecture. Events are monitored and handled by the *Blackboard Monitor*. Each knowledge source registers itself with the blackboard monitor as a responder to one or more events. When an event is triggered, the blackboard monitor identifies knowledge sources that have registered to respond to the event and whose preconditions are met. A scheduling agenda is then updated with all potential actions by the blackboard monitor, which are ranked and selected by the scheduler for execution.

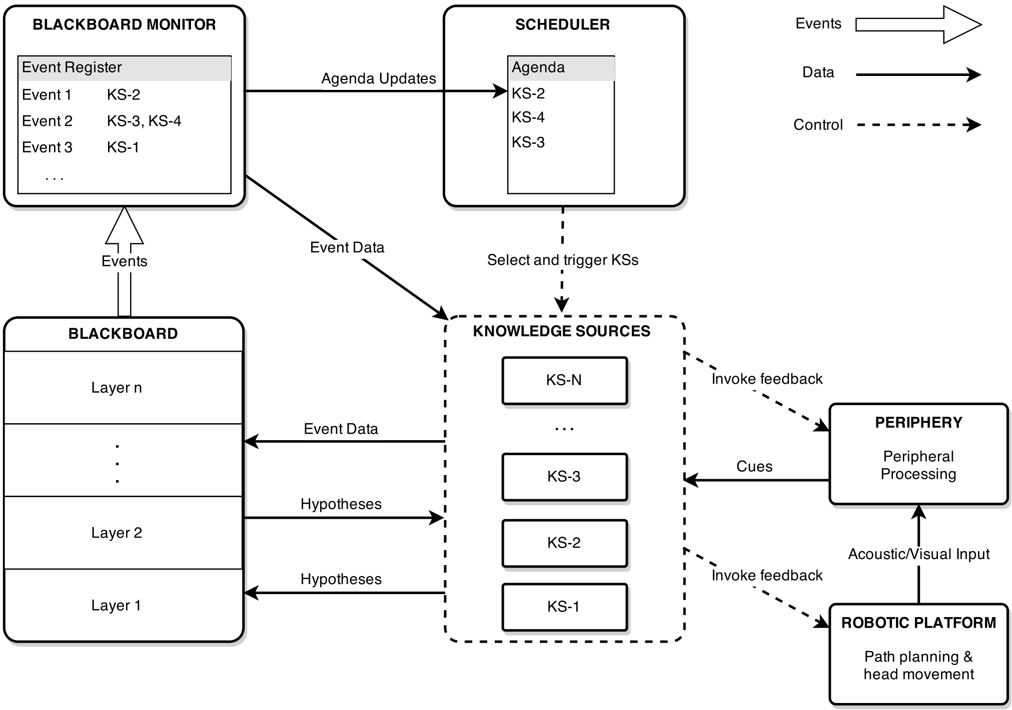

Figure 4.3: Event-driven architecture for communication with the blackboard system.

This event-driven architecture is designed to allow efficient interaction between different components within the blackboard system. Communication is only necessary when an event is triggered. Therefore, the blackboard system can focus its resources to the knowledge sources that subscribe to the triggered event, instead of constantly checking the preconditions of *all* the knowledge sources. More importantly, the event-driven mechanism provides an effective way for incorporating topdown feedback. This will be discussed in more detail in Section 4.2.3.

There are broadly two types of communications within the blackboard system:

- Interaction between a knowledge source and the blackboard (Section 4.2.2) involves mainly how hypotheses are placed on the blackboard and obtained from it. Placing hypotheses on the blackboard triggers an event so that other relevant knowledge sources can choose to respond.

- Topdown feedback (Section 4.2.3) is typically initiated by a top-level knowledge source by triggering a feedback-requesting event. Such an event invokes the relevant knowledge sources which have subscribed to it and the responding knowledge sources then perform relevant actions to complete the feedback pathway. Any feedback data (e.g., a new parameter for tuning characteristics of cochlear filters) can be sent together with the event.

4.2.1 Event-based communication

This section defines the standards for event-based communication. An event registration and notification system is used, which is a common design pattern in object-oriented systems (for example, see Gamma *et al.*, 1994).

Event Register

The blackboard monitor maintains internally an event register mapping each event to a list of knowledge sources that have chosen to respond to the event. It provides a method for a knowledge source to subscribe to one or more events as a responder.

```
BlackboardMonitor.registerEvent(eventName, KS-1, KS-2, ...)
```

where `eventName` defines the name of the triggering event and `KS-1, KS-2, ...` are a list of knowledge sources that want to subscribe to the event. A knowledge source can choose to respond to more than one event.

Event Notification

When a knowledge source makes changes to the blackboard space, or needs to invoke a feedback, an event is triggered and the blackboard is notified. This is typically done by a knowledge source using one of the two methods:

- Use `notify(blackboard, eventName)` to trigger an event specified by `eventName`.
- Use `notify(blackboard, eventName, BlackboardEventData(data))` to also attach some information to the event for its responders.

Registered events will be monitored by the blackboard monitor.

Event Data

Events provide information to the responding knowledge sources by attaching an event data argument to the event. The `BlackboardEventData` class contains only one public property `data`. An `BlackboardEventData` object can be constructed at the time when an event is being triggered.

4.2.2 Interaction with the blackboard

The `Blackboard` class provides various methods for accessing the different layers on the blackboard space. A knowledge source places hypotheses on a layer of the blackboard by calling the corresponding `addHypotheses()` method. An event is then triggered to notify the blackboard that new hypotheses are added to a particular layer. The knowledge sources that have subscribed to the event will be identified and eventually executed.

In response to an event indicating that some hypotheses have been placed on the blackboard, a knowledge source obtains the hypotheses by accessing the modified layer. The indices of the new hypotheses can be provided in the event data attached to the event for more efficient access.

4.2.3 Feedback pathways

An important aspect of the Two!Ears framework is the use of top-down feedback pathways that allow high-level decisions to influence bottom-up processing. For example, a high-level model could request a change in parameters used for low-level periphery processing, or plan a head rotation in order to solve front/back confusion in source localisation.

A feedback pathway is typically initiated by a high-level knowledge source by triggering an event that indicates a top-down request. Extra information regarding the request can be attached to the event as event data, e.g., a new set of periphery processing parameters or the planned head rotation azimuth. The feedback pathway is completed by a low-level knowledge source that invokes the feedback and responds to the event. Fig. 4.4 illustrates a typical feedback pathway based on the event-driven communication architecture.

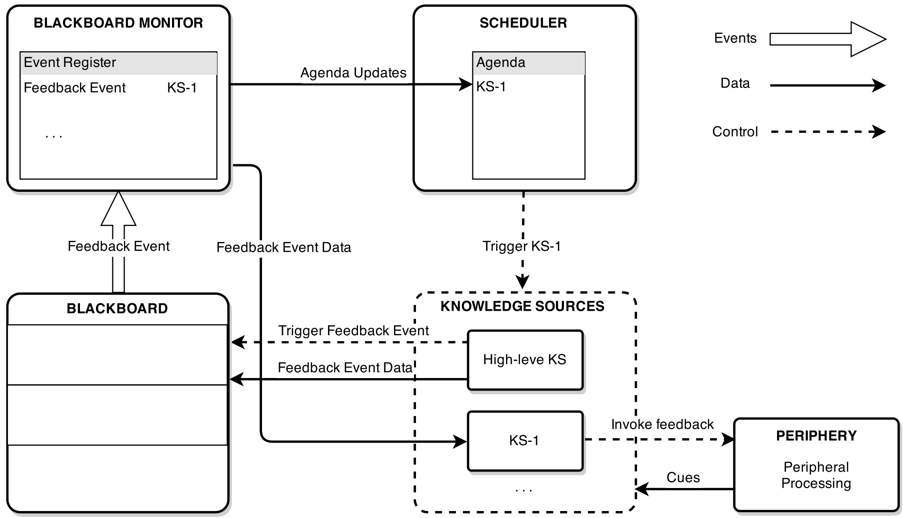

Figure 4.4: Illustration of feedback pathways based on the event-driven communication architecture.

4.3 Role of graphical models in the blackboard

To initialize and maintain the belief state of the system, graphical models have been selected due to their capability for fusing structural and statistical knowledge.

4.3.1 Motivation

Graphical models have recently attracted great interest within the fields of machine learning and cognitive systems. They describe relationships between statistical variables in the form of simple graph structures. In these graphs, each node corresponds to a variable, and each edge indicates a dependency relationship between variables.

In this way, graphical models can be used to describe the dependencies between all variables that are of interest, effectively providing a world model, which is not only mathematically useful but also interpretable.

Graphical models come in many different specific forms, like Hidden Markov Models, Markov Random Fields, or dynamic state space models, which are suitable for creating precise descriptions of the constituent components of acoustic or audiovisual scenes. Efficient algorithms have been developed, which allow to find the optimal fit between the model parameters and the observations taken from all sensors of a system.

In effect, this means that, based on a graphical model of the audiovisual objects in an environment, the Two!Ears-system will be able to find the best explanation of all available information, optimally fusing prior knowledge (e.g., linguistic or acoustic knowledge) with the currently available sensor input.

Taking graphical models as building blocks furthermore allows us to

- consecutively build models of the audiovisual environment from smaller, well-understood models of environmental objects (including state-of-the art statistical models of auditory objects)
- understand sensory data as a composition of these source models and a model of the system's own "perception" (describing sensory characteristics of the robot and its internal sensory processing)
- and to understand the system's interpretation of the audiovisual environment, by virtue of the interpretability of each component and of their connections.

Since the model is statistical in nature, the resulting interpretation of the environment will not only denote the type, number, location and — if applicable — the possible intention of all objects of interest, but also contain estimates of the variances (or probability distributions) of all of these quantities. This will endow the system with the ability to judge the reliability of its own interpretation, and can ultimately be used to design active listening and active exploration, so as to ensure that the most relevant variables are determined with sufficient reliability.

4.3.2 Interaction of graphical models with the blackboard

In practice the blackboard system maintains a general graphical model (GM) that represents interpretations of the environment. The KSs are able to access the general GM and can perform inference on certain nodes to cause new evidence to be placed on the blackboard.

Certain nodes within the GM can be switched on/off by knowledge sources before performing inference. The sequence and timing of inference with the general GM is controlled by the scheduler. This approach provides a flexible framework for handling data flow towards and within the blackboard. Since inference becomes computationally more demanding with growing model size, the scheduler provides an efficient means of specifying a computational agenda that performs inference only if a sufficient amount of new data is gathered.

A knowledge source can also have its own graphical model. A KS graphical model is entirely owned by the KS and cannot be accessed by other knowledge sources. Therefore its structure and information are the 'knowledge' of the knowledge source itself. However, KS graphical models can still interact with the general GM by providing new evidence or hypotheses to the blackboard. The evidence can be used as observation of certain nodes of the general GM.

4.4 Planned extensions to the software architecture

The current implementation of the blackboard architecture is embedded in a simulation environment (Wierstorf *et al.*, 2011), that generates ear signals via convolution of sound sources with head-related impulse response (HRIR)s. Though changes in the orientation of the head are already included, the simulation is restricted to limited sets of HRIRs. To overcome this limitation, it is planned to replace the current simulator with the SoundScape Renderer (SSR) (Geier *et al.*, 2008) that is provided and maintained by WP1. The SSR will allow the acoustic simulation of more complex scenes, including

- dynamic sound sources that move along prespecified trajectories,
- the simulation of room acoustics,
- a more realistic modeling of the robot kinematics, especially head movements, movement of the robot in space and
- the simulation of noise that is caused by the robot's actuators during movement.

In addition to that, the upcoming work on the software architecture will also focus on the integration of visual information into the simulation environment. It is planned to establish a link between the acoustic simulation and the Bochum Experimental Feedback Testbed (BEFT) (Walther and Cohen-L'hyver, 2014) that has been recently developed by work package four (WP4). The BEFT provides a visual simulation of the robotic platform and its surrounding environment, allowing the generation of artificially degraded visual cues, which can then be used as additional inputs to the blackboard system.

Finally, we note that the blackboard system will be ported on a robot, on top of a low-level real-time functional architecture (which will implement not only a C/C++ implementation of the functions developed in WP2, but also functions associated with robot locomotion, motion planning, localisation, execution of planned motions with reactive obstacle avoidance etc.). It will therefore be extended in order to handle messages (warnings, errors, failures) that arise from low-level robot processing.

5 Proof of concept

Here we present an example of the generic architecture – a system that localises and identifies a single source, solving front/back confusions in the process. Strictly this goes beyond the remit of the deliverable, but we anticipate that it will be helpful in communicating to the reader how the architecture will work in practice.

5.1 Scenario

The implementation of the specific Blackboard System that is described within this chapter is targeted towards a sound source localisation and identification scenario. The basic setup is illustrated in Fig. 5.1. It is assumed that the listener in this scenario is static, but changes in head orientation $\psi \in [0, 360]$ are possible. The head orientation corresponds to a world-coordinate system, where $\psi = 0$ denotes that the listener is facing towards north along the y-axis in Fig. 5.1. The orientation angle increases in a clock-wise direction. Additionally, a static sound source is placed at an arbitrary angle $\phi \in [0, 360)$ in the horizontal plane relative to the head orientation of the listener, which will be referred to as the *relative target source position*. In this scenario it is assumed that the listener and the sound source are located in an environment with free-field conditions. The simulation of the scenario is generated using HRIRs (Wierstorf *et al.*, 2011) acquired from a KEMAR dummy-head, recorded at a distance of 3 m between the head and the source.

The scenario involves the completion of two tasks: determination of the relative target source position and identification of presented sounds. The latter will be based on a variety of speech and non-speech sounds that can be presented within the scene. A core aim of the scenario is the reduction of localisation estimation errors caused by front/back ambiguities. Therefore, feedback, for instance by carrying out head movements for disambiguation, is necessary.

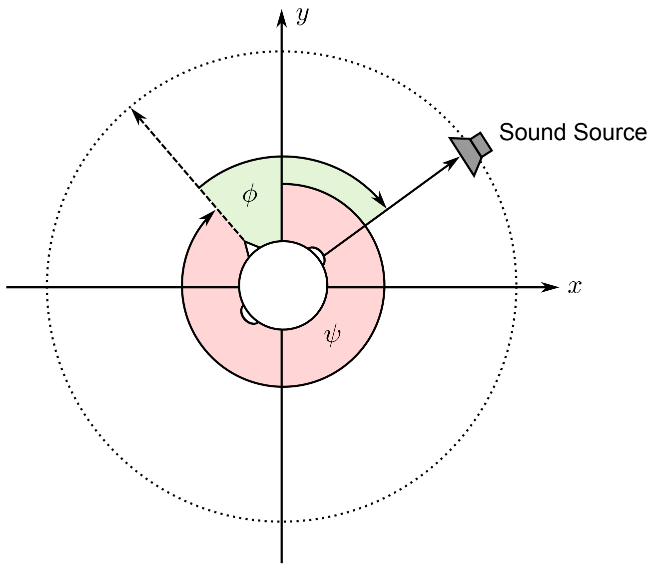

Figure 5.1: Illustration of the source localisation and identification task, showing the correspondence between the head orientation ψ and the relative target source position ϕ.

5.2 WP2 signal and cue extraction

To localize and identify a target source as described in Section 5.1, the WP2 software package is used to extract a set of monaural and binaural cues. Specifically, ITDs and ILDs are estimated for 20 ms time frames for the localization task, and the ratemap is extracted for the identification task. The configuration of the WP2 framework is specified as follows:

```
% Input signal parameters
SET.fsHz        = 44.1E3;      % Sampling frequency
SET.bRemoveDC   = false;       % Flag for DC removal
SET.bNormRMS    = false;       % Flag for RMS normalization

% Auditory periphery
SET.nErbs       = 1;           % ERB spacing of gammatone filters
SET.fLowHz      = 80;          % Lowest center frequency in Hertz
SET.fHighHz     = 8E3;         % Highest center frequency in Hertz
SET.ihcMethod   = 'halfwave'; % Hair-cell processing

% Binaural cross-correlation processor
SET.maxDelaySec = 1.1E-3;

```

```
% Framing parameters
SET.winSizeSec = 20E-3;     % Window size in seconds
SET.hopSizeSec = 10E-3;     % Window step size in seconds
SET.winType    = 'hann';    % Window type

% Specify cues that should be extracted
strCues = {'ratemap_power' 'ild' 'itd_xcorr' 'ic_xcorr'};

% Initialize all WP2-related parameters
STATES = init_WP2(strCues,SET);
```

Given the ear signals **earSignals** and the corresponding sampling frequency **fsHz**, the actual processing is performed by the following function call:

```
% Perform WP2 computation
[SIGNALS,CUES,STATES] = process_WP2(earSignals,fsHz,STATES);
```

The returned structures SIGNALS and CUES contain all involved signal representations, as well as the requested monaural and binaural cues, that are further processed by the subsequent processing stages.

5.3 Blackboard system for analysing a single-source scenario

5.3.1 The Blackboard

Fig. 5.2 shows a graphical representation of the blackboard architecture that is implemented for this demonstration scenario. The following sections give detailed descriptions about individual components. The blackboard workspace is arranged into a hierarchy of five layers:

- Layer 1: **Signal** is the lowest layer of the blackboard and provides raw signals that the blackboard system analyses. It is divided into two sublayers: a) a signal layer that provides waveform signal blocks; b) a periphery layer that provides signals from an auditory periphery model.
- Layer 2: **Acoustic Cues** provide cues extracted from the periphery signals. Currently spatial cues including ITDs, ILDs and interaural coherence (IC), and ratemap cues are used by the blackboard.
- Layer 3: **Location + Identity hypotheses** is the layer for probability distributions about possible source locations and possible identifications of certain sound classes.
- Layer 4: **Confusion hypotheses** are groups of locations where there could be a

confusion.

- Layer 5: **Perceptual hypotheses** are perceived source locations after confusions are resolved.

5.3.2 Knowledge Sources

The blackboard system includes eight KSs, each of which operates on different layers of the blackboard. From bottom to top, they include:

Signal KS

The **Signal** KS is the most primitive of the knowledge sources. It is used to create "bottom-up" waveform signal blocks for the blackboard when there are no "top-down" actions that can be taken.

Precondition: This KS checks a flag on the blackboard which indicates that no further actions can be taken and the system is ready for the next block. The precondition is satisfied if this flag is set and there are unprocessed incoming signals.

Action: The KS creates a new waveform block and places it on the blackboard. The size of a block can be specified by events that trigger this KS. The default block size is 500 ms. The KS causes the blackboard to generate a *NewSignalBlock* event.

Periphery KS

The **Periphery** KS includes the bottom-up auditory processing provided by the WP2 framework. It consists of the gammatone filterbank and inner hair cell processing stages and computes the corresponding auditory periphery signals for both ears and the normalized interaural cross-correlation. The outputs from this stage form a new periphery signal block.

Precondition: This KS is triggered by the *NewSignalBlock* event and checks if there are unprocessed signal blocks on the blackboard.

Action: The KS creates periphery signal blocks and places them on the blackboard. The KS causes the blackboard to generate a *NewPeripherySignal* event.

AcousticCue KS

The **AcousticCue** KS extracts various acoustic cues, namely ITD, ILD, IC and ratemaps, by using the corresponding processing functions that are provided by WP2.

Precondition: This KS is triggered by the *NewPeripherySignal* event and checks if there are unprocessed periphery signal blocks on the blackboard.

Action: The computed ITD and ILD cues are placed on the blackboard as new evidence for the localisation GM, along with the IC (currently not used, but will be incorporated in future versions of the blackboard system) and ratemap cues which are the main input for source identification.

Identity KS

An **Identity** KS has knowledge of acoustical cues for a certain sound class. Many Identity KSs can be used concurrently, for instance one for each sound class to be identified. In this demonstration, identification is done by a linear model trained with a SVM on separate data.

Precondition: This KS is triggered by the *NewAcousticCues* event. The precondition is met when new acoustical cues are placed on the blackboard which have not been processed.

Action: The KS predicts, based on the incorporated model, whether the current sound block is a member of the sound class this KS represents. An identity hypothesis is placed on the blackboard and a blackboard event *NewIdentityHypothesis* is generated.

Location KS

The **Location** KS has knowledge of how spatial cues should appear at each azimuth location. Here, it adopts a static GM which consists of an observed continuous random variable that represents spatial features, and a hidden discrete random variable that represents all possible location azimuths. An angular resolution of 5 degrees is used so that the cardinality of the location variable is 72. The probability distribution for the features conditional upon each location is modelled with Gaussian distributions. The GM parameters are estimated from training data generated using the same simulation.

Precondition: This KS is triggered by the *NewAcousticCues*. The precondition is met when new spatial cues are placed on the blackboard and have not been processed.

Action: The KS performs inference on the graphical model to compute posterior probabilities of all the locations given the spatial cue observation (ITD and ILD). A location hypothesis containing the probability distribution is then placed on the blackboard and a blackboard event *NewLocationHypothesis* is generated.

Confusion KS

The **Confusion** KS checks location hypotheses from the same frame and decides whether there is a confusion.

Precondition: This KS checks if new location hypotheses are put on the blackboard. The precondition is satisfied if the new location hypotheses have not been processed.

Action: Location hypotheses from the same frame are examined for potential confusion. In this demonstration scenario a confusion is identified if there are multiple location hypotheses in one frame. When a confusion is identified, a confusion hypothesis is created which includes all the competing locations and the blackboard event *NewConfusionHypothesis* is triggered. When a confusion is not found, a perceived source hypothesis is created and the *NewPerceivedLocation* event is triggered.

HeadRotation KS

The **HeadRotation** KS has knowledge on how to move the robotic head in order to solve confusions in source localisation.

Precondition: This KS checks if there is already a scheduled head rotation. The precondition is satisfied if there is a confusion and no head rotation has been scheduled.

Action: The KS stops the listening process and rotates the head by 10 degrees. After the rotation is completed, it raises the blackboard flag indicating that the system is ready for the next frame and generates an event *ReadyForNextBlock*.

ConfusionSolving KS

The **ConfusionSolving** KS solves localisation confusions by predicting the location probability distribution after head rotation, and comparing it with new location hypotheses received after head rotation. If an hypothesised azimuth location reflects a true source location, then the predicted location distribution and the observed distribution after head rotation should overlap at the location. Otherwise the hypothesised location is considered a 'ghost'.

Precondition: This KS searches the blackboard for an unprocessed confusion hypothesis. The precondition is satisfied when new location hypotheses are placed on the blackboard and the head orientation is changed from the value it had when the unprocessed confusion hypothesis was created.

Action: The KS makes predictions of source locations after head rotation, based on the confusing location hypotheses. The predicted locations are then checked with the new location hypotheses after head rotation. A source location is selected when there is a match and a source hypothesis is created and put on the blackboard. It is discarded if a match does not exist. The KS also checks whether the discarded location hypothesis is a 'ghost' from a front/back confusion, and the score of a ghost is added to the source hypothesis.

5.3.3 Blackboard Monitor

The blackboard monitor maintains a register for blackboard events and an agenda. When a blackboard event is triggered, the monitor identifies knowledge sources that have subscribed to the event and their preconditions are checked. The knowledge sources ready to respond to the event are added as actions to the agenda which are then ranked and selected by the scheduler for execution. Table 5.1 shows all the events and corresponding actions.

Table 5.1: Blackboard events and corresponding actions used in the agenda-based blackboard system for scenario 1

Event	**Responding Knowledge Sources**
ReadyForNextBlock	Signal KS
NewSignalBlock	Periphery KS
NewPeripherySignal	AcousticCue KS
NewAcousticCues	Location KS, Identity KS
NewIdentityHypothesis	None
NewLocationHypothesis	Confusion KS, ConfusionSolving KS
NewConfusionHypothesis	HeadRotation KS
NewPerceivedLocation	None

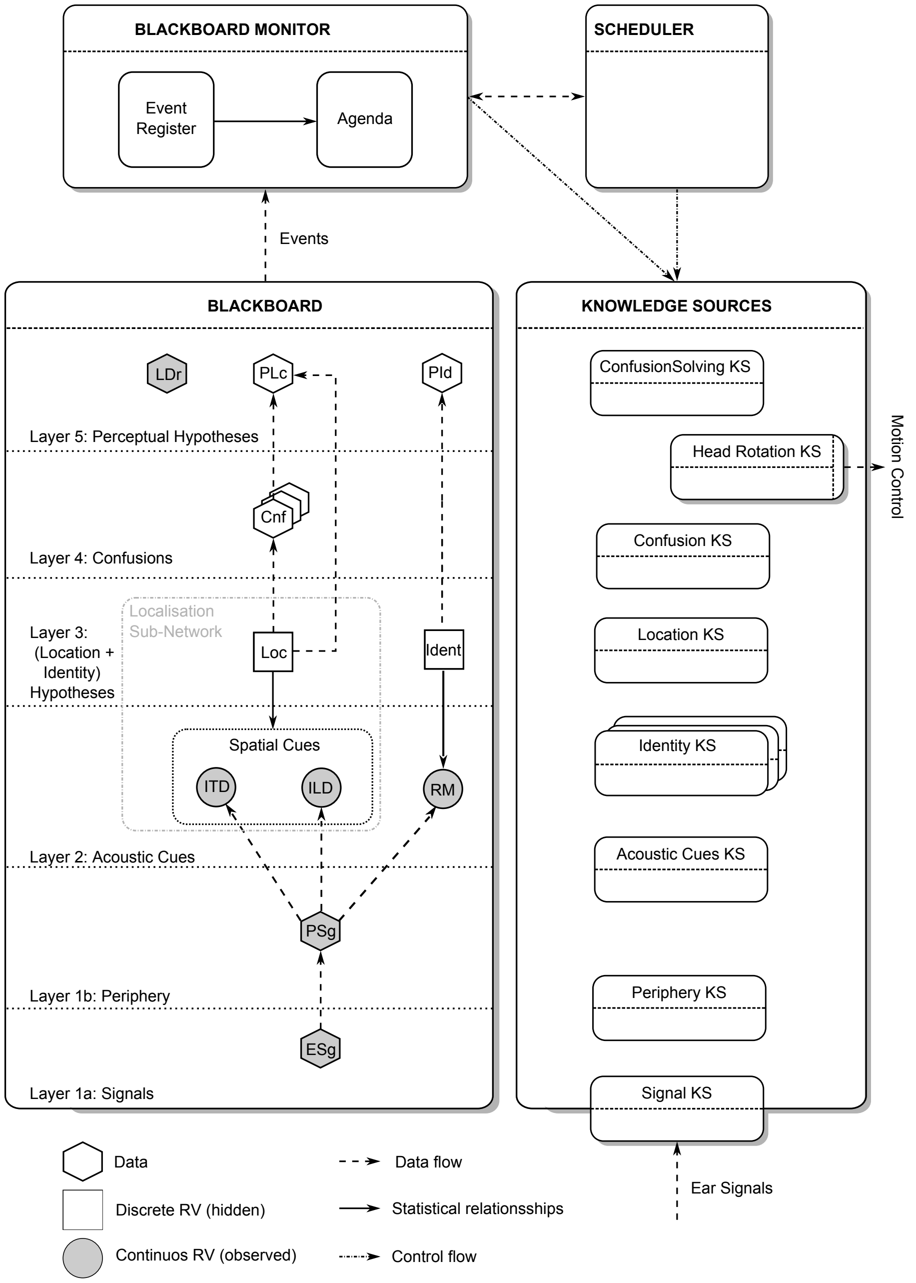

Figure 5.2: The agenda-based blackboard architecture adopted for the demonstration scenario.

5.4 Results

5.4.1 Sound localisation

Fig. 5.3 shows typical output from different layers of the blackboard system while localising a single sound source. The bottom panels show binaural waveform signals received from the left and right ears. Here the sound source is a speech signal located at 60° azimuth to the right, and it is clear that the amplitude of the right ear channel is significantly higher than that of the left ear channel. On top of the waveforms are the inner hair cell (IHC) signals generated by the auditory periphery model, which divides the signals into a number of frequency channels as shown along the Y-axis. These periphery signals form the basis from which the monaural and binaural cues are extracted. The corresponding binaural cues, namely the ITDs, ILDs and the IC are shown above as a function of the gammatone channel index and the frame index, along with the monaural cue ratemaps.

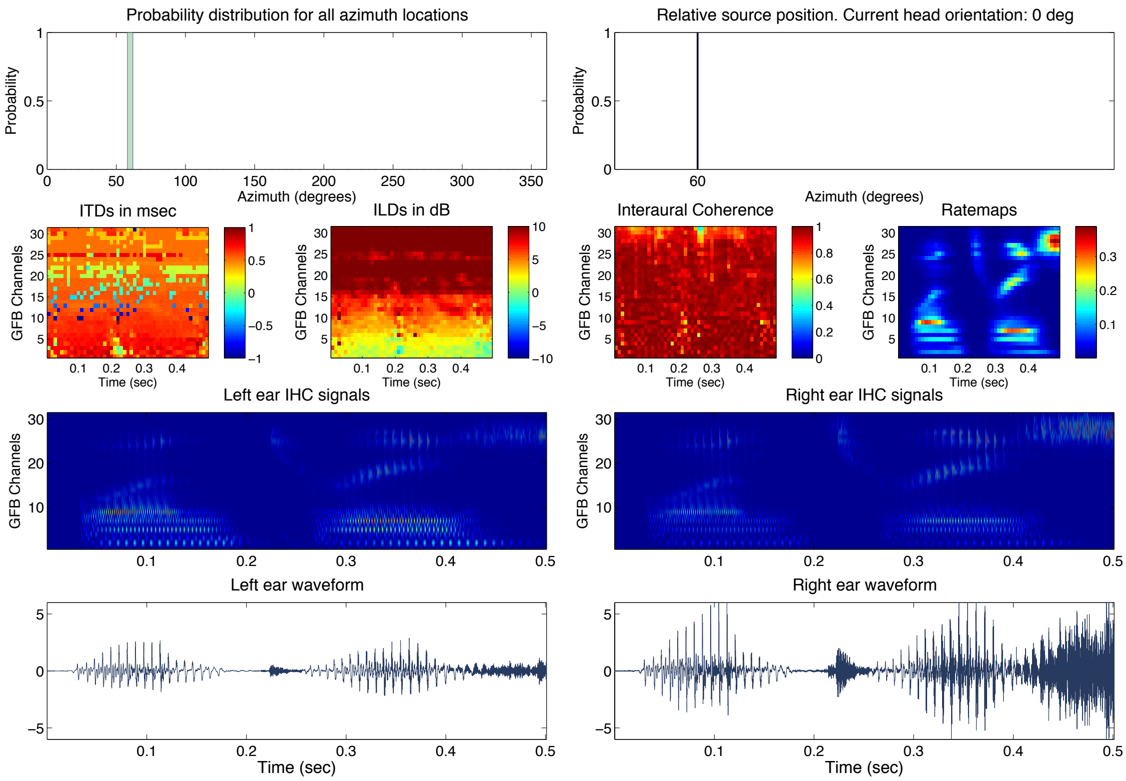

Figure 5.3: Typical output from the blackboard system at different layers while localising a single sound source.

The upper left panel shows a location hypothesis – a probability distribution for different azimuth locations – produced by the **Location** KS based using just the ITD and ILD cues. The 60° azimuth is clearly the most probable source position with a probability close to 1. It is confirmed by the **Confusion** KS that there is no ambiguity here, and 60° azimuth is output as the relative source position as shown in the upper right panel.

Front/back confusion

Fig. 5.4 gives an example where the blackboard system has to solve front/back confusion. Here a speech source is located at 30° azimuth. In the upper panel, the location hypothesis produced by the **Location** KS exhibits a high location probability at both 30° and 150° azimuth, as a result of the front/back confusion. The confusion is identified by the **Confusion** KS which triggers a confusion event and initiates a feedback pathway for a head rotation request.

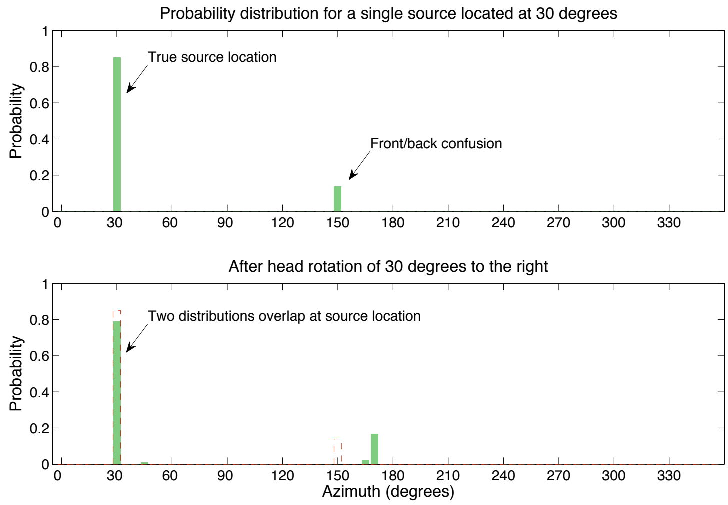

Figure 5.4: Illustration of front/back confusion solving in the demonstration scenario. The upper panel shows probability distribution for different positions for a source located at 30° azimuth, and there clearly exists a 'ghost' at 150° azimuth. The lower panel shows the predicted location distribution in dotted lines and the actual distribution after head rotation by 10°. The two distributions overlap at 30° azimuth which suggests a true source position.

The **HeadRotate** KS invokes head rotation and completes the feedback pathway. After the head is rotated and new signals are received, the **Location** KS places a new location hypothesis on the blackboard, as shown in the lower panel of Fig. 5.4. In order to solve the front/back confusion, the **ConfusionSolving** KS makes a prediction of the new distribution based on the location hypotheses before head rotation and this is shown as dashed lines in the lower panel of Fig. 5.4. The two distributions overlap at the 30° azimuth and this azimuth is selected as the most likely source position. Since the 150° azimuth before head rotation no longer exhibits a high probability in the new distribution it is considered as a 'ghost'.

5.4.2 Sound identification

Concerning the identification task, the blackboard system (in particular the Identity KSs), produces hypotheses about sound event identities. Each Identity KS is responsible for detecting one particular event class. In this demonstration, two Identity KSs are employed with models to detect keys put on a table and knocking on a door, respectively. Figure 5.5 shows output of the blackboard system after having simulated a scene which included various different sound events, displaying the hypotheses of the system (red) versus the true labelings (green).

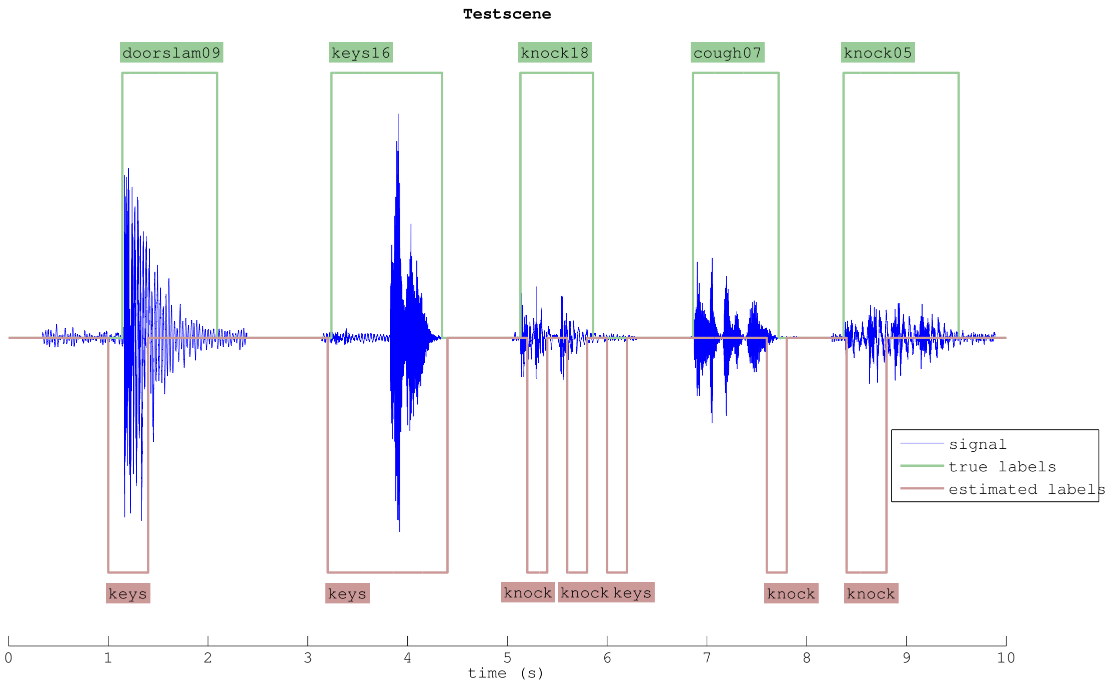

Figure 5.5: Typical output from the blackboard sound event identification system. The red line shows the times at which active identity hypotheses were put on the blackboard and their classes, while the green line indicates the true label of the sounds at these times in the scene.

6 Reference

6.1 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 6.1: List of available signal processors. A detailed description of the individual processors can be found in Section 3.3.

6.2 WP3 reference

Class	Parameters (type)	Description
`Blackboard.m`	`scene`(Scene)	scene object describing the simulated scene
`SignalBlockKS.m`	`bb`(Blackboard) `simParams`(struct)	blackboard object struct containing the simulation parameters
`PeripheryKS.m`	`bb`(Blackboard) `simParams`(struct) `wp2States`(struct)	blackboard object struct containing the simulation parameters auditory frontend parameters
`AcousticCuesKS.m`	`bb`(Blackboard) `wp2States`(struct)	blackboard object auditory frontend parameters
`IdentityKS.m`	`bb`(Blackboard) `modelname`(char)	blackboard object name of the linear SVM model to be used
`LocationKS.m`	`bb`(Blackboard) `gmName`(char) `dimFeatures`(int) `angles`(int)	blackboard object name of the graphical model to be used dimensionality of the feature vectors vector of possible angular positions
`ConfusionKS.m`	`bb`(Blackboard)	blackboard object
`ConfusionSolvingKS.m`	`bb`(Blackboard)	blackboard object
`RotationKS.m`	`bb`(Blackboard) `simParams`(struct)	blackboard object struct containing the simulation parameters
`BlackboardMonitor.m`	`bb`(Blackboard)	blackboard object
`Scheduler.m`	`bm`(BlackboardMonitor)	BlackboardMonitor object

Table 6.2: List of available blackboard classes in the MATLAB framework.

Acronyms

ACF auto-correlation function

ASA auditory scene analysis

BEFT Bochum Experimental Feedback Testbed

BM basilar membrane

CCF cross-correlation function

CASA computational auditory scene analysis

ERB equivalent rectangular bandwidth

GM graphical model

HRIR head-related impulse response

ILD interaural level difference

ITD interaural time difference

IC interaural coherence

IHC inner hair cell

KS knowledge source

RMS root mean square

SSR SoundScape Renderer

WP1 work package one

WP2 work package two

WP3 work package three

WP4 work package four

Bibliography

Bernstein, L. R., van de Par, S., and Trahiotis, C. (**1999**), "The normalized interaural correlation: Accounting for N_oS_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)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (**1994**), *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley. (Cited on page 26)

Geier, M., Ahrens, J., and Spors, S. (**2008**), "The SoundScape Renderer: A Unified Spatial Audio Reproduction Framework for Arbitrary Rendering Methods," in *124th AES Conv.* (Cited on page 30)

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)

Walther, T. and Cohen-L'hyver, B. (**2014**), "Multimodal feedback in auditory-based active scene exploration," Submitted for publication. (Cited on page 30)

Wierstorf, H., Geier, M., Raake, A., and Spors, S. (**2011**), "A Free Database of Head-Related Impulse Response Measurements in the Horizontal Plane with Multiple Distances," in *130th Convention of the Audio Engineering Society.* (Cited on pages 30 and 31)