A Biologically Based Framework for Distributed Sensory Fusion and Data Processing

Ferro M. and Pioggia G.
Interdepartmental Research Center "E. Piaggio"
Faculty of Engineering, University of Pisa,
Italy

1. Introduction

The increasing complexity of the artificial implementations of biological systems poses issues in sensory feature extraction and fusion, drift compensation and pattern recognition, especially when high reliability is required [1, 2, 3, 4]. In particular, in order to achieve effective results, the pattern recognition system must be carefully designed. At present, these instruments often fail to give the expected results and research is under development. This happens for a series of concomitant causes, ranging from the measurements, to the limits relevant to instability and non-reproducibility of most existing sensors, up to the inappropriate use of the pattern recognition scheme, i.e. the perception of an odour/taste and its classification through the comparison with similar stimuli perceived in the past. Many techniques are used for this purpose, but recently, the processing architectures are often performed by models inspired by biology, such as genetic algorithms and Artificial Neural Networks (ANN) [4, 5, 6]. Enhancing the reliability of high-level processing systems represents the next critical step. Such architectures require high-efficiency interconnection and co-operation of several heterogeneous modules, i.e. control, data acquisition, data filtering, feature selection and pattern analysis. Heterogeneous techniques derived from chemometrics, neural networks, fuzzy-rules used to implement such tasks may introduce module interconnection and cooperation issues [7, 8]. It may not be reliable to establish a multi-channel communication among common artificial neural networks tools, feature extraction and selection processes, and acquisition and control systems. Moreover, high level interfaces often do not allow adapting of the architecture and/or the processes topology at run-time. As a result complex processing methods have to be designed. A real-time approach for data analysis requires the realization of interconnected modules which are capable to establish an efficient communication channel. In this way the application should be able to control all modules of the elaboration chain, including analysis protocol management and sensory and actuating interfaces.
The body is felt as a unity, with different qualities at different times and the brain mechanism that underlies the experience also comprises a unified system that acts as a whole and produces a neurosignature pattern of a whole body [9]. A distributed processing throughout many areas of the brain, comprises a widespread network of neurons that generates patterns, processes information that flows through it, and ultimately produces the

Source: Sensor and Data Fusion, Book edited by: Dr. ir. Nada Milisavljević,
ISBN 978-3-902613-52-3, pp. 490, February 2009, I-Tech, Vienna, Austria

pattern that is felt as a whole body possessing a sense of self. The stream of neurosignature output with constantly varying patterns riding on the main signature pattern produces the feelings of the body-self with constantly changing perceptual and emotional qualities. The new breakthroughs made in the past few decades in material science in order to develop intelligent sensing materials built in compliance, non-linearity and softness allow to mimic the multi-component and bi-phasic nature of biological matter. Moreover, intelligent algorithms allow the transduction signals to be effectively reconstructed. In order to provide an artificial neural network architecture with the capabilities of processing, coding and fusing in real-time the distributed information continuously flowing from an artificial distributed sensing network, in this chapter a mammalian cortex inspired model is described.

According to the biological sensory systems, where environmental stimuli are deconstructed and then reconstructed in the brain to create perceptions [10], the presented architecture may help dealing with a dynamic and efficient management of multi-transducer data processing techniques, as well as serving as an initial step in the reconstruction of a fused image from its deconstructed features. The raw signals obtained from artificial implementations of biological systems can be preprocessed in order to extract relevant features. Features vectors constitute the dataset for sensory fusion and the pattern recognition processes. Fusion and processing are achieved by the homogeneous software frameworkwhere, in order to gain short-term priming in co-operation with other modules, artificial neural models and architectures inspired to the mammalian cortex [11, 12, 13, 14] are implemented. Artificial neuron models with high computational efficiency and biological accuracy are adopted to obtain a learning strategy able to avoid catastrophic interference [15, 16, 17] and to enable a selection of neuronal groups [18]. To take into account this theory the time variable in the learning task is used, so that neural groups may raise from a selection process.

The framework is able to manage at the same time transducer devices and data processing. Synchronization among modules and data flow is managed by the framework offering remarkable advantages in simulation of heterogeneous complex dynamic processes. Specific control processes, pattern recognition algorithms, sensory and actuating interfaces may be created inheriting from the framework base structures. The architecture is library-oriented rather than application-oriented and starting from the base models available in the framework core dedicated models for processes, maps and connections may be derived. Such a strategy permits the realisation of a user-defined environment able to automate the elaboration of cooperating processes. Etherogenous processes will be able to communicate each other inside the framework as specified by the user. The framework architecture has been designed as a hierarchical structure whose root is a manager module. It is realised as a high-level container of generic modules and it represents the environment in which process modules and I/O filtering interfaces are placed. Modules may be grouped recursively in order to share common properties and functionalities of entity modules belonging to the same type. Communication channels are realised as connections through specific projection types that specify the connection topology. Connections are delegated to dispatch synchronization information and user-defined data. The filtering interface modules are able to drive the transducer hardware and to dispatch information to process modules. All base modules manage dynamic structures and are designed to maintain data consistency while the environment state may change. High level processes such as control processes and pattern recognition algorithms are defined as application processes inside the framework. Such processes inherit properties and functionalities from the framework base structures, taking advantage of automation capabilities provided by the framework core. The framework allows to create a communication language between the framework core and the hardware architecture. This guarantees an increased flexibility thanks to the presence of interfaces performing the function of interpreters for the specific hardware and filters which specify the way the framework core senses and communicates the information.

2. A framework solution for high complex tasks

From a general point of view, a complex system consists of different modules cooperating in order to perform data acquisition from multiple sensors, data analysis through several techniques and data redirection to the actuator systems. The architecture here proposed addresses three main issues:

- acquisition from sensors: a protocol interface will be available to dispatch data coming from input systems; for each hardware sensory system the user will realize the software driver to filter the signals and to dispatch data to the framework core via the framework I/O interface.
- data processing: inside the framework core all the processes will be specified by the user; for each process the user will specify the algorithms, the connection topology between other processes and, optionally, the geometrical structure.
- actuator driving: a protocol interface will be available to dispatch data from the framework core; for each hardware actuating system the user will realize the software driver to filter the data and to dispatch the signals from the framework I/O interface to the actuating systems.

The design of a versatile instrument for data management and elaboration should be suitable for those systems which are equipped with distributed transducer devices, where a particular attention should be paid to inter-process communication. Applications of such instrument space from the simple elaboration of signals supplied by sensory and actuating networks, to pattern analysis and recognition techniques. Design specification included the ability to let the system to be able to operate in real-time. The realization of a framework able to perform a parallel device management should give to all the modules the ability to cooperate and the possibility to share data coming from sensory systems and directed to the actuating systems. The possibility to operate in real time imposes critical efficiency requirements to each single module. The framework design pays attention to the management and the synchronization of data and processes. Control modules and pattern recognition algorithms are defined as application processes inside the framework. The framework is realised as a software library in order to exploit the potentials of the computational algorithms and to enhance the performances of the processing techniques based on artificial neural networks. The architecture is able to manage at the same time transducer devices and data processing. Synchronization among modules and data flow is managed by the framework offering remarkable advantages in simulation of heterogeneous complex dynamic processes. Specific control processes, pattern recognition algorithms, sensory and actuating interfaces may be created inheriting from the framework base structures.

In order to exploit the potentials of the computational algorithms and to enhance the performances of control processing techniques, the framework is realised as a C++ software library. In this way the library architecture is a re-programmable instrument available to the user to develop specific applications. It has been designed to be portable to any software platform in order to gain abstraction from the operating system. The framework however needs a low-level software layer to perform kernel re-building and low-level system calls. An Intel-based personal computer is actually being used and a commercial operative system grants the low-level communication.

2.1 I/O device communication

The framework core and application processes are interfaced with the outside world through the framework I/O interface. This layer has been developed in order to act as a buffer for the flow of information coming in from the sensors and out to the actuators. With this strategy sensory fusion is gained enabling an abstraction with respect to the specific technology of the transducers used. Signals coming from the sensors are gathered in parallel and are encoded according to a standard protocol. The encoded information is received by a specific filter for each sensor, which then sorts them to framework I/O interface. For each actuating system a mirror image architecture has been reproduced with respect to the one described for the sensors. The information available in the framework I/O interface is encoded by a filter using the same standard protocol. A specific interface for each actuator pilots its specific hardware system. This architecture allows setting up a communication language between the framework core and sensory and actuating devices. This guarantees an increased flexibility thanks to the presence of interfaces performing the function of interpreters for the specific hardware and filters which specify the way the framework core senses and communicates the information. Fig. 2.1 shows the flow of information to and from the framework core. Communication channels are established as connections between application processes so that framework is able to perform a low-level inter-process communication. The domain of data flowing through connections and the flow chart of the application processes are user-defined.

2.2 Parallel distributed processing

Synchronization among modules and data flow is managed by the framework offering remarkable advantages in simulation of heterogeneous complex dynamic processes. Specific control processes, pattern recognition algorithms, sensory and actuating interfaces may be created inheriting from the framework base structures, taking advantage of process automation provided by the framework core. A spatial definition of the entities involved in the framework can be supplied, making this information available to the control system for subsequent processing. To guarantee the execution of real-time applications an inner synchronization signal is provided from the framework core to the processes and to the framework I/O interface, enabling to gain time-space correlation. A dynamic geometrical representation can be visualised by a high efficiency 3D graphic interface, giving a support during experimental setup debug. Processes and connections are managed at run time and they can be manipulated under request. The presence of dynamic structures implies a configurable resource management, so the framework offers an optimised interface for enumeration and direct access requests.

2.3 Control and processing modules

The framework architecture has been designed as a hierarchical structure whose root is a manager module (*3DWorld*). It is realised as an high-level container of generic modules

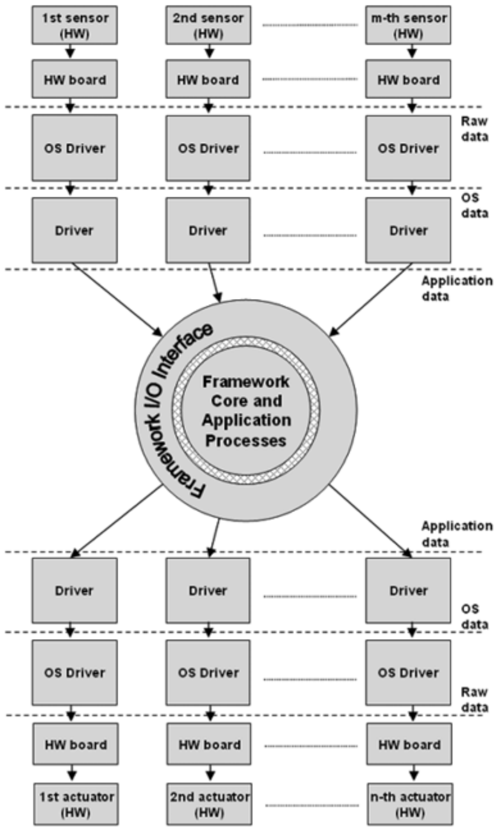


Fig. 2.1. Architecture of the framework for the parallel management of multiple elaboration processes. The transducer devices are synchronized and controlled through an appropriate I/O interface

representing the environment in which process modules (*W*) and I/O filtering interfaces are placed (*Drivers*). All these modules inherit low-level properties and functionalities from a base module (*3DObject*) realised as an element able to populate the process environment. Virtual and pure-virtual functionality strategies have been applied to this base module in order to obtain an abstraction with respect to the generic application task. In this way the core is able to process user-defined functionalities without being reprogrammed. Moreover, modules may be grouped recursively (*WGroup*) in order to share common properties and functionalities of entity modules belonging to the same type. Communication channels are realised as connections (*WConnection*) through specific projection types that specify the connection topology. Connections are delegated to dispatch synchronization information and user-defined data (*WConnectionSpec*). The filtering interface modules are able to drive the transducer hardware and to dispatch information to process modules. All base modules manage dynamic structures and they are designed to maintain data consistency while the environment state may change. This behaviour permits the execution of dynamic and real-time parallel distributed processing while synchronization and data flowing are managed by the environment. All modules are realized as running processes while their control and synchronization is managed by the framework. The hierarchical and collaboration chart of the base structures is shown in Fig. 2.2.

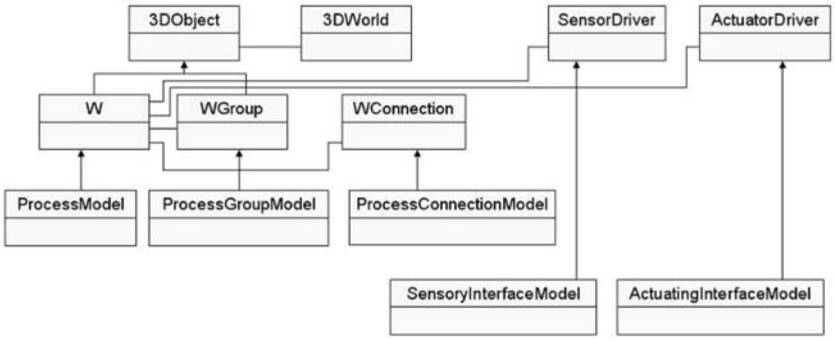


Fig. 2.2. Inheritance and collaboration diagram of the main modules of the framework core: I/O interfaces, communication channels, processes

3. Framework overview

Architecture implementation details will be showed in this section making use of the Unified Modelling Language (UML) representation. In Fig. 3.1 the legend is showed.

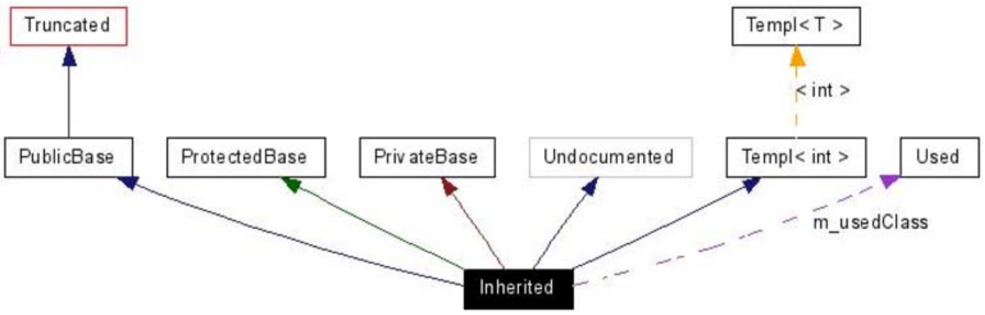


Fig. 3.1. Graph relationship legend

The boxes in the above graph have the following meaning:

- A filled black box represents the struct or class for which the graph is generated.
- A box with a black border denotes a documented struct or class.
- A box with a grey border denotes an undocumented struct or class.
- A box with a red border denotes a documented struct or class for which not all inheritance/containment relations are shown. A graph is truncated if it does not fit within the specified boundaries.

The arrows have the following meaning:

- A dark blue arrow is used to visualize a public inheritance relation between two classes.
- A dark green arrow is used for protected inheritance.
- A dark red arrow is used for private inheritance.

- A purple dashed arrow is used if a class is contained or used by another class. The arrow is labeled with the variable(s) through which the pointed class or struct is accessible.
- A yellow dashed arrow denotes a relation between a template instance and the template class it was instantiated from. The arrow is labeled with the template parameters of the instance.

The implementation will be described using technical object-oriented language terms. To avoid confusion main terms are resumed:

- an *object* is an synonymous word for a *class* or a *structure* and it defines a new data type.
- an *instance* of an object is a variable declared as object type.
- an object's property is called *member*.
- an object's function is called *method*.
- a *derived* object is a child of another object, and it inherits properties and functionalities from his father object, which is called *base* object.
- a *private* member or method is accessible only inside the object.
- a *protected* member or method is accessible both from the object and from a derived object.
- a *public* member or method is always accessible.
- while structures start implicitly with public definitions, classes start implicitly with private definitions.
- a *virtual* method defines a function that, if it is overridden in a derived object, cause the call to derived method even it is called on the base object.
- a *pure-virtual* method is undefined in the base object and, as a result, the base object can not be instantiated.
- an *abstract* object contains only pure-virtual methods.

3.1 Portability

Since the framework library is written using the C++ programming language, the software portability is guaranteed by the standard ANSI-C/C++ definitions. However the low-level interfaces depend on the particular libraries of the operative system. For this reason a low-level layer was defined to include all the dependencies for the specific platform. The low-level layer (*ARI_Macro*) has been realised defining a set of operations for run-time memory management (allocation and deallocation), file I/O interface, log reports and window assert dialogs. All these operations are implemented as C++ macro and they will be used by the framework for all the low-level operative system interfacing.

Graphic User Interface (GUI) is not embedded in the framework in order to not slow down the application efficiency and to let the user to be able to choose his preferred tools. Since each of the framework objects provides functions to get information about the status and the output data, the GUI tools are developed as external modules that can be linked to the architecture. Main graphic output is guaranteed by OpenGL rendering, which libraries are available for many hardware platforms. If the user choose to use OpenGL support, then he must link OpenGL libraries to his application. Application GUI is actually supplied for Microsoft Windows operating system, including useful tools for layered graphs (*MGraph*) and OpenGL dialog windows (*glCDialog*). While data storage is already supplied by the framework, additional end-user tools are available for file tables (*TabData*) and database tables (*TabDataConn*) supporting MySQL, SQLServer and general ODBC drivers. Since all these tools are external they not compromise the architecture efficiency, making the user able to choose his appropriate strategy.

3.2 Containers

Container structures have been developed adopting a template-object strategy. A chunk-memory-allocation strategy has been applied to dynamic containers in order to obtain a configurable compromise between flexibility and direct memory access efficiency. Iterators have been defined for such dynamic structures in order to perform high-efficiency list browsing. Static arrays are able to perform real-time memory reallocation. Basic containers are showed in Fig. 3.2.

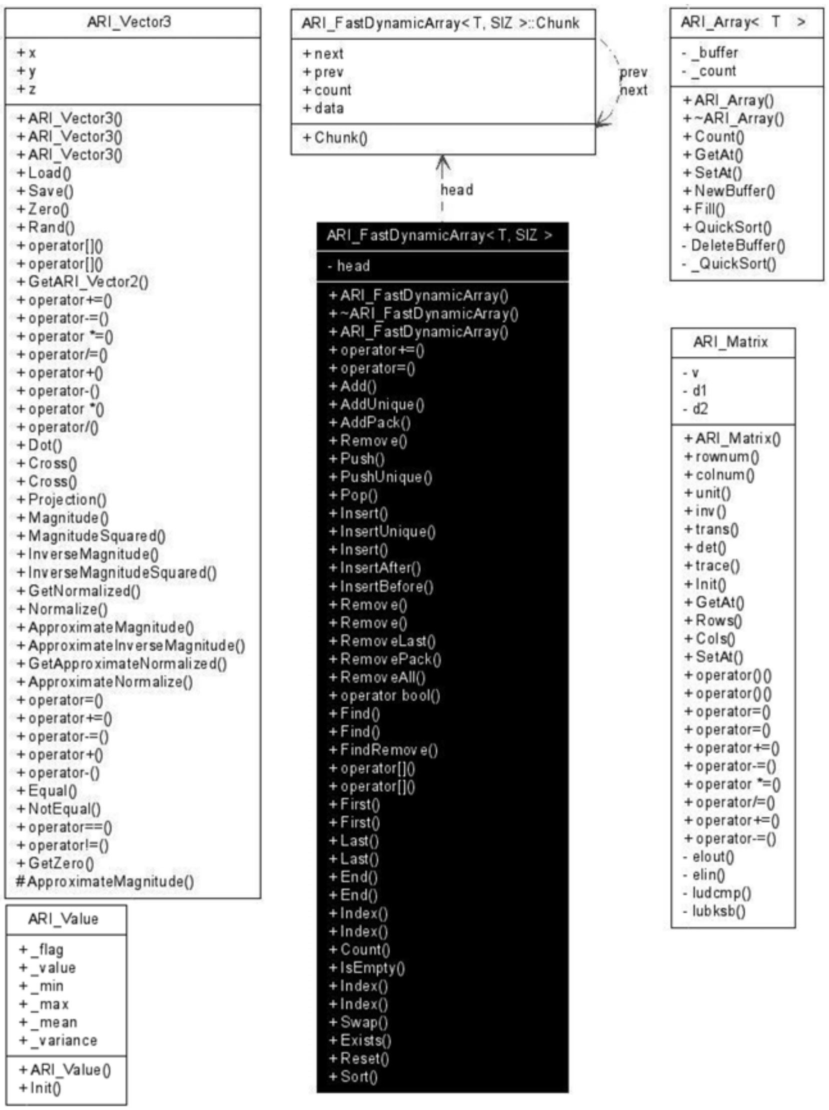


Fig. 3.2. Basic containers implemented in the framework. A chunk-memory allocation stategy has been applied to dynamic containers

3.3 The base structure: 3DObject

The basic element of the framework is called *3DObject*. This is an element able to perform basic functions which are useful to many of the modules included in the framework.

3DObject properties include a label, a 3D position, a 3D radius and a set of user-defined flags. The label will be useful for log reports, application debug and for easy high-level object identification. The default label will be associated with the hexadecimal memory address of the object, which is unique during the execution of the application. Three-dimensional position and radius will be useful for graphic rendering under user request. Such values are initialised with null values, since the rendering functions are optional. Flags specify the way in which the object manager will process the object (see *3DWorld*). By default no flag is set for a *3DObject*.

Several functions has been supplied for this base object in order to guarantee I/O transfers, 3D management and flags maintenance. Since process objects will be derived from *3DObject*, a set of virtual functions has been defined to perform process synchronization and update. The use of virtual functions guarantees the ability of the object manager to call redefined functions on derived objects without knowing them. User-defined processes have to follow the base virtual protocol, redefining the way in which synchronization and update operations are performed. In Fig. 3.3 the *3DObject* architecture is showed.

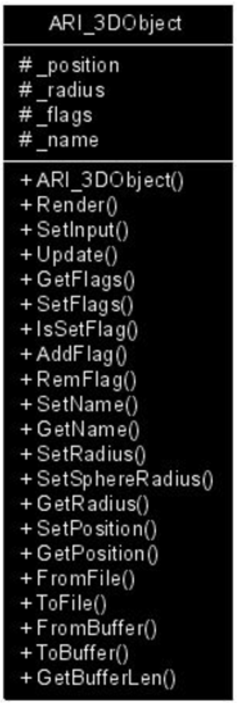


Fig. 3.3. Architecture of the 3DObject structure. This is the base object used to populate the process environment managed by the 3DWorld structure

Defined virtual functions are *Render*, *SetInput* and *Update*. Each of these functions will be called by the object manager only if the appropriate flag is set. Such strategy makes the user

to be able to specify virtual overrides for his specific processes. *Render* function will specify the way in which the object will be graphically rendered. OpenGL is the default renderer, permitting an high efficient three-dimensional output. Graphic rendering is useful for object status monitoring during application execution and debug. The definition of a rendering function does not reduce object efficiency since *Render* virtual function will be called only if screen repainting is needed and if the rendering flag has been set.
SetInput and *Update* functions specify the core process algorithms. The first method is needed to workaround the serial processing of all the objects. In fact the parallel execution of all the process objects will be serialized by the low-level CPU scheduling. The order of the execution of each process may compromise the effective result of the process network in the case of multiple cooperating processes. For this reason the object manager will first call the *SetInput* method over all the processes with the intent of buffering the actual output data over communication channels (see *WConnection*). After this buffering step, the *Update* method will be called over all the objects making use of the buffered connection data instead of the object actual output data. Such a strategy guarantees the independence from the process execution order. Default base method just defines the virtual processing and rendering protocol.
The same virtual strategy has been adopted for buffers and files data storage. A set of virtual functions has been defined to perform an object independent way for file I/O (*ToFile*, *FromFile*) using platform independent file operations. Such methods use buffer virtual functions (*ToBuffer*, *FromBuffer*, *GetBufferLen*) which will be specified by the user for each process. Default implementation of buffer virtual methods just defines the buffering I/O protocol.

3.4 The object manager: 3DWorld

Synchronization and update of all the running processes is managed by *3DWorld*. This structure is a collection of *3DObject*. Elements are organised in a dynamic list where the access order is often sequential. *3DWorld* provides the execution of specific methods for all the objects added during an initialisation step. Internally *3DWorld* manages the reference of each object and not the object itself. This strategy speeds up the enumeration of the elements, giving to the user the opportunity to override the manager behaviour and to gain the direct control of each single element. Elements may be added and removed at run-time (*AddObject*, *RemoveObject*) while the execution of process virtual methods is managed accordingly to the active flags of each elements (*Render*, *SetInput*, *Update*).

3.5 The base process structure: W

The process base structure inherits properties and functionalities from *3DObject*. *W* is a generic transferring function, which is able to communicate with structures of the same base type. Communication channels are established as connections and the domains of input and output data are defined by the user. For this reason a pure-virtual strategy has been applied to this structure to take into account a processing method (*Process*) which is still not defined at this level. For this reason a *W* structure can not be instantiated. It only defines the process standard protocol and provides topology connection methods. *Render*, *SetInput* and *Update* flags are automatically activated in the initialization step. The architecture of *W* structure is shown in Fig. 3.4.

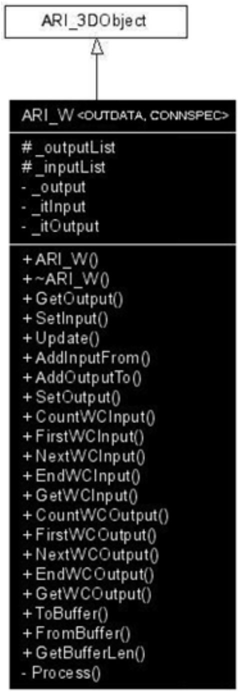


Fig. 3.4. Architecture of the W structure. It represents the base process model

W is a template structure where *OUTDATA* and *CONNSPEC* types are user-defined. While *OUTDATA* represents the output pattern, *CONNSPEC* contains the specification for each input connection. The *3DObject* virtual method *Update* is redefined to set the new output value accordingly to the result of the pure virtual method *Process*, which will be specified in derived process structures. Actual process output can be retrieved using *GetOutput* method. *W* provides methods to specify the connection topology of the single process (*AddInputFrom*, *AddOutputTo*) in respect to the other input and output processes. *SetInput* method is redefined to perform a scan over all the input connections and to transfer the actual output value of input process to the connection buffer (see *WConnection*). For sake of efficiency the starting point for both input and output connections is stored in each *W* structure. Since connections are usually browsed sequentially, enumeration methods are provided to perform high efficient navigation over input and output connections (*First*, *Next*, *End*). In derived structures the *Process* method has to be redefined to perform the core process algorithm, taking into account the specific data connection and the data output given by each input process, which are already buffered into the corresponding input connection. *Process* method is defined as private function, so it can not be directly called by the user since it is automatically managed by the framework during the update step. With this strategy the process, which core is still undefined, is able to manage the unknown information of the user-defined process.

Buffer I/O operation (*GetBufferLen*, *ToBuffer*, *FromBuffer*) are redefined to provide data storage for actual process output and for the actual specification values for all the input connections.

3.6 The connection between two processes: WConnection and WConnectionSpec

A connection between two process is realised by the *Wconnection* template structure. Since input connections for a given process are embedded inside the process itself, the template types are the same used for *W* structure. *WConnection* stores references of both source and destination processes. A field of *OUTDATA* type is stored to perform data buffering. In fact during the *SetInput* step on the destination process, the *GetOutput* method is called over the input process and the resulting value is stored inside the connection. While the *W* structure contains the references of both the first input connection and the first output connection, the next links to connections are stored into two dynamic lists inside the *WConnection* structure. In Fig. 3.5 an example of this strategy is shown.

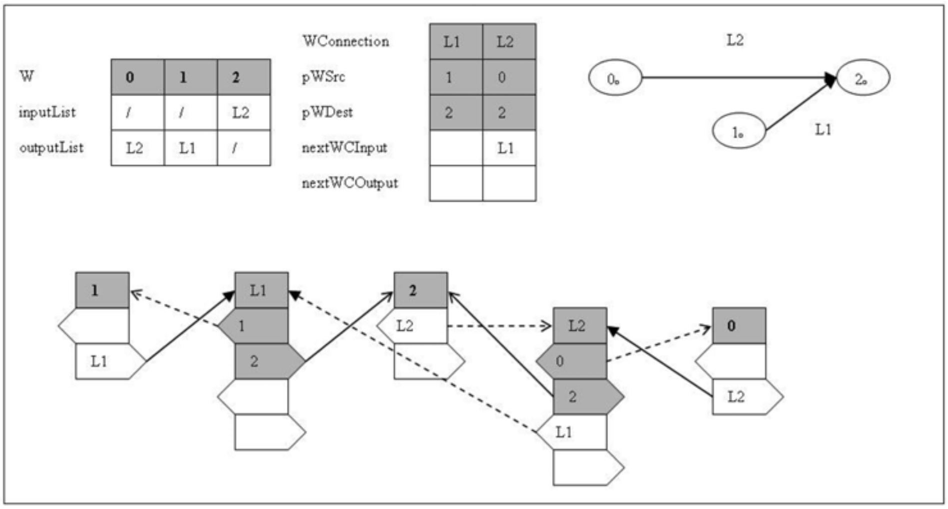


Fig. 3.5. Connection strategy example. Each process contains the references of both the first input connection and the first output connection. Each connection contains the references of both the source and the destination process, plus the references to both the next input connection and the next ouput connection

Inside the *WConnection* structure a field of *CONNSPEC* type is used to store the specific information needed by the destination process to evaluate the output data given by the source process. The base class for the *CONNSPEC* type is the *WConnectionSpec* structure. This structure is an abstract class where only a pure virtual method is defined (*Init*). This method will be automatically called during the creation of the connection to perform the initialisation of the *CONNSPEC* values on the basis of the user-defined parameters. This structure only defines the connection initialisation protocol and must be redefined in the derived structures.

3.7 Grouping processes: WGroup and WProjectionSpec

Processes may be grouped into a *WGroup* structure, which template arguments are the same of the *W* structure (*OUTDATA*, *CONNSPEC*). Since this structure only contains the references to several *W* structure, the user does not lose the control over each single process. Grouping processes results in a logical managing of several objects, which can be added and removed at run-time (*Add*, *Remove* methods). *3DObject* is the base class for the *WGroup* structure, permitting the redefinition of the virtual functions for flag, synchronization and I/O mangement over all the grouped entities. *WGroup* contains its own flags, with the possibility to propagate flags to each grouped entity (*AddFlags*, *RemFlags*, *SetFlag*, *IsSetFlag*). With the same strategy adopted for the *3DWorld* structure, *WGroup* is able to propagate rendering and synchronization signals over grouped entities according to their active flags. By default these flags are activated only for the group and they are not propagated to the sub-entities. Since the *WGroup* structure has its own geometrical position and volume, a *Dispose* function is supplied for geometrically disposing all the grouped entities according to their volumes. Such information will be automatically taken into account during the framework rendering processing.

WGroup contains useful methods to create connections both to other groups and to other single processes. Since these methods (*AddInputFrom*, *AddOutputTo*) may involve more than one entity, connections are realised through projection specifications (*WProjectionSpec*). During the projection initialisation step, the framework looks for projection flags in order to perform the user-specific connection strategy. Currently *one-to-one*, *N-to-one*, *uniform random-to-N*, *sub-group-to-N* and *polar-random-to-N* flags are supported (see *WGroup.h* header file).

Since the entities inside a group are often browsed sequentially, high efficient iterators are defined also for this structure. I/O buffering operations are redefined to automatically join I/O buffering operations of each grouped entity. The *Wgroup* architecture is shown in Fig. 3.6.

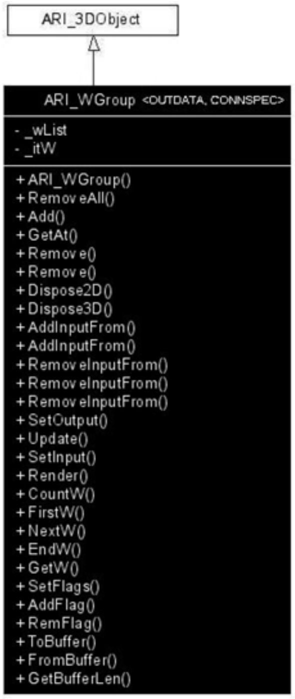


Fig. 3.6. Architecture of theWGroup structure. Many entities may be grouped togheter in order to create an high-level entity able to populate the process environment

3.8 SensorDriver and ActuatorDriver

The *SensorDriver* structure is the starting point for filtering input data coming from the generic input system. This structure directly derives from *3DObject* and uses *IFNeuron* and *IFNeuronGroup* structures. These two structures are specifically designed for artificial neural networks using the integrate-and-fire model, as it will be discussed later. However at this level they are used as buffering structures where the process functions (*Set-Input*, *Update*) are skipped by the framework. This choice, as it will be shown, does not cause lack of efficiency and of generality.

Since sensor information have to be available before then the framework update step, the *SensorDriver* objects have to be added into *3DWorld* by the user before than other process object. During the construction of the structure, only the *Set-Input* flag is set. *SetInput* method will be redefined for the specific hardware in derived structures, where the data filtering algorithm will be specified.

During the initialisation step (*InitDriver*) the user will specify the length of the sensor buffer data needed during the hardware acquisition step. At this point the driver just allocates the memory space in a static array, and it waits for the registration of the *IFNeuron* entities. These entities represent the objects where sensor data will be mapped. Such entities will be specifled using the *Register* method, where the *SetInput* and *Update* flags are automatically removed from each registered entity. *SensorDriver* structure will be operative only when all of the needed mapping entities will be registered by the user. In order to speed up the acquisition process, the references of the *IFNeuron* objects are indexed during the registering phase. Such a strategy permits a direct memory access over all the registered entities. The user may choose to switch on/off the driver using the *SetPowerOn* method. The *SensorDriver* architecture is shown in Fig. 3.7. The *ActuatorDriver* structure follows a similar architecture.

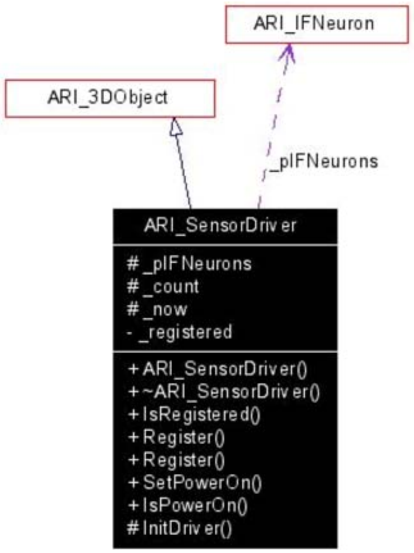


Fig. 3.7. Architecture of the SensorDriver structure. This object represents the base model for the interface of a generic input system

4. Cortical-based artificial neural networks

The concept of artificial neural networks is to imitate the structure and workings of the human brain by means of mathematical models. Three basic qualities of the human brain form the foundations of most neural network models:

- knowledge is distributed over many neurons within the brain;
- neurons can communicate (locally) with one another;
- and the brain is adaptable.

The terminology with which neural networks are described is derived from these three qualities of the human brain, and is as follows:

- structure of the neuron;
- network topology;
- adaptation and learning rule.

The neurons, or processing units, which make up the neural network are single elements and consist principally of four components:

- a connection function;
- an input function;
- an activation or transfer function;
- an output function.

A neuron receives signalsthrough several input connections. These are weighted at the input to a neuron by the connection function. The weights employed here define the coupling strength (synapses) of the respective connections and are established via a learning process, in the course of which they are modified according to given patterns and a learning rule. The input function compresses these weighted inputs into a scalar value, the so-called network activity at this neuron. Simple summation is generally employed here. In such cases, the network activity, which results from the connection function and the input function, is the weighted sum of the input values. The activation function determines a new activation status on the basis of the current network activity, if appropriate taking the previous status of the neuron into account. This new activation status is transmitted to the connecting structure of the network via the output function of the neuron, which is generally a linear function. By way of reference to biological neurons, the activation status at the output of a neuron is also known as the excitation of the neuron.

A process unit is of interest only as a unit of a network consisting primarily of homogeneous elements. In artificial neural nets, these elements are generally interconnected to form a rigid network structure, as a result of which the learning algorithm only rarely includes provision for the formation of new connections and the removal of old connections, such as occurs in biological systems. A layered connecting structure is generally employed, whereby the layer on which the input signals act is referred to as the input layer; the layer at which the results are collected is known as the output layer; and the layers located between these are known as hidden layers. The neurons are generally fully connected on a layer-by-layer basis. The number of layers often determines the performance of a network.

A distinction can be made between feedforward, lateral and feedback connections for the method of linking the different layers. Both feedforward and feedback connections over several layers are conceivable. The connecting structure and the choice of processing units determine the structure of a network. In order to carry out a data fusion and a classification, the network must be taught a task by presenting it with examples in a training phase. The training phase normally proceeds as follows: random values are initially assigned for the weights of the neurons. Patterns from a training data record are then presented to the network and the weightings are adapted on the basis of the learning rule and training pattern until a convergence criterion, e.g. a defined error threshold, is attained. A test phase is then carried out, in which unknown test patterns are presented to the network to establish the extent to which the network has learnt the task in hand. Selection of the patterns for the training phase is a particularly important aspect. These patterns must describe the task as completely as possible, as in later use the network will only be able to provide good results for problems which it has learnt. This means that patterns must be selected which cover all classes and, where possible, describe the boundary ranges between the classes.

Most of these architectures are not able to proceed in new learning processes without loosing memory of the past learning processes (catastrophic interference) [16, 17]. In order to overcome these issues models able to gain short-term priming in co-operation with other modules have been developed. In particular, hippocampus-based models operate a pattern separation avoiding the catastrophic interference [11, 12]. Input patterns are spread among different interconnected modules following the McCloskey and Cohen model [11] consisting of several interconnected two-dimensional self-organising maps of artificial neurons (Fig. 4.1). The Input Entorhinal Cortex and the Output Entorhinal Cortex maps represent respectively the input and the output of the net. Input and output maps have the same dimension in order to evaluate the activation and deactivation error by a one-to-one comparison of neuronal activity. Activation error represents the percentage of neurons that are firing in the Output Entorhinal Cortex and that are under threshold in the Input Entorhinal Cortex. Deactivation error represents the percentage of neurons that are under threshold in the Output Entorhinal Cortex and that are firing in the Input Entorhinal Cortex.

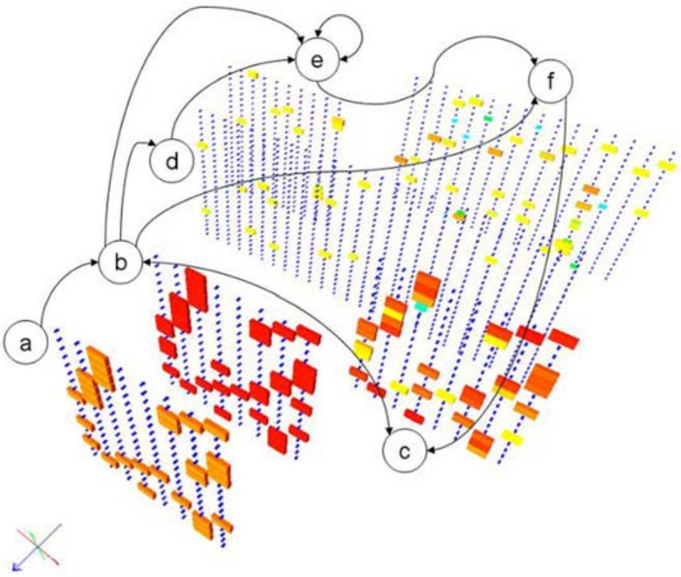


Fig. 4.1. Hippocampus model proposed by McCloskey and Cohen; a) Input pattern; b) Input Entorhinal Cortex; c) Output Entorhinal Cortex; d)Dentate Gyrus; e) CA3; f) CA1

4.1 Learning strategy: selection of neuronal groups

The Theory of Neuronal Group Selection (TNGS) proposed by Edelman [18, 19], suggests a novel way for understanding and simulating neural networks. To take into account this theory we have to use the *time* variable in the learning task, so that neural groups may raise from a selection process. This strategy has been adopted by Izhikevich, who simulated a minimal neural network which is able to show the property of *polychronization* [20]. In such a network a correspondence between synaptic weights and axonal delays exists as a result of the neuron bahaviour. One neuron can belong to many groups, which count is usually higher than the count of the neurons theirself. This guarantees a memory capability which is higher than the capability reached by the classical neuronal network. Such an architecture has been implemented into the framework here presented, giving the possibility to connect the neuronal groups to sensory and actuating systems. The advantage of the use such an approach makes it possible to gain time-space correlation on input signals.
The classical approach in artificial neural networks simulation takes into account the modulation of the action potential rithm as the only parameter for the information flowing to and from each neuron. Such a strategy seems to be in contrast with novel experimental results, since neurons are able to generate action potential which are besed on the input spike timings, with a precision till to one millisecond. The spike-timing synchrony is a natural effect that permits a neuron to be activated in correspondence of synchrounous input spikes, while the neuronal activation of the post-synaptic neuron is negligible if pre-synaptic spikes arrives asynchronously to the target neuron. Axonal delays usually lie in the range [0.1 , 44] milliseconds, depending on the type and location of the neuron inside the network. Such a property becomes an important feature for the selection of the neural groups as it is exposed by Edelman. In the artificial neural network model, the synaptic connection are modified according to the STDP rule. If a spike coming from an excitatory pre-synaptic neuron causes the fire of the post-synaptic neuron, the synaptic connection if reinforced since it given the possibility to generate an other spike in order to propagate the signal. Otherwise the synaptic connection is weakened. The values of the STDP parameters are choosen in order to permit a weakening that is grater than the reinforcement. Such a strategy permit the progressive removal of the unnecessary connections and the persistance of the connections between correlated neurons.

5. Implementation of artificial neurons: towards real-time data fusion and processing

The complexity of a biological neuron may be reduced by using several mathematical models. Each of these reproduce some of the functionalities of real neurons, such as the excitability in response to a specific input signal. The most accurate model for a biological neuron has been developed by Hodgkin and Huxley [13] and it is able to exactly reproduce the shape of the action potential of a neuron by taking into account the ionic currents. Beside of this, the model is computationally expensive and it takes about 1200 FLOPs (FLoating Point Operations) to simulate one millisecond of a single neuron activity. Several attempts have been made in order to reduce the mathematical complexity of this model. The most effective result has been obtained by the Morris-Lecar model [13], which is able to describe the oscillations of the muscular fibers of the giant squid and it is still close to the Hodgkin-Huxley model accuracy. Unfortunately the computational complexity is still high, since it takes about 600 FLOPs for one millisecond of neuron activity. Since these bottom-up

approaches are focused on the characterization of the biophysic properties of the cell membrane, a different approach has been adopted by Fitzhugh and Nagumo [13], taking into account the information of the nervous signal as a temporal distribution rather than an action potential shape. This top-down approach leads to the development of parametric differential equations with the aim to match them with experimental results. The Fitzhugh-Nagumo model, wich takes about 72 FLOPS for one millisecond of neuron activity, is based on a variant of the Van Der Pol oscillator. Studies on the dynamics of non-linear systems swoed a large variety of behaviours. Actually, the use of mathematical analogies seems to be the only way to simulate a large number of interconnected artificial neurons.
For this reason the integrate-and-fire model (and its variant models) is the simpler and most used model for classification and prediction tasks in pratical scenarios.

5.1 The integrate-and-fire model

The integrate-and-fire model is the simplest model of a spiking neuron that takes into account the dynamics of the input. The basis of the integrate-and-fire model is the simple compartmental model of a neuron. The equivalent electric schema is showed in Fig. 5.1.

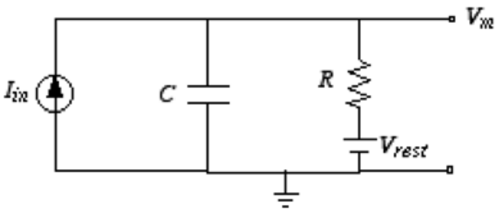


Fig. 5.1. The integrate-and-fire artificial neuron model: equivalent electric schema

The computational implementation of the integrate and fire model follows the schema showed in Fig. 5.2.

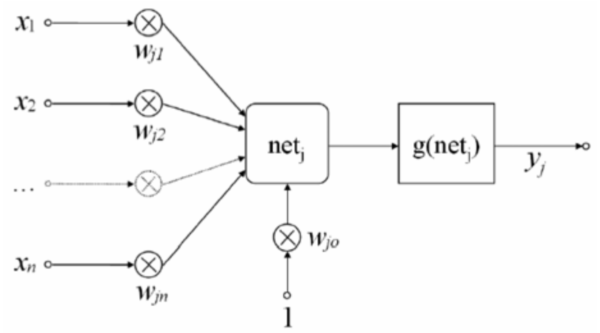


Fig. 5.2. The integrate-and-fire artificial neuron model: computational schema

An *IFNeuron* structure has been implement in the framework as a running process directly deriving from the *W* structure. Template arguments have been specialised to obtain an *OUTDATA* as a real number (double precision floating point value) and a *CONNSPEC* as a *IFNeuronConnectionSpec* structure, which is shown in Fig. 5.3.
The connection structure for such a process uses a real number to manage the synaptic weight. The value may be initialised by the user or randomly chosen by the framework

according to the value initialisation parameters. A weight buffer value is needed for internal operations during supervised learning using the multi layer perceptron process, which will be discussed later. The *IFNeuron* structure defines the private virtual method *Process* in order to perform the weighted sum of signal coming from input connections. The result value is then filtered using the sigmoid function according to the integrate and fire model. The structure internally saves a value to speed up the delta-rule algorithm adopted during supervised learning. The I/O buffering operations simply manage internal members and recall the base class methods. The rendering function provides the graphic visualisation of the soma and of the input connections. The architecture is shown in Fig. 5.4.

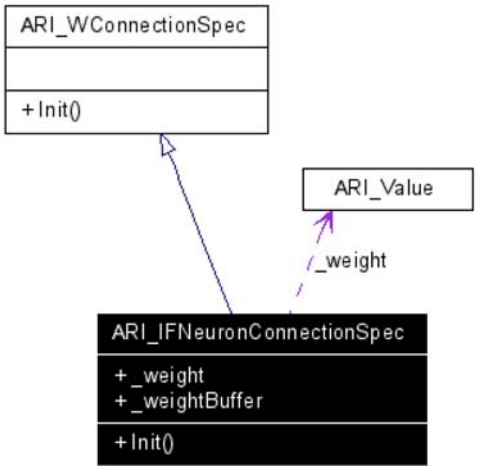


Fig. 5.3. The IFNeuronConnectionSpec structure

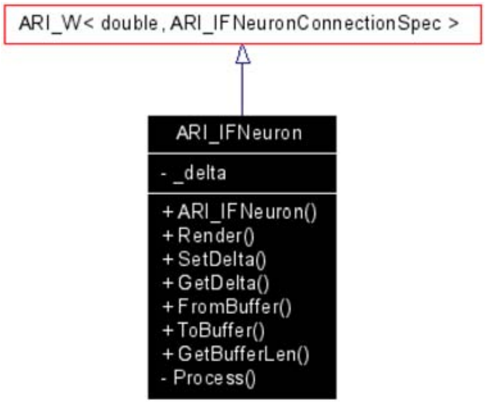


Fig. 5.4. Architecture of IFNeuron structure

The *IFNeuronGroup* structure, which represents a group of *IFNeurons*, has been derived from the *WGroup* base structure. The *IFNeuronGroup* structure will be used by high-level processes in order to perform supervised and unsupervised learning tasks based on the integrate and fire neuron model.

5.2 The leabra neuron model

The Leabra base model [11, 12] is a simplified version of the Hodgkin-Huxley model. Both models are shown in Table. 1.

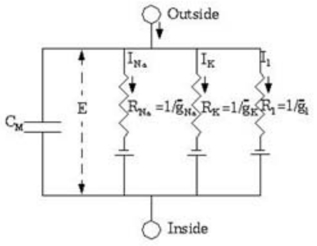
	Equations
Hodgkin-Huxley	$I = C_m \cdot \frac{\partial V}{\partial t} + \bar{g}_K \cdot n^4 \cdot (V - V_K) + \bar{g}_{Na} \cdot m^3 \cdot h \cdot (V - V_{Na}) + \bar{g}_l \cdot (V - V_l),\ V(0) = V_0$ $\frac{dn}{dt} = \phi \cdot (\alpha_n \cdot (1-n) - \beta_n), n(0) = n_0$ $\frac{dm}{dt} = \phi \cdot (\alpha_m \cdot (1-m) - \beta_m), m(0) = m_0$ $\frac{dh}{dt} = \phi \cdot (\alpha_h \cdot (1-h) - \beta_h), h(0) = h_0$ $\phi = 3^{(T-6.3)/10}$  Outside I_{Na} I_K I_l C_M E $R_{Na} = 1/\bar{g}_{Na}$ $R_K = 1/\bar{g}_K$ $R_l = 1/\bar{g}_l$ Inside
Leabra Model	a) $g_e(t) = (1 - dt_{net}) g_e(t-1) + dt_{net} \left(\frac{1}{n_p} \sum_k s_k \frac{r_k}{\sum_p r_p} \frac{1}{\alpha_k} < x_i w_{ij} >_k + \frac{\beta}{N} \right)$ b) $g_i^{\Theta} = \frac{g_e^* \bar{g}_e (E_e - \Theta) + g_l \bar{g}_l (E_l - \Theta)}{\Theta - E_i}$ $g_i(t) = g_i^{\Theta}[k+1] + q(g_i^{\Theta}[k] - g_i^{\Theta}[k+1])$ c) $V_m(t+1) = V_m(t) + dt_{vm}[g_e(t)\bar{g}_e(E_e - V_m(t)) + g_i(t)\bar{g}_i(E_i - V_m(t)) + g_l(t)\bar{g}_l(E_l - V_m(t))]$ $y_j = \frac{\gamma[V_m - \Theta]_+}{\gamma[V_m - \Theta]_+ + 1}$ $\quad y_j^*(x) = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{z^2}{2\sigma}} y_j(z-x)\,dz$

Table 1. Top: Hodgkin and Huxley neuron model, based on chemical species. Bottom: Leabra model; a) Excitatory conductance; b) Inhibitory k-WTA function; c) Membrane potential; d) Activation function

Leabra uses a point neuron activation function that models the electrophysiological properties of real neurons, while simplifying their geometry to a single point. This function is nearly as simple computationally as the standard sigmoid activation function, but the more biologically-based implementation makes it considerably easier to model inhibitory competition, as described below. Further, usingthis function enables cognitive models to be more easily related to more physiologically detailed simulations, thereby facilitating bridge-building between biology and cognition.

Leabra uses a kWTA (k-Winners-Take-All) function to achieve inhibitory competition among units within a layer (area). The kWTA function computes a uniform level of inhibitory current for all units in the layer, such that the k+1th most excited unit within a layer is generally below its firing threshold, while the k-th is typically above threshold. Activation dynamics similar to those produced by the kWTA function have been shown to result from simulated inhibitory interneurons that project both feedforward and feedback inhibition. Thus, although the kWTA function is somewhat biologically implausible in its implementation (e.g., requiring global information about activation states and using sorting mechanisms), it provides a computationally effective approximation to biologically plausible inhibitory dynamics. For learning, Leabra uses a combination of error-driven and Hebbian learning. Implementation diagrams are shown in Table 2.

Connection	Neuron	Neuron Group

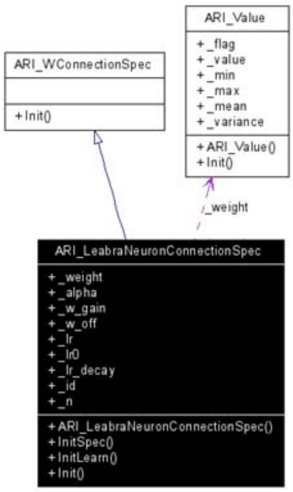
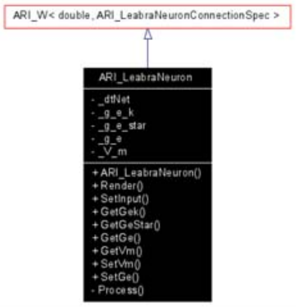
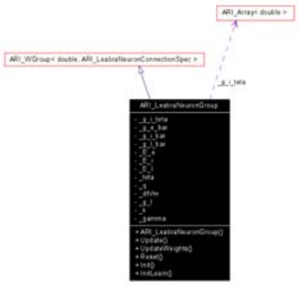
 ARI_WConnectionSpec
+Init()
ARI_Value
+_flag
+_value
+_min
+_max
+_mean
+_variance
+ARI_Value()
+Init()
_weight
ARI_LeabraNeuronConnectionSpec
+_weight
+_alpha
+_w_gain
+_w_off
+_lr
+_lr0
+_lr_decay
+_id
+_n
+ARI_LeabraNeuronConnectionSpec()
+InitSpec()
+InitLearn()
+Init()
	 ARI_W< double, ARI_LeabraNeuronConnectionSpec >
	ARI_LeabraNeuron
	-_dtNet
	-_g_e_k
	-_g_e_star
	-_g_e
	-_V_m
	+ARI_LeabraNeuron()
	+Render()
	+SetInput()
	+GetGek()
	+GetGeStar()
	+GetGe()
	+GetVm()
	+SetVm()
	+SetGe()
	-Process()
		 ARI_Array< double >
		ARI_WGroup< double, ARI_LeabraNeuronConnectionSpec >
		_g_i_teta
		ARI_LeabraNeuronGroup
		-_g_i_teta
		-_g_e_bar
		-_g_i_bar
		-_g_l_bar
		-_E_e
		-_E_i
		-_E_l
		-_teta
		-_s
		-_dtVm
		-_g_l
		-_k
		-_gamma
		+ARI_LeabraNeuronGroup()
		+Update()
		+UpdateWeights()
		+Reset()
		+Init()
		+InitLearn()

Table 2. Leabra model: inheritance and collaboration diagrams of structures for leabra model implementation

5.3 The Izhikevich artificial neuron

Izhikevich recently developed a simple model for an artificial neuron wich is able to reproduce all the behaviours showed above [13]. The model takes 13 FLOPs for simulate one millisecond of neuron activity and it is based on a top-down approach, using two differential equation with four parameters. The introduction of axonal delays shows the possibility to create a neural network able to perform classification and prediction tasks. The connection of several maps follows and the Spike-Timing-Dependant Plasticity (STDP) rule, which permits the implementation of a real time learning rule based on signals which continuously flow from input systems. This architecture follows the theories of Edelman about the selection as the basis for the learning process.

The model proposed by Izhikevich for the artificial neuron simulation shows the ability to reproduce the same accuracy of the Hodgkin and Huxley model. It can be resumed in the following relations:

$$\begin{cases} v' = 0.04v^2 + 5v + 140 - u + I \\ u' = a(bv - u) \end{cases}$$

A reset condition is needed:

$$\text{if } v \geq +30mV, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases}$$

The four parameters (*a*, *b*, *c* and *d*) are dimensionless values. The v variable represents the membrane potential of the neuron, while *u* keeps into account the activation of K^+ ionic currents and the deactivation of the Na^+ ionic currents. The *I* variable takes into account the synaptic currents and the bias currents as the input signal of the neuron. Depending on the values of the four parameters, the system may have a steady-state (which corresponds to a lack of activity in the neuron) and an unsteady-state (which corresponds to the presence of activity in the neuron). The reset condition is needed to perform the return of the system into the steady state after the neuron has fired. Table 3 shows the values of the four parameters in order to obtain the known neuron behaviours.
In order to implement a network able to use the polychronization feature as it is described above, a software module has been realised. An *IzhikevichNeuron* structure (see Fig. 5.5) has been implement in the framework as a running process directly deriving from the *W* structure.

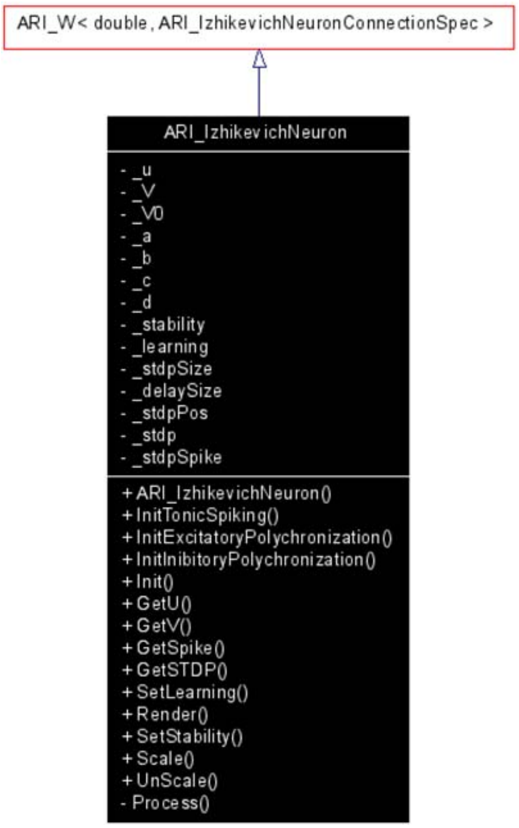


Fig. 5.5. The IzhikevichNeuron structure

Template arguments have been specialised to obtain an *OUTDATA* as a real number (double precision floating point value) which represents the membrane potential of the neuron, and a *CONNSPEC* as a *IzhikevichNeuronConnectionSpec* structure, which is shown in Fig. 5.6.

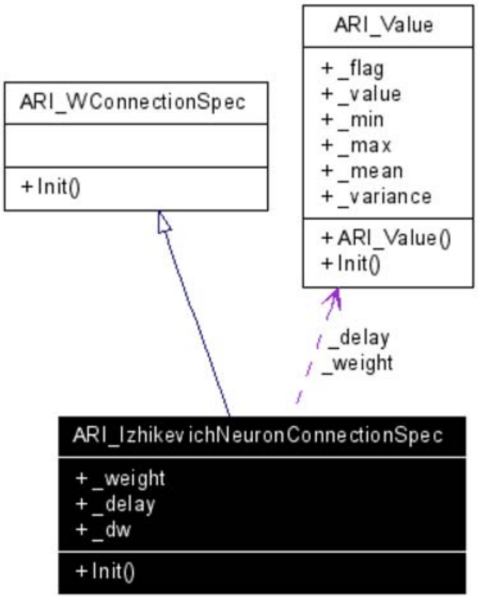


Fig. 5.6. The IzhikevichNeuronConnectionSpec structure

The connection structure for such a process uses real numbers to manage the synaptic weight and the synaptic channel delay. The values may be initialised by the user or randomly chosen by the framework according to the value initialisation parameters. A delta-weight value is needed for internal operations during the learning process, as it will be discussed later.

The *IzhikevichNeuron* structure is initialised using the *Init* method in order to setup the internal parameters (*a*, *b*, *c*, *d*) which specify the behaviour of the artificial neuron. Several initialisation wrapper methods are provided to use predefined behaviours as they are showed in Table 3. The STDP algorithm (Fig. **??**) is implemented with a time-window of size equal to 1000 milliseconds. During this period the delta-weight values are updated according to the STDP rule, while weights are updated at the end of each period. During each period the structure traces the firing activity and the STDP status of the neuron, storing the information in two static arrays. The structure defines the private virtual method *Process* in order to perform the learning task. If the neuron is firning, the Process method reset the internal status (*u*, *v*) and the STDP value is reported to a value equal to 0.1. Otherwise the STDP value is decreased with a time-constant equal to 20 milliseconds. Subsequently the input connections are browsed to update input current, whose contribute depends on the thalamic input neurons and on those neurons who fired with a timing equal to the connection delay. According to the STDP rule, the STDP value of the post-synaptic neuron is increased if it fired synchronously with the the pre-synaptic neuorn, and it is decreased if the pre-synaptic fire caused no firing in the post-synaptic neuron. Finally the status is updated following the Izhikevich model, and, if 1000 milliseconds are enlapsed, the synaptic weights of the connections coming from the excitatory neurons are updated with the actual delta-weigth values. During this step the weights are clamped within a convenient range and the delta-weight values are decreased with using a decay coefficient equal to 0.9.

	a	**b**	**c**	**d**
A) Tonic Spiking	0.02	0.2	-65	6
B) Phasic Spiking	0.02	0.25	-65	6
C) Tonic Bursting	0.02	0.2	-50	2
D) Phasic Bursting	0.02	0.25	-55	0.05
E) Mixed Mode	0.02	0.2	-55	4
F) Spike frequency adaption	0.01	0.2	-65	8
G) Class 1 excitable	0.02	-0.1	-55	6
H) Class 2 excitable	0.02	0.26	-65	0
I) Spike latency	0.02	0.2	-65	6
J) Subthreshold oscillation	0.05	0.26	-60	0
K) Resonator	0.1	0.26	-60	-1
L) Integrator	0.02	-0.1	-55	6
M) Rebound Spike	0.03	0.25	-60	4
N) Rebound burst	0.03	0.25	-52	0
O) Threshold variability	0.03	0.25	-60	4
P) Bistability	0.1	0.26	-60	0
Q) Depolarizing after-potential	1	0.2	-60	-21
R) Accomodation	0.02	1	-55	4
S) Inhibition-induced spiking	-0.02	-1	-60	8
T) Inhibition-induced bursting	-0.026	-1	-45	-2

Table 3. Values of the four dimensionless parameters used to obtain the corresponding neuron behaviour.

The I/O buffering operations simply manage internal members and recall the base class methods. The rendering function provides the graphic visualisation of the soma and of the input connections.

The *IzhikevichNeuronGroup* structure (Fig. 5.7a), which represents a group of *IzhikevichNeurons*, has been derived from the *WGroup* base structure. The *IFNeuronGroup* structure will be used by high-level processes in order to perform the monitoring of the activity of the neurons during the learning and test tasks. Methods are provided to obtain the activation percentage (*GetActPerc*) and to retrieve the sub-group identification relating to a specified input pattern. A specific structure (*ARI_ING_Record*) has been realised to store the neuron reference and the activation time for each neuron belonging to the sub-group. Such records can be enumerated using the the iterator methods (*First*, *End*, *Next*, *Get*).

Moreover, an *IzhikevichMap* structure (Fig. 5.7b) has been derived from *W* base structure in order to speed-up the artificial neural group processing. This structure includes all the previous described structures, optimising the memory usage and computational efficiency.

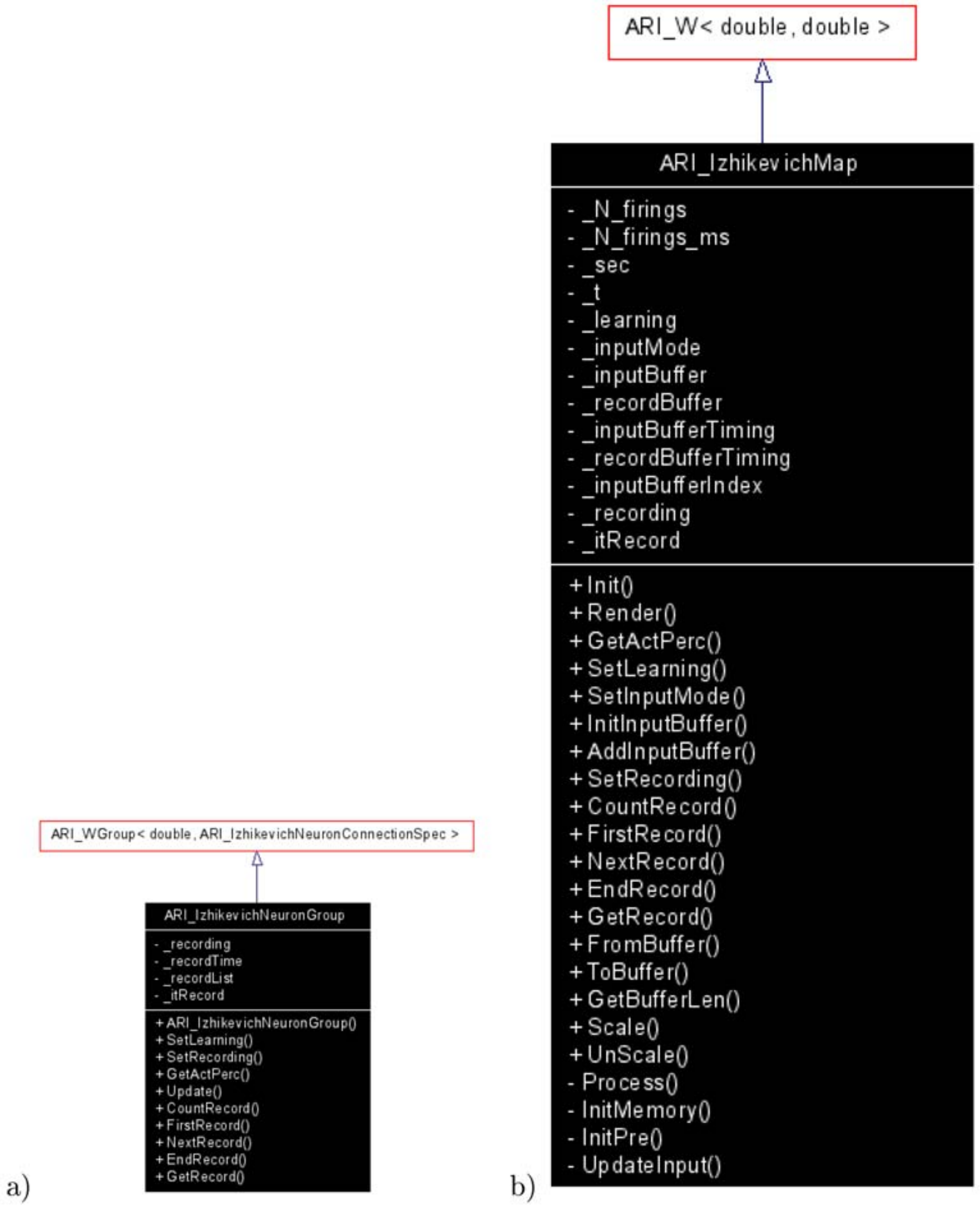


Fig. 5.7. a) The IzhikevichNeuronGroup structure. b) The IzhikevichMap structure

A *ThalamicRandomSensorDriver* structure has been developed to train the architecture with random signals. Such signals are the basis of the cortico-thalamic interplay of neural assemblies and temporal chains in the cerebral cortex. A *Mic-SensorDriverStructure* has been used to test the architecture with audio signals. For such signals the power spectrum has been obtained using the *ARI_FFT* structure. Both structures are showed in Fig. 5.8.

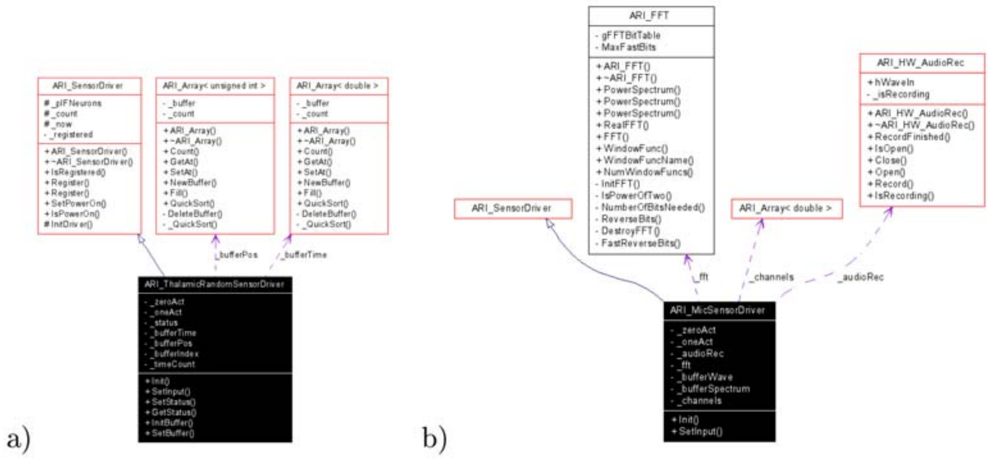


Fig. 5.8. a) The ThalamicRandomSensorDriver structure. b) The MicSensorDriver structure

6. Conclusions

In this work authors describe a high-efficiency architecture for parallel sensory fusion and real-time management of heterogeneous multi-transducers data processing. The interfaces with the external sensors and actuators, the specific control and processing methods and the data flowing through inner communication channels can be defined. For such entities the framework offers extendable structures, whose base implementation allows the realisation of high-efficiency data processing.Systems equipped with multiple transducers, tasks execution that are running as cooperative processes, off-line and real-time data aquisition and analysis tools, general stand alone applications represent some of the potential application areas.
A library-oriented interface was preferred to a user-oriented interface. Real-time analysis and actuation is gained for all the transducers and for all the running processes. Multi-process cooperation is possible thanks to a homogeneous communication language. The user can create extensions of new models of entities and processes. The data acquisition from sensor devices is granted by a protocol interface that is able to dispatch data coming from input systems. The data processing may be specified by the user inside the framework core. The actuator driving is granted by a protocol interface that is able to dispatch data from the framework core. Filters for sensory and actuating systems can be redefined according to the particular device technology; the efficiency of the filtering and buffering processes over the data coming from sensors and over the data directed to actuating devices is delegated to appropriate interfaces. The portability is allowed by a layered structure, an abstraction, and by the specification of the I/O drivers. A modular, reusable and object-

oriented architecture grants a parallel distributed processing, making the framework base architecture available to the researcher as a structured programming environment. Such features make the framework a solution for high-complex simulation tasks, representing a powerful instrument for the development of complex simulation tools operating as off-line and real-time applications.

7. References

[1] Lee, K. & Schneeman, R., (2000). Distributed measurement and control based on the IEEE 1451 smart transducer interface standards. *IEEE Transactions on Instrumentation and Measurement*, Vol. 49, No. 3.

[2] Steinberg, A.N. (2001). Data fusion system engineering. *IEEE Aerospace and Electronic Systems Magazine*, Vol. 16, No. 6.

[3] Martin, C.; Schaffernicht, E.; Scheidig, A. & Gross, H.-M. (2006). Multi-modal sensor fusion using a probabilistic aggregation scheme for people detection and tracking. *Robotics and Autonomous Systems*, Vol. 54, No. 9, 721-728.

[4] Li, C.; Heinemann, P. & Sherry, R. (2007). Neural network and Bayesian network fusion models to fuse electronic nose and surface acoustic wave sensor data for apple defect detection. *Sensors and Actuators B: Chemical*, Vol. 125, No 1, 301-310.

[5] Pioggia, G.; Ferro, M. & Di Francesco, F. (2007). Towards a real-time transduction and classification of chemo-resistive sensor array signals. *IEEE Sensors Journal*, Vo. 7, No. 2, 237-244.

[6] Ping, W.; Qingjun, L.; Wei, Z., Hua, C. & Ying, X. (2007). The design of biomimetic artificial nose and artificial tongue. *Sensors & Materials*, Vo. 19, No 5, 309-323.

[7] Soria-Frisch, A.; Verschae, R. & Olano, A. (2007). Fuzzy fusion for skin detection. *Fuzzy Sets and Systems*, Vol. 158, No. 3, 325-336.

[8] Härter, F.P. & Fraga de Campos Velho H. (2008). New approach to applying neural network in nonlinear dynamic model. *Applied Mathematical Modelling*, Vol. 32, No. 12, 2621-2633.

[9] Damasio, A. (2000). *The feeling of what happens*. London: William Heinemann.

[10] Driver, J. & Noesselt, T. (2008). Multisensory Interplay Reveals Crossmodal Influences on 'Sensory-Specific' Brain Regions, Neural Responses, and Judgments. *Neuron*, Vol. 57, No. 1, 11-23.

[11] O'Reilly, R.C. & Munakata, Y. (2000). Computational Explorations in Cognitive Neuro-science: Understanding the Mind by Simulating the Brain. MIT Press: Cambridge.

[12] O'Reilly, R.C. (2006). Biologically Based Computational Models of High-Level Cognition. Science, 314, 91-94.

[13] Izhikevich, E.M. (2007). Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting. The MIT press.

[14] Izhikevich, E.M. & Edelman, G.M. (2008). Large-Scale Model of Mammalian Thalamocortical Systems. *PNAS*, 105:3593-3598.

[15] Whittington, M.A.; Traub, R.D.; Kopell, N.; Ermentrout, B. & Buhl, E.H. (2000). Inhibitionbased rhythms: experimental and mathematical observations on network dynamics. *Int. J. of Psychophysiol.*, Vol. 38, 315-336.

[16] Sougnè, J.P. & French, R.M. (2002) Synfire Chains and Catastrophic Interference, *Proceedings of Annual Conference of Cognitive Science Society*.

[17] Norman K.A.; Newman E.L. & Perotte A.J. (2005). Methods for reducing interference in the Complementary Learning Systems model: Oscillating inhibition and autonomous memory rehearsal. *Neural Networks*, Vol. 18, No. 9, 1212-1228.
[18] Izhikevich, E.M.; Gally, J.A. & Edelman, G.M. (2004) Spike-Timing Dynamics of Neuronal Groups. *Cerebral Cortex*, Vol. 14, 933-944.
[19] Edelman, E. (1987). Neural Darwinism: The Theory Of Neuronal Group Selection. Basic Books.
[20] Izhikevich, E.M. (2006). Polychronization: Computation With Spikes. *Neural Computation*, Vol. 18, 245-282