Dextrous
Assembler
Robot
Working with embodied
INtelligence
DARWIN

DARWIN 270138

Dextrous Assembler Robot
Working with
Embodied Intelligence

SEVENTH FRAMEWORK
PROGRAMME

SEVENTH FRAMEWORK PROGRAMME
ICT Priority

Deliverable

User Manual for Darwin Architecture

Start date of project: 1st February 2011

Duration: 48 months

Coordinator:
Dr. Christian Eitzinger
Profactor GmbH

Revision 1

Lead Beneficiary: IIT
Contributors:PROFACTOR

Project co-funded by the European Commission within the 7th Framework Programme		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission	
CO	Confidential, only for members of the consortium (including the Commission Services)	X

Table of Contents

1. Introduction

The aim of this document is to concisely outline details about the installation and use of DARWIN architecture on multiple embodiments, with special focus on working with iCub humanoid robot (at IIT, Genova, Italy) and Stäubli RX130B and TX90L robots (at Profactor Gmbh, Steyr, Austria). The following sections will describe the software implementation of the overall architecture, how a prospective user can install, run and use the Darwin architecture while keeping in view the user story chosen for third-fourth year of the project which is the assembly of fusebox setup. Multiple aspects ranging from basic dependencies, preparatory steps to be taken to install and run Darwin to advanced aspects related to learning the body schema for any robotic embodiment, engaging in exploration during failures, formation and recall of episodic memories: with sequences of actions that the user needs to take, expected snapshots of console outputs and the resulting behavior of the robot. Further, the link between the deployed computational models (neural networks) and their implementation in software is described with information related to core software functions, the interfaces between them, the mathematical model/neural network and dynamics that is implemented. This is done mainly in relation to modules developed by IIT i.e. Forward/Inverse model of Action (PMP), Neural Episodic Memory module and the Observer module, that together from the core of the cognitive architecture.

1.1. Document outline

This document will cover the following topics:

- Description of DARWIN's software architecture and performing of assembly process using the DARWIN driven robots
- Full Installation procedure of the Darwin architecture
- Using the architecture to perform various tasks, Generalization of cognitive architecture, learning of the Neural PMP for any robotic embodiment, formation, recall and use of episodic memory of the Darwin robot.
- Link between the deployed computational models (neural networks) and their functional implementation in software. This is mainly in relation to modules developed by IIT i.e. Forward/Inverse model of Action (PMP), Neural Episodic Memory module and the Observer module, that together from the core of the cognitive architecture.
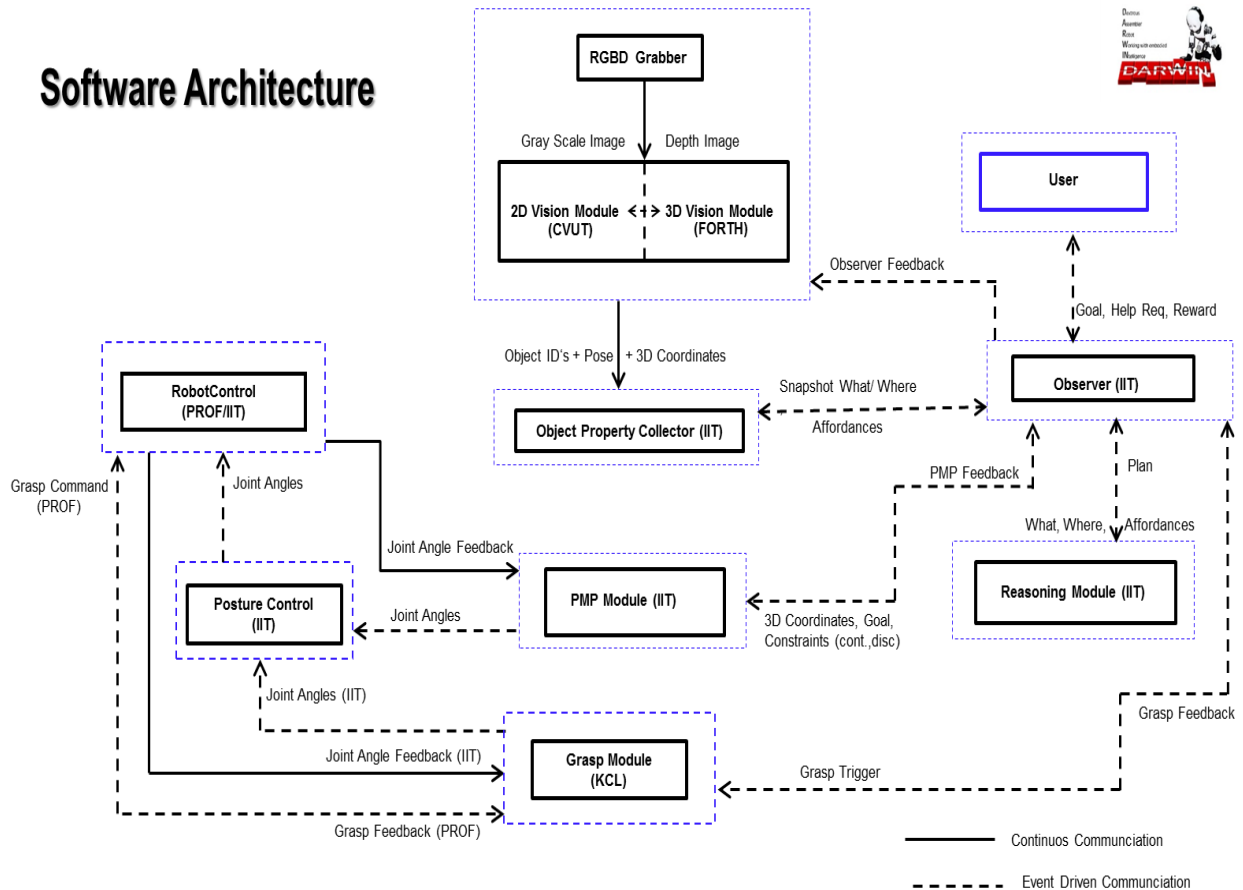
2. Darwin Architecture

DARWIN is aimed at enabling robots to exhibit intelligent behavior in manual processes like assembly tasks, exhibiting a fair amount of dexterity and flexibility while realizing its goals. Though DARWIN project focusses its work on two robots Stäubli (at Profactor) and iCub (at IIT), a fair amount of its design and software architecture is implementable on other embodiments. A pictorial representation of the integrated DARWIN cognitive architecture and underlying information flows between modules presently functional at both platforms (iCub and industrial robots) is shown in Figure 1, with a short textual description related to the basic input/output interfaces and the functionality achieved by each module. The overall architecture is designed keeping in mind that DARWIN is domain agnostic (caters to multiple tasks, reuses and composes new functionalities building up on primitives), partially "embodiment" agnostic (works on multiple platforms), goal driven (caters to the user needs and learns from the user inputs), "cumulatively learning and reasoning" (expands its knowledge base and its memories gradually and acts accordingly) and partially "self-driven" (can substitute the user and create one's own goals when necessary). The text below describes the core functionalities achieved by the various modules.

Figure 1: Diagram of the DARWIN software architecture and data flows.

The following modules, shown in the above diagram, were involved in the evaluation and demonstration on the industrial and the service platform.

RGBD Grabber: provides a continuous stream of images (RGB+Depth map images) coming from the Carmine PrimeSense 1.09 range sensor system.

2D Vision Module: takes as input grayscale images originating from the color (RGB) component of the depth sensor, performs object detection and rough localization on them and returns an output containing an object ID, rough 3D pose estimation and a bounding box for each object detected in the image. This information is forwarded to the 3D Vision module

3D Vision Module: receives the depth images from the RGBD grabber and the output from the 2D vision module. The module then performs 3D localization to finally provide full 6D pose information about each object which is detected by

the 2D vision module, using the internal models of the recognized objects and the RGBD sensor calibration data.

RobotControl/iCubInterface/iCub Simulator: manages the low-level communication with the robot's control unit. It processes grip triggers/fingers, torso and arm joint angles coming from the Grasp module and the Posture Control module and initiates the actual motion of the robotic platform.

PostureControl modulates the joint angle values coming from the PMP module and the Grasp module including the speed variations and communicates them to the robot for safe robot action coordination.

Observer: is the server for the user. It is responsible for the realization of the user goal by communicating to different subsystems (importantly, Episodic memory and reasoning, OPC (advanced perception), Event driven action subsystems like PMP). Complementing the issue of micro goals based on the task at hand, Observer also monitors success of every micro-event taking place, receives bidirectional feedback from its servers and takes decisions as to whom to contact in the future.

Reasoning Module: Advises the observer as to what to do in abstract terms in order to realize the goal at hand, by recalling the past learnt experiences in relation the context, combining them in novel ways as necessary. The system facilitates simulation of possible future events, formation of flexible plans and predictions.

PMP Module: Performs inverse kinematics "without" doing kinematic inversions, provides the motor commands to control the robot. The system acts as a forward-inverse model of the body that is common to both "action execution", "mental simulation of action" and "action perception.

Object Property Collector: Presently communicates "what and where" related information to the Observer (Client) in an event driven fashion.

Grasp Module: Performs the grasp functionality using two fingers that are closed electrically for the industrial platform and for all finger joints in iCub's hands. The gripper/hand allows continuous monitoring of the position and current. This module receives the trigger to start an operation (grasp/release) from the Observer module. The grasp module then replies back to the Observer module, about the result of the operation.

3. Setting up an fully functional DARWIN driven system

3.1. Hardware Requirements

Following is a list of components required for an efficient execution of the whole setup
-Personal Computers: A minimum of 2 PCs with dual core processors or more powerful processors with a RAM of 2 GB at least. Windows Vista (64-bit) or Windows 7 operating system (64-bit) should be installed on the PC's.
-Graphics cards: NVIDIA GTX 760 or higher Graphics card
-Cameras: Carmine Sensor 1.09 or Kinect
-Objects: A fusebox and three fuses
-Robot: A robot system with an interface to the robot (iCub or Staubli)
-Additional hardware: In case robot has additional hardware required like grippers, they should be installed and an interface to work with them should be available.

3.2. Software Dependencies

Following software's are needed to be pre-installed in order to make the whole architecture run smoothly on a system. The links to download softwares are also provided.
Be sure to
-Install Visual Studio 2010 with service pack 1 and a C++ programming environment.
http://www.microsoft.com/en-us/download/details.aspx?id=2680
 -Install a latest version of CMake
http://www.cmake.org/
-Install an SVN Client like tortoiseSVN for subversion control
 http://tortoisesvn.net/downloads.html
-The link to the root directory of DARWIN software for SVN versioning is given below. Username and password to access the repository can be provided on request. The code is needed to be downloaded from
https://pinguin.profactor.at/mas/darwin/trunk
-Install Cuda - cuda_6.0.37_winvista_win7_win8.1_general_64
-Install Device Drivers (OpenNI-Windows-x86-2.2.0.33) for camera
-install YARP version yarp_2.3.22_v10_x86_0 http://wiki.icub.org/wiki/Downloads
-To run the architecture on iCub or iCub Simulator, install iCub
http://wiki.icub.org/wiki/Windows:_installation_from_sources

- To run the architecture on Staubli or Staubli Simulator, install the Profactor robot controller
https://pinguin.profactor.at/mas/darwin/trunk/profactor/RobotControl

3.3. Working with the repository
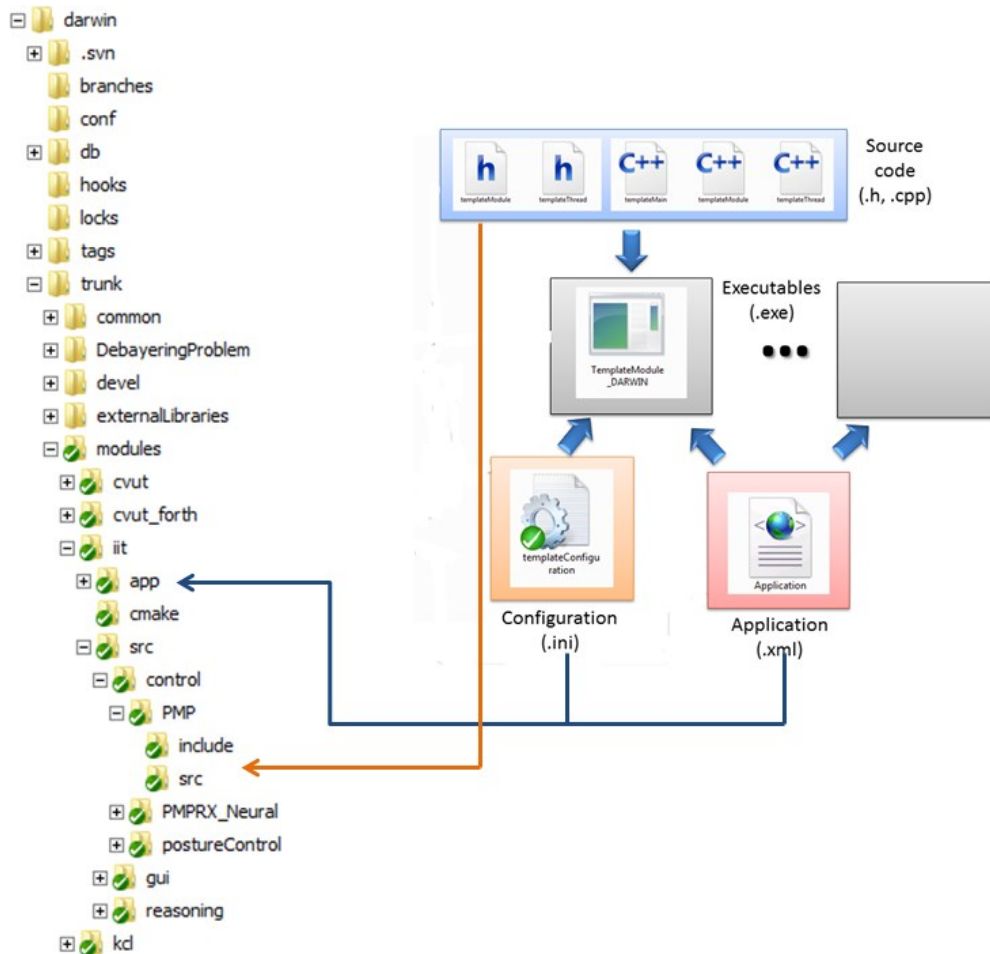
Figure 2: Structure of DARWIN software repository.

Figure 2 shows an overview of the repository structure compatible with the YARP standards that we follow in DARWIN. This hierarchy is needed to run the code directly on the robot. According to this file tree, each partner's source code is stored in dedicated subfolders of the *modules* folder. Each subfolder contains a *build*, an *include* and a *src* folder somewhere inside the structure, containing the Compiler Project Solution, the header files and the cpp source files respectively. In addition, in each subfolder of *modules* folder, a CMakeLists.txt file exists in order to generate the Compiler Project Solution through CMake. The *commons*

folder contains all the configuration files needed for standardized communication protocols between the modules.
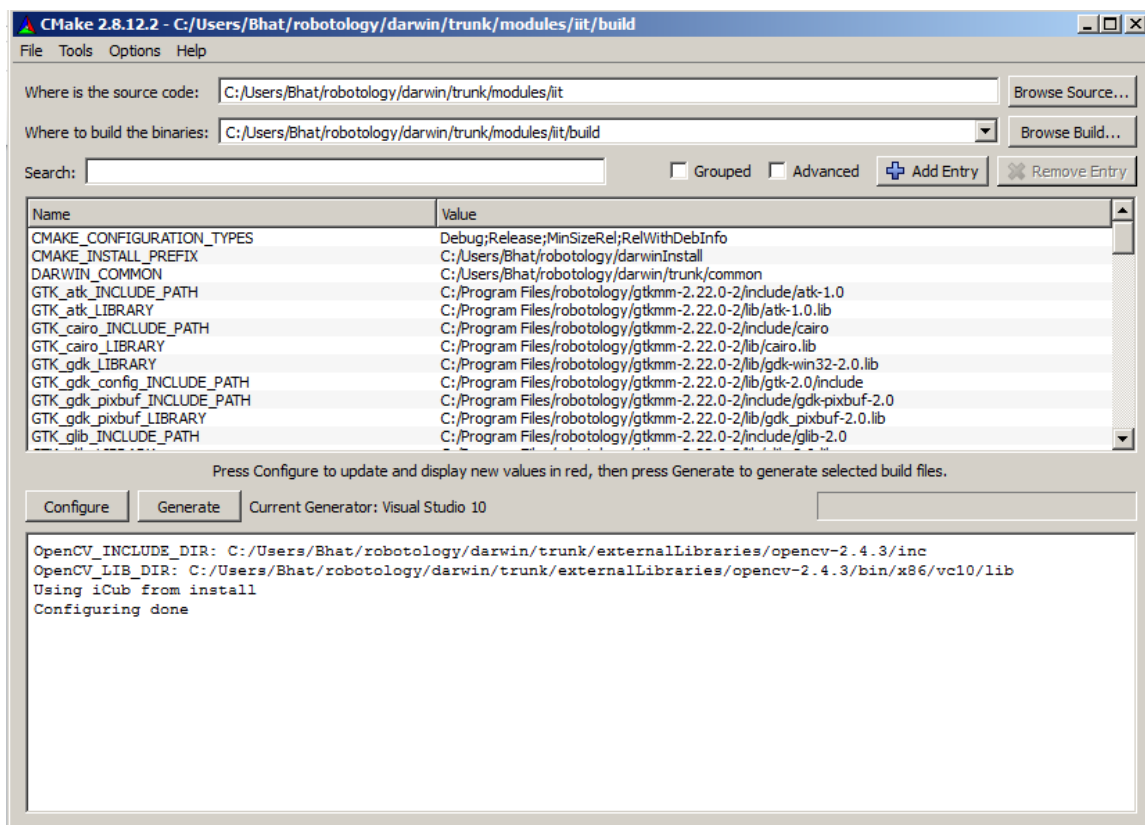
Figure 3: Snapshot of CMake-ing a partner's modules

In order to build a partner's modules, Open CMake with administrative privileges. The CMake options should be filled in as Figure 3 shows:

- Where is the Source Code : the path in which the CMakeLists.txt file exists. The paths for different modules are as follows:

For Vision modules
darwin\trunk\modules\cvut_forth\objDet

For Cognitive modules, Action coordination and iCub posture control modules
\darwin\trunk\modules\iit

For Grasping Modules
 \darwin\trunk\modules\kcl

- For Where to build the binaries : a *build* folder

After you have configured it set *CMAKE_INSTALL_PREFIX* value. Preferably create a folder *darwinInstall* and set the value to its path. Press Configure button until all the fields are no more red, then Generate the solution using Visual Studio 10 compiler chosen as default by CMake. A Compiler Project Solution should appear in the *build* folder. Open the Compiler Project Solution in the *build* folder and BUILD it. In order to create the executable in the *darwinInstall \bin* directory compile the INSTALL project inside the Compiler Project Solution. Except for the vision modules, for which the executable will be in the *Release* sub-folder of the *objDet\bin* directory. Following executables are needed to run a complete experiment on a platform:

For Industrial Staubli platform:
ObjDet_standalone.exe, *ObserverIndustrial.exe*, *PMPRX_Neural.exe*, *TheNewEpiM.exe*, *OPC.exe*, *GraspModule.exe*

For iCub Humanoid platform
ObjDet_standalone.exe, *TheNewObserver.exe*, *PMP.exe*, *TheNewEpiM.exe*, *OPC.exe*, *GraspModule.exe*, *postureControl.exe*

Setting environment variables: If yarp and iCub are installed using executables provided from the website mentioned above then, YARP_DIR and YARP_DATA_DIRS variables are automatically set. Add following new environment variables:
DARWIN_INSTALL and set it *darwinInstall/bin* folder's path as created above.
DARWIN_ROOT and set it to *darwinInstall\share\darwin* folder's path.
YARP_POLICY and set it to DARWIN_ROOT

4. Customization of Modules before use

4.1. Vision Module

Calibration of the camera
The vision module requires the calibration information (intrinsic and extrinsic parameters) of the 3D sensor as a prerequisite. The intrinsic and extrinsic parameters of the sensor can be calculated using the steps provided in the attached document titled "*CvUtils_Documentation.pdf*".
Following this go to the folder *\cvut_forth\objDet\data\forth* and the path to the calibration file *.calib file* generated above needs to be pasted against the entry "*calib*": in the *.json* file you are using (depending on the robot).

Setting up paths to training data

Once the cameras are calibrated and the paths to calibration files and training data are set properly, create two folders *outres* and *data* in the *cvut_forth\objDet\bin\Release* directory. These two folders are created to store the images and the debug information. This can switched off in the code. Compile the complete solution and build. The vision system is now setup.

For more information on the output of the vision system and how it communicates with other modules, check the link http://darwin-project.eu/?incsub_wiki=vision-module-cvutforth

4.2. Observer, Reasoning, PMP, OPC and Grasping Modules

- Once modules from *iit* folder are installed, the required application templates will also be placed in *\darwinInstall\share\darwin\templates\applications* folder.
- The configuration files for Observer (*TheNewObserver.exe/Observerindustrial.exe*), OPC, PMP and Reasoning (*TheNewEpiM.exe*) modules will be placed in *darwinInstall\share\darwin\contexts\perceptionActionCycleApp* folder;
- Configuration files for PMPRX_Neural will be placed in *\darwinInstall\share\darwin\contexts\RXNeural* folder. The application templates need to be renamed to .xml files to run them with the yarpmanager.
- Similarly, after installing Grasping module from *kcl* folder, the related application templates will also be placed in *\darwinInstall\share\darwin\templates\applications* folder and the configuration files in *\darwinInstall\share\darwin\contexts\GraspModuleApp* folder.
- These application files can be used to run the GraspModule and set up the connections between GraspModule and Other related modules.
- Check the link http://darwin-project.eu/?incsub_wiki=grasp-module-kcl for more details on the grasp module working and use.

5. Using the Darwin architecture for different tasks

5.1. Performing an assembly task

Once all the executables, configutaion files and applications files are in the *\darwinInstall* folder and vision module is properly built in

cvut_forth\objDet\bin\Release folder, the whole system can be used run together to perform an assembly process.

Basic Procedure *(repeated for all further scenarios)* Start by setting up objects (fuses and fuseboxes) in the robot's workspace area and then opening a command window and type on the terminal

 yarpserver

Figure 4: Running a yarp server

This will start a yarp server which serves to facilitate communication between the modules. Go to the *cvut_forth\objDet\bin\Release* folder and run *objDet_standalone.exe* by double clicking the executable. Now the vision should be running and two windows showing vision output; one 2D bouding boxes on detected objects in the environment and other 3D pose estimation of the detected objects.

 On Industrial robots, run these modules: *RobotController.exe*, *GraspModule.exe*, *ObserverIndustrial.exe*, *OPC.exe*, *TheNewEpiM.exe*, *PMPRX_Neural.exe*.

On iCub, run these modules: *iCubInterface.exe* if you are working on the robot or *iCub_SIM.exe* if you are running on a simulator, *GraspModule.exe*, *TheNewObserver.exe*, *OPC.exe*, *TheNewEpiM.exe*, *PMP.exe*, *PostureControl.exe*.

Figure 5: Multiple modules running together and communicating

Once the above modules are running, in the *ObserverIndustrial.exe* or *TheNewObserver.exe* (whichever you are running) type the following two key words for performing assembly task of inserting a fuse into a fusebox.

Type *asm* and press enter. (This is an acronym for to perform an assembly process)

Type *comf* and press enter. (This is an acronym referring to composite-fuse box meaning fusebox with the fuse inserted). Following this the system should perform the assembly process of its own.

Figure 6: iCub performing an assembly task

5.2. Performing sub-tasks through User-Observer Protolanguage

The system is designed such that it can also process the micro-goals that need to be executed in an assembly process as independent goals on their own. Hence the system be asked to perform sub actions via language: reach an object or grasp an object or push something etc. Teaching assembly plans (i.e. sequences of actions on different objects leading to some assembly) via proto language is discussed in section 7.6. after the description of Observer and Episodic memory module.

Reaching an Object: Run all the modules that are mentioned above. In this case *GraspModule.exe* is not needed to be run as the goal is only to reach the object. To reach an object, type in the Observer terminal *rea* command and enter; followed by *fuse* (to reach the fuse) or *stan* (to reach the fuse box) or *comf* (to reach the composite fuse box).

Grasping an Object: Run all the modules that are mentioned above including the *GraspModule.exe* as the goal is to grasp the object after reaching. To grasp

an object, type in the Observer terminal *grsp* command and enter; followed by *fuse* (to grasp the fuse).

Pushing an Object: Run all the modules that are mentioned above. In this case *GraspModule.exe* is not needed to be run as the goal is only to push the object. To push an object type in the Observer terminal *push* command and enter; followed by *stan* (to push the fuse box) or *comf* (to push the composite fuse box).

6. Neural PMP: Learning the Forward/Inverse model of the Body (for any embodiment)

Action generation system in DARWIN employs a neural implementation of Passive Motion Paradigm (PMP). PMP is basically a forward/inverse model that is used to both generate motor commands to drive the robot and simulate consequences of actions. We employ a neural representation of the body schema of a robot inside the structure of PMP which requires well-trained artificial neural network for coding the mapping between the intrinsic joint space of the robot and extrinsic 3D space in the environment. Both the PMP modules in the repository (PMP for iCub and PMPRX_Neural for Industrial robot) have the same neural PMP implementation except the differences due to different joint spaces which ask for representation in two different neural networks. The process of setting up a usable neural representation for PMP inside the DARWIN architecture can be divided into; acquiring data sets for training, followed by training the network and finally fitting the neural network into the system set for use. The three steps as discussed below in more detail:

6.1 Data Generation

   To train a neural network, a large data set is needed, this dataset consists of vectors that represent a possible posture in the robot's joint space and the corresponding location of robot's end-effector in 3D space. To generate the dataset one has to find for each joint angles set of the robot in the usable workspace a corresponding 3D location values set of the end-effector. This can be achieved by the kinesthetic learning of the robot. This method is very similar to the way motor babbling occurs in infants. The end-effectors (hands in case of iCub and grippers in case of industrial robots) are moved in the peri-personal space of the robot and the end-effector location is tracked using vision and/or sensory systems. Internally, the robot's task-specific body joint space (which in case of iCub is the torso and arm and in case of the industrial robot is the arm) is also tracked. Another way to generate the data is to iterate the joint angle values of the robot in the reachable workspace to generate corresponding end-effector

locations using forward kinematics of the robot. The size of the dataset required depends on the size of joint space; the larger the joint space, the bigger is the dataset needed. This data set can then be used to train a neural network. We implemented the codes in MATLAB and C++ for this purpose which are attached. *DatGenICUBR.m* file can be used to generate the data for iCub's training. *industrialTX_data_gen.cpp* code can be used to generate training data for Industrial Robot arm TX90 while as *industrialRX_data_gen.cpp* code can be used to generate training data for Industrial Robot arm RX130. In the files generated, the joint-space datafile is named as *Proximal.txt* and the end-effector space datafile is named as *Distal.txt*.
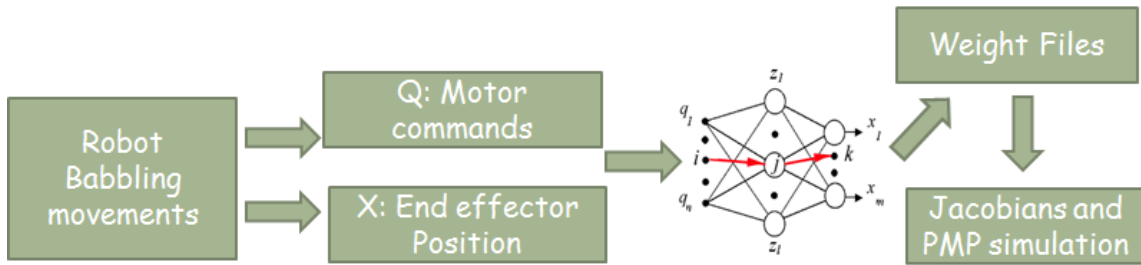

Figure 7. Shows the block diagram of the stages to learn the Neural PMP for a new robot

6.2 Training the neural network

Once the data is generated, *rough_code_trainall.m* file can be used to train the neural network using the data generated. It uses the joint angle vectors in *Proximal.txt* as input data and the end-effector location vectors in *Distal.txt* as target data. Our code uses *newff* function in MATLAB to train a neural network. It is important to note that one has to set the size of the layers of neural network in this MATLAB code file. The ones we chose for iCub's neural network were 48 neurons for first hidden and 55 neurons in second hidden layer owing to larger joint space of iCub (3 joints of torso and 7 joints of an arm). For industrial robots we chose 32 neurons and 41 neurons as two layers' size respectively. It is advised to train the network with at least 0.3 million data points so that the network prediction is fairly accurate. We achieved a mean square error of 0.1 from our trainings. Once the network is trained the MATLAB workspace should be saved and the corresponding weight files can be generated by running the script in *weightSaver.m*.

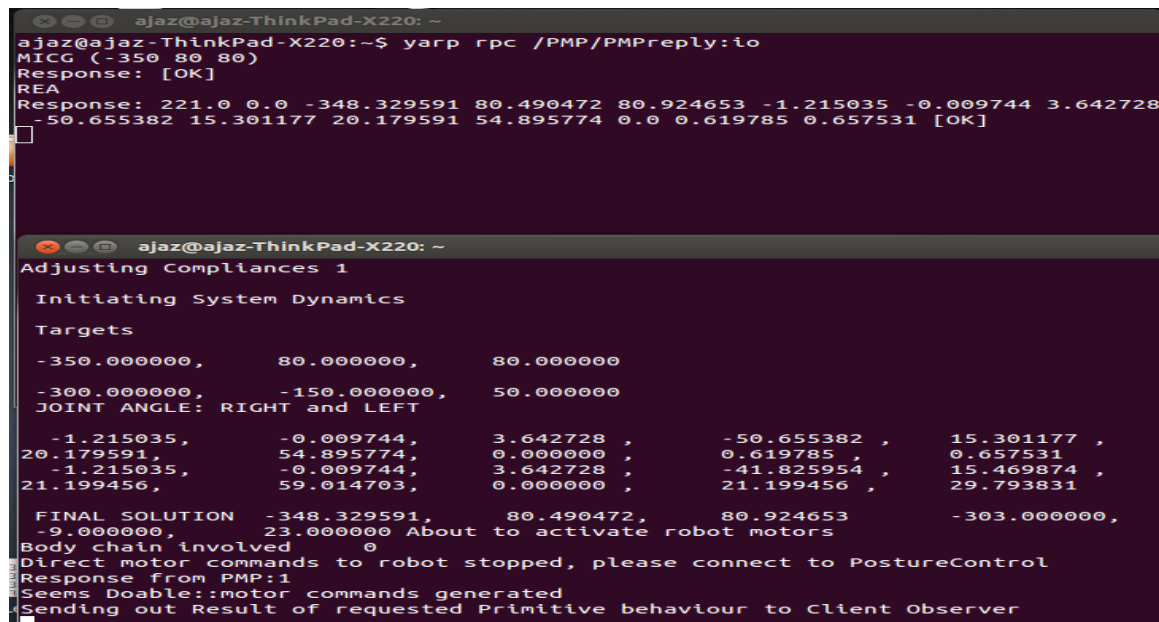## 6.3 Deployment and testing of the neural network

These weight files generated (6 in number for each robot) should be placed in *\darwin\trunk\modules\iit\app\perceptionActionCycleApp* folder for iCub and *\darwin\trunk\modules\iit\app\RXNeural* folder for industrial robots. The CMake and BUILD is needed to be run again after placing the files in corresponding folders. This will copy the files to *\darwinInstall\share\darwin\contexts\perceptionActionCycleApp* folder for iCub; and *\darwinInstall\share\darwin\contexts\RXNeural* folder for industrial robots for the exectables to use the data properly. The PMP module can be tested independently of the rest of the DARWIN architecture also. For example in case of iCub, neural PMP code executable *PMP.exe* can be tested in the following way:

Open a windows command prompt and type "*yarpserver*".

Run the *PMP.exe* executable, can be found in *\darwinInstall* folder after installation. As the executable runs, it prints the messages about if the neural network files are properly loaded or not.

Further open another command prompt and type *"yarp rpc /PMP/PMPreply:io"* and press enter. This opens an yarp rpc port to communicate with the PMP module. Here, one can directly give a goal to PMP to reach; type (for example) *"MICG (-350 80 80)"* and press enter. This directs the PMP to reach a target of 350 millimeters in *-x* direction and 80 millimeters in *y* as well as *z* direction. See Figure 8 for an example.

Figure 8: Running a PMP module and communicating using yarp rpc service

6.4 Neural PMP: Computational model vs. Implementation of the Neural network as a software package

The Neural PMP module basically implements a forward/inverse model that is both used to generate motor commands and simulate consequences of potential actions during goal directed reasoning. Figure 9 shows the basic information flow inside the software module (Neural PMP), when a micro goal (for example, in the simplest case reaching a target X_d) is issued by the Observer. As seen, When the goal X_d is issued, there are basically 4 critical mappings: Generation of the force field F, transformation from force to torque using the Jacobian transpose J^T, From torque to joint velocity through the admittance matrix and joint velocity to end effector displacement (through the forward kinematics). From the weights of the trained neural network (as described in sections 5.3.1-5.3.3) , both the J^T and the Forward kinematics are computed. This loop of transformations as described in figure 9 runs iteratively, till the time $x=x_d$ (i.e. the force F=0). At this point the dynamics reaches equilibrium, and the final values of Q are the desired joint angles to position the robot at x_d. Of course the motor commands can be sent to the robot in case the simulation converges or the final result of the PMP simulation can be sent back to the observer (for example if the target is not reachable, the non-convergence of the PMP simulation provides geometric information which is the starting point to reason further, for example select an appropriate tool).
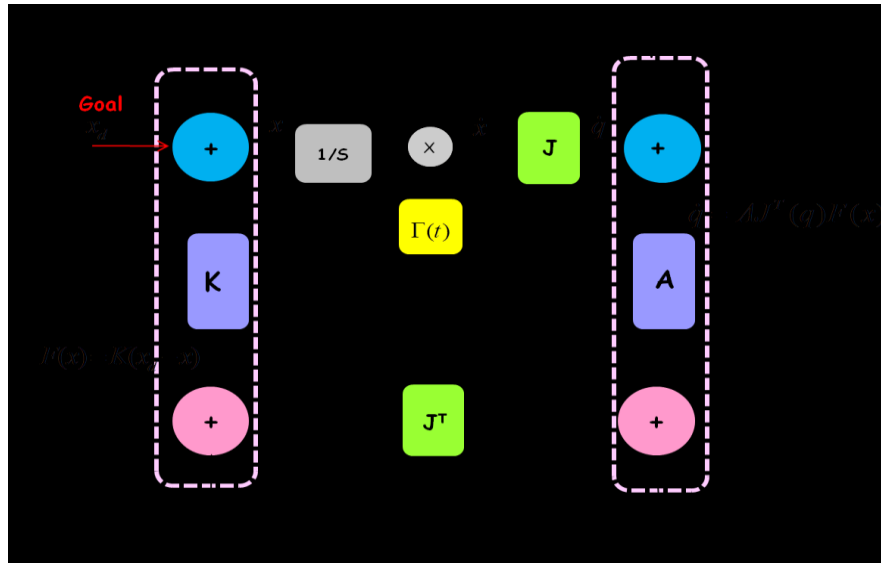

Figure 9. Central loop of transformations taking place in the Neural PMP forward/inverse model. When the goal X_d is issued, there are basically 4 critical mappings: Generation of the force field F, transformation from force to torque using the Jacobian transpose J^T, From torque to joint velocity through the

admittance matrix and joint velocity to end effector displacement (through the forward kinematics). From the weights of the trained neural network (as described in 5.3.1-5.3.3), both the J^T and the Forward kinematics are computed.

6.4.1 Bidirectional Interface between Observer and PMP

With the contextual information related to the core information flows as depicted in figure 6, we now briefly summarize how this overall loop of PMP dynamics, the computations related to the trained neural network are realized in the NeuralPMP module. Figure 10, shows the Interface between Observer and PMP. The interface from the Observer to Neural PMP is implemented through the *PrimBodySchema function in ObserverThread.cpp* of the Observer, by means of a RPC port. Based on the evolution of the higher level assembly plan , the Observer sends information necessary to configure the PMP network: *Note that PMP is a task specific model, can be configured at runtime based on the goal and constraints.* The message sent from the observer to PMP contains the following:

- GoalCode (i.e. what action has to be performed);
- MsimFlag (that communicates weather the action has to be executed or simulated in order to get the predicted consequence bas to the observer, for example consequences of pushing, reaching etc. that allows the Observer to engage in further goal directed reasoning)
- WristOrient (that is the desired configuration of the hand, which is an additional task specific constraint in addition to reaching the target)
- TrajType (That indicates if obstacles have to be avoided, in which case the PMP generates a trajectory to take into account the obstacle in the scene: for example pick up the fuse without hitting the fuse box that is very close to the grasped fuse)
- ObjectID (gives information related to the object on which action is supposed to take place, this information is used to control the speed of the robot variably based on the context)
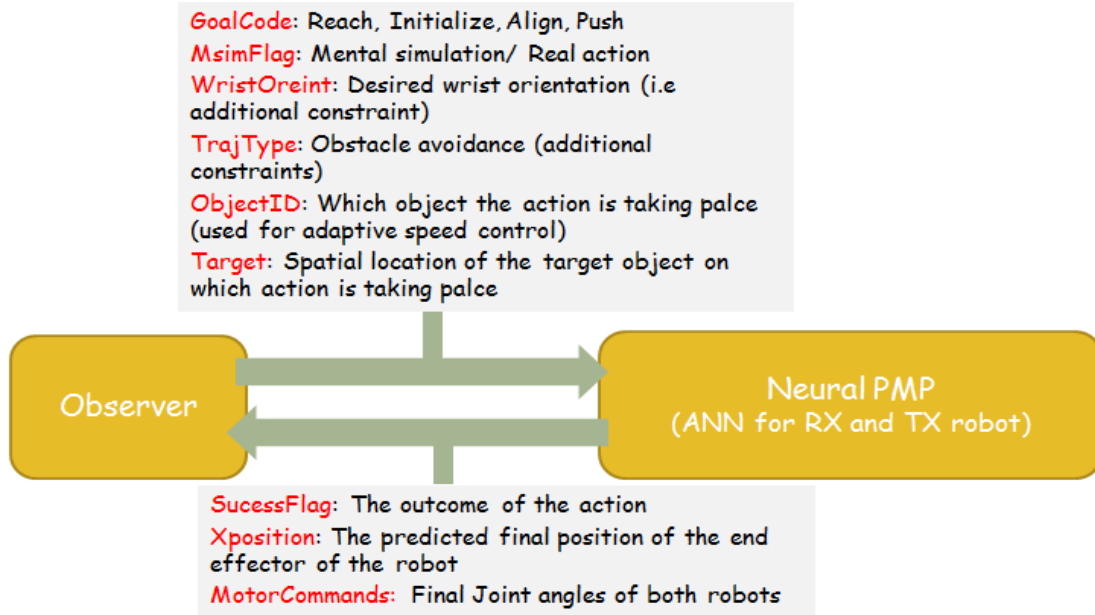
GoalCode: Reach, Initialize, Align, Push
MsimFlag: Mental simulation/ Real action
WristOreint: Desired wrist orientation (i.e additional constraint)
TrajType: Obstacle avoidance (additional constraints)
ObjectID: Which object the action is taking palce (used for adaptive speed control)
Target: Spatial location of the target object on which action is taking palce

Observer ⇄ Neural PMP (ANN for RX and TX robot)

SucessFlag: The outcome of the action
Xposition: The predicted final position of the end effector of the robot
MotorCommands: Final Joint angles of both robots

Figure 10: Bidirectional Interface between Observer and the Neural PMP module

- Target (3D location of the desired target, estimated by the vision system)

The message from the Observer is interpreted by the PMP module and is used to configure the neural forward./inverse model, taking into account task constraints. At the end of the PMP simulation, the PMP module returns back the following pieces of information to the Observer.

SucessFlag: Indicating whether the PMP simulation converged and goal was achieved

XPosition: The predicted final position of the end effector of the robot

Motor Commands: The final joint angles of the robot, to realize the goal.

6.4.2 PMP dynamics and Neural Implementation in Software

The last subsection basically described the bidirectional interface between the Observer and Neural PMP, clarifying the contents of the messages being exchanged between the two modules. In this section, we go inside the PMP module to describe how the PMP module performs its functions as a Integrated forward/Inverse module, How the neural network and underlying dynamics is implemented in software. Figure 11 outlines the basic software structure inside the PMP module, listing some of the central functions and internal information flow between them. Below we concisely outline the precise role of each function, the underlying computations implemented inside them and how the overall goal is achieved.

Run: This function implements the basic I/O interface of the PMP module with the Observer. Then it triggers a set of Initialization functions related to loading the weights of the neural network of the appropriate robot (performing/simulating) the

action, initializing the stiffness, admittance and timing parameters (K,A, and T of figure 6) and then communicating the goal to the VTGS function (Virtual trajectory generation system) that implements the core of the PMP dynamics.

Load ANN: As the name implies, it loads the weights and biases of the appropriate neural network for the robot performing the action. At present for both iCub and the two industrial platforms, a two layer backpropogation network is used, details as mentioned in 5.3.3.

Kompliance: This function assigns the parameters related to stiffness K and admittance A, that govern the transformation form displacement to force and torque to joint velocity as described in figure 6. Other parameters like number of iterations in the PMP dynamics from the initial condition to the final solution, parameters necessary for terminal attractor dynamics (Function: Gamma, GammaInt) are assigned through this function.
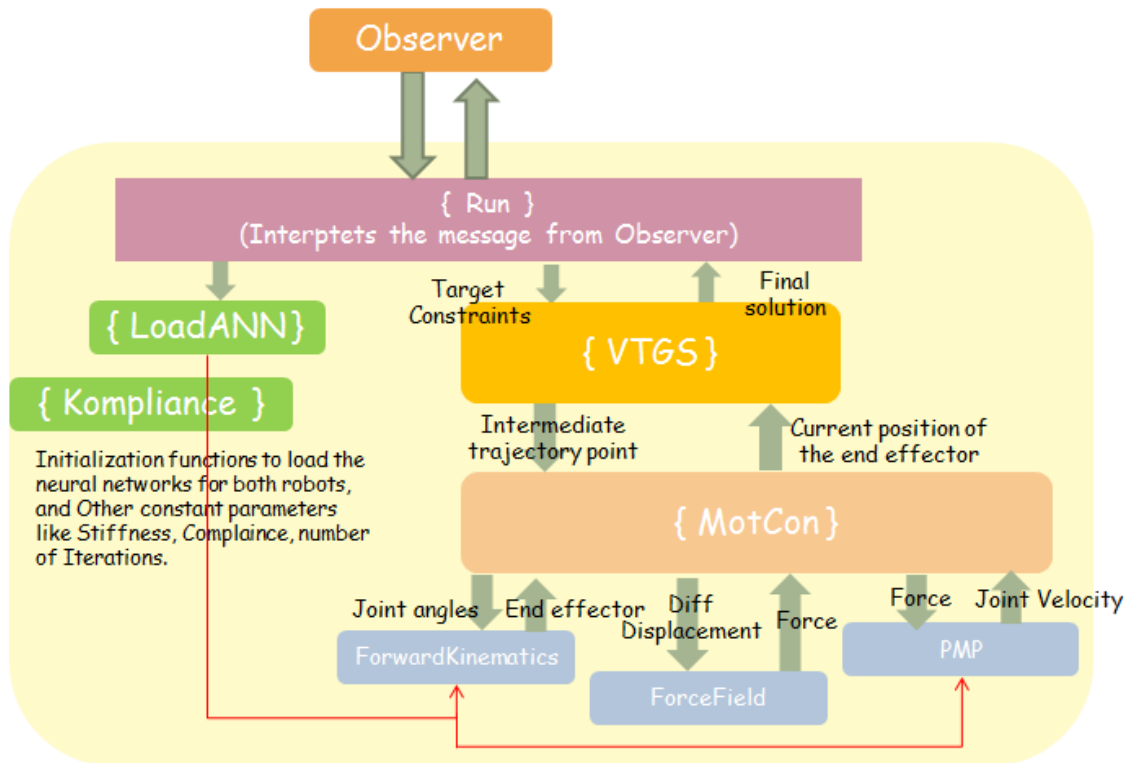

Figure 11: Pictorially depicts the basic software structure inside the PMP module, listing some of the central functions and internal information flow between them

VTGS: VTGS stands for virtual trajectory generation system, and basically generates the moving point attractor to trigger PMP dynamics. In the default case, a straight line trajectory from initial condition to target is synthesized, but during obstacle avoidance, curved trajectories escaping the obstacle are synthesized. VTGS system may be considered as the puppeteer that pulls the

end effector along a specified path. Let Xini ϵ (x,y,z) be the initial condition i.e. the point in space from where the generation of the trajectory is expected to commence (usually initial condition). If there are N via points (in the default case there will be no via point only the final target, but while avoiding obstacles there may be an intermediate via point), the spatiotemporal evolution of virtual trajectory (x,y,z,t) is equivalent to integrating a differential equation that takes the form of equation 1 that is implemented inside the VTGS function:

$$\dot{x}_{ini} = \sum_{i=1}^{N} K_i \gamma_i(t) \cdot \left(x_{CP_i} - x_{ini} \right) \tag{1}$$

Where K is the stiffness and γ is the timing, loaded from the Kompliance function. Xini is the initial starting point and X_{cp} either the final target or a via point. In the present implementation, from the initial position to the goal we generate 1000 intermediate points like a trajectory. ***So while the run function communicates the initial and final desired position, VTGS generates a suitable motion trajectory between them.***

MOTCON: Motcon stands for Motor control. For every intermediate point communicated in the trajectory communicated by the VTGS, motcon generates the necessary joint angles. In simple terms, if these joint angles are fed to the actuators, the robot will actually reproduce the trajectory (hence VTGS is called as Virtual trajectory generation system, because it is the input that drives the synthesis of the real trajectory). Being a core function, it deploys many different sub functions.

Forward Kinematics: Given an intermediate target by the VTGS, the first step is to know where the end effector is presently, this is needed to compute the force field (see figure 6). We are presently using a multilayer feed forward neural network with two hidden layers, to learn the mapping $X = f(Q)$ where $Q = \{q_i\}$ is the input vector (of joint angles), $X = \{x_k\}$ is the output vector (representing 3D position/orientation of the end-effector) and $Z = \{z_j\}$ and $Y = \{y_l\}$ vectors are the output of first and second hidden layer units respectively. Equation 2 expresses the mapping implemented by this function, where $\Omega = \{\omega_{ij}\}$ are connection weights from the input layer to first hidden layer, $O = \{o_{jl}\}$ are the connection weights between two hidden layers, $W = \{w_{lk}\}$ are the connection weights from the second hidden layer to the output layer, $H = \{h_j\}$ are the net inputs to the neurons of the first hidden layer and $P = \{p_l\}$ are net inputs to the second hidden layer. Neurons of the two hidden layers fire using the hyperbolic tangent function; the output layer neurons are linear.

$$X = f(Q) \Rightarrow \begin{cases} h_j = \Sigma_i \, \omega_{ij} \, q_i \\ z_j = g(h_j) \\ p_l = \Sigma_j \, o_{jl} \, z_j \\ y_l = g(p_l) \\ x_k = \Sigma_l w_{lk} \, y_l \ = \Sigma_l w_{lk} \cdot \, g(\Sigma_j \, o_{jl} \, z_j) \\ \Rightarrow \quad x_k = \ \Sigma_l w_{lk} \cdot \, g(\Sigma_j \, o_{jl} \cdot \, g(\Sigma_i \, \omega_{ij} \, q_i)) \end{cases} \tag{2}$$

Force field: This function takes the intermediate trajectory point and the current predicted end effector position to generate the next incremental force $F = K_{ext}(x_T - x)$, where K is the stiffness.

PMP: This function implements the transformation from force (computed by force field), into a torque field through the transpose Jacobian, and then the transformation from torque to joint velocity through the admittance matrix. Additional task specific constraints like desired wrist pose; joint limits are also integrated at this point to converge to a solution (i.e. motor commands) that take into account multiple task specific constraints. The Jacobian is a function of the evolving simulated joint angles as the iterations progresses, and is computed from the weights of the neural network. Precisely, the following equation is implemented in this function to compute the Jacobian from the weights of the neural network.

$$J = \frac{\partial x_k}{\partial q_i} = \ \Sigma_l w_{lk} \ \cdot g^{-1}(p_l) \ \ \Sigma_j \, o_{jl} \ \cdot g^{-1}(h_j) \, \omega_{ij} \tag{3}$$

Where, $\{\omega_{ij}\}$ are connection weights from the input layer to first hidden layer, $O = \{o_{jl}\}$ are the connection weights between two hidden layers, $W = \{w_{lk}\}$ are the connection weights from the second hidden layer to the output layer, $H = \{h_j\}$ are the net inputs to the neurons of the first hidden layer and $P = \{p_l\}$ are net inputs to the second hidden layer. Neurons of the two hidden layers fire using the hyperbolic tangent function; the output layer neurons are linear. The PMP function outputs the desired joint angles for reaching the intermediate point determined by the VTGS, this loop continues till the whole trajectory determined by the VTGS is synthesized, in parallel deriving the joint angles to execute the movement (joint velocity and joint torques) or predict the consequences of it (resulting end effector position or force).

7. Learning, Remembering and exploiting episodes of experiences

While the previous section described the process of learning the Neural PMP for generation and simulation of action in any robot, this section describes how new episodic memories of the robot can be formed, how past experiences can be merged with explorative actions to learn further and how such experiences can be remembered and used by the reasoning system in novel situations. The example provided illustrates sequentially the procedure to be followed taking the Fuse Box assembly task as a case study. The central challenge in formation and recall of episodic memory is to cumulatively train the connectivity matrix of the episodic memory (of the order 10^6 connections). The connectivity matrix is stored in *\darwin\trunk\modules\iit\app\perceptionActionCycleApp\WMems77N.txt.* In the example, we show how the episodic memory can be trained at runtime starting from the point where only one basic primitive schema i.e. Reaching is already present in the memory.
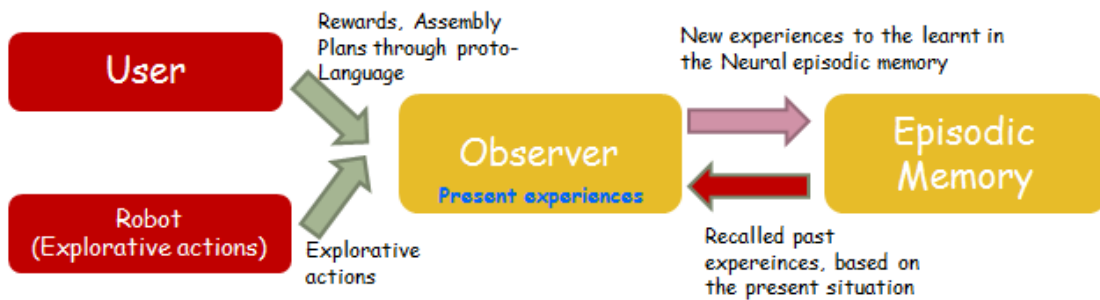

Figure 12. Block diagram of interfaces between User , Robot, Observer and Episodic memory modules while learning new experiences, encoding them in the episodic memory network and recalling them based on context in the future.

The number of episodes of experiences stored in the memory can be configured through *\darwin\trunk\modules\iit\app\perceptionActionCycleApp\numepi.txt.* A maximum of 235 episodes can be stored in the same network consisting of 1000 neurons. Figure 12 summarizes the overall picture as existing at present. Learning is possible either through explorative actions initiated by the robot that can be combined with past experience to form new memories, or through user input like rewards. Observer module being the central coordinator always keeps track of the present situation, which can be encoded as a new memory in the episodic memory network by updating "Weight77N.txt" that holds a 1000x1000 matrix representing the connections between neurons in the episodic memory. Once new memories are formed, the dynamics of the episodic memory allows

these experiences to be remembered in the future based on partial cues (sent from the observer) and reason about plans.

7.1 Observer-Episodic Memory Loop: Neural Implementation in software

This section concisely describes the software implementation of the Observer-Episodic memory loop that is the core of the reasoning system. The details of the neural networks involved, the underlying dynamics and implementation of the system in software is described. The bidirectional interface between the Observer and Episodic memory is shown in figure 10. The system is triggered when the user issues a goal (Example: Reach Fuse or Build the tallest possible stack) to the Observer. The Observer receives feedback from the vision system related to the scene analysis (what, where) and now has the information from the user related to the Goal to be realized. This information is sent to the episodic memory, in order to synthesize a plan in the given context. The goal Id, information related to objects in the scene are the basic information required by the episodic memory network to recall past experiences and generate a possible plan (Example, if the goal is to Stack, objects are Mushroom and Cube, the episodic memory recalls past experiences related to this context and sends a plan to place the mushroom on top of the cube). In situations where the plan communicated by the episodic memory fails, the observer transmits the present context once again to the EM module, to request an alternative solution if known (from past experience). This can happen in many situations for example failure of primitive actions (reach, grasp), novel objects in the environment of which no past experience exist and others.

7.2 Episodic Memory module: Computational model vs. Software implementation

Details of the implememnted neural network: The episodic memory network consists of 1000 neurons organized in a sheet like structure with 20 rows each containing 50 neurons. The memory circuit is characterized by "all-to-all" connections between the N excitatory neurons (thus the connectivity matrix is of the order NxN). Memories are stored in the network by updating the connections between different neurons using Hebbian learning. In addition, there is an inhibitory network that is equally driven by all N excitatory neurons and in turn inhibits equally all excitatory units. A rate-based model is used, in which the instantaneous firing rate of each neuron is a function of its instantaneous input current. More formal details can be found in (Mohan et al, 2014). In this document, we specifically focus on how the neural memory and underlying dynamics is implemented in software. Figure 13shows the basic software
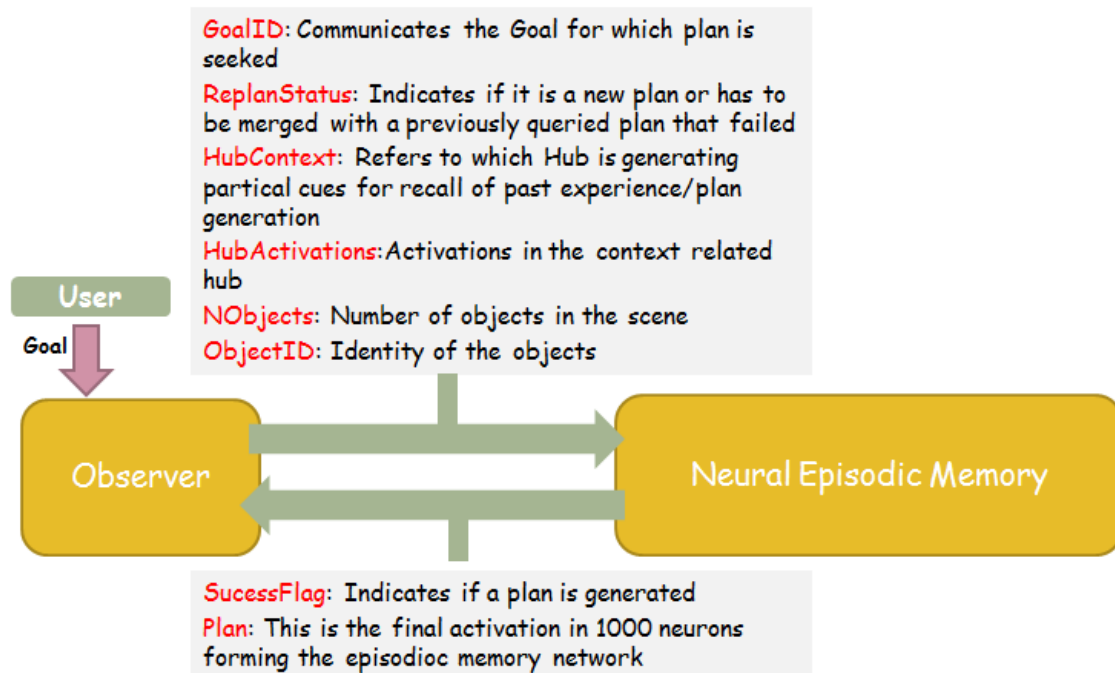
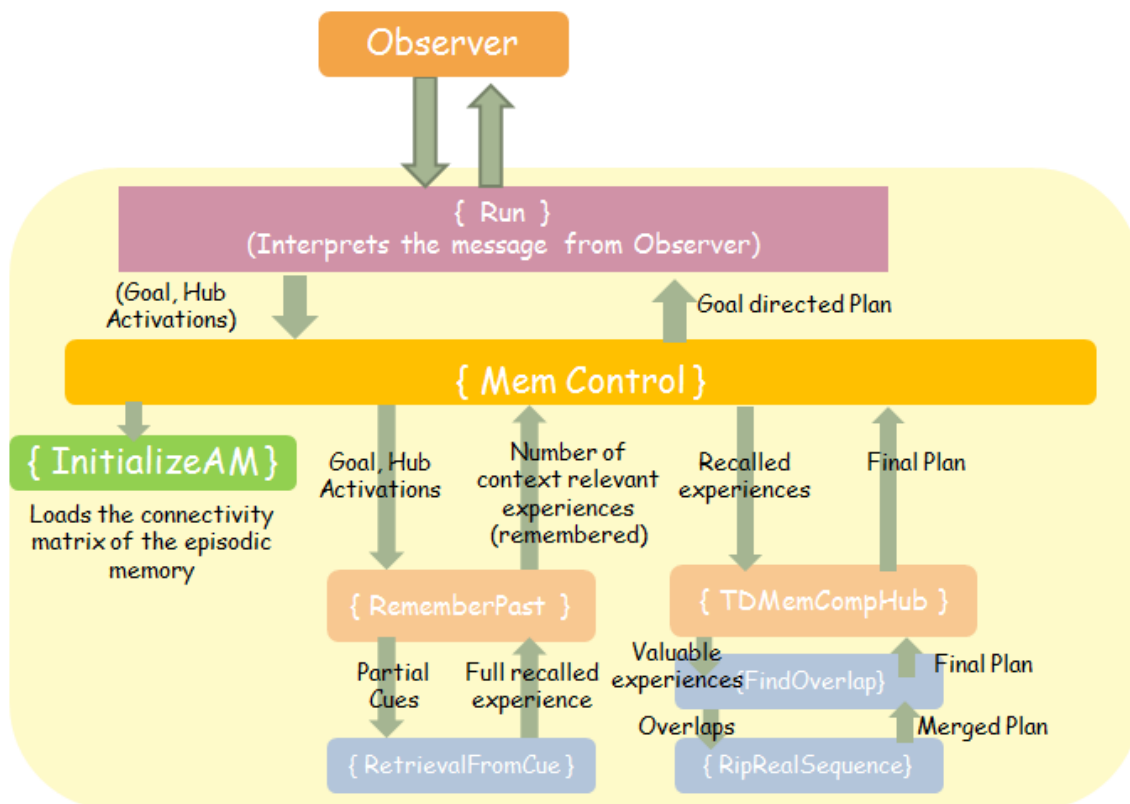Figure 13: Basic interface between Observer and Episodic memory module

Figure 14: Internal structure of the episodic memory module with central functions that implement the neural dynamics and interfaces between them

structure inside the episodic memory module, listing some of the central functions and internal information flow between them, that ultimately leads to the synthesis of the plan. Below we describe each function in detail, connecting the computational model of the neural network with the implementation in software.

Run: This function implements the basic I/O interface of the episodic memory module and the Observer. Receiving the inputs related to the present context (i.e. intended goal, objects available in the scene), it triggers the MemControl function (i.e. Memory Control).

MemControl: This coordinates the overall plan formation and reasoning process: involving the recall of context related experiences, choosing the most valuable among multiple remembered experiences, synthesis of a possible goal directed plan that can be executed in the present, anticipation of the future consequences, which is then communicated to the Observer. This functionality involves complex neural dynamics whose implementation is distributed across several other functions that we describe below.

RememberPast: As the name suggests, this function implements the computations necessary to recall context related past experiences encoded in the episodic memory (based on objects in the world, user goal communicated by the observer). This is done in two steps 1) Generation of partial cues that involves rearranging the serial information from the observer into the 20x50 matrix format that is used in the episodic memory; and 2) Triggering the dynamics of the neural network to reconstruct full experience from partial cues (this dynamics is implemented in the RetrievalfromCue function).

RetrievalfromCue: This function implements the basic neural dynamics of the episodic memory that allows the system to recall full experience based on partial cues (example: perceiving a Fuse in the environment and recalling the plan to assemble the fuse box set up and so on). The implemented dynamics in this function is as described in equation 4,

$$\tau_{rel} \overset{\bullet}{V}_k = -V_k + \sum_{j=1}^{N} T_{k,j} V_j + I_{inhib}$$

$$I_{inhib} = g(-\alpha^{in} + \beta \sum_k V_k)$$

$$g(i) = 0, if\,(i < 0), else, g(i) = i. \tag{4}$$

Where, V_K is the activity in the Kth neuron (in the 50x20 network). T is the connectivity matrix of the episodic memory (1000x1000: stored in WMems77N.txt). 'I' is the current coming from the inhibition network that is modeled as a single neuron. The function of the inhibitory network is to keep the excitatory system from running away, to limit the firing rate of the excitatory neurons. At low levels of excitation the inhibitory term generally vanishes. For all experiments in relation to Darwin α^{in} was chosen as 30, τ_{rel} as 1000 and β as 3.5. The effect of changes in these parameters and the performance of the episodic memory module are described in Mohan et al 2014. Finally, the output of this

function is the complete set of remembered experiences (i.e. what the robot knows about the present situation or the issued user goal).

TDMemCompHub: While the dynamics implemented by the previous function extracts all the set of experiences known to the robot in a given situation, this function implements the competition between such recalled memories so as to pick up the most valuable plan to be executed. At present the winning memory among all those recalled is decided based on the "anticipated reward" that could be obtained by the robot if the remembered episodic memory (or a part of it) is reenacted to realize the present goal at hand. In this sense, this function selects the most valuable set of experiences that could be executed in the present situation, out of all that is recalled. In case there is only one winner, the plan is directly available, otherwise the valuable set of experiences are merged together in the FindOverlap function and RipRealSequnce function, to ultimately produce a plan to be communicated to the Observer.

In sum, the set of core functions in the episodic memory module realize the transformation from a request from the observer (i.e. Goal, Present situation) to a suitable plan to be executed. This involves generation of partial cues, recall of past experiences from the partial cues, choosing the most valuable plan in the present context that is implemented the different functions described.

7.3 The Observer Module

In the previous sections, we briefly outlined the implementation details related the neural networks associated with the episodic memory module (storing experiences, generating plans) and the PMP module (that implements forward/inverse model of action).This section describes the third critical component i.e. the observer, that basically functions like a central executive, communication both with the user and other modules (vision, PMP, Episodic memory, Grasp) to ultimately realize the goal requested by the user. As seen in figure 15, the software implemented in the observer can be broadly categorized into four groups each involving an interrelated group of functions that implemented the requisite behavior:

a) Implement the interface to the user to communicate via proto language
b) Maintain a dynamic internal representation of the object, action and body state as the plan evolves in time and implement the dynamics necessary to activate the associated hubs (Object, Action and Body)
c) Implement the interfaces with other core modules related to Vision, Action, Grasp affordance, Episodic memory in order to jointly realize the user goal, monitor the status of execution and take corrective actions (contact reasoning, explore, seek user help etc).
d) Working memory: As the plan evolves, the observer is the converging point for different kinds of information related to Objects in the scene (what, where), micro actions being executed with the outcome, the plan coming from the episodic memory and the present status of the plan execution. Move over this information is not static, but keeps changing as the plan evolves. The working memory holds such information during the

lifetime of the goal, allowing other functions in the Observer to access it as necessary.
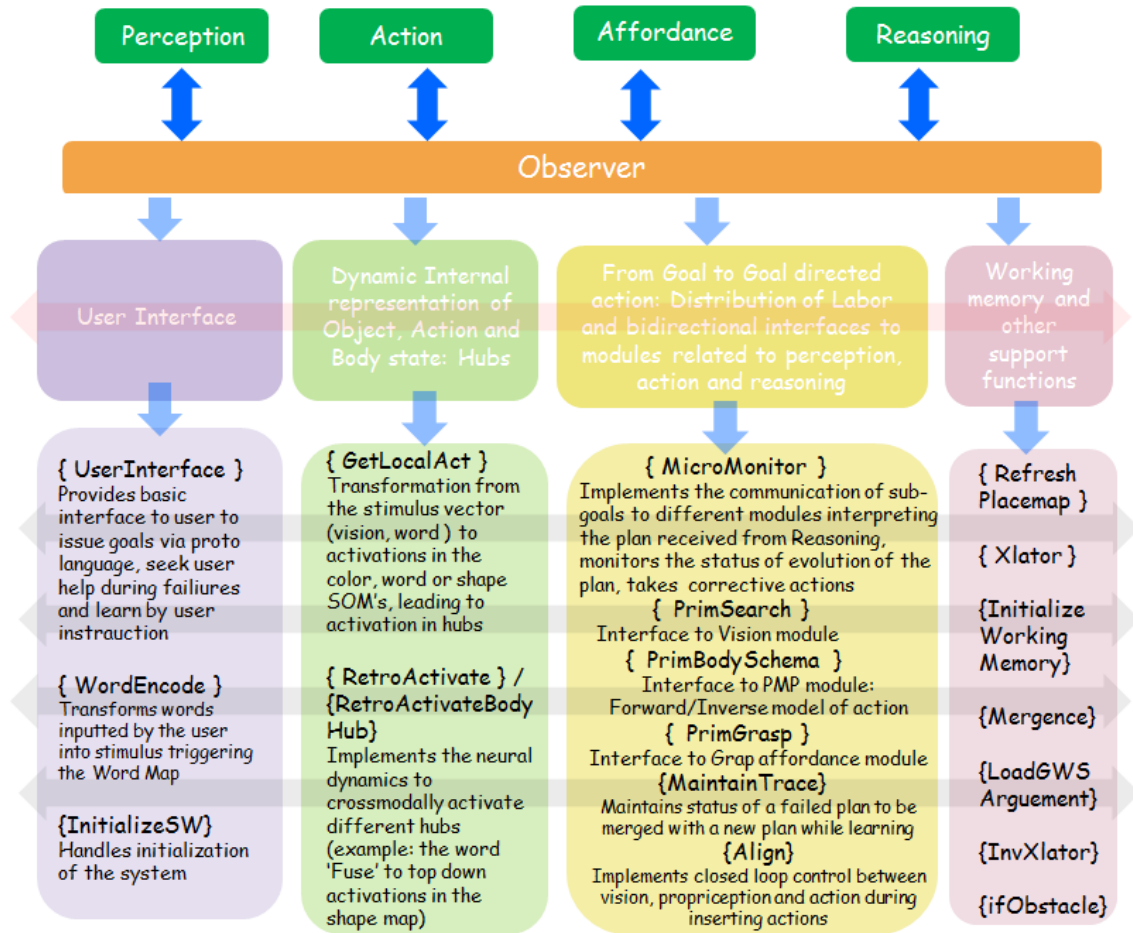

Figure 15. Internal organization of functionality implemented in the observer

Below, we go into details of each of the core functions in the Observer that collectively implement the executive control in Darwin. For simplicity we break it down into two sets: a) Hub related functionality and 2) Monitoring and execution.

7.3.1 Hub related functionality

UserInterface, Wordencode: Presently, there is a preliminary Proto-Language mechanism to facilitate interaction with the user, particularly in terms of issuing goals linguistically (Ex: Reach Fuse, Assemble xyz, Push Fuse Box etc), with extensions enabling the user to teach new assembly plans (i.e. provide the initial recipe, this is work in progress when the document is being written) or alternatively provide the robot a way to seek user intervention (when thing fail and no past experience exists to come of the situation). Basically in this

mechanism we deal with nouns and verbs. User interface deals with verbs and wordencode deals with nouns, using the same mechanism but the former triggering the action hub (consisting of 12 neurons) and the latter the word map (consisting of 42 neurons). To this effect, firstly "word" inputs (i.e. sequence of alphabets) entered by the teacher using the key board are converted into stimulus vectors on the basis of letter usage frequencies in English language as is done in (Hopfield, 2008). The stimulus vector in the case of action verbs forms the feed forward input to Action hub and in the case of nouns froms the feed forward input to Word map.The transformation from stimulus vector to activations in the corresponding maps is done using standard SOM procedure and is implemented in the next two functions described below.

GetLocalAct: As the name implies (Get local activation), this function implements the transformation from the stimulus vector (generated by the previous function) to neuronal activations in the corresponding maps using the standard SOM method. Before describing the underlying dynamics, we summarize the details of the neural network as existing in the present software.
Let N be the number of neurons in any SOM and S be the dimensionality of the bottom up stimulus feeding the map. Then the connectivity matrix has a dimensionality of N x S. Since we are dealing with multiple maps here, for clarity we address N_C, N_S, N_W, N_H , N_B , N_A as the number of neurons in the color, shape, word, Object hub, Action hub and Body Hub respectively. In the present implementation N_C=30, N_S=30, N_W=30, N_H=42, N_B=42, N_A=12 neurons respectively. This choice was made also in relation to the structure of the neural episodic memory (i.e. consisting of 1000 neurons arranged in a sheet like structure 20x50) that was described in section 7.2. Only the activations in the hubs (object, action and body) enter the episodic memory (and not color, word or shape that just feed the Hubs). The biological inspiration for such a structure is described in detail in Mohan et al 2014. Since color, word and shape SOM activity forms the bottom up input to the object hub as shown in figure 13, the connectivity matrix of the Object hub has a dimensionality of N_H x (N_C+N_S+N_W). To transform input stimulus into activations in the SOM, standard method is used. Basically, a Gaussian kernel compares the sensory weight s_i of neuron i with input stimulus vector s, to determine the activation of each neuron (S_i) in the map according to equation 6.

$$S_i = \frac{1}{\sqrt{2\pi}\sigma_s} \, e^{\frac{-(S_i-S)^2}{2\sigma_s^2}} \tag{6}$$

where in all the cases (word, action, shape etc), $\frac{1}{\sqrt{2\pi}\sigma_s}$ is empirically chosen as

5.64 and $2\sigma_s^2$ is chosen as 0.25. The result is then normalized. In this way, we transform for example, a word "Fuse" into a stimulus vector (implemented by the previous function) and then from stimulus vector to activations in the Word SOM. The same method is used for the case of action word "Push", leading to

activations in the action hub. Further formal details can be found in Mohan et al (2013, 2014).

RetroActivate-RetroActivatBodyHub: While the former function deals with the transformation from input stimulus to activations in the associated SOM, this function implements the dynamics that allows activations in one map to cross modally activate other maps. For example, the word "fuse" inputted by the user activates the word map, but these activations in the word map can corssmodally activate the shape map (i.e. anticipating what to expect from vision in relation to the word fuse). And finally activations in the shape and word map, leads to activations in the object hub. While the function Retroactivate implements the dynamics between color, word, shape and object hub, the function RetroActivateBodyHub implements the dynamics between body hub and action hub. A illustrative example is given in section 7.4. Note that only information from the hubs (object, actions and body state) go to the episodic memory and not (lower level information related to color, word, shapes).
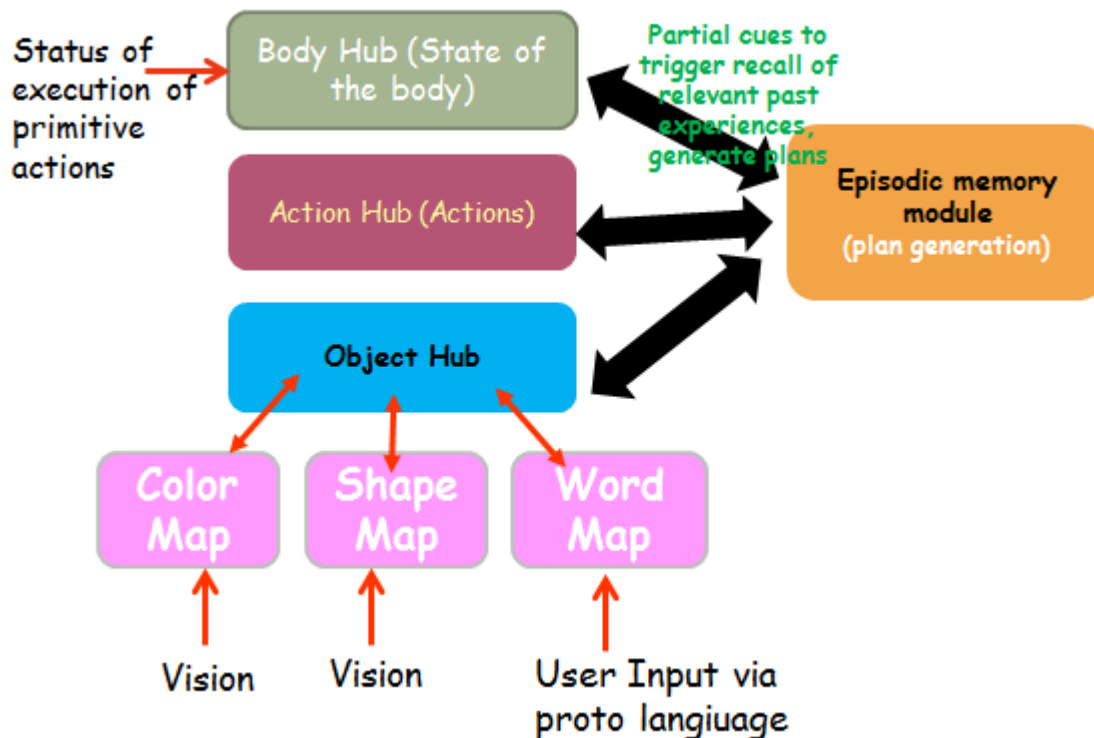
Figure 16: Pictorially depicts the link between hubs and the episodic memory module. Color and shape maps are triggered by the output of the vision system (at present, color vision is no longer used for the industrial platform). Word map receives input from the user (entered through keyboard). Body Hub represents the present status of the body, based on the outcome of execution of primitive actions as the plan evolves. Note that only information from the hubs (object, actions and body state) go to the episodic memory module: to recall past experiences (what could be done with an object, plan to realize some action, or plans in the context of the state of the body)

The network dynamics implemented in the Retroactivate functions builds upon the idea of neural fields (Amari, 1979) and supplements it with novel concepts (Mohan et al, 2011) and is given by equation 7 and 8. Let h_i be the activity of the ith neuron in the hub and x_{prop} be the activity of a neuron in any of the property specific SOM's connected to the object hub (example, color, word and shape SOMs). Let $W_{prop,hub}$ encode the connections between the property specific maps and the provincial hub. Basically, $W_{prop, hub}$ is a N_{PH} x (N_C+N_S+N_W) matrix learnt as described before. Its transpose encodes backward connectivity from hub to individual maps. The network dynamics of hub neurons and neurons in the property specific SOM are governed by equations 1 and 2 respectively:

$$\tau_{hub}\,\dot{h}_i = -h_i + (1-\beta)\sum_{i,j}(W_{prop,hub}X_{prop} + \beta.(Topdown))$$

(7 and 8)

$$\tau_{prop}\,\dot{x}_{prop} = -x_{prop} + (1-\beta)S_{prop} + \beta.\sum_{i,j}(W_{hub,prop}h_{hub})$$

Where,

$$S_{prop} = \frac{1}{\sqrt{2\pi}\sigma_s}\,e^{\frac{-(S_i-S)^2}{2\sigma_s^2}}$$

The instantaneous activation of any neuron in the hub or the property specific maps is governed by 3 different components: The first term induces an exponential relaxation to the dynamics. The second term is the net feed forward (or alternatively bottom up) input. The third component is the top down component: for the property specific SOM's the top down input comes from the hub to which they are connected. Further details of the implemented dynamics can be found in Mohan et al 2013, 2014.

To summarize, the set of functions described so far implement the basic functionality related to hubs that dynamically keeps changing as the plan evolves. The activations in the hub forms the input to the episodic memory module, that recalls known past experiences (what could be done in relation to an object, action or body state) and generates plans to be executed in the present. The next section describes the counterpart i.e. functions that implement monitoring and execution based on received plans.

7.3.2 Monitoring and Execution

MicroMonitor: This function is responsible for the detailed implementation of the plan coming from reasoning, communicating micro goals to other modules (Vision, PMP, Grasp, Align/Insert, Interrupt, Learn new behavior, form new memory), receiving feedback of the outcome, contacting episodic memory in case of failures, maintaining full trace of the evolution of the behavior, which in case of cumulative learning is encoded as a new memory in the episodic memory neural network (section 7.4, gives an example of this formation of new memory via learning).

RefreshPlacemap: This function implements the interface with vision, and on request gets the most updated information related to the scene in front of the robot (what, where). This information is stored in a data structure called PlaceMap that is a component of the working memory, to be accesed by other primitive functions (example: 3D location of an object of interest is required by the PMP, which objects are there in the scene is required for reasoning).

PrimSearch: This function searches for specific requested object in the scene (as per the evolving goal directed plan), using the latest information stored in the PlaceMap. If the desired object is found, its relative location in the place map is sent back, along with a flag representing the outcome. It also handles cases where multiple objects of the same class (i.e. requested) are available in the scene.

PrimReach: This function implements the basic interface with the PMP module as described in figure 7.
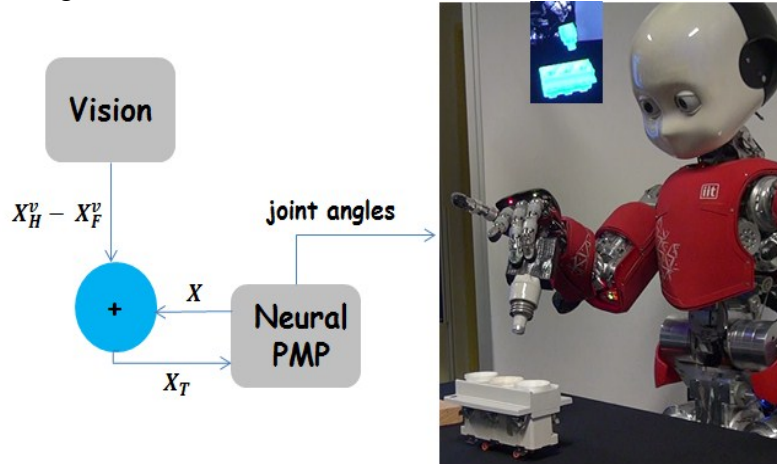
Figure 17 shows the loop of integration between vision, motion and proprioception. Using information from vision the distance between the location of hole X_H^v and the location of fuse X_F^v is added to the current location of the end-effector (from the forward model of neural PMP) to set a new goal for the end-effector. The alignment continues till vision estimated distance between the fuse and hole < 9 mm.

PrimGrasp: This function implements the basic interface with Grasp module.

Align: This function implements the closed loop control between vision, proprioception and action in order to align the fuse to the hole, in order to complete the insertion, taking into account inaccuracies in both pose estimation and reaching. The bottom up input from vision, and the actual position of the end effector (coming from PMP) is iteratively integrated such that the alignment between the tip of the fuse and the center of the hole is <9mm.

7.4 An illustrative example: Merging a failed plan with explorative actions to gain new experience

Here we explain the whole loop consisting of firstly the robot failing to realize the user goal, with no experience conducting exploration, forming a new memory and updating the episodic memory weights and using this experience in a new situation. For starting from the scratch with only one basic reach schema, set numepi (in the config file *\darwin\trunk\modules\iit\app\perceptionActionCycleApp\numepi.txt*) as 1. We now describe the process of cumulative learning, formation and recall of new memories. Start the Yarp server, place a fuse or any other recognized object in the scene. Run all the modules: Observer, Episodic memory, OPC, Vision, PMP, Grasp. Note that now there is only one primitive schema in the episodic memory and we will illustrate the process of how more experiences are learnt, stored and exploited in novel situation.

With the set-up, in the first trial Type: "Rea Fuse" as the user goal. The observer contacts the episodic memory and receives a plan (the basic existing schema for reaching). Contacting other relevant modules, the robot should reach the object and realize the goal. Now we take an unexpected case where the user removes the object from the scene, or requests the robot to reach any other object that is not present in the scene. In this situation, the goal cannot be realized directly.

Figure 18. Shows the console outputs the user must expect in the observer and episodic memory modules when the system is triggered with a user goal, and only one primitive schema present in the episodic memory.

Figure 18 shows the resulting behavior. The basic plan is received from the episodic memory as in the previous case and the robot begins to execute the basic plan. However "Search" fails (Figure 18:Top Panel), the Observer contacts the episodic memory to reason further (Figure 18:Middle panel) , but since there is no past experience to deal with failure situations the *episodic memory returns a NULL plan (Figure 18:Bottom panel). So this is an example where the past experience is not sufficient to realize the user goal, and the robot has to explore and learn/form new memories to intelligently resolve such situation, use such experience intelligently in the future.*

Figure 19: Top panel shows the failure of the basic plan, middle panel shows the communication sent by the Observer to the episodic memory querying for a revised plan. Bottom panel shows that the episodic memory module returns a NULL (since no relevant experience exists). The robot has no option but to engage in possible explorative actions.

The robot can quit the goal, report the error to the user and wait for user help, recall what else is known form the present situation (which is not valid because the episodic memory has already retuned a NULL). Randomly, instead of quitting the robot seeks user help and waits for user to respond. The user puts the fuse in the scene and enters "1" (see figure 18, bottom panel), signaling the robot to proceed further.

Figure 19, shows the sequences of events, starting from merging the explorative action "seek user help" with the previously received plan that failed. Now the

search succeeds, PMP is triggered and the object is reached, hence realizing the user goal. Now, the robot also has a novel experience of the sequence of events that occurred starting from failure to realize the goal, NULL returned from episodic memory, engaging in an explorative action, response form the user , reinitiating the goal and fulfilling it successfully. At present the robot issues itself a *"self-reward"* based on the number of actions needed to realize the user goal (the lesser: greater the reward, basically encouraging the robot to solve the user goal in minimum number of steps).

Figure 20. Shows the console output, when the robot seeks user help, merges the explorative action with a previously failed plan coming from the episodic memory to gain new experience. Note that in the present system (customized for assembly), note that the robot computes its reward directly (based on the number of action events needed to realize the goal). Alternatively, the rewards can also be inputted by the user for a different task setup through simple modifications in the source code of the Observer.

7.5 Encoding the new experience in the neural episodic memory

The new experience of recovery from failure is stored as a new episodic memory that can be recalled in the future based on partial cues. Now we outline the process of how this new episode of experience is stored in the neural connectivity matrix of the episodic memory. Present experience of the robot is tracked continuously by the Observer and is stored in *\perceptionActionCycleApp\ThePresent.txt (see figure 21, left panel).* To update the weights of the neural episodic memory with the new experience, run the script "TrainingEMPresent.m" also located in the same directory. The script firstly transforms the content stored in ThePresent.txt, into the appropriate
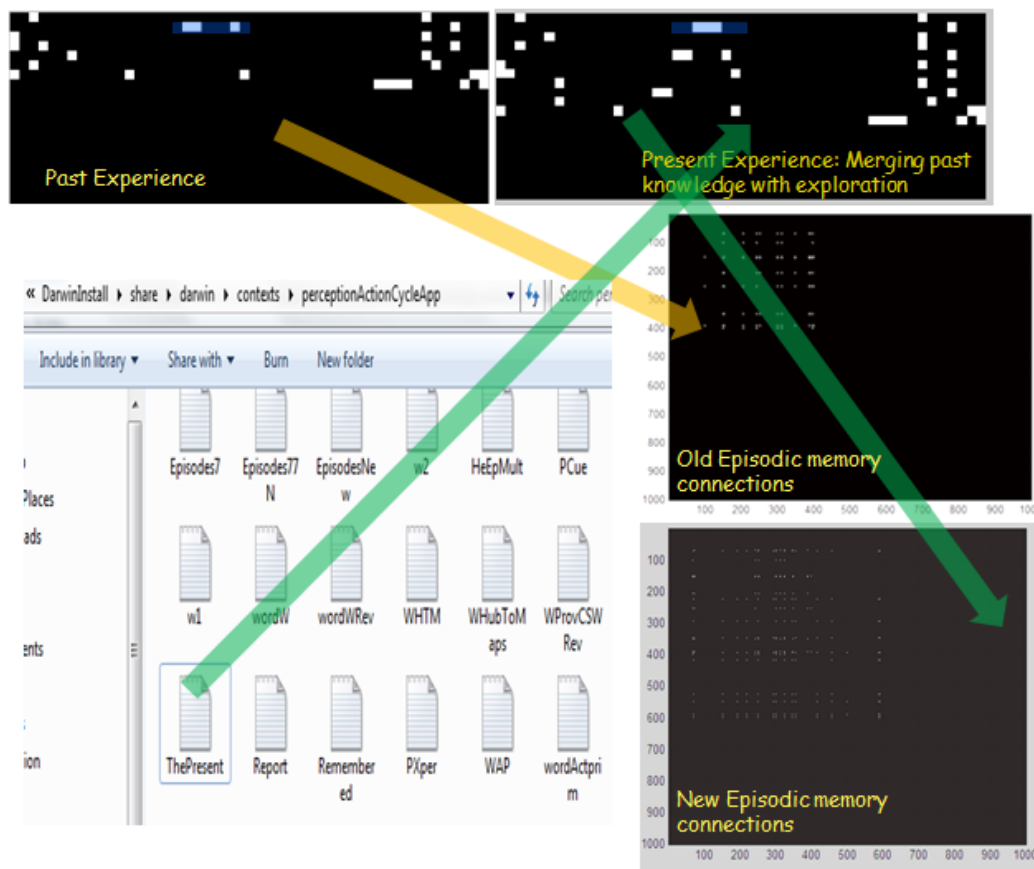
Figure 21.Bottom left panel shows the file generated by the Observer that encodes information related to the present experience of the robot. Top panels show the difference between present and past experiences. Note that the present experience has greater events because of merging explorative actions with past plan: hence resulting in a greater reward (in the present framework, reward=energy that the robot seeks to minimize in its behaviors: see D4.4 for details). Bottom right panels show the difference in the neural connections in the episodic memory (the growth in neural connectivity as a result of encoding new experience can be visualized here).

format on the episodic memory i.e. a network of 1000 neurons arranged in a 20x50 sheet. Running the script expect to see in figure 1 the present experience gained by the robot encoded in this format (figure 10 top right panel). Secondly, the script updates the neural weights of the episodic memory taking this new experience into account. Expect to see a figure 100, showing the updated neural connectivity. It encodes values of 1000x1000 connections (figure 10, bottom right panels show the old and new connections). Thirdly the script automatically generates the new weight file with the name "WeightNO.txt". Expect this file to be bulky as it

stores 10^6 numbers. We have named this file differently, so that the user can choose if he wants to use the new updated episodic memory or the old one. In case the user chooses to use the updated episodic memory, just rename "WeightNO.txt" to "Weight77N.txt". The episodic memory module (TheNewEpim.exe) when initialized does all the necessary further actions to use this new experience. Just in case the user wishes to test the dynamics of recall, i.e remembering the experience from a partial cue we also provide another script "RememberingfromPartialCue.m", running which you should see the full experience reconstructed from partial activations in the episodic memory patch. Running this script is not mandatory, only for checking the dynamics of the episodic memory in the inverse situation: i.e from storing to remembering.

7.5 From Storing to remembering: Exploiting the newly gained experience

In the previous sections, we saw how failure while realizing the user goal, led the robot to explore, have a new experience that is encoded in the neural episodic memory by updating the connectivity matrix. Now we illustrate how this memory is exploited in a novel situation. Run all the modules as before. Keep no object in the scene and issue a user goal "Rea FBox" in the Observer console (this means the user is requesting the robot to reach the fuse box: another object presently recognized by the vision system). Figure 22 shows the console outputs from the Observer and the Episodic memory modules, that the user must visualize when triggering the Darwin system with the updated neural episodic memory (learnt in the previous section). As seen, the basic plan still fails as there are no objects recognized in the scene. The observer sends the failure information to the episodic memory module (green arrows), that now generates partial cue related to the information communicated by the observer. The new updated weight matrix allows the reconstruction of full context relevant past experience from the partial cue (brown arrows). Because a relevant past experience was remembered, the episodic memory does not return a null plan (as in figure). So now the robot knows something to recover from the failure with a new sequence of actions. Note that this process of remembering occurred only because of the present context (the object fuse box not being there in the scene). In this sense experiences are remembered only based on present context and as and when necessary. Recall of the new experience based on the present situation allows the robot to now realize the user goal without any exploration.

After the goal is realized, the system should terminate and wait for the next User goal.
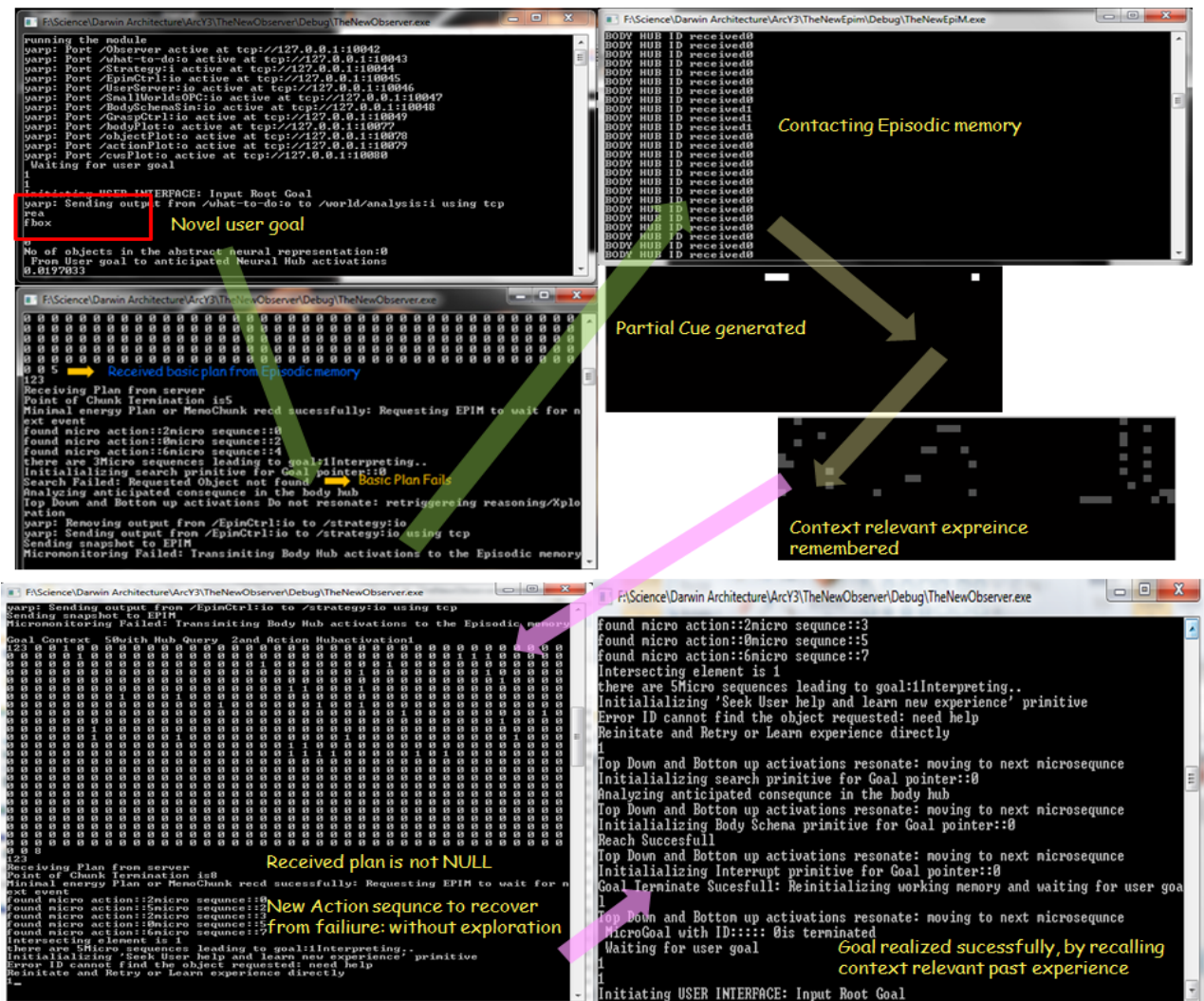

Figure 22. Panels show the console outputs the user must visualize while running Darwin architecture with the updated episodic memory learnt from past explorative experience. Note that the received plan while recovering from failure is not NULL (as in figure 8), and the robot realizes the user goal without any explorative actions. This is a result of the growth in the neural episodic memory due to new learnt experiences and how they are exploited by the reasoning system in a context relevant fashion.

  In summary, we illustrated the whole loop of failure in plans, the robot engaging in exploration, merging explorative action with a past plan, forming a new experience, storing this experience in the neural episodic memory and then remembering it in a different context to realize the user goal.

7.6 Encoding Assembly plans in the episodic memory through Observer proto-language

While the previous sections described an example of how a failed plan can be merged with explorative actions and ultimately be encoded in the episodic memory, this section outlines another alterative i.e. teaching assembly plans through the User-Observer proto language. This mechanism facilitates flexible user-Darwin communication allowing swift change over to new assembly tasks using linguistic communication. The instructions by the user are transformed into neural activations (in the hubs), hence forming an episodic memory trace. Once the assembly is encoded in the episodic memory, in the future issuing the goal results in robot executing the behavior (with the flexibility to combine with other experiences already encoded in the memory). The assumption here is also that novel objects are learnt offline by vision giving then a specific descriptor, but the assembly plan is learnt online through proto language. We illustrate this with an example of user teaching the robot the fuse box assembly, outlining the sequence of steps to be followed:

1. Start the Yarp server,Run all the modules: Observer, Episodic memory, OPC, Vision, PMP, Grasp.
2. To input a new assembly plan, type NewAsm in the observer console (figure). The system will initialize to cater this functionality and ask to enter the intended assembly goal (figure 23, panel 1).
3. The plan when described in proto-language is a sequence of verbs and nouns (i.e. sequence of actions on objects). Verbs generally remain the same (though the sequence can change) and nouns vary based on the objects (Ex: if the goal is Assembly, it may be assemble fuse box, assemble Big Fuse box etc., with internal sequences of actions with other objects). To teach the fuse box assembly, type 'Asm' (standing for assembly) and "Comf" (the label associated with the composite fuse box: i.e. the final goal). The words are transformed into neural activations in the hub as described in section 7.3 using the dynamics implemented through userinterface, getlocalact and retroactivate funtions. Once the root goal is encoded, the system asks for instructing the desired actions on objects to be performed to realize the assembly (Figure 23 panel 2).
4. Now type the sequences of actions to be executed by the robot (pick and place the fuse on the fuse stand). For this the user inputs one action verb (pick and place) and two nouns related to the referred objects. These words are transformed into neural activations using the same procedure described earlier. The system before terminating asks for the fetched reward (Figure 23: panel 3).
5. On entering the reward, communication terminates, the set of instructions inputted through proto language is transformed into an episodic memory trace (i.e activations in 20x50 memory network described in section 7.2). Also the new memory trace can also be found in *\perceptionActionCycleApp\ThePresent.txt.* To update the weights of the neural episodic memory with the new experience, run

Figure 23: Procedure to instruct an assembly plan via User Observer proto language to be encoded in the neural episodic memory

the script "TrainingEMPresent.m" also located in the same directory following the steps outlined in section 7.5 that is standard for storing any new experience in the episodic memory network (weather it is learnt by exploration or through user observer proto language).

8 Concluding remarks

The document concisely outlines how a prospective user can install, run and use the Darwin architecture. Multiple aspects ranging from basic dependencies, preparatory steps to be taken to install and run Darwin to advanced aspects related to learning the body schema for any robotic embodiment, engaging in exploration during failures, formation and recall of episodic memories, details of the implementation of the neural network in software are summarized: with sequences of actions that the user needs to take, expected snapshots of console outputs and the resulting behavior of the robot. The document at several places includes additional links that the user can access for further information. Further details on theoretical aspects, advancement to the state of the art can be found in related references [1-3]. For further details on the software framework, troubles in installation, bugs, suggestions for improvements the interested user is requested to contact (ajaz.bhat@iit.it, Sharath.akkaldevi@profactor.at, vishwanthan.mohan@iit.it).

[1] Mohan V, Morasso P, Sandini G (2014). A neural framework for organization of episodic memory in cumulatively developing baby humanoids, Neural Computation 26(12), pp 1-43, MIT Press. Doi:10.1162/NECO_a_00664..

[2] Mohan V and Morasso P (2011) Passive motion paradigm: an alternative to optimal control. Front. Neurorobot. 5:4. doi: 10.3389/fnbot.2011.00004.

[3] Mohan V, Morasso P, Sandini G, Kasderidis S (2013) Inference through embodied simulation in cognitive robots. Cognitive Computation,5(1).