Team-NUST

Project Documentation - 2017

Saifullah, Asbah Ashfaq, Muhammad Talha, Imran Abdul Rehman, Maham Tanveer, Shams ul Azeem, Abdul Haseeb Ayub, Umair Hasan, Idrees Hussain

Dr. Yasar Ayaz, Dr. Muhammad Naveed, Fahad Islam, Saadia Qamar

Robotics & Intelligent Systems Engineering Lab (RISE),
School of Mechanical and Manufacturing Engineering (SMME),
National University of Sciences and Technology (NUST), Islamabad, Pakistan

Revised: Dec 24, 2017

Chapter 1

Introduction

1.1 About Team-NUST

Team-NUST was established formally in 2013 with the aim of carrying out research in the rapidly progressing field of humanoid robotics, artificial intelligence, machine vision, motion planning, kinematics and navigation; with the motivation to participate in RoboCup Standard Platform League (SPL). We are working on robust and predictable kicking motion, multi objective behavior coordination, motion planning, situational awareness based on efficient perception and robust probabilistic multiagent localization. The team is working in RISE Research Center, part of SMME, NUST, with publications in the field of cognitive robotics, machine intelligence focused on design, control and motion planning for robotics systems including mobile robots, humanoid robots, multi legged robots, intelligent bionics and robotic manipulators.

Chapter 2

Getting Started

This chapter will give a brief overview of the basic components along with the instructions on how to setup the working environment, to build the code and run it on the robot NAO. The instructions given assume that the reader has a sufficient knowledge of Linux OS, CMake and software toolchains. Downloading a few software packages is also required as a prerequisite to compile the code. Furthermore, the code can be executed in simulation, through remote-robot connection (will soon become obsolete), or directly on the robot.

2.1 Code basics

1. **Src:** This folder contains the main source code of the software.
2. **Utils:** This folder contains the necessary utility header files made for ease of code generation, third-party libraries, and simulation tools.
3. **.qi:** The qi folder contains the working tree for the **naoqi (Aldebaran Software for NAO)** provided toolchain which is necessary for building and compiling the code. The main directory also contains **qiproject.xml** file which is used to hint the naoqi build utility (**qibuild**) the name of the project.
4. **Config:** This folder as the name suggests consists of various configuration settings for different modules used in the main source code.
5. **build-TC:** The build folder is made once the source code is configured with the toolchain of name **TC** (This name is assigned by the user). Once the code is compiled on the system, the **remote module** (read on remote and local modules of naoqi at http://doc.aldebaran.com/2-1/dev/naoqi/index.html) executable can be found at **build-TC/sdk/bin** and in case of a **local module**, a shared library (.so) file is generated at **build-TC/sdk/lib**. As the code compilation is currently based on **naoqi**, the remote modules built by our software can run on the robot through a remote connection (**ssh**) and the local modules by running the shared library (.so) file onto the robot as one of the **naoqi**'s own modules.
6. **CMakeLists.txt:** This file defines the building procedure of the code such as including directories, defining environment variables, linking libraries. It uses **qibuild** package to build and compile the code.

2.2 Building the code

2.2.1 Prerequisites

1. **Ubuntu (14.04.5 LTS Trusty Tahr)** - Currently our software is only supported for Linux distributions, and the use of Ubuntu (14.04.5 LTS Trusty Tahr) OS is recommended.
2. **Qibuild:** As our software's compilation is based on naoqi's toolchains, the following software packages are necessary for the required tasks. Following commands can be executed for easy installation:

```
sudo apt-get install python-pip && sudo pip install qibuild
```

3. **Naoqi-sdk-2.1.4.13-linux64/32:** For building and running the code as a remote module.
4. **Simulator-sdk-2.1.2.17-linux64/32:** For simulating the robot based on **naoqi**.
5. **ctc-linux32-atom-2.1.4.13:** For building with cross-compilation as a local module and running the code directly on the robot.
6. **VREP_Linux:** The V-rep simulator for running the code in simulations on a local system.

2.2.2 Environment Setup

1. To be able to configure the code, the following environment variables are needed to be set:

```
PATH_TO_TEAM_NUST_DIR,     PATH_TO_VREP_DIR,    PATH_TO_SIM_DIR
```

2. The variables can be set by the following terminal commands with `/path/to/` is the path to the given directory on your pc:

```
echo 'export PATH_TO_TEAM_NUST_DIR=/path/to/team-nust-dir' >> ~/.bashrc
echo 'export PATH_TO_SIM_DIR=/path/to/simulator-sdk' >> ~/.bashrc
echo 'export PATH_TO_VREP_DIR=/path/to/vrep' >> ~/.bashrc
```

2.2.3 Compilation

1. The first step is to setup the toolchain with which we are supposed to compile our code. Configuring a toolchain has the same procedure for any type of toolchain but which toolchain is used to compile the code depends on whether the code is to be built for remote usage, local usage or simulations.
2. **Remote:** The following commands can be used to set up a toolchain for building our software without having to run on the robot.

```
qitoolchain create REMOTE_TOOLCHAIN /path/to/naoqi-sdk/toolchain.xml
qibuild add-config REMOTE_TOOLCHAIN -t REMOTE_TOOLCHAIN
```

3. **Local:** The following commands can be used to set up a toolchain for building our software for running directly on the robot:

```
qitoolchain create LOCAL_TOOLCHAIN /path/to/ctc-linux/toolchain.xml
qibuild add-config LOCAL_TOOLCHAIN -t LOCAL_TOOLCHAIN
```

4. We first need to initiate the qibuild working tree in our team-nust-spl main directory:

```
qibuild init
```

5. To build the code as a local or remote module we need to configure our **CMakeLists.txt** file by setting the flag **MODULE_IS_REMOTE** to **ON/OFF**. As remote modules can be run directly from our pc on the robot or in simulations, we generally need an executable file to run it. If the flag **MODULE_IS_REMOTE** is set **ON**, the compiler builds an executable and as in this case we do not need cross-compilation, we can use the toolchain derived from **naoqi-sdk** (with the name **REMOTE_TOOLCHAIN**).

```
qibuild configure -c REMOTE_TOOLCHAIN
qibuild make -c REMOTE_TOOLCHAIN
```

6. Additional configuration flags are defined in **CMakeLists.txt** which are used based on the required build config and must be set accordingly. Please see **CMakeLists.txt** for additional details on the configuration parameters.

7. If the code is to be compiled as local module, we can just set the flag **MODULE_IS_REMOTE** to **OFF** and cross-compile the code with the toolchain (of name **LOCAL_TOOLCHAIN**) derived from ctc-linux (a cross-compiling toolchain to build the code in compatibility with Robot OS).

```
qibuild configure -c LOCAL_TOOLCHAIN
qibuild make -c LOCAL_TOOLCHAIN
```

2.3 Working with Simulator

For the time being, simulations can be performed in V-REP which is an open-source simulation software. For this purpose, a scene is designed in V-REP based on the real robocup field. The robot models are already provided in the simulator which can easily generate most of the sensor data. We've also added additional sensors such as accelerometers, gyroscopes and updates to existing sensors such as calibration of camera output.

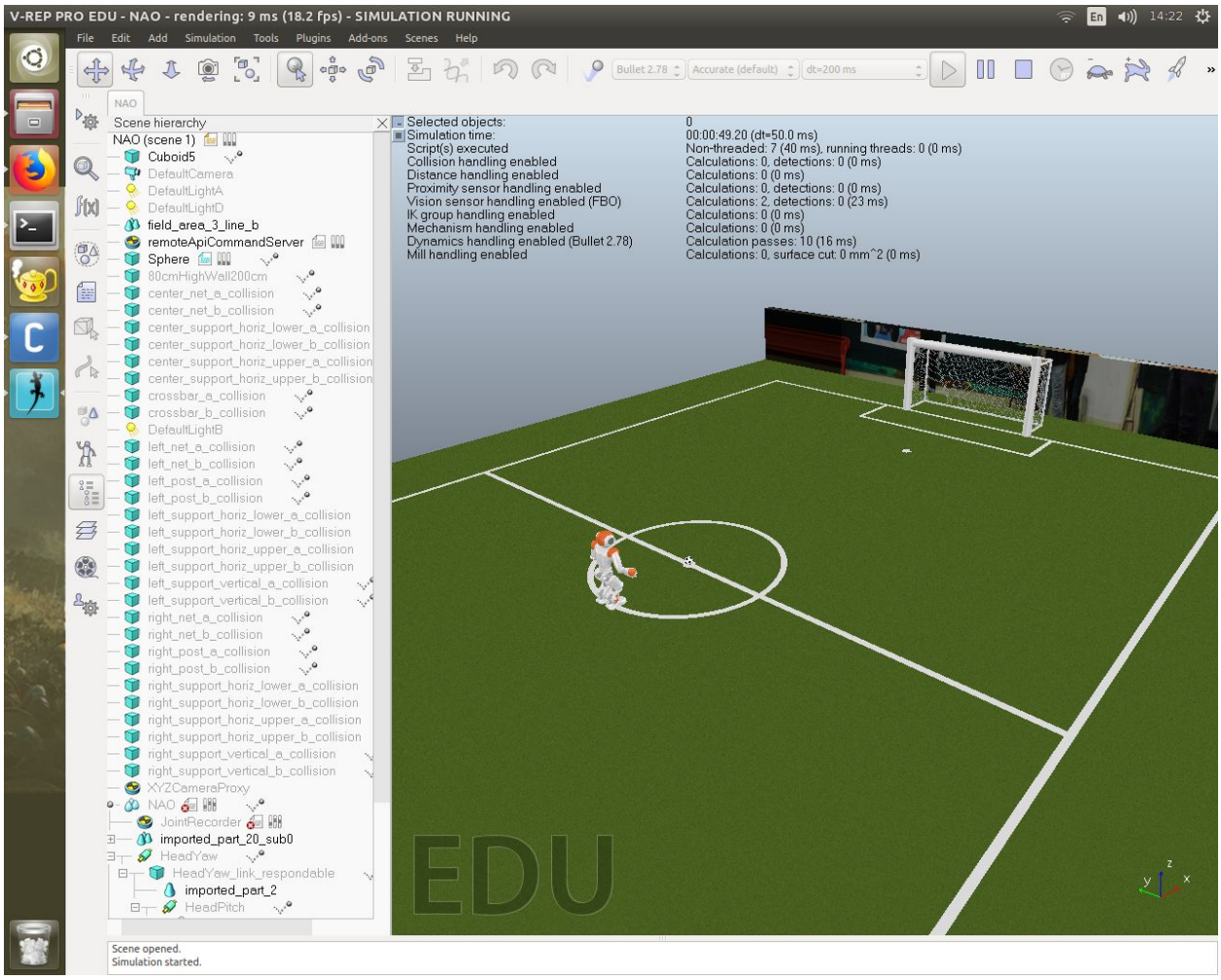

Fig 2.1. A snapshot of the simulation environment based on VREP-Naoqi interface plugin.

To counter the compatibility issue of running the same code on V-REP and on real NAO robot, we have made a V-REP/Naoqi interface plugin which can be used to update the naoqi's simulated robot from the V-REP sensor data. Similarly, the actuation data from the simulated naoqi's robot is sent directly to

V-REP and so at the end, our code can still be based on naoqi and run on both the real NAO and the simulator.

Following script can be used to start up the simulator:

```
cd PATH_TO_TEAM_NUST_DIR/Utils/Simulator
./simulation.sh
```

The remote modules can then be directly run and tested on the V-REP Simulated robot. However, the simulator is still not very good for actual gameplay with multiple robots as it is too computationally expensive and for that we might need to design our own lightweight simulator.

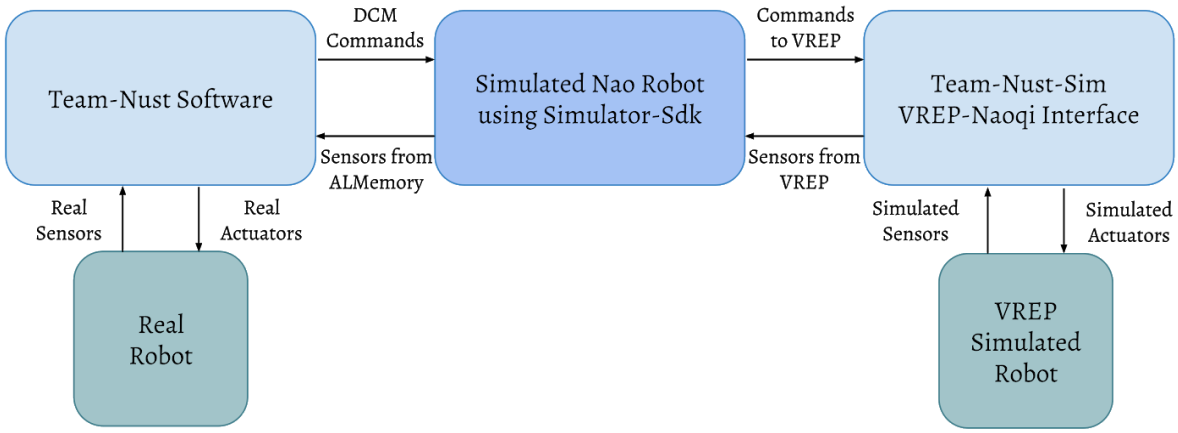

Fig 2.2. An overview of the interaction of the software with the VREP-Naoqi interface plugin and the simulator.

Chapter 3

Software Overview

This chapter will give a detailed overview of the software code along with the instructions on how to make updates to the code as a developer. The software is divided into several modules each dealing with a different aspect such as motion, vision, and planning. The infrastructure of the code also requires some basic syntax to follow for new code generation.

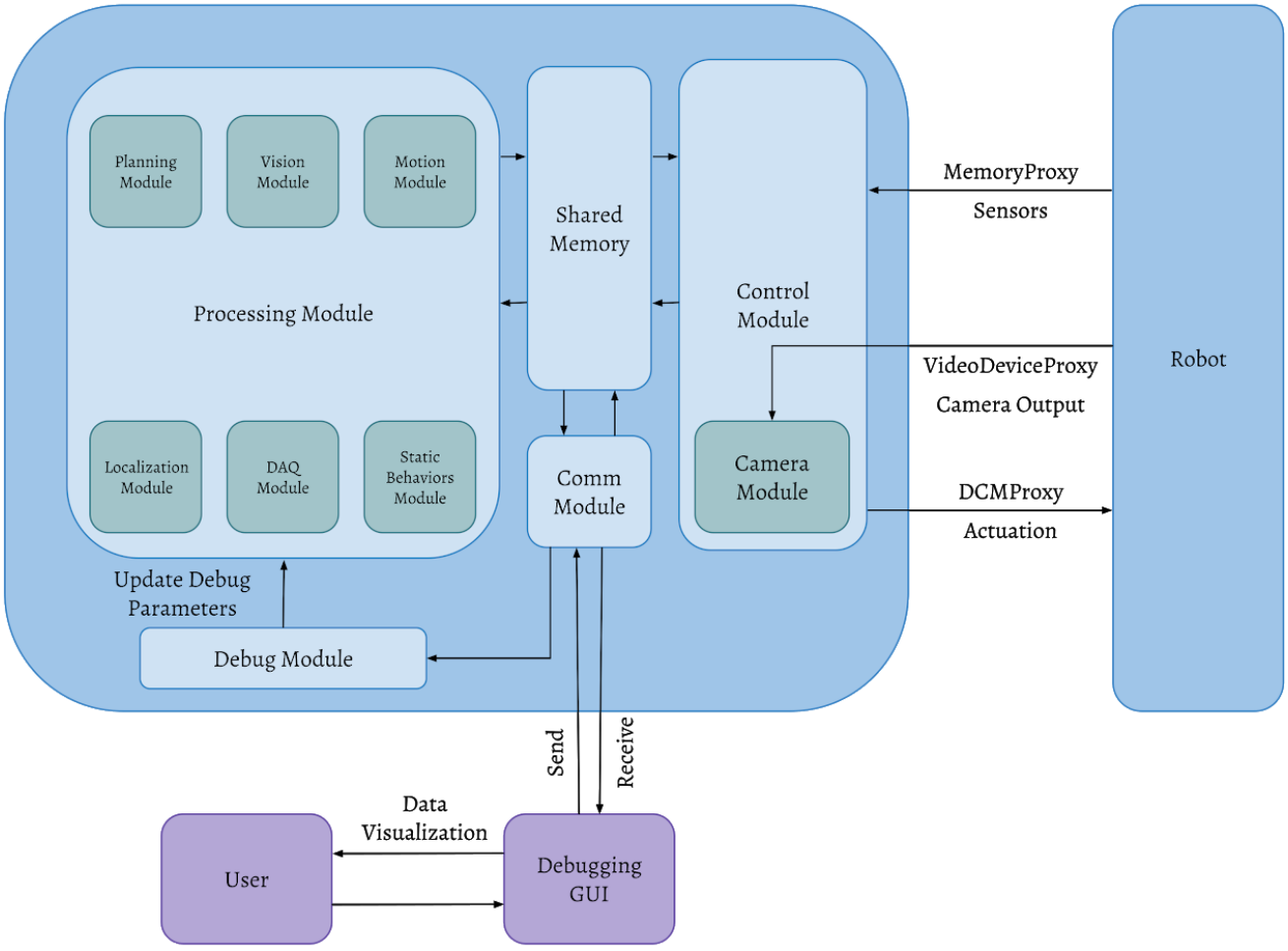

Fig 3.1. An overview of the interaction of different software packages with each other, the robot, and the user connected through the debugging graphical user interface.

3.1 TeamNUSTSPL Class

As our software is based on the naoqi local/remote modules, our software starts by making a connection with the naoqi broker manager and adds our software as a naoqi module. The **TeamNUSTSPL** folder consists of the files that are used as the starting point for our software architecture. The module added to naoqi is defined by the class **TeamNUSTSPL** which basically starts up the whole software.

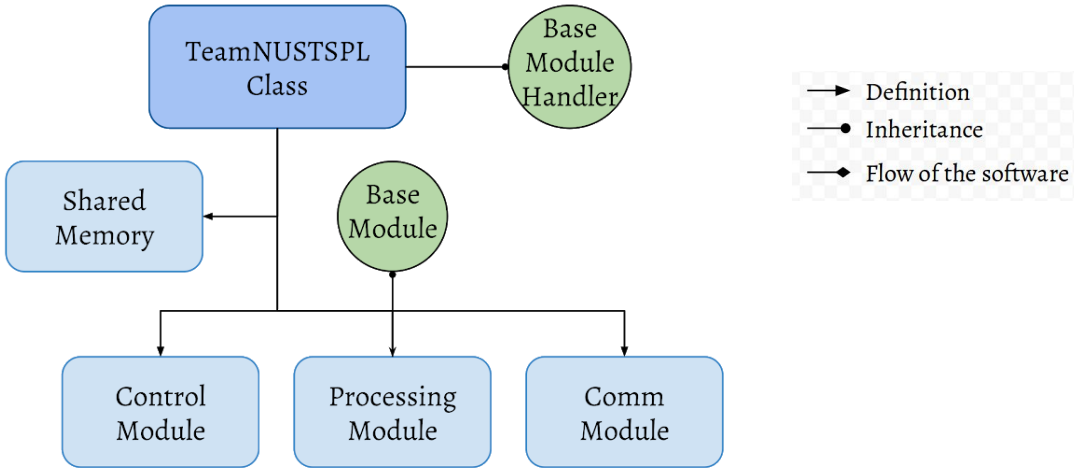

Fig 3.2. An overview of the startup of different modules. The three main classes inherit from the **BaseModule** class.

3.2 Memory Module

The **MemoryModule** folder consists of several classes for the definition of shared memory, shared variables and macro definitions for the access and definition of these variables.

1. **SharedMemory** definition takes place as soon as the **TeamNUSTSPL** class is loaded.
2. **SharedMemory** class as the name suggests defines the memory blackboard and either defines or declares all the shared variables to be shared between different threads. The enumerations for these variables are defined in **MemoryDefinition.h** . The macros used by this class to define and declare the variables are as follows:

```
DECLARE_VARIABLE(type, name)
DEFINE_VARIABLE(type, name, value)
```

3. The classes **Variable** and **ThreadSafeVariable** (both defined in Utils) provide the definitions of templated type variables to be created by the **SharedMemory**. The local variable copies that are made by the individual threads are also defined in the similar fashion. These variables are constructed in **SharedMemory** through the macros defined as above. For further details of the usage of macros for memory definitions, see **ConnectorMacro.h** and **InoutConnectors.h .**

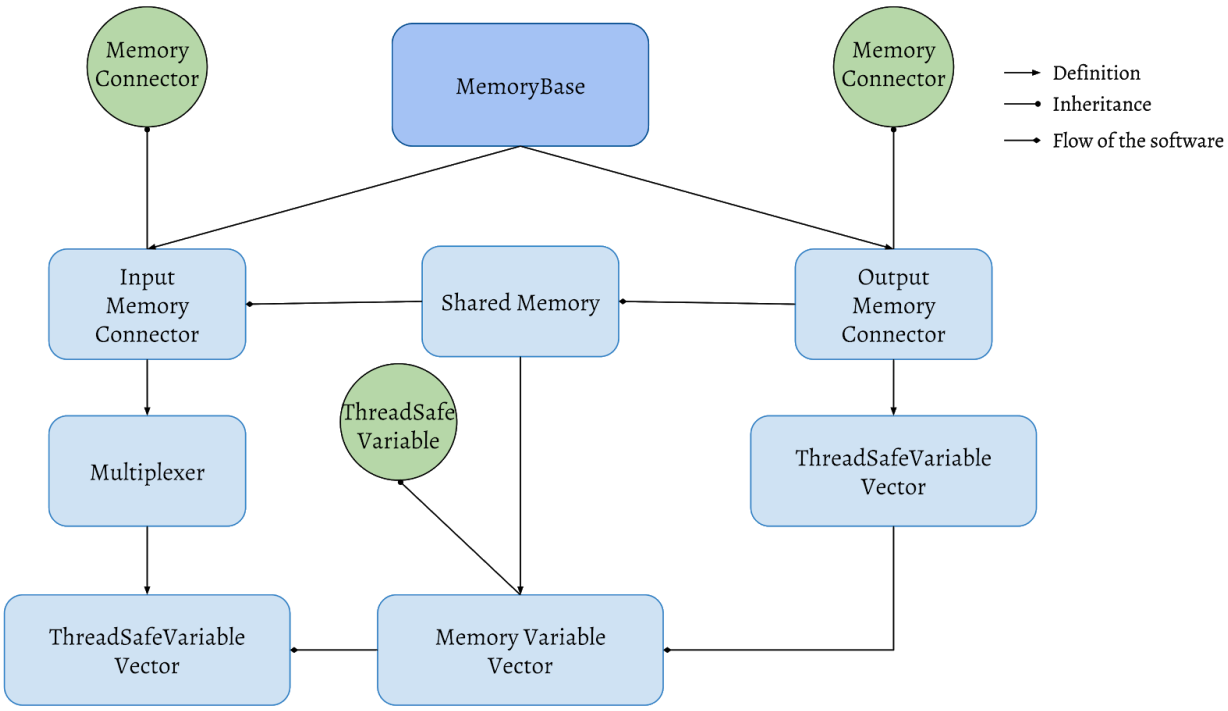

Fig 3.3. An overview of the usage of different MemoryModule classes.

4. The class **MemoryConnector** defines input and output connections to access the memory variables and any class is able to access the memory with its unique input and output connector as long as it inherits the class **MemoryBase (**The class defines a member variable pointer of the Input and Output **MemoryConnectors**).

5. The class **BaseModule** is the base class for all the core threaded modules in our software. A class that inherits from **BaseModule** gains its own thread with its period defined in **SharedMemory** and also directly gains access to the **SharedMemory** on definition. Furthermore, only the classes inheriting from **BaseModule** have the ability to define which of the **SharedMemory** variables will be their input and which variables will be taken as an output from the given class. These input and output variables must be defined for every class inheriting from **BaseModule**, by using the macros defined in **InoutConnectors.h**.

6. The given macros must follow the syntax:

```
CREATE_INPUT_CONNECTOR((type, name1), (type, name2), (type, name3),)
CREATE_OUTPUT_CONNECTOR((type, name1), (type, name2), (type, name3),)
```

   It must be noted that the variables (name1, name2) being connected to from the thread must be declared/defined in **SharedMemory** with the same names (name1, name2, etc).

7. In this manner, every **BaseModule** (from now on, we will call every **BaseModule** child class a **BaseModule**) takes required variables as an input from the **SharedMemory** and updates some variables of the **SharedMemory** as an output. Furthermore, every **BaseModule** can be

suspended and the period of the thread can be changed on runtime given the need. Once a **BaseModule** completes its loop, it synchronizes the local input and output variables copies with the same variables defined **SharedMemory**. For further functional details see **BaseModule.h** .

8. To give access of **SharedMemory** to any non-threaded class used for processing, it must inherit from the **MemoryBase** class and use the **MemoryConnector** of one of the **BaseModule**s already defined (For example, **MotionBehavior** class is dependant upon **MotionModule** which is a **BaseModule**, therefore it must provide the **MotionModule** class pointer to its parent **MemoryBase**. Thus every dependant class must receive a pointer to the independent **BaseModule** class, if it needs to gain access to memory.

3.3 Control Module

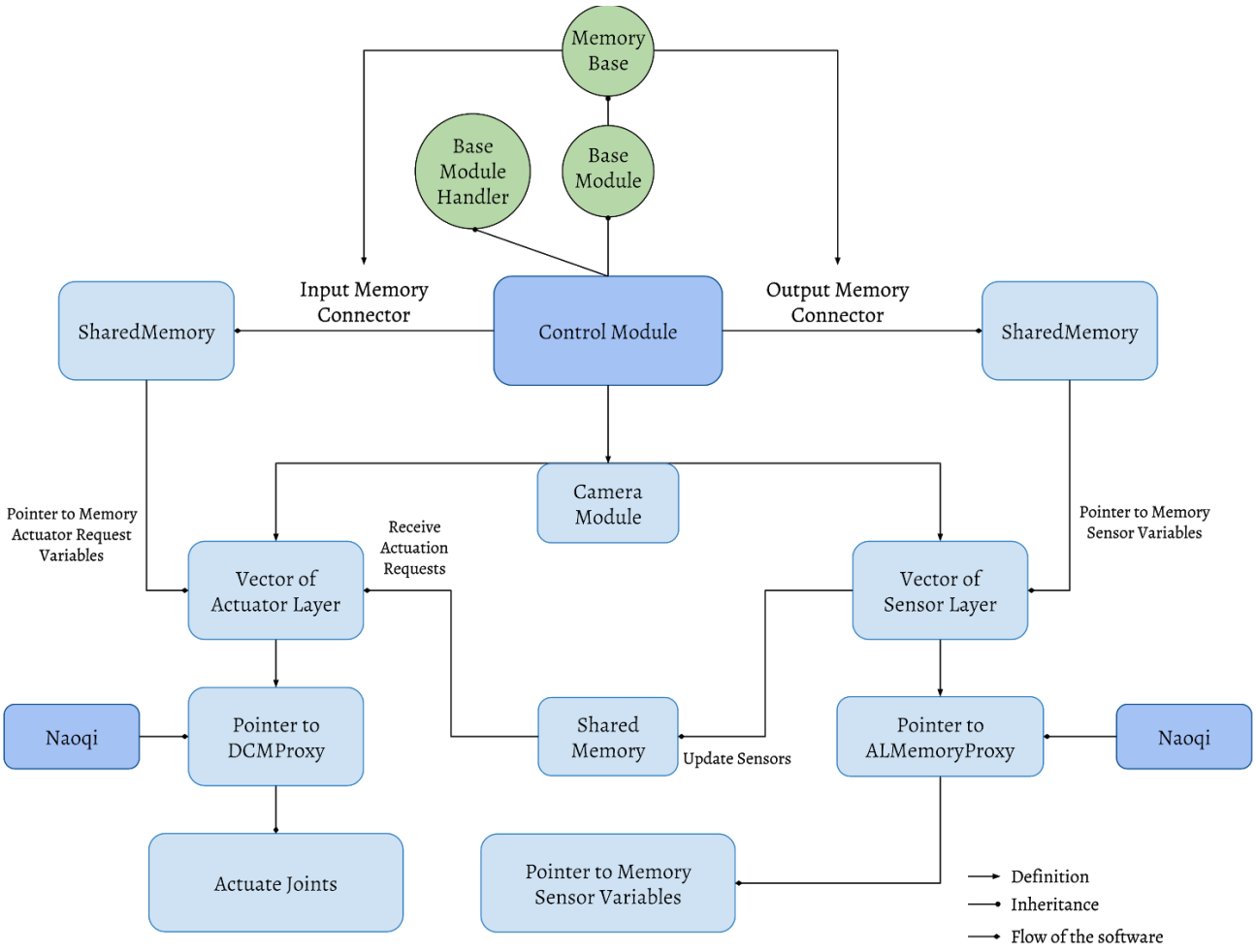

Fig 3.4. An overview of the usage of different Control Module classes.

The **TeamNUSTSPL** class on initiation starts three **BaseModule**s, namely **ControlModule**, **ProcessingModule,** and **CommModule**. The **ControlModule** performs the following operations:

1. It creates an interface with naoqi using two of the naoqi proxies; **ALMemoryProxy** and **DCMProxy** (A proxy to the **DCM, Device Communication Manager**).
2. As **ALMemoryProxy** already provides us the functions to gain pointer access to sensor values with the given keys, they can be easily employed to gather the sensor data.
3. The data gathered from all the sensors is passed on to the **SharedMemory** thus giving sensor data access to other modules.
4. This module takes three variables as inputs from the **SharedMemory**; the JointRequest, StiffnessRequest, and LedRequest. These variables are updated as required from other **BaseModule**s to perform the required actuation (motion execution still has delay issues therefore its use in realtime is not recommended for the time being). Each of the three Request variables consist of the actuator values to be reached in the next iteration of the **ControlModule** cycle. The requested actuator commands are then sent to the **DCMProxy** to carry them out. For now the **ControlModule** is only able to send **time-separate** commands to the **DCMProxy**, and thus to perform actuation we must send actuation commands in real-time with a time difference of 10ms and therefore JointRequests must not be sent to the real robot with a remote connection. Although these requests are implemented and work alright, for safety, we are still using naoqi's motion interpolation function for motion execution until we can completely remove the usage of naoqi's ALMotionProxy (Currently ALMotionProxy is necessary for steps generation, and as we cannot remove it, why not use its helpful functions (:).
5. With the issues given above, we also need to expand the **ControlModule** to send **time-mixed** commands to the **DCMProxy**, meaning that a set of requests with a given time vector for each actuation command be sent at once to the **DCM**. For a detailed description of DCM and its commands, visit http://doc.aldebaran.com/2-1/naoqi/sensors/dcm.html.

3.4 Camera Module

The **CameraModule** is another **BaseModule** defined under **ControlModule** that uses naoqi's **ALVideoDeviceProxy** to gain access to the camera input. To get a detailed explanation of how the proxy is used to get the camera input, visit http://doc.aldebaran.com/2-1/naoqi/vision/alvideodevice-tuto.html?highlight=camera.

It performs the following operations:

1. The **CameraModule** also consists of the **Camera** class with several options for configuring the input image properties such as type, resolution, etc. These properties are defined in **Config/UpperCamera.cfg** and **Config/LowerCamera.cfg** files based on the required camera calibration. For more information on Camera class, see **CameraModule/Camera.h** and **CameraModule/alvisiondefinitions.h** .
2. The resulting images extracted from both the upper and lower cameras are updated to **SharedMemory** as opencv matrices for further processing.

3. It must be noted that in case the code is built with **-DMODULE_IS_REMOTE=ON**, the **CameraModule** will send BGR images (As the simulator V-REP is only able to generate BGR images) to the **SharedMemory**, thus the colorspace of the image must be converted to the required one, later on, while processing the image. However, in case we extract the image directly from the robot, naoqi's **ALVideoDeviceProxy** provides the option to directly extract the image in any required color space, thus in that case, we will not need to change the colorspace of the image during its processing. To get a detail of these options, visit http://doc.aldebaran.com/2-1/family/robots/video_robot.html#robot-video.
4. The color space currently being used for image processing is HSV and so that color space must be set as default configuration for the cameras in real time operation as opposed to the simulation where the default configuration must use BGR color space. Meaning we need to update the mentioned **".cfg"** files for each case.

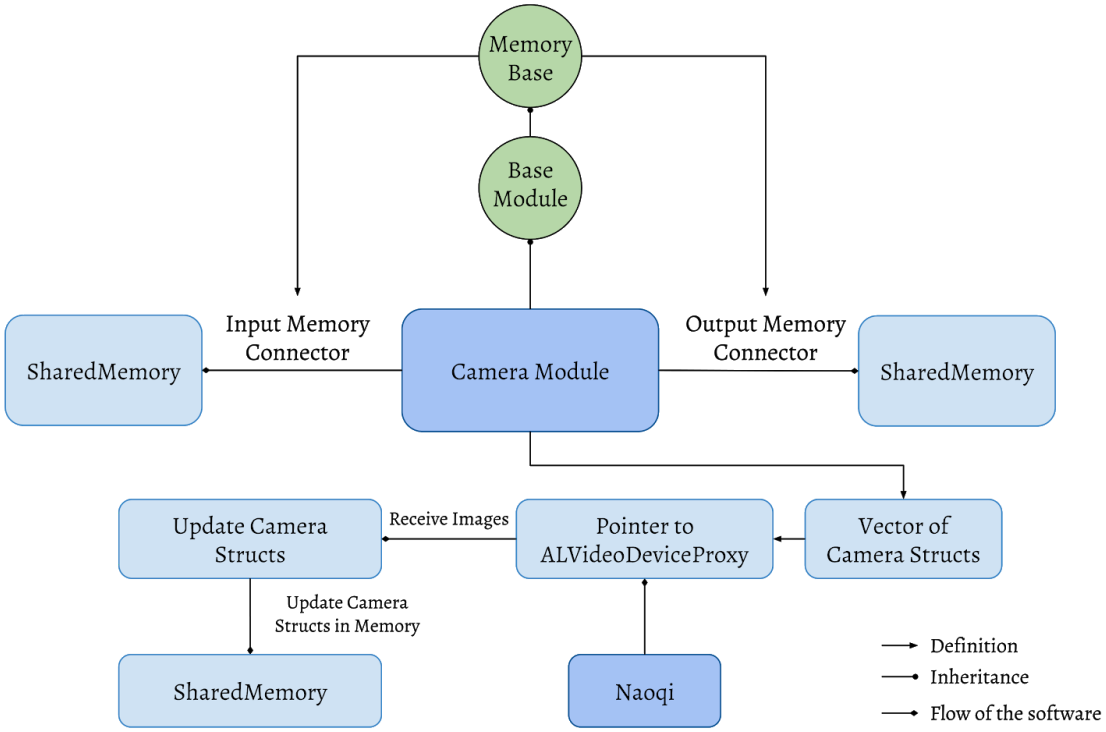

Fig 3.5. An overview of the Camera Module class.

3.5 Comm Module

The **CommModule** is also a **BaseModule** defined by the base class **TeamNUSTSPL** that performs all the operations pertaining to the communication of the robot with the user, other robots, and with **GameController**. It performs the following operations:

1. For debugging purposes, we have our own debugging graphical user interface which can be used to connect to the robot via a TCP connection. The robot acts as a server in this communication whereas the gui is connected as a client.

2. The **CommModule** uses two separate ports for images and data send/receival, namely, **dataPort** and **imagePort**.
3. It provides several debugging resources such as;
    a. The ability to send log messages from anywhere in the code using its static function **CommModule::addToLogMsgQueue** which provides threadsafe access to the log messages queue.
    b. Special messages such as the path planned, particle filter states, planned footsteps, etc can also be sent in the form of bytes array.
    c. Each message type is assigned an identity, details of which can be seen it **CommMsgTypes.h** .
4. Furthermore, on each update the module sends all the data in **SharedMemory** to the connected clients on the **dataPort**.
5. Similarly, the **SharedMemory** variable **imageToSend** updated by the **VisionModule** is received and sent to the client via **imagePort**.
6. It also deals with the commands received from the user, sends it to the **DebugModule** for processing. For details on syntax of a debugging command, see section 3.16.

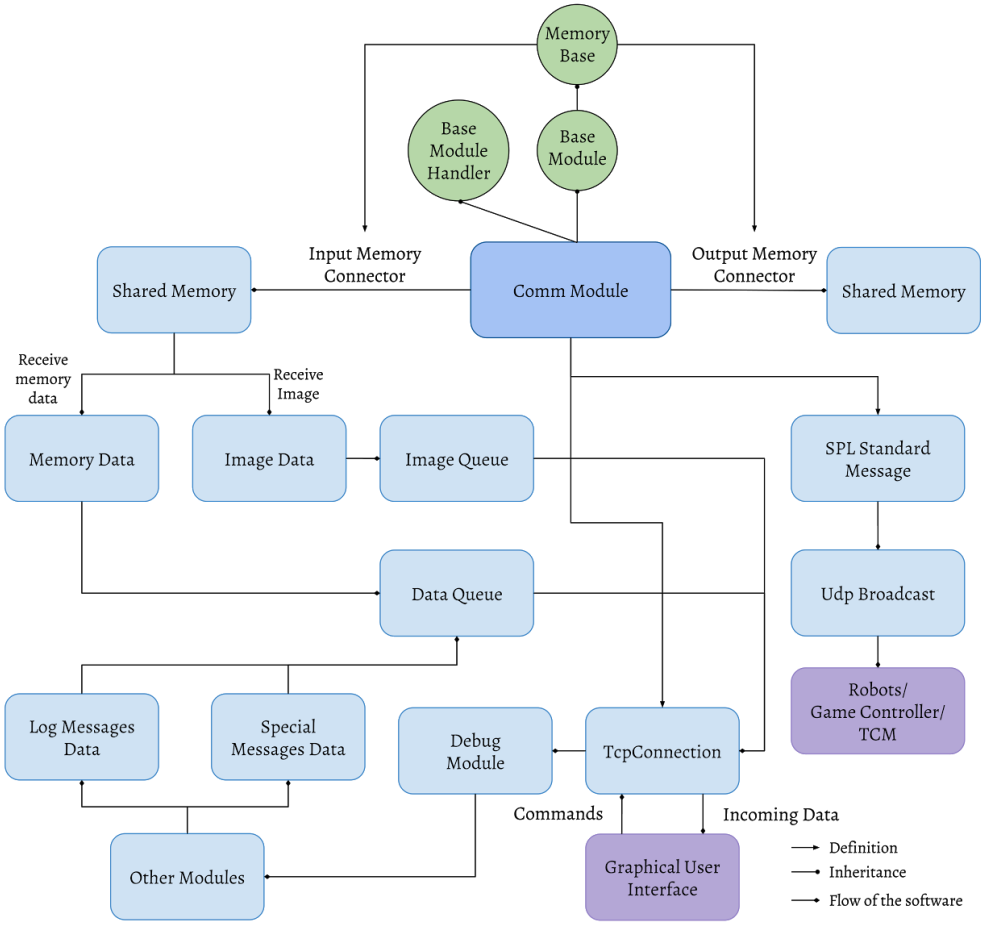

Fig 3.5. An overview of the Comm Module class.

7. For inter-robot communication, **CommModule** also broadcasts the **SPLStandardMessage** packet on the team port (which is basically 10000 + teamNumber) using the UDP protocol (as required by the SPL rules 2017). For details on the **SPLStandardMessage**, see **GameController/include/SPLStandardMessage.h** .

3.6 Debug Module

The **DebugModule** is a simple interface class which can be inherited by any class to gain access to some debugging variables, which can be updated via gui interface. It works in the following manner:

1. Any child of the class **DebugBase** must pass its name (or any name with which to point that class out), and its pointer to the base class **DebugBase**.
2. It must also use the Macro **INIT_DEBUG_BASE(...)**, to define the debugging variables required by the said class. For further information on how to define a debug variable, see **DebugModule/DebugBase.h** .
3. The static function **DebugBase::processDebugMsg** takes an input string message and checks whether it is a valid debug message. A valid debug message can be defined in one of the following ways:
   ClassName:VariableName:Value
   ClassName:{Value1, Value2, … ,ValueN}
   (N must be equal to the number of variables)
4. If a valid message is received, the specified debug variables of the given class are updated during runtime.
5. Although a debug variable can be any variable whose string to variable conversion is defined in **Utils/DataUtils.h**, for safety purposes, it is recommended to keep debug variables as simple switches to either visualize some output or turn it off.

3.7 Processing Module

The **ProcessingModule** performs all the processing operations required to perform any task. Therefore, it acts as the starting point for other several **BaseModule**s, each of which performs its own processing in its own thread to give some output to the **SharedMemory**.

The **BaseModule**s controlled by the **ProcessingModule** include:

1. PlanningModule
2. MotionModule
3. SBModule
4. VisionModule
5. LocalizationModule
6. DAQModule

3.7.1 Planning Module

PlaningModule is the most important module in our software which not only controls the overall robot behavior but also controls the working of other modules. This module has following properties:

1. The most important class in this section is the **BehaviorManager** class. This class provides the interface for initiating, updating and cleanly finishing any behavior running on the class it gets inherited from.
2. **PlanningModule** class is one of the **BaseModule**s that inherits from **BehaviorManager** and for this case the **BehaviorManager**, controls the **PlanningBehaviors**. The behaviors to run must be defined separately for each class inheriting from the **BehaviorManager**. In **PlanningModule**, every behavior inherits from the class **PlanningBehavior**, which in turns inherits from the base class **BehaviorRequest.**
3. Every behavior must also be provided with a separate config struct inherit from the class **BehaviorConfig**, which basically defines the initializing parameters of each behavior. For details on how to define a **BehaviorConfig** for the given class, see **BehaviorConfig.h** . The id and the types of each **PlanningBehavior** are defined in **PlanningBehaviorIds.h** . These ids and types are used to easily distinguish between separate configurations.
4. Currently two **PlanningBehavior** are defined under this module, namely, **RobotStartup** and **RobocupGameplay**.

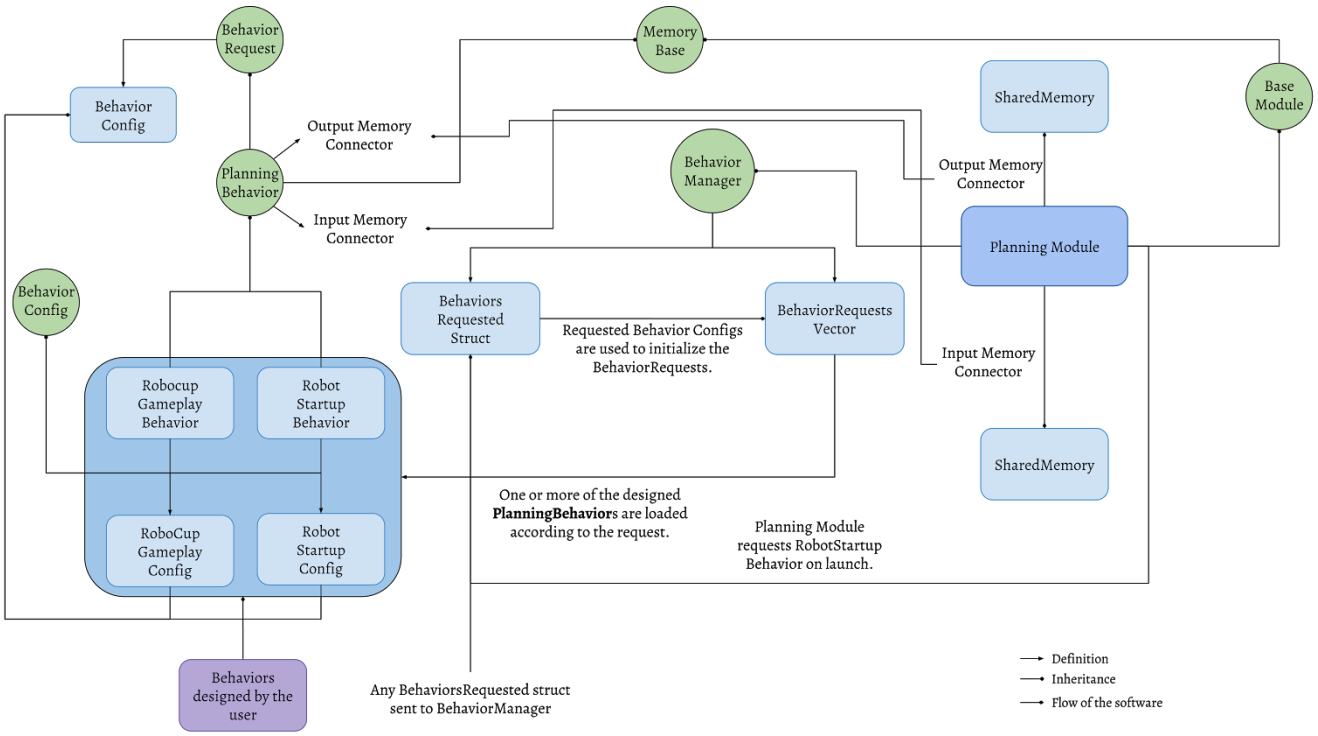

Fig 3.6. An overview of the Planning Module class.

5. **RobotStartup** is started on the initiation of the class **PlanningModule**. It sets the robot stiffness to max, sends the robot to crouch position and waits for a chest button press. Once the chest button is pressed, the **RobocupGameplay** behavior is started.
6. The **RobocupGameplay** behavior deals starts with setting the stiffnesses required for gameplay and sending the robot to stand position. It further performs the following behaviors:
    a. At the start, the robot's state is set to **STATE_INITIAL** (for details on state, see **GameController/include/RoboCupGameControlData.h**), as defined in **GameController**. In this state, the changes in secondary states are accounted for accordingly, to send the robot to main gameplay behavior, or penalty behavior.
    b. The robot's states are given in the **RoboCupGameControlData** packet which is updated by the libgamectrl.so (for further details see https://github.com/bhuman/GameController) local module, running in parallel to our code on the robot. The updated **RoboCupGameControlData** packets are added to **SharedMemory** in **ControlModule** and therefore can be accessed.
    c. In **STATE_READY**, we start up the **VisionModule** and **LocalizationModule**, and the robot starts to interact with the environment. The **STATE_READY** behavior requires the robot to reach its initial position in the field. (In actual gameplay, the robots are placed on the sidelines in our own half, and so we already know a guess of our own position which is used to initialize the localization module. For more details, see http://spl.robocup.org/wp-content/uploads/downloads/Rules2017.pdf). Once the localization is achieved, the robot is required to reach its desired position. The role is assigned to the robot based on its initial position estimate. Based on the given role, the robot is required to reach its initial position (See SPLRuleBook and **PlanningBehaviors/TeamPositions.h**, for more details on starting positions). The PlanningModule, initiates the movement behavior to reach the desired position and once the robot accomplishes that, it is sent to **STATE_SET**,
    d. In **STATE_SET**, the robot waits for the whistle or for the **GameController** command, to go into the **STATE_PLAYING**.
    e. In **STATE_PLAYING**, each robot performs its own behavior based on the assigned role.

3.7.2 Motion Module

MotionModule class deals with all the processing related to the robot's motion; robot kinematics, robot dynamics and control, stability, and motion planning. This module has following properties:

1. The module takes as an input the required sensor values from the **SharedMemory** and updates the **JointRequest** objects in the **SharedMemory**.
2. To control the hierarchy of executed motions, the **MotionModule** also inherits from the **BehaviorManager** class is used. Every defined motion such as kick, or reaching a desired posture, etc is defined by a separate class and each of these classes inherit from the **MotionBehavior** class (inherits from **BehaviorRequest** class similar to **PlanningBehavior** class**)**.
3. Each **MotionBehavior** must also provided with an id, type and a config (See

MotionBehaviorIds.h and **PlanningModule/BehaviorConfig.h** for details**).**

4. The hierarchy of motion execution is dealt with by the **BehaviorManager**, which basically allows two behaviors to be running together only if they are initialized together and then waits for both of them to finish. Currently, running multiple behaviors are not recommended as it can lead to dangerous movements and the safety module for whether required behaviors can run together is still needed to be implemented.
5. It must be noted that the joint request of every **MotionRequest**, starts with assigning **'NAN'** values to the vector. Sequentially, the required joint values are updated throughout the given **MotionRequest**, and the unmoved joints are left with **'NAN'** values. This is done because the naoqi's **DCM** module ignores all the **'NAN'** values passed to it for a given joint actuation and so only the joints with values assigned to them are actuated.

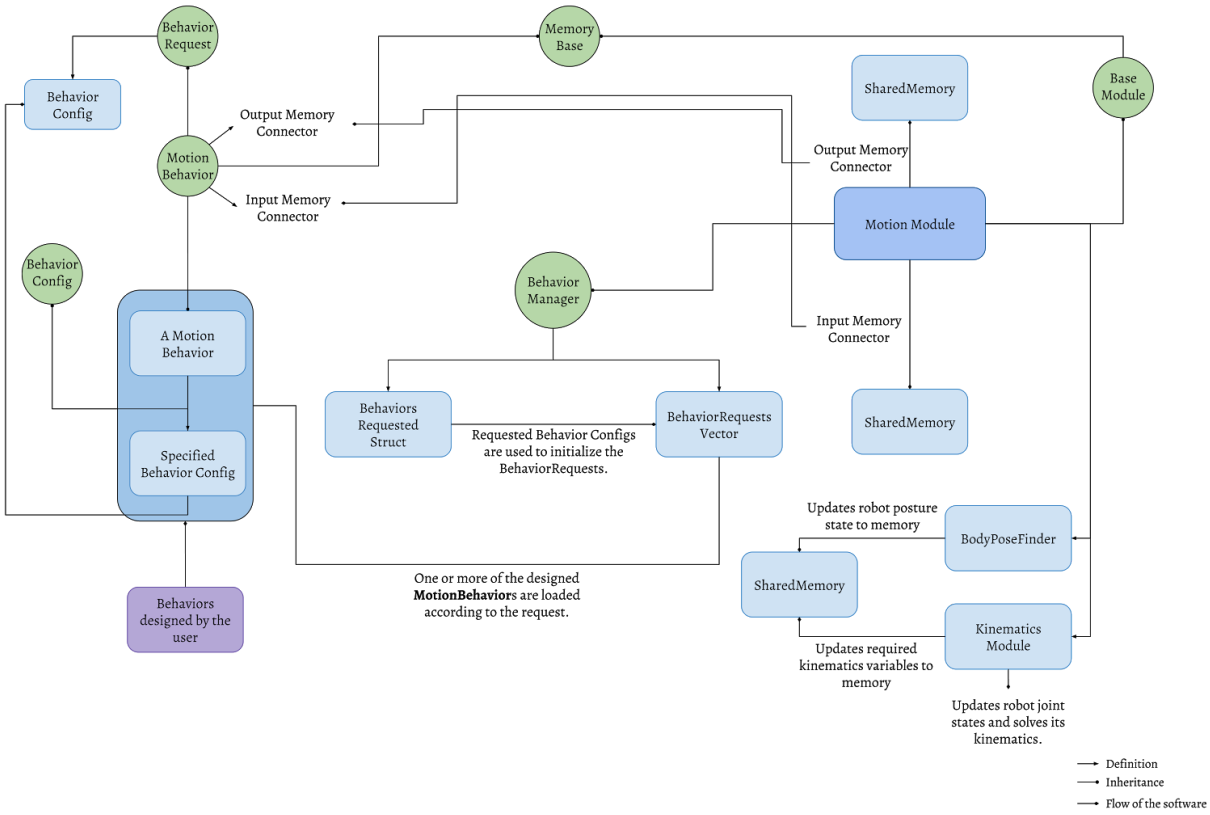

Fig 3.7. An overview of the Motion Module class.

6. For preplanned motions, a function called **naoqiJointInterpolation** is defined in **MotionBehavior** class for easier motion execution instead of updating joints in real-time using DCM. Currently, real-time updates faces issues due to time lag between sent commands and their execution causing severe jerks. This issue is probably a result of many naoqi modules running in parallel including ALMotionProxy and therefore, the commands cannot be processed as fast as needed.
7. One of the most important part of the **MotionModule** is the **KinematicsModule**. Although

named **KinematicsModule**, currently it contains the resources and algorithms required for solving both the robot kinematics and dynamics. The input joints, fsr, and inertial sensor data is currently being passed to the **KinematicsModule**, which it uses to update the robot's kinematic and dynamic model in each iteration. Currently this module provides the algorithms for solving the following:

   a. Forward kinematics.
   b. Analytical/numerical inverse kinematics.
   c. Center of mass kinematics.
   d. Center of mass, link and end-effector jacobians.
   e. Chain inertia matrix.
   f. Generalized center of mass inverse kinematics with a given base limb.
   g. Recursive newton-euler algorithm (RNEA) for solving the joint torques for the given joint states (position, velocity and acceleration of all the joints of the chain).
   h. Zmp computation with respect to the left or right feet using RNEA on all the chains of the robot as defined above.
   i. It also provides the option to perform separate computations on actual or commanded joint states, or any other self defined states for validation purposes.

7. **TrajectoryPlanner** is another module of the **MotionModule** which provides us a few different types of trajectory generation algorithms based on cubic splines, b-splines, or quintic splines. The b-splines are generally used for cartesian space planning whereas the cubic splines are used for joint space planning. This module also provides time-optimized joint space trajectories given the velocity, and torque constraints based on cubic splines.
8. **BodyPoseFinder** class updates the body state of the robot based on its perceived orientation from the inertial measurement unit (IMU) and ground feet contact using the force sensors.
9. **PostureModule** is one of the modules that inherits **MotionBehavior**. This module is used to send the robot to a desired posture smoothly. Currently only two postures are defined; crouch and stand. Although safe with current postures, this module needs further improvements as it does not, as of yet, checks whether the desired posture is reachable, stabile or self-colliding.
10. **BalanceModule** also inherits from **MotionBehavior**. This module uses either of the two criterias to balance the robot:
    a. Center of mass based balancing which uses **PID control** on the center of mass error from the desired position.
    b. Zero Moment Point (ZMP) based balancing techniques to find a stable center of mass trajectory using a inverted cart-table model based ZMP planner.
    c. The resulting center of mass trajectory is followed by finding required joint trajectories based on generalized center of mass inverse kinematics.
11. **KickModule** which is another **MotionBehavior** child class defines the kicking motions of the robot and is specifically designed for Robocup.
12. **HeadTracking** provides the implementation of smooth tracking of any given point of interest in the camera image.
13. **BallGrasping** defines the ball grabbing and throwing behavior.
14. **MovementModule** is the **MotionBehavior** that once started initially finds the required footsteps to reach a desired end pose footstep on the map and then executes the resulting steps. The steps

execution is carried out using naoqi's **setFootSteps** function (for details, see http://doc.aldebaran.com/2-1/naoqi/motion/control-walk.html#control-walk). The footsteps are generated using the PathPlanner module (See **MotionModule/PathPlanner**) based on the open-source ROS package (ported to our code) given at http://wiki.ros.org/footstep_planner. For further details on its working, see ADD REFERENCE, and http://hrl.informatik.uni-freiburg.de/research/animations/footsteps/. The **MovementModule** also calculates the commanded position of the robot to the **LocalizationModule** however it does not currently use joint sensor data or IMU data to solve odometry (Needs to be implemented).

15. On details of the possible configurations of each of the mentioned **MotionBehavior**s, see **BehaviorConfig.h** .

3.7.3 Static Behaviors Module

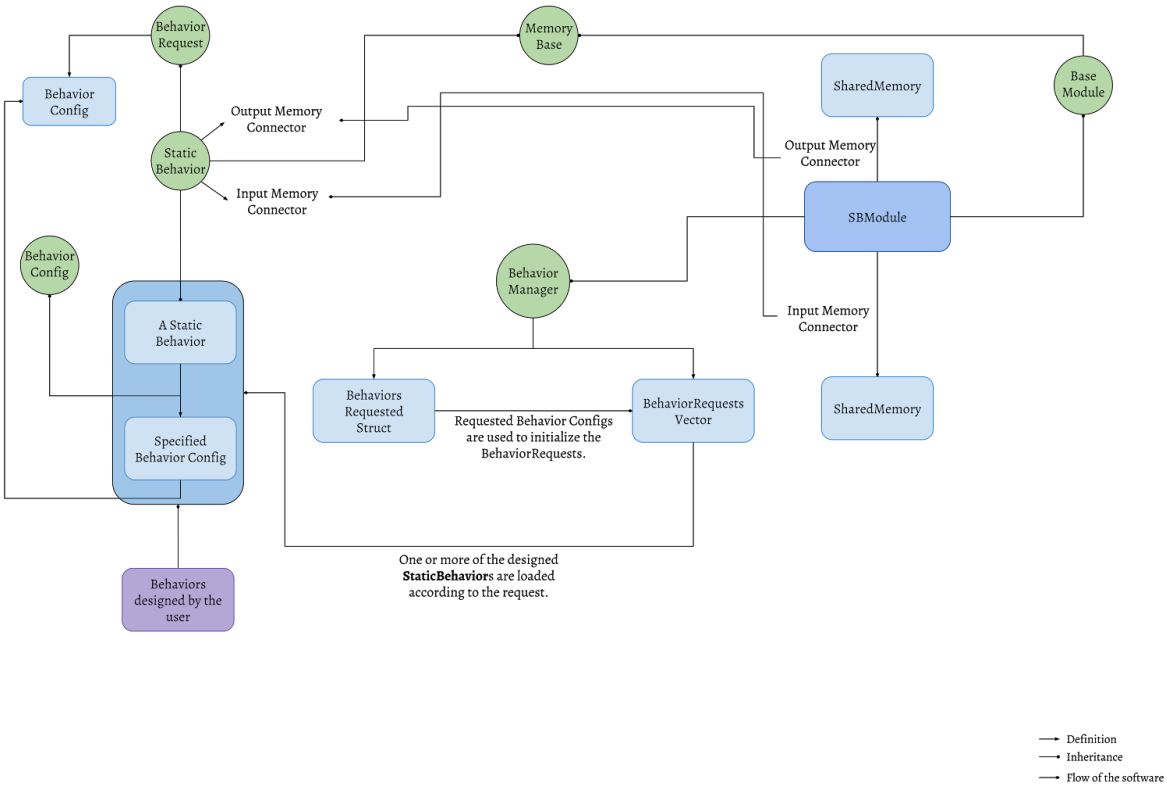

Fig 3.8. An overview of the Static Behaviors Module class.

SBModule class is very similar in working to the **MotionModule** class with the only difference being that it deals with stiffness and led actuators instead of joint actuators. This module has following properties:

1. The module takes as an input the required sensor values from the **SharedMemory** and updates the **StiffnessRequest** and the **LedRequest** objects in the **SharedMemory**.

2. Similar to **MotionModule**, it also also inherits from the **BehaviorManager** class. Every defined static behaviors such as setting stiffnesses or leds is defined by a separate class and each of these classes inherit from the **StaticBehavior** class (inherits from **BehaviorRequest** class similar to **PlanningBehavior** and **MotionBehavior** classes**)**.
3. Each **StaticBehavior** must also provided with an id, type and a config similar to Planning and Motion Behaviors.
4. Similar to joint request, every **StiffnessRequest** or **LedRequest**, starts with assigning **'NAN'** values to the vector. Sequentially, the required joint values are updated throughout the given cycle, and the unwanted actuators are left with **'NAN'** values.
5. Similar to **naoqiJointInterpolation**, a **naoqiStiffnessInterpolation** function is defined in static behaviors for updating the stiffnesses through a naoqi's function (**ALMotionProxy** does not work with stiffnesses updated through DCM even if the robot's stiffnesses are indeed set to some value). Unlike joint movements, the stiffnesses or leds can be updated by sending the required actuator requests as needed.
6. Two **StaticBehaviors** are defined here, namely, **StiffnessModule** and **LedsModule** each dealing with their related actuators.

3.7.4 Vision Module

The **VisionModule** class deals with all the processing related to the robot's visual perception; image processing, feature extraction, and the transformation from image to world coordinate system. Robocup SPL challenge has a predefined field so we can use the known information for help in extraction of features and therefore many of our algorithms make use of the known features. Furthermore, all of our extraction algorithms use OpenCV library for basic operations. The **VisionModule** has following properties:

1. The **ImagePreprocessor** class is used to directly preprocess the images being received from the **SharedMemory**. The term preprocessing here means noise reduction, application of filters, or color space conversion if needed be. As filters can be very expensive, we are currently only using **contrast-limited adaptive histogram equalization (CLAHE)** on the **Value** of the **HSV** image to make the image intensity uniform.
2. The camera frame transformations and conversion from image to world and vice versa are provided by the **ImageTransform** class. It works in the following manner.
    a. The camera intrinsic matrix are defined once at the startup for both the cameras.
    b. The camera extrinsic matrix from the camera to ground plane is found by first receiving the camera to feet transformation matrix from the **SharedMemory** updated by **KinematicsModule** and then taking its inverse.
    c. To find the given image coordinate in the world, one must provide the height of the point in the world frame defined at the center of the robot feet on the ground plane. Thus every field landmark height can be set to zero and for ball it can be set equal to its radius.
    d. Similarly, world to image transformation can be done by providing an X-Y-Z

coordinates of the point in world frame.

3. The **ColorHandler** class provides the color definitions and their ranges based on the **HSV** colorspace.
4. For extracting features, we have a **FeatureExtraction** base class which is inherited by several feature extraction modules such as **GoalExtraction**, **BallExtraction**, etc. **FeatureExtraction** class also inherits from the **MemoryBase** class using the **MemoryConnector** of **VisionModule** to give **SharedMemory** access to all the feature extraction child classes. Furthermore, it provides protected pointers to ImageTransform and ColorHandler classes for usage in child classes.
5. The preprocessed images are sent to each of the feature extraction classes to detect features. The detailed explanation of the algorithms used for feature extraction are described in the sections below.

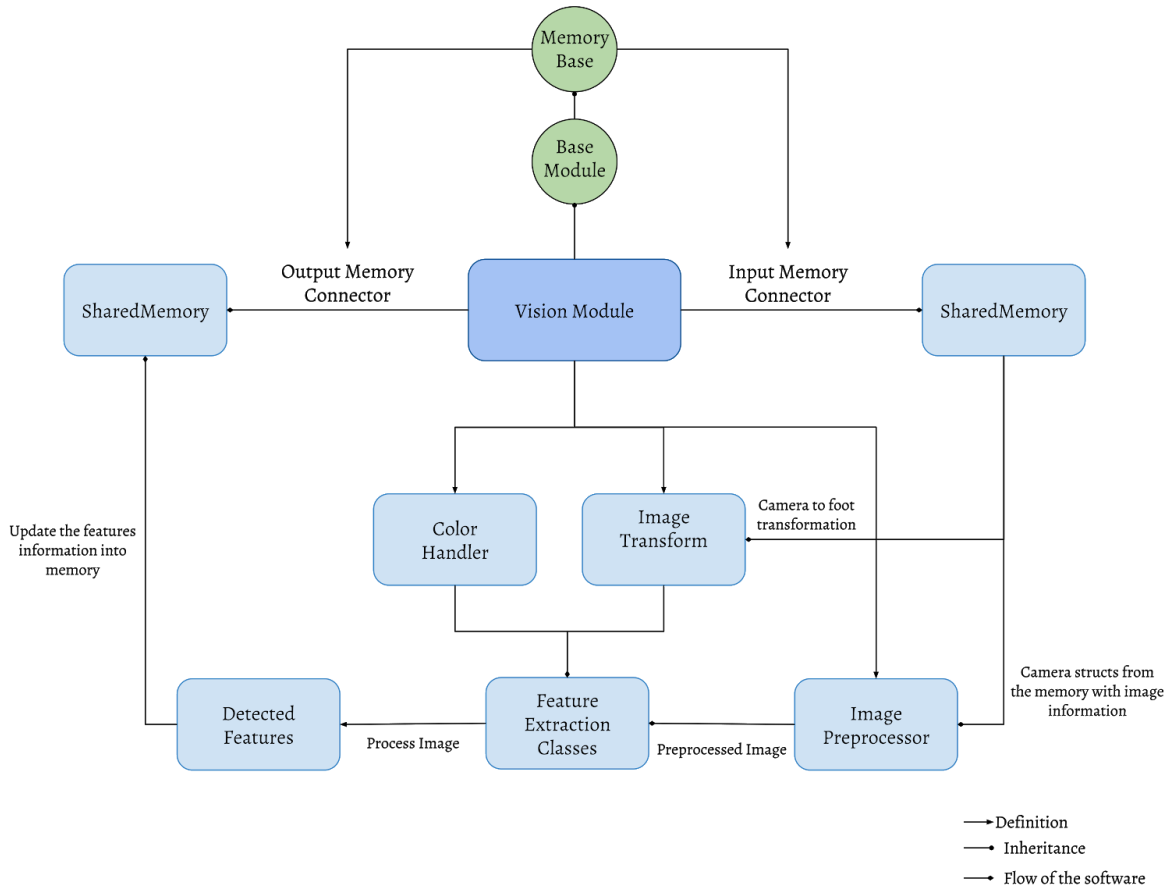

Fig 3.9. An overview of the Vision Module class.

3.7.4.1 Field Extraction

3.7.4.2 Goal Extraction

3.7.4.3 Robot Extraction

3.7.4.4 Lines Extraction