Master in Artificial Intelligence and Robotics (MARR)
- Elective in AI, Robot Programming

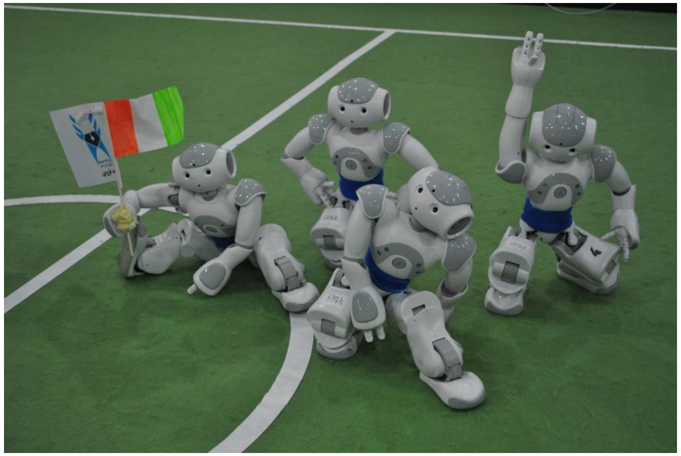
SAPIENZA
UNIVERSITÀ DI ROMA

Programming NAO-Robots

Francesco Riccio

October 7th 2014

SPL – Standard Platform League



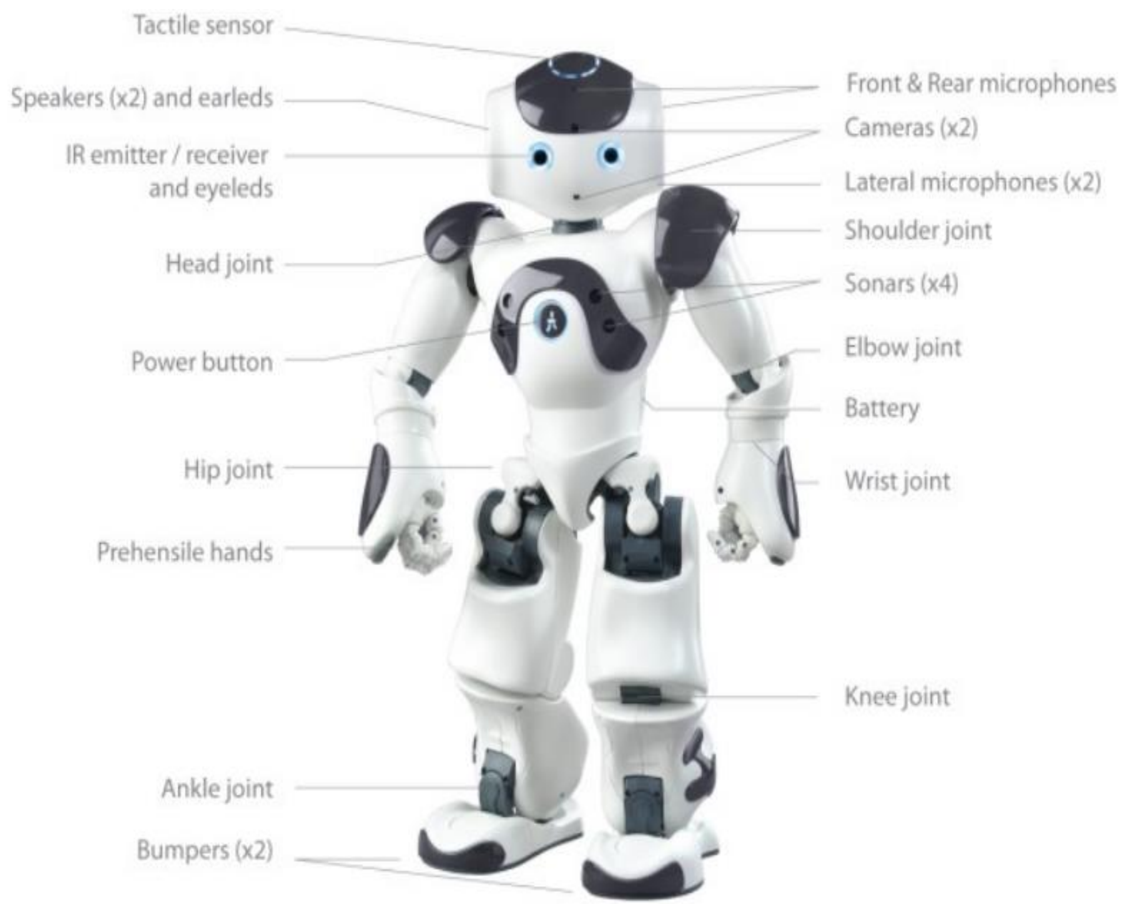
S.P.Q.R. (Soccer Player Quadruped Robots) is the RoboCup team of the **Department of Computer, Control, and Management Engineering "Antonio Ruberti"** at **Sapienza university of Rome**

- Middle-size 1998-2002;
- Four-legged 2000-2007;
- Real-Rescue robots since 2003;
- Virtual-Rescue robots since 2006;
- Standard Platform League since 2008;

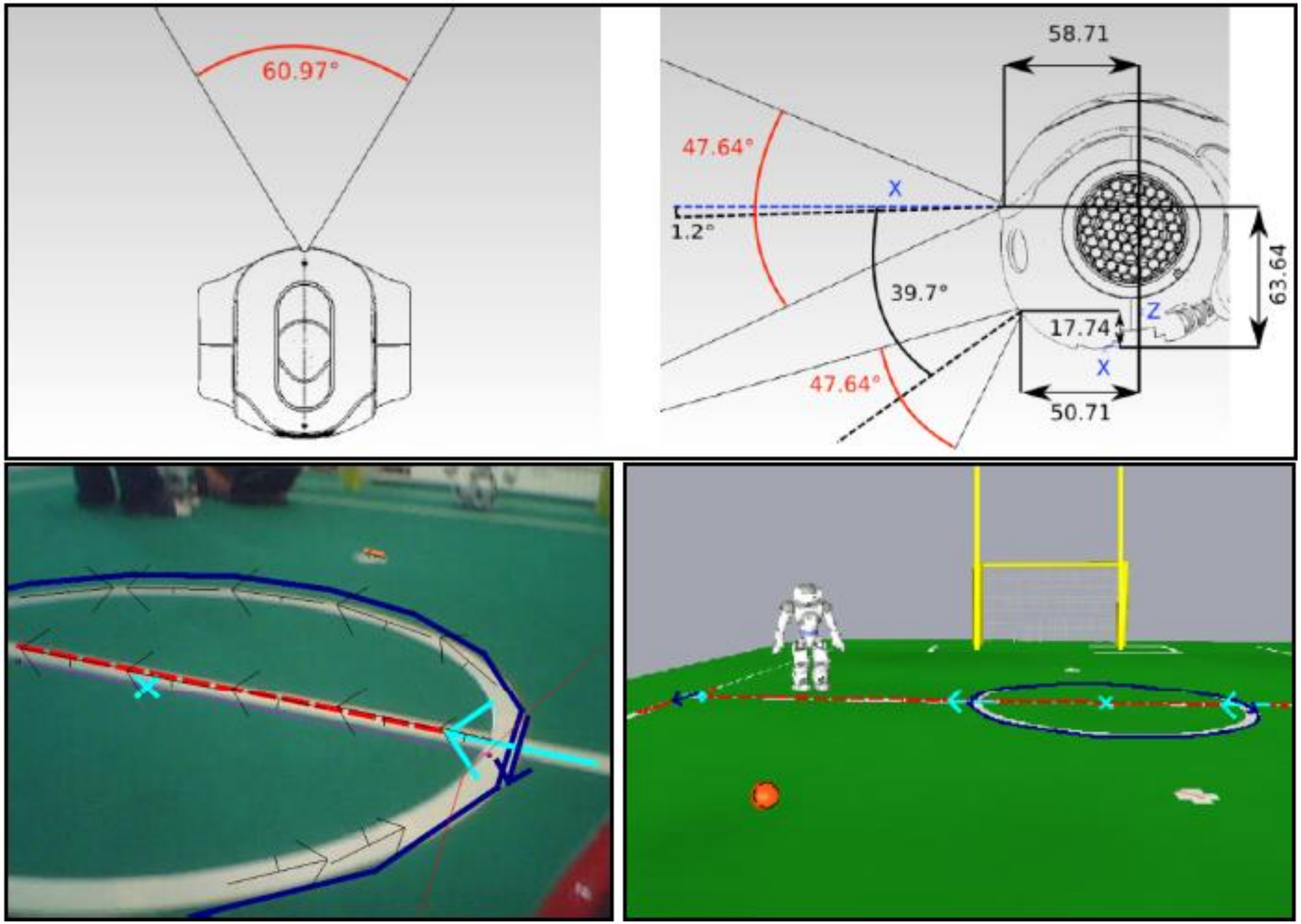
The Aldebaran Nao robot

Nao is an autonomous, programmable, medium-sized humanoid robot.

ATOM Z530 1.6GHz CPU 1 GB RAM / 2 GB flash memory / 4 to 8 GB flash memory dedicated to user purposes



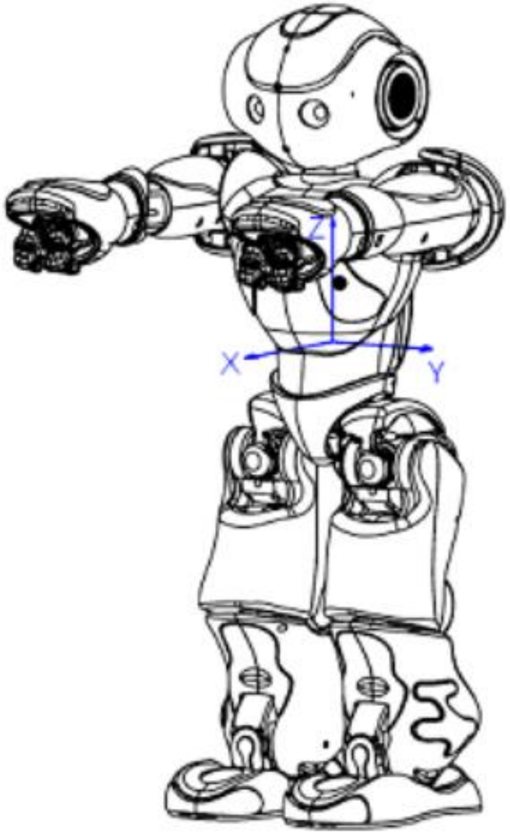
Camera Nao



Inertial unit

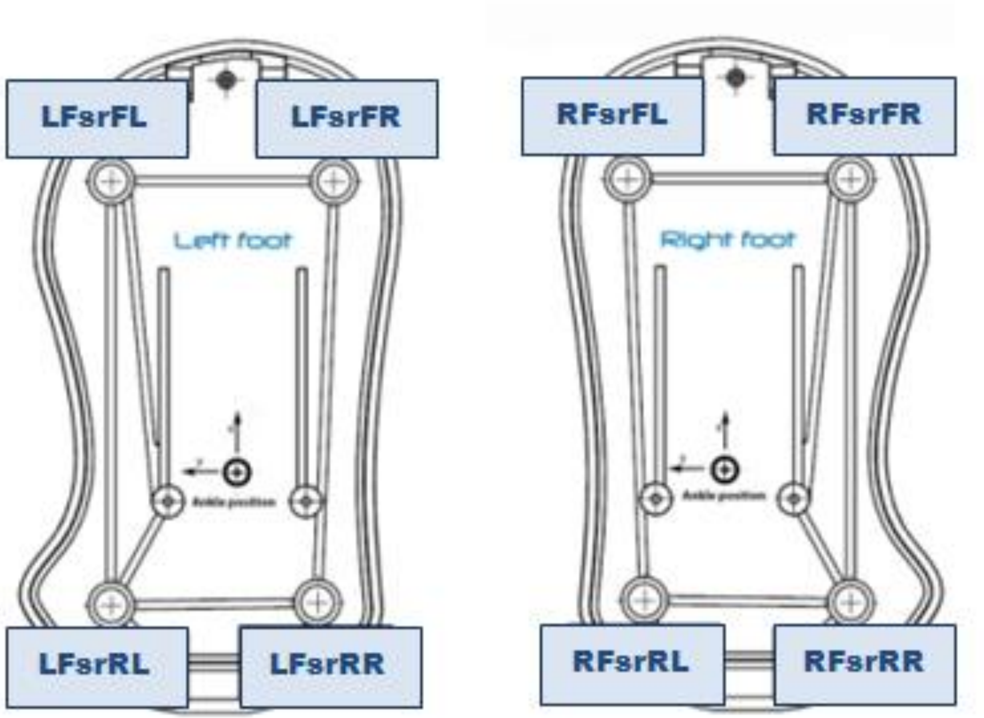
- 2 axes gyrometers
- 1 axis accelerometers

The **Inertial unit** is located in the torso



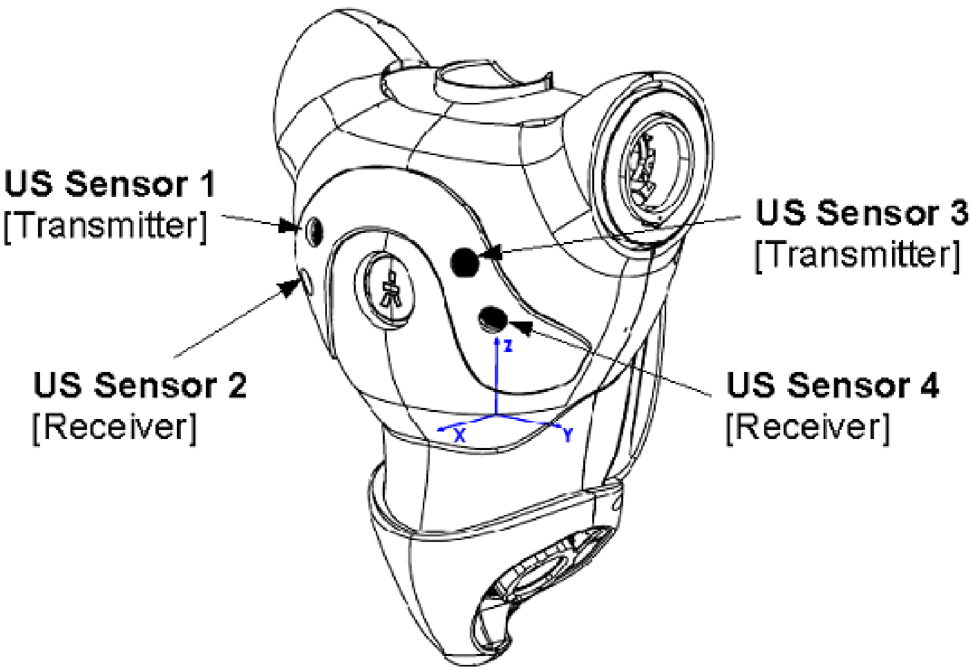
FSR – Force Sensitive Resistor

These sensors measure a resistance change according to the pressure applied.

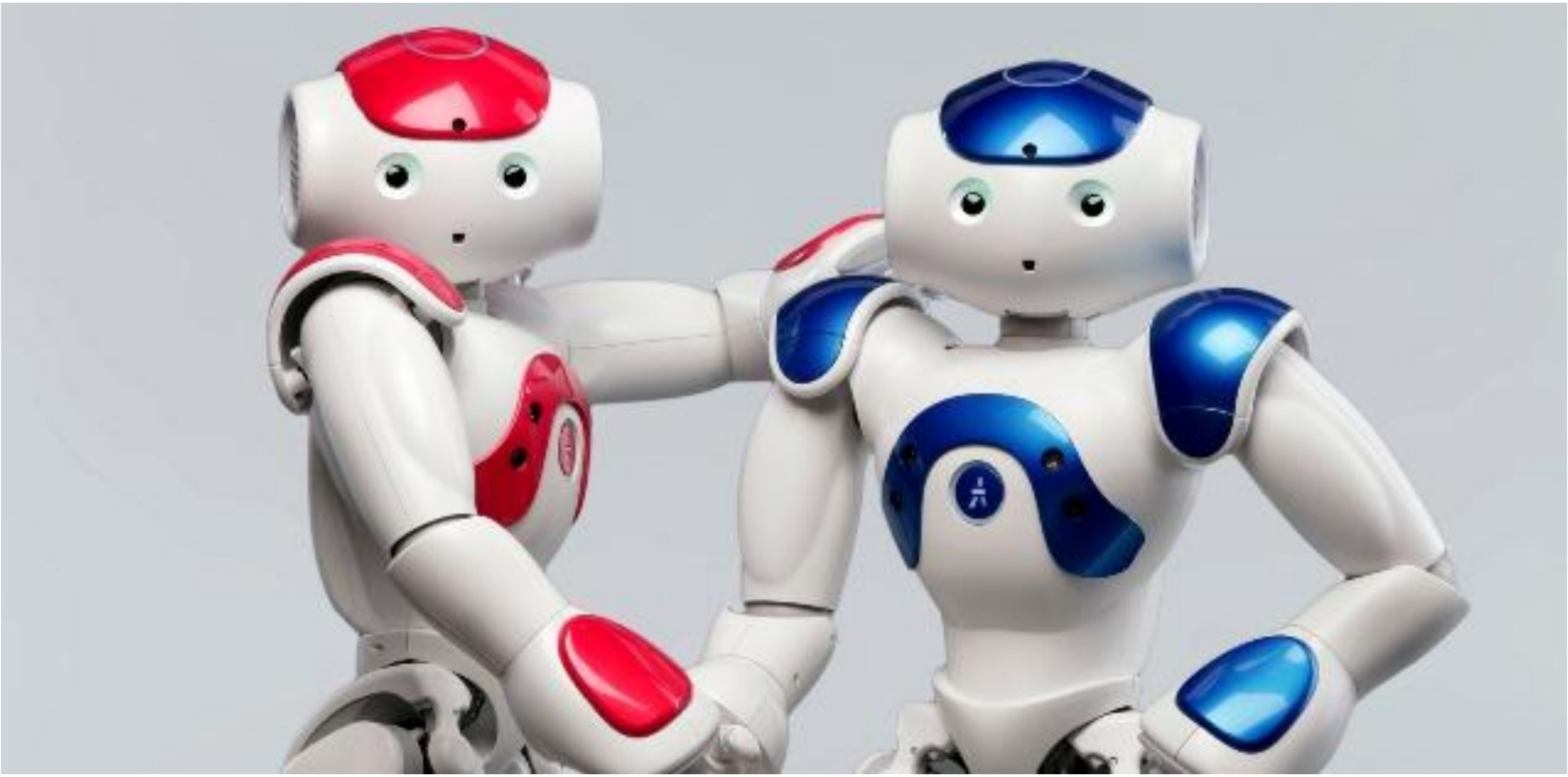


Sonars

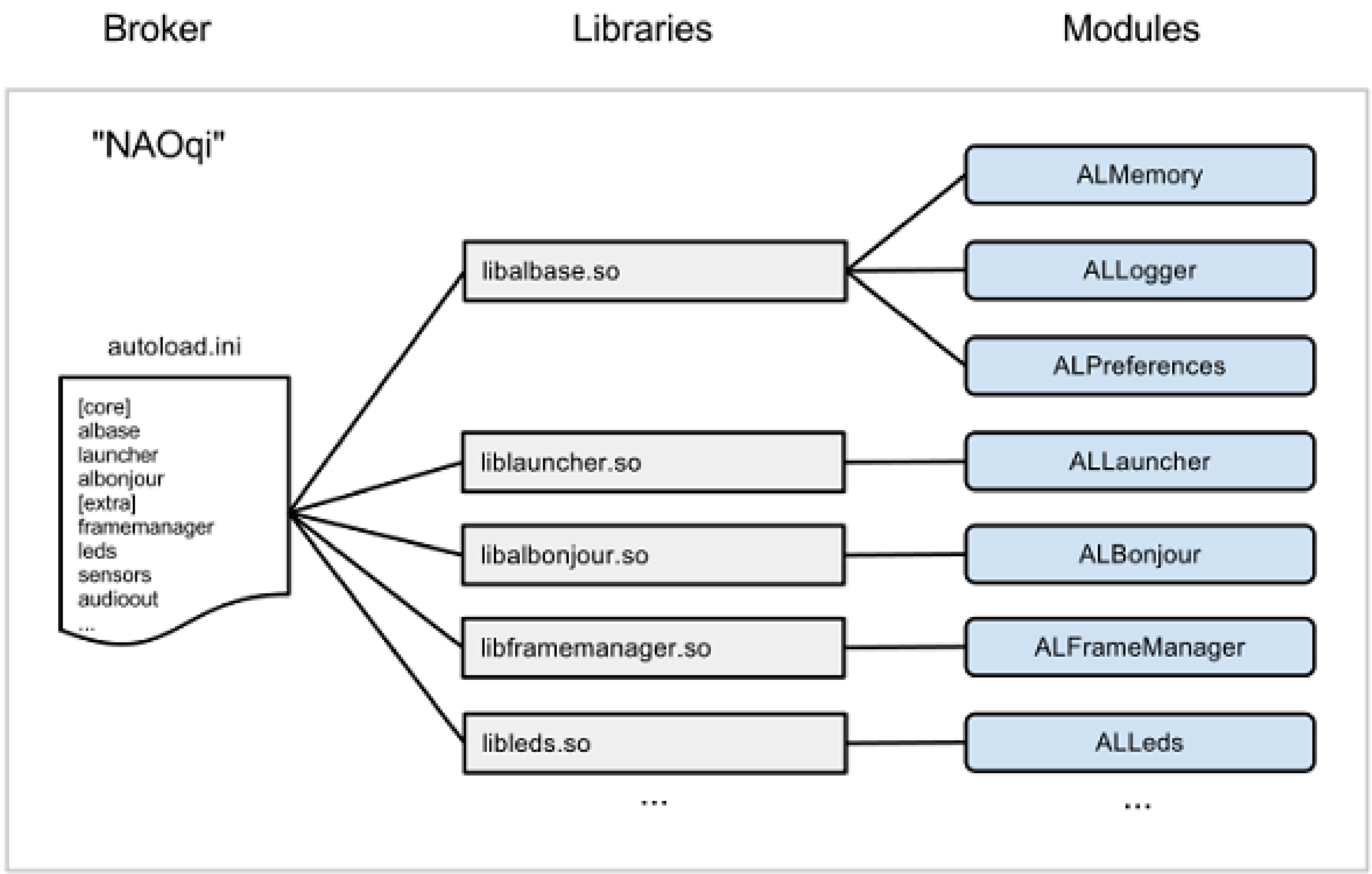
- Resolution: 1cm;
- Frequency: 40kHz;
- Detection range: 0.25m - 2.55m;
- Effective cone: 60°;



Nao Robot Software support

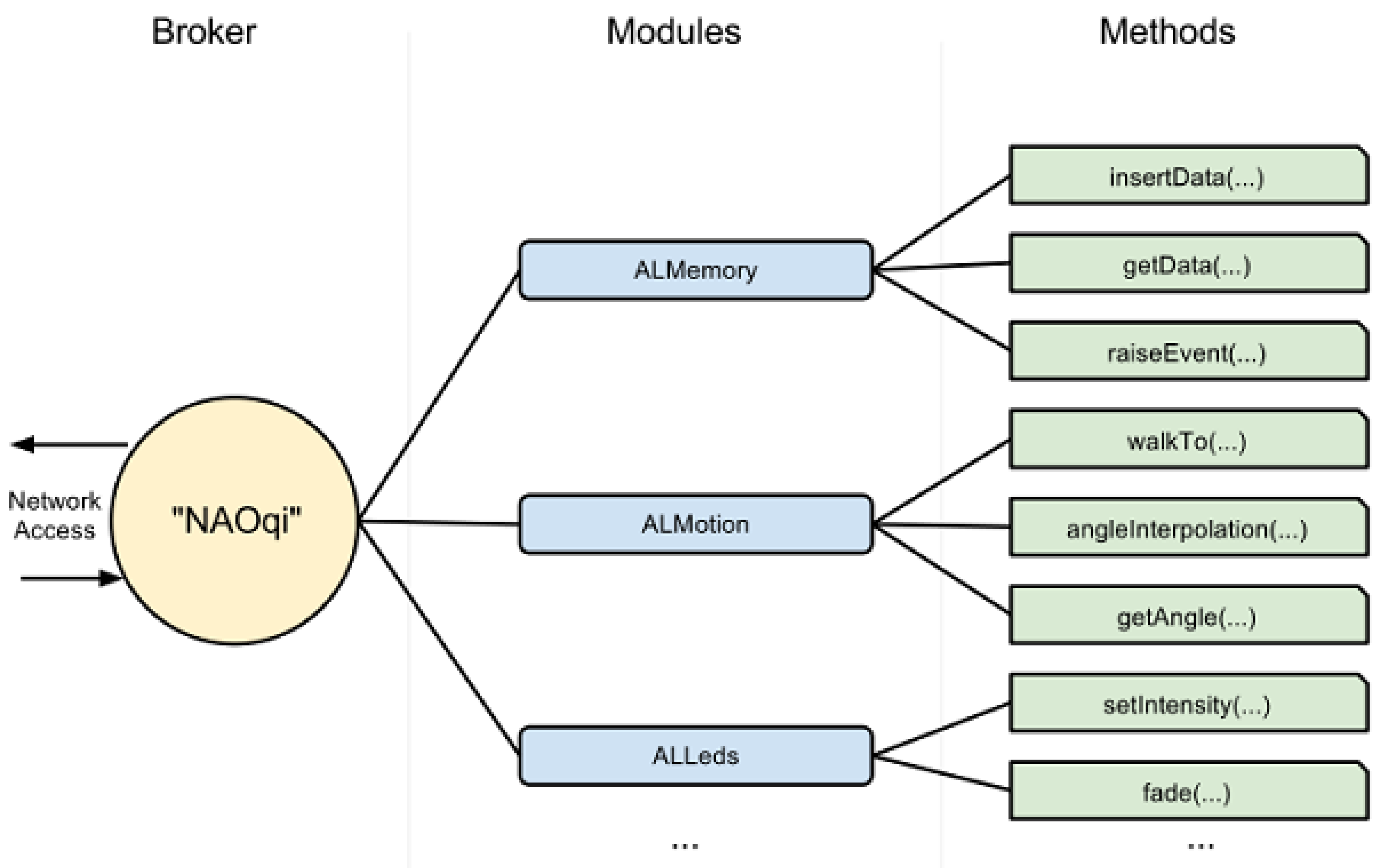


Naoqi API



https://community.aldebaran-robotics.com/doc/

Naoqi API


Broker
Modules
Methods
Network
Access
"NAOqi"
ALMemory
ALMotion
ALLeds
...
insertData(...)
getData(...)
raiseEvent(...)
walkTo(...)
angleInterpolation(...)
getAngle(...)
setIntensity(...)
fade(...)
...

Naoqi API

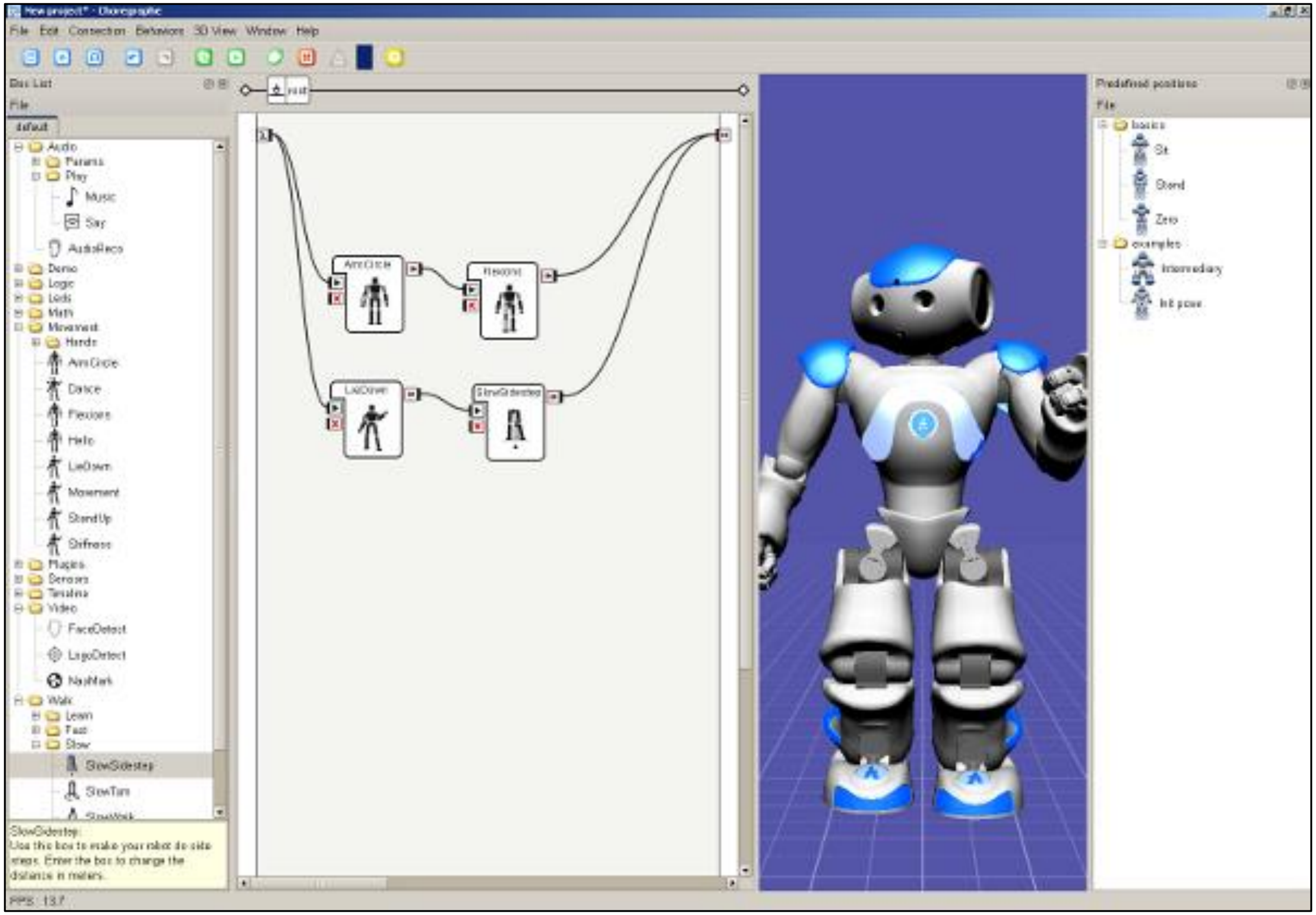
A **broker** is an object that provides two main roles:

- It provides directory services: Allowing you to find modules and methods.
- It provides network access: Allowing the methods of attached modules to be called from outside the process.

A **proxy** is an object that will behave as the module it represents.
For instance, if you create a proxy to the *ALMotion* module, you will get an object containing all the *ALMotion methods.*

A **Module** is a class within a library. When the library is loaded from the autoload.ini, it will automatically instantiate the module class.

Choregraphe



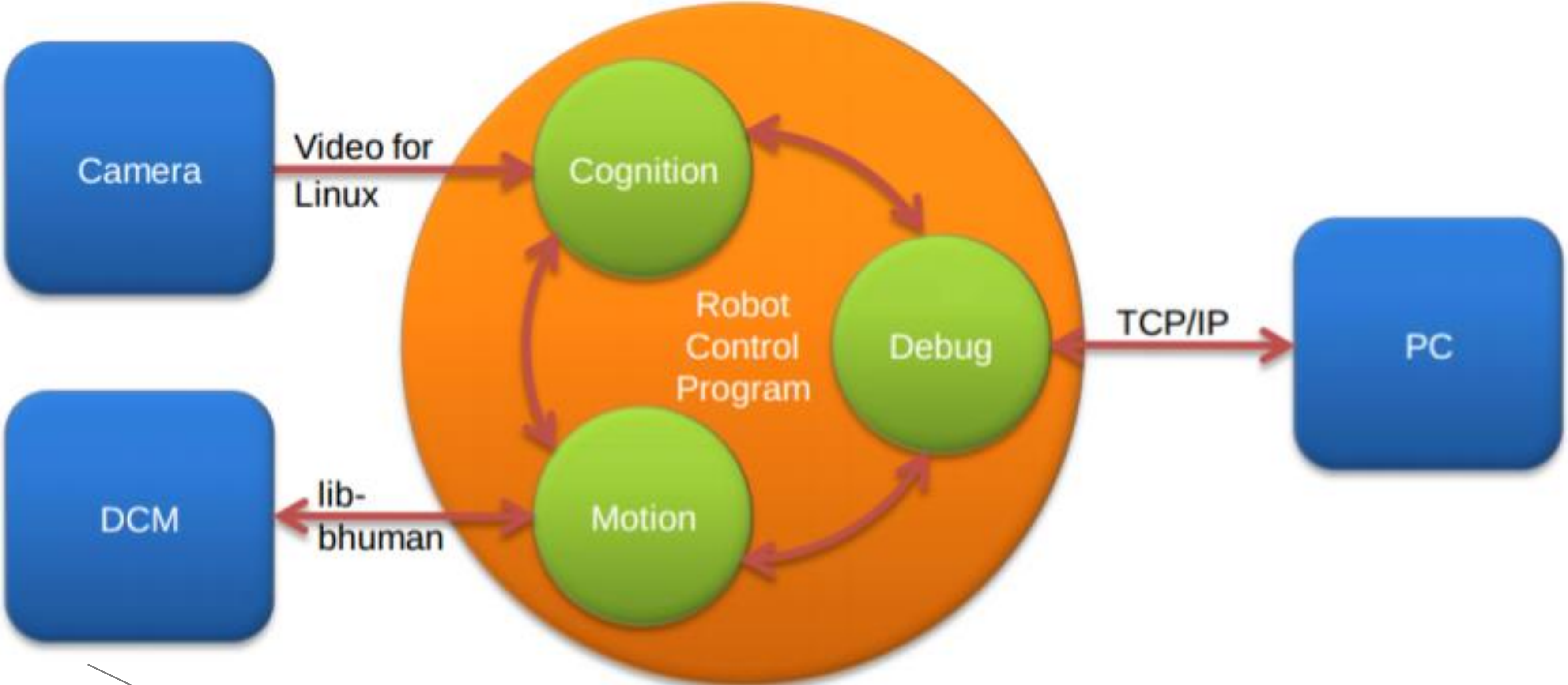
B-Human Framework Architecture

- Based on the original framework of the GermanTeam, developed by:
  - *University of Bremen;*
  - *German Research Center for Artificial Intelligence* (DFKI).

- Since 2009 used in the Standard Platform League by many teams as a base framework.

- Documentation:
  - http://www.b-human.de/downloads/publications/2014/CodeRelease2014.pdf
  - http://www.b-human.de/downloads/publications/2013/CodeRelease2013.pdf

B-Human Framework Architecture



Naoqi communication

Processes

- **Cognition**:
  - Inputs: Camera images, Sensor data;
  - Outputs: High-level motion commands.

- **Motion**:
  Process high-level motion commands and generates the target vector **q** for the 25 joints of the Nao.

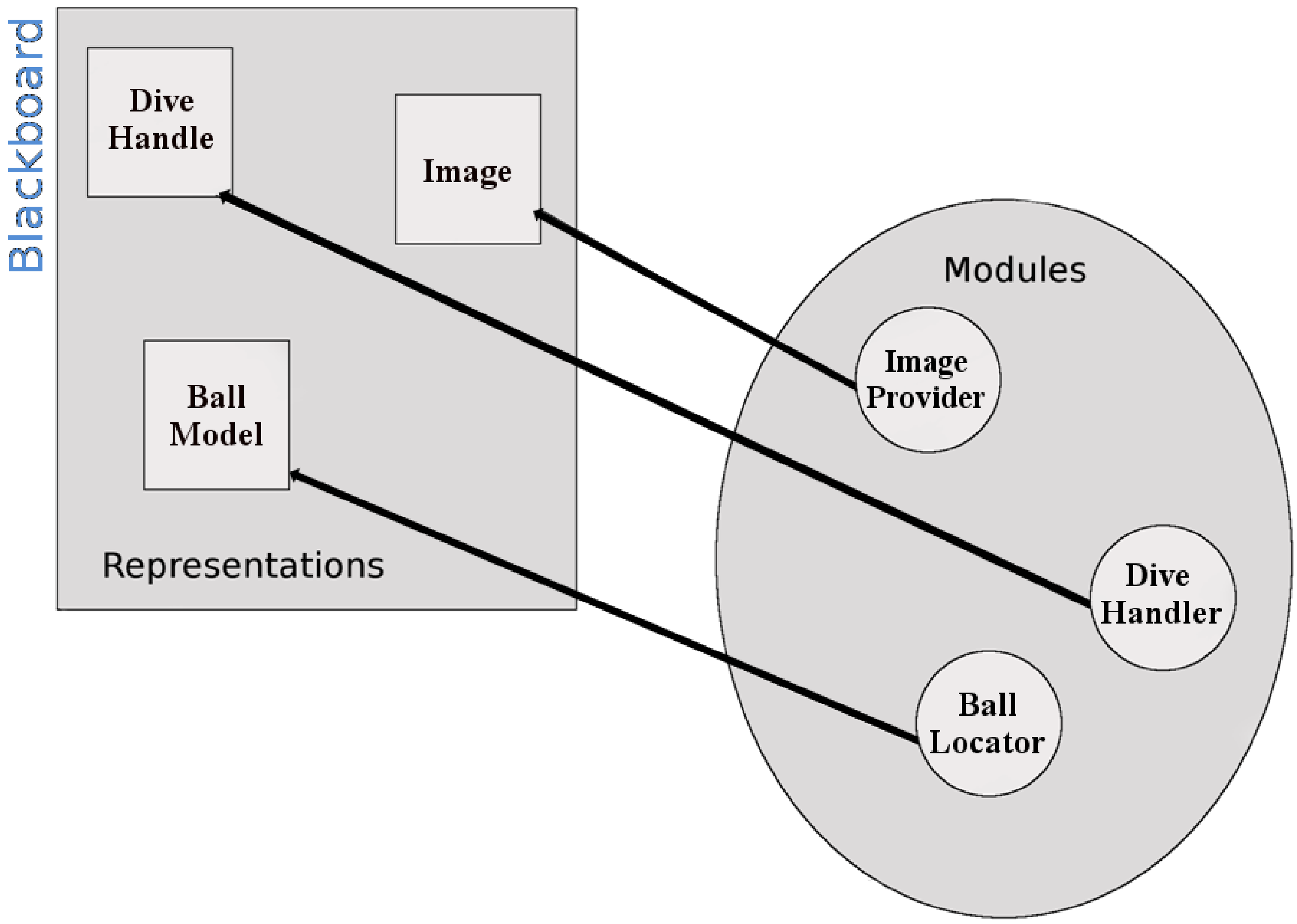
- **Debug**:
  Communicates with the host PC providing debug information (e.g. raw image, segmented image, robot pose, etc.)

Modules and Representations

- The robot control program consists of several **modules**, each performing a certain task.
- Modules usually require inputs and produce one or more outputs, i.e. **representations**.

The framework uses a **Scheduler** to automatically determines the right execution sequence, which depends on the inputs and the outputs of the modules.

Modules and Representations



Representation template

Path to representations : /spqrnao2014/Src/Representations/

```
class Foo :  public Streamable
{
  private:
    void serialize(In* in, Out* out)
    {
      STREAM_REGISTER_BEGIN;
      STREAM(a);
      STREAM(b);
      STREAM_REGISTER_FINISH;
    }

  public:
    float a;
    int b;

    Foo() :  a(0.0), b(0) {;}
};
```

Update BlackBoard

```
//...
class GlobalBallEstimation;
class Coordination;
class RobotPoseSpqrFiltered;
class Foo;
//...
class Blackboard
{
  protected:
    //...
    const GlobalBallEstimation& theGlobalBallEstimation;
    const Coordination& theCoordination;
    const RobotPoseSpqrFiltered& theRobotPoseSpqrFiltered;
    const Foo& theFoo;
    //...
};
```

Remember to update the BlackBoard.cpp

Update modules.cfg

Path to representations :
/spqrnao2014/Config/Locations/<location>/

```
representationProviders = [
  ...
  {representation = RobotInfo; provider = GameDataProvider;},
  ...
  {representation = Coordination; provider = Coordinator;},
  {representation = Foo; provider = FooModule;}
];
```

Module template

Path to modules: /spqrnao2014/Src/Modules/

Modules performs a certain task requiring specific *inputs* and providing specific *outputs*:

- **0...n** Inputs (REQUIRES or USES)
- **1…m** Outputs (PROVIDES)

It must defines an **update function** for each provided representation.

Modules template

```
#include "Tools/Module/Module.h"
#include "Representations/Perception/BallPercept.h"
#include "Representations/SPQR-Representations/Foo.h"

MODULE(FooModule)
  REQUIRES(BallPercept)
  PROVIDES(Foo)
END_MODULE


class FooModule :  public FooModuleBase
{
  private:
    //...
  public:
    FooModule();
    void update(Foo& foo);
};
```

Module template

```
#include "FooModule.h"

MAKE_MODULE(FooModule, SPQR-Modules)

FooModule::FooModule() {;}

void FooModule::update(Foo& foo)
{ if (theBallPercept.wasSeen)
  {
    foo.a = 1.0;
    foo.b = 10;
  }
  else
  {
    foo.a = 2.0;
    foo.b = 5;
  }
}
```

Scheduler

```
MODULE(A)
  PROVIDES(Foo1)
END_MODULE
```

```
MODULE(B)
  REQUIRES(Foo1)
  PROVIDES(Foo2)
END_MODULE
```

The execution order is defined by the required representations. In this case module **B** cannot be executed before **A**.

Therefore the order is **A** and then **B**

Scheduler

```
MODULE(C)
  REQUIRES(Foo3)
  PROVIDES(Foo1)
END_MODULE
```

```
MODULE(B)
  REQUIRES(Foo1)
  PROVIDES(Foo2)
END_MODULE
```

Considering input *Foo3* as available:

the order is **C** and then **B**

Scheduler

```
MODULE(D)
  REQUIRES(Foo2)
  PROVIDES(Foo1)
END_MODULE
```

```
MODULE(B)
  REQUIRES(Foo1)
  PROVIDES(Foo2)
END_MODULE
```

D cannot be executed before **B**.
B cannot be executed before **D**.

$\Rightarrow$ **Deadlock**, the code compiles but it does not execute.

How can we *discover* deadlock in the structure?

Scheduler

```
MODULE(D)
  USES(Foo2)
  PROVIDES(Foo1)
END_MODULE
```

```
MODULE(B)
  REQUIRES(Foo1)
  PROVIDES(Foo2)
END_MODULE
```

D can be executed before **B**.

Warning: *USES* macro does not guarantees that the representation *Foo2* is updated up to the last value.

Tip: pay attention to the initialization of the "used" representations

Compiler

The compiler is **clang** based, which is an open source project and it is designed to be highly compatible with gcc;

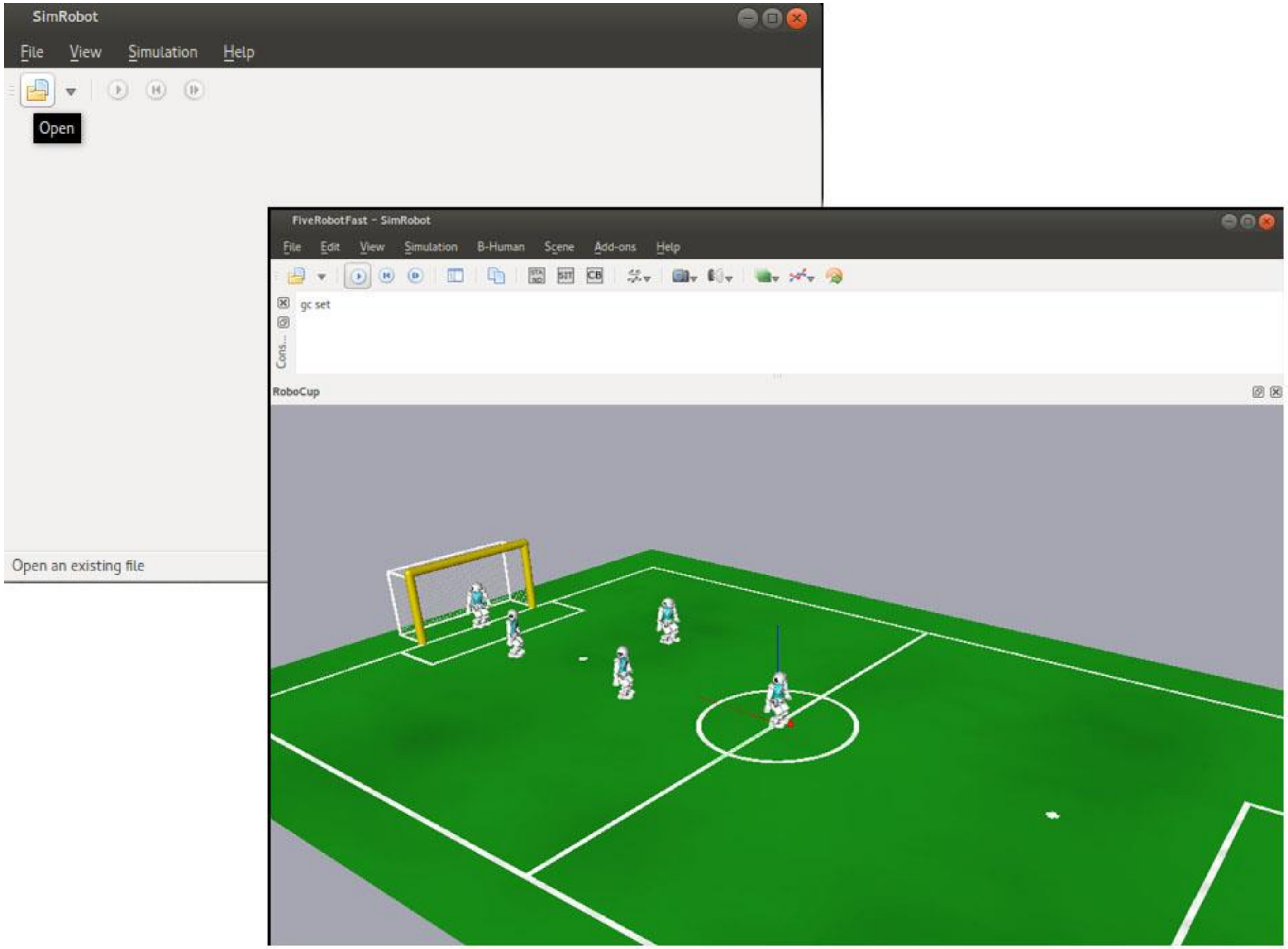
Links:

- http://clang.llvm.org/
- https://gcc.gnu.org/

The code can be compiled in different configuration:

Make CONFIG=<Debug/Develop/Release>

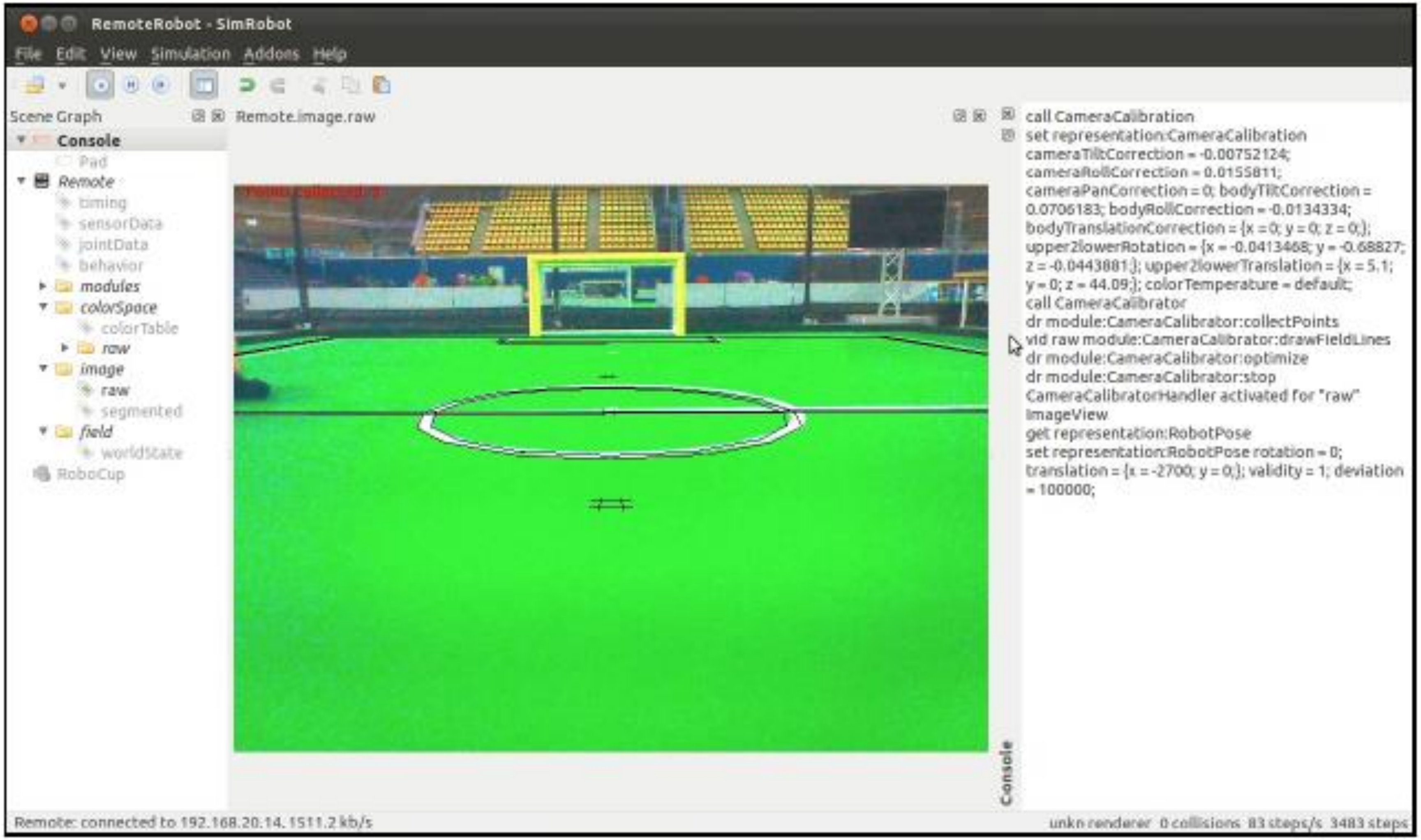
SimRobot


SimRobot
File View Simulation Help
Open
Open an existing file
FiveRobotFast - SimRobot
File Edit View Simulation B-Human Scene Add-ons Help
gc set
Cons...
RoboCup

SimRobot: functionalities

- ✓ Simulate the code;
- ✓ Connect the robot;
- ✓ Calibrate the color table;
- ✓ Calibrate the camera parameters;
- ✓ Calibrate sensors;

SimRobot: camera calibration



SPQR code: tips and useful paths

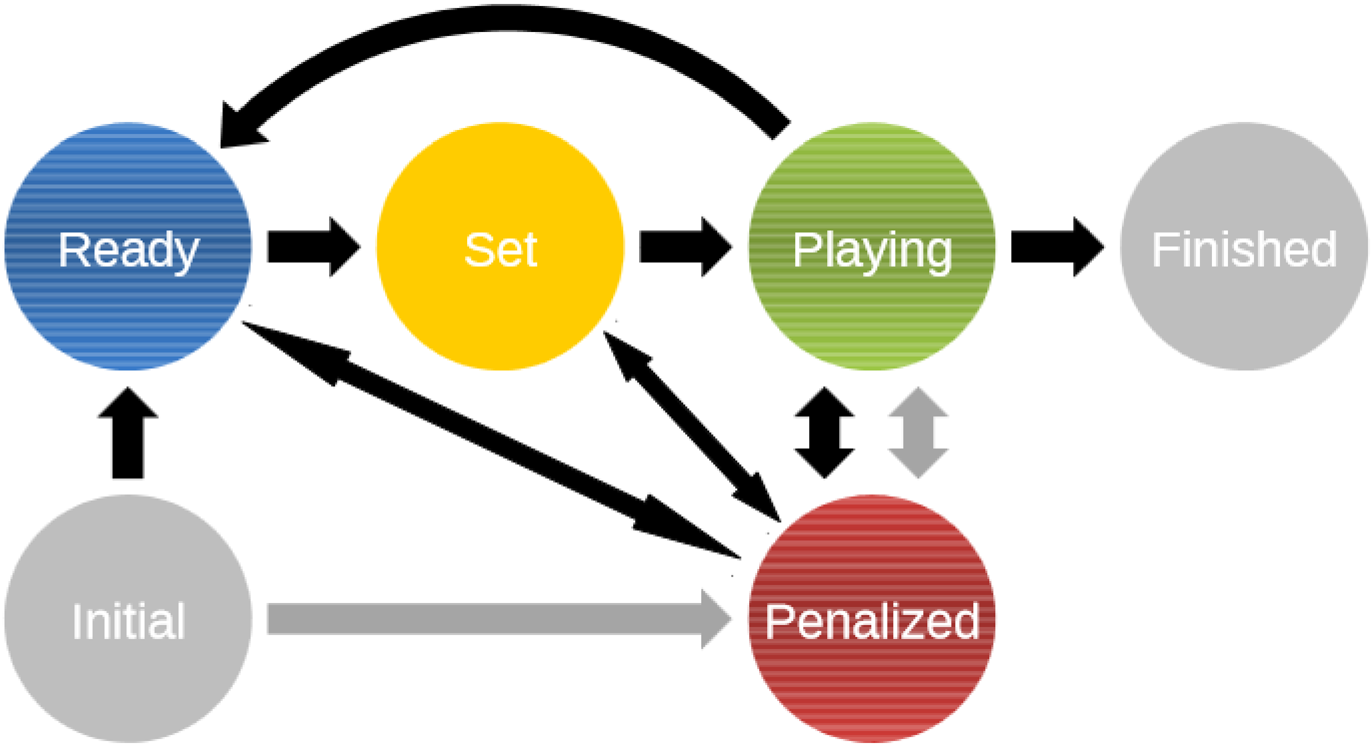
- ✓ bash_aliases;
- ✓ compile in Develop;
- ✓ Use grep: $ grep –r “<string>” .*

Paths (move to the RoboCup/spqrnao2014/ folder)

- SimRobot: Build/ SimRobot/Linux/<Debug/Release/Develop>
- Make: Make/Linux/
- Install: install/
- Scenes: Config/Scenes/
- Locations: Config/Locations/
- Behaviours: Src/Modules/BehaviorControl/
- Options.h: Src/Modules/BehaviorControl/BehaviorControl2103/Options.h

→ Look at this file if you want to add options

Game states



SimRobot console commands

- **gc ready**: the robot runs the ready behavior and gets into their default position;
- **gc set**: places the robot into the default set positions;
- **gc playing**: starts the game;
- **mr RobotPose CognitionLogDataProvider**: if you want to provide a perfect localization.

10 mins break?

C-based Agent Behavior Spefication Language (CABSL)

- It is a derivative of **XABSL: eXtensible Agent Behavior Specification Language**
- It is designed to describe and develop an agent's behavior as a **hierarchy of state machines**.
- CABSL solely consists of C++ preprocessor macros and can be compiled with a normal C++ compiler.
- A behavior consists of a set of **options** that are arranged in an **option graph**.

CABSL

Adopted by the German Team since the RoboCup 2002

Good choice to describe behaviors for autonomous robots or NPCs in computer games.

Code downloadable at: http://www.xabsl.de

CABSL: Options

General structure
CABSL comprises few basic elements: options, states, transitions, actions.

Each option is a **finite state machine** that describes a specific part of the behavior such as a skill or a head motion of the robot, or it combines such basic features.

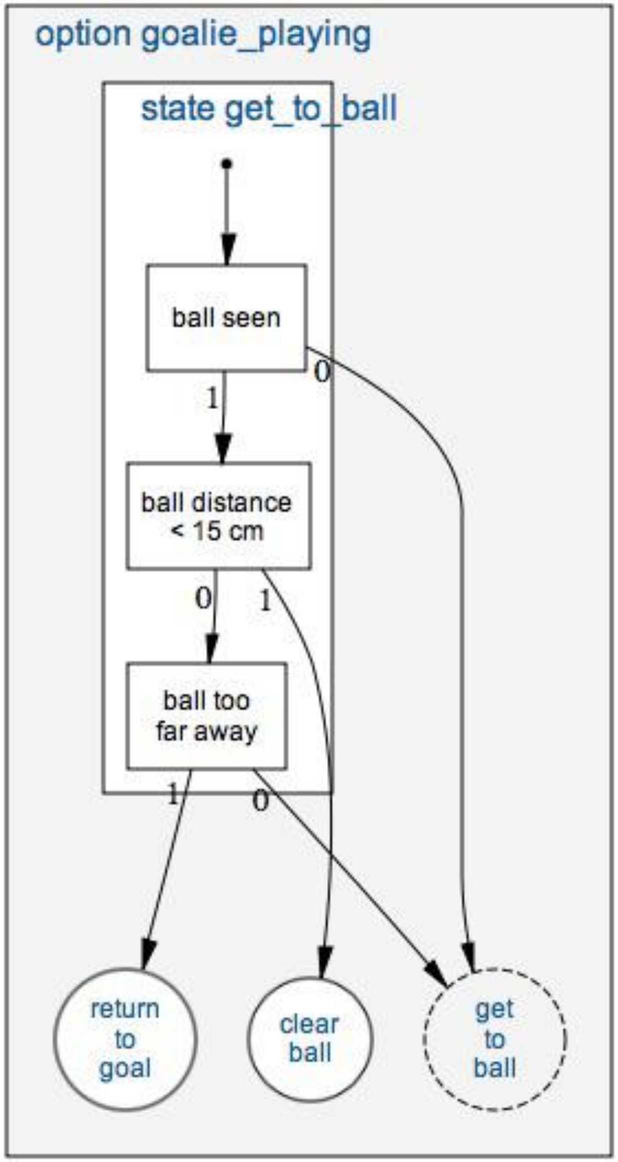
Tip: Deeply debug the inner state machine in order to avoid loops.

CABSL: Options

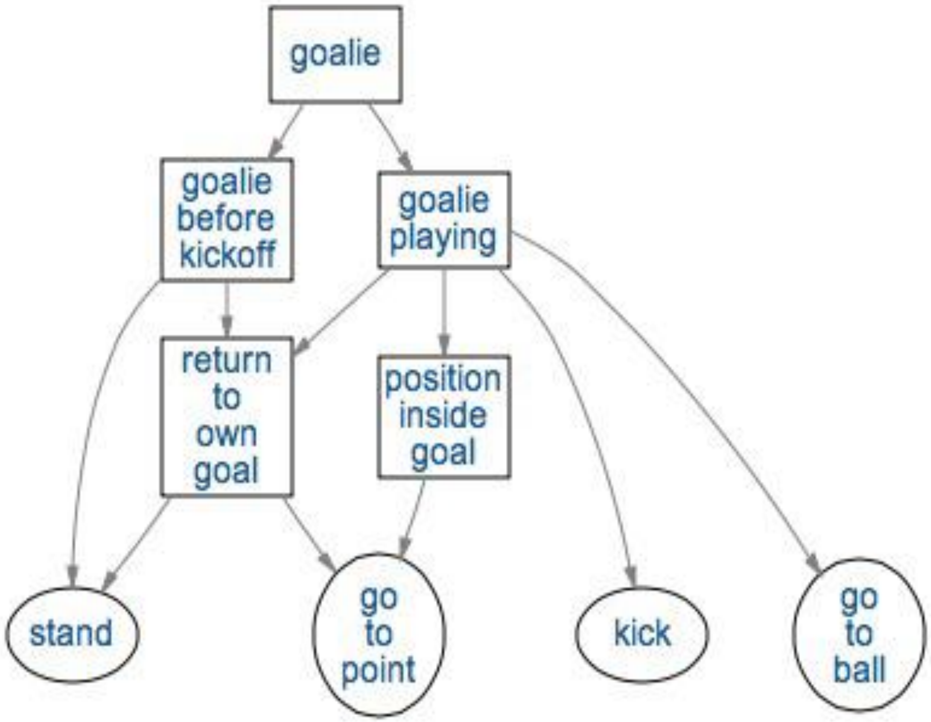
Each **state** has a decision tree with transitions to other states.

For the decisions, other sensory information (representations) can be used.

Tip: take into account how long the state has been active



CABSL



Pseudo-code:

Foreach iteration

{

the execution of the tree **starts** from the root and controls the flux of the option grah top-down;

do

{

if the transition is within the current node **continue** the execution;

else jump to the lower level;

} **until** *current_node* is a leaf node;

}

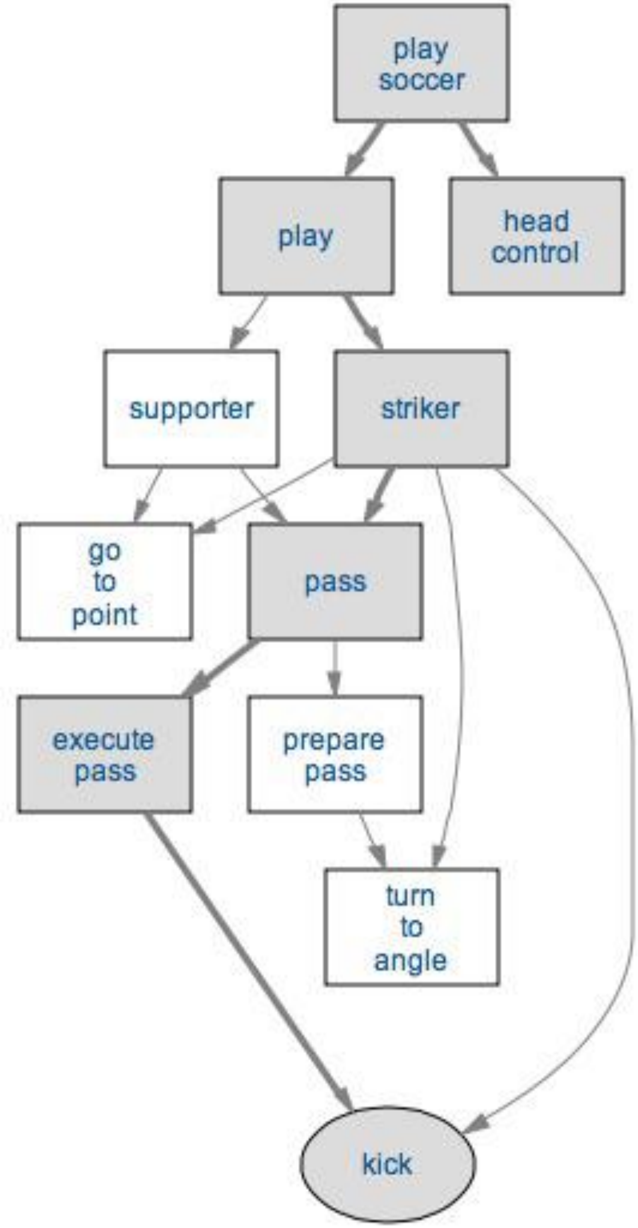
Task of the option graph:

activate one of the leaf behaviors (proceeding top-down), which is then executed.

CABSL:
Option Activation Tree

Options are activated at a specific time step from a rooted tree.

Such tree is a <u>sub-tree</u> of the more general option graph and it's called ***option activation tree***.



CABSL: Libraries

- A library is a normal C++ class, a single object of which is instantiated as part of the behavior control and that is accessible by all options.
- Libraries can have variables that keep their values beyond a single execution cycle.

```
class LibExample : public LibraryBase
{
public:
  LibExample();
  void preProcess() override;
  void postProcess() override;
  bool boolFunction(); // Sample method
};
```

CABSL examples and templates

CABSL: Options

```
option(exampleOption)
{
  initial_state(firstState)
  {
    transition
    {
      if(booleanExpression)
        goto secondState;
      else if(libExample.boolFunction())
        goto thirdState;
    }
    action
    {
      providedRepresentation.value = requiredRepresentation.value * 3;
    }
  }
```

CABSL: Options

```
state(secondState)
{
  action
  {
    SecondOption();
  }
}
```

Warning: Pay attention to this kind of states.

CABSL: Options

```
state(thirdState)
{
  transition
  {
    if(booleanExpression)
      goto firstState;
  }
  action
  {
    providedRepresentation.value = RequiredRepresentation::someEnumValue;
    ThirdOption();
  }
}
```

Parallelism through the activation graph.

CABSL: Options

```
option(OptionWithParameters, int i, bool b, int j = 0)
{
  initial_state(firstState)
  {
    action
    {
      providedRepresentation.intValue = b ? i : j;
    }
  }
}
```

Arguments can generalize the options.

CABSL: Options

```
common_transition
{
  if(booleanExpression)
    goto firstState;
  else if(booleanExpression)
    goto secondState;
}
```

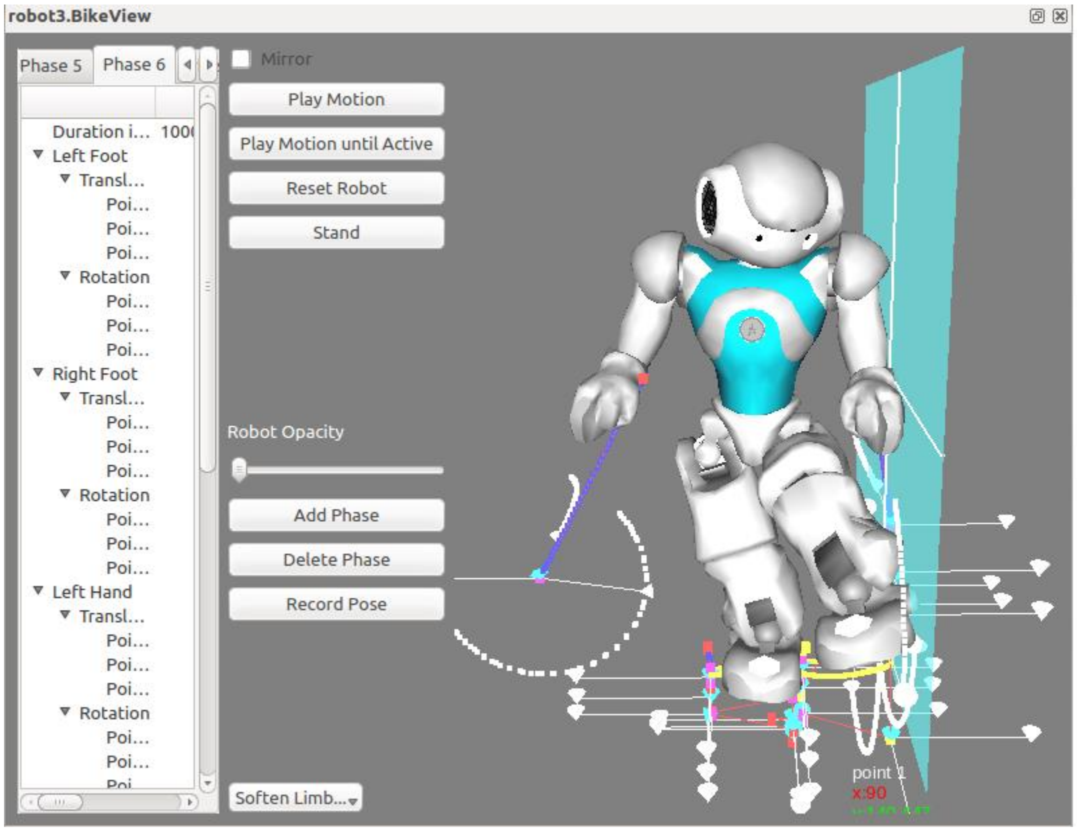
CABSL: add representations to the Behaviors Engine

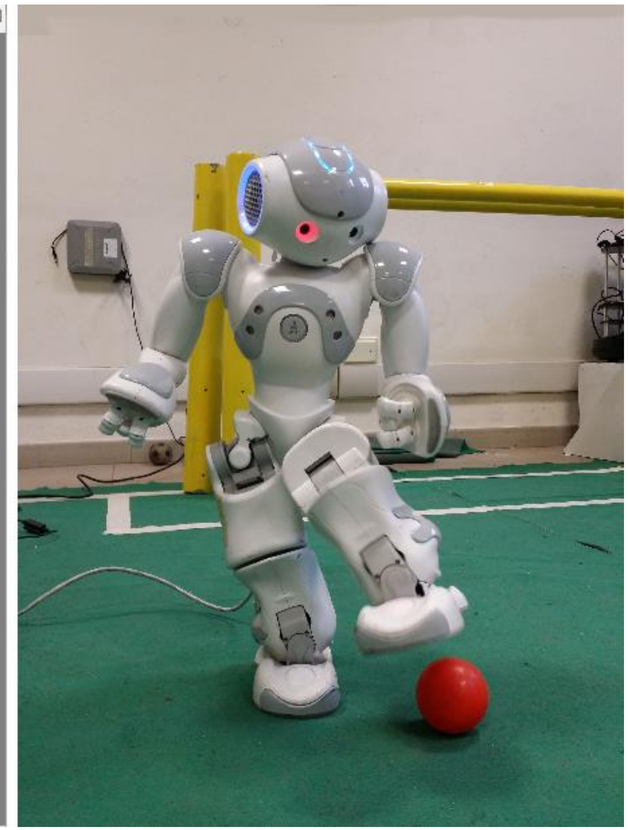
BehaviorControl2013.h <Select Symbol>

```
#include "Representations/Sensing/FallDownState.h"
#include "Representations/Sensing/FootContactModel.h"
#include "Representations/Sensing/GroundContactState.h"
#include "Representations/Sensing/TorsoMatrix.h"

#include "Representations/SPQR-Representations/ConfigurationParameters.h"
#include "Representations/SPQR-Representations/RobotPoseSpqrFiltered.h"
#include "Representations/SPQR-Representations/GlobalBallEstimation.h"
#include "Representations/SPQR-Representations/Coordination.h"
#include "Representations/SPQR-Representations/DiveHandle.h"
#include "Representations/SPQR-Representations/BallPrediction.h"

#include <Core/Processors/Processor.h>

#include <limits>
#include <algorithm>
#include <map>
#include <fstream>

MODULE(BehaviorControl2013)
  REQUIRES(GlobalBallEstimation)
  REQUIRES(RobotPoseSpqrFiltered)
  REQUIRES(Coordination)
  REQUIRES(DiveHandle)
  REQUIRES(BallPrediction)

  REQUIRES(ArmContactModel)
  REQUIRES(ArmMotionEngineOutput)
  REQUIRES(BallModel)
  REQUIRES(BallTakingOutput)
  REQUIRES(BikeEngineOutput)
  REQUIRES(CameraInfo)
  REQUIRES(CameraMatrix)
  REQUIRES(CombinedWorldModel)
```

Motion interface: Bike scene

- Bikes: spqrnao2014/Config/Bike/

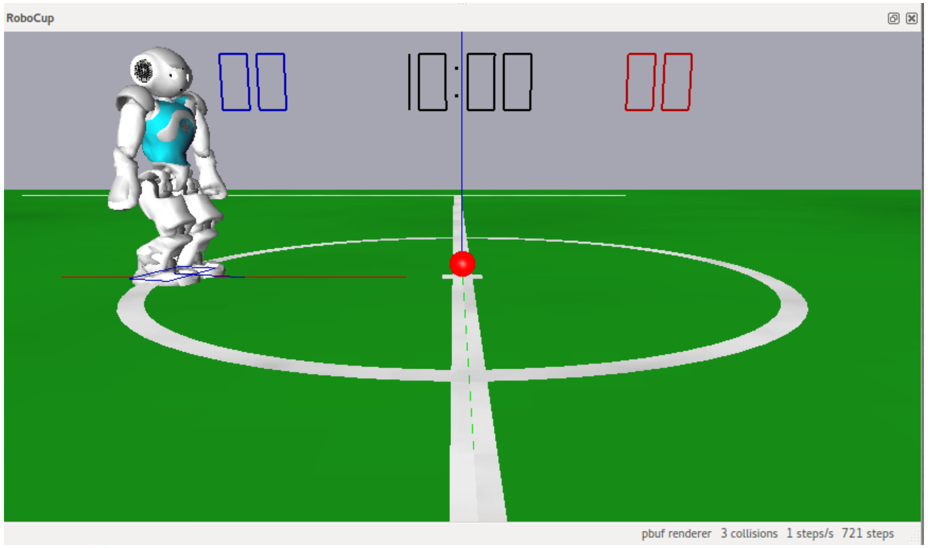


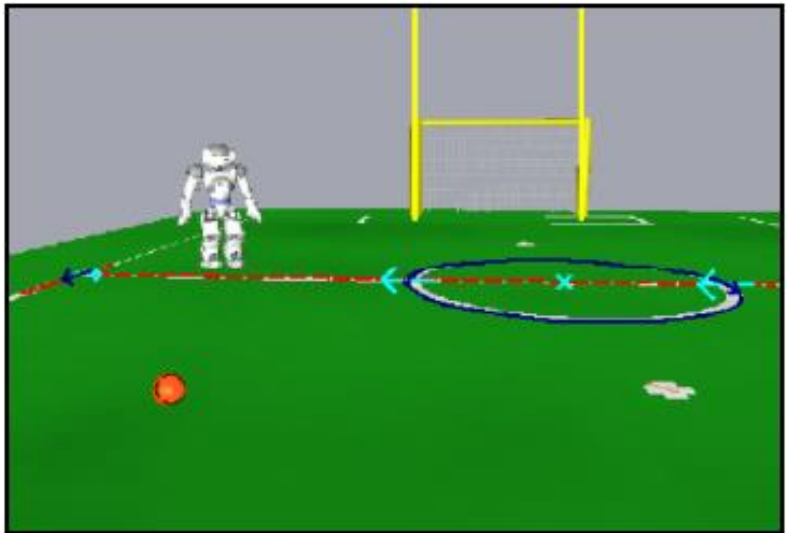


Ball recognition and evaluation

- BallPercept.h
  - USES BallModel
  - PROVIDES BallPercept
- BallModel.h
  - REQUIRES BallPercept
  - USES BallModel
  - PROVIDES BallModel

1. Evaluate ball spots;
2. Check noise;
3. Calculate ball in image;
4. Calculate ball on field;
5. Check jersey;





Homeworks

1.A Make an account on github.com, send me an email with your git username (***“[Elective RoboCup] Name LastName” as email subject***) and install the software;

1.B Create a new *Representation* and a new *Module*: the update function of the module has to display:

- the robot pose <**x, y, theta**>;
- the ball position <**x, y**> (both relative and *global*);
- joints value;

2.A Filter the ball perception and make the robot disregard balls that are more then 2 meters away from the robot;

Homeworks

2.B Use the previously written module to save some images acquired from the camera;

2.C Detect the edges contained in the pictures using OpenCV;

3.A Write a behaviour that makes the robot "WalkTo" the ball;

3.B Extend the previous behaviour and make the robot walk around the ball;

4.A Write a striker behavior that makes the robot kicking the ball towards its own goal;

4.B Test everything simulating two robots (striker and goalie).