The RoboSoc 2.8 Manual
Version 0.1

Fredrik Heintz, frehe@ida.liu.se
David Rosèn, davro399@student.liu.se

November 13, 2002

Contents

Chapter 1

Introduction

This is the second version of the user's and reference manual for the RoboSoc system for developing RoboCup agents. The manual consists of this short introduction, an overview of the system, a guide to compiling and running the system and some information and examples of the different concepts in RoboSoc.

The manual assumes that the reader knows what RoboCup is and roughly how the Soccer server works. If you do not know what RoboCup is please take a look at the official RoboCup web-site [2]. If you want to know more about the Soccer server look at its manual [1], which can be downloaded from the Soccer server web-site [4].

For the latest information about the current release of RoboSoc, look at the `README` file included in the latest release. For information about how to install the software look at the `INSTALL` file also included in the release.

To download the latest version of RoboSoc, including the latest version of the manual, or to get more information about the development of it look at the RoboSoc web-site [3].

If you have comments or find bugs or other problems please report them to Fredrik Heintz at `frehe@ida.liu.se`.

1.1 Acknowledgments

We would like to thank Anneli Dahlström, Martin Jacobsson, Johan Thapper and all the students involved in the AI Programming course given at Linköping university over the years for testing the system, fixing bugs and proof-reading the manual.

1.2 Known Bugs and Problems

A problem with the current release is that RoboSoc uses namespaces, because of a name clash between the Unix socket management and the standard template library. Therefore you need a compiler that supports namespaces, like GCC 3.x and CC 5.0. Here at IDA we currently use GCC 3.01.

The frameworks debugging features may slow down the agent. For the agent this means loosing the ability to decide what to do in some cycles. A redesign of the debug handling is planed.

With the last update of the code and the inclusion of a world model class in the framework many of the old skills, predicates and views developed for no longer work. Porting of old skills and the writing of new will be a priority during this semester.

1.3 Future Work

The debug handling and new/old skills has already been mentioned in the section above. The `TODO` file contains updated information about what's planed. More class related information can be found in the @todo tag in the class header file and this information is collected in RoboSoc online class documentation.
Some of the most important "ToDos" are:

1.3.1 Fix Synchronization

At this moment the command from the agent is always sent to the Soccer server at the middle of the current cycle. Sometimes visual information arrives very late in the cycle and since the command then already has been sent, we miss the opportunity to take the fresh visual information into account when deciding what to do.

1.3.2 Handle Errors In `Vector` And `Point`

The handling of errors is (at the moment) not implemented in the classes `Vector` and `Point`. Since these two classes are the ones most used when reasoning about the world, we should be able to reason about how confident their values are. Since `Vector` and `Point` depend on two variables the errors need to be handled a bit different from the objects using the `WithError` class.

1.3.3 More Examples And Better Documentation

The system needs more documentation and also some good examples for making the process of getting used to the system and developing own skills and agents as easy as possible.

Chapter 2

Overview

This user's manual is intended to work as a practical reference manual where you can look when you want to accomplish something specific, like creating a new view or extending an existing skill.

2.1 What Is RoboSoc?

RoboSoc is a system for creating RoboCup agents. The purpose of the system is to help people create their own RoboCup teams without having to spend a lot of time implementing the basic functionality of an agent like interacting with the Soccer server. Another goal of RoboSoc is to create a platform for supporting, encouraging and promoting the sharing of different parts of the agent, like skills and information processing algorithms.

The RoboSoc system consists of three different parts, Library, BasicSystem and Framework.

2.1.1 Library

The library contains common types, classes and utilities needed by the rest of the system. Examples are geometry classes (`AngleDeg`, `CoordPoint`, ...), server command classes (`TurnCommand`, `DashCommand`, ...) and classes for objects we need to reason about (`PlayerObject`, `MarkerObject`, ...). The Library also contain classes useful when working with (possible) unknown values, and values with different precision. This is handled using the `WithUnknown`, `WithError` and `WithUE` classes. Many of the geometry classes are able to handle this problems and has specific implementations (`AngleDegUE, PointUnknown,` ...). Information and examples about how to use RoboSoc geometry classes can be found in section 4.1, and how to use the unknown/error classes in section 4.2.

2.1.2 Basic System

The BasicSystem contains modules that takes care of the basic interaction necessary with the Soccer server. Here you find communication classes, methods for parsing sensor information, synchronization and action commands classes. The sensor information arriving from the server is handled by the `Sensors` which parses the message, creates objects and

Figure 2.1: The RoboSoc Architecture

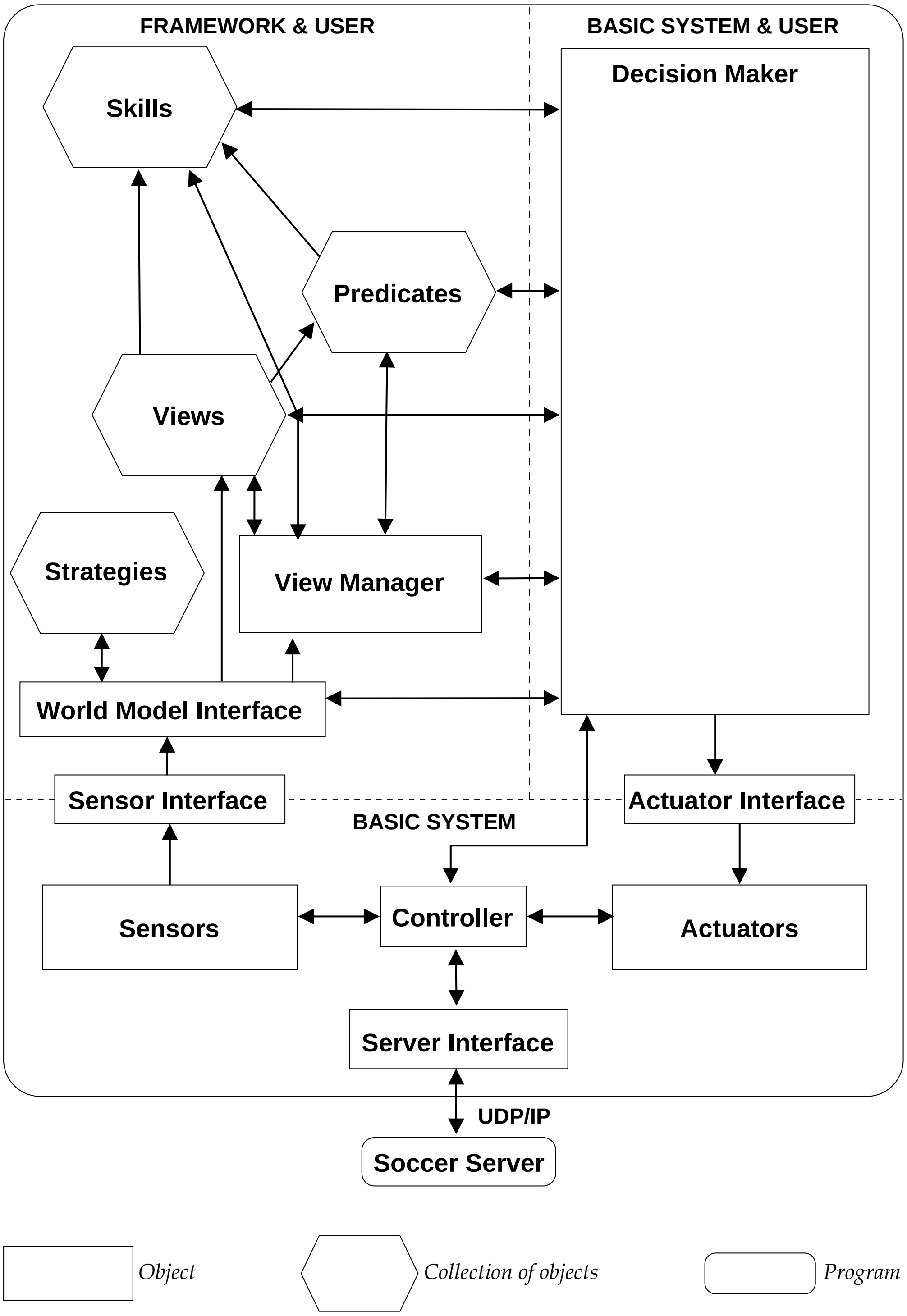
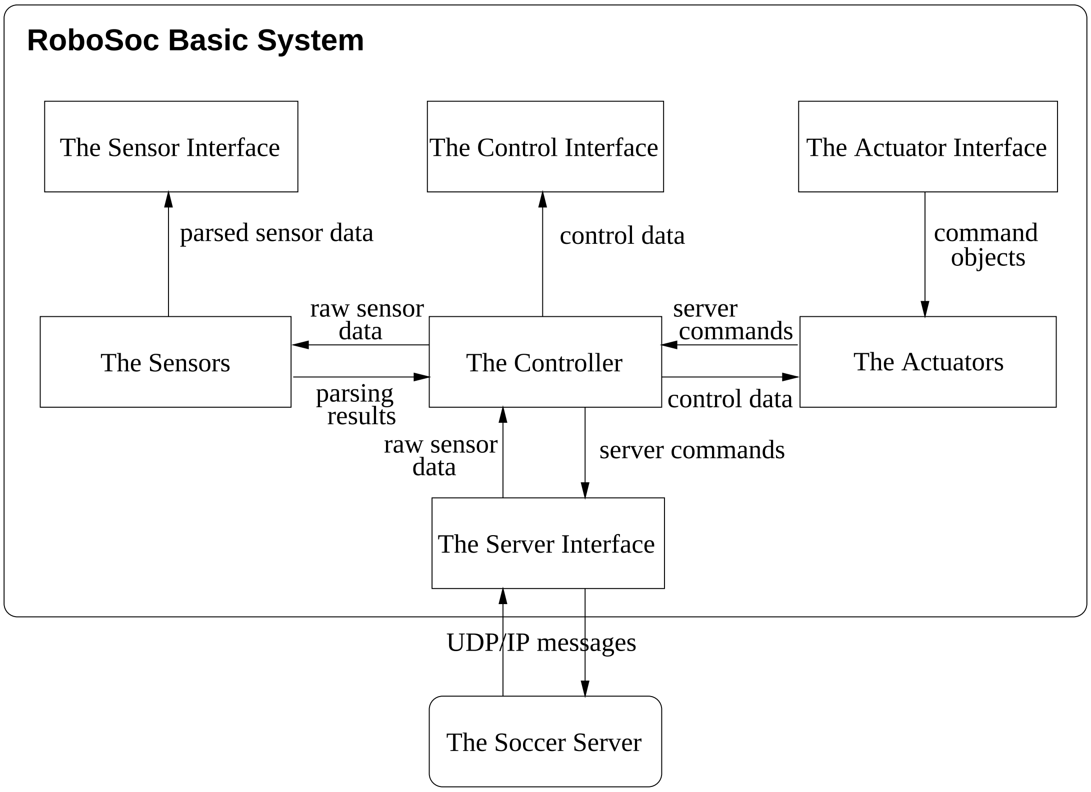


Figure 2.2: The Dataflow in the Basic System



passes this information to the `SensorInterface` for more processing. The `Controller` handles synchronization and alerts the agent's decision procedure to generate a new command at the right time. It's also responsible for checking that everything is working i.e. we receive new information and our commands are executed by the server. Another important class in the BasicSystem is `WorldFacts`. This class contains information about the world that is (mostly) fixed during a game, and known at all time. The parameters that controls the maximum speed of the ball, the size of the players etc. are stored in this class. More information about this class and how it can be used can be found in section 4.3.1.

2.1.3 Framework

The Framework define ways of creating, extending, and sharing common concepts used in the RoboCup agents. It's here the agent builds a model of the world and uses this model to make decisions about how to act. It's here most of your work with creating a soccer playing agent will take place. The Framework itself consists of four major parts: Views, Predicates, Skills and Strategies, which will be explained below.

The framework contains some important classes for the agent's world model. These classes are `WorldModel` and `WorldObservation`. After a message is parsed in the sensors and forwarded to the sensor interface, which is building a `WorldObservation` object containing the last cycles observations about the world. This object is forwarded to the `WorldModelInterface` which is using the different strategies (`BasicSeeStreategy`, `BasicSenseStrategy`, ...) found in the Strategy part of the Framework to update the agent's world model in an appropriate way. The `WorldModelInterface` contains a history of past knowledge of the world and past observations and lets the views use this information to give the agent the best possible information to make a decision from (actually there is a lot of work left to "best" at this moment), and the strategies the best possible information to update to world model from.

Strategies

The strategies are used to create or improve the world model of the agent based on new sensor information, known facts about the world and old beliefs (older world models). There are different strategies for updating the world model based on the different kinds of new information that reaches the agent. If the agent has executed a command, we let the `CommandStrategy` decide how to update the world model, if the agent has received new visual information from the Soccer server we let the `SeeStrategy` process this information and merge the new knowledge in the agent's world model. The strategies can be replaced when new data processing algorithms becomes available. For more information about the world model and the strategies see section 4.3 and section 4.4

Views

Views extract information about a special concept from the world model and let the decision procedures get easy access to this information. The views can calculate things like how long can we kick the ball, using different values (in this case, the agent's position, the ball's position, the maximum acceleration of the ball and so on). The views may later support simulation of future information but is at this moment only extracting (hidden) data from the world model. RoboSoc at the moment provides the view *AgentView*, keeps track of the agent's status, *PlayerView*, keeps track of the opponents and teammates, *GameView*, information about the current play mode, game time, etc., *BallView*, keeps track of the ball. For more information about views see section 4.5 [1]

Predicates

A predicate is a test on the state of the world at a certain point in time. They can for example be used to test whether the ball is in our penalty area at a time t. For more information see section 4.6

[1] In the 2.7.0 version of RoboSoc the views where also responsible for updating their part of the agent's world model, and you will see some relics of this in the code. The old features are still available using the `ViewManager` but the views provided by the system do not depend on this functionality. The problem with the old approach was that the building of a good world model needed information from many different views and the order which the views should update it's data where a bit complex. There were also some potential problems with planned support for simulation.

Skills

The skills are specialized short term goal directed planners. Each skill should generate plans consisting of primitive commands that will take the agent closer to the goal state of the skill when executed. The primitive skills are either simple skills or server actions. The different skills may later be divided into different layers. We may identify four different layers at this point.

- Server Actions: Skills that are the actions actually sent to the server, the classes for these commands are defined in the Library part of RoboSoc
- Single agent reactive skills: Skills that do not have to concern itself with high level information, like offside line of the opponent team, or the direction of the closes teammate. Run to a position, or turn towards a point can be examples of low-level skills.
- Single agent deliberative skills: Skills that will use the low-level skills to separate itself from the exact format of the server actions. By using the low-level skills more complex skills can be written.
- Multiagent skills: Skills should take the overall situation in account when generating a plan. A pass skill should not pass to a teammate who is offside at the moment for example.

For more information about skills i RoboSoc see section 4.7

Chapter 3

Installing and running RoboSoc

Before you can use RoboSoc you have to install the files and compile the library. How this is done is described below.

3.1 Requirements

Currently RoboSoc only supports Unix platforms with appropriate signal and socket handling. The platforms which it is tested on includes Solaris SunOS 5.7.

The software needed to install and compile RoboSoc are:

- a C++ compiler which supports namespaces, like GCC 3.x
- gunzip (to uncompress the distribution).
- tar (to unpack the distribution).
- make (to install and compile RoboSoc).
- autoconf (only needed if you want to make changes to the configure script, like adding new directories).
- automake (only needed if you want to make changes to the makefiles).
- libtool (to handle the RoboSoc libraries).
- RoboCup Client Parser (to parse the server messages).
- Soccer server (to parse the coach language).

RoboSoc works with the Soccer server communication protocol version 7.1.

3.2 Installation

To install RoboSoc you have to decompress the `RoboSoc-2.8.tar.gz` file with `gunzip RoboSoc-2.8.tar.gz`, and then unpack the tar archive with `tar xvf RoboSoc-2.8.tar`, or if you use the GNU version of tar you can do it with a single command `tar xzvf`

`RoboSoc-2.8.tar.gz`. This will unpack all the files in a new directory called RoboSoc-2.8 in the directory where you unpacked the files.

When the files are installed you have to run the configure-script in the RoboSoc-2.8 directory by executing `./configure`. If you run into problems when running the script that you are able to fix please report it to the author at `frehe@ida.liu.se` so he can include the fixes in the next release. If you run into problems that you can not fix please contact the author as well.
Example installation.

```
% gunzip RoboSoc-2.8.tar.gz
% tar xvf RoboSoc-2.8.tar
% cd RoboSoc-2.8
% ./configure
```

Now the system should be ready for compilation.

3.3 Compilation

If everything worked during the installation you should now be able to type `make` to compile the RoboSoc library. If this doesn't work try to use the GNU version of make instead, on some system called `gmake`. If neither of these work please contact the author.

This is the simplest way to compile RoboSoc, but since the author can't foresee all the uses of the system there are a number of compiler directives that can be used to customize the compilation. There are four special files containing compiler directives:
`./Library/include/machine_dependent_compiler_directives.h`,
`./Library/include/library_compiler_directives.h`,
`./BasicSystem/include/basic_system_compiler_directives.h`,
`./Framework/include/framework_compiler_directives.h`.

The directives are described in detail in section 3.5. To change the compilation either change one of the files above or give `-D` or `-U` flags to the compiler.

3.4 Environment Variables

The following environment variables is used to locate the components RoboSoc needs.

- `RSPATH` is used to locate the root directory of RoboSoc.
- `RSLIBPATH` is used to locate the RoboSoc libraries if you use a precompiled version.
- `RSINCLUDEPATH` is used to locate the RoboSoc include files.
- `RCCPARSER_INCLUDEDIR` is used to locate the RoboCup Client Parser include files. Since RoboSoc version 2.8 this is used to parse the server messages.
- `RCCPARSER_LIBDIR` is used to locate the RoboCup Client Parser libraries, *rccparser*

- `RSSS_INCLUDEDIR` is used to locate the Soccer server include files. This is needed since the RoboCup Client Parser uses the coach language parser included in the Soccer server.

- `RSSS_LIBDIR` is used to locate the Soccer server libraries, or to be more specific, the Soccer servercoach language parser library, *rcssclangparser*.

3.5 Compiler Directives

The following directives can be defined in `library_compiler_directives.h`:

- `RS_TRIG_IN_DEG`: If trigonometry functions should return angles in degrees (defined as default).

- `RS_TRIG_IN_RAD`: If trigonometry functions should return angles in radians (not defined as default).

- `RS_USE_EXCEPTIONS`: If exceptions should be used (defined as default).

- `RS_USE_DEBUG`: If the debug features should be used (defined as default).

Only one of the first two directives can be defined, otherwise the compilation will fail.

3.6 Trying FCFoo

When you compiled RoboSoc you also compiled a test agent called FCFoo. It is available in the RoboSoc`/Framework/example/FCFoo` directory. To run it you first have to start the Soccer server, then type `FCFoo -host <host>`, where *host* is the name of the computer where the Soccer server is running, the default value is localhost. Other command line options are:

- `-team_name <name>` which will set the name of the team (default is RoboSoc).

- `-goalie true` which will make the agent a goal keeper (default is false).

- `-coach true` which will make the agent a online coach for the team (default is false).

If you want to load the command line options from a file you can do this by using the flag `-f file`, there is a sample config file called `client.conf`. There are also a sample server config file called `server.conf` which include the server parameters, but since the current Soccer server sends it's parameters when a client connects and RoboSoc 2.8 handle those messages, the use of this file can be seen as deprecated.

If everything works like it's supposed to and the client can find a server to connect to you will see like the output in Example 3.1 when starting a FCFoo agent and Example 3.2 when starting a coach.

Example 3.1 Output from a test run of FCFoo started as a player

```
~/RoboSoc/Framework/example/FCFoo> ./FCFoo -team_name Testteam
Initializing the world...
Initializing the worldModelInterface...
Setting worldModel update strategies...
Starting decision...
As a player...
Initializing the FrameworkAgent...
Starting the agent...
-------------------------------------------------------------------------
Connected player 1 on team 'Testteam'
He's playing on the left side of the field.
-------------------------------------------------------------------------
```

Example 3.2 Output from a test run of FCFoo started as a coach

```
~/RoboSoc/Framework/example/FCFoo> ./FCFoo -team_name Testteam -coach true
Initializing the world...
Initializing the worldModelInterface...
Setting worldModel update strategies...
Starting decision...
As a coach...
Initializing the FrameworkAgent...
Starting the agent...
-------------------------------------------------------------------------
Connected coach on team 'TestTeam'
He's coaching the team on the left side of the field.
-------------------------------------------------------------------------
```

Chapter 4

Using RoboSoc

This section describes and gives examples on how some of the important concepts in RoboSoc works. More concrete examples can be found in existing strategies, skills, views and predicates.

4.1 Using The Geometry Classes

The geometry classes in RoboSoc is found in the Library part of the system. The classes are designed to handle the geometry in the world and are used in many parts of the system. Example of geometry classes are `Point`, `Vector` and `AngleDeg`.

When working with geometries in RoboSoc it's important to remember that all positions of object in the system are global field coordinates. The information received by the agent in the sensor messages are relative to the agent, but this is converted to the more useful global representation in the agent's strategies. There are many occasions when we want to have relative directions and vectors to objects, and methods for getting this information exist in the views. Many of the geometry classes are implemented as template classes and type defined to the name used in the code. This is good to know when looking for class documentation of a specific class based on it's name. In the current version of RoboSoc the following type definitions can be found.

```
typedef unsigned short      UInt16;
typedef unsigned int        UInt32;
typedef unsigned long long  UInt64;
typedef short               Int16;
typedef int                 Int32;
typedef long long           Int64;
typedef float               Float32;
typedef double              Float64;
typedef UInt16              Uint;
typedef Int32               Int;
typedef Float64             Float;
typedef AngleDegrees<Float> AngleDeg;
typedef AngleRadians<Float> AngleRad;
typedef CoordVector<Float>  Vector;
typedef CoordPoint<Float>   Point;
```

These definitions can be found in the file `Library/src/library_types.h` and `Library/src/machine_dependent_types.h`

4.1.1 Point

The `Point` class is used to represent a position in the world. A `Point` in RoboSoc is represented by two coordinates, x and y. A `Point` can be constructed by giving it x and y values, or giving a start `Point` and a `Vector`. Valid operations on `Points` are:

- Comparison: `Points` can be check for equality (using operators ==, ! =). A `Point` is equal to another if both x and y coordinates are of the same value.
- Selectors: To take out the different parts of the `Point` the methods `getX()` and `getY()` can be used.
- Arithmetic: `Points` can be added, subtracted, multiplied and divided by each other (using the operations +, -, *, /), this means the chosen operation is carried out on the x and y coordinates of the `Point` (So dividing a `Point` with another `Point` is not a very common operation). A `Point` can have a `Vector` added or subtracted to itself resulting in a new `Point` and can be multiplied by or divided by a scalar.
- Modifiers: There are a few different operations to modify a `Point`. `addX()` adds a scalar to the x coordinate of the `Point`, `addY()` is the same method for the y coordinate. `setX()` and `setY()` sets the x or y coordinate (leaving the other one unchanged), `addXY()` and `set()` adds or sets both the x and y coordinate of the `Point`.
- Utility: It's often useful to use a `Point` to get a distance, direction to another `Point` without the explicit construction of a `Vector` between this two `Points`. To do this the `Point` class provides the methods `getDirectionTo(Point p)` which returns the direction to the given `Point` and `getDistanceTo(Point p)` which returns the distance. To get the `Vector` between two points the method `getVectorTo(Point p)` can be used which returns the `Vector` between our `Point` and the argument.

4.1.2 Vector

The `Vector` class is used to represent vectors in the world. A `Vector` in RoboSoc is very much like a mathematical vector. A `Vector` can be constructed by specifying a direction and a distance, two `Points` (you then get the `Vector` between these two `Points`) or even by a single `Point` (the `Vector` created is then directed from origo to the given `Point`). Valid operations on `Vectors` are:

- Comparison: All comparison operators (==, ! =, <, <=, >, >=) exists and are working with the length of the `Vector`.
- Selectors: To take out the different parts of the `Vector` the methods `getAngle()` and `getLenght()` can be used.

- Arithmetic: `Vectors` can be added and subtracted to each other (using the operators +, -), multiplied or divided with a scalar which modifies the length of the `Vector` (using the operators *, /).
- Modifiers: `Vectors` can also be rotated (using `rotate()`) or stretched (using `scale()`).

In Example 4.1 you find an example using some of the features in the `Vector` class to check if the ball is catchable by the goalie and calculate the direction for the `CatchCommand`. At this point you can ignore that we're using `VectorUE` instead of `Vector`. [1]

Example 4.1 Small example how to use some features from `Vector`.

```
// Get information objects.
WorldFacts* facts = WorldFacts::instance();
AgentView* agentView;
BallView* ballView;
getView(agentView, AGENT_VIEW_ID);
getView(ballView, BALL_VIEW_ID);

// Get the vector between the goalies position and the ball.
VectorUE goalieToBallVector(agentView->getPosition(), ballView->getPosition);

// Check if the ball is in the goalies catchable area.
if (goalieToBallVector.getLength() <= facts->SP_catchable_area_l) {

  // Rotate the vector according to the current body direction of the goalie.
  VectorUE relativeBallVector(goalieToBallVector.rotate(-agentView->getBodyDirection()));

  // Get the angle to do the catch.
  AngleDegUE catchAngle(relativeBallVector.getAngle());

  // Do things...
  }
// The ball is not catchable by the goalie, do other things...
```

4.1.3 AngleDeg

The `AngleDeg` class is used to represent an angle in degrees in RoboSoc. The angle is normalized to be in the interval [-180, 180] after all operations. Valid operations on angles are:

- Comparison: All comparison operators ($==, !=, <, <=, >, >=$) exist. If you're worried about comparing values near the discontinuity using `abs()` might be a good idea.
- Selectors: The methods `getDeg()` and `getRad()` returns the value of the angle in degrees or radians.

[1] If you do want to actually use the information about if the ball is catchable there exist a method in BallView named isBallCatchable() which is what you want to use.

- Arithmetic: Angles can be added, subtracted, multiplied and divided by another angle or a scalar (using operators +, -, /, *). You can also use the functions `sin(AngleDeg)`, `cos(AngleDeg)` and `tan(AngleDeg)` on angles. The arc-functions will return a RoboSoc angle and is called `ASin(T val)`, `ACos(T val)`, `ATan(T val)` and `ATan2(T val, T val2)`.

In Example 4.2 you find a small example using the geometry classes to find out if a agent can kick the ball so far that there is a chance of scoring.

Example 4.2 Example using some geometry classes.

```
// The code checks if the ball (assumed not to be moving) can
// be kicked as far as the opponent's goal.

// Get information objects.
WorldFacts* facts = WorldFacts::Instance();
AgentView* agentView;
BallView* ballView;
getView(agentView, AGENT_VIEW_ID);
getView(ballView, BALL_VIEW_ID);

// Check if ball is kickable.
if (ballView->isBallKickable()) {
  // Calculate the maximum kickpower.
  AngleDegUE relBallDirection = ballView->getRelativeDirection();
  FloatUE relBallDistance = ballView->getRelativeDistance();
  FloatUE kickPower = facts->SP_kick_power_rate *
                      (1.0 - .25 * relBallDirection / 180 - .25 *
                       (relBallDistance - facts->SP_player_size - facts->SP_ball_size)
                       / facts->SP_kickable_margin);

  // Calculate the ball's acceleration
  FloatUE acceleration = kickPower * facts->SP_maxpower;

  // See how far we can kick.
  FloatUE ballSpeed = acceleration;

  // Compute how far we can kick ("formule de suite geometrique")
  FloatUE howFar = ballSpeed * ((1-pow(facts->SP_ball_decay,50)) /
                               (1-facts->SP_ball_decay));

  if (howFar >= agentView->getPosition().getDistanceTo(facts->getTheirGoal())) {
     // We can kick the ball as far as their goal.
  }
  // Do something else.
}
// Ball is not kickable, do something else.
```

4.2 Using The `Unknown` And `Error` Classes

One of the (many, many) problems for a RoboCup agent is that the information about the world is not complete and often contain some error. The RoboSoc system handles this using the classes `WithUnknown`, `WithError` and `WithUE`. When we have a value that isn't sure to be known we use some class derived from the `WithUnknown`, when a value isn't known exactly we use some class derived from the `WithError` class. At almost all occasions we are facing both of these problems, the value may be unknown, or known but have an error. The `WithUE` class is constructed to handle this problem. In RoboSoc all values that are of an `UE` type has this property and when used has to be checked if it's unknown before trying to use the value. If you try to take the value when it's unknown an exception is thrown.

The standard primitive classes and RoboSoc geometry classes are extended to handle this problems by introducing the following classes:

- `Float` is extended to `FloatUnknown`, `FloatError` and `FloatUE`.
- `Int` is extended to `IntUnknown`, `IntError` and `IntUE`.
- `UInt` is extended to `UIntUnknown`, `UIntError` and `UIntUE`.
- `AngleDeg` is extended to `AngleDegUnknown`, `AngleDegError` and `AngleDegUE`.
- `Point` is extended to `PointUnknown`, `PointError` and `PointUE` .
- `Vector` is extended to `VectorUnknown`, `VectorError` and `VectorUE`.

Some of these classes are instantiations from the `WithError`, `WithUnknown`, `WithUE` template (`Float`, `Int`, `UInt`) and some are classes (`AngleDeg`, `Point`, `Vector`). The type definitions for these classes can be found in `Library/src/library_types_with_unknown.h`, `Library/src/library_types_with_error.h` and `library_types_with_ue.h`.

One important thing to remember when reasoning about an agent's decision is to keep in mind that not all information about the world is known. If we want to run to the ball, we have to make sure that the agent's position and body direction is known (It should be all the time though) and the ball's position is known. If we try to use a value that is unknown an `UnknownValueException` will be thrown. The error information can be seen as extra information and in simple decision making it's is not necessary to take it into account, feel free to ignore it completely. There are some useful methods common for all `Unknown`, `Error` and `UE` classes. This methods are explained below.

4.2.1 The `WithUnknown` Interface.

Classes that implements this interface is `IntUnknown`, `UIntUnknown`, `FloatUnknown`, `AngleDegUnknown`, `PointUnknown`, `VectorUnknown` and all `UE` classes.

- Tests: There are two methods for checking if a value is unknown `IsUnknown()` and `IsKnown()`.

- Comparison: All comparison operations used in the value class can be used in the `Unknown` implementation of the class. In size comparison on a unknown value the return value is always false. This means that if obj1 is unknown and obj2 known, both obj1 $<$ obj2 and obj1 $>=$ will return false. Two objects are considered equal if the values are equal or both objects are unknown.

- Arithmetic: All arithmetic operations allowed in the value class can be used in the `Unknown` implementation. If any of the operands in a arithmetic operation is an unknown value the result is also unknown.

- Selectors: If the object is known the actual value can be received from the various `getValue()` classes. This function changes name in the different implementations, for `AngleDeg` the name is `getDeg()` for `Point` it's `getPoint()` and for `Int`, `UInt` and `Float` it's `getValue()`.

- Other: Other methods implemented in the value class is also working in the `Unknown` implementation of the class, but if the value needs to be known an `UnknownValueException` is thrown if the value is unknown.

In Example 4.3 you find a small program demonstrating an agent checking if it can catch the ball and how to do it. As you see the uncertainty is handled in the various `isKnown()` and `isUnknown()` calls.

Example 4.3 Example using some features from the `Unknown` and `Error` classes.

```
// Get information objects.
WorldFacts* facts = WorldFacts::instance();
AgentView* agentView;
BallView* ballView;
getView(agentView, AGENT_VIEW_ID);
getView(ballView, BALL_VIEW_ID);

// Check if the agent is a goalie.
if ( agentView->isGoalie() &&
// Check if the agent's position is known.
     agentView->getPosition().isKnown() &&
// Check if the ball's position is known.
     ballView->getPosition().isKnown() &&
// Check if the ball is catchable
     ballView->isBallCatchable() ) {
  // Calculate angle for the catch.
  AngleDegUE catchAngle(ballView->getRelativeBallDirection());
  // Note that since the position of the agent and the ball is known this angle
  // has to be known as well since it doesn't depend on anything but these two
  // positions. We can check it anyway in this example.
  if (catchAngle.isUnknown()) {
    // Something very strange has happened...
  }
  else {
    Float catchAngleValue(catchAngle.getDeg());
    // Use the catchAngleValue...
  }
}
// The ball is not catchable by the player. Do something else.
```

4.2.2 The `WithError` Interface

The error information is extra information about a value and can be striped at any time, without you having to worry about exceptions or strange things happening. At this point there is no implementation of `PointError` and `VectorError`. Constructors for the class includes creating a value without error, with specified upper and lower bounds and getting the bounds of a quantize value from the server. Classes that implements this interface is `IntError`, `UIntError`, `FloatError`, `AngleDegError` and all `UE` classes (except `PointUE` and `VectorUE` at this moment).

- Comparison: All comparison operations used in the value class can be used in the `Error` implementation of the class. The comparison is based on the actual value of the object and is not affected by any error information.

- Arithmetic: All arithmetic operations allowed in the value class can be used in the `Error` implementation. The error information updated with these operations. When using nonlinear arithmetic operations (like sin, cos, etc.) the error value may be distorted, please be careful if you trying to use the error values after such an operation.

- Selectors: The object can be stripped from error information using the functions `getValue()` for the `Error` classes and `getValueWithUnknown()` for the `UE` classes. `getMax()` and `getMin()` can be used to get the bounds of a value. `getPlus()` and `getMinus()` can be used to get the distance from the actual value to the upper and lower bound. Note that these functions may throw a `UnknownValueException` in the `UE` classes if the value is unknown.
- Improvement: The methods `improve()` and `disimprove()` can be used to improve of worsen a value. Improving means that the resulting upper bound is the minimum of the two objects upper bounds and the resulting lower bound is the maximum of the objects lower bounds, i.e. you take the intersection of the two intervals. Disimproving has the opposite effect, i.e. you take the union of the two intervals.
- Other: Other methods implemented in the value class is also working in the `Error` implementation of the class.

4.3 The Agent's World Model

The agent's world model is created and updated through the agent's strategies based on sensor information received from the Soccer server. This information is available by using the views to extract knowledge from the world model or by using the `WorldModelInterface` `getWorld()` method directly. In the world model you find information about the agent itself, the teammates, the opponents, the ball, and some other useful information about the agents beliefs about the world. See the class documentation about `WorldModelInterface` and `WorldModel` for available methods.

The world model is returning `const` references to the different objects in the world. The thought behind this is that the agents world model should only be changed using the different strategies based on new observations. The main objects of the world can be found in the `Library` part of RoboSoc and builds a small class hierarchy as seen in Figure 4.2.

4.3.1 `WorldFacts`

The world model is complemented with the `WorldFacts` class which stores the different server parameters, marker positions, team names etc. This class is implemented as a singleton since there should be only one and it's used in many different parts of the system. `WorldFacts` is initialized first from command line options, defining critical things like the hostname of the Soccer server and if the agent should be a coach or a goalie. When this is done the server parameters are set to the standard values used in the Soccer server. When the agent connects to the Soccer server it receives the values of the changeable server parameters and updates the values in the `WorldFacts` accordingly. Apart from server parameters the class also contains information about the different markers in the world, this information can be accessed using the `getMarker(MarkerID name)`. In `WorldFacts` you also find information about the current score, `getOurScore()` and `getTheirScore()`,

Figure 4.1: The World Model Overview

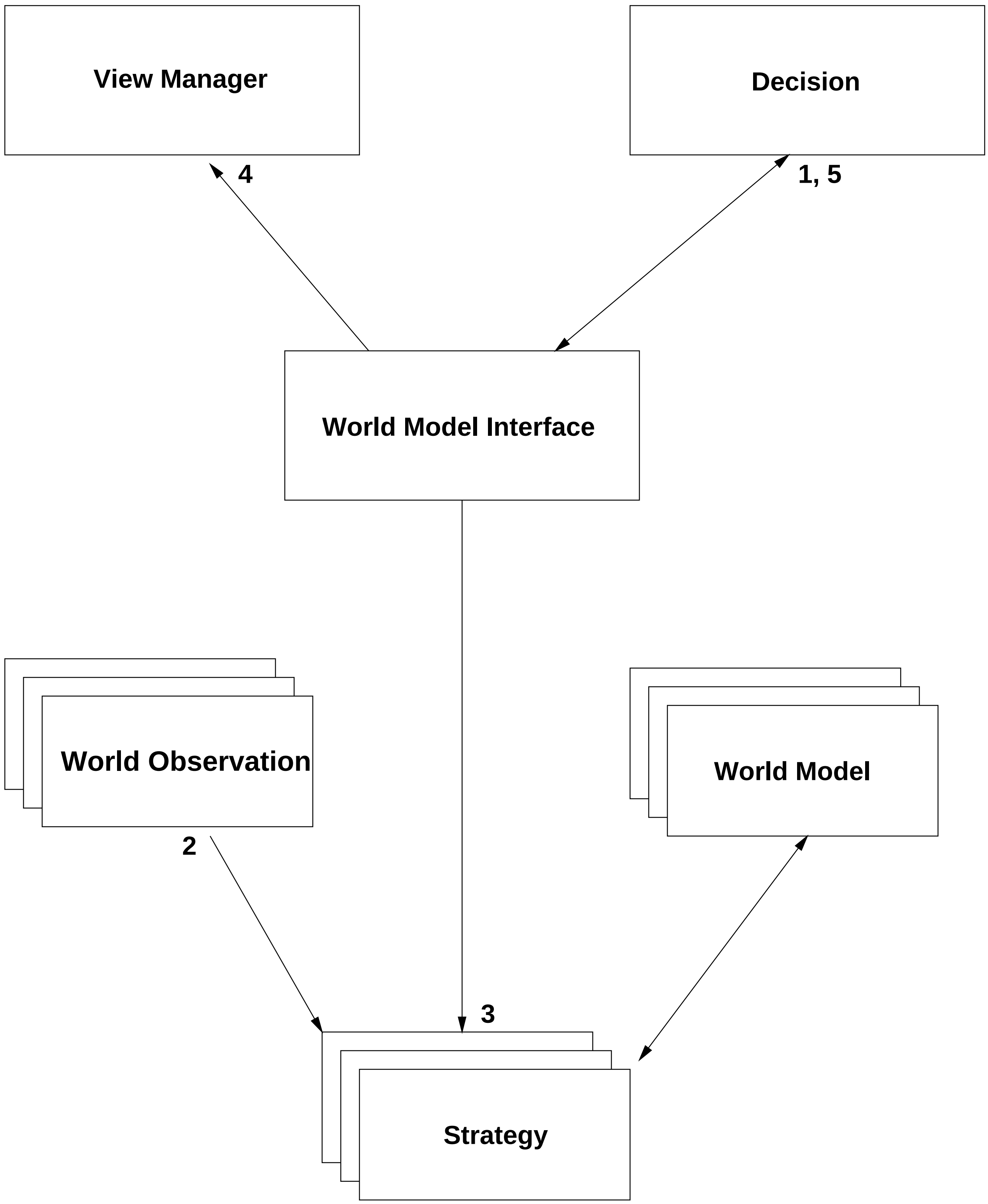
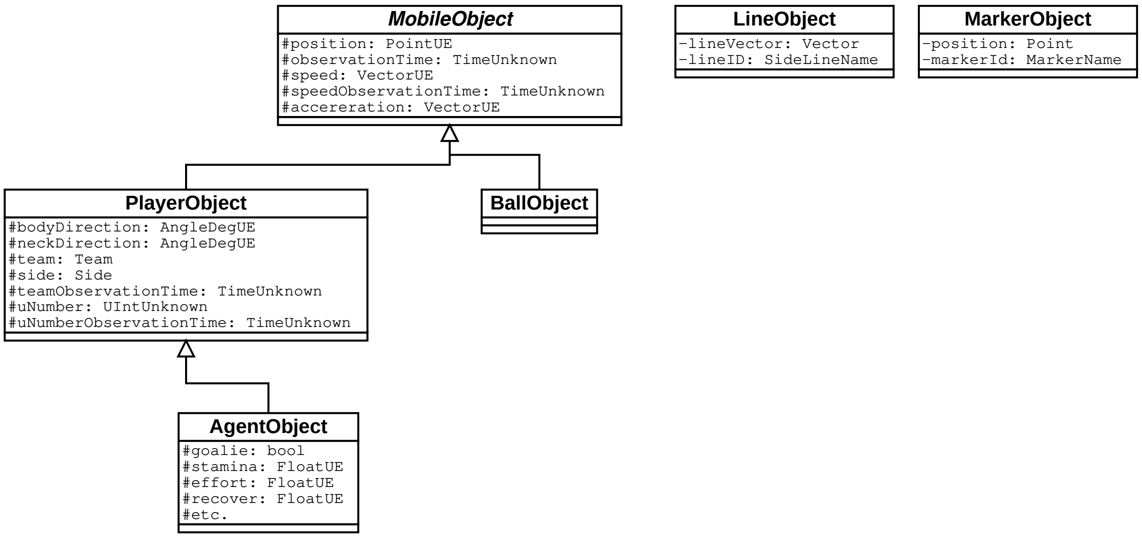


Figure 4.2: Class hierarchy of game objects



the current side of the teams, `getOurSide()` and `getTheirSide()`, the agent's uniform number, `getUniformNumber()` and the names of the teams.

4.4 Strategies

With RoboSoc version 2.8 a central world model is introduced and the information in the old views where moved to this class. To update the world model based on new information from the Soccer server the concept of strategies where introduced. The update strategies are set using the set-Strategy methods in `WorldModelInterface`. Currently there are eight different strategies used in RoboSoc:

- `SeeStrategy` is maybe the most important strategy. It's called when we get new visual information from the Soccer server. This new visual information is merged with the current beliefs to create a new, better world model.

- `HearStrategy` is called when we hear another player. When implementing a team which uses communication it's often useful to be able to update the agents world model based on what has been said. To do this write your own hear strategy and enable your team to benefit from communication.

- `CommandStrategy` is called when the agent has issued a command to the Soccer server. Since we know some things about how an agent's actions modifies the world we can modify our world model based on the actions, this is done by the `CommandStrategy`.

- `SenseStrategy` is called when the agent receives new sense information. The sense information contain information about the agent's current speed, stamina, etc. The world model, and especially the `AgentObject` part of the `WorldModel`.

- `InitStrategy` is called when we get the initialization message from the Soccer server.

- `CoachMessageStrategy` is called when we get a message from our coach. This enables teams to build there own support for different kind of coach messages in the same way that `HearStrategy` provides for messages from other players.

- `CoachInformationStrategy` is called when the coach receives complete information about the world. This is used to update the world model so the coach can use the same framework as the agents (with the difference that the information is complete and flawless) to make decisions.

- `NewCycleStrategy` is called when we detect a new cycle. The objects move according to speed and collisions are resolved. This strategy tries to do the same thing to the agent's world model as the server do to the world when a new cycle is issued.

In `Framework/strategy/` the basic update strategies can be found. These are called `BasicSeeStrategy`, `BasicHearStrategy`, etc. Look at these for hints on how to write your own update strategy.

4.5 Views

The purpose of the views is to extract, process and store information. To make this possible they need to know when new information has arrived and they need to be able to access the world model and the past observation objects. The `View` base class has a pointer to the `WorldModelInterface` to be able to receive this information. If information from other views are needed it's reachable through the view manager pointer, `viewManager`, stored in each view.

In order to know when new information is available the views have a method for each type of update in the system. They are:

- newCycle is called by the view manager when a new server cycle is about to begin. It's called before the updateBeforeNewCycle() method of the sensor interface is called. This is the last chance to use any data stored in the sensor interface, afterwards it will be reset.

- updateAfterInit is called when the init message has arrived from the server.

- updateAfterSee is called when new see information has arrived from the server.

- updateAfterHear is called when new audio information has arrived from the server.

- updateAfterSense is called when a new sense body message has arrived from the server.

- updateAfterCommand is called when a command was sent to the server.

4.5.1 Using The Views

First of all you need the wanted view object, or a pointer to it. Then you can call it's public methods and access its public attributes.

To get a pointer to a view object you need to ask the view manager if the view is stored there. This is done by calling the `getView` method. The argument to `getView` is the id-number of the view you need. Symbolic constants for the id-numbers of the views can usually be found in `Framework/view/view_constants.h`. The `getView` method will return either a NULL-pointer if the view doesn't exist or a pointer to the object if it exists. The problem is that the pointer is of the type `View*`, and not the correct view-pointer you need. Therefore you have to use the `dynamic_cast` template to convert the pointer from one type to another. Eg. to convert v_ptr from a `View*` to a `BallView*` you do:

```
BallView* b_ptr = dynamic_cast<BallView*>(v_ptr);
```

To simplify this there is a set of templates called `getView` which will take a pointer variable of the desired view type (in the above case `BallView*`) and also the `ViewId` of the desired view, so that it can fetch the view from the `ViewManager`. Ex:

```
BallView* b_ptr; getView(b_ptr, BALL_VIEW_ID);
```

4.5.2 How Do I Create A View Object?

First of all, create an instance of the view you need. Then you need to store the view in the view manager. This is done by calling the view manager's `addView()` method with the address of the view and a persistence flag. If the persistence flag is set the view will not be destroyed when no one is using the view, otherwise it will be destroyed when no one is using it. With no one using the view I mean that there are no pointers to the view active in the system. An active pointer to a view is given when you ask the view manager for a view with the `getView()` method and they are deactivated with a call to the `releaseView()` method. The reason for storing the view in the view manager is to make them available to the rest of the system.

4.5.3 How Do I Extend Views?

The basic rule is to create new views which uses information from the view you want to extend instead of changing the code of the existing view. If you really would like to modify an existing view, inherit from the view you want to extend and then add the things you miss and redefine the necessary methods.

4.5.4 How Do I Write My Own Views?

The best way to add new functionality regarding information processing in RoboSoc is to create a new view. To create a new view all you have to do is to inherit from View (or any other view that you want to use as your starting point) and write your own update methods. I.e. you need to define what the view should do when new information has arrived and what to do when a new cycle begin. If you need to store data (which is very likely) you will need to create the necessary data structures and initialize them in the constructor, destroy them in the destructor and supply methods to extract and use the information stored in them. If you need access to other views, the best way to do this is to store pointers to those views in the object and initialize them in the constructor. Then you'll know that the views you want are always accessible.

For examples and hints, look at the code in `Framework/view` that are included in the RoboSoc-package.

4.5.5 What Views Are Included In RoboSoc?

Currently four views are supplied with RoboSoc 2.8.
The views are:

- GameView: which stores information about the status of the game (play mode, score and current time).
- AgentView: keeps track of the agent's status (absolute position, stamina, effort, recover, absolute face direction, body and neck directions)
- BallView: keeps track of the ball (absolute position, relative distance and direction and a very rough estimation of the speed)
- PlayerView: keeps track of the other players. You will get access to three structures: teammates, opponents and unknown players. Each will give information about absolute position, relative distance and direction, body, face direction and uniform number (if available).

Note that in RoboSoc version 2.8 the supplied views are not updated from the `ViewManager` and contain no data of their own. All the relevant information is extracted from the world model.

4.5.6 How Are The Views Updated?

When new information arrives to the `WorldModelInterface` and the strategies has done their job with it, the `WorldModelInterface` will tell the view manager that it is time to update after XXX (which is the type of the information). The view manager will then generate an unique update number and iterate over all the stored views and for each one of them call their `updateAfterXXX()` method with the update number as the argument. This method will check to see if the view has already been updated or not. If it has not been updated then it will call the method `myUpdateAfterXXX()`. If the view has been updated nothing happens, if the view detects a circular dependency it will abort the program.

If your view needs updated information from some other view then you need to call its `updateAfterXXX()`-method with the same update number you got, which is stored in the view as currentUpdateNumber.

4.6 Predicates

A predicate is a test on the state of the world at a certain point in time.

4.6.1 How To Use Predicates

Create an instance of the predicate you want. Call the `evaluate()` method, which will return a `FuzzyBool`, i.e. a continuous value between 0 and 1, where 0 is certainly false and 1 is certainly true otherwise the truth value is somewhere between false and true.

4.6.2 How To Create Your Own Predicates

Create a new class that inherits from the `Predicate` class. Implement the `FuzzyBool evaluate()` method.

4.6.3 How To Extend Existing Predicates

Create a new class that inherits from the predicate you want to extend. Call its `evaluate()` method if its functionality is useful, otherwise rewrite it from scratch.

4.7 Skills

The skills are specialized short term goal directed planners.

4.7.1 Using Skills

Create an instance of the skill you want. To check whether the skill is applicable call its `applicable()` method which will return true if it is applicable otherwise false (a `FuzzyBool` is actually returned). To get the next command generated by the skill call its `generatePlan()` method. If the skill wants to be called again the next cycle it will set the persistence flag of the skill. This flag is checked with the `persistent()` method.

4.7.2 How To Create Your Own Skills

Create a new class that inherits from the `Skill` class. Implement the `FuzzyBool applicable()`, `FuzzyBool persitent()` and `Plan generatePlan()` methods.

4.7.3 How To Extend Existing Skills

Create a new class that either inherits from the skill you want to extend and then use its `applicable`, `persistent` and `generatePlan` methods to create the new skill or inherit from some other skill (or `Skill`) and create a local instance of the skill you want to extend and calls its `applicable`, `persistent` and `generatePlan` methods to implement the class.

4.8 Using The Debugging And Logging Features

When working with a system as large as RoboSoc many things can go wrong and features for logging during the match is essential. To handle this RoboSoc provides the classes `LogStream`, `LogArea` and the macro `RS_LOG`. Since you usually don't want to write all logging information to `std::cout` the logging classes allows information to be divided into categories, called LogAreas, and information written to files named by the user. These categories can be turned on and off so that information not interesting at the moment do not slow down the rest of the system. All LogAreas also have a parent LogArea, if a LogArea is turned off, so is all it's children.

4.8.1 Different LogAreas

The different LogAreas in the system are defined in the file Library/src/LogAreaName.h. Some important areas and their function are:

- LA_ROBOSOC is the top LogArea of the RoboSoc logging.
- LA_WARNING is used to log warning messages produced by RoboSoc. This can be that the agent has failed to send a command the past cycle, or other important things. As default this information is also echoed to std::cerr.
- LA_SENSOR_DATA is used to log all the server data received by the agent.
- LA_STRATEGY is used to log which strategies that are running, and what they do.
- LA_COACH is used by the coach to log information. It can be interesting to log the exact positions of objects and that information is always available to the coach.

4.8.2 Logging Is A Time Consuming Process

Remember that logging large amounts of information takes a lot of time. Logging some of the areas in RoboSoc may create several MB of data and writing this to a file or std::cout takes time. This may cause the agent to fail sending a command to the server in time, and therefore loosing cycles. If you see lots messages looking like:

```
Missed action(s)! Last action time = #Time(0, 100)
```

Try to deactivate some LogAreas and see if that solves the problem. Especially logging to std::cout and std::cerr is expensive.

Chapter 5

Creating Agents

To create an agent using the RoboSoc framework you have to create a main function where you start the different parts. In Example 5.1 you can see how the code for a basic agent might look like.

Example 5.1 An example of a basic agent using the framework.

```
#include "framework_compiler_directives.h"
#include "view_constants.h"
#include "FrameworkAgent.h"
#include "WorldModelInterface.h"
#include "FCFooDecision.h"

int main(int argc, char** argv) {
  try {
    std::cerr << "Initializing the world...\n";
    RS_NAMESPACE::WorldFacts::init(argc, argv);

    std::cerr << "Creating the decision maker...\n";
    RS_NAMESPACE::FCFooDecision decision;

    std::cerr << "Initializing the FrameworkAgent...\n";
    RS_NAMESPACE::FrameworkAgent agent(&decision);

    std::cerr << "Starting the agent...\n";
    agent.start();

  } catch (RSException &excpt) {
    std::cerr << "Caught an exception: " << excpt.What() << std::endl;
    return 1;
  }
  return 0;
}
```

5.1 The Main Loop In RoboSoc

As you see in Example 5.1 the last thing done in `main()` is to call `agent.start()`. The method initialize the `Controller` and the decision maker, starts the controller and finally calls `Controller.run()`. The code from `FrameworkAgent.start()` can be found in Example 5.2.

Example 5.2 `FrameworkAgent.start()`.

```
bool FrameworkAgent::start() {
  RS_LOG( LA_FRAMEWORK_AGENT, "Initializing the controller..." );
  if ( !controller->init() ) {
    RS_FATAL("The controller could not be initialized!");
    return false;
  }

  RS_LOG( LA_FRAMEWORK_AGENT, "Initializing the decision maker..." );
  if ( !decision->init(actuatorInterface) ) {
    RS_FATAL("The decision maker could not be initialized!");
    return false;
  }

  RS_LOG( LA_FRAMEWORK_AGENT, "Starting the controller!" );
  if ( !controller->start() ) {
    RS_FATAL("The controller could not be started!");
    return false;
  }

  RS_LOG( LA_FRAMEWORK_AGENT, "Running the controller!" );
  controller->run();

  return true;
}
```

5.1.1 The Controller

What's happening in `Controller.start()` and `Controller.run()` one might wonder. The system is driven by two different kinds of events. When new information from the Soccer server is received we want to parse it, and update our world model based on this new information, and before the cycle ends we want to send a command to the Soccer server, this is currently done in the middle of the cycle. This is done using two signals, `SIGIO` indicating that new information has been received, and `SIGALRM` which is generated every 10ms and lets us check if it's time to send a command to the server. These signals are initialized and started in `Controller.start()`. The code for `Controller.start()` can be found in Example 5.3.

Example 5.3 `Controller.start()`.

```
bool Controller::start() {
  // Set the start flag, the internal timer will be started
  // after the first sense_body message.
  startClock = true;

  sigprocmask(SIG_UNBLOCK, &sigiomask, NULL);
  sigprocmask(SIG_UNBLOCK, &sigalarmask, NULL);

  // Initialize the decision maker,
  // and send an init message to the server.
  ServerCommand* cmd = theControlInterface->initAgent();
  if ( cmd == NULL )
    RS_ERROR("[Start]: no init command generated by the decision!");
  else {
    RS_LOG( LA_CONTROLLER, "Sent init message: " << cmd->serverString() );
    sendCommand(cmd);
  }

  return true;
}
```

When the system is initialized and the signal handlers has been set up, the system enters `Controller.run()` which will loop as long as the server is alive. This method waits for `SIGIO` or `SIGALRM` and then handles the events using the methods `handleSigIO()` and `handleSigAlarm()`. The code for an implementation of `Controller.run()` can be found in Example 5.4.

Example 5.4 `Controller.run()`.

```
int Controller::run() {
  while ( serverAlive() ) {
    // Wait for IO or ALARM
    if ( pause() == -1 && errno != EINTR )
      RS_ERROR("Something went wrong in pause!");
    while (IOFlag || AlarmFlag) {
    // Handle IO
      if (IOFlag) {
        IOFlag = false;
        handleSigIO();
      }
      // Handle Alarm
      else if (AlarmFlag) {
        AlarmFlag = false;
        handleSigAlarm();
      }
    }
  }
  return true;
}
```

5.1.2 The Signal Handler Methods

The actual signal handlers are called directly when the signal is received, but since the system can be busy updating the world model or choosing the right action at the moment, the actual signal handlers only set the `IOFlag` or `AlarmFlag`. When the system is ready to handle the event, one of the methods `handleSigIO()` or `handleSigAlarm()` is called.

`handleSigIO()`

When new information is available from the Soccer server this method is called. First the decision maker is notified that new data is to be processed, by calling the `onBeforeSensorData()` method. The information is then read and forwarded to `Sensors` for parsing. The `Sensors` uses RoboCup Client Parser to parse the information and builds server objects of the data (`ServerMarkerObject`, `ServerAgentObject`, etc.). When a message is parsed the analyzed information is forwarded to the `SensorInterface`. Based on which type of sensor information that was received the controller calls the different on-Data methods in the decision, a complete reference of these methods can be found in Table 5.1. More about this event driven interface can be read in section 5.2

`handleSigAlarm()`

When an alarm signal has been detected it's time to control if it's time to send a command. First the decision maker is notified that a new tick is to be processed. Then we advance our tick counters, check if there are some problems, i.e we have missed actions or the estimated server time is different from the actual server time. If we have passed the limit `ticksBeforeCommandWarning`, which can be given as a client parameter when starting the agent, the decision maker is notified that it's time to decide what to do. The method `onCommandWarning()` is called. When the decision maker has issued a command the `handleSigAlarm()` is almost finished and the last thing done is that decisions `onAfterTick()` is called.

5.2 How To Do The Decision Making?

The initialization of the agent is done in the `Decision` class, and since the base class `Decision` does not send any init message or other commands to the server you have to write your own decision class.

To add the decision making you have to the following. Create a new class that inherits from `Decision`. You have to implement some of the event handlers that the `Controller` will call. The event handlers are described in Table 5.1. For an example implementation of a basic agent look at the basic decision class in Example 5.5. More examples can be found in `FCFoo` decision classes `FCFooDecision` and `FCFooCoachDecision`.

Table 5.1: The events available to the decision maker.

Event	**Generated**
BeforeTick	Before a tick of the internal clock
AfterTick	After a tick of the internal clock
BeforeSensorData	Before starting to receive sensor data
AfterSensorData	After receiving sensor data
ActuatorSensorData	After receiving actuator feedback
AuralSensorData	After receiving aural sensor data
PhysicalSensorData	After receiving physical sensor data (`sense_body`)
VisualSensorData	After receiving visual sensor data
Init	After receiving init data
SensorError	After sensor errors
NewCycle	When a new cycle has started
EstimatedNewCycle	When the controller estimates that a new cycle has started
CommandWarning	When it is time to send a command before it is too late
DelayedActions	When a sense_body indicates that more actions were performed than was sent the previous cycle
MissingActions	When a sense_body indicates that less actions were performed than was sent the previous cycle

Example 5.5 An example of a basic decision maker.

```
ServerCommand* BasicDecision::initAgent() {
  return new InitCommand("RoboSoc", 7.1, false);
}

void BasicDecision::decide() {
  // Server command to send to the actuators.
  ServerCommand* scmd = 0;

  AngleDegUE ball_dir = ballView->getRelativeDirection();
  AngleDegUE turn_dir = ball_dir - agentView->getBodyDirection();
  if (turn_dir.isUnknown())
    // If the direction to the ball or the agent's bodydirection is
    // unknown, we turn 30 deg and hope we see the ball the next cycle.
    scmd = new TurnCommand(30);
  else if ( abs(turn_dir) > 10 )
    // If the angle to the ball is more than 10 deg, turn towards the ball.
    scmd = new TurnCommand(turn_dir.getAngle());
  else if ( gameView->getPlayMode() != PM_BEFORE_KICK_OFF ) {
    // If the game has begun, run towards the ball or kick it.
    if ( ballView->getRelativeDistance() > facts->SP_kickable_margin)
      scmd = new DashCommand(50);
    else {
      // The ball is kickable, can we kick it to the ball.

      // Get the direction to the opponents goal.
      AngleDegUE goal_dir = agentView->getPosition().getDirectionTo
        (facts->getTheirGoal().getPosition());
      // Kick the ball towards the goal.
      scmd = new KickCommand(50, (goal_dir -
                                  agentView->getBodyDirection()).getAngle());
    }
  } else
    // The game has not begun, do nothing.
    scmd = new TurnCommand(0);

  if ( scmd != 0 ) {
    // Send the command to the actuators.
    actuatorInterface->addToPlan(scmd);
    delete scmd;
  }
  return true;
}

void BasicDecision::onInit() {
  actuatorInterface->addToPlan(new MoveCommand(Point(-10, -10)));
}

void BasicDecision::onCommandWarning() {
  decide();
}
```

5.3 How To Send Commands To The Server?

To send a command to the server you have to create a new pointer to the wanted server command object, for example `new DashCommand(75)` to create a new command `(dash 75)`, then send it to server by using the method `addToPlan(Step step)` or `addToPlan(Command *command)` available in the `ActuatorInterface`. The actuator interface contain a `Plan` for the agent and sends the commands from this plan to the `Actuator` when the time is right. The `Plan` contains a number of `Step`s and a step contains the agent's actions for one cycle. Usually the decision making procedures only produce a single step for the agent since we can expect receiving new sensor information before we have to make a new decision and send a command to the Soccer server. The RoboCup environment is nondeterministic, we don't know exactly how our actions affects the world nor how our teammates or our opponents will act and by that influences the world. Because of this property the new information often cause us to revise an old plan and the effort of creating a plan with several steps do not necessarily improve the agents decisions.

Bibliography

[1] Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Patrick Riley, Timo Steffens, Yi Wang, and Xiang Yin. *Soccerserver Manual Ver. 7*, 2001.

[2] The Robot World Cup Initiative web-page. http://www.robocup.org, September 2002.

[3] The RoboSoc web-site. http://www.ida.liu.se/~frehe/RoboCup/RoboSoc/, September 2002.

[4] The Soccer Server web-page. http://sserver.sourceforge.net, September 2002.