RoboSoc an Agent Framework for RoboCup

Fredrik Heintz
Linköpings universitet, IDA
2006-09-09

The Architecture

RoboSoc Overview
- Basic server communication
- Localizes agent
- Constructs and predicts world model
- Tries to keep synchronization with the soccer server for command cycles
- Framework for higher level skills, strategies and agents

Basic Server Communication
- Receive data sent by the server
- Use this data to decide on an action
- Send the action, with the correct parameters, back to the server

Timing
- Keeping track of the current simulation step
- Send commands in time for them to be executed in the current step
- Use the best available information when making the decision

World Modeling
- Keep track of the state of the simulation based on the sensory data received from the server
- Easy to use representation
- Use the noisy and incomplete sensor data to draw correct conclusions about the world
Support for Decision Making

- Based on the current world model, decide on what action to do
- Notifications when new information arrive from the server and when a new cycle is started
- Higher level abstractions when it comes to actions

---

**Game Objects**

- FuzzyBool
  - Represents a boolean with some kind of certainty. Its value is between 0 and 1, where 0 represents certainly false and 1 certainly true. 0.5 represents an unknown value.
  - Methods:
    - `Float getValue()`
    - `bool isTrue()`
    - `bool isFalse()`
RoboSoc an Agent Framework for RoboCup

Types With Unknown

```cpp
template <typename T> class WithUnknown {
    WithUnknown<T>& makeUnknown();
    bool isKnown() const;
    bool isUnknown() const;
    T getValue() throw(RSException);
    ...
};
```

Types With UE

```
UE = Unknown and Error

template<typename T> class WithUE :
    protected WithUnknown<
       WithError<T> > {
    T getValue() throw(RSException);
    T getMin() throw(RSException);
    ...
};
```

Types With Error

```
template <typename T> classWithError {
    T getValue() const;
    T getMin() const;
    T getMax() const;
    T getPlus() const;
    T getMinus() const;
   WithError<T>& improve(WithError<T>&);
   WithError<T>& disimprove(WithError<T>&);
    ...
};
```

AngleDeg and AngleRad

```
typedef AngleDegrees<Float> AngleDeg;
typedef AngleRadians<Float> AngleRad;

Float getDeg() const;
Float getRad() const;
```

Point

```
typedef CoordPoint<Float> Point;

Point(Float x, Float y);
Point(Point, Vector);
Float getX() const; Float getY() const;
Point addX(Float); Point addY(Float);
Vector getVectorTo(Point) const;
AngleDeg getDirectionTo(Point) const;
Float getDistanceTo(Point) const;
```

Vector

```
typedef CoordVector<Float> Vector;

Vector(Float dir, Float dist);
Vector(AngleDeg dir, Float dist);
Vector(Point, Point);
Float getLength() const;
AngleDeg getAngle() const;
Vector setLength(Float);
Vector setAngle(AngleDeg);
```
RoboSoc an Agent Framework for RoboCup

Time

```cpp
class Time {
    UInt getT() const;
    UInt getS() const;
    static bool isClockStopped();
    static Time currentTime;
    ...
}
```

World Facts

```cpp
const MarkerObject& getMarker(MarkerName) const;
const LineObject&
    getLine(SideLineName) const;
Side getOurSide(); Side getTheirSide();
UInt getUniformNumber() const;
UInt getOurScore() const;
UInt getTheirScore() const;
const MarkerObject& getOurGoal() const;
const MarkerObject& getTheirGoal() const;
```

WorldModelInterface

```cpp
Command* getLastCommand();
worldModel&
    getWorld(Int i = NOW);
worldObservation&
    getObservation(Int i = NOW);
const WorldFacts* getWorldFacts();
const Time& getCurrentTime();
```

World Observation

```cpp
std::vector<PlayMode> playModes;
ServerBallObject ball;
ServerAgentObject agent;
std::vector<ServerPlayerObject> players;
std::vector<ServerMarkerObject> markers;
std::vector<ServerLineObject> lines;
std::string ownMessage;
std::string playerMessage;
std::string coachMessage;
Time gameTime;
```

World Model

```cpp
std::vector<PlayerObject>&
    getOpponents();
std::vector<PlayerObject>&
    getTeammates();
std::vector<PlayerObject>& getUnknown();
BallObject& getBall();
AgentObject& getAgent();
PlayMode getLastPlayMode() const;
Time& getGameTime();
TimeUnknown& getLastSeeTime();
TimeUnknown& getNextSeeTime();
```

MobileObject

```cpp
class MobileObject {
    PointUE getPosition() const;
    TimeUnknown getObservationTime() const;
    VectorUE getSpeedVector() const;
    TimeUnknown getSpeedObservationTime() const;
    VectorUE getAcceleration() const;
};
```

KTH
PlayerObject – Body Directions

```cpp
AngleDegUE getBodyDirection() const;
AngleDegUE getRelativeBodyDirectionToPoint(const PointUE& point) const;
AngleDegUE getRelativeBodyDirectionTo(const AngleDegUE& dir) const;
VectorUE getRelativeVectorTo(const PointUE& point) const;
```

PlayerObject – Neck Directions

```cpp
AngleDegUE getNeckDirection() const;
AngleDegUE getRelativeNeckDirection() const;
AngleDegUE getMinNeckDirection() const;
AngleDegUE getMaxNeckDirection() const;
AngleDegUE getRelativeNeckDirectionToPoint(const PointUE& point) const;
AngleDegUE getRelativeNeckDirectionToDirection(const AngleDegUE& dir) const;
```

AgentObject

```cpp
bool isAgentOutsideField() const;
AngleDeg getAngleForTurn(AngleDeg angle) const;
Float getPowerForDash(Point point) const;
AngleDeg getViewWidthAngle() const;
bool inViewCone(const AngleDeg& reldir) const;
FuzzyBool inViewCone(const AngleDegUE& reldir) const;
FuzzyBool inViewCone(const PointUE& point) const;
```

Strategies

- SeeStrategy
- HearStrategy
- CommandStrategy
- SenseStrategy
- InitStrategy
- CoachMessageStrategy
- NewCycleStrategy
- NewCycleStrategy
- CoachInformationStrategy

SeeStrategy

- BasicSeeStrategy
  - Localize the agent, the ball (with some checks), and the players
  - No persistence and few checks of past
- NewSeeStrategy
  - Use player matching to update world model
- PersistentSeeStrategy(cycles)
  - Use history about the agent and the ball in case new information is not available

Prediction

```cpp
cast CommandStrategy* cmd_strategy
  = WorldModelInterface::instance()->getCommandStrategy();
const NewCycleStrategy* new_cycle_strategy
  = WorldModelInterface::instance()->getNewCycleStrategy();
// Get the current state of the world, i.e. world model
WorldModel predicted_world
  = WorldModelInterface::instance()->getWorld(NOW);
// Predict the state of the world after executing the command
cmd_strategy->updateAfterCommand(predicted_world, command);
// Predict the state of the world the next cycle
new_cycle_strategy->updateBeforeNewCycle(predicted_world);
```
**Views**

Specialized information processor
- `bool init()`
- `void myNewCycle()`
- `void myUpdateAfterCoachMessage()`
- `void myUpdateAfterInit()`
- `void myUpdateAfterSee()`
- `void myUpdateAfterHear()`
- `void myUpdateAfterSense()`
- `void myUpdateAfterCommand()`

**The View Manager**

Singleton
- `ViewManager* instance()`
View management:
- `bool addView(View*, bool)`
- `View* getView(ViewId)`
- `void releaseView(ViewId)`

Existing views:
- `AgentView`, `BallView`, `GameView`, `PlayerView`

ViewId constants are in `view_constants.h`

**Predicates**

- Fuzzy tests of the world state
  - `FuzzyBool operator()`
  - `FuzzyBool evaluate()`

**Skills**

- Specialized short-term planners
  - `Plan generatePlan()`
  - `AgentStep generateStep()`
  - `FuzzyBool persistent()`
  - `FuzzyBool applicable()`
  - `FuzzyBool succeed()`
  - `FuzzyBool failed()`

**Actuator Interface**

- `bool addToPlan(Command*)`
- `bool addToPlan(AgentStep)`
- `bool addToPlan(Step)`

**AgentStep**

- `setBodyCommand(TurnCommand*)`
- `TurnCommand, DashCommand, KickCommand, CatchCommand, TackleCommand, MoveCommand`
- `setNeckCommand(TurnNeckCommand*)`
- `setSayCommand(SayCommand*)`
- `setPointToCommand(PointToCommand*)`
- `setChangeViewCommand(ChangeViewCommand*)`
- `setChangeModeCommand(ChangeModeCommand*)`
- `merge(const AgentStep&)`
**Decision**

- ServerCommand* initAgent()
- void onInit()
- void onActuatorSensorData()
- void onAuralSensorData()
- void onPhysicalSensorData()
- void onVisualSensorData()
- void onBeforeSensorData()
- void onAfterSensorData()

**Decision cont.**

- void onSensorError()
- void on_beforeTick()
- void onAfterTick()
- void onNewCycle(const bool forced)
- bool onEstimatedNewCycle(UInt16, UInt16)
- void onCommandWarning()
- void onDelayedActions(…)
- void onMissingActions(…)

**Example Decision Maker**

Example Decision Maker

AgentStep step;
if ( score.applicable() ) {
    step = score.generateStep();
} else if ( moveToPositionWithBall.applicable(goal) ) {
    step = moveToPositionWithBall.generateStep();
} else if ( gotoBallAvoidingObstacles.applicable() ) {
    step = gotoBallAvoidingObstacles.generateStep();
} else if ( findBall.applicable() ) {
    step = findBall.generateStep();
} actuatorInterface->addToPlan(step);

**The Main Program**

```
int main(int argc, char** argv) {
    try {
        RS::WorldFacts::init(argc, argv);
        RS::MyDecision decision;
        RS::FrameworkAgent agent(&decision);
        agent.start();
    } catch ( … ) { return 1; }
    return 0;
}
```

**The Main Loop**

- Is actually in Controller::run()
  - while ( serverAlive ) {
  - if ( pause() == -1 && errno != EINTR ) {
    RS_ERROR( "Pause failed" );
  }
  - while ( IOFlag || AlarmFlag ) {
  - if ( IOFlag ) handleSigIO();
  - if ( AlarmFlag ) handleSigAlarm();
  - }
  }

**Debugging and Logging**

- In rs_error.h:
  - RS_ERROR( msg )
  - RS_WARNING( msg )
  - RS_FATAL( msg )
- In rs_debug.h:
  - RS_DEBUG( msg )
- In rs_log.h:
  - RS_LOG( LogAreaName, msg )
Log area names in LogAreaName.h
### Debugging and Logging cont.

```cpp
std::ofstream logFile("playerlog");
LogStream *logStream = LogStream::instance();
logStream->addLogArea(LA_ROBOSOC, LA_TOP, true, &logFile);
logStream->addLogArea(LA_WARNING, LA_ROBOSOC,
true, &logFile, false, &std::cerr);
logStream->addLogArea(LA_ERROR, LA_ROBOSOC,
true, &logFile, true, &std::cerr);
logStream->addLogArea(LA_SENSOR_DATA, LA_ROBOSOC,
true, &logFile, false, &std::cerr);
logStream->addLogArea(LA_SERVER_COMMANDS, LA_ROBOSOC,
true, &logFile, false, &std::cerr);
```

### Agent Localization

- Use the furthest line to determine absolute face direction
- Find the closest marker and then add the relative direction to it to its absolute position
- `agent_pos = marker_pos - marker_vec.rotate(face_dir)`

### Implemented Skills

- AlignNeckToBody
- TurnNeckToPoint, TurnNeckToBall
- TurnNeckToAbsoluteDirection
- TurnBodyToPoint, TurnBackToPoint
- TurnBodyToAbsoluteDirection
- SearchBall
- DashToPoint
- FreezeBall, KickBallCloseToBody
- AccelerateBallToVelocity
- CatchBall

### Neck Skills

- Can be merged with Body Commands!
- AlignNeckToBody
  - Simple skill, just align
- TurnNeckToPoint, TurnNeckToBall
  - Predicts state next cycle and turns neck accordingly
- TurnNeckToAbsoluteDirection
  - Predicts state, body direction and turns neck

### Merge Body & Neck Command

```cpp
AgentStep turn_dst;
AngleDegUE rel_neckdir = agent.getRelativeNeckDirection();
if (rel_neckdir.isKnown()) { turn_dir = -rel_neckdir.getDeg(); }
else { turn_dir = 0; }
if (ball.getPosition().isKnown() and agent.inViewCone(ball.getPosition())) {
  AngleDegUE ball_dir = agent.getRelativeNeckDirectionToPoint(ball.getPosition());
  if (ball_dir.isKnown()) { turn_dir = ball_dir.getDeg(); }
}
turn_dir = minMax(agent.getMinNeckDirection(),
turn_dir, agent.getMaxNeckDirection());
TurnNeckCommand *cmd = new TurnNeckCommand(turn_dir.getDeg());
AgentStep step; step.setNeckCommand(cmd);
```
RoboSoc an Agent Framework for RoboCup

**Turn Skills**
- TurnBodyToPoint, TurnBackToPoint
  Predicts next position, compensates for inertia and turns.
- TurnBodyToAbsoluteDirection
  Predicts next position, compensates for inertia, head direction and turns.

**SearchBall**
- Compensates for view width and quality and turns until it sees the ball.
- Only applicable if the ball is not known.

**DashToPoint**
- Tries to get as close as possible to a point with a single dash. (This means it does not turn.)
- Current version doesn’t take stamina into account (dashes with full power even if the agent has top speed) and is not good at dashing backwards.

**FreezeBall**
- Tries to stop the ball with a single kick.
  - Calculates the ball vector,
  - inverts it, and
  - calculates which power it needs to kick with to get that kick vector taking into account where the ball will be with relation to body direction and distance.

**KickBallCloseToBody**
- Tries to put the ball in a favorable position for a kick in a certain direction next cycle.
- Calculates kick power, predicts the ball and kicks.
- If it cannot get the ball where it wants it, it will try to stop the ball instead.

**AccelerateBallToVelocity**
- Tries to give the ball a certain speed vector.
- If it cannot give the ball the wanted speed vector, it will focus on direction and give it as much speed as it can by compensating with kick direction.
CatchBall
- Calculates where the ball will be next cycle and issues a catch command in that direction.

RoboSoc
- Basic Server Communication
- Constructs and predicts world model
- Localizes agent
- Tries to keep synchronization with RCSS for command cycles
- Framework for higher level skills, strategies and agents