A Plugin-Based Architecture For Simulation In The F2000 League

Alexander Kleiner[1], Thorsten Buchheim [2]

[1] Institut für Informatik
Universität Freiburg
79110 Freiburg, Germany
kleiner@informatik.uni-freiburg.de

[2] Institut für Informatik
Universität Stuttgart
70565 Stuttgart, Germany
buchheim@informatik.uni-stuttgart.de

Abstract. Simulation has become an essential part in the development process of autonomous robotic systems. In the domain of robotics, developers often are confronted with problems like noisy sensor data, hardware malfunctions and scarce or temporarily inoperable hardware resources. A solution to most of the problems can be given by tools which allow the simulation of the application scenario in varying degrees of abstraction and the suppression of unwanted features of the domain (like e.g. sensor noise). The RoboCup scenario of autonomous mobile robots playing soccer is one such domain where the above mentioned problems typically arise.
In this work we will present a flexible simulation platform for the RoboCup F2000 league developed as a joint open source project by the universities of Freiburg [14] and Stuttgart [8] which achieves a maximum degree of modularity by a plugin based architecture and allows teams to easily develop and share software modules for the simulation of different sensors, kinematics and even complete player behaviors.
Moreover we show how plugins can be utilized to implement benchmark tasks for multi robot learning and give an example that demonstrates how the generic plugin approach can be extended towards the implementation of hardware independent algorithms for robot localization.

1 Introduction

Simulation has become an essential part in the development process of autonomous robotic systems. In the domain of robotics, developers often are confronted with problems like noisy sensor data, hardware malfunctions and scarce or temporarily inoperable hardware resources. A solution to most of the problems can be given by tools which allow the simulation of the application scenario in varying degrees of abstraction and the suppression of unwanted features of the domain (like e.g. sensor noise).

The RoboCup scenario of autonomous mobile robots playing soccer is one such domain where the above mentioned problems typically arise. In contrast to the the simulation league [6] where there is a predefined type of player with fixed action and perception capabilities in the simulated environment, the F2000 league permits a great diversity of robots in shape, kinematics and sensorics. Typically, teams in the F2000 league use multi sensor approaches to gather relevant information about their environment for the task of playing soccer. Here, all kind of local sensors, like ultrasonic or

infrared sensors, laser range finders, cameras or omni-vision sensors, may be used in arbitrary combinations on the robot.

Existing simulation tools for the F2000 league usually are rather specialized for a certain robot architecture of one team, restricted to a certain software language or deeply interwoven within the team's software structure [1] . Even if a certain adaptability to different kinds of robot setups or robot control properties partially was considered within some software designs [2], usually a lot of work still has to be spent to adapt the software to the individual needs of one team without any general benefit for other teams.

In this paper we present a flexible simulation platform for the RoboCup F2000 league developed as a joint open source project by the universities of Freiburg [14] and Stuttgart [8]. The simulator is based on a client/server concept that allows the simulation of diverse robots and also the simulation of team play.

The simulator achieves a maximum degree of modularity by a plugin based architecture. Software plugins are well known from the field of software engineering, particularly in the Internet context.They are basically dynamic libraries which extend the functionality of an application while loaded at runtime. We introduce a generic plugin concept for the simulation of the dynamic model and the sensors of an autonomous robot. This concept makes it possible to easily develop and share modular software components which can be re-used in various robot setups of different teams.

Furthermore we introduce plugins for robot behaviors. These are software components which implement a full player behavior in the sense of reading and reacting on sensor data while running as integrated module of the simulation. We show how they can be utilized to implement benchmark tasks for multi robot learning.

Finally we give an example that demonstrates how the generic plugin approach can be extended towards the implementation of hardware independent algorithms for robot localization. The latest version of the simulator is freely available from our home page [12].

The remainder of this paper is structured as follows. Section 2 gives an overview of the server and section 3 of the client part of the simulator. The concept of plugins which is used by both parts, will be described in more detail in Section 4. In section 5 we sketch some applications like localization and learning and in section 6 we give an outlook to future developments.

2 Server architecture

The core of the server is a 2D physics simulation that continuously computes the state of the world within discrete time intervals δ_t. At time t, the simulation calculates the subsequent world state for time $t + \delta_t$ by taking into account velocities of objects and their modified internal states, like for instance committed motion commands to a robot.

As indicated in figure 1, the physics simulation is basically influenced by robot objects existing on the server. Robots can either be controlled by clients connecting via the *message board* over a TCP/IP socket or by a *player plugin* which is started within the graphical user interface (*GUI*). In case a player plugin is created by the GUI, a robot

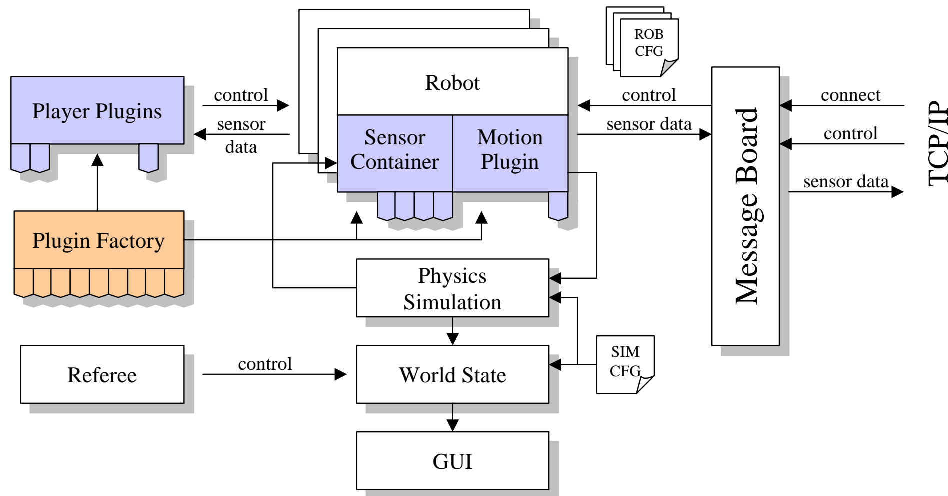


Fig. 1. Overview of the server architecture

is created and assigned to the plugin. If created by the message board, the robot will be controlled by the remote connected client.

To each robot there is a *motion plugin* and a *sensor container* associated. The motion plugin simulates the robot's dynamics, whereas the sensor container aggregates various plugins that implement the simulation of its sensors. The creation of the different plugins is done by a *plugin factory* which detects the available plugins at server startup. A more detailed description of the plugins will be given in section 4.

The individual setup of a robot is defined by a robot configuration file which contains information on its sensor and motion plugins. Furthermore the file includes a description of the robot's geometry and manipulators.

Manipulators are designed as variable parts of the robot geometry which can be switched between a set of states with differing geometric setups. Furthermore manipulator states can contain an additional parameter describing their ability to accelerate the ball which is needed for the definition of kicking devices.

Additionally, the state of the world can be influenced by the *referee module*. The intention of this module is to implement a set of rules restricting the robots actions in some way. At the current stage, however, the referee is limited to count the score, taking the ball back into the game if crossing the field boundary and resetting the game when a goal was marked. The physics simulation and the communication will be described in more detail in the following section.

2.1 Physics Simulation

The physics simulation is carried out in the plane. Simulation parameters, such as friction, restitution and cycle time are configurable by the simulation configuration file. The file also defines the setup of the environment which is described by a set of geometric primitives like rectangles, ellipses and triangles which nearly any scenario can be built from. As some domains may require three-dimensional information, each primitive contains two optional parameters specifying their beginning and end within the z-plane. Robots are described by these primitives as well, but within their specific robot configuration file. With a world described by primitives, a simulation of rigid body

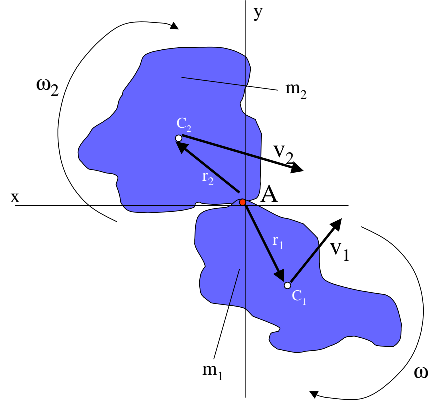


Fig. 2. An impact of two objects in the plane

collisions can be carried out. Unfortunately at the beginning of the simulator project there was no open source package for this simulation available. Therefore we implemented calculations based on ”Dynamics”, an old but enlightening book from Pestel and Thomson [11].

Figure 2 points out the problem of calculating rigid body collisions in the plane: Given two objects with centers of mass $C_1 = (x_1, y_1), C_2 = (x_2, y_2)$, the velocities $v_{1x}, v_{1y}, v_{2x}, v_{2y}$, rotations ω_1, ω_2 and masses m_1, m_2 that collided in A, compute the velocities $c_{1x}, c_{1y}, c_{2x}, c_{2y}$ and rotations Ω_1, Ω_2 after the collision. For a solution, we need six equations to determine the six unknowns. By the law of conservation of momentum along each space axis we get:

$$m_1 v_{1x} + m_2 v_{2x} = m_1 c_{1x} + m_2 c_{2x} \tag{1}$$

$$m_1 v_{1y} + m_2 v_{2y} = m_1 c_{1y} + m_2 c_{2y} \tag{2}$$

The angular momentum of each body with respect to the origin is conserved. Due to the restriction that the simulation is carried out in the plane the angular momentum is one-dimensional and because there are two bodies, this results in two equations:

$$\omega_{1z} I_1 + m_1 (x_1 v_{1y} - y_1 v_{1x}) = \Omega_{1z} I_1 + m_1 (x_1 c_{1y} - y_1 c_{1x}) \tag{3}$$

$$\omega_{2z} I_2 + m_2 (x_2 v_{2y} - y_2 v_{2x}) = \Omega_{2z} I_2 + m_2 (x_2 c_{2y} - y_2 c_{2x}) \quad (4)$$

where I_1 and I_2 denote the moment of inertia of the two bodies. After Newton's hypothesis the ratio of the relative velocity in the X direction (if we assume the impact is in X direction) before and after the collision equals a constant $-e$, where e is the elasticity (also known as restitution):

$$c_{1x} - c_{2x} + \Omega_{1z} y_1 - \Omega_{2z} y_2 = -e (v_{1x} - v_{2x} + \omega_{1z} y_1 - \omega_{2z} y_2) \quad (5)$$

In order to get the sixth formula, we have to make assumptions about the friction between the two bodies and thus about forces in the tangent plane. In the simulator so far, we use a simple solution that interpolates between the two extremes "stickiness" and "no friction" , which has been introduced by John Henckel[5]. The shown calculation can be generalized for the three dimensional case with little effort.

One difficulty in simulation is to determine the exact moment of collision. Usually when a collision is detected the two objects involved overlap by some degree due to a limited time scale resolution of the simulation. Hence the simulation calculates back n time steps within the last cycle if a collision occurred. So far, the accuracy of the collision detection suffices for a simulation in real-time and up to five times faster when executed on a Pentium 3 600MHz computer. We look forward to improve the collision detection to enable even faster simulations.

Another crucial component of robot simulation is a model of the robot's dynamics. Due to the rich set of existing models we decided to leave it open for the users to develop plugins for specific robots. Nevertheless, the simulator currently includes basic models for robots equipped with a differential or omnidirectional drive which will be further described in section 4.

2.2 Communication

Robots can be controlled by clients connecting via a TCP/IP socket to the message board. This has the advantage that clients can be programmed in any kind of language, as long as they comply with the protocol. Clients programmed in C/C++ can be linked directly with a provided client library and communicate with the server by function calls.

In the sake of simplicity, the protocol is based on ASCII strings that start with a unique command identifier followed by a list of parameters. Besides commands manipulating the state of the robot there are also commands to control the simulation, for example to perform a reset of the simulation or to receive the current score of the game.

Due to the generic plugin architecture, the protocol between server and client does not prescribe a format for the exchange of sensor data. Thus, plugins have to implement two abstract methods, one for reading and one for writing an array of *unsigned integers* respectively. This array, which represents the sensor's current data, is then transmitted together with a unique sensor identifier and a timestamp from the server to the client. On the client side, the received array and timestamp are written into the appropriate sensor plugin.

Since computer clocks are usually running asynchronously, we had to integrate a time synchronization mechanism between server and client. This is done by exchanging

timestamps between server and client during the connection phase which are used to calculate a time offset value. When transmitting sensor data to a client this offset is added to the timestamp of the package.

In contrast to other simulation servers which broadcast the state of the world within fixed time cycles [10] clients have to trigger the message board for receiving an update of the world. This permits clients with individual cycle times to connect to the simulation. The client framework currently handles the sensor updates in a separate thread with a default cycle time of $100ms$.

The time duration until the world state is completely transmitted depends strongly on the amount of data the client's sensor plugins requests. We measured on our internal network for a client requesting 180 beams of a Laser Range Finder (LRF) sensor, distance and angle to all lines, poles, and goals on the field, and odometry information, a duration between $2ms$ and $3ms$.

The latency of the server, which depends on the number of simultaneously connected clients, has been measured between $6ms$ and $8ms$ in the case of 3 actively operating clients.

3 Client architecture

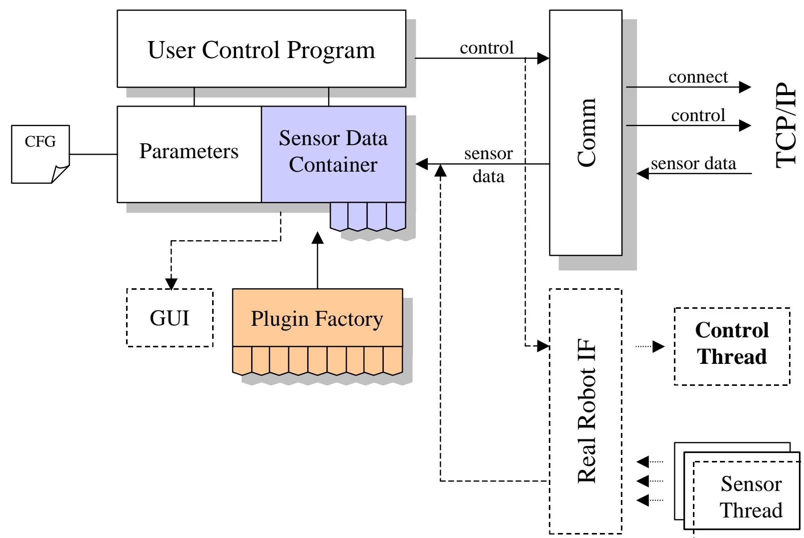


Fig. 3. Overview of the client architecture. Dashed components are optional

Figure 3 shows an overview of the client architecture. The client consists of a module that manages the communication with the server (*Comm*), a module for parsing and accessing information from the robot configuration file (*Parameters*) and a *sensor container* holding sensor plugins which are created by the *plugin factory*.

The *Parameters* module parses the client's robot configuration file which can then be used for accessing information on the position of sensors or maximal velocity settings. On the client side information on the sensor configuration is used for creating the respective sensor plugins, for example the type of feature the created sensor plugin provides. The file is also transmitted to the server during the server connection phase and used there for the creation of an equivalent robot object in the simulation.

Sensor plugins created on the client side are equal to plugins created on the server, but reduced of the ability to generate data. They can be considered as data buffers which are filled by the communication module.

As indicated by the dashed modules in the figure, one could also fill the container using threads that process information from real sensors. This is particularly useful when designing high-level algorithms based on the sensor container concept which work on the generated features. In this case the algorithms can easily be evaluated both in the simulation and on real robots.

The sensor plugins are automatically generated and inserted into the sensor container which provides functionality to search for and access specific sensor data objects by certain characteristics. Within the container, sensor plugins are distinguished by three identifiers:

1. The specific model of the sensor, for example DFW500[1] or LMS200[2]
2. The type of the sensor, for example camera or LRF
3. The name of the particular feature, for example *ballDistance* or *180DegreeScan*

In order to address a particular plugin, one can query the container by providing one or more of the above identifiers. This has the advantage that features can be addressed in a more abstract way by their functionality rather than by a specific sensor name. Thus plugins that provide equal features, e.g. *distanceToGoal* from a vision or LRF plugin, can be exchanged in the container without direct consequences for the user program.

Furthermore the client package provides an optional GUI module that can be used to display information from the sensors or additional information generated by algorithms on a higher level.

4 Plugin Architecture

The plugin concept was chosen since it offers the highest degree of flexibility and extensibility while maintaining an independent and stable simulator core. The plugin concept evolved in stages with the increasing need for a stronger modularity of certain parts of the simulator until reaching the current state which, we hope, meets most of the expectations and requirements for the teams of the F2000 league. Figure 4 shows the plugin architecture for the sensorics and motion simulation which will be described in the following.

[1] Firewire camera used by the University of Freiburg

[2] SICK laser range fi nder used by the RoboCup teams of Freiburg and Stuttgart

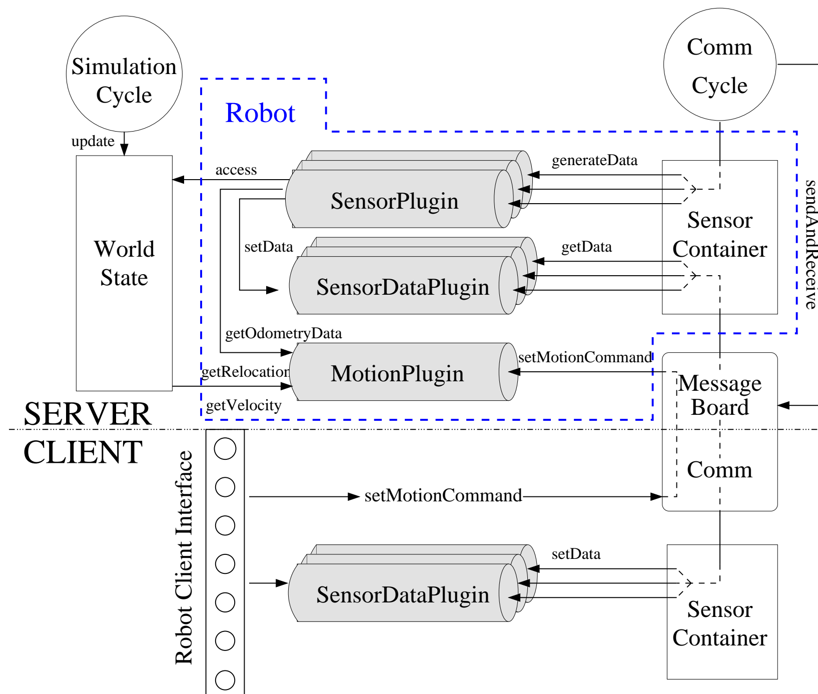


Fig. 4. Server/client view of the sensor and motion plugin concept

Sensor Plugins Usually the biggest discrepancies between robots of different teams apart from the robot chassis are the sensors used on the robots and the data representation used by the data processing modules, like differing coordinate frames or a camera image data format needed by a specific color segmentation algorithm. Consequently the sensorics play a central role within the plugin architecture of the simulator.

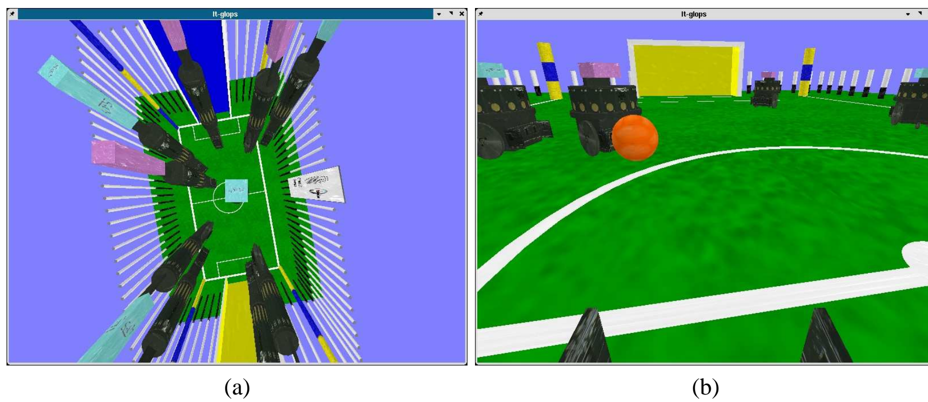
Although sensor plugins were referred to as one plugin type in the previous chapters for the sake of simplicity, the sensor concept consists of two separate plugin types, one for data generation and one for data storage. *Sensor plugins* access the current world state of the simulator to generate data of any desired form. This data is stored within the second plugin type named *sensor data plugin* which is needed for a transparent communication between server and client as well as for a data buffering on the client side. One functionality of the *sensor data plugins* is the transformation of their specific data representation to the data format required by the communication module and vice versa. Therefore a *sensor data plugin* has to implement the abstract methods *getData* and *setData* that return or take an array of *unsigned integers*, representing the current data of the sensor.

On an update request from the client, *sensor data plugin* objects on the server side are filled by the *sensor plugins*, transformed to the data transmission format, transferred to the client and re-transformed to the original data format by the corresponding *sensor*

data plugin objects on the client side. This enables the replication of the sensor data from the server on the client side and thereby yields a server/client transparency.

One major advantage of this concept is that sensor data processing can either be done on the client or on the server side as both share the same view on the sensor data. This can be extremely useful when the amount of data grows too large to transmit it to a client in a reasonable amount of time like e.g. a rendered high resolution camera image.

To build a new *sensor plugin* basically one method *generateData()* has to be implemented by the developer. Within this method the position of the sensor itself as well as information about all other objects concerning position, color and geometric shape are accessible by the framework within an absolute coordinate frame. This information can then be used to create the data as usually provided by the real sensor. Optionally there are auxiliary functions for simulating sensor noise or for two dimensional ray tracing.



(a) (b)

Fig. 5. Two screenshots from the plugin for generating artificial camera images: (a) an omnidirectional view as seen by a perfect warp free omnidirectional camera using an isometric mirror shape as proposed by [9]. (b) a conventional camera perspective.

Until now there exist several sensor plugins for different kinds of sensors based on the sensor equipment used by the teams of Freiburg and Stuttgart:

– **odometry sensor** providing the current translational and rotational velocities and a position within the odometry coordinate frame. Optionally it may supply additional data from the motion plugins which will be described in the following section.
– **camera for iconic data** of ball, goal and corner posts considering a limited field of view and a maximal view distance.
– **iconic omnivision camera** detecting field lines and goals, corner posts and the ball in a 360° field of view
– **laser range finder** providing either raw distance data obtained by ray tracing or iconic information about obstacles and an absolute position within the field
– **ultrasonic sensor** detecting the nearest obstacle within the opening angle of the sensor
– **camera image by 3D scene reconstruction** delivers a 3D rendered view of the scene as seen either by a commonly used 2D camera or a perfect omnivision sensor using a warp free isometric mirror shape as shown in Figure 5.

Motion Plugins The integration of different kinematics and controller types is facilitated by a further plugin type of the architecture named *motion plugin*. Within the simulator a robot's motion is represented by a change of its pose (x, y, θ) within a certain interval of time. This change of pose (relocation) of the robot is requested from the motion plugin within each update cycle of the simulation to update the robot's pose in the environment.

For the physics simulation the motion plugins must provide further information about its current velocity vector and the rotational speed which is needed for the collision calculation.

To set motion commands motion plugins can either be addressed by a set of standard commands, such as *setRotationalVelocity* or *setTranslationalVelocity*, or by a n-dimensional vector. The latter allows to address more complex motion models, as for example implemented by the omni-directional motion plugin which is currently developed at the University of Dortmund.

Since a motion model may require to send data about its internal state back to the client the motion plugin interface provides one further method to retrieve motion data that is automatically invoked by the default odometry sensor plugin of the server which makes the data available on the client side as well.

Currently there are three different motion plugins available. One model is based on a neural network that basically predicts the robot's state (x, y, θ, v_r, v_t) at t_1 depending on the previous state and velocity settings (v_t^{set}, v_r^{set}) at t_0 and was trained for Freiburg's Pioneer hardware base. A small application is also available to train the model for other robot bases by a log file recorded on the real robot while performing representative motions. A similar method has also been used for state prediction by the Agilo RoboCuppers [3,2].

A second model is a mathematical calculation of a two wheel differential drive based on a a linear acceleration assumption and parameter tuned to realistically map the behavior of the Nomadic Scout hardware base of the Stuttgart CoPs team. A third implementation for a holonomic three wheel robot base was developed by the University of Dortmund.

Player Plugins After various experiments and applications of the simulation server, we found it a useful feature to facilitate the implementation of simple robot behaviors that run within the simulator without the overhead of a remote connected client. This turned out to be useful to simulate opponent players like e.g. goalkeepers or for gathering simulated robot sensor data to create an environment map which e.g. was used for a Monte Carlo localization approach. For this kind of application the framework offers a third type of so called *player plugins* which is shown in figure 6.

Each player plugin controls one robot which is created based on its configuration file that is specified within the plugin. Its core functionality is implemented within a method *senseAndAct* that retrieves sensor information from the sensor data plugins and determines an action which usually is a command to the motion plugin or a triggering of its robot manipulators. Alternatively a player plugin can access the world state of the simulator directly and manipulate it if needed. This may either be useful when im-

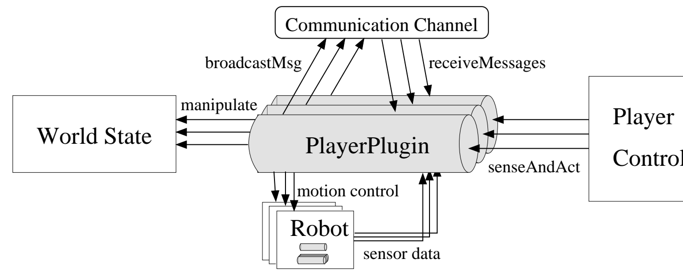


Fig. 6. Player plugin concept

plementing a perfect player behavior without limitations by its sensors or when certain environment situations shall be created artificially.

Each player plugin is identified at the start of the server and selectable by a menu entry within the graphical user interface of the server. Player plugins can be started and stopped by the GUI but can also terminate by themselves when their task is accomplished. The player control module supervises all active player plugins and its assigned robots. If either of them terminates or is deleted it assures the removal of its counterpart from the simulator.

To model coordinated or cooperative group behaviors of different player plugins the simulator provides a common communication channel for message broadcasts among player plugins. Each player can thus broadcast and receive messages from other player plugins to synchronize group actions.

Currently there are two player plugin types. A goalkeeper plugin which can be run with different modes of varying difficulty intended for AI benchmarks and a plugin to learn sensor data models for a Monte Carlo localization approach which will be described in the following chapter.

5 Applications

5.1 A benchmark for Multiagent Learning

The proposed architecture comes with two crucial advantages that makes it valuable for robot learning: Firstly, customized sensor and motion plugins allow for different kinds of robots to be used within the same learning environment. Secondly, the usage of player plugins on the server offers a modular way of designing and distributing benchmark tasks while keeping the same setup.

Within the simulator framework, algorithms can be designed for learning on either the client or server side. In both cases, the algorithm's input is taken from a sensor data container and the output is a command to a motion plugin. Due to the abstract interfaces it is possible that an algorithm can be applied to different robot types. As already discussed in section 3, sensors are addressable by their specific model, their type or the name of a feature they provide. By addressing the feature directly, e.g. *DistanceToGoal*, the algorithm can be executed, regardless by which sensor the data was produced. Note

that this works for plugins that provide features with equal identifiers and equal data format.

Another important issue is the evaluation of learning algorithms. Generally, this turns out to be hard in the context of autonomous robots, where frequently occurring changes in the environment make it nearly impossible to find the same testing conditions twice.

Hence we introduced the player plugins for implementing unique benchmark situations. One plugin, which is included in the simulator package, implements the control of a goalkeeper and can be started with different policies. The plugin can be used to evaluate the performance of an opponent (or a team), that learns policies to score against the defending goalkeeper, that was started at a specific level of skill. The goalkeeper plugin can be started with the policies *not moving*, *randomly moving*, *moving by a pattern*, *blocking the ball*, *intercepting the ball* and *mixed policy*. The benchmark has recently been introduced within the RoboCup Special Interest Group (SIG) on Multiagent Learning.

Our own experiences with learning a complex control task have shown that behaviors learned in the simulation can be executed successfully on a real robot as well [7]. Even tricky tasks, such as dribbling and approaching the ball, were managed well by the pre-trained robots that have been trained within games against hand-coded opponents or against themselves. Also the team from the University of Stuttgart uses the simulation for learning skills.

5.2 Plugin based localization

In this section we give an example that demonstrates how the plugin approach can be extended towards hardware independent implementations of robot localization algorithms. We extended the sensor plugin interface for the utilization of the well known Monte Carlo Localization method (MCL)[13].

The key idea behind Markov localization is to maintain a belief $Bel(l_t)$ of the current robot pose l_t. The belief is updated by successively integrating sensor readings s and actions a executed by the robot:

$$Bel(l_t) = p(l_t \mid s_t, a_{t-1}, s_{t-1}, a_{t-2}, \ldots, s_0) \tag{6}$$

Under the assumption of independence between the sensor readings and executed actions, the belief can be updated for sensor readings by

$$Bel(l_t) = \eta \, p(s_t \mid l_t) Bel(l_{t-1}) \tag{7}$$

where η denotes a normalization constant and $p(s_t \mid l_t)$ the perceptual model, and for executed actions by

$$Bel(l_t) = \int p(l_t \mid a_{t-1}, l_{t-1}) \, Bel(l_{t-1}) \, dl_{t-1} \tag{8}$$

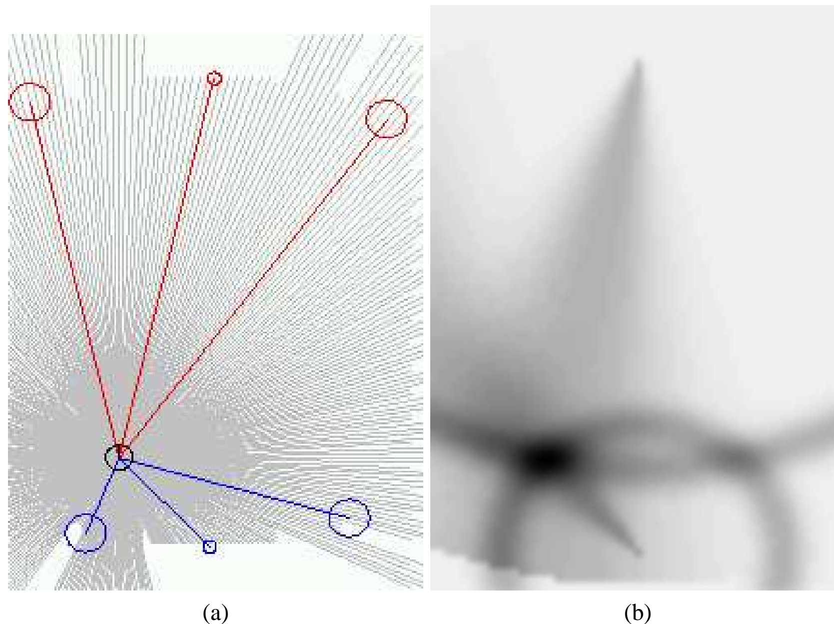
where $p(l_t \mid a_{t-1}, l_{t-1})$ denotes the robot's motion model.

MCL improves Markov localization by a more efficient representation of the belief $Bel(l_t)$. Instead of maintaining a belief for all poses, only a set of weighted samples is

updated by equation 7 and 8. Since the method has been well documented by several other researchers [13,4], we will not discuss it in detail any further here. Instead, we will focus on the perceptual model and its integration to the sensor container.

To implement MCL, one also needs to know the perceptional model $p(s \mid l)$. In the case of containers, s is a vector of n features, where each feature is stored by a plugin in the container. If we assume independence among the features with respect to the pose, the perceptual model can be calculated by $p(s_t \mid l) = p\left(s_t^1 \mid l\right) p\left(s_t^2 \mid l\right) ... p\left(s_t^n \mid l\right)$.

We extended the abstract interface of the sensor data plugins by the two methods *learnProbability(l_t)* and *getProbability(l_t)*. These methods are supposed for learning the sensors observation and accessing the perceptual model for the pose l_t respectively. It is defined that in case the sensor has an expectation for the queried pose, *getProbability(l_t)* returns a number between 0 and 1 and -1 otherwise. There exists also an interface *getProbability($l_t, \mathcal{S}$)* that allows to weight a sample set $\mathcal{S}$ according to the current sensor reading at pose l_t. Note that it is assumed that there was either a call of *generateData* or *setData* beforehand to assure that the latest observations are available inside the plugin. The interface leaves open how the perceptual model has to



(a) (b)

Fig. 7. Calculating perceptual models: (a) the robots expectation of the features *LRF Beam*, *PoleDistance*, *PoleAngle*, *GoalDistance* and *GoalAngle* (b) the mixed perceptual model of the features *distance* and *angle* to the blue and yellow goal. Darker regions indicates a higher probability of the robot's position. Sensor noise is modeled by a Gaussian distribution with $\sigma = 20mm$ and $\sigma = 5°$ for distances and angles respectively.

be represented. We used a common representation that separates the perceptual model in two tables, one for the expectation of each pose and one for the sensor model itself [13]. In this case it is assumed that *learnProbability* is called with noise free data, since the data is stored in the table of expectations. The second table is pre-calculated

from a Gaussian distribution that reflects the sensors noise. The return value of *get-Probability* is then simply calculated by a nested lookup of the two tables. Figure 7(a) shows the expectation of the features *LRF Beam*, *PoleDistance*, *PoleAngle*, *GoalDistance*, *GoalAngle*, and 7(b) the mixed perceptual model of the latter two for the blue and yellow goal.

The perceptual model can be learned at once for all features by a special player plugin in the simulator. The plugin basically moves the robot to all poses on the field and executes *generateData* and *learnProbability* for all Markov plugins found in the container.

During localization, the MCL implementation executes *getProbability* on the container for weighting the sample set representing the current belief. We are planning to develop a similar interface for the motion plugins. By this, the localization algorithm can be designed completely independent of the robot's hardware.

6 Summary and outlook

In this paper we presented a modular and extendible multi-robot simulation environment for the F2000 league that is suitable for simulating sensors, robot control tasks and cooperative team play. An open plugin architecture has been introduced that permits teams to design their own robot models regarding sensor configuration and motion control. With a growing community of teams using this simulator even full test matches can be held on a regular basis without physical presence of the teams. Furthermore we showed exemplarily, how the simulator framework can be utilized for Multiagent learning.

Past experiences in robotics have shown that besides a good idea for e.g. a new localization or feature extraction algorithm the concrete implementation plays an even more important role. Unfortunately, there are only a few implementations freely available and those are in turn tailored for particular hardware or data structures.

One of the key motivation of our work is to foster the exchange of software for autonomous robots. The proposed simulator architecture makes clear how this can be achieved in the case of a robot simulation. Even more beneficial, however, can be the exchange of software components of real robots. For accomplishing this goal, one has to introduce a level of abstraction and to define standardized methods for accessing this level.

We believe that the concept of sensor containers provides a good level of abstraction on real robot systems due to the following reasons:

- In the same way, as the proposed sensor plugins generate data inside the simulation, there can also be sensor plugins running on a robot that generate feature data from real sensors. RoboCup teams using the same kind of sensors could then easily share those components.
- As demonstrated by the MCL implementation in section 5.2, the abstract interface of the plugins can be extended for localization algorithms. The example shows generally how algorithms can be designed that operate on generic containers rather than on specific implementations for sensor data processing. Algorithms that are

based on generic containers can easily be shared among teams that use the same abstract interface.
- The concept allows to introduce ”virtual sensor” plugins that implement the communication of high-level features among different teams.

The German research project SPP1152-RoboCup is concerned with certain aspects regarding software architecture, system integration and learning. One aim is to find a common description for sensor types and properties as well as robot geometry and physics. We will adapt and contribute our approach to the project and look forward to get one step closer to the goal of a common description of robots and more efficient software development in the RoboCup domain.

References

1. A. Bredenfeld and G. Indiveri. Robot behavior engineering using DD-designer. In *Proceedings of the IEEE/RAS International Conference on Robotics and Automation (ICRA 2001)*. IEEE, 2001.
2. S. Buck, M. Beetz, and T. Schmitt. M-ROSE: A multi robot simulation environment for learning cooperative behavior. In H. Asama, T. Arai, T. Fukuda, and T. Hasegawa, editors, *Distributed Autonomous Robotic Systems*, volume 5, Berlin, Heidelberg, New York, 2002. Springer-Verlag.
3. S. Buck, R. Hanek, M. Klupsch, and T. Schmitt. Agilo robocuppers: Robocup team description. In *RoboCup 2000: Robot Soccer World Cup IV*, Berlin, Heidelberg, New York, 2000. Springer-Verlag.
4. S. Enderle, M. Ritter, D. Fox, S. Sablatng, G. K. Kraetzschmar, and G. Palm. Vision-Based Localization in Robocup Environments. In Peter Stone, Tucker Balch, and Gerhard K. Kraetzschmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, volume 2019 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, Berlin, 2001.
5. John Henckel. Simulation of rigid bodies. `http://www.geocities.com/Paris/6502/`.
6. H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada. The RoboCup synthetic agent challenge,97. In *International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997.
7. A. Kleiner, M. Dietl, and B. Nebel. Towards a life-long learning soccer agent. In *Proc. Int. RoboCup Symposium '02*. Fukuoka, Japan, 2002.
8. R. Lafrenz, M. Becht, T. Buchheim, P. Burger, G. Hetzel, G. Kindermann, M. Schanz, M. Schulé, and P. Levi. Cops-team description. In A. Birk, S. Coradeschi, and S. Tadokoro, editors, *RoboCup-01: Robot Soccer World Cup V*, pages S. 616 – 619. Springer Verlag, 2002.
9. Carlos Marques and Pedro Lima. A localization method for a soccer robot using a vision-based omni-directional sensor. In *Proceedings of EuRoboCup Workshop 2000*, 2000.
10. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: a tool for research on multi-agent systems. In *Applied Artificial Intelligence*, volume 12, pages 233–250, 1998.
11. E. C. Pestel and W. T. Thomson. *Dynamics*. McGraw-Hill, New York, 1968.
12. Simsrv. A robocup simulator for the F2000 league. `http://www.informatik.uni-freiburg.de/~simsrv`.
13. S. Thrun, D. Fox, W. Burgard, and Dellaert. F. Robust monte carlo localization for mobile robots. *Artificial Intelligence*, 128(1-2), 2001.
14. T. Weigel, J.-S. Gutmann, M. Dietl, A. Kleiner, and B. Nebel. CS-Freiburg: Coordinating robots for successful soccer playing. *IEEE Transactions on Robotics and Automation*, 18(5):685–699, 2002.