

Modeling Soccer-Robots Strategies Through Interaction Protocols

Pierre-Yves Oudeyer¹ and Jean-Luc Koning²

¹ École Normale Supérieure, 46 allée d'Italie, 69007 Lyon, France

² Leibniz-Esisar, 50 rue Laffemas - BP 54, 26902 Valence cedex 9, France
<http://www-leibniz.imag.fr/MAGMA/People/koning/koning.html>

Abstract. Straightforward approaches to team coordination with the expressive power of finite state automata are doomed to fail under a wide range of heterogeneity due to the combinatorial explosion of states. In this paper we propose a coordination scheme based on operational semantics, which allows an extremely compact and modular way of specifying soccer-robot team behaviors. The capabilities of our approach are demonstrated on two examples, which, though just being simple demo implementations, perform very well in a simulator tournament.

1 Soccer Strategies as Interaction Protocols

Building a strategy for a team of soccer-robots amounts to specifying the overall system's intention as far as its operational mode. A strategy encompasses general goals and features one may want to see arise during the course of a game, especially the main goal of winning the game. This naturally impacts on the robots cooperation/coordination.

In the multiagent system community communication and coordination issues are typically addressed by means of interaction protocols that give more or less flexible guidelines to the agents in their communication with each other [3]. Given its knowledge bases, what it perceives from its outside world and the possible messages it gets from teammates, an agent communicates with others both through message passing (direct communication) and modification of the world (indirect communication). More precisely, interaction protocols should enable the agents to know which types of message they may receive, which types they may send and at what time, and they should clarify to a certain extent the agents' side effects on the outside world.

When dealing with an open multiagent system such as a soccer-robot system, agents are allowed to quit or enter their team at any time during the course of the game. This flow can be forced or intentional. It is for example forced by the break-down of a robot or noisy radio-communication. It is intended when a certain type of player gets replaced with another one—to strengthen the defense after a decent lead in the score for example. Furthermore, we assume that the ontologies are predefined and known to all of the team agents. In other words, they all know the allowed message types and are fully capable of interpreting

them. They do not have to depend on a learning mechanism in order to process them.

For our system, protocols may be manifold thus enabling the agents to embody different roles and possibly dynamically change it during the course of the game. For instance, at some point a same robot may shift from a forward to a defender type of protocol. This assumption imposes the various protocols involved in the conversation between two agents to be compatible at a given time. That is, it is necessary for the messages being sent by an agent to be interpretable by the other one.

The design of protocols [2] classically relies on building a graph, where vertices represent the system's conversational states (i.e., the current state of the conversation the agents are participating in) and edges correspond to conversational events, (i.e., the sending or receiving of a message). Such protocol graphs describe the possible conversation courses. Yet several inherent weaknesses are attached to this approach.

This kind of representation only offers a syntactic description of the protocol. No semantic information can be taken into consideration. Moreover, the particular family of protocols that are thus addressed falls into the class of regular languages. This means multiagent systems with such protocols deny themselves any chance of a larger interaction expression power.

Making such protocols operational requires each agent to possess a general and quite flexible protocol execution algorithm, namely primitives for reading, processing, and sending messages. This piece of code should be totally independent from any specific protocol. Modifying a multiagent system strategy comes down to substituting the state transition graph description for a new one. A protocol is not actually hard-coded within the agents. Even though schemes have been devised to automate the translation of a graphical representation to an equivalent computable piece of code, such a solution is not fully satisfying. Among other things, they do not lead to a modular enough way of implementing strategies. For instance, adding or removing conversational states to a protocol forces a complete change of protocol.

Note, that when implementing a limited number of heterogeneous roles in a team of a few homogeneous robots, the standard approach is, though being tedious, still more or less feasible. But for larger teams, based on additional heterogeneity, this is obviously not the way to go. Therefore, we propose here the *Protocol Operational Semantics* (POS), inspired by the Structural Operational Semantics [12] in programming languages [4].

Section 2 briefly presents the POS model which features rules with algebraic data types and pattern matching. In section 3 we examine several basic behaviors of the JavaSoccer simulation environment as it is later used along with JavaBots to demonstrate POS' capabilities. In section 4, two team strategies are modeled using POS. We demonstrate how POS is a suited theoretical tool and leads as well to extremely compact and modular code due to its high expressive power. We also show that the two strategies defined perform very well in a JavaSoccer tournament against other predefined strategies.

2 Overview of the POS Model

Basically, making use of the Protocol Operational Semantics (POS) consists for each agent in specifying a set of rules (called a protocol) that will monitor the use of basic behaviors depending on evaluations of the world and messages exchanged with other agents. The abstract architecture of an agent using a protocol is thus the coupling of two subparts: the set of rules defining the protocol and the set of basic behaviors and evaluations the agent can perform.

POS' strength and specificity is that it uses algebraic data types and pattern-matching, allowing to describe powerful protocols (i.e., stronger than those based on finite state automata) in a very compact way. It is important to understand that this model is not only a theoretical framework, but also to a computational one due to the existence of adequate programming languages, like Pizza/Java [11] and other ML languages [10].

In a theoretical paper [8] we have extensively developed the POS model. The current section briefly summarizes its main characteristics.

The three kinds of rules available in POS are:

$$\begin{array}{l}
 \text{type 1 } \overbrace{\langle (sk, par), \phi(world) \rangle}^{\text{trigger}} \xrightarrow{[Send]} \overbrace{\langle (sk', par'), [\mathcal{A}(world)] \rangle}^{\text{consequence}} \\
 \text{type 2 } \langle (sk, par), \phi(world) \rangle \xrightarrow{msg} \langle (sk', par'), [\mathcal{A}(world)] \rangle \\
 \text{type 3 } \langle (sk, par), \phi(world) \rangle \xrightarrow{\varepsilon} \langle (sk', par'), [\mathcal{A}(world)] \rangle
 \end{array}$$

where:

- sk and sk' are skeletons in a string format for enhanced readability
- par and par' are parameters in form of objects with an algebraic data type on which pattern-matching can be performed; a pair of skeleton and parameter represents a parameterized state
- msg is a message pattern, which is also given in the form of an object with an algebraic data type on which pattern-matching can be performed
- $\phi(world)$ is a predicate on the world
- $\mathcal{A}(world)$ is a list of side-effects performed by the agent onto the world; here this corresponds to the activation of a basic behavior
- $send$ is a list of message sendings with one of the following possible types
 - *SendToId id msg*: sends the message msg to agent id
 - *SendToGroup list msg*: sends the message msg to a group of agents listed in $list$
 - *SendToAll msg*: sends the message msg to all agents

Type 1 rules correspond to the sending of messages, hence they are called sending rules. Type 2 rules correspond to the reading of a message from the agent's FIFO mailbox, so they are denoted as receiving rules. With type 3 rules no message is exchanged, they are denoted as ε rules.

A rule is fired when its parameterized state matches the current agent state and its predicate is true. Then, the corresponding list of messages is sent and the

side-effects are performed upon the world. Furthermore, the agent state changes to the parameterized state indicated by the rule. The rule engine we use here, is as follows:

```
repeat:
While (FIFO mailbox is not empty)
    read first message;
    try to interpret it (try to fire a receiving rule);
    throw it;
endWhile
try to fire a sending rule or an  $\epsilon$ -rule;
```

This is not the most general engine, but we have found it sufficient for our purpose. It is suited for protocols where either a read message can always be interpreted, or a loss of uninterpreted messages is not important. As mentioned, we have found it sufficient so far, yet it can easily be changed to fit protocols that do not have these properties.

As indicated above, POS' major feature lies in the notions of abstract data-types and pattern matching capabilities on agent states and message patterns. This has led to a high expressive power, and therefore offers the possibility to get a very compact and modular code.

3 The JavaBots Simulation Environment

The so-called JavaBots [1], freely available via [5], along with a soccer world (JavaSoccer) are an excellent tool-kit suited for several purposes within the RoboCup framework¹ [13]. First the nice graphical interface of the JavaBots allows to easily display what each part of the system "believes", in terms of where the robots are and what they are doing, and so on. Second, JavaBots are an excellent simulation tool with which strategies can be easily prototyped and tested. We use this property here to demonstrate the expressive power of POS.

JavaSoccer is different from quite many other simulators. Whereas common types of simulations are rather abstract and only remotely linked to the physical world, JavaBots tries to reproduce the robots' "body" and the soccer field with their physical laws. Nevertheless, JavaBots is relatively simple to use, which is very important in terms of ease to perform a high number of experiments. In concrete terms, the user is given access to the many robot's sensors and actuators.

Sensors: detect whether the robot is in a position to kick the ball, get a vector pointing to the ball, get a vector pointing to the team's goal, get a vector pointing to the opponent's goal, get an array of vectors pointing to all other players, get an array of vectors pointing to the robot's teammates, get an array of vectors pointing to the robot's opponents, get the player's number on the team (0-4, where 0 is the goalie), get the player's heading, get the time in milliseconds since the game started.

Actuators enable a robot to: kick the ball at 0.5 m/s, push the ball by driving on it, set the desired heading. The robot can turn at 360 degrees/sec, set desired speed up to 0.3 m/s.

¹ The Robot World Cup Soccer Initiative (RoboCup), is an annual event which is supposed to be a standard problem for Artificial Intelligence research [6] [7].

The JavaSoccer proceeds in time steps as follows. Each robot is given the new sensor values and computes a vector move, which determines the direction the robot should try to take, as well as the speed the robot should try to reach.

The following basic behaviors are the basis for the team behaviors or strategies of our demo-teams as presented later on:

Reaching a strategic position. The basic behavior *goTo(vp)* tries to go to the place indicated by vector *vp* denoting the vector between the robot and a desired goal position *p*.

Obstacle avoidance. The basic behavior *avoidCollision()* tends to keep a robot away from other robots, i.e., from the teammates and the opponents.

Getting in a strategic position behind the ball. The basic behavior *getBehind(vp,orientation)* tends to get the robot behind the position indicated by *vp* in respect to *orientation*. It is inspired by one of the functions of the *DreamTeam* provided with the JavaBots package. This behavior is especially useful when *vp* indicates the ball's position.

Attacking with the ball. This behavior *driveBall()* results in an attack on the opponents' goal. Like *getBehind()*, this function is also inspired by a function from the *DreamTeam*. It operates as follows: if the robot is behind the ball and oriented towards the goal, then the robot starts charging the goal and kicks the ball when it is close enough to the opponents' goal; otherwise it gets behind the ball and avoids colliding with other players.

Shooting and receiving passes. Passes mainly rely on two behaviors running on different robots, namely *passBall(orientation)* and *WaitPass()*. The behavior *passBall(orientation)* gets the robot behind the ball in relation to the *orientation* and then the robot kicks the ball.

The *WaitPass()* behavior can be used for the reception of passes. In doing so, several sub-tasks are accomplished: the robot keeps away from others (third term) while following the progression of the ball (second term) and it tries to go forward on the field (first term).

Spreading on the field. In real soccer, not every player runs after the ball. Instead, the team is spread over the field along some strategic position. The *Demarcate()* behavior can be useful to implement according strategies on soccer-robots. This behavior gets a robot to find a place on the field where it is more or less alone, while following the progression of the ball. This algorithm is similar to *WaitPass()*. But the additional feature is when the robot is in a position that could prevent one of its teammate to shoot forward, the robot gets out of the way instead of trying to get the ball.

Predicates. Together with the basic behaviors, the following predicates are used to form the building-blocks of different strategies that can easily be implemented with POS: *ballIsInRadius()*, *opponentInRadius()*, *ballIsDangerous()*, *ballIsInteresting()*, *AmIclosest()*, *findClosest()*. These predicates are self-explaining.

4 Modeling Strategies

4.1 A Straightforward POS Strategy

The first strategy we are presenting here is rather straightforward. Let us remind that in this article, our main intention is to demonstrate how POS can be used to describe

complex situations, and not to present a highly successful team strategy for RoboCup. Nevertheless, despite its simplicity this first team performs quite well as we shall see later on.

All robots in this team are heterogeneous with respect to the team's behaviors or roles they can take and dynamically change. The possible roles are: *Demarcate*, *Defend*, *Attack*, *Pass*, *WaitPass*, *Intercept* and *Possessor* (of the ball).

Different phases in the game are classified into the following three categories:

Phase 1: An agent tries to get the ball towards the opponent goal. But upon seeing there might be a difficulty, the agent tries to pass the ball to a teammate, while other agents try to demarcate.

Phase 2: If the ball gets really interesting (i.e., close to the opponent goal), a squad of two forwards launches an attack and repeatedly try to shoot the ball into the opponent's goal.

Phase 3: If the ball looks too threatening, a squad of two defenders is (dynamically) formed and comes to help the goal keeper.

At the beginning of a match, an arbitrary robot is set to the state *Possessor* and the others are set to the *Demarcating* state. In the following presentation of the protocols, $b := \text{beh}$ is used for denoting the taking up of behavior *beh* by the robot.

Let us note how easy it is to specify encapsulated set of rules in POS since each of them leads to a well-defined and meaningful behavior.

Phase 1:

- (1) $\langle \text{Demarcating}, \text{True} \rangle \longrightarrow \langle \text{WaitingPass}, b := \text{WaitPass} \rangle$
- (2) $\langle \text{WaitingPass}, \text{True} \rangle \xrightarrow[\text{Go}]{\text{PassToYou}} \langle \text{Intercepting}, b := \text{Intercept} \rangle$
- (3) $\langle \text{Intercepting}, \text{ballIsInRadius}() \rangle \xrightarrow{\epsilon} \langle \text{Possessor}, b := \text{DriveBall} \rangle$
- (4) $\langle \text{Possessor}, \text{opponentInRadius}() \rangle \xrightarrow[\text{sendTo}(\text{findDest}(), \text{PassToYou})]{\text{Passing}(\text{findDest}())} \langle \text{Passing}(\text{findDest}()), b := \text{Pass}(\text{findDest}()) \rangle$
- (5) $\langle \text{Passing}(\text{dest}), (\text{IHaveKicked} || \text{ballIsInRadius}()) \rangle \xrightarrow[\text{sendTo}(\text{dest}, \text{Go})]{\text{Demarcating}, b := \text{Demarcate}}$

This set of rules alone is already self-sufficient for defining a powerful group behavior. Robots tend to get the ball closer to the opponent's goal while keeping it away from these opponents by passing the ball among themselves. However, their behavior can be made more competitive by making them more offensive when arriving near the goal:

Phase 2:

- (6) $\langle \text{States}, \text{True} \rangle \xrightarrow[\text{Attack}]{\text{Attack}(s, b(\text{old}))} \langle \text{Attack}(s, b(\text{old})), b := \text{Attack} \rangle$
- (7) $\langle \text{Passing}(\text{dest}), \text{ballIsInteresting}() \rangle \xrightarrow[\text{sendTo}(\text{findDest}(), \text{Attack})]{\text{AttackPos}(\text{Passing}(\text{dest}), \text{Pass}(\text{dest}), \text{findDest}())} \langle \text{AttackPos}(\text{Passing}(\text{dest}), \text{Pass}(\text{dest}), \text{findDest}()), b := \text{Attack} \rangle$
- (8) $\langle \text{Possessor}, \text{ballIsInteresting}() \rangle \xrightarrow[\text{sendTo}(\text{findDest}(), \text{Attack})]{\text{AttackPos}(\text{Possessor}, \text{DriveBall})} \langle \text{AttackPos}(\text{Possessor}, \text{DriveBall}), b := \text{Attack} \rangle$
- (9) $\langle \text{Attack}(\text{States}, \text{behaviorbe}), \text{True} \rangle \xrightarrow[\text{AttackFinished}]{\text{AttackFinished}} \langle s, b := \text{be} \rangle$
- (10) $\langle \text{AttackPos}(\text{States}, \text{behaviorbe}, \text{dest}), !\text{ballIsInteresting}() \rangle \xrightarrow[\text{sendTo}(\text{dest}, \text{AttackFinished})]{\text{AttackFinished}} \langle s, b := \text{be} \rangle$

As far as this new set of rules let us point out two remarks. We had decided to get all robots to attack (as it is the case in the strategy presented in the next section) instead of having the current possessor along with another robot as only attackers, only two rules would have been sufficient.

Second, these rules are only enter/exit rules with a simple attack scheme: just the two forward most robots try to score, regardless of the other ones. An additional or new set of rules, i.e., a sub-protocol dedicated to cooperation of these two robots can easily be added without bothering about the rest of the entire protocol. Here again, we see POS' modularity potential.

Finally, this strategy is improved with a way to handle a dangerous ball. These rules are symmetrical to the attacking rules in phase 2.

Phase 3:

- $$\begin{aligned}
(11) \quad & \langle States, True \rangle \xrightarrow{Help} \langle Defend(s, b(old)), b := Defend \rangle \\
(12) \quad & \langle Defend(States, behaviorbe), True \rangle \xrightarrow{Finished} \langle s, b := be \rangle \\
(13) \quad & \langle Possessor, ballIsDangerous() \rangle \xrightarrow{sendTo(findDest(), Help)} \langle DefendPos(Possessor, DriveBall, findDest()), b := Defend \rangle \\
(14) \quad & \langle Passing(dest), ballIsDangerous() \rangle \xrightarrow{sendTo(findDest(), Defend)} \langle DefendPos(Passing(dest), Pass(dest), findDest()), b := Attack \rangle \\
(15) \quad & \langle DefendPos(States, behaviorbe, dest), !ballIsDangerous() \rangle \xrightarrow{sendTo(findDest(), Finished)} \langle s, b := be \rangle
\end{aligned}$$

Let us notice the relatively small size for the strategy's description, namely 15 rules. This is also very small in terms of actual code with a suited programming language. In some other recent work [9]² we have detailed a concrete transition from POS formula to ML which is a language that naturally allows for the implementation of algebraic data types and pattern matching. Here we use the Pizza language, a super-set of Java adding these two fundamental features needed for a POS implementation. The generated code is completely Java-compliant so that the type of work presented here can be repeated in any Java framework. An example showing the ease for implementing the rules as well as their interpretation is given below:

```

public void FireReceive(CoreAgent coreag,message mess,agentTeam t)
{
    // rule (1)
    switch(Pair(currentState,mess)) {
    case Pair(Demarcating,PassToYou):
        currentState = WaitingPass; // new parameterized state
        coreag.behave = WaitPass; // side effect
        break; };

    // rule (2)
    switch(Pair(currentState,mess)) {
    case Pair(WaitingPass,Go):

```

² In that paper, we implemented the Collect protocol which makes a fleet of little and possibly heterogeneous agents cooperate in order to achieve a joint task. This type of protocol applies to a world where several kinds of tasks exist and where no one agent is capable of handling all of them.

```

        currentState = Intercepting;
        coreag.behave = Intercept;
        break; };

// rule (11)
switch(Pair(currentState,mess)) {
case Pair(stateProt s,Help):
    currentState = Defend(s,coreag.behave);
    coreag.behave = Defending;
    break; };

// rule (12)
switch(Pair(currentState,mess)) {
case Pair(Defend(stateProt s,behavior b),Finished):
    currentState = s;
    coreag.behave = b;
    break; };
...

```

The related engine is also straightforward:

```

public void engine(CoreAgent coreagent,agentTeam t) {
    while ( !box.is.empty() ) {
        FireReceive(coreagent,box.take(),t); };
    FireSend(coreagent,t); }

```

The benefits of using the expressive power of algebraic data types and pattern matching compared to finite state automata with conventional if-then-rules is demonstrated best on an example. With conventional approaches, rule (11) would require 10 additional transitions, and accordingly the formulation of rules. In a general framework, replacing a single POS rule may amount to writing an infinite set of state-transitions (i.e., rules) since POS is Turing equivalent and not solely regular.

4.2 A Simpler Yet Stronger Strategy

This second strategy is simpler (in terms of the number of rules), but at the same time stronger (in terms of the performance). Coordination here is achieved entirely through direct perception of the world. Four roles are possible for the robots, namely *Demarcate*, *DriveBall*, *Defend*, and *Attack*. The basic idea is rather straightforward: the robot closest to the ball tries to get it closer to the opponent's goal while the others are trying to demarcate (phase 1); when the ball is dangerous, everyone comes in defense (phase 2); when it is interesting, everyone attacks (phase 3).

Phase 1:

- (1) $\langle \text{Demarcating}, \text{AmIclosest}() \rangle \xrightarrow{\varepsilon} \langle \text{DrivingBall}, b := \text{DriveBall} \rangle$
- (2) $\langle \text{DrivingBall}, !\text{AmIclosest}() \rangle \xrightarrow{\varepsilon} \langle \text{Demarcating}, b := \text{Demarcate} \rangle$

Phase 2:

(3) $\langle \text{Statep}, \text{ballIsDangerous}() \rangle \xrightarrow{\varepsilon} \langle \text{Defending}, b := \text{Defend} \rangle$

(4) $\langle \text{Defending}, !\text{ballIsDangerous}() \rangle \xrightarrow{\varepsilon} \langle \text{Demarcating}, b := \text{Demarcate} \rangle$

Phase 3:

(5) $\langle \text{Statep}, \text{ballIsInteresting}() \rangle \xrightarrow{\varepsilon} \langle \text{Attacking}, b := \text{Attack} \rangle$

(6) $\langle \text{Attacking}, !\text{ballIsInteresting}() \rangle \xrightarrow{\varepsilon} \langle \text{Demarcating}, b := \text{Demarcate} \rangle$

The first remark is that again phase 2 and phase 3 are just enter/exit attack or defense, and we could easily refine both of them by independently designing a dedicated attack protocol and include it here for instance. A second remark is the really small size of the description. Here finite state automata would have been able to describe the protocol, but would still be much more complicated (because of rules (3) and (5) that would map to many states and transitions). Despite its extreme simplicity it appears this strategy leads to a really efficient team work as seen in section 4.4.

4.3 Test Games

To demonstrate that team strategies modeled with POS are not only simple, in terms of compactness of code due to its expressive power, but also efficient, the two team strategies presented above are tested in a soccer tournament against the following other teams.

DTeam. The so-called *DTeam* (DT) provided in the JavaSoccer package is composed of heterogeneous robots who do not explicitly cooperate:

- an off-side player always sticking with the opponent goalie, trying to hinder it;
- one player constantly trying to get behind the ball and to push it in the direction of the opponent's goal;
- one robot playing backwards and defending when necessary;
- one player staying in the center and driving the ball to the opponent goal when it gets the opportunity;
- one robot is the goalie.

The specificity of this team is that four robots out of five are designated a more or less fixed location on the field. They always have the same role. This team is an example of a well-performing team without special coordination features.

SchemaDemo. The *SchemaDemo* (Sch) team is also provided within the JavaSoccer package. All robots are homogeneous (except for the goalie).

The individual behavior is: get behind the ball, then move to it and kick it, while keeping away from teammates. The specificity of this team is that robots are very mobile on the field and do not tend to keep the ball within a close range for dribbling, but they rather kick it ahead.

PyTeam. The *PyTeam* (PyT) is a team strategy we have designed. It is composed of heterogeneous robots with fixed roles:

- one robot is the goalie;

- one robot stays in defense, following the movements of the ball like the goalie, and defends when necessary with the *DriveBall()* behavior;
- a second robot stays also in defense, but it works differently: it tries to get behind the ball perpendicular to the heading of an attacking player with the ball; so, it tries to steal the ball;
- one robot always tries to get the ball closer to the opponent goal by passing it to more off-side mates or by using *DriveBallRear()*;
- one robot stays around the opponents' goal and launches an attack when the ball becomes interesting.

NopTeam. The *NopTeam* (Nop) is extremely simple, as all robots just do not move. It is of more significance than it might seem at a first glance, because this kind of situation is not too uncommon in the present state of art of RoboCup.

RandomTeam. The *RandomTeam* (R) is a variation where all robots just move randomly around.

4.4 Comparing the Performances

Before looking at the actual scores of different soccer games, let us focus on some general properties shown by teams T1 and T2 which demonstrate POS' capabilities with strategies presented in sections 4.1 and 4.2. First, both protocols never get into a deadlock, and they correctly perform the task they were designed for in every game against each other. This has been achieved right from the beginning without any trial-and-error debugging.

Second, the number and frequency of exchanged messages for the first protocol is low. On average there are 15 exchanged messages per minute for the whole team, while the player and the ball move in "real-time", i.e., similar to the speed of a real soccer game. This can be viewed as an indication of a low computational complexity, based on measures similar to the ones of communication protocols in distributed systems. In other words, this shows a high efficiency. In addition, the frequency with which the parameterized states change is also relatively low for both teams. On average this frequency is 15 times per minute for the first protocol, and 5 times per minute for the second one. This is an indication of a balanced relation between the dynamics of the strategies and the dynamics of the game.

Now, let us see the actual scores of the different teams playing against each other. The games of the *NopTeam* against the *RandomTeam* are omitted as they are of very limited interest. In our tournament the teams played against each other for a set of matches, each lasting 10 minutes. The average results are classified as follows: win (W), win by far (WF) where the scores are very cutting like 10 to 0, lose (L), lose by far (LF), equal (E) where the dominant team can not be accurately determined, deuce (D) where hardly any or no goal is scored at all, and no game (\emptyset). The results are presented in table 1, where an entry shows how team A (leading row) performs against team B (leading column), i.e., an entry of WF at (A,B) means that team A beats team B by far on average.

Table 1. The performance of the different teams in a tournament

A/B	DT	PyT	Sch	Nop	Rand	T1	T2
DT	\emptyset	E	E	W	WF	W	E
PyT	E	\emptyset	L	W	WF	E	D
Sch	E	W	\emptyset	WF	WF	E	L
Nop	L	L	LF	\emptyset	\emptyset	L	LF
Rand	LF	LF	LF	\emptyset	\emptyset	LF	LF
T1	L	E	E	W	WF	\emptyset	D
T2	E	D	W	WF	WF	D	\emptyset

The only team that never loses is *Team 2*. Three teams win 3 times out of 6: *Team 2*, *SchemaDemo*, and *DTeam*. But *Team 2* beats *SchemaDemo*, and *DTeam* has difficulties to win against *NopTeam* whereas *Team 2* has no such difficulties. Hence, *Team 2* is somehow the best one in this tournament.

Furthermore, we did an experiment to demonstrate the robustness of *Team 2*. In doing so, an arbitrary player is dis-activated, such that the team has always to get along with only three players against a complete adversary team. Table 2 shows the results.

Table 2. Robustness of team *T2* when an arbitrary player dis-activated

	DT	PyT	Sch	Nop	Rand	T1
T2	L	D	W	WF	WF	W

Still, *T2* performs very well. Only the *DreamTeam* can outrun this performance and now beat *T2*.

5 Conclusion

In a more theoretical and technical paper [8] we have presented the Protocol Operational Semantics (POS), an interaction protocol model based on abstract data-types and pattern matching capabilities. Endowed with these special features, POS has a high expressive power which generally makes it an interesting approach for coordinating agents in multiagent systems. Furthermore, POS allows for an easy verification of semantic properties of protocols.

The resulting compactness and modularity is especially of interest for robot soccer teams as we have argued in this article. In this particular application, heterogeneity on several levels, ranging from basic body features to abstract roles, can be very beneficial, as it allows to adapt to the almost unlimited number of potential opponent teams, game situations, and so on. But this so-to-say large

scale heterogeneity leads to a combinatorial explosion with classical protocols, where graphs with conversational states as nodes are constructed.

The high expressive power of POS is not only a theoretical property, but it is reflected in actual implementations with suited programming languages like Pizza, a super-set of Java. In this article, we have presented two simple teams for the soccer world of JavaBots to illustrate the potential of POS. Though these teams are intended for demonstration purposes only, they perform very well in a tournament against other strategies.

Contrary to real time reactive robots as we are dealing with here, we have also used POS to handle information agents in knowledge base environments [8]. This demonstrates POS' versatility in that it is applicable to completely different types of setting too.

References

1. T. R. Balch and A. Ram. Integrating robotics research with JavaBots. In *Working Notes of the AAAI 1998 Spring Symposium*, 1998.
2. Birgit Burmeister, Afsaneh Haddadi, and Kurt Sundermeyer. Generic, configurable, cooperation protocols for multi-agent systems. In C. Castelfranchi and Jean-Pierre Muller, editors, *From Reaction to Cognition*, volume 957 of *Lecture notes in AI*, pages 157–171, Berlin, Germany, 1995. Springer Verlag. Appeared also in MAAMAW-93, Neuchatel.
3. S. Cammarata, D. Mac Arthur, and R. Steeb. Strategies of cooperation in distributed problem solving. In A.H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 102–105. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.
4. M. Hennessy. *The Semantics of Programming Languages: An Introduction Using Structured Operational Semantics*. Wiley, 1990. Out of print.
5. Javabots information page. <http://www.cs.cmu.edu/trb/JavaBots>.
6. Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *First International Conference on Autonomous Agents (Agents-97)*. The ACM Press, 1997.
7. Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The robocup synthetic agent challenge 97. In *IJCAI-97*, 1997.
8. Jean-Luc Koning and Pierre-Yves Oudeyer. Introduction to POS: A protocol operational semantics. *International Journal on Cooperative Information Systems*, 2000. To be published.
9. Jean-Luc Koning and Pierre-Yves Oudeyer. Modeling and implementing conversation policies using POS. In B. d'Auriol, editor, *International Conference on Communications in Computing (CIC-00)*, Las Vegas, NV, June 2000. CSREA Press.
10. R. Milner. A proposal for standard ml. In *ACM Conference on Lisp and Functional Programming*, 1987.
11. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *4th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.
12. G. Plotkin. A structural approach to operationnal semantics. Technical Report DAIMI FN-19, Aarhus university, Computer Science Department, Denmark, 1981.
13. <http://www.robocup.org>.