Constraints, 7, 7–28, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Constraint-Based Robotic Soccer Team

YU ZHANG yzhang@cs.ubc.ca
ALAN K. MACKWORTH mack@cs.ubc.ca
Laboratory for Computational Intelligence, Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, Canada

Abstract. It is a challenging task for a team of multiple fast-moving robots to cooperate with each other and to compete with another team in a dynamic, real-time environment. For a robot team to play soccer successfully, various technologies have to be incorporated including robotic architecture, multi-agent collaboration and real-time reasoning. A robot is an integrated system, with a controller embedded in its plant. A robotic system is the coupling of a robot to its environment. Robotic systems are, in general, hybrid dynamic systems, consisting of continuous, discrete and event-driven components. Constraint Nets (CN) provide a semantic model for modeling hybrid dynamic systems. Controllers are embedded constraint solvers that solve constraints in real-time. A controller for our robot soccer team, UBC Dynamo98, has been modeled in CN, and implemented in Java, using the Java Beans architecture. A coach program using an evolutionary algorithm has also been designed and implemented to adjust the weights of the constraints and other parameters in the controller. The results demonstrate that the formal CN approach is a practical tool for designing and implementing controllers for robots in multi-agent real-time environments. They also demonstrate the effectiveness of applying the evolutionary algorithm to the CN-modeled controllers.

Keywords: constraint nets, robotic architecture, multi-agent collaboration, evolutionary algorithm, real-time control

1. Introduction

Compared to a traditional AI task, such as chess, robotic soccer, as a task domain, has the following characteristics:

1. Multi-agent cooperation, communication and competition among friendly and hostile agents.
2. Real-time dynamic environment.
3. Partially observable environment.
4. Nondeterministic environment.
5. Discrete/continuous hybrid environment.

With the recent achievement by the Deep Blue team, which beat Garry Kasparov, a human grand master, computer chess is close to its original goal. Soccer as a test domain is sufficiently rich to support research integrating many branches of robotics and AI [2, 8, 11, 13, 14] and is suitable as a new challenge for the next generation of AI systems.

To satisfy the need for a common environment, the Soccer Server was developed by Noda Itsuki [4–6] to make it possible to compare various algorithms for multi-agent systems. Because the physical abilities of the players are all identical in the server,

individual and team strategies are the focus of comparison. The Soccer Server is used by many researchers and has been chosen as the official simulator for the RoboCup Simulation League [7]. We developed a controller, Dynamo98, for our team of soccer players playing on the soccer server, using the constraint-based approach [24].

The main goal of this paper is *to present the constraint-based approach as a solution for robot control in real-time multi-agent dynamic environments*.

The rest of the paper is organized as follows. Section 2 describes the basic concepts of the constraint nets model (CN) and constraint-based control. Section 3 describes the constraint-net-based robot architecture for soccer-playing robots. Section 4 describes constraint-based control for soccer-playing robots. Section 5 describes how to design and implement a coach for soccer-playing robots using an evolutionary algorithm. Section 6 describes related work on multi-agent robot systems and soccer-playing robots. Section 7 concludes the paper.

2. The Constraint Nets Model and Constraint-Based Control

Ying Zhang and Alan Mackworth developed a semantic model for dynamic systems, Constraint Nets, in order to establish a foundation for modeling, specifying and verifying discrete/continuous hybrid systems. Constraint-based control was also developed by them as an integrated approach to the design and analysis of robotic systems and behaviors [20].

2.1. The Constraint Nets Model

A *robot* is an integrated system, with a controller embedded in its plant. A *robot controller* (or control system) is a subsystem of a robot, designed to regulate its behavior to meet certain requirements. A *robotic system* is the coupling of a robot to its environment. Robotic systems are, in general, hybrid dynamic systems, consisting of continuous, discrete and event-driven components. The dynamic relationship of a robot and its environment is called *the behavior of the robotic system*.

Constraint Nets (CN), a semantic model for hybrid dynamic systems, can be used to develop a robotic system, analyze its behavior and understand its underlying physics. Using this model, we can characterize the components of a system and derive the behavior of the overall system. CN is an abstraction and generalization of data-flow networks. Any (causal) system with discrete/continuous time, discrete/continuous (state) variables, and asynchronous/synchronous event structures can be modeled. Furthermore, a system can be modeled hierarchically using aggregation operators; the dynamics of the environment as well as the dynamics of the plant and the controller can be modeled individually and then integrated [21].

A constraint net consists of a finite set of locations, a finite set of transductions and a finite set of connections. Formally, a *constraint net* is a triple $CN = \langle Lc, Td, Cn \rangle$, where Lc is a finite set of *locations*, Td is a finite set of labels of *transductions*, each with

an *output port* and a set of *input ports*, *Cn* is a set of *connections* between locations. A location can be regarded as a wire, a channel, a variable, or a memory cell. Each transduction is a causal mapping from inputs to outputs over time, operating according to a certain reference time or activated by external events.

Semantically, a constraint net represents a set of equations, with locations as variables and transductions as functions. The *semantics* of the constraint net, with each location denoting a trace, is the least solution of the set of equations [20, p. 48]. Intuitively, a trace denotes changes of values over time. Formally, a mapping $v\colon \mathcal{T} \to A$ from time $\mathcal{T}$ to domain A is called a *trace* [20]. *Time* can be abstracted as a linearly ordered set $\langle \mathcal{T}, \leq \rangle$ with a least element $\mathbf{0}$ as the start time point, and a metric d on set $\mathcal{T}$.

Given *CN*, a constraint net model of a dynamic system, the abstract behavior of the system is the semantics of *CN*, denoted $[\![CN]\!]$, i.e., the set of input/output traces satisfying the model.

A complex system is generally composed of multiple components. A *module* is a constraint net with a set of locations as its interface. A constraint net can be composed hierarchically using modular and aggregation operators on modules. A compound module can be constructed from simple ones. There are three basic operations that can be applied to modules to obtain a new module. They are *union*, *coalescence* and *hiding* [22]. The semantics of a system can be obtained hierarchically from the semantics of its subsystems and their connections.

A constraint net is depicted by a bipartite directed graph where locations are depicted by circles, transductions by boxes and connections by arcs. A module is depicted by a box with rounded corners.

Furthermore, the environment of the robot can be modeled as a module as well. A robotic system can be modeled as an integration of a plant, a controller and an environment (Figure 1). A plant is a set of entities which must be controlled to achieve certain requirements, for example, a car with throttle and steering. A controller is a set of sensors and actuators, which, together with software/hardware computational systems, (partially) senses the states of the plant (X) and the environment (Y), and computes the desired control inputs (U) to actuate the plant. An environment is a set of entities beyond the (direct) control of the controller, with which the plant may interact. For example, obstacles to be avoided and objects to be reached can be considered as the *environment* of a robotic system.

In most cases, desired goals, safety requirements and physical restrictions of a robotic system can be specified by a set of constraints on variables $U \cup X \cup Y$. The controller is then synthesized to regulate the system to satisfy the set of constraints. The semantics (or behavior) of the system is the solution of the following equations:

$$X = \mathit{PLANT}(U, Y), \tag{1}$$

$$U = \mathit{CONTROLLER}(X, Y), \tag{2}$$

$$Y = \mathit{ENVIRONMENT}(X). \tag{3}$$

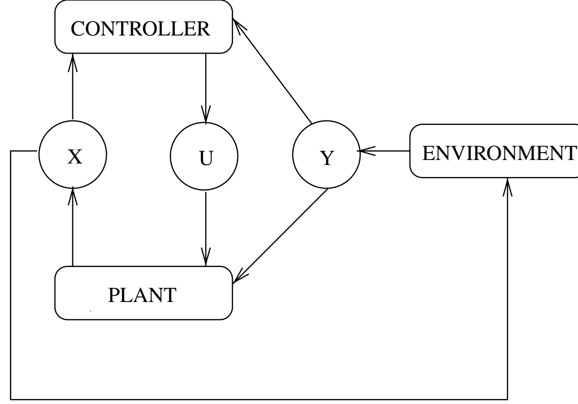

Figure 1. A robotic system.

Note that *PLANT*, *CONTROLLER* and *ENVIRONMENT* are transductions mapping input traces to output traces, and the solution gives X, Y and U as tuples of traces.

2.2. Constraint-Based Control

Control systems are designed to meet certain requirements. A *requirements specification* $\mathcal{R}$ for a system *CN* is a set of allowable input/output traces of the system: the behavior of *CN* satisfies its requirements specification $\mathcal{R}$, written $[\![CN]\!] \models \mathcal{R}$, iff $[\![CN]\!] \subseteq \mathcal{R}$.

The problem of control synthesis can be formalized as follows: given a requirements specification $\mathcal{R}$, the model of the plant *PLANT* and the model of the environment *ENVIRONMENT*, synthesize a model of the controller *CONTROLLER*, such that $[\![CN]\!] \models \mathcal{R}$ where *CN* is modeled by Eqs. (1), (2) and (3).

Ying Zhang and Mackworth consider constraints as relations on a set of state variables; the solution set of the constraints consists of the state variable tuples that satisfy all the constraints. They call the behavior of a dynamic system constraint-based if the system is asymptotically stable at the solution set of the given constraints, i.e., whenever the system diverges because of some disturbance, it will eventually return to the set satisfying the constraints. Various constraint methods fit into this framework for constraint satisfaction [23].

Most robotic systems are constraint-based, where the constraints may include physical limitations, environmental restrictions, and safety and goal requirements. Most learning and adaptive dynamic systems exhibit some forms of constraint-based behaviors as well.

A controller is an *embedded constraint solver* if the controller, together with the plant and the environment, satisfies the given constraint-based specification. Given a constraint-based requirements specification, the design of the controller becomes the synthesis of an embedded constraint solver which, together with the dynamics of the plant and the environment, solves the given constraints on-line.

3. Constraint Nets (CN) Based Robot Architecture for Soccer-Playing Robots

Ying Zhang and Mackworth propose two kinds of hierarchy in a robot control system: one is the composition hierarchy, the other is the interaction hierarchy [20, p. 166].

The Constraint Net model supports composition hierarchies with modules, which have a set of inputs and outputs and perform a transduction from input traces to output traces. The *composition hierarchy* characterizes the hierarchy of composing complex modules from simple ones. A complex module can be incrementally composed of simpler ones. A system can be tested and verified structurally.

The *interaction hierarchy* imposes the hierarchy of interaction or communication between modules. A control system is modeled as a module that may be further decomposed into a hierarchy of interactive modules (Figure 2). The higher levels are typically composed of event-driven transductions and the lower levels are typically analog control components. The bottom level sends control signals to various effectors, and at the same time, senses the state of effectors. Control signals flow down and the sensing signals

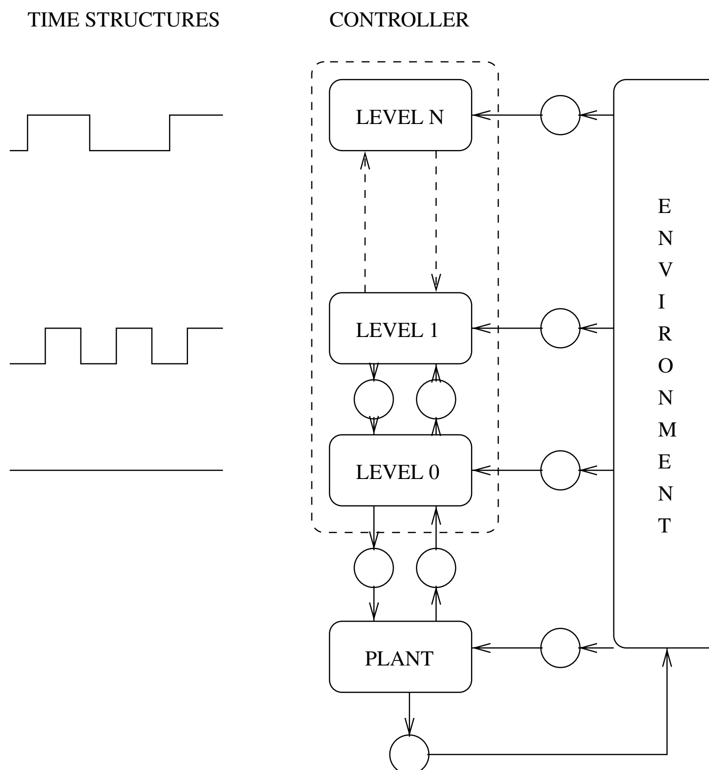

Figure 2. Abstraction hierarchy.

flow up. Sensing signals from the environment are distributed over levels. Each level is a black box that represents the causal relationship between the inputs and the outputs. The inputs consist of the control signals from the higher level, the sensing signals from the environment and the current states from the lower level. The outputs consist of the control signals to the lower level and the current states to the higher level. Usually, the bottom level is implemented by analog circuits that function with continuous dynamics and the higher levels are realized by distributed computing networks.

This CN architecture is object-oriented, parallel, and event-flow based. Events are used for communication among these CN modules. An event is an object which contains signals and (or) data. There are two types of CN modules, *event-driven* or *fixed-sample-time-driven*.

An event-driven CN module does nothing until it is woken by one or more events sent by other CN modules. Then it adjusts its states according to the signals in those event objects and processes the data in those event objects. After it finishes processing, it may produce its own events which contain the processed data, and send these events to wake other CN modules. Then the module goes to sleep again.

We define two kinds of input ports, *alarm* input ports and *non-alarm* input ports. Events that arrive on alarm input ports can wake up the CN module. Events that arrive on non-alarm input ports cannot wake up the CN module, they are only treated as transports for data. An event-driven CN module can only be woken when events are present on all of the alarm input ports.

A fixed-sample-time-driven CN module works and rests according to a module independent fixed time schedule. It has only non-alarm input ports. When it is time to work and there are events waiting to be processed, the module gets the events and processes the data in them. When it is time to sleep, no event can wake it up.

Execution of a CN module causes the events on the input ports to be removed and new events to be produced on the output ports. When two or more events come to an input port, the old event might get kicked out of the port by the new event. This also depends on the type of input port. As for buffering, there are three types of input port:

1. *Non-buffered* input port. It can only support one event. The newly arrived event will kick out the old event.
2. *Buffered* input port. It supports an event queue. The newly arrived event waits behind the old event to be processed.
3. *Hybrid* input port. The event itself decides its own fate. If an event belongs to an event sequence, it will carry the information telling the input port that it is a member of an event sequence and the number of events that follows. The port will buffer it if it's in an event sequence so it can't be discarded by the new one. When it indicates that itself is an independent event, the new event still can kick it out of the port.

In a highly reactive real-time dynamic environment like the soccer domain, non-buffered or hybrid input ports are highly recommended. The agent has no time to waste on the out-of-date situations, it must react quickly to the newly arrived situations. The hybrid input port is more suitable for real world problems since sometimes events are related to each other and it is better to let the event itself decide its importance.

The soccer-playing robot system is modeled as an integration of the soccer server and the controller (Figure 3). The soccer server provides 22 soccer-playing softbots' plants and the ball. Each softbot can be controlled by setting its throttle and steering. When the softbot is near the ball (within 2 meters), it can use the kick command to control the ball's movement. For the controller for one of the soccer-playing softbots, the rest of the players on the field and the ball are considered as its environment. The sensor of the controller determines the state of the plant (position and direction) by inference from a set of landmarks it 'sees.' The rest of the controller computes the desired control inputs (throttle and steering) and sends them to the soccer server to actuate the plant to move around on the field or kick or dribble the ball.

Considering the interaction hierarchy of the CN based architecture, we have designed a three-level controller for the soccer-playing robot shown in Figure 4. The lowest level is the *Effector&Sensor*. It receives ASCII sensor information from the soccer server then translates it into the World model. It also passes commands from the upper level down to the soccer server. The middle level is the *Executor*. It tries to translate the action (goal) which comes from the upper level into a sequence of commands and sends them to the lowest level. The Executor also evaluates the situation and sends it to the top level. The highest level is the *Planner*. It decides which action (goal) to take based on the current situation and it may also consider the next action assuming the current action will be correctly finished on schedule.

As for the composition hierarchy, the robot controller is composed of three CN modules. The Planner module forms the highest level. The Executor module forms the middle

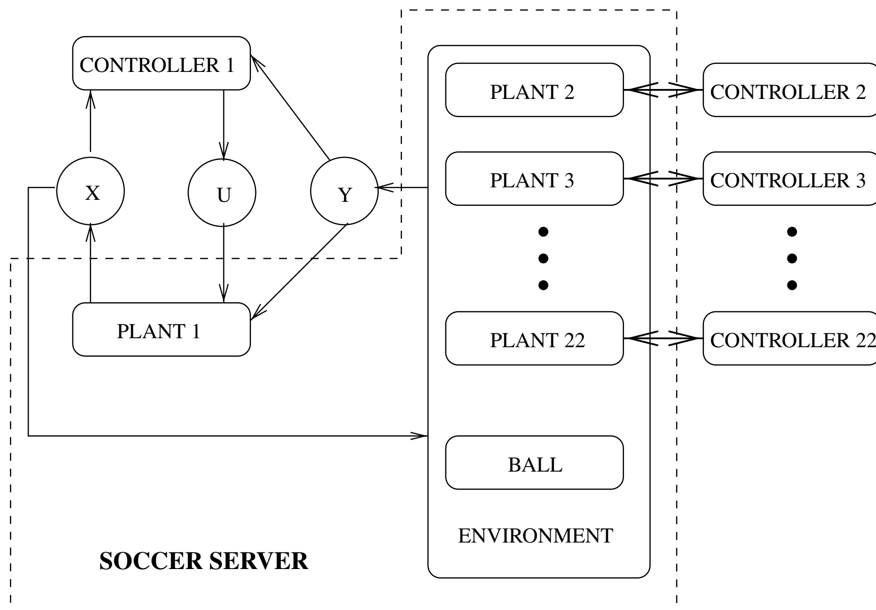

Figure 3. The soccer-playing softbot system.

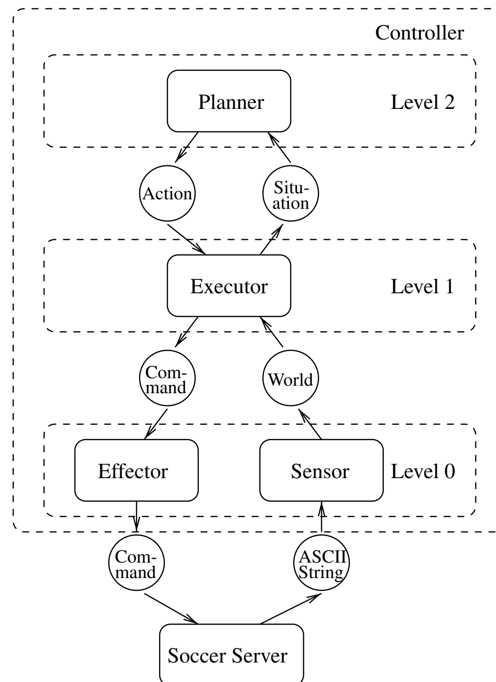

Figure 4. The soccer-playing controller hierarchy.

level. The Effector&Sensor module forms the low level and itself is composed of two sub modules, they are the *Effector* module and the *Sensor* module (Figure 4).

The Sensor is an event-driven module. It has one alarm non-buffered input port for receiving ASCII information from the soccer server. The Sensor module wakes up when new information arrives on its input port. It then processes the data, updates the world model, and sends an event to the Executor. The Sensor goes to sleep when there is no information waiting on its socket.

The Executor is an event-driven module. It has one non-alarm, non-buffered input port for receiving updated world model events from the Sensor, and one alarm non-buffered input port for receiving action events from the Planner.

The Executor module receives the event from the Sensor, then it processes the world model and updates the situation states. These situation states tell the Planner if it can kick the ball, if the ball is in its sight, if it is the nearest player to the ball, if there are obstacles on its way, if it is offside, if the action from the Planner has finished or not, and so on. Any change of situation creates an event and triggers the higher level Planner module.

The main part of the Executor executes actions passed down from the Planner. It wakes up when it receives an action event from the Planner module. It produces a sequence of commands which are supposed to achieve goals (actions) when they are performed.

Some of these commands are sent to the Effector's *Movement_command* input port. Other commands are sent to the Effector's *Sensing_command* input ports, they are *Say_message* input port, *Change_view* input port, and *Sense_body* input port. The Executor goes to sleep when there is no action waiting for its processing.

The Planner is an event-driven module. It has one alarm non-buffered input port for receiving updated situation events from the Executor. The Planner module wakes up when triggered by a situation-changed event from the Executor. It then produces an action event and pushes it into the Executor's action input port and then it goes to sleep until a new event comes.

The Effector is a fixed-sample-time-driven module. It has one non-alarm hybrid input port for receiving movement commands from the Executor. It is set to hybrid because some movements should be executed in sequence and be treated as a single, indivisible, atomic action. It has three other non-alarm, non-buffered input ports. They are Sensing_command input port, Say_message input port and Change_view input port. Every 100ms, it gets one command from each non-empty port and sends them to the soccer server.

4. Constraint-Based Control for Multi-agent Systems

In the framework for control synthesis, constraints are specified at different levels on different domains, with the higher levels more abstract and the lower levels more plant-dependent [21].

A control system can also be synthesized as a hierarchy of interactive embedded constraint solvers. Each abstraction level solves constraints on its state space and produces the input to the lower level. Typically the higher levels are composed of digital/symbolic event-driven control derived from discrete constraint methods and the lower levels embody analog control based on continuous constraint methods.

4.1. Combined Utility Constraint

In the framework of control synthesis, constraint solvers at the same level of the interaction hierarchy are coordinated via various arbitrations to compromise among different kinds of constraint, which form the *arbitration hierarchy*. The constraints with higher priority should be solved earlier than those with lower priorities.

Another way to deal with various constraints at the same level of the interaction hierarchy is to combine these constraints into one utility constraint. This combined utility constraint is to maximize the *utility function*:

$$U(a) = \sum_i k_i * P_i(a) \tag{4}$$

$U(a)$ is the action a's utility. $P_i(a)$ is the probability of satisfying the constraint i when taking the action a. k_i is the weight for the constraint i.

The constraint solver for this combined utility constraint will output the action a with the highest utility. These weights can be set by hand. They can also be tuned by a learning method, such as reinforcement learning. Also the utility function $U(a)$ need not be *linear*; it might be obtained by using neural network learning.

4.2. Special Constraints in a Multi-agent Environment

In the real world there are many tasks which can only be achieved successfully by a group of agents acting together. There is increasing interest in the issue of group co-operation amongst agents in order to more effectively achieve tasks. As robots become more adept at operating in the real world, the issues of collaborative and adversarial planning and learning are becoming more important. Here is an example in the robotic soccer domain to explain why we need collaborative systems. If there is no cooperation between robotic soccer players, teammates often attempt the same action and get in each others way. On the contrary, if we implement cooperation between robotic soccer players, keeping alert and constantly studying the position of opponents and teammates will turn a wild aimless kicking game into a smooth, beautiful passing attack.

Based on the Constraint Nets (CN) model, we simply treat other agents as a part of the environment which is modeled as Eqs. (3). This multi-agent part of the environment also provides some constraints on the agent as the rest of the environment does.

These multi-agent constraints can be classified into three types:

1. *Role Constraints*. Because of limited abilities of a single agent, each agent plays a separate role within the team so the whole team can achieve the goal or achieve it more efficiently.

2. *Social Law Constraints*. A simple way to provide coordination among agents is social laws or conventions. These social laws are the rules that define common actions which any agent has to do under certain conditions. If these social laws or conventions are obeyed by all the agents, no conflicts can happen and the common goal might be achieved. Social law constraints are very useful when different cooperative agents conflict with each other while trying to perform their roles.

3. *Communication Constraints*. A team of agents can also communicate with each other to coordinate their actions. This is especially effective when the environment is partially observable by the agents. An agent may not have a reasonable action when it can only get partial information. Another agent situated somewhere else might have a better view of the whole situation and think its teammate has to do certain actions under that condition. So it communicates with its teammate. The agent gets the information from its teammate, it can do its own actions or act as its teammate suggests. This decision depends on their ranks in the team and the confidence factor of the commands they communicate. For example, if the subordinate is sure that it knows the situation better, it may choose a different action from the one commanded by its leader.[1]

4.3. Constraint Methods in the Executor

The Executor module can be seen as an embedded constraint solver on its world state space. It solves the constraint-based requirements passed down from the higher layer Planner module. There are 7 major actions (constraint-based requirements):

1. *Find its own position*. If the player has lost its own position for a certain period, which is caused by running out of field and not having seen enough landmarks, it turns around until it finds enough landmarks to calculate its own position.
2. *Find the ball*. If the player has not seen the ball for a certain period, it turns around until it finds the ball.
3. *Avoid getting stuck*. If several players run into each other, they get stuck together. When this happens, the player dashes backwards. A sequence of commands act as this atomic action that will be passed down to the effector.
4. *Avoid being off-side*. When the player finds itself at an off-side position, it dashes towards its own side. A sequence of commands act as this atomic action which will be passed down to the effector.
5. *Kick the ball*. The planner passes down the kick action with its parameters telling which direction and position to kick the ball to, and how fast the ball should be at the destination. This is not a trivial problem. We have to consider the dynamic simulation inside the soccer server to set the right kick direction and power. Sometimes the required ball speed cannot be reached, then the Executor will provide the fastest it can. Sometimes even the required direction cannot be set, then the Executor will just kick the ball to the opponents' side.
6. *Intercept the ball*. This is also not as trivial a problem as it looks. The simplest method is to always let the player dash to the ball's present position. This may eventually let the player catch the ball. But a better method is for the player to predict the ball's future position and dash to there. The best method is for the player to predict ball-player collision position and dash to it (Figure 5). In this problem, we know the ball's position and velocity, we also know how fast the player can run. We want to know in which direction the player should dash to intercept the ball. Let (X_b, Y_b) be the ball position, (V_{bx}, V_{by}) be the ball velocity; (X_p, Y_p) be the player position, (V_{px}, V_{py}) be the player velocity. The constraint requirement is that after time t, the player and the ball meet at a certain position.

So we have the constraint equations:

$$\begin{cases} X_p + V_{px}t = X_b + V_{bx}t \\ Y_p + V_{py}t = Y_b + V_{by}t \end{cases} \tag{5}$$

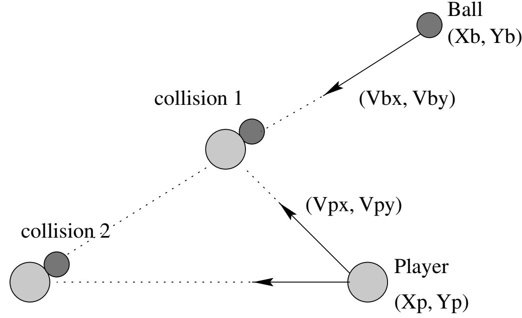

Figure 5. The player intercepts the ball, by setting different directions, the player can intercept the ball at two different positions when its speed is constant.

Let V_p be the player speed, V_b be the ball speed; Let $X_b - X_p = R_x$, $Y_b - Y_p = R_y$; we have

$$\left(V_b^2 - V_p^2\right)t^2 + 2\left(R_x V_{bx} + R_y V_{by}\right)t + \left(R_x^2 + R_y^2\right) = 0 \tag{6}$$

Now by solving this equation, we can know how long it will take to intercept the ball. Then we can find the collision position because we already know the ball's velocity. When the collision position is decided, the player can turn to it and dash to that position.

From Figure 5, we know there might be two collision positions. Of course, we choose the closer one. In the equation, it means t has two solutions, we pick the smaller one. Sometimes there is no solution for the equation, then we let the player dash to the position where the ball stops.

To determine the effectiveness of the ball interception method, we equipped one team with our constraint-based method, and another team with the method predicting the ball's position 500 ms into the future.

The results are shown in Table 1.

We can see the constraint-based team is much better. In six games, it won five games, and only lost one.

4.4. Constraint Methods in Planner

The Planner module can be seen as an embedded constraint solver on its situation state space. The ultimate constraint here is: the number of goals should be more than its

Table 1. Constraint-based team vs. normal team

Game	1	2	3	4	5	6
Constraint-based team	6	1	5	2	3	5
Normal team	1	3	1	0	1	1

opponent's. To satisfy this ultimate constraint, the robot has to satisfy a series of other constraints first.

The planner chooses actions to satisfy the constraints according to their priorities. Finding its own position has the highest priority. When the robot loses its own position for a certain amount of time, it sends *find_me* action down to the Executor. Finding the ball is the second priority, when the robot can't find the ball for a certain amount of time, it sends *find_ball* action down to the Executor. Avoiding collision is the third priority. When the robot senses that it will collide with other players, it sends an *avoid_collision* action down to the Executor. Avoiding offside is the fourth priority. It sends down an *avoid_offside* down to the Executor if it finds itself is at an off-side position.

According to the robots' distance to the ball, three different independent groups of constraints form the fifth arbitration level.

If the robot senses its own team players are nearer than itself to the ball, according to the social law constraint, it won't chase the ball, instead it goes to a certain position to cover a certain area on the field. Where it goes depends on its role constraint and the game situation. For example, if it is the goalie, it goes to the front of the goal.

The game has two situations, *offensive* and *defensive*. When the opponents have the ball, the situation is defensive. When any robot in a team is close enough to the ball to kick it, we say this team has the ball. When its own team has the ball, the situation is offensive. When both teams have the ball or neither has the ball, if the ball is in the opponents' side, it is offensive, otherwise it is defensive.

Each player role has been assigned two different areas to cover. Which one to cover depends on the game situation. If it's defensive, it covers the defensive area. When the situation changes to offensive, it goes to the offensive position.

The robot tries to *intercept* the ball if it senses that it is nearer to the ball than its teammates.

When the player is very near the ball, it sends out a message telling its teammates it will control the ball. And its teammates who get the message will give up intercepting the ball if they are doing that and go to certain predefined offensive or defensive positions to assist the intercepter.

If the robot is close enough to the ball to kick it, it has to choose where to kick. It can shoot, pass or dribble (pass to itself). Dribble is a dangerous action and is not recommended, because when the robot is dribbling the ball, the robot and the ball's speed are so slow that opponent robots can easily kick the ball away.

In order to win, the robot has to consider some constraints, such as, its own team's time in possession of the ball should be longer than its opponent's team; the ball should be near enough to the opponent's goal; the ball should be as far away as possible from its own goal; and the ball should be kicked into the opponent's goal instead of its own goal.

The best action should best satisfy the constraints listed above. Here we have two problems. First, the robot can't be absolutely sure that certain actions will satisfy certain constraints because the soccer server provides a noisy, dynamic world. For example, it's impossible for the robot to choose a kick direction which makes sure that its teammates will get the ball first. We can only say that if the robot chooses to kick in this direction,

the probability of its teammates getting the ball first is high or low. Second, the robot can't find a kick direction that can maximize all the probabilities of satisfying all the constraints. For example, if the robot chooses the kick direction which makes the probability of its teammates getting the ball very high, the ball might be kicked away from its opponent's goal and near its own goal. That probability depends on the number of opponents positioned near the receiver and their distances to the receiver.

We solve this by using the *combined utility constraint* (Eq. 4). The player can kick in one of 36 directions. These directions are uniformly spaced at 10 degree intervals. The player chooses the kick direction with the highest utility.

Some of the constraints considered in the utility function are: keeping the ball in the field, moving the ball to the opponent's side, letting its own team's time in possession of the ball be longer than its opponent's team, and shooting into the opponents' goal.

We have designed a coach program using an evolutionary algorithm to adjust these weights and other parameters in the controller.

Also, the player will send out a message telling its teammates if it gets the ball. Those teammates who get the message will observe themselves to find out if they are in a good position to get the ball. A good position is one in which the robot has few opponents nearby or is near the opponents' goal. If they think they are, they will tell the passer to pass the ball to them. If the passer gets one such message, it will give this communication constraint a higher priority and pass the ball to the message sender.

Communication is not only for cooperation, but also for sharing the world information among teammates. The map information, positions of objects seen by the robot, is also included in the messages used to cooperate among teammates. And every 10 steps, the player will send out a message sharing its map with its teammates.

4.5. Java Implementation of the Constraint Based Controller

The controller for our soccer robots is implemented in Java. The Java model is a new way of computing, based on networks and the idea that the same software should run on many different kinds of computers, consumer gadgets, and other devices. It is an object-oriented language which has a very good thread programming environment and provides a very good event mechanism. Java is a good candidate language for constraint-based controller implementation because of these features.

5. Designing the Coach Using an Evolutionary Algorithm

5.1. Evolutionary Algorithm

Nature has a robust way of evolving successful organisms. A variety of evolutionary algorithms (EAs) have been proposed based on this successful natural process [3]. Generally speaking, EAs maintain a population of individuals that evolve according to the rules of variation, selection and reproduction.

Each individual in the population receives a measure of its fitness in the environment. The highly fit individuals are selected and given rights to reproduce and the ill-suited individuals die off. Recombination and mutation are applied to those successful individuals to produce their offspring, thus providing general heuristics for exploring the available fitness information. Since the evolutionary process learns an agent function based on some special rewards like the rights to survive and to produce offspring, it can be seen as a form of reinforcement learning.

A general evolutionary algorithm EA [3] is shown in Figure 6.

In our EA approach, an individual is represented as a pair of integer vectors. A vector is called a *chromosome*. An integer in the vector is called a *gene* which partly decides one of the parameters in the controller. Each of the two chromosomes carries the same characteristic set of genes in a specific order. The sum of each pair of genes decides one parameter in the controller.

```
Algorithm EA
 //start
 t := 0;
//initialize a random population of individuals
 P = initpopulation(t);
//evaluate fitness of individuals in population
 evaluate(P);
//test for termination condition (time, fitness, etc.)
while not done do
        //increase the time counter
          t := t + 1;
       //select parents
          PARENTS := select(P, t);
       //reproduction
          OFFSPRING := reproduce(PARENTS);
       //evaluate the offspring's fitness
          evaluate(OFFSPRING);
       //select the survivors to form the new generation
          P := survive(P, OFFSPRING);
end while
end EA
```

Figure 6. A general evolutionary algorithm EA.

Three groups of genes are considered:

1. Genes representing the weights in the *utility function*. They decide what to do when the soccer player gets the ball: to shoot or to pass.
2. Genes representing roles' control areas. They decide where to go when the soccer player does not have the ball. Each role has two strategic positions, offensive and defensive. There are 11 roles in the soccer team. Every soccer player has all the functions of these 11 roles. But when the game begins, each player is assigned a role and this role cannot be changed during the game.
3. Genes representing the intercepting strategy. They decide the intercepting range threshold and the relation to other teammate interceptors. If the ball is out of intercepting range threshold, the player gives up chasing and goes to strategic positions. If the other teammate interceptor is nearer to the ball, the player also gives up chasing. How much nearer to the ball the other teammate should be is decided by these genes.

Two steps are needed to recombine the genes of the selected parents:

1. Each parent randomly chooses one gene from each pair of genes to form its sex chromosome.
2. An offspring is formed by receiving one sex chromosome from each of its parents and pairing them together.

Our EA uses tournaments to select fit individuals because of the competitive nature of the soccer domain. Each trial solution in the population faces competition against a preselected number of opponents and receives a "win" according to the rules of the game. Selection then eliminates those solutions with the least wins.

By this evolutionary algorithm, the constraint-based controller for the robot evolves to better satisfy the constraint requirements from the environment.

5.2. The Design of the Coach

The software architecture of the coach is similar to that of the controller for the soccer player. It has three layers. The lowest layer is the *Teacher&Observer*. It receives ASCII sensor information from the soccer server to get all the players' and ball's positions. It also passes commands from the upper layer down to the soccer server. The middle level is the *Tester*. It gets the genetic information from the upper layer and starts/finishes the game and decides which individual is better. The highest level is *Nature*. It maintains a population, selects a pair of parents and produces children.

As for the composition hierarchy, the coach is composed of three CN modules. The Nature module forms the highest level. The Tester module forms the middle level. The Teacher&Observer module forms the lowest level and itself is composed of two sub modules, they are the *Teacher* module and the *Observer* module (Figure 7).

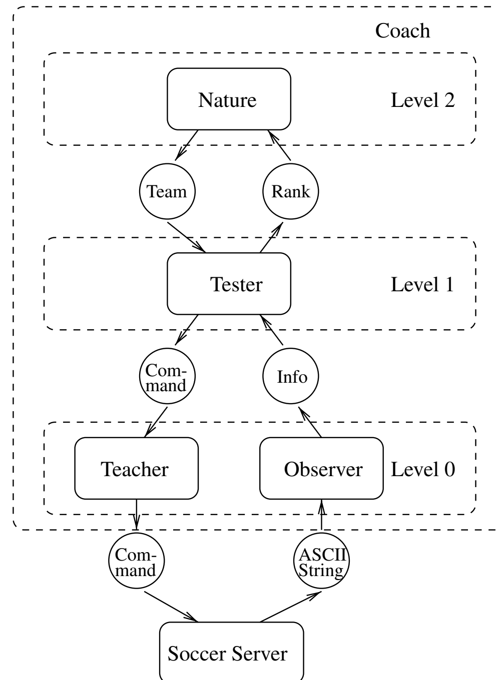

Figure 7. The architecture of the coach.

The Observer is a fixed-sample-time-driven module. It has one non-alarm, non-buffered input port for receiving ASCII information from the soccer server. Every 100 ms, it wakes up to process the new information waiting at its input port. It then processes the data, updates all the players' and the ball's positions, and sends an event containing this information to the Tester.

The Tester is an event-driven module. It has one non-alarm, non-buffered input port for receiving Team events which contain both the teams' genetic information from the Nature, and one alarm non-buffered input port for receiving Info events which contain position information from the Observer.

The Tester module receives the Team event from the Nature. Then it begins to send the command events down to the Teacher. When the game is on, the Tester monitors and records how long each team possesses the ball, how long the ball is in each side and the score of each team. It also monitors if the game is stuck or deadlocked. That is, players from both teams get stuck together trying to kick the ball. If this happens, the Tester will stop the game, move the ball to center, and restart the game.

When the game finishes, the Tester will evaluate which team is better and send this Rank event up to the Nature. One team is better than the other if:

1. It scores more.
2. The ball stays longer in the opponents' side if both teams have the same score.

3. It has the ball longer if both teams score the same number of goals and the ball spends an equal amount of time in both sides.

Nature is an event-driven module. It has one alarm non-buffered input port for receiving Rank events from the Tester. Nature divides the population into two equal-sized groups, selects two individuals in a group, and sends down the genetic information about these two individuals to the Tester. Then it goes to sleep. It wakes up when triggered by a Rank event from the Tester which tells which individual is better. In this way, it selects parents, produces children, and forms the next generation.

The Teacher is a fixed-sample-time-driven module. It has one non-alarm input port for receiving commands from the Tester. Every 100ms, it gets one command from the non-empty port and sends them to the soccer server.

5.3. Results

After 50 generations of evolution, we pick one team from the 50th generation, and let it compete with the hand-tuned team. Then after 300 generations of evolution, we let the evolved team compete with the hand-tuned team again. The results are shown in Tables 2 and 3.

An individual in generation 50 still can't beat the hand-tuned opponent. It lost all six games. But an individual in generation 300 performs much better. In six games against the hand-tuned opponent, it won five games and lost one. This shows that EA is successful here.

6. Related Work

Because of the inherent complexity of multi-agent systems, machine learning is an interesting and promising area to merge with multi-agent systems. Peter Stone and Manuela Veloso developed a method called Layered Learning to control their robot soccer players. Layered Learning is an approach to complex multi-agent domains that involves incorporating low-level learned behaviors into higher-level behaviors [15–17].

Milind Tambe and his colleagues built their soccer team on a domain-independent explicit teamwork model called STEAM [12, 18, 19]. Their agent uses a two-layered architecture, consisting of a higher-level that makes decisions and a lower-level that

Table 2. Evolved team from the 50th generation vs. hand-tuned team

Game	1	2	3	4	5	6
Evolved team	1	2	5	2	1	1
Hand-tuned team	3	4	6	3	4	2

Table 3. Evolved team from the 300th generation vs. hand-tuned team

Game	1	2	3	4	5	6
Evolved team	4	5	3	3	4	1
Hand-tuned team	2	0	2	0	3	2

handles various time critical functions linked to perception and action. There are two key aspects of their approach to teamwork. The first is the explicit representation of team activities via the use of explicit representation of team operators. Team operators explicitly express a team's joint activities. The second key aspect of teamwork is its approach to coordination and communication via the use of a general-purpose teamwork model.

Sean Luke used Genetic Programming (GP) to evolve coordinated team behaviors and actions for the soccer players [10]. Genetic programming is one kind of Evolutionary Algorithm. The objects that constitute its population are not fixed length character strings that encode possible solutions to the problem, but are programs that, when executed, are the candidate solutions to the problem. These programs are expressed as parse trees in genetic programming.

The CN model is an online model of constraint programming—more elabaration on using it to realize various constraint-solving techniques can be found in [23].

There are many other approaches to this robotic soccer domain. Some of them emphasize layered robotic architecture, such as Jung's approach based on the agent model InteRRaP [1]. Some of them emphasize machine learning. In the machine learning group, some prefer Neural Networks, others Reinforcement Learning. Interested readers are referred to [1, 9].

7. Results and Evaluation

7.1. World RoboCup98

To compare our approach with other teams' that are different in models, architectures and control methods, we took part in the World RoboCup98 which was held on July 4–8, 1998 in Paris, France.

The first game we played against *NIT Stones 98*, which took third place in RoboCup Japan Open 98. The opponent team had an interesting strategy with many of its players swarming around the ball and kicking the ball forward. There was almost no passing. One advantage of this strategy is that off-side seldom happens even if the designer does not consider avoiding off-side. One big disadvantage is that the team can only cover a small area. We won this game, with a score of 4 : 1.

We played against *Mainz Rolling Brains* in the second game. This team's strategy was to move the full-backs up in an offside trap to push the opponents' forwards back.

But its forwards didn't try to avoid offside positions, they just kept their positions near the opponent's goal. This strategy was used by many teams in World RoboCup98 and should be the best strategy if there was no offside rule. We drew this game, the score was 0 : 0. Our team's advantage is that our players can sense if they are at offside positions, and if they are, they can try to avoid that situation by moving towards their own side. Our players' low level skills like kicking backwards were not as good as those of the opponent's team. Lots of shots by the opponents were saved because their forwards were offside. Our players advanced near the opponent's goal many times, but their shots lacked adequate strength to score.

We played against *CAT-Finland* in the third game. This team's original strategy was to keep its full-backs near its own goal and its forwards near the opponent's goal. It's a fixed position strategy and it was also used by many teams in World RoboCup98. When CAT-Finland competed with Mainz Rolling Brains, the disadvantage of their strategy was shown in the score 0 : 4. When CAT-Finland played against our team, they changed their strategy to that used by Mainz Rolling Brains. Some teams belonging to this category also changed their strategy later as CAT-Finland did. We lost this game; the score was 0 : 1. Lots of shots by CAT-Finland were also saved because their forwards were offside. At one point, one of our full-backs slowed down to keep energy, so CAT-Finland's forwards got a chance to shoot. Our goalie missed the ball.

7.2. Discussion

Our team won one game, drew one game and lost one game in World RoboCup98. Although we lost the game, we don't think our team is worse than CAT-Finland. We know there are many random factors in the soccer server and network communication between the server and clients is not stable either. Winning was not our purpose. Our team was successful in the World RoboCup98 from a research point of view. It shows that constraint-based control and evolutionary algorithms are effective methods in multi-agent real-time robot design. It also shows that Java is fast enough, in our design model, to compete in a traditional real-time C++ world.

8. Conclusions

Soccer as a task domain is sufficiently rich to support AI research in a multi-agent, real-time dynamic, partially observable, nondeterministic, discrete/continuous hybrid environment. In this paper, we propose a constraint-based approach to robot control in such a complex domain. This approach has two major parts:

— Constraint Net (CN) based robot architecture. It supports composition hierarchies which allow a complex module to be incrementally composed of simpler ones. It also uses the idea of the interaction hierarchy.

— Constraint-based control methods. Constraints are specified at different levels on different domains, with the higher levels being more abstract and the lower levels being more plant-dependent. A control system can also be synthesized as a hierarchy of interactive embedded constraint solvers. Each abstraction level solves constraints on its state space and produces the input to the lower level. Constraint solvers at the same level of the interaction hierarchy are coordinated via various arbitrations to compromise among different kinds of constraint. Another way to deal with various constraints at the same level of the interaction hierarchy is to combine these constraints into one utility constraint. This combined utility constraint is to maximize the utility function. An evolutionary algorithm (EA) is used to adjust the weights in the utility function and other parameters in the controller. Based on the constraint net (CN) model, we simply treat other agents as a part of the environment. This multi-agent part of the environment also puts some constraints on the agent as the rest of the environment does.

This constraint-based approach could be applied to many other single or multiagent real-time dynamic domains. Of course the users would still have to change the interfaces and constraint-based methods to suit their domains.

In short, we have demonstrated that the CN model is a formal and practical tool for designing and implementing, in Java, constraint-based controllers for robots in multi-agent, real-time environments.

Acknowledgments

We appreciate the contributions of Ying Zhang and the comments of the reviewers. This research was supported by the Natural Sciences and Engineering Research Council of Canada and the IRIS Network of Centres of Excellence.

Note

1. *rank* and *confidence factor* are not implemented in the current soccer robot. But the concept is useful and we plan to implement it in future research.

References

1. Asada, M., & Kitano, H., editors (1999). *RoboCup-98: Robot Soccer World Cup II*. Springer-Verlag: LNAI 1604.
2. Barman, R. A., Kingdon, S. J., Mackworth, A. K., Pai, D. K., Sahota, M. K., Wilkinson, H., & Zhang, Y. (1993). Dynamite: A testbed for multiple mobile robots. In *IJCAI Workshop on Dynamically Interacting Robots*, pages 35–45, Chambery, France.
3. Heitkotter, J., & Beasley, D. (1998). *The Hitch-Hiker's Guide to Evolutionary Computation: A List of Frequently Asked Questions (faq) in Usenet comp.ai.genetic*. Available via anonymous FTP from rtfm.mit.edu:/pub/usenet/news.answers/ai-faq/genetic.

4. Itsuki, N. (2002). *Soccer Server Manual*. Available at http://sourceforge.net/projects/sserver.
5. Itsuki, N. (2002). *Soccer Server System*. Available at http://sourceforge.net/projects/sserver.
6. Itsuki, N. (1995). Soccer server: A simulator for robocup. In *JSAI AI-Symposium 95: Special Session on RoboCup*. Also available at http://ci.etl.go.jp/noda/soccer/server/index.html.
7. Kitano, H. *Robocup*. Available at http://www.robocup.org/.
8. Kitano, H. ed., *RoboCup-97: Robot Soccer World Cup I*. Springer-Verlag: LNAI 1395, 1998.
9. Kitano, H., Veloso, M., Matsubara, H., Tambe, M., Coradeschi, S., Noda, I., Stone, P., Osawa, E., & Asada, M. (1998). The robocup synthetic agent challenge 97. In Kitano, H., ed., *RoboCup-97: Robot Soccer World Cup I*. Springer-Verlag: LNAI 1395.
10. Luke, S., Hohn, C., Farris, J., Jackson, G., & Hendler, J. (1998). Co-evolving soccer softbot team coordination with genetic programming. In Kitano, H., ed., *RoboCup-97: Robot Soccer World Cup I*. Springer-Verlag: LNAI 1395.
11. Mackworth, A. K. (1993). On seeing robots. In A. Basu and X. Li, eds., *Computer Vision: Systems, Theory, and Applications*, pages 1–13. World Scientific Press.
12. Marsella, S., Adibi, J., Al-Onaizan, Y., Erdem, A., Hill, R., Kaminka, G. A., Qiu, Z., & Tambe, M. (1999). Using an explicit teamwork model and learning in robocup: An extended abstract. In Asada M., Kitano H., eds., *RoboCup-98: Robot Soccer World Cup II*. Springer-Verlag: LNAI 1604.
13. Sahota, M. K. (1994). *Real-time Intelligent Behaviour in Dynamic Environments: Soccer Playing Robots*. Master's thesis, University of British Columbia.
14. Sahota, M., & Mackworth, A. K. (1994). Can situated robots play soccer? In *Artificial Intelligence 94*, pages 249–254, Banff, Alberta.
15. Stone, P., & Veloso, M. (1998). A layered approach to learning client behaviors in the robocup soccer server. In *Applied Artificial Intelligence (AAI)*, 12. Also available at http://www.cs.cmu.edu/People/pstone/.
16. Stone, P., & Veloso, M. (1999). Team-partitioned, opaque-transition reinforcement learning. In M. Asada, H. Kitano, eds., *RoboCup-98: Robot Soccer World Cup II*. Springer-Verlag: LNAI 1604.
17. Stone, P., & Veloso, M. (1998). Using decision tree confidence factors for multiagent control. In H. Kitano, ed., *RoboCup-97: Robot Soccer World Cup I*. Springer-Verlag: LNAI 1395.
18. Tambe, M. (1996). Executing team plans in dynamic, multi-agent domains. AAAI Fall Symposium on Plan Execution. Also available on http://www.isi.edu/soar/tambe/.
19. Tambe, M. (1996). Teamwork in real-world, dynamic environments. International Conference on Multi-agent Systems(ICMAS). Also available at http://www.isi.edu/soar/tambe/.
20. Zhang, Y. (1994). *A Foundation for the Design and Analysis of Robotic Systems and Behaviors*. PhD thesis, Department of Computer Science, University of British Columbia. Technical Report 94-26.
21. Zhang, Y., & Mackworth, A. K. (1995). Synthesis of hybrid constraint-based controllers. In *Hybrid Systems II*, pages 552–567. Springer-Verlag: Lecture Notes in Computer Science 999.
22. Zhang, Y., & Mackworth, A. K. (1995). Constraint nets: A semantic model for hybrid dynamic systems. *Journal of Theoretical Computer Science*, 138(1): 211–239.
23. Zhang, Y., & Mackworth, A. K. (1995). Constraint programming in constraint nets. *Principles and Practice of Constraint Programming*, pages 49–68. MIT Press.
24. Zhang, Y., & Mackworth, A. K. (1999). A multi-level constraint-based controller for the dynamo98 robot soccer team. In M. Asada, H. Kitano, eds., *RoboCup-98: Robot Soccer World Cup II*, pages 402–409. Springer-Verlag: LNAI 1604.