A Multi-level Constraint-based Controller for the Dynamo98 Robot Soccer Team

Yu Zhang and Alan K. Mackworth

Laboratory for Computational Intelligence, Department of Computer Science, University of British Columbia, Vancouver B.C. V6T 1Z4, Canada, yzhang@cs.ubc.ca, mack@cs.ubc.ca

Abstract. Constraint Nets provide a semantic model for modeling hybrid dynamic systems. Controllers are embedded constraint solvers that solve constraints in real-time. A controller for our new softbot soccer team, UBC Dynamo98, has been modeled in Constraint Nets, and implemented in Java, using the Java Beans architecture. An evolutionary algorithm is designed and implemented to adjust the weights of constraints in the controller. The paper demonstrates that the formal Constraint Net approach is a practical tool for designing and implementing controllers for robots in multi-agent real-time environments.

1 Background and Introduction

Soccer as a task domain is sufficiently rich to support research integrating many branches of robotics and AI [3, 6]. To satisfy the need for a common environment, the Soccer Server was developed by Noda Itsuki [1] 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 [2].

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 [8–10]. CN is an abstraction and generalization of dataflow 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 [7]. A controller for our new softbot soccer team, *UBC Dynamo98*, has been developed using CN.

The rest of the paper describes CN and how we use it to model and build the controller for our soccer-playing softbot *UBC Dynamo98*. Section 2 introduces the CN model of the controller for our soccer-playing softbot. Section 3 discusses constraint-based control and shows how the controller satisfies the constraints in the soccer domain. Section 4 shows our team's performance in RoboCup98. Section 5 concludes the paper.

2 The CN Architecture of the Controller for a Soccer-playing Softbot

2.1 Modeling in Constraint Nets

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 and ports. 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. For *trace* and some other basic concepts of dynamic systems, the reader is referred to [10].

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. The semantics of a system can be obtained hierarchically from the semantics of its subsystems and their connections.

A control system is modeled as a module that may be further decomposed into a hierarchy of interactive modules. 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 sensors. Control signals flow down and state signals flow up. Sensing signals from the environment are distributed over levels. Each level is a grey 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.

2.2 The CN Architecture of the Controller

The soccer-playing softbot system is modeled as an integration of the soccer server and the controller (Fig. 1). 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 the ball.

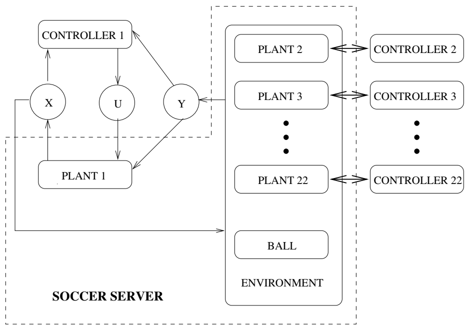

Fig. 1. The soccer-playing softbot system

For the soccer-playing softbot, we have designed the three-level controller shown in Fig. 2. The lowest level is the Effector&Sensor. It receives ASCII sensor information from the soccer server and 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 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 its evaluation up to the Planner. The highest level is the Planner. It decides which action 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.

The controller is composed of four CN modules. The Effector module combines with the Sensor module to form the lowest level Effector&Sensor. The Executor module forms the middle level and the Planner module forms the highest level (Fig. 2).

The controller is written in Java [4]. The Java Beans component architecture [5] is used here to implement the CN modules. Events are one of the core features of the Java Beans architecture. Conceptually, events are a mechanism for propagating state notifications between a *source* object and one or more target *listener* objects. Under the new AWT event model, an event listener object can be registered with an event source. When the event source detects that

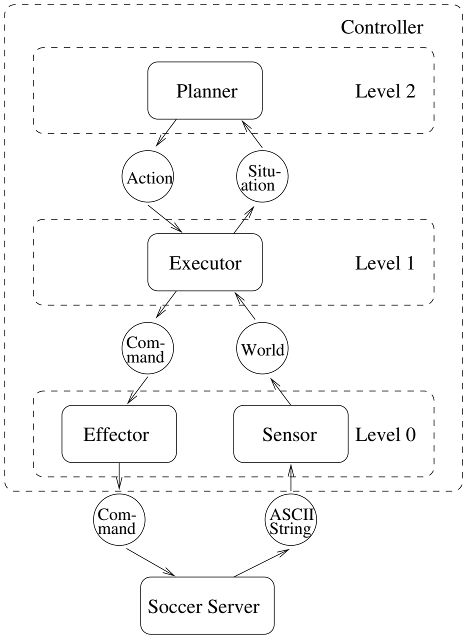

Fig. 2. The soccer-playing controller hierarchy

something interesting has happened it calls an appropriate method in the event listener object.

CN model is a data-flow model; each CN module can be run concurrently on different processors to improve the speed of the controller. Since these modules are event-driven and fixed-sample-time-driven, they are best implemented as Java threads to improve efficiency on a single CPU too. If no event arrives, they go to sleep so the CPU can deal with other softbots. In such a multi-threaded environment where several different threads may be simultaneously delivering events and/or calling methods and/or processing event objects and/or setting properties, special considerations are needed to make sure these beans properly coordinate their behaviour, using wait/notify and synchronization mechanisms.

The Sensor module wakes up when new information arrives. It then processes the ASCII information from the soccer server, 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 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, whether 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. This part of the Executor runs in the same thread as the Sensor 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* buffer. Other commands are sent to the Effector's *Sensing_command* buffers, they are *Say_message* buffer, *Change_view* buffer, and *Sense_body* buffer. The Executor goes to sleep when there is no action waiting for its processing.

The Planner module wakes up when triggered by a situation-changed event from the Executor. It then produces actions and pushes them into Executor's action buffer and sends an event to trigger the Executor to execute actions. Then it goes to sleep until a new event comes.

The Effector module is a fixed-sample-time-driven module. Every 100ms, it gets one command from each non-empty buffer and sends them to the soccer server.

3 Constraint-Based Control for Soccer-playing Softbot

Constraints are considered to be relations on a set of state variables; the solution set of the constraints consists of the state variable tuples that satisfy all the constraints. The behavior of a dynamic system is 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. 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 [8].

A controller is an *embedded constraint solver* if the controller, together with the plant and the environment, satisfies the given constraint-based specification. In the CN 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. 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 [7].

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. For example, if the action from the Planner is to intercept the ball at (x_b, y_b, vx_b, vy_b), and the state variables of the robot soccer player are (x_p, y_p, vx_p, vy_p), the constraints are $x_p + vx_p * t = x_b + vx_b * t$ and $y_p + vy_p * t = y_b + vy_b * t$.

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 scored by its team should be more than its opponent's. To satisfy this ultimate constraint, the robot has to satisfy a series of other constraints first.

These constraints have their priorities. The constraints with higher priority must be solved earlier. The constraint of knowing its position and the ball's should be solved first. Then the robot will try to solve the constraints of collision and offside. In order to win, the robot will consider some other 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 opponent's goal instead of its own goal.

It chooses actions to satisfy the constraints at this level. When the robot loses its own position or the ball's position for a certain amount of time, it sends $find\_me$ or $find\_ball$ actions down to the Executor. When the robot senses that it will collide with other players, it sends $avoid\_collision$ action down to the Executor. It also sends down $avoid\_offside$ down to the Executor if it finds itself is at offside position. The robot tries to *intercept* the ball if it senses that it is nearer to the ball than its teammates, if not, it goes to a certain position to *assist* its teammate's interception. If the robot gets the ball, it has to choose where to kick it. The action here should best satisfy the constraints listed above. The problem is that sometimes the robot can't find a kick direction that satisfy all the constraints. For example, if the robot chooses the kick direction which can make sure that its teammates can get the ball, the ball might be kicked away from its opponent's goal and near its own goal. We solve this by combining these constraints into one utility constraint. This combined utility constraint is to maximize the *utility function*:

$$U(o) = \sum_i k_i * P_i(o) \tag{1}$$

$U(o)$ is the action o's utility. $P_i(o)$ is the probability of satisfying the constraint i when taking the action o. k_i is the weight for the constraint i. The constraint solver for this combined utility constraint will output the action o 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.

We also designed a coach program using an evolutionary algorithm to adjust the weights of constraints and other parameters in the controller. The coach maintains a *population* of *individuals*. Each *individual* consists of a pair of *chromosomes*. A *chromosome* is an array of parameters, which we call *genes*. Thus, each *individual* has two copies of each *gene* as a consequence of *biparental inheritance*. The sum of each pair of *genes* determines one parameter in the robot. The coach selects the fittest *individuals* as parents via a tournament, performs *crossover* and *mutation* on parent's *chromosomes*, then passes them down to their children. We believe this kind of simulation of *natural selection* will evolve a very good robot team if enough time and supervision are given.

The robots also communicate with each other to share information and to coordinate their actions among them. For example, if one robot comes near the ball, it says "my ball" to its teammates, the teammate who gets the message will send back "kick here" if it is in a good receiving position or go away from the ball if it is also near the ball.

4 Results

To compare our approach with other teams' that differ 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**. The opponent team had an interesting strategy with many of its players swarming around the ball and kicking the ball forward. 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. 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 an chance to shoot. Our goalie missed the ball.

So 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 to compete in a traditional C++ world.

5 Summary and Conclusions

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.

The soccer-playing softbot system is modeled as an integration of the soccer server and the controller. The three-level controller is composed of four modules. The Effector module combines with the Sensor module to form the lowest level Effector&Sensor. The Executor module forms the middle level and the Planner module forms the highest level. The controller is written in Java. The Java Beans component architecture is used here to implement the CN modules and we use the Java event mechanism to implement communication among these CN modules. They are implemented in Java threads to improve efficiency.

The controller for soccer-playing softbot is synthesized as a hierarchy of interactive embedded constraint solvers. Each level solves constraints on its state space and produces the input to the lower level. We have also designed a coach program using an evolutionary algorithm to adjust the weights of constraints and other parameters in the controller.

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.

References

1. Noda Itsuki. Soccer Server System. Available at http: //ci.etl.go.jp/ noda /soccer /server.html.
2. Hiroaki Kitano. Robocup. Available at http: //www.robocup.org /RoboCup /New /index.html.
3. A. K. Mackworth. On seeing robots. In A. Basu and X. Li, editors, *Computer Vision: Systems, Theory, and Applications*, pages 1–13. World Scientific Press, Singapore, 1993.
4. Sun Microsystems. Java. Available at http: //java.sun.com/.
5. Sun Microsystems. Java Beans. Available at http: //java.sun.com/ beans/ index.html.
6. M. Sahota and A. K. Mackworth. Can situated robots play soccer? In *Proc. Artificial Intelligence 94*, pages 249 – 254, Banff, Alberta, May 1994.
7. Ying Zhang and A. K. Mackworth. Synthesis of hybrid constraint-based controllers. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, Lecture Notes in Computer Science 999, pages 552 – 567. Springer Verlag, 1995.
8. Ying Zhang and A. K. Mackworth. Constraint Programming in Constraint Nets. Principles and Practice of Constraint Programming, MIT Press, 1995, p.49–68.
9. Ying Zhang and A. K. Mackworth. Constraint Nets: A Semantic Model for Hybrid Dynamic Systems. Journal of Theoretical Computer Science, Vol. 138, No. 1, 1995, p.211–239, Special Issue on Hybrid Systems.
10. Ying Zhang. A foundation for the design and analysis of robotic systems and behaviors. Technical Report 94-26, Department of Computer Science, University of British Columbia, 1994. Ph.D. thesis.