Team Sicily

John Fry, Lyen Huang and Stanley Peters

Stanford University
Center for the Study of Language and Information
Stanford, CA 94305-4115 USA

Abstract. Team Sicily, our entry in the RoboCup-97 simulator track, is designed to achieve cooperative soccer behavior using a realistic and efficient balance of communication and autonomy. The team is implemented in the multithreading logic programming language Gaea, and builds on the dynamic subsumptive architecture model developed by Noda and Nakashima [3].

1 Background and Goals

For the past few years our Situated Agents group at CSLI has been working on the general problem of cooperative agent behavior, and especially on the role of communication in cooperative software architectures. In the process we have studied a few paradigmatic problems, notably cooperative elevator dispatch in an office building ([4] and [5]) and cooperative taxi dispatch in a city [5]. Our elevator simulations, for example, demonstrated that an autonomous but cooperative architecture with a small amount of communication is more efficient than the top-down centralized controller model. However, constant n-way communication among n agents is generally unnecessary and even counterproductive for all but the most complex tasks.

Our RoboCup-97 entry, Team Sicily (*Sicily* is one pronunciation of *CSLI*) represents a somewhat more sophisticated application for cooperative agents. Nonetheless, in our Team Sicily implementation we have attempted to preserve all the advantages of a cooperative agent architecture, and specifically the following five properties:

1. **Efficient Communication** Agents should communicate in an appropriate and timely manner, but only as often, and with as much information, as required to act cooperatively.
2. **Fault Tolerance** The failure of individual agents should cause graceful degradation of system performance.
3. **No Single Point of Breakdown** There should be no one component, such as a central controller, whose failure can bring down the entire system.
4. **Equitable Load Distribution** The work load should be evenly divided among cooperating agents, with no single overloaded component acting as a bottleneck.

5. **Open System** The system should be flexible enough to easily accommodate more agents, and the addition of new agents should not require existing ones to be reprogrammed.

The nature of the RoboCup problem (playing real-time soccer against an opponent) and the physical implementation of the Soccer Server both present interesting new challenges for a cooperative architecture approach. Team Sicily is designed to explore the role of *explicit communication* in a multi-agent system where players sequentially use a non-sharable resource (the ball) in cooperative way to obtain a specified result (get it into the opponent's goal without allowing it to get into one's own goal).

We contrast explicit communication between agents (players) with implicit exchange of information by means of reasoning from other agents' actions to their intentions. Our plan was to measure the value of explicit communication by comparing our team's performance when communication was enabled against its performance with communication disabled. Of course, the team had to conform to the RoboCup rules, an important one being that soccer players are only allowed to communicate through the soccer server—clients cannot communicate directly, but only indirectly using *say* and *hear* commands. Messages are restricted to ASCII strings of 512 bytes and only one command per simulation cycle is accepted. In such a restricted and real-time reactive system, then, the issues of *when* and *how much* to communicate take on paramount importance. Rather than issuing a *say* command at every cycle, a player agent should communicate only when there are significant changes in the environment—for example, when he becomes free and wants the ball passed to him, or is being blocked and wants to pass.

2 Program Design

2.1 Subsumptive Architecture and Gaea Programming Language

Team Sicily is implemented in Gaea [2], a multithreading logic programming language. A Gaea program consists of (1) multithreading *processes* and (2) *cells* that store fragments of programs. The cell is the level at which name to content mapping and background conditions are implemented. The *environment* of a given process is the collection of cells which are currently active for it (Figure 1).

In general we have found Gaea useful as an implementation language because (1) like Prolog, it facilitates rapid prototyping, and (2) it allows cooperative agents to be implemented straightforwardly as processes which can be forked, suspended, resumed, and killed during the course of an application.

For the Team Sicily implementation, the most important feature of Gaea is its dynamic version of Brooks' subsumptive architecture [1]. For a given process, programs in deeper cells in the environment are valid unless they are explicitly negated by programs in shallower cells. Thus the structure of the cells can be viewed as a dynamically formed inheritance hierarchy, permitting *dynamic subsumptive programming*: programs in higher functional layers override programs

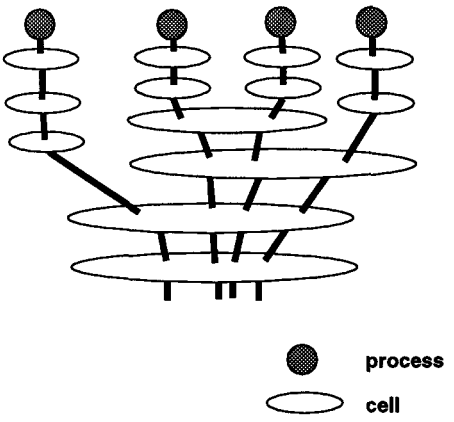

Fig. 1. Multiple Gaea processes in their own environments

in lower layers, and the system can backtrack to lower levels when a higher level program fails to perform its function. Furthermore, the cell modules can be pushed, popped, and swapped on the fly in order to adapt to new conditions without recompiling the whole system.

This technique was originally exploited for soccer in Itsuki Noda's 1996 RoboCup team [3]. Noda's implementation used Gaea's layering of modules in order to flexibly handle the interactions between high-level modes (e.g. *offense, defense*) and low-level modes (e.g. *chase-ball, dribble, shoot, pass*).

2.2 Structure of Individual Players

Our basic design uses four controlling threads for each player: *command*, *watchdog*, *listener* and *sensor* threads. Primary control belongs to the *command* thread. Here, a set of individual action cells is loaded and unloaded on top of a basic set of Gaea cells in which technical information (server address, player memory, etc.) and general team information is stored, as illustrated in Figure 2. The top cell has the highest importance in the execution tree; if no rules apply from the top cell, then the next cell is analyzed, and so on. This lets lower level cells to control basic player actions such as running and maintaining positions, unless overruled by higher cells.

The *watchdog* thread keeps track of the passage of time and the strategic development of the game. As the game progresses, the watchdog adjusts the strategy cells in the command thread in order to change the player's positions and passing routes. Similarly, the *listener* thread takes incoming messages and, after analyzing the messages and deciding whether or not to accept the communicated command, changes the cells in the command thread accordingly.

Based on each round of sensory information, processed by the *sensor* thread,

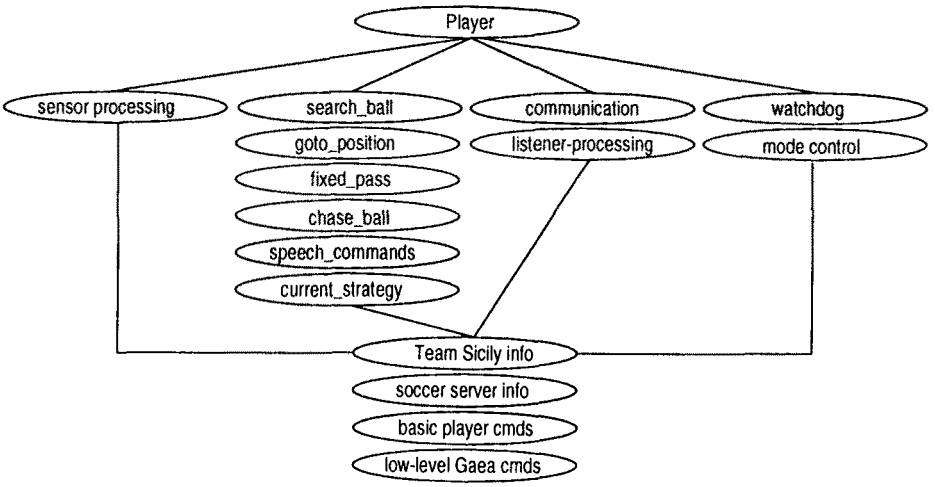

Fig. 2. Process and cell structure of a player

the player can either push an action cell onto the command thread, in order to react to a new situation, or change the priorities of actions by swapping the order of cells in the thread. This design also allows a player to delay executing a play in order to take care of more immediate concerns, like avoiding an enemy player; once the interruption has passed, the top cell is popped and the tactical or strategic cell again takes highest priority.

3 Lessons Learned

In the first round of the tournament we had the misfortune to face Humboldt University (Germany) and Tokyo Institute of Technology, the teams who went on to to finish in first and second place in the tournament, respectively. In any case, Team Sicily lost all three games it played, scoring 0–25 against Humboldt, 1–4 against Chubu University (Japan), and 0–23 against Tokyo Institute of Technology.

3.1 Difficulties with Gaea

Perhaps the greatest handicap our team faced was the sluggish speed of our implementation. Since the Gaea language, which is itself implemented in Lisp, is still under development, it lacks some of the standard optimizations found in production-quality logic programming languages. Bugs and brittleness in Gaea would occasionally cause stack overflows in threads, causing individual players

to crash. While the team could still function with diminished numbers of players, it was at a definite disadvantage missing three defenders, as happened early in the tournament. One lesson that can be drawn regarding the implementation language is that for real-time applications like RoboCup, the standard logic programming operations of backtracking, recursion and unification are only appropriate for high-level reasoning. Lower-level tasks like I/O and position calculations would have been better implemented in appropriately lower-level languages.

3.2 Effects of the RoboCup Domain

Another conclusion we have drawn is that, paradoxically, increased explicit communication improves teamwork (i.e. produces behavior closer to real soccer), yet degrades performance within the RoboCup domain. This is because the constraints imposed by the RoboCup rules and the soccer server implementation serve to discourage communication. Player communication is limited to one message per round, and there is no guarantee that the intended target will hear the message if another player in the area speaks at the same time. Moreover, communicating expends as much time and effort as moving or kicking. In order to overcome these bottlenecks, we were forced to limit communications severely, to prevent players from wasting precious cycles communicating instead of chasing the ball. In the end, our players spoke between one and ten percent of the times they had the opportunity to, depending on circumstances. This compromise allowed just barely enough communication to facilitate teamwork.

3.3 Effects of Communication on Player Actions

From qualitative data based on observation of our team's performance with and without explicit communication, we observed two primary differences between the communications-enabled condition and the non-speaking one. The first difference lay in the communicating team's ability to react more quickly to a developing situation. On offense, for example, if a pass did not arrive at its target location, the players could turn and react more quickly to the ball's new location and movement when they could inform each other about the ball's position. And when the defense was able to communicate the ball's position to a player, he would react to it more effectively, especially if the ball had been kicked out of his view. Instead of wasting time searching 360 degrees for the ball, the player was able to quickly turn to the ball's last reported location and begin searching there.

The second improvement lay in the team's strategy and tactics. Without communications, the players were more predictable and less effective in dealing with their environment. Passes and movement were executed only in patterns we hard-coded into their behavior, so each successive game tended to appear similar to previous ones. With communications enabled, the games had the same overall strategic patterns as originally coded, but the individual and small-group tactical play became significantly more dynamic. The team tended to deal with enemy

players more effectively, passing around and defending against them, and this was reflected in the increased proportion of time the communicating team spent in control of the ball.

We are currently in the process of validating the above qualitative observations with a suite of quantitative tests. For example, we are looking at the difference between soccer players that store position information in an agent-centered way (as in Itsuki Noda's original program) compared with players that convert the information to an absolute (or at least, bird's eye) coordinate system and store it that way. Originally we found that the latter was less efficient because agents always had to reconvert the information to self-centered form, based on their current (possibly new) position, before using it. We hope to confirm this with statistical tests comparing situated vs. unsituated memory for this type of task.

4 Some Reflections on RoboCup and Communication

An important point brought out at RoboCup 97 is the difference between implicit and explicit communication. During the workshop, many participants downplayed the need for explicit communication and concluded that teamwork was best coordinated through implicit understanding. We are convinced, however, that teamwork and coordination are better managed by explicit rather than implicit communication, and we can't help but wonder how strongly their conclusion was biased by the RoboCup guidelines.

Consider real-world professional soccer players. Certainly verbal communication is not always an option for players to coordinate their teamwork. Among other problems, distances on a soccer field are large, and messages can sometimes be intercepted by the eavesdropping opposing team. Nevertheless, real-world soccer players do communicate extensively, albeit not always verbally. To illustrate this point: Suppose one took one player from each of several professional teams and combined them into a new team. At first they would communicate verbally quite a bit as they adapted to each other's playing style. The more they played together, the less verbal communication they would need. This does not entail, however, that the overall amount of communication has decreased; in fact, the content of the messages can remain constant even as the *medium* changes. Instead of a relatively long, slow verbal message, 'body language' becomes more prevalent. In this new 'protocol', implicit communication—based on understanding what a player intends when he makes a movement or pass in a certain way—establishes a space of possibilities within which explicit communication—now 'encrypted' into a faster medium of gestures—coordinates the details of the actual execution of the play.

To illustrate, consider the give-and-go play. Player A approaches Player B while dribbling the ball. Opponent C stands between Player A and his target goal. Player A makes a quick pass and then runs to the reflected pass from Player B. The positions of the players provide the implicit understanding on which cooperation in the situation can be based. Nevertheless, without eye contact, a

verbal cue or perhaps even the words "give and go!", the actual coordination of the play will be difficult to effect.

Since such a protocol is heavily dependent on the familiarity players build up with each as they play together for long periods of time, the code is very difficult for the opposing team to read. Verbal communication is still necessary in situations where the target player may not be able to 'listen' visually. These include situations where an opposing player is approaching from behind ('man on!'), where a teammate is nearby and ready to receive a pass but is not in the direct view of the player with the ball, and warnings about the ball's location relative to a player who has lost track of it.

4.1 Whither RoboCup?

An observation frequently made by competitors at RoboCup 97 was that explicit communication requires too many steps to coordinate teamwork, since it requires one player to propose a play and other players to confirm its execution. If the conclusion is correct, it puts us squarely in the dilemma of determining exactly what is the focus of RoboCup. If the aim is to model the real world, then the current system of communication, we agree, constrains teams too much from using explicit communication as a key element in their strategy. The soccer server's communication mechanism is too costly in terms of player actions (with speaking equivalent to running or kicking) as well as too limited in number of channels (allowing a player to hear only one message from his team per cycle).

If, however, RoboCup is not intended to model the real world but only to provide a forum for software and hardware agents, then explicit communication can be made an even more effective tool. Communication between software agents can take place much more quickly than verbal communication between humans. Hardware robots have the option of using wireless communications to transmit data at very high rates. Under this view, RoboCup should take advantage of the available bandwidth and enable the player agents to not only propose strategies, but also negotiate and even converge on consensus about them.

Acknowledgements This work was supported in part by the Information Technology Promotion Agency, Japan, as part of the R&D of Basic Technology for Future Industries "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization), and in part by a summer internship for Symbolics Systems students sponsored by CSLI at Stanford. We're very grateful to Itsuki Noda, Stefan Kaufmann and Hideyuki Nakashima for their assistance.

References

1. R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–160, 1991.
2. H. Nakashima, I. Noda, K. Handa, and J. Fry. GAEA programming manual. Technical Report ETL-TR-96-11, Electrotechnical Laboratory, Tsukuba, Japan, 1996.

3. I. Noda and H. Nakashima. Multiagent soccer on Gaea. In *Proceedings of the International Symposium on New Models for Software Architecture IMSA-96*, pages 9–14, Tokyo, December 1996. IPA.
4. S. Peters and J. Fry. Real-world applications for situated agents. In *Proceedings of the International Symposium on New Models for Software Architecture IMSA-95*, pages 7–14, Tokyo, October 1995. IPA.
5. S. Peters, J. Fry, and S. Kaufmann. Communication strategies for cooperative behavior. In *Proceedings of the International Symposium on New Models for Software Architecture IMSA-96*, pages 27–36, Tokyo, December 1996. IPA.