Abstract

The RoboCup Simulator League provides an excellent platform for research on swarm computing. The University of Virginia RoboCup team is interested in how group behaviors emerge from collections of actors making decisions based on local information. Our Wahoo Wunderkind simulator team is designed for experimenting with behavioral primitives defined over groups and mechanisms for combining those behaviors.

Introduction

A fundamental challenge for computer scientists over the next decade is to develop methods for creating, understanding, and validating properties of programs executing on swarms of communicating computational units. The key difference between swarm programming and traditional programming is that the behavior of a swarm program emerges from the total behavior of its individual computing units. Unlike traditional distributed programming, swarm programs must deal with uncertain environments and mobile nodes. The scientific challenges of swarm computing are to understand how to create programs for individual units based on the desired behavior, and conversely, how to predict high-level behavior based on the programs running on the individual computing units.

We began the Wahoo Wunderkind FC during the summer of 2001 for the RoboCup 2001 competition. In the spring of 2002, we continued development with the inclusion of fifteen new undergraduate students. Over the summer of 2001, we concentrated on developing a modular architecture for experimentation using the principles of swarm programming. Recently, our team has researched methods of combining actions, methods of using learning methods to incorporate noisy data into the player's world model, methods of simulating the world to account for the infrequency of sensor data, methods for players to interpret the world using different simulation levels of detail (SLOD), and more complex actions. Originally, we began with source code from the Mainz Rolling Brainz Team from the 2000 competition because it was well documented, freely available, and modular in design. Our team is based on a clear modular structure, uses a `MasterControl` which has a `WorldModel`, a `Body` to execute `Actions`, and a facility to evaluate recommended actions from many `Behavior` modules.

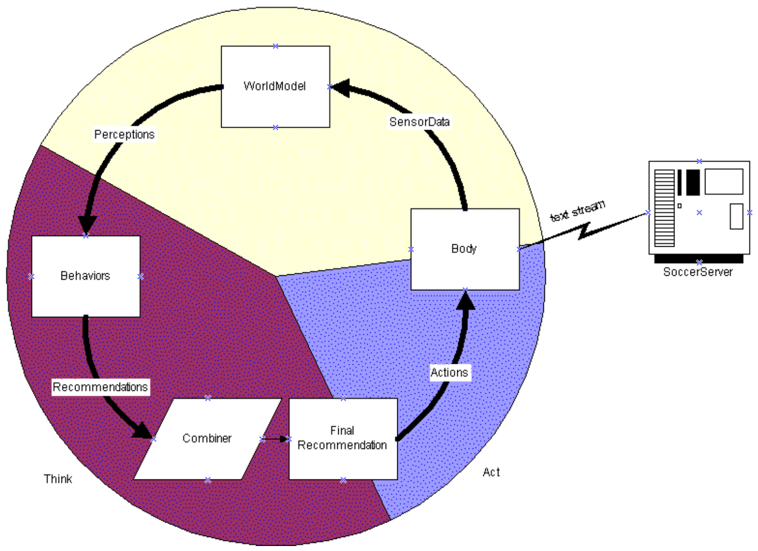
Swarm Programming

Our work focuses on creating and reasoning about swarm programs in principled ways. We believe these programs are best implemented by combining primitives defined over groups of agents. We are developing a library of primitive operations for swarm programs. A primitive defines a functional behavior of a swarm of devices, such as "disperse" or "flock". These primitives are described formally in terms of constraints on the state of the devices in the swarm. For example, disperse has the constraint that no two devices are within a certain distance. In addition to functional behavior, non-functional behavior of an implementation of a primitive is formally described. With respect to robot soccer, this technique captures uncertainty in the agents' environments, resilience to malfunction of other agents, and unpredictability of opponents' actions.

We are investigating different mechanisms for composing swarm primitives to create new behaviors. The goal is to reliably predict the properties of the resulting program. In addition, we hope to synthesize programs by combining primitives with known functional and nonfunctional properties. We will take a functional description of a swarm program along with formal models of its execution environment and devices, and select a combination of primitive implementations from a library based on the desired properties.

Architecture

Our agent consists of a `WorldModel`, `Body`, and `Behaviors`, where each `Behavior` is implemented as a module. `MasterControl` has a three-part architecture, divided into `listen`, `think`, and `act`. *Listen* takes sensory data received by the agent through the `Body` and inserts the data into the `WorldModel`. *Think* loops over each `Behavior`, determines the behavior's usefulness and then returns a recommended `Action`. *Act* commands the body to execute a recommended action.



`WorldModel` stores all the information that comes from the server in an object oriented format. The objects in the world have attributes such as position, velocity, and age. The world model can be logged in an XML format for debugging with a replayer. The flexibility of XML allows the future development of more complex debugging without deprecating our existing replayer that was based on the MRB replayer.

The `Body` represents a layer of separation between the higher order control and the simulated robot. The body has the ability to execute a complex action and return sensor information from the server. All commands sent to the server pass through the body. This structure proved useful for debugging, since it allowed us to isolate unusual behavior by distinguishing between commands that were generated improperly and commands that were never sent. Additionally, this structure will allow our design to support a future implementation on physical robots.

Behavior Modules and Perceptions

A *behavior module* consists of a *behavior, perception*, and a *recommendation*. Each *behavior* represents a primitive desire or action such as dispersing, centering, or pushing up on the ball. Some behaviors are designed to establish and maintain global properties of the team. For example, disperse strives to maintain the property that no two players are within a certain distance of each other. Other behaviors deal with special situations, often associated with the closest player to the ball, such as passing or shooting. Each behavior has two main methods, one to determine its usefulness and the other to generate a `Recommendation`. A behavior uses *perceptions* to view the world model in different levels of detail. The check for usefulness is done to eliminate unnecessary calculations. For example, if a player does not have the ball, it need not calculate the best direction to kick. If a behavior is useful, additional perceptions are used to calculate information about the world state in order to calculate parameters for the suggested action.

A *perception* is a collection of functions that operate on the world model to interpret the state of the game. The complexities of different perceptions vary considerably, from merely checking the play mode to performing intercept calculations. The Perceptions are written as utility libraries, so that multiple behaviors can share perceptions. If efficiency becomes a problem, we intend to implement a caching facility in the perceptions.

The `Recommendations` from each module are compiled into a list and returned to `MasterControl`. `MasterControl` then uses combination logic, based on the principles of swarm programming, to select the best recommendation, and thus the best behavior for a particular situation. The `ActionList` from the winning recommendation is sent to the body to be transmitted to the server.

Recommendation and Action List

The *recommendation* produced by each behavior module contains a grade and an `ActionList`. An action list is a list of *actions* to be executed in one server cycle. Currently, a Grade is implemented as a floating point number ranging between 0 and 100. A grade of 0 represents impossibility, while a grade of 100 forces a specific action. For example, when the goalie judges that it can catch the ball, it forces the catch, even if another module such as center recommends the goalie move to the center of the box. If two modules return recommendations of 100, the one that was returned first wins.

All of our actions have an associated duration. For instance, the kick command takes a complete server cycle, so the Kick Action's duration is one. Turn Neck's can be sent more than once per server cycle, hence, they have a duration of zero. We have successfully implemented a combination method that will strip the zero-duration actions out of losing Recommendations and append them to the winning Recommendation. One result of this combination is that a player can dribble the ball up the field and turn his neck to be sure the ball stays in his line-of-sight. This is significant because the `DribbleBehavior` and the `ScanBehavior` are strictly independent.

Strategy

We put most of our efforts into developing a defense. The principles of swarm programming proved to be very useful in balancing the various requirements of a defensive player. We have independent modules that seek to establish different, often conflicting, properties. For example, don't be too close to teammates, center on the ball, try to push the offside line up the field. Vector summing is an effective tool for combining defensive positioning behaviors.

Programming the offensive objectives of a soccer agent with swarm programming practices presents some new challenges. Swarm programming is best suited to large numbers of relatively simple actions. For the commands accepted by the soccer server, this is an excellent application. However, an attack on goal by a single player is best implemented as a complex set of conditionals, usually including a long list of actions to be executed sequentially. Behaviors of this type have a unique structure, and present a particular challenge in the development of combinational methods. We are investigating how behavior based control can be combined with long range planning.

Level of Detail

Currently, we are researching methods to extrapolate strategic and high level information about the world and methods of directing all of the players at an individual level. One technique of utilizing higher-level information about the world and enabling players to work and reason as a group is simulation level of detail (SLOD).

SLOD will allow us to effectively control groups of soccer players as they coordinate and compete in a complex environment. The technique builds abstractions to represent the limitations of a player's maneuverability and passing ability. Low-level, individual, abstractions are built using such data-centric methods as principle components analysis and temporal differencing. A hierarchy of SLODs, developed using optimization methods and heuristics, allows us to describe individual players and even groups of players in progressively simplified ways. These simplified versions of players expedite the evaluation of strategy search algorithms and provide a common framework for describing heterogeneous agents. The shared player description framework allows higher levels of control to abstract the implementation details of lower levels, thus supporting reusability and interchangeability. For example, a defensive strategy that entails a lot of movement from the defenders need only evaluate the energy level of the defensive players (a simple SLOD) without considering the overall state of each defender. Likewise, the lower levels of control for the defensive players will use more detailed SLODs to partition the defensive portion of the field among the individuals. SLODs fit well with swarm programming as we can consider swarm primitives that operate at different levels.

It is difficult to analyze one component of a soccer agent without compensating for a large number of external variables. The implementation of simulation SLOD in our agent will overcome these difficulties while taking full advantage of our modular structure. New perceptions which perform the SLOD calculations can be added without affecting any existing functionality. New Behavior Modules can take advantage of SLOD without deprecating existing modules.

Summary

Our team demonstrates that it is possible to produce complex behavior in a modular and structured way. Our architecture is designed to make it easy to develop new primitive behaviors and adapt how those behaviors are combined. The framework upon which our team is constructed is well designed and should facilitate and further research in swarm computing.

Authors

Team Leaders: David Evans and David Brogan
Undergraduate Team Members:

Keen Browne
Ye Peter Chen
Serge Eglemen
Rahul Gupta
Nathan Hoobler
Sameer Huque
Derek Juba
Chris Mason
Ben Mott
Jon McCune
Sharmil Shah
Ford Sleeman
Chris Snook
Arsalan Tavakoli
Paul Turner
Mike Peck
Robbie Yan