Team GAMMA: Agent Programming on Gaea

NODA, Itsuki[1]
noda@etl.go.jp

ETL
Umezono 1-1-4, Tsukuba
Ibaraki 305, JAPAN

1 Introduction

We are developing a new software methodology for building large, complicated systems out of simple units[Nakashima, 1991]. The emphasis is on the architecture used to combine the units, rather than on the intelligence of individual units.

We call the methodology "organic programming", referring to the characteristics of organic systems, *context dependency* and *partial specification*. Our approaches toward those characteristics are situatedness and reflection. Situatedness corresponds to the former characteristic, and reflection to the latter.

In this paper we describe an application of organic programming language, Gaea, to the programming of players of a multi-agent game, soccer. Our approach to multiagent systems is recursive. Each agent is programmed in a multiagent way.

In programing a complex agent like soccer player, we need various kinds of mechanisms of mode-change, interruption and emergency exit. We propose "dynamic subsumption architecture" as a flexible methodology to realize such mechanisms.

2 Gaea

Gaea[Nakashima *et al.*, 1996] is a logic programming language using organic programming methodology [Nakashima, 1991]. From the viewpoint of logic programing, Gaea is a variation of Prolog with the following new features:

- Multi-process (multi-thread): In Gaea, we can start a new process by "fork(GOAL)" predicate, by which "GOAL" is solved independently from the original process. All process share cells describes below.
- Multi-environment: In Gaea, definitions of predicates are stored in "cells". "Cell" is similar to "module" in recent prolog systems. The main difference of cell and module is that cell-structure (environment described below) is dynamic, while relation between module is static. Each process has an environment that is a list of cells, and searchs for matching definitions in the list. Moreover, the process can manipulate its and other's environment dynamically.

Figure 1 shows a conceptual image of "processes" and "cells".

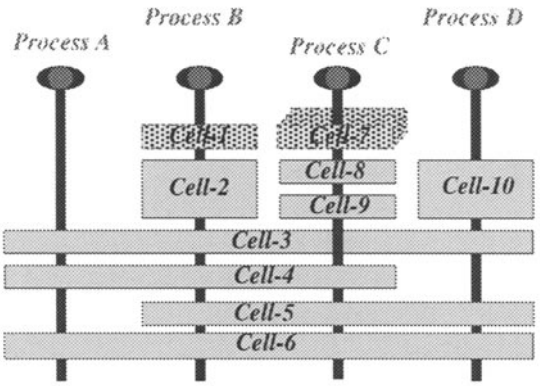

Fig. 1. Multi-Process and Multi-Environment

3 Programming Players in Gaea

3.1 Dynamic Subsumption Architecture

In programming a complex agent like a soccer player, it is essential that the agent accepts many modes. The same agent may respond to the same input differently according to the mode. For example, in soccer, a player may react differently to a ball according to whether his team is on offense or defense.

The mode of the agent must be changeable flexibly. The mode changes as a result of agent's action and plan, and it also changes according to the change of situation. Moreover, modes will be divided into a couple of levels. For example, in soccer, "offensive mode" and "defensive mode" are relatively high-level modes. On the other hands, "chase-ball mode" and "dribble mode" are relatively low-level modes. In addition to it, we can consider a couple of types of changes of modes. For example, an agent may "goto" a new mode, or it may "enter" a new mode and "return" when it exits from the new mode.

In order to realize such flexible mode-change, we use an extension of subsumption architecture[Brooks, 1991], called *dynamic subsumption architecture*. This architecture is the combination of subsumption architecture and dynamic environment change. Since subsumption architecture assumes fixed layers of functions, it is either difficult or inefficient to implement multiple modes on top of it. It is straightforward in organic programming, using context reflection.

We assume the following conditions for dynamic subsumption architecture:

1. Overriding is done by name
2. The same ontology (global name) is used by all cells.

Actually, dynamic subsumption archtecture is implemented using dynamic environment manipulation (Figure 2).

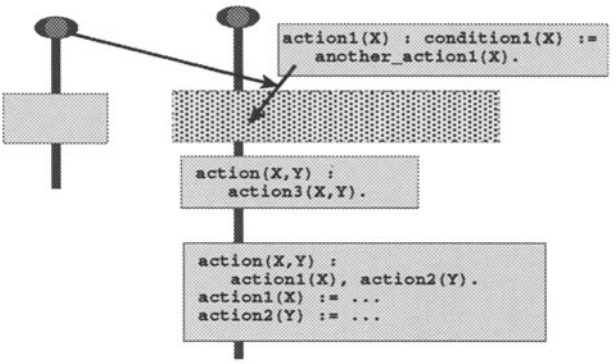

Fig. 2. Dynamic Sybsumption Archtecture in Gaea

3.2 Overview of the Architecture of a Soccer Player

As described in Section 1, we program an agent in multi-agent way. A soccer player consists of the following processes(agents):

- **Sensor Process**: receives sensor informations sent from the Soccer Server, analyzes them, and puts the results into the common cell.
- **Command Process**: sends control commands to the Soccer Server. It is controlled to send one command per 100 milli-seconds, because Soccer Server accepts only one command per 100 milli-seconds. In other words, this process is a resource manager of sending control commands.
- **Action Process**: controls low-level modes of player's action by manipulating the environment of the command process.
- **Object Detection Process**: checks the way to the target of chasing or kicking, and changes the behavior by modifying the environment of the command process. if there are objects on the way,
- **Communication Process**: controls high-level modes of player's action according to the message from referee and teammates.

The top level of each process is a loop that repeatedly calls "`cycle(top)`". It is defined in the "`basic`" cell, that is shared all processes, as follows:

```
/* In "basic" cell */
toplevel() := repeat(cycle(top)).
```

3.3 Basic Action Modes

A player is situated in certain action modes, such as *chase-ball*, *dribble*, *shoot*, *pass*, and so on, during the play. Each mode is defined in one or in a set of cells.

The mode is changed by pushing cells into the environment of the command process of the player. For example, in *chase-ball* mode, the role process pushes the following "`chase`" cell into the environment of the action process. In each cell, "`cycle(top)`" for the command process is defined accordingly.

```
/* In "chase" cell */
 cycle(top) ::=
    target(Target),
      /* turn to the target */
    cycle(turn(Target)),
      /* and dash */
    cycle(dash(Target)).
 cycle(turn(Target)) ::=
    direction(Target,Dir),
    turn(Dir)./* send turn com. */
 cycle(dash(Target)) ::=
    distance(Target,Dist),
    dash(Dist)./* send dash com. */
 target(ball).
```

In Gaea, the environment of a process can be manipulated by the process itself and also by other processes. In out program, the action process manipulates the environment of the command process in order to control basic action mode. The followings are a sample of definition of "`cycle(top)`" for the action process.

```
/* In "dribbler" cell */
 cycle(top) :
    distance(ball,BDist), BDist < 2,
      /* if ball is near */
    distance(goal,GDist), GDist < 10 :=
      /* and in front of goal */
    change_command_env([shoot]).
      /* then shoot !! */
 cycle(top) :
    distance(ball,BDist), BDist < 2 :=
      /* if ball is near */
    change_command_env([dribble]).
      /* then start dribble */
 cycle(top) :=
    change_command_env([chase]).
      /* otherwise chase the ball */
      /* Replacing new mode cells */
      /* into the environment of */
      /* the command process */
 change_command_env(NewMode) :
```

```
    command_process(PID),
    environment(Env,PID),
    include(NewMode,Env) := .
change_command_env(NewMode) :
    command_process(PID),
    environment(Env,PID),
    remove_mode_cell(Env,NEnv),
    append(NewMode,NEnv,NewEnv),
    set_environment(NewEnv,PID).
```

In this program, the action process checks the situation of the field, selects one "**shoot**", "**dribble**" and "**chase**" cells, and swaps it with the current mode cell in the environment of the command process.

The merit of this architecture, in which the action process manipulate the environment of the command process, is that the command process can send commands constantly even if it take a lot of time for the action process to check the situation.

Control of high-level mode is also realized in the same manner. In this case, communication process manipulate the environment of the action process in order to change higher-level of behavior like "chasing ball" and "supporting" (Figure 3).

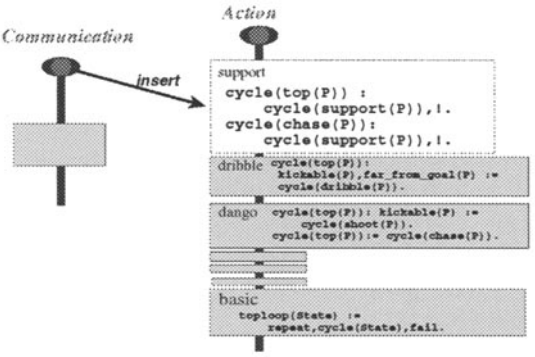

Fig. 3. Changing Behavior by Communication Process

3.4 Modifying Basic Actions

It is easy to realize small modification of behavior by manipulating environments.

In order to avoid objects, the object detection process pushes the following "**avoid**" cell into the environment of the command process.

```
/* In "avoid" cell */
 target(Target) :
    remove_cell(avoid),
      /* remove this cell */
    target(T1),
      /* find original target */
    Target = T1 + rel_dir(60).
      /* shift relative angle in 60 */
```

This cell only override the definition of "`target(Target)`" rather than "`cycle(top)`". Moreover, this definition is used only once, because the "`avoid`" cell is removed when "`target(Target)`" is called. (See `remove_cell`.)

The program for the object detection process is as follows:

```
/* In "check_object" cell */
 cycle(top) :
    command_process(PID),
    environment(Env,PID),
    in(Env,target(Target)),
      /* get Target information */
      /* in the command environment */
    not_clear_to(Target),
      /* check if the target direction */
      /* is clear */
    add_action_env([avoid]).
      /* add avoid into the */
      /* command environment */
```

3.5 Interruption

Interruption is a kind of temporal change of modes. The feature of interruption is to suspend current job, execute another job, and return to execute the current job.

For example, consider the case that the player must reply its position immediately when a teammate says "`tell your position`". This behavior can be realized by pushing the following cell into the environment of the command process.

```
/* In "tell_pos" cell */
 cycle(_) :
    estimate_current_pos(X,Y),
    unum(UNum),
    say([UNum,position,X,Y]),
    remove_cell(tell_pos),
    fail.
```

This definition is used whenever "cycle/1" is called, and discarded after the execution in failure. Then original definition of "cycle/1" is called.

We can control the level of interruption by specifying the argument of "cycle/1". For example, consider the case the player loses site of a ball when it is chasing the ball. In this case, the other process will interrupt the command process to search the ball. In the chase mode, the command process calls "cycle(turn)" and "cycle(dash)" sequentially. If the interruption is defined as "cycle(_)" like the above example, the definition is executed when "cycle(dash)" is called. But the interruption of searching ball will change player's direction, so that the player may dash to the wrong direction. In order to avoid this, we can simply define this interruption as "cycle(top)" as follows:

```
/* In "search" cell */
 cycle(top) :
    target(Target),
    cycle(look(Target)),
    remove_cell(search).
 cycle(look(Target)) :-
    cycle(turn(Target)),
    cycle(wait_visual_sensor),
    new_info(Target),cut.
 cycle(look(Target)) :-
    cycle(search(Target)).
 cycle(search(Target)) :-
    turn(90),
    cycle(wait_visual_sensor),
    new_info(Target),cut.
 cycle(search(Target)) :-
    cycle(search(Target)).
 cycle(wait_visual_sensor) :-
    current_time(CTime),
    repeat,
    usleep(100000),
    cycle(v_sensor_newer(CTime)),cut.
 cycle(v_sensor_newer(CTime)) :
    visual_sensor_time(VTime),
    VTime > CTime,cut.
```

We can use any kinds of infon on the argument of "cycle/1", so that it is able to realize flexible control of interruption.

3.6 Emergency Exit

It is also possible to realize emergency exit of execution of plays. For example, consider the case that the player must go back to its own goal immediately because of the clutch. This behavior is realized by pushing emergency, guard_goal

and **chase** cells into the environment of the command process, where these cells have the following definitions.

```
/* In "emergency" cell */
 cycle(top) :
      /* when cycle(top) is called, */
      /* exit emergency */
    remove_cell(emergency) :=
    fail.
      /* all other cycle(_) fails */
 cycle(_) := fail.
/* In "guard_goal" cell */
 target(owngoal).
```

4 Conclusion

Thorough our experience of programming soccer players in Gaea, we are convinced that the language is suitable for complex multiagent programming.

In programing complex agents, we need various kinds of mechanisms of mode-change, interruption and emergency exit. We proposed "dynamic subsumption architecture" as a flexible methodology to realize such mechanisms.

We could also program the system in layer-by-layer manner, from a simple behavior to more complex ones, taking full advantage of subsumption architecture design.

References

[Brooks, 1991] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–160, 1991.

[Nakashima *et al.*, 1996] Hideyuki Nakashima, Itsuki Noda, Kenichi Handa, and John Fry. GAEA programming manual. TR-96-11, ETL, 1996. Gaea system is available from `http://cape.etl.go.jp/gaea/`.

[Nakashima, 1991] Hideyuki Nakashima. New models for software architecture project. *New Generation Computing*, 9(3,4):475–477, 1991.