Kappa-II

NODA, Itsuki

CSLI, Stanford / ETL

1 Introduction

In order to realize flexible strategic planning in multi-agent systems that are working in dynamic environment, it is necessary to provide a mechanism to integrate hierarchical planning (include team planning) and reactive behavior. The main issues of this integration are:

- how to switch the context of plan

In the dynamic environment, it is important how to terminate making and executing a plan when the environment changes so that the plan is not useful any more.

- how to organize multiple planning

It is better that agents in a complex environment can have ability to making multiple planning, because such agents may have multiple goals in the same time. For example, in the case of soccer, while agents have an obvious goal “win the game (or score goals)”, also the agent should have another instinctive goals, that is “not to miss their position”, “follow the rule”, and so on, in the same time. The similar requirement will be happen when agents try to make a consensus by communication during they were acting something. So, it will make the problem simple that the agent has parallel planning process, that is action planning and communication planning.

In order to attack these issues, we are proposing a programming language called Gaea and programming methodology on it.

2 Team Development

Team Leader: NODA, Itsuki

Team Members:

NODA, Itsuki

- Stanford Univ. / ETL
- USA
- visiting scholar
- attended

Damon Otoshi

- Stanford Univ.
- USA
- under graduate student

- not attended
- Stanley Peters
 - Stanford Univ.
 - USA
 - Professor
 - not attended

Web page <http://ci.etl.go.jp/noda/research/kyocho/kappaII/>

3 World Model, Communication, and Skills

The World Model of kappa-II is simple. They scan visual information from the server and calculate their global positions using a revised version of `libscclient`. Then, it calculate positions of visible objects. In order to handle the ball smoothly, they also simulate movements of the ball in few steps ahead. Using this world model, a player calculates suitable parameters of turn, dash and kick commands.

Our players use the communication to require passes to a ball player. When a team is offending (keeping the ball) and a player A is keeping the ball, another player B requires if player B is locating a good position to receives the ball. Then player A can decide whether he should pass the ball to player B or not by itself.

For lower level of skill to handle the ball, we refered the code of CMUnited. Especially, circle-kick skill and intercept skill is based on the code. For higher level, we use multi-thread logic programming to behave reactively to the dynamic environment. The details are described below.

4 Strategy

As discussed in the introduction, the main issue of agent programming is:

how to combine hierarchical planning and reactive behavior

In order to solve this issue, SOAR[5], that is used STEAM system in ISIS team[6] uses hierarchical structure of operators. In SOAR, the hierarchical structure are realized by asserting structured state data to identify where a system is extracting the plan in a decision tree. Then each rule recognizes the state of decision process by checking the state in its condition part.

Our approach takes the different realization using the similar representation of hierarchy. When a operator is selected, then the system forks a thread for execution of the operator. The original thread remains and continue to check the condition is satisfied. The new thread checks conditions of all sub-operators in the case the operator is complex operator, or execute action. In other words, All operators on a path in a decision tree becomes threads and keep alive while the path is active.

In addition to it, the system allows a parallel planning. If a operator is denoted as *multiple*, then all sub-operators are forked immediately, and keep alive until the mother operator is alive.

The merits of this architecture:

- We can realize quick and slow planning by controlling the priority of execution of each thread. Basically, high level operator does not change so quickly, while low level operator should be checked frequently. In order to realize such features, we can specify sleeping durations of each cycle of execution of thread.
- Using *multiple* operators, we can easily realize instinctive processes. Instinctive processes should always be alive and continue to check the condition. *multiple* operators enables to realize such instinctive processes in a hierarchical manner. In order to reduce the cost of such checking,

Fig. 1. Example of a tree of operators

5 Special Team Features

We use Gaea to implement our players. Gaea is a logic programming system [2, 3, 4], which has the following features:

- prolog style *logic programming* system
- *multi-thread* with flexible control mechanisms
- *dynamical program manipulation*

Because Gaea is *logic programming*, it is easy to make a translator that decomposes a hierarchical planning description to lower reactive rules. Moreover, such rules can be evaluated in parallel using *multi-thread* features. Therefore, it is easy to realize parallel processes to behave reactively or intensionally like subsumption architecture [1]. *Dynamical program manipulation* enables to combine program modules for each thread and to change the behavior of agent.

Compared with the previous RoboCup competition, Gaea has been improved in the following features:

- light weight multi-thread generation
- rich control mechanism of waiting event

6 Discussion

Compared with reactive production systems, the proposed architecture has an advantage in parallel process. In both systems, all behaviors are described in reactive rules. In the usual production systems like SOAR, these rules are checked uniformly in a single selection-application cycle. So, in every cycle, all conditions of operator should be checked. On the other hand, in the proposed architecture, the system can have multiple processes for each node of the operator tree. This enables to realize *lazy checking mechanism*. Usually, low-level operators should be tested almost every execution cycle, while conditions of high-level operators may not be checked so frequently, in other words, the conditions can be checked *lazily*. In the proposed architecture, users can specify the length of interval that each process take a sleep in each cycle. So, we can realize such *lazy checking mechanism*.

Compared with the subsumption architecture[1], the proposed system has an advantage in dynamics of architectures. The problem of subsumption architecture is its static structure to subsume lower behavior modules. Because of the static structure, all combination of planning should be generated before the execution, and define the subsuming relation of each modules. In Gaea, we can realize similar subsuming mechanism by overriding definitions of predicates by manipulation of program in real-time. So, we need not prepare all combination of plan before running the agents.

References

1. Rodney A. Brooks. Intelligence without representation. Technical Report Tech. Rep., MIT, 1988.
2. Hideyuki Nakashima, Itsuki Noda, and Kenichi Handa. Organic programming for complex systems. In *Proc. of Poster Session of Fifteenth International Joint Conference on Artificial Intelligence*, page 76. IJCAI, Aug. 1997.
3. Itsuki NODA. Agent programming on gaea. In *Proc. of The First International Workshop on RoboCup*, pages 147–150. IJCAI, Aug. 1997.
4. Itsuki NODA. Kappa: Agent program by gaea. In Minoru Asada, editor, *RoboCup-98 (Proc. of second RoboCup Workshop)*, pages 387–392. The RoboCup Federation, July 1998.
5. Soar manual. <http://bigfoot.eecs.umich.edu/~soar/>.
6. Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, Sep. 1997.