

## Kappa-II: Agent Programming for Flexible Team-work

### KappaII

*Itsuki Noda, Damon Otoshi, Stanley Peters*

---

---

IN: CSLI, Stanford Univ./ ETL

DO: Stanford Univ.

SP: Linguistics/CSLI, Stanford

**Abstract.** *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 (1) how to switch the context of plan (2) how to organize multiple planning. In order to attack these issues, we are proposing a programming language called Gaea and programming methodology on it. Using facilities of Gaea, we represent a strategy as a tree of situations, in which the system evaluate conditions of situations asynchronously and determines an appropriate situation dynamically. Sharing a plan and assigning of roles are also realized in the same manner of determination of situation.*

## 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

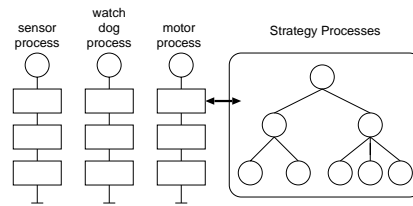


Figure 1: Architecture of Agent

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. In this article, we describe about Gaea and its programming methodology for making agents.

## 2 Gaea On C++

We are developing a programming system called Gaea [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.

Gaea was implemented on EusLisp in the version 1, and in order to improve the speed, it is now being re-implemented on C++. In addition to the original Gaea on EusLisp, C++ version has following new features:

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

## 3 Tree of Situation

### 3.1 Agent Architecture

Fig. 1 shows an architecture of Kappa-II player. In the architecture, the sensor process receives sensor information from Soccer Server, and the motor process sends commands to the server. The strategy processes are a

set of processes that determine a strategy and decide the behavior of the motor process by manipulating its program. The mechanism of the decision process is described below. The watch-dog process check low level dead lock and perform rule-related behaviors.

### 3.2 Strategy as Composition of Situation

In Kappa-II, a strategy is represented as a tree of situations (Fig. 2). The most global strategy, “play-a-match”, is represented as the most global situation “in-a-match”, and its two sub-strategies, “play-offensive” and “play-defensive”, are represented as two sub-situations, “offensive” and “defensive” respectively. In the same way, each strategy is divided into situated sub-strategy.

Each situation has a *state*. The *state* of the situation is one of ‘sleep’, ‘watch’ and ‘active’. In the ‘sleep’ state, the situation does nothing. In the ‘watch’ state, the situation checks environment and calculate its confident factor of the situation. In the ‘active’ state, the situation wakes its sub-situations up, and finds a sub-situation that have the maximum confident factor. If the confident factor is larger than a threshold, then it activates the sub-situation. Unless the situation is active, the sub-situations are in the sleep state.

We call a situation is *effective* if and only if the situation is active and no its sub-situations are active. Because of the above definitions, only a situation is *effective* in the tree. The motor process executes actions defined in the effective situation.

### 3.3 Asynchronous Condition Check

In the tree of situations, all situations in watch- and active-states evaluate their conditions and calculate confident factors asynchronously. And the evaluations are done only when relevant conditions change. This mechanism reduces the cost of evaluation to search a suitable situation in the tree for every changes of conditions.

### 3.4 Dynamic Subsumptive Behavior

The most important feature of this mechanism is that the search of a situation in the tree need not to reach the leaf node. As described in the above definition of “effective situation”, an intermediate situation can be the effective situation, and the motor process execute actions of the current effective situation. So, if a player can not get enough information of the environment, or, if a player is facing an unexpected situation, then a relatively general situation becomes effective. Therefore, we can realize an robust and flexible behavior by writing general codes of actions in general situations and the more situation-specific codes in specific situations in the similar way of *dynamic subsumption architecture* [2].

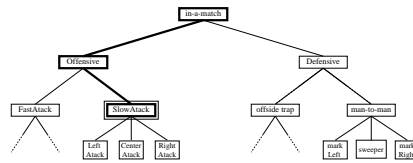


Figure 2: Example of a tree of situations

### 3.5 Communication and Shared Situation

During a match, each player informs his effective situation with its confident factor. Using this information, players can share the situation and can assign exclusive roles with each other. Because the tree of situations is shared with all teammates, each player can know which situation in the tree other teammate are thinking. So, the player can determine the most specific situation shared among players. If the situation is a plan that should be shared with teammate, then, he increases the confident factor of the situation. On the other hand, if the situation represent an exclusive role of a team, then he decreases the confident factor.

For example, if a player informs he is in ‘offside-trap’ situation, which is a typical plan that should be shared, then another player try to increase its own confident factor of ‘offside-trap’ situation. If a player informs he is in ‘one-two-pass-receiver’ situation, which is an exclusive role in ‘one-two-pass’ situation, then another player try to increase the confident factor of ‘one-two-pass’ situation and to decrease the confident factor of ‘one-two-pass-receiver’.

## 4 Discussion

Here, we like to brief discussion about comparison with related frameworks, that is, reactive productions systems and subsumption architecture.

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 [5, 6], 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 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 defini-

tions 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.