Organic Programming Language GAEA for Multi-Agents *

Hideyuki Nakashima, Itsuki Noda, and Kenichi Handa
Cooperative Architecture Project Team
ETL
Umezono 1-1-4, Tsukuba 305 Japan
nakashim@etl.go.jp, noda@etl.go.jp, handa@etl.go.jp

Abstract

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

We named the methodology "organic programming" after the flexibility of organic systems such as plants and animals. In this paper, we describe the application of an organic programming language "Gaea" to multiagent systems. One of the advantages resides in that we can program the system in a subsumptive manner.

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. In other words, we are interested in the mechanism by which large systems, such as intelligent agents, are composed from smaller units.

Our approach to multiagent systems is recursive. Each agent is programmed in a multiagent way. We call the methodology "organic programming", and it is used in constructing both single agents and multiagent systems out of those agents. In this sense, both weak and strong notions of agents are covered, as they are defined in the survey by Wooldridge and Jennings(Wooldridge & Jennings 1995).

By "organic"[1], we refer to the following characteristics of living organisms such as plants and animals.

1. *Context dependency.* Although all cells have the same program (genetic information encoded in genes), they behave differently according to their environment (including surrounding cells).
2. *Partial specification.* Genes (viewed as programs) do not contain all the information. The environment supplies a part.

*This work is supported by New Models for Software Architecture Project of MITI.

Two key concepts of our approach are situatedness and reflection. Situatedness corresponds to the former characteristic, and reflection to the latter.

Situatedness, or context dependent inference is essential for flexibility and efficiency (for more detailed discussion, see, for example, Subramanian and Woodfill(Subramanian & Woodfill 1989)). Rigid (context-free) inference based on fixed representations cannot be easily modified to accommodate changes in the environment or other agents' behavior. On the other hand, if the agent is always requested to compute on the applicability condition of rules, it is computationally inefficient. Situated representation (Nakashima & Tutiya 1991) (together with context reflection described below) kills two birds with one stone. It guarantees efficiency as long as the situation remains the same, and guarantees flexibility when the situation changes.

In order to dynamically adapt to changes in its environment, an agent has to modify itself. Such an agent is said to be reflective. The behavior of an agent is defined by a meta-level system whose behavior is in turn defined by yet another meta-level system, *ad infinitum*. The objective is to seek general mechanisms that realize a reflective agent. In organic programming, we use a variant of reflection, called context reflection, in which meta-levels and object-levels are amalgamated.

In this paper, we will describe the concept of organic programming and then the design of a specific language called Gaea. The flexible combination of modules, called "cells", allows us to program a multiagent system in a subsumptive manner.

Organic Programming

In this section, we describe our implementation of the concept "organic" into the framework of programming.

[1] **organic: 5 b (1) : constituting a whole whose parts are mutually dependent or intrinsically related : having a systematic coordination (2) : forming a complex entity in which the whole is more than the sum of the individual parts and the parts have a life and character deriving from their participation in the whole : having the character of an organism (Webster's Third New International Dictionary)**

1. *Context dependency.* Although not as extreme as in the case of living organisms where all the cells have the same genes, organic programs are context sensitive. The meaning of the program in a cell is determined by other programs in other cells and the structure of those cells.
2. *Partial specification.* The structure of cells, which functions as environment for programs, is not predefined. Like all other data structures manipulated by programs, the structure of cells are determined at run time, reflecting information gathered during run-time.

We proceed from our basic assumption about the environment: a program (of an agent) has no direct access to the real environment. It can only access the representation of the environment that is formed through I/O devices.

The representation of the environment, which is the actual[2] environment for the agent, is determined by the program. In this sense, the resulting agent is pro-active(Wooldridge & Jennings 1995) or teleo-reactive(Nilsson 1994). It not only reacts to the environment but also makes plans for changing it.

In the tradition of computer science, the representation of states used for computation is called the "context", which consists of variable bindings, stacks of return addresses, and so on. A context, however, rarely includes programs or represents knowledge. Here we extend the concept of context to cover the latter case, which we have been calling "actual environments". At this point, we can further extend the concept of reflective programs to include reflection on the environment, and call it "context reflection".

An organic program consists of the following:

- Processes: the execution of a program in a certain environment.
- Cells: the storage of program fragments. The full semantics of those programs are determined by the environment. Unlike the cells of living creatures, however, those programs may differ between cells.
- Environments for processes: The environment is formed by a collection of cells. Each cell contains information on certain aspects of the environment, and the overall environment is determined by the interaction of those fragments.

Cells provide the following two functions to programs:

1. Name to content mapping:

   The most important factor in programming methodology is in controlling the mapping between names and their contents (lambda binding of variables, module structures, etc.)

   In organic programming, the mapping from the name (of a function, a predicate, a subroutine, a cell, etc.) to the content (of the program or the cell) is controlled by cells.
2. Maintaining background conditions:

   Any program is written with some background assumptions (applicability conditions) in mind. These are implicitly hidden within the cell structure so that agents do not need to infer the conditions.

[2]Following Kimura(Kimura 1994), we use *real* to denote physical and objective entities out there in the world, and *actual* to denote the reality conceived subjectively by the agent.

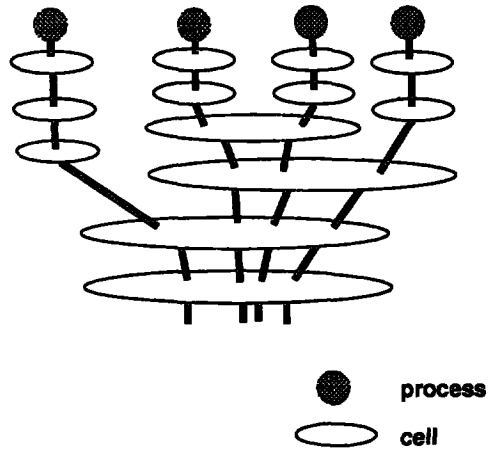

Figure 1: Environment for multi-processes

The environment for each process may adopt any structure describable by a program, but for the sake of simplicity, we will assume it is a linear list of cells. In that case, the environment for multiagent systems as a whole generally forms a tree structure, as depicted in Figure 1. Although the figure looks like a tree, the sharing of lower cells by several processes is not mandatory. Each process maintains its own cell structure as a linear list. However, for multiple processes to communicate with each other, they must share a cell for their communication field, to which they write and read data.

Programs in deeper cells in the environment are valid unless they are explicitly negated by programs in shallower cells. This mechanism can be used to inherit programs from other cells. In fact, the structure of the cells can be viewed as a dynamic version of IS-A hierarchy frequently used in knowledge representation.

When viewed from one of the processes, the outermost (furthest from the process) element of the environment is the interface to the real world in which the process is running. Other agents in the real world or in other machines are viewed as other processes (Figure 2).

The Language Features of Gaea

Gaea(Nakashima *et al.* 1996) is an instance of organic programming methodology, as Smalltalk is of object

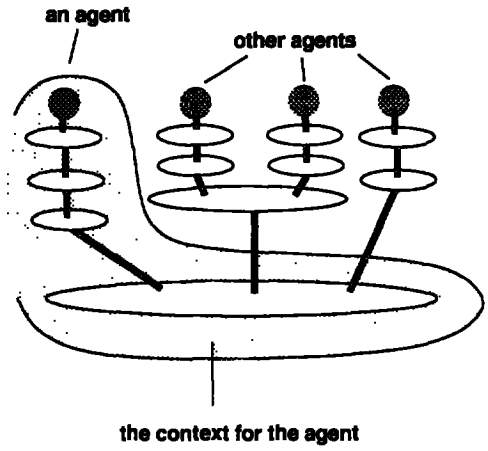

Figure 2: The environment contains other agents

oriented programming. In the case of object oriented programming, there are combinations with procedural language (Smalltalk, C++), functional programming (LOOPS), and logic programming (ESP(Chikayama 1983)). The same variety of combinations are possible in organic programming. Gaea is a logic programming language.

Infons

An "infon"(Devlin 1991) is a unit of information. In Prolog, it corresponds to a single literal (a predicate plus its arguments). In Gaea, it is a set of role-value pairs (Nakashima, Osawa, & Kinoshita 1991) represented as:

```
@time(4@hour, 30@minutes, jst@zone)
```

Although it is represented as a list, the order of the pairs is insignificant.

When some role value is fixed in an environment, it can be omitted. For example, in Japan, while `jst@zone`, i.e., Japan Standard Time, is guaranteed by the environment, it can be omitted, leaving:

```
@time(4@hour, 30@minutes)
```

The missing information ("zone" in the above case) is treated as a cell variable (with its value "jst").

The operation to turn argument roles into a background information attached to the cell is called "specialization". Its application to reasoning and dialogue are described elsewhere ((Nakashima & Tutiya 1991),(Nakashima & Harada 1996)).

For convenience in programming, and also for the sake of efficiency, Gaea accepts the conventional way of identifying arguments by their position. When an infon or a term is written without `@`, it is treated as such:

```
time(4, 30, jst)
```

Clauses

Constraints are relations among infons that always hold, or should always hold. By using these constraints, one can deduce a new infon from known infons.

The inference rules are written as a form of clause.

P : Q := R.

The above clause is read as "To prove P where Q is true, prove R", or "To execute P where Q is true, execute R".

We designed the language to minimize the possibility of unwanted global backtracking. The concept of deep backtracking does not make sense in real-time applications, while shallow backtracking is useful to select a matching rule. Once the condition Q is known to hold, other clauses are discarded.

Constraints

It is sometimes more convenient to express programs in the form of constraints. Constraints are relations among infons that always hold, or should hold. By using constraints, one can deduce a new infon from known infons.

The constraint

info1 `<=` *info2*

is used to determine whether *info1* holds by checking *info2*. This form of constraint is used to form Prolog-like backward chain reasoning.

The constraint

info1 `=>` *info2*

is used to make *info2* hold when *info1* holds.

These constraints can be defined as macros.

```
(P<=Q)::=assert(P::=Q).
(P=>Q)::=assert(action():P:=Q).
```

Forward chaining rules are activated by executing the following goal at the top-level:

```
repeat(action()).
```

Cells

A cell is described by a set of constraints. To check whether a certain infon holds in a cell,

`in(`*cell*`,` *infon*`)`

is used. However, it is recommended not to be used `in` explicitly, so that the program can work in any environment. When no explicit cell is mentioned, the current environment is searched for proper constraints.

Cell Variables

As the result of specialization, a cell may contain role values as implicit information. One of the essential ideas of "situated reasoning"(Nakashima & Tutiya 1991) is that the system need not access these values regularly. However, it is sometimes necessary to access the value for the sake of flexibility: one must check if

the system is in proper state. These implicit role values can be accessed and updated by using the following predicates:

1. `cv_read(`*cell*`,`*role*`,`*value*`)`
   accesses the *value* of the *role* in the *cell*.
2. `cv_write(`*cell*`,`*role*`,`*value*`)`
   updates the *value* of the *role* in the *cell*.

A programmer may regard these roles as global variables which are defined in the cell. They are therefore called cell variables.

Processes (Threads)

Each agent is associated with at least one process. Since processes normally share some part of their environment (cells). it is natural to implement them on shared memory architecture. Sometimes called "threads", to be distinguished from processes without shared memory, they are controlled by the following primitives.

1. `fork(`*program*`,`*process-id*`)`
   Process-id is returned as the result of fork.
2. `sleep(`*seconds*`)`
3. `kill_process(`*process-id*`)`

Environments

By forming multiple cells into a specific structure, infons can interact with each other to yield new information. We refer to these structures as *environments* for processes. For each activation of programs in a process, its environment is searched for matching definitions (clauses).

In Gaea, only a linear stack of cells is allowed. This limitation, however, can easily be discarded when another structure is required by some application in the future.

Gaea provides the following operations for cells:

1. `new_cell(`*cell*`)`
   creates a new (empty) *cell*.
2. `push(`*cell*`)`
   pushes a *cell* into the current environment.
3. `pop(`*cell*`)`
   pops the topmost cell (unified with *cell*) from the current environment.
4. `subst_cell(`*cell1*`,`*cell2*`)`
   swaps the *cell2* in the current environment with *cell1*. For an example of the use of this primitive, see the next section.
5. `environment(`*ENV*`)`
   observes the current environment and returns a list of cells to *ENV*.
6. `specialize(`*cell*`,`*role*`,`*value*`)`
   creates a new cell by fixing the value of one of the roles.
   The "specialize" operation creates a static hierarchy (lattice structure) of cells. This hierarchy is orthogonal to the hierarchy created by "push" and "pop".

   Part-of hierarchy, which is commonly used in knowledge representation, is a kind of specialization hierarchy.

Dynamic Environment Change

Let us illustrate the use of the dynamic environment change through the simple example of driving a manual-shift car. Here we are attempting to program a robot to drive the car, and particularly on the action of shifting gears. Since the robot has a lot of tasks other than shifting gears, we want to minimize the resources (memory or computation time) required for the task.

The simplest version of the rule required for shifting gears may be formalized as follows:

```
high_rpm,not(gear(high))-->shift(up)
low_rpm,not(gear(low))-->shift(down)
low_rpm,gear(low)-->clutch
```

Each line should be understood as a production rule:

LHS`-->`*RHS*

where the action in *RHS* is activated when conditions in *LHS* becomes true.

The problem in the above program is that the cost of checking the position of the shift lever is too high. The robot must sense the position either visually or manually. Moreover, when the car is in high gear and running at high speed, the condition `high_rpm` always holds. Therefore the robot must always check if `not(gear(high))` holds. One can use a technique such as a Rete algorithm(Forgy 1982) to optimize the computation, but this will not eliminate the necessity of checking the condition when the rule requires it.

Our solution is to 'memorize' the gear position [3] as the state of the robot.[4]

In Gaea, the above rules are divided into three cells: normal, high and low. In the "high" cell, there is no need to check for high_rpm because nothing can or should be done in that case. Similarly in the "low" cell, detection of low_rpm can be directly coupled with the clutch without noting the gear position. Only the "normal" cell contains rules for shifting both up and down, as the original rules did. Note that there is no need to check the gear position before shifting, because being in "normal" guarantees that the gear is neither

[3] Memorizing gear position makes sense while memorizing rpm does not. Here, we use the fact that the gear position does not change unless the agent acts on it, while the rpm does change. This knowledge is implicit in the program.

[4] We do not claim that our solution is the only one. We can easily imagine another way to memorize the condition, but we suggest that ours is one of the simplest.

in low nor high. Gear positions are checked after shifting and used to swap the "normal" cell with the proper one if necessary.

```
push(normal).
high_rpm()=>
  shift(up),
  if(gear(high), subst_cell(high,normal)).
low_rpm()=>
  shift(down),
  if(gear(low), subst_cell(low,normal)).
pop(normal).

push(high).
low_rpm()=>
  shift(down), subst_cell(normal,high).
pop(high).

push(low).
high_rpm()=>
  shift(up), subst_cell(normal,low).
low_rpm()=> clutch().
pop(low).
```

Since the gear position is obviously known just after shifting it. the robot does not expend resources to check it. And by useing the proper set of rules in the cell for the position. there is no need to check the position again.

The above program is not complete. For example, the following definitions are missing:

```
high_rpm()<=rev(X), X > 4000.
low_rpm()<=rev(X), X < 1500.
rev(X)<=cv_read(engine,rev,X).
```

We assume that these definitions are stored in a proper cell (for example "engine") and pushed somewhere in the environment. We also require the existence of other processes to compute the correct rpm from the accelerator pedal, shift position. and so on.

Differences in programming methodologies result in differences in the timing of *tests* and *actions*. In conventional programming (eg. production systems). a series of tests must be conducted to recognize the current situation, as depicted in Figure 3. These tests delay the proper action necessitated by a change in situation.

In organic programming on the other hand, some of the tests can be postponed until after taking a proper action (Figure 4). The test of shift position in the above program is one of such examples. Therefore organic programs can react to a change in situation with less computation and, after completing the proper action, can re-examine the situation.

Subsumptive Programming

Subsumption Architecture

The basic concept of subsumption architecture(Brooks 1991) can be characterized as follows:

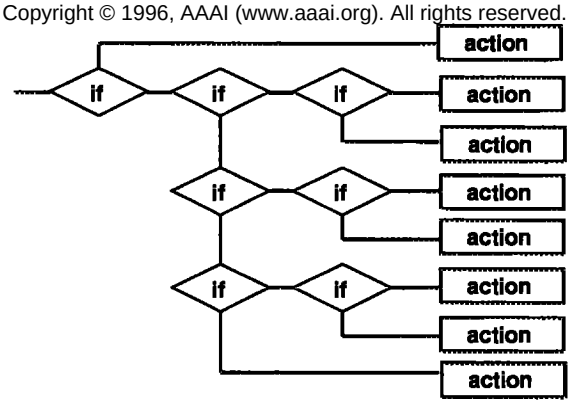

Figure 3: Conventional programming

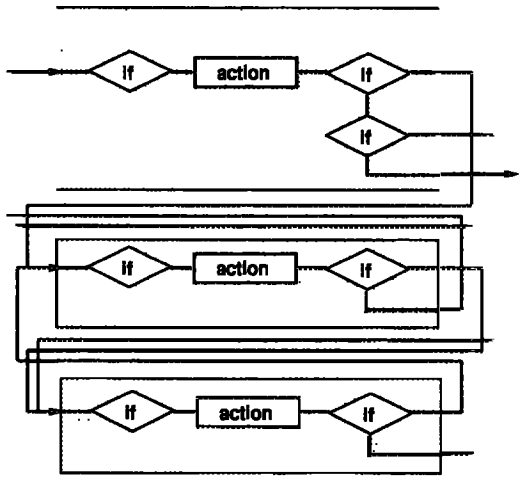

Figure 4: Organic programming

1. Each module is connected in parallel between the input and output (in contrast to conventional serial processing).
2. Modules form layers in which higher layers can subsume lower layer functions (hence the name of "subsumption").
3. Lower layers govern basic behaviors and higher layers provide sophisticated behavior.
4. The total behavior of the system can be changed by adding a new layer at the high without changing existing layers.

When a new top layer is added, it is sometimes necessary to change the behavior of existing layers accordingly. For example, when a layer "seed a target" is added. it is necessary to change the direction of "walk straight". It is ideal to perform this alternation of behavior without changing the internal structure of lower layers, as shown in Figure 5.

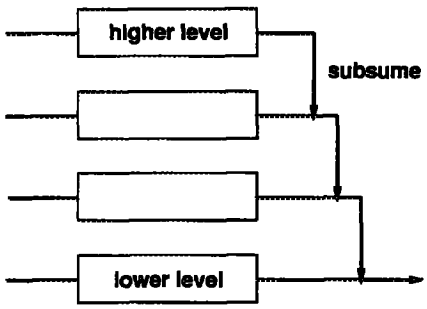

Figure 5: Concept of Subsumption Architecture

Actual implementation is something like Figure 6, in which higher layers send bias-signals to lower layers. For example, the above mentioned change of direction is achieved by applying bias to the sensory input of motor revolution in either one of two motors which produce a straight moving line when both revolutions match each other.

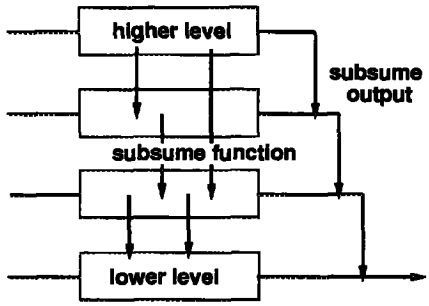

Figure 6: Implementation of Subsumption Architecture

An obvious shortcoming of this interconnection is that modularity is lost. There is yet another, more severe shortcoming in that the circuit is fixed and cannot be easily rearranged to adapt to a new situation.

Subsumptive Programming in GAEA

We can program an agent in a way similar to subsumption architecture:

1. We can program a system by functional layers.
2. Programs in higher functional layers can override programs in lower layers.

The top-level process of an agent calls programs by their names. Those programs may exist in any layer. If more than one program of the same name exists, the program in an upper layer is given precedence over those in lower layers. If we use logic programming, as in Gaea, there is a further possibility to backtrack to lower levels when a higher level program fails to perform its function.

One of the advantages of subsumptive programming is that we can easily modify the rules to accommodate a new condition. As an example, let us consider driving up or down hills. We do not want to shift up at the same point as when we drive on level ground. We can override the definition of high_rpm by **push**-ing down the following new cell over existing cells:

```
push(down_hill).
high_rpm()<=rev(X), X>5000.
pop(down_hill).
```

In the case of Brooks' subsumption, the whole system must be re-compiled to accommodate the new condition. As the result of re-compilation, the actual information flow is altered and higher layers give bias signals to lower layers. In the above example, the "high_rpm" detector in the lower layer is given a signal which is lower than the real revolution by 1000rpm.

In organic programming, the same effect is achieved by re-defining "high_rpm". The old definition still remains in the lower cell. Only the name for definition mapping is changed.

This is similar to what has been called "polymorphism" in object oriented programming. In OOP however, the mapping from a method name to its body is fixed for each object. It is polymorphic only when viewed from the sender. In organic programming, no part of the mapping is fixed. The equivalent of the object class changes during computation.

Dynamic Subsumption Architecture

The combination of subsumption architecture and dynamic environment change as depicted in Figure 4 leads to an extension of subsumption architecture, called "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 offence or defense.

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. The details can be explained by using an actual example program for a soccer player.

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.

Soccer Game as a Multiagent System

We selected soccer as our bench test because the game has many of the essential properties of multiagent systems:

1. Needless to say, soccer is played by teams of multiagents (a two-level multiagent system).

2. It is a real-time game.
3. Robustness against failure of tactics, loss of some agents, and so on, is required.
4. There is no single person giving directions during play (in contrast with American football).
5. All players must cooperate all the time (in contrast with baseball, especially with its offensive phase).
6. Verbal communication is very limited.
7. Simplification to a two-dimensional simulation does not abolish the essential properties of soccer.

A Soccer Program

In this section, we give a detailed example of dynamic subsumptive programming for soccer players in Gaea. The top level is a loop that repeatedly calls "act". It is defined in the "basic" cell as follows:

```
/* In "basic" cell */
toploop()::=repeat(act()).
```

We use cells to represent the players' modes. In each cell, an "act" is defined as performing different actions so that players can *act* properly according to their mode. The default mode (lower in the subsumption hierarchy) is "chase_ball" where the "act" is defined as running toward the ball. Note that there is no mention or avoidance of other players in default mode.

```
/* In "chase_ball" cell */
act():target(Dist,Dir):=
   turn(Dir),
   Power is Dist * 0.5,
   dash(Power).  /* dash toward the ball */

target(Dist,Dir)::=
   find_object([ball],Dist,Dir)).
```

The avoidance of opposing players is described in another cell, "avoid".

```
/* In "avoid" cell */
target(Dist,Dir)::=
   without(avoid,target(TDist,TDir)),
      /* Get the true target distance
       and direction */
   find_cpath(TDist,TDir,Dist,Dir).
      /* Search for a clear path around
       true direction */
```

We want to limit the amount of computational resources allocated to the search for a clear path. There are several ways to do so, but here we use the simplest method: try ten times and then give up.

Now we have to describe how we can subsume the "dash" process by the "avoid" cell. We need a reflective capability here.

To enter the avoidance mode, we push a new cell "avoid" into the environment of a player. This changes the subsumption hierarchy by adding a higher layer. In the "avoid" cell, "target" is re-defined to avoid the directions of opposing players. However, the re-definition must be synchronized with the "dash" action so that no unwanted combination of behavior arises. That is, we do not want to change the cell structure during the execution cycle of "act". The synchronization is established by pushing a new cell into the environment where "act" is re-defined, to manipulate the cell structure as desired before removing the cell itself together with the altered definition of "act". This new definition is activated when "act" is called the next time, that is when previous call of "act" has completed. Then one cycle is used to re-arrange the cell structure and a new "act" is called, which in turn performs the desired action.

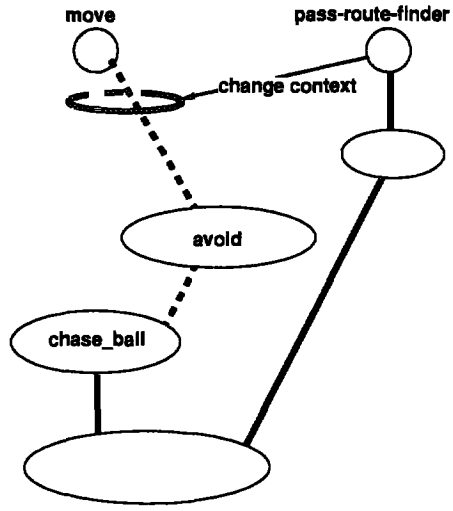

Figure 7: Interruption of another thread

The following program changes the environment of another process (Figure 7) to perform the function described above.

```
/* In "dash_monitor1" cell */
act()::=
   dash_process(PID),
   environment(ENV,PID),
      /* Get environment of dash_process.  */
   in(ENV,target(Dist,Dir)),
      /* Get a target bearing in ENV.  */
   in([avoid|ENV],not_clear(Dist,Dir)),
      /* Check if the target direction is clear.
       If not, prepare for avoidance action.  */
   push_second(push_avoid,PID),
      /* Push 'push_avoid' into the second
       cluster because the top cluster is
       used for local data management.  */
   subst_cell(dash_monitor2,dash_monitor1).
      /* change mode into dash_monitor2  */
```

```
/* In "dash_monitor2" cell */
act()::=
   dash_process(PID),
   environment(ENV,PID),
   in(ENV,without(avoid,target(Dist,Dir))),
      /* Get the target's real coordinates */
   in([avoid|ENV],clear(Dist,Dir)),
      /* If that direction is clear */
   push_second(pop_avoid,PID),
      /* Then pop_avoid, and */
   subst_cell(dash_monitor1,dash_monitor2).
      /* return to dash_monitor1 */
```

The following program pushes the "avoid" cell into the environment. To synchronize the replacement of the cell with "act", the caller only pushes "push_avoid" into the environment. When "act" is called the next time, it will actually push "avoid".

```
/* In "push_avoid" cell */
act()::=
   subst_cell(dash_monitor1,dash_monitor2).
```

The following program pops the "avoid" cell from the environment. Similar indirection as "push_avoid" is used here.

```
/* In "pop_avoid" cell */
act()::=
   remove_cell(pop_avoid),
   remove_cell(avoid).
```

The following is the initial setup:

```
/* In "init" cell */
/* Start 'dash' and 'dash_monitor'. */
start()::=
   gensym(Cell,player),
   start_dash(Cell),
   start_dash_monitor(Cell).

start_dash(Cell)::=
   without(init,
      fork(with([Cell,chase_ball,basic],
                toploop()),
           PID,dash)),
   asserta_in(Cell,dash_process(PID)).

start_dash_monitor(Cell)::=
   without(init,
      fork(with([Cell,dash_monitor1,basic],
                toploop()),
           PID,dash_monitor)),
   asserta_in(Cell,
        dash_monitor_process(PID)).
```

In our method, we assume the following:

1. Every thread uses "act" as the top-level action loop, so that re-defining "act" changes the behavior of the thread.
2. If we know a subprogram called by "act", we can redefine one of the subprograms alone. If we do not know the name, we have to redefine the whole "act".

Conclusion

We have presented the concept of organic programming and its application to multiagent systems.

We showed that organic programming is suitable for subsumption architecture. Moreover, we can change the subsumption hierarchy dynamically.

References

Brooks, R. A. 1991. Intelligence without representation. *Artificial Intelligence* 47:139–160.

Chikayama, T. 1983. Esp – extended self-contained prolog – as a preliminary kernel languages of fifth generation computers. *New Generation Computing* 1(1):11–24.

Devlin, K. 1991. *Logic and Information I: Infons and Situations.* Cambridge Univ. Press.

Forgy, C. L. 1982. RETE: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19:17–37.

Kimura, B. 1994. *KOKORO-NO BYORI-WO KANGAERU (In Japanese. Considerations on Mental Mulfunction).* Iwanami Shoten.

Nakashima, H., and Harada, Y. 1996. Situated disambiguation with properly specified representation. In van Deemter, K., and Peters. S., eds., *Semantic Ambiguity and Underpecification.* CSLI Publications. 77–98.

Nakashima, H., and Tutiya, S. 1991. Inference *in* a situation *about* situations. In *Situation Theory and its Applications, 2.* CSLI. 215–227.

Nakashima, H.; Noda, I.; Handa, K.; and Fry, J. 1996. GAEA programming manual. TR-96-11, ETL.

Nakashima, H.; Osawa, I.; and Kinoshita, Y. 1991. Inference with mental situations. TR-91-7, ETL.

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

Nilsson, N. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research* 1:139–158.

Subramanian, D., and Woodfill, J. 1989. Making situation calculus indexical. In *Proc. of The First International Conference on Principles of Knowledge Representaion and Reasoning (KR-89)*, 467–474. Morgan Kaufmann.

Wooldridge, M. J., and Jennings, N. R. 1995. Agent theories, architectures, and languages: a survey. In Wooldridge, M. J., and Jennings, N. R., eds., *Intelligent Agents.* Springer-Verlag. 1–39.