433-682
Software Agents
Term project, 2008

A Study of MindiGolog by implementing an Extreme Kitchen Management System

Xiaohui Wu Wade Huang Xi Liang

May 25, 2008

Abstract

MIndiGolog is a Golog variant suitable for distributed, cooperative execution by a multi-agent team. In MIndiGolog, agents work concurrently to perform a sequence of actions in order to accomplish some task. The domain in which agents operate is static and deterministic, meaning each discrete action produces a predetermined result. The purpose of the report is to explore and demonstrate the functionality, capability, strength and withdraws of MIndiGolog by implementing an Extreme Kitchen Management (EKM) model. The MIndiGolog enables cooperative execution of a shared high-level program by a team of autonomous agents.

Outline

1. Introduction

The purpose of the report is to explore and demonstrate the functionality, capability, strength and withdraws of MIndiGolog by implementing an Extreme Kitchen Management (EKM) model. The EKM system enables organization of a dinner with a number of people, a limited number of utensils, resources and time available within the domain. The implementation of making a dish requires completing ordering, cooking, having dinner and dish washing.

The key investigation focuses on achieving the best efficiency of accomplish a robust combination of true concurrency of cooking actions. The results show MIndiGolog does not have an efficient knowledge sharing mechanism or effective error checking and reporting system. It also shows poor performance on the solver. In order to avoid failure conditions, a checking procedure is performed before any cooking action takes place.

2. Model description

In the EKM system, the dinner is organized for a large number of people and there are a limited number of utensils and time available within the domain. In Fig.1, it illustrates the goal model for EKM system which has eight roles including Guest, Utensil Manager, Resource Manager, Waiter, Chopper, Assorter, Roaster and Dish washer.

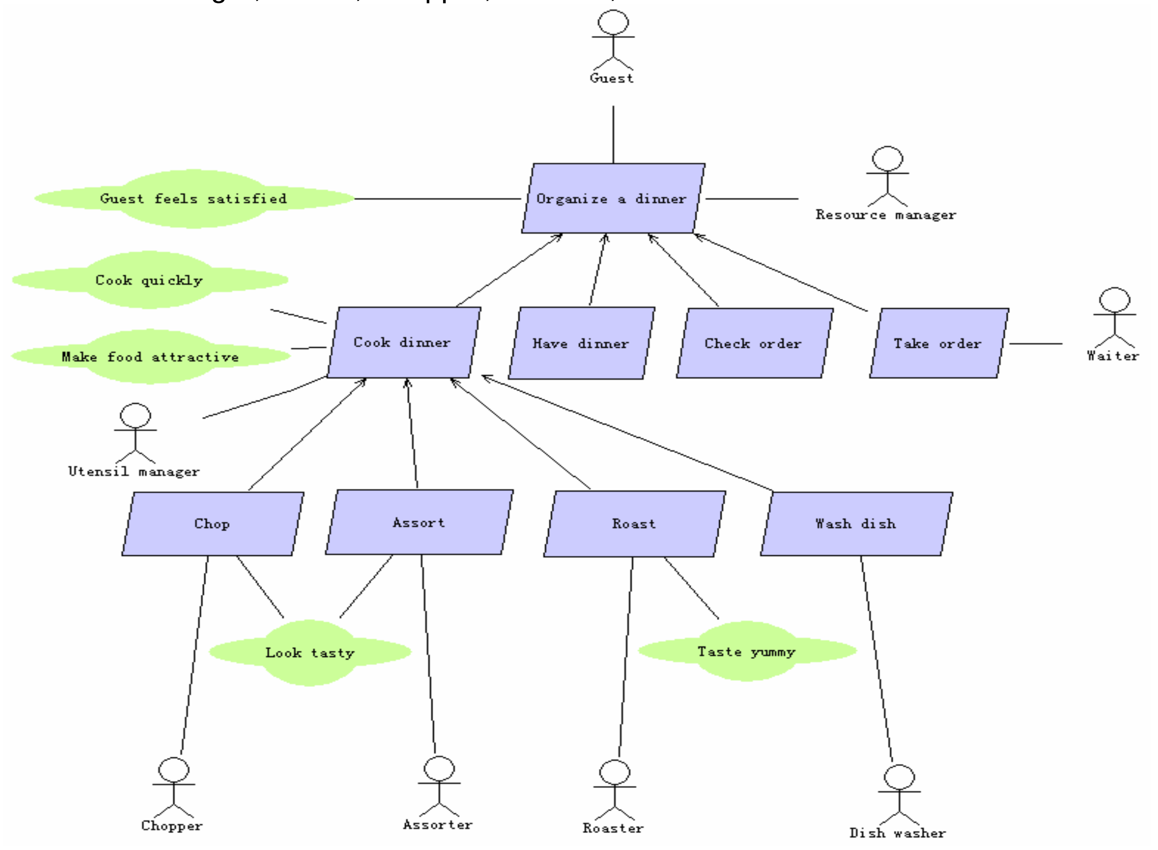


Figure 1: The goal model for an EKM system

3. Implementation and result

The EMK system is implemented in MindiGolog and many effects are spent on achieving the best efficiency of accomplish a robust combination of true concurrency of cooking actions.

A complete cooking procedure is defined within an approach of 8 steps. Each step is in charged by a corresponding agent shown as Table 1.

Step	Task	Corresponding agent
1	Acquiring an order	Waiter agent
2	Checking the order	Resource Manager
3	Provides plate(s)	Utensil Manager
4	Chopper agents acquire relevant resources	Chopper agent
5	Roaster agents acquire relevant resources	Roaster agent
6	Assorter agents acquire relevant resources	Assorter agent
7	Having dinner	People (Guest)
8	Washing dishes	Dish washer

Table 1. Cooking steps and the corresponding agent

3.1 A sample cooking procedure

Figure 2 gives a sample cooking procedure and the followings paragraphs are the explanation for each step.

1> the model starts from acquiring an order from the user: (by waiter agent). The following code shows the input format.

```
: steak.
```

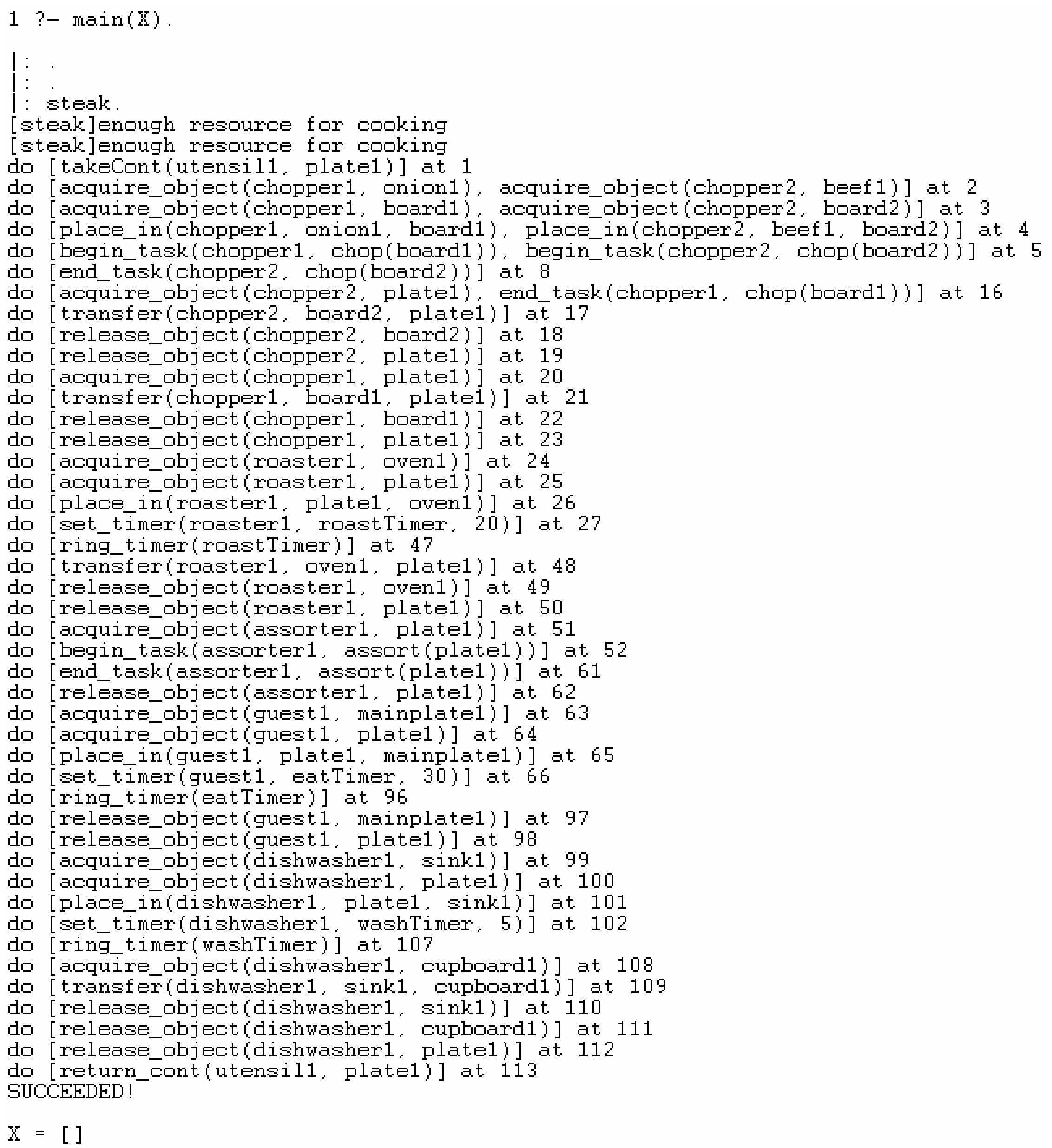
Order list could consist more than 1 dish. e.g., if users input 3 dishes in the command line, these 3 dishes will be processed concurrently.

```
:steak salad salad.
```

2> Resource Manager checks the availabilities of the required resources (e.g. beef and onion) and the capability of the agents. If the required resource is missing or lack of capability, the agents therefore recognize that the dinner can not be cooked at the end which resulting the program legally terminates. A detailed explanation of resource availability checking and evaluation will be illustrated in the following section.

3> Utensil Manager (agent type of Resource manager) provides ***plate1***. (***Plate1*** is the communication medium we introduced for providing effective communication between the agents.) In the section 3.3, a detailed explanation of how the communication is implemented will be given.

4> Chopper1 (agent type of Chopper) acquires onion1and board1, then transfers the onion to ***plate1*** after it is chopped. At the same time, chopper2 acquires onion2 and board2, and then transfers the onion to ***plate1*** after it is chopped which effects the concurrent attributes of MIndigolog.



```
1 ?- main(X).

|: .
|: .
|: steak.
[steak]enough resource for cooking
[steak]enough resource for cooking
do [takeCont(utensil1, plate1)] at 1
do [acquire_object(chopper1, onion1), acquire_object(chopper2, beef1)] at 2
do [acquire_object(chopper1, board1), acquire_object(chopper2, board2)] at 3
do [place_in(chopper1, onion1, board1), place_in(chopper2, beef1, board2)] at 4
do [begin_task(chopper1, chop(board1)), begin_task(chopper2, chop(board2))] at 5
do [end_task(chopper2, chop(board2))] at 8
do [acquire_object(chopper2, plate1), end_task(chopper1, chop(board1))] at 16
do [transfer(chopper2, board2, plate1)] at 17
do [release_object(chopper2, board2)] at 18
do [release_object(chopper2, plate1)] at 19
do [acquire_object(chopper1, plate1)] at 20
do [transfer(chopper1, board1, plate1)] at 21
do [release_object(chopper1, board1)] at 22
do [release_object(chopper1, plate1)] at 23
do [acquire_object(roaster1, oven1)] at 24
do [acquire_object(roaster1, plate1)] at 25
do [place_in(roaster1, plate1, oven1)] at 26
do [set_timer(roaster1, roastTimer, 20)] at 27
do [ring_timer(roastTimer)] at 47
do [transfer(roaster1, oven1, plate1)] at 48
do [release_object(roaster1, oven1)] at 49
do [release_object(roaster1, plate1)] at 50
do [acquire_object(assorter1, plate1)] at 51
do [begin_task(assorter1, assort(plate1))] at 52
do [end_task(assorter1, assort(plate1))] at 61
do [release_object(assorter1, plate1)] at 62
do [acquire_object(guest1, mainplate1)] at 63
do [acquire_object(guest1, plate1)] at 64
do [place_in(guest1, plate1, mainplate1)] at 65
do [set_timer(guest1, eatTimer, 30)] at 66
do [ring_timer(eatTimer)] at 96
do [release_object(guest1, mainplate1)] at 97
do [release_object(guest1, plate1)] at 98
do [acquire_object(dishwasher1, sink1)] at 99
do [acquire_object(dishwasher1, plate1)] at 100
do [place_in(dishwasher1, plate1, sink1)] at 101
do [set_timer(dishwasher1, washTimer, 5)] at 102
do [ring_timer(washTimer)] at 107
do [acquire_object(dishwasher1, cupboard1)] at 108
do [transfer(dishwasher1, sink1, cupboard1)] at 109
do [release_object(dishwasher1, sink1)] at 110
do [release_object(dishwasher1, cupboard1)] at 111
do [release_object(dishwasher1, plate1)] at 112
do [return_cont(utensil1, plate1)] at 113
SUCCEEDED!

X = []
```

Figure 2: A sample cooking procedure

5> After both chopper1 and chopper2 completing their task, roaster1 (agent type of Roaster) acquires oven1 and ***plate1***, then transfer the stuff from ***plate1*** to oven1. After steak is roasted, it will be transferred to ***plate1*** again.

6> After assorter1 (agent type of Assorter) assorts dish in ***plate1***, steak will be transfer from ***plate1*** to guest's main plate.

7> Guest1(agent type of Guest) having dinner

8> dishwasher1 (agent type of Dish washer) will wash ***plate1*** in sink, and then put the plate into cupboard.

3.2 Checking

In the program, the checking of the capability and ability of each agent and availability of each object is implemented before any action and communication among agents.

Before executing a task, the estimated task processing costs will be evaluated against a check of availability and capacity of kitchen resources. The processing cost for this check is much cheaper than the execution of a task that potentially ends in failure after running for too long, or is determined to eventually fail due to lack of available resources. For instance, in Fig. 3, there are not enough resources for the current cooking task but it still tries to run as far as it can by exhausting all the available resources. This is meaningless and time consuming to start and implement a task that is unable to complement. In a comparison, in Fig.4, it legally terminates the program by printing out the evaluation and checking message which improves the performance of the solver.

```
1 ?- main(X).

do [acquire_object(thomas, lettuce1), acquire_object(richard, tomato1),
acquire_object(harriet, egg1)] at 1
do [acquire_object(thomas, board1), acquire_object(richard, carrot1),
acquire_object(harriet, bowl2)] at 2
do [place_in(thomas, lettuce1, board1), acquire_object(richard, board2), place_in(harriet,
egg1, bowl2)] at 3
do [begin_task(thomas, chop(board1)), place_in(richard, carrot1, board2),
release_object(harriet, bowl2)] at 4
do [begin_task(richard, chop(board2)), end_task(thomas, chop(board1))] at 7
do [acquire_object(thomas, bowl1), end_task(richard, chop(board2))] at 12
do [transfer(thomas, board1, bowl1)] at 13
do [release_object(thomas, board1)] at 14
do [release_object(thomas, bowl1), acquire_object(richard, board1)] at 15
do [place_in(richard, tomato1, board1)] at 16
do [begin_task(richard, chop(board1))] at 17
do [end_task(richard, chop(board1))] at 22
do [acquire_object(richard, bowl1)] at 23
FAILED!
Remaining: conc(nil, seq(seq(seq(test(agent(_G19390)), doPlaceTypeIn(_G19390, flour,
bowl2)), pi(agt, ?agent(agt):doPlaceTypeIn(agt, sugar, bowl2)):pi(agt,
?agent(agt):acquire_object(agt, bowl2):begin_task(agt, mix(bowl2, 5)):end_task(agt,
mix(bowl2, 5)):release_object(agt, bowl2))), pi(myOven, ?obj_is_type(myOven,
oven):pi(agt, ensureHas(agt, myOven):ensureHas(agt, bowl2):place_in(agt, bowl2,
myOven):set_timer(agt, cakeTimer, 35)):ring_timer(cakeTimer):pi(agt, pi(myBoard,
?obj_is_type(myBoard, board):doTransfer(agt, myOven, myBoard))))))
```

Figure 3: Cooking without checking

```
2 ?- main(X).
|: stead salad salad steak
[stead, salad, salad, steak]not enough resource for cooking
[stead, salad, salad, steak]not enough resource for cooking
FAILED!
Remaining: control
```

With the checking procedure, the program avoids failure conditions and legally terminates before reaching any other cooking actions therefore avoids loss of efficiency.

Figure 4: Checking before the cooking action.

3.3 Communications

To achieve the cooperation among agents effective communication is required. We can view a collection of agents that work together co-operatively as a small society and for any society to function coherently we need a communication medium. In MindiGolog, the communication medium is critical for the cooperation between agents. If each agent works in isolation and is unable to interact with their peers, the software agents would be very inefficient and ineffective. A good solution to this is to use an existing agent communication mechanism; therefore we introduced *Plate* as a communication medium between different agents.

The *plate* is involved within the following listed 4 activities as the communication medium, and Fig. 5 shows how the plate is defined in the program.

```
pi(myPlate, ?obj_is_type(myPlate, plate)
        : takeCont(AgtUtensil, myPlate)
        : make(H,myPlate): eatDish(myPlate) : washDish(myPlate)
        : return_cont(AgtUtensil, myPlate)
        )
```

Figure 5: Checking before the cooking action.

The following parts demonstrate the communication mechanism by showing the MindiGolog code.

1> Assign myPlate(a plate) for each dish firstly.

```
pi(myPlate, ?obj_is_type(myPlate, plate)
        : takeCont(AgtUtensil, myPlate)
        ………..)
```

Figure 6: assign a plate for each dish

However, after an agent getting an *obj* which is the cooking resource by using acquire_object(agt,obj), this *obj* couldn't be used by other agents until it is released . In order to solve this problem, a new procedure takeCont(agt,obj) is designed to assign a plate for each dish. After a "utensil" agent gets a cooking resource *obj* by using takeCont(agt,obj), *obj* could not be used by any other “utensil” agents, but is available for other agent type, such as chopper and dishwasher.

2> myPlate is transferred in the chain of chopper→roaster→assorter→guest→dishwasher, which is implemented in MindiGolog as shown in the following clause.

```
: make(H,myPlate): eatDish(myPlate) : washDish(myPlate)
```

Figure 7: transfer chain

3> Release myPlate after it is washed.

```
: return_cont(AgtUtensil, myPlate)
```

Figure 8: release myPlate after it is washed.

4> Then myPlate is read to be assigned for another dish.

3.4 Concurrent actions

In order to achieve the best efficiency of accomplish all cooking tasks; this was done by assigning different cooking actions to the corresponding agents. e.g., cook steak involves 3 tasks: chopping, roasting and assorting. Thus in the implementation, tasks are assigned to different agent types like chopper, roaster and assorter.

1> The following code illustrates different dishes are cooked concurrently, and cooking each individual dish requires actions of ordering, cooking, having dinner and dish washing

```
proc(makeList([H|T], AgtUtensil),
        makeList(T,AgtUtensil)
        //pi(myPlate, ?obj_is_type(myPlate, plate)
       : takeCont(AgtUtensil, myPlate)
       : make(H,myPlate): eatDish(myPlate) : washDish(myPlate)
       : return_cont(AgtUtensil, myPlate)
      )
).
```

Figure 9: Differebt dishes are cooked concurrently

2> Concurrent actions in cooking. Here is an example of acquiring beef and onion concurrently in cooking steak shown in Fig. 10.

```
proc(make(steak, Dest),
     pi(agt,?prim_agent(agt,chopper): pi(obj, ?obj_is_type(obj,onion)
                : acquire_object(agt,obj)
               : doChopInto(agt,obj,Dest)
           )
       )
     //
     pi(agt,?prim_agent(agt,chopper): pi(obj, ?obj_is_type(obj,beef)
                : acquire_object(agt,obj)
                : doChopInto(agt,obj,Dest)
           )
       )
     ………..
   ).
```

Figure 10: concurrent actions in cooking

4. Discussion and conclussion

4.1 Poor performance on the solver

The interpreter takes significant time to conclude 'no' causing the MIndiGolog program seems to hang. The reason is that there are many legal executions of the program, and different ways to interleave actions from the two procedures. MIndiGolog explores each of these executions in turn, but of course they all fail when they are instructed to add an ingredient that is not available, and it eventually returns 'No' as expected.

The interpreter has a low efficiency of picking the name object because it has a chance of picking them over and over again, however the speed can be improved by make object choices in a random order.

```
  prim_obj(Obj,sugar) :-
     randomize_list([item1,item2,item3,item4],RList),
     member(Obj,RList).
```

Figure 11: pick an object in random order

It would be interesting to make a placeNumTypeIn(agt,type,num,dest) procedure that places a specific number of units of an ingredient in a bowl, which could fail early if the ingredients were not available. This is also a case where MIndiGolog's control of the search space would be useful.

Using the if-then-else statement, the clause for concurrency doesn't leave any choice points - it tries full concurrency, and if it doesn't work it goes straight ahead and does either-or concurrency.

4.2 Poor communication and knowledge sharing mechanism

We find it is a challenge to communicate between different agents in MIndiGolog for this language doesn't provide effective message transfer mechanism.

In the beginning of each action, the agent must acquire a couple of resources, and then release all resources at the end of the action. However, the consistency among multi actions is not guaranteed. The following table compares the real world and ideal design

	Real World Cases	Ideal Design
Step 1:	Chop action in steak: stuff has been chopped and placed in the plate A.	
Step 2:	End of chop action in steak: the agent releases the relation with plate A.	
Step 3:	Chop action in steak: stuff has been chopped and placed in the plate A.	Begin of roast action in steak: the agent acquires plate A which contains stuff.
Step 4:	Begin of roast action in steak	
Results:	The agent **fails** to acquire plate A which contains stuff.	The agent **succeeds** in acquiring plate A which contains stuff.

Table 2: A comparison of real world and ideal design

4.3 Poor syntax checking system and error message report system.

The MindiGolog has poor syntax checking and error message report system. As an example, the program compiles after the syntax checking but fails during the runtime as shown in Fig. 13, when the source code apparently misses a closing blanket as shown in Fig.12.

```
proc(control,
   pi(agt,?prim_agent(agt,utensil) )
```

Figure 12: A example of syntax error

```
3 ?- main(X).
FAILED!
Remaining: control
```

Figure 13: an syntax error causes failsure in the runtime

4.4 Contribution

The model legally terminates earlier by doing preliminary checking before the solver starting to find the first valid solution. By doing so, the program will not start to do the tasks without knowing whether the model is able to find a valid solution. This highly improves the ability and performance of the search solver.

5. Further work

The implementation only checks the availability of cooking resources. However, it does not check the availabilities of utensils and cooks. These functionalities will be further extended, while more tasks will run concurrently. We also considered that the checking and evaluation of resource and utensil are quite different, because the utensil is reusable which makes it very hard to predict.

There are some work can be done to improve the efficiency of cooking cooperation among different agents:

1> Multi-agent cooperation will adopt hierarchical tree structure. For example, “Head chef” is the parent node of “Waiter” and “Dish washer”, and parent node share all the knowledge of his children nodes, so “Head chef” could wash dish if there is no enough “Dish washer” in the system.

2> Assigns agents based on there current position. For example, after the stuff in the board has been chopped by agent A, system should assign a “Roaster” who is closest to A in order to save stuff transfer time.

3> Assigns agents based on their experience. System should assign the agent who has more experience firstly.

In the previous sections it has been shown that there are a couple of problems in MindiGolog, such as poor performance on the solver, bad communication and bad knowledge sharing. So the improvement of the implementation of MindiGolog itself will be part of our future work.