## 433-482 Software Agents: Project II

# A First Implementation of Extreme Kitchen Domain & Concurrency Issues

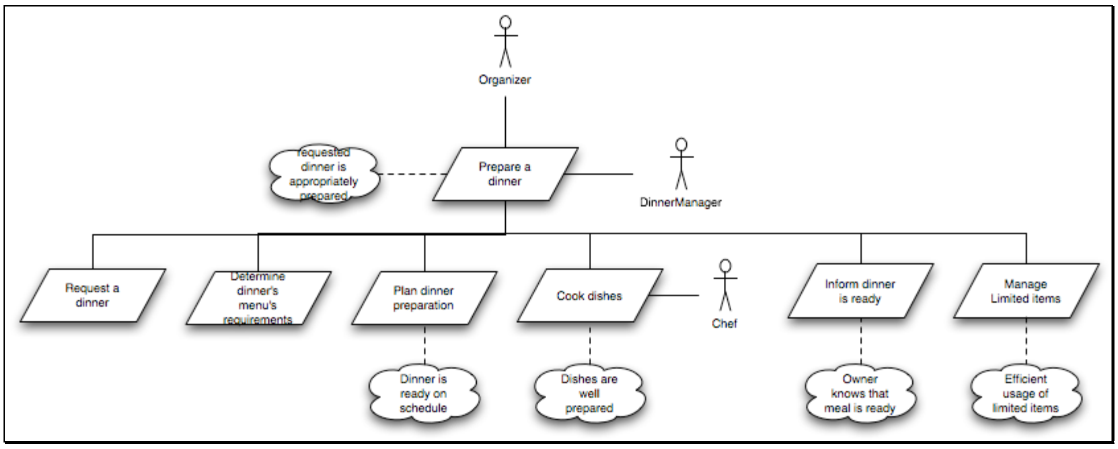
Puming Zhao

## 1. Introduction

In the Project II, I tried to make a very simple implementation of the Extreme Kitchen Domain, and found some Concurrency problems in the course of implementing. Then I chose a technique to solve this problem. In this report I will briefly describe the implementation method for the domain, and discuss about this Concurrency problem. After that, I will describe some other problems that I have met during the process.

## 2. Extreme Kitchen Domain

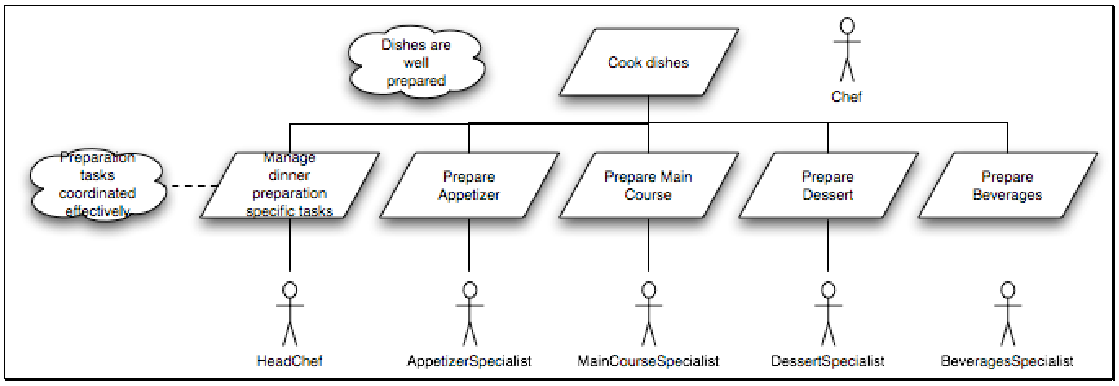
The domain to be implemented is the same as specified in project I, (I did this project with Andria Arisal). Follows is the top level goal model of the Extreme Kitchen Domain:


Organizer
requested dinner is appropriately prepared
Prepare a dinner
DinnerManager
Request a dinner
Determine dinner's menu's requirements
Plan dinner preparation
Cook dishes
Chef
Inform dinner is ready
Manage Limited items
Dinner is ready on schedule
Dishes are well prepared
Owner knows that meal is ready
Efficient usage of limited items

**Agents:** Organizer, Kitchen Manager, Chef

**Goals**

The focus in the first implementation is on 'Cook dishes' goal. The sub goal model for this goal is as follows:



The goals prior and subsequent to 'Cook dishes' are not appropriate to implement in Mindigolog, so I have chosen to ignore them in the first implementation, and defer them into further works.

The implementation decision for goals, and why:

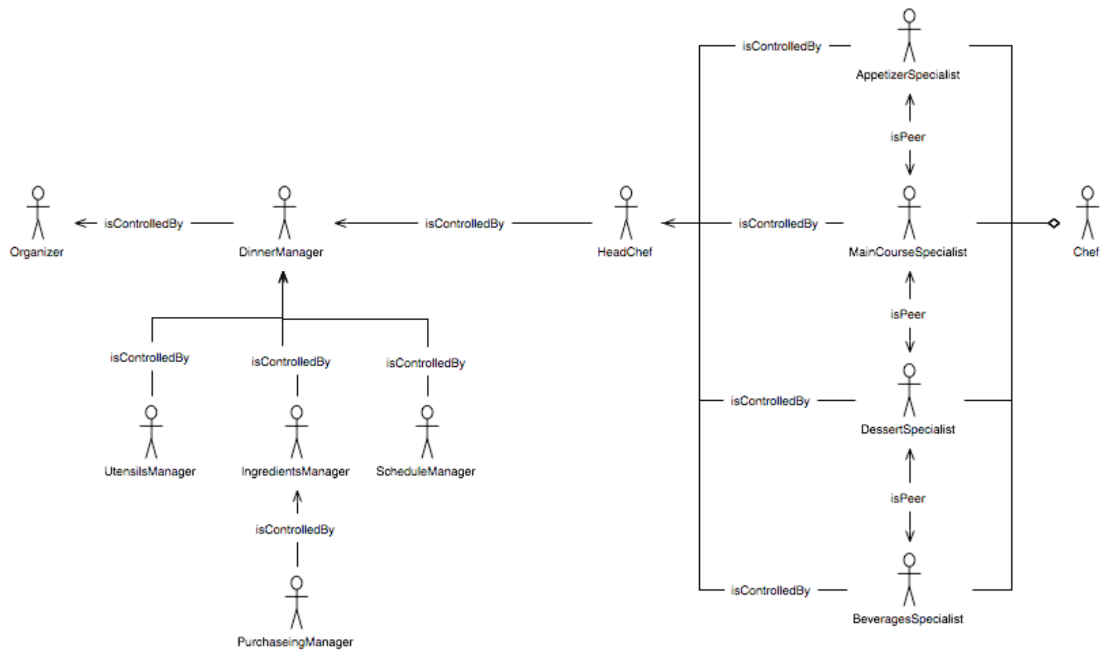
1. The first goal, ‘Request Dinner’, is more about communication between agents (Organizer and Kitchen Manager). Similar situations apply to ‘Inform Dinner is ready’.
2. ‘Determine Dinner Menu’s Requirements’. This should be implemented as a set of knowledge items: a Recipe Book of the Menu. This item is easy to implement in JADE, using an xml data file, but not as easy in Mindigolog. So I chose to implement the menu as directly written code processes. This is the same way as the ‘Make a Dinner’ domain that comes as an example of the Mindigolog. If implemented in JADE, the process should be :
   a) Agent Organizer requests the Menu, which is a list of dishes to be prepared.
   b) Agent Kitchen Manager refers to the knowledge base to determine a recipe for each dish.

But in Mindigolog, I have written the dishes into the code. Further explanation in next section.

Organizer and Kitchen Manager are ignored in the implementation.

**Roles**

The role organization model are as follows:



Roles that are **implemented** are:

Chef, HeadChef, 4 kinds of Specialists.
UtensilsManager,
IngredientsManager

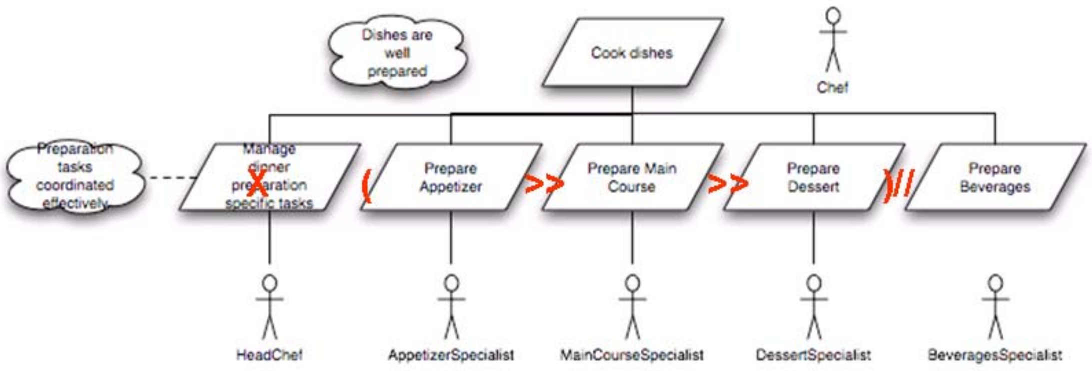
The Chef, HeadChef, and Specialists are in a hierarchy. Designing like this will make role assignment more viable. E.g. If a HeadChef is not available for a Main Course dish, then a specialist in main course will take the job; if no specialists are available, then a chef of any kind will take the job.

UtensilsManager is used when washing the utensils.
IgredientsManager is used when preparing ingredients.

## 3. The First Implementation

### 3.1. Goal: make dishes

The process to achieve make dishes are like



The first sub-goal ‘Manage dinner preparation specific tasks is ignored.
The process for cooking Appetizer, main course, and dessert are in a prioritized manner. And ‘prepare beverage’ is concurrent with three processes.

The main control procedure of ‘make a dinner’ is like:

```
proc(control,
            (
            makeAppetizer
            >> makeMainCourse
            >> makeDessert
            )
            // makeBeverage

     ).
```

To show the course of making a special kind of dishes , we can view the makeMainCourse procedure. The makeMainCourse procedure is like:

```
proc(makeMainCourse,
        makePasta
        // makePasta
        // makeLasagna
        // makeRoastChicken
).
```

And a the procedure of a particular dish is similar to makeSalad(bowl) in the example domain. Except for that not only agents are randomly selected, but also ingredients and utensils are selected by pi() function.
e.g.:

```
proc(makePasta,
      %get tomato and beef
      pi(agt, ?could_do_main_couse(agt)
                : pi(obj, ?obj_is_type(obj,tomato)
                          : acquire_object(agt,obj)
                          : doChopInto(agt,obj,dest)
                 )
        )
      //pi(agt, ?could_do_main_couse(agt)
                : pi(obj, ?obj_is_type(obj,beef)
                          : acquire_object(agt,obj)
                          : doChopInto(agt,obj,dest)
                 )
        )
      % boil tomato and beef into the sauce
      :pi(a_pan, ?obj_is_type(a_pan,pan)
            : acquire_object(agt, a_pan)
            : pi(agt, ?could_do_main_couse(agt)
                 : ensureHas(agt,dest)
                 : begin_task(agt,mix(dest,1))
                 : end_task(agt,mix(dest,1))
                 : doTransfer(agt, dest, a_pan)
                 : begin_task(agt, boil(a_pan, obj, 10))
                 : end_task(agt,boil(a_pan,obj,10))
                 : doTransfer(agt, a_pan, dest)
             )
         )
      :
      % boil the pasta
      pi(median, ?obj_is_type(median,boiler)
          : pi(dest, ?obj_is_type(dest, cups)
            : pi(agt, ?could_do_main_couse(agt)
                          : pi(obj, ?obj_is_type(obj,pasta)
                                        : acquire_object(agt,obj)
                                        : begin_task(agt,boil(median,obj,8))
                                        : end_task(agt,boil(median,obj,8))
                                        : doTransfer(agt, median, dest)
                                      )
                              )
                          )
                      )
     % mix pasta and sauce together
    : pi(agt, ?could_do_main_couse(agt)
                  : ensureHas(agt, dest)
                  : begin_task(agt,mix(dest,obj,8))
                  : end_task(agt,boil(dest,obj,8))
      )
    ).
```

---

## 3.2. The Roles

Roles for the four kinds of dishes are assigned in a similar way. E.g.:
In makePasta:

```
----------------------------------------------------
pi(agt, ?could_do_main_couse(agt)
                  : ensureHas(agt, dest)
                  : begin_task(agt,mix(dest,obj,8))
                  : end_task(agt,boil(dest,obj,8))
      )
----------------------------------------------------
```

When using pi() to select an agent, we add a test ?could_do_main_course(agt) to filter into an appropriate agent type.

And the filter test is like :

```
------------------------------------------------------
could_do_main_course(Agt) :-
    agent(Agt),
    ( is_main_course_expert(Agt) ->
          true
       ;
          is_chef(Agt)
    ).

is_main_course_expert(Agt) :-
   ( is_headchef(Agt) ->
        true
    ;
        member(Agt,[thomas, timothy])
    ).
```
-----------------------------------------------------

And This is where the hierarchy of selection takes effect.

## 4. Concurrency Issue & Solution

### 4.1. Problem Description

In my implementation of the Kitchen Domain, the procedure for making dishes is supposed to be very parallel. But in Mindigolog, concurrency optimization has lead to infeasible scalability. If there is many procedure in parallel, the program will wait for a *very* long time before the actual procedure goes. At most of the times it just hangs there and gives no result. This is a case of very bad Responsiveness.

I found the simplest way to hang the program:

makeSalad // makeCake // makeSalad

in the example domain.

### 4.2. Explanation of the problem

The problem lies in the True Concurrency nature of the Mindigolog Language. The language tries to find an optimized path of execution of concurrent procedures. It uses exhaustive search to find a global optimized path. Which is exponential (or more than that) to the number of concurrent procedures. In this way, it is not practical to find a solution for any parallel processes more than the toy program of the example domain.

The problem lies in the implementation of conc transition:

------------------------------------------------------------------

```
trans(conc(D1,D2),S,Dp,Sp) :-
        step(D1,S,Dr1,do(C1,T,S)),
        step(D2,S,Dr2,do(C2,T,S)),
        \+ ( member(A,C1), member(A,C2), actor(A,_) ),
        union(C1,C2,CT), trans(CT,S,nil,Sp),
        Dp = conc(Dr1,Dr2)
        ;
        Dp = conc(Dr1,D2), trans(D1,S,Dr1,Sp)
        ;
        Dp = conc(D1,Dr2), trans(D2,S,Dr2,Sp)
```
------------------------------------------------------------------

As shown in the first two lines, the trans(conc(…)) predicate calls step() predicate to make transition. But let's look at the step() predicate:

------------------------------------------------------------------

1. step(D,S,Dp,Sp) :-
2. Sp = do(_,_,S), **trans(D,S,Dp,Sp)**
3. ;
4. **trans(D,S,Dr,S)**,step(Dr,S,Dp,Sp).

------------------------------------------------------------------

For each step, the resolution tries to call trans() predicate again. This leads to a long chain of step/trans calls if the procedure is very much parallel.

pi() functions works in trans(D,S,Dr,S), (which is in line 4 of step() ).
And if pi() functions have many possibility to return, the search tree in every level will become very wide.

So In the case of a program with high parallel degree, we have a very deep, wide search tree, and we have to search it exhaustively. This is impractical.

## 4.3. Possible Solutions

As we can see from last section, the problem lies in the width, the depth of the search tree, and the way we search the tree. So we can solve this problem in three ways:

1. Do not make an exhaustive search
2. Limit the width of the search tree
3. Limit the depth of the search tree

We discuss the three possibilities in following sections.

### 4.3.1. Do not make an exhaustive search

Designing how we search is a very import way to solve search problems. Using heuristics is a very important method. But due to my lack of understanding in Prolog and Mindigolog, I did not find a good or even workable heuristic, nor did I get to know how to implement heuristics into Mindigolog.

So I used simpler ways to narrow the search. The method I used in the implementation, is from the report of Wirawan[1], called 'Nested If Then Elses', which is a very simple way to narrow searches: make nested tests at the beginning of each level, and fail the test otherwise. Then if any

of the test fails, backtracking will continues straight to the top of the test nest, instead of just backtracking for one step. Which will evaluate much more predicates.
The implementation needs just to alter the conc predicate:

```
-----------------------------------------------------------------
trans(conc(D1,D2),S,Dp,Sp) :-
        (step(D1,S,Dr1,do(C1,T,S))->
                (step(D2,S,Dr2,do(C2,T,S)),
                  union(C1,C2,CT),
                  \+conflicts(CT,T,S) ->
                        (\+ ( member(A,C1), member(A,C2), actor(A,_) )->
                              (trans(CT,S,nil,Sp),Dp = conc(Dr1,Dr2))
                        ;
                              fail
                        )
                ;
                        fail
                )
        ;
                fail
        )
        ;
        Dp = conc(Dr1,D2), trans(D1,S,Dr1,Sp)
        ;
        Dp = conc(D1,Dr2), trans(D2,S,Dr2,Sp)
.
-------------------------------------------------------------------
```

Compared to the standard implementation:

```
-------------------------------------------------------------------
trans(conc(D1,D2),S,Dp,Sp) :-
          step(D1,S,Dr1,do(C1,T,S)),
          step(D2,S,Dr2,do(C2,T,S)),
          \+ ( member(A,C1), member(A,C2), actor(A,_) ),
          union(C1,C2,CT), trans(CT,S,nil,Sp),
          Dp = conc(Dr1,Dr2)
          ;
          Dp = conc(Dr1,D2), trans(D1,S,Dr1,Sp)
          ;
          Dp = conc(D1,Dr2), trans(D2,S,Dr2,Sp)
-------------------------------------------------------------------
```

There are many situations that the nested test fail and quickly returns to sequencing the D1, D2.

This solution is easy to implement, quick to run, but hurts the degree of concurrency.

## 4.3.2. Limit the width of the search tree

To limit the width of the search tree, we need to narrow the possible executions in each step.
One intuitive way is to narrow the pi() selection.
Suggestions for narrowing the pi() selection:

1. Make the role assignment dynamic.
   This is a different topic of technique, and could be very complicated. So I will not go further into it. Refer to the report of kgtan [2].
2. Make the utensils assignment dynamic.
   Similar to dynamic role assignment, we can use a heuristic to sort the utensils lists, so that each time we select a utensil, the program will just return the most usable (one scheme: earliest cleaned) utensil.
3. Make the pi selection of ingredients into getNextIngredient.
   Because ingredients are not reusable, we can delete the item that has been used from the list. But we then have to make a set of dynamic lists (one for each kind of ingredient), that shortens with transitions. The way to implement it is to change the semantic of situation transitions, which is very difficult to implement -- at least to me. (Sorry for my poor Mindigolog skills). But as we can see from further discussions, many solutions tend to need a new information block in situations. Similar to T block in each situation, which stands for current time in the situation. So if we have one way to add that easily, the implementation will ease.

I don't have enough energy to take them into implementation. Just talk about some suggestions.

## 4.3.3. .Limit the depth of the search tree

The intuitive way to implement this is to give the tests a limit parameter, as in the report of Wirawan [1].
The solution Wirawan suggests are like this:

```
--------------------------------------------------------------------------------
is_pianist(Name,Searched,WidthSearchLimit,IsPianist) :-
    check_pianist(Name,Searched,Option,Status),
    (Status = success,IsPianist = yes
    ;
    Status = failure,
    WidthSearchLimit1 is WidthSearchLimit - 1,
    (WidthSearchLimit1 > 0,
    append(Searched,[Option],Searched1),
    is_pianist(Name,Searched1,WidthSearchLimit1,IsPianist)
    ;
    IsPianist = no))
.
```

```
check_pianist(Name,Searched,Option,Status) :-
        pianist(A),\+member(A,Searched),
        (A = Name, Status = success
        ;
        Option = A,
        Status = failure)
--------------------------------------------------------------------------------
```

The test is is_pianist. And he added two parameters to the test: ‘Searched’,’WidthSearchLimit’, to make the limit magic work. The solution is intuitive, but then you have to add parameters to every test predicate in the domain, which is not a trivial task. I have not used this method.

## 5. Further work

1. Due to my bad programming skills in Mindigolog, I have not come to a good result. The first part of further work is to make a run and give an output.
2. I’d like to change pi() semantic for ingredients.
3. I came to the idea of Degree of Concurrency (DoC), I came to a first rating equation: DoC could be valued as the average of concurrent procedures per situation. And the method suggestion to implement it:
   a) For each situation, find the number of agents that is busy.
   b) Make a statistic on this number.
      i. Change the semantic of the do(D,T,S) predicate in S.
      ii. do(D,T,Nbusy,S), where Nbusy stands for how many agents are busy.
      iii. Compute the average DoC using a function similar to show_action_history(S).

   So the work here is to implement it.

## 6. References

[1] Edwin Wirawan, Optimizing Choice between True Concurrency and Interleaved Concurrency, Project Reports for 433-481, 2007.

[2] Jarrod Sibbison (jarrods), Kong Guan Tan (kgtan), Dynamic Skill-based Role Assignment, Project Reports for 433-481, 2007.