

# AI Sous-Chef Project Report

---

**Team ID:** 47

**Team Members:**

1. Ali Khaled Koheil - 55-24778 - T-14
2. Khaled Ashmawy - 55-25730 - T-23
3. Youssef Nasser - 55-7165 - T-27

**Date:** December 5th, 2025

## Code Implementation with Detailed Comments

### 1. Knowledge Base Inclusion

```
: - include('KB1.pl').  
% : - include('KB2.pl').
```

**Comment:** Loads the ordering constraints (**above/2** facts) from the knowledge base. Can be switched to KB2.pl for testing different constraint sets.

---

### 2. SUCCESSOR STATE AXIOM: **stacked/2**

This is the **only fluent** and **only successor state axiom** in our implementation.

```
stacked(_, s0) :- fail.  
  
stacked(I, result(A, _)) :-  
    A = stack(I), !.  
  
stacked(I, result(_, S)) :-  
    stacked(I, S).
```

#### ARGUMENTS:

- **I** (or **Ingredient**): An atom representing the ingredient
- **S** (or **result(A, S)** or **result(\_, S)**): A situation term representing the world state

#### SUCCESSOR STATE AXIOM FORM:

##### 1. **Base Case:** **stacked(****\_****, s0) :- fail**

- No ingredients are stacked in the initial situation s0
- Establishes the starting state of the world

## 2. Positive Effect Axiom ( $\gamma^*$ ): `A = stack(I), !`

- The ingredient becomes stacked if the action performed is `stack(I)`
- This is how actions change the world state
- The cut (!) prevents further checking once we find the action that stacked this ingredient

## 3. Frame Axiom (Persistence): `stacked(I, S)`

- The ingredient remains stacked if it was already stacked in the previous situation
- This solves the frame problem - things persist unless explicitly changed

---

## 3. Helper Predicate: `count_stacks/3`

```
count_stacks(_, s0, 0).

count_stacks(Ingredient, result(stack(Ingredient), S), Count) :-  
    count_stacks(Ingredient, S, PrevCount),  
    Count is PrevCount + 1.

count_stacks(Ingredient, result(stack(Other), S), Count) :-  
    Other \= Ingredient,  
    count_stacks(Ingredient, S, Count).
```

### ARGUMENTS:

- `Ingredient`: The ingredient to count
- `S`: The situation to examine
- `Count`: Output - number of times the ingredient was stacked

**Comment:** Recursively traverses the situation structure backwards from the final state to `s0`. When it finds the ingredient being stacked, it increments the count. When it finds a different ingredient, it maintains the count.

—  
PROF

---

## 4. Helper Predicate: `one_of_each_ingredient/1`

```
one_of_each_ingredient(S) :-  
    count_stacks(bottom_bun, S, 1),  
    count_stacks(patty, S, 1),  
    count_stacks(lettuce, S, 1),  
    count_stacks(cheese, S, 1),  
    count_stacks(pickles, S, 1),  
    count_stacks(tomato, S, 1),  
    count_stacks(top_bun, S, 1).
```

### ARGUMENT:

- **S**: The situation to validate

**Comment:** Ensures each ingredient appears exactly once by verifying `count_stacks/3` returns 1 for each ingredient. Prevents duplicate ingredients.

---

## 5. Helper Predicate: `first_action/2`

```
first_action(result(Action, s0), Action).

first_action(result(_, S), Action) :-  
    first_action(S, Action).
```

### ARGUMENTS:

- **S**: The situation
- **Action**: Output - the first action performed from s0

**Comment:** Recursively searches backwards through the situation until it reaches s0, then returns the action immediately after s0. Used to verify `bottom_bun` is the first ingredient stacked.

---

## 6. Helper Predicate: `last_action/2`

```
last_action(result(Action, _), Action).
```

### ARGUMENTS:

- **S**: The situation (must be in form `result(Action, _)`)
- **Action**: Output - the last (most recent) action

PROF

**Comment:** Simply returns the outermost action from the situation structure. No recursion needed since the outermost action is the most recent. Used to verify `top_bun` is the last ingredient stacked.

---

## 7. Helper Predicate: `stacked_before/3`

```
stacked_before(I1, I2, result(stack(I2), S)) :-  
    stacked(I1, S), !.

stacked_before(I1, I2, result(_, S)) :-  
    stacked_before(I1, I2, S).
```

### ARGUMENTS:

- **I1**: The ingredient that should be below (stacked earlier)

- **I2**: The ingredient that should be above (stacked later)
- **S**: The situation to examine

**Comment:** Traverses the situation backwards. When it finds **I2** being stacked, it checks if **I1** was already stacked in the previous situation. The cut (!) stops the search once found for efficiency. This implements temporal ordering - if **I1** was stacked before **I2**, then **I1** is physically below **I2** in the burger.

---

## 8. Helper Predicate: **poss/2**

```
poss(stack(X), S) :-  
  \+ stacked(X, S).
```

### ARGUMENTS:

- **stack(X)**: The stacking action
- **S**: The situation

**Comment:** Checks if it's possible to stack ingredient **X** in situation **S**. An ingredient can only be stacked if it hasn't been stacked already.

---

## 9. Helper Predicate: **constraints\_satisfied/1**

```
constraints_satisfied(S) :-  
  forall(  
    above(Upper, Lower),  
    (stacked(Upper, S), stacked(Lower, S), stacked_before(Lower,  
Upper, S))  
  ).
```

PROF

### ARGUMENT:

- **S**: The situation to validate

**Comment:** Validates all ordering constraints from the knowledge base. Uses **forall/2** to check every **above(Upper, Lower)** fact. For each constraint: (1) both ingredients must be stacked, and (2) **Lower** must have been stacked before **Upper** (i.e., **Lower** is below **Upper** in the burger). If any constraint fails, the entire predicate fails.

---

## 10. Goal Predicate: **burgerReady\_goal/1**

```
burgerReady_goal(S) :-  
  one_of_each_ingredient(S),  
  first_action(S, stack(bottom_bun)),
```

```
last_action(S, stack(top_bun)),  
constraints_satisfied(S).
```

#### ARGUMENT:

- **S**: A situation to validate as a complete burger

**Comment:** Defines the goal state for IDS. All four criteria must be satisfied:

1. Each ingredient appears exactly once (no duplicates)
2. Bottom bun is the first action
3. Top bun is the last action
4. All **above/2** constraints from KB are satisfied

This is the validation function that determines if a candidate solution is valid.

---

#### 11. Situation Generator: **build\_situation/1**

```
build_situation(s0).  
  
build_situation(result(stack(I), S)) :-  
    build_situation(S),  
    member(I, [bottom_bun, patty, lettuce, cheese, pickles, tomato,  
top_bun]),  
    poss(stack(I), S).
```

#### ARGUMENT:

- **S**: A situation built from s0 through stacking actions

**Comment:** Generates candidate situations for IDS to explore. Recursively builds situations by choosing ingredients and stacking them. **member/2** provides backtracking to try different ingredients. The **poss(stack(**I**), **S**)** check ensures no ingredient is stacked twice during generation.

---

#### 12. MAIN PREDICATE: **burgerReady/1**

```
burgerReady(S) :-  
    ids_burgerReady(S, 1).
```

#### ARGUMENT:

- **S**: Either unbound to find solutions or bound to validate a burger configuration

**Comment:** Main entry point for the system. Uses Iterative Deepening Search. Calls IDS with an initial depth limit of 1.

---

## 13. IDS IMPLEMENTATION: `ids_burgerReady/2`

```
ids_burgerReady(S, Limit) :-  
    call_with_depth_limit(  
        build_situation(S), burgerReady_goal(S)),  
        Limit,  
        Result  
    ),  
    ( number(Result) -> true  
    ; Result = depth_limit_exceeded,  
        NewLimit is Limit + 1,  
        ids_burgerReady(S, NewLimit)  
    ).
```

### ARGUMENTS:

- `S`: The situation to find or validate
- `Limit`: Current depth limit for the search

**Comment:** Implements Iterative Deepening Search using `call_with_depth_limit/3`. The algorithm:

1. Attempts to prove `(build_situation(S), burgerReady_goal(S))` within the current depth limit
2. If `Result` is a number: solution found at that depth, return success
3. If `Result` is `depth_limit_exceeded`: increment limit by 1 and recursively try again

This ensures completeness - if a solution exists, IDS will eventually find it by systematically increasing the depth.

---