

# Informatics 2D Coursework 2: Situation Calculus and Planning

Mihai Dobre and Alex Lascarides

Deadline: 4pm on Monday the 21st March 2016

## Introduction

This assignment is about the Situation Calculus and Planning. It consists of three parts. Part 1 (worth 45%) is a written exercise that requires you to formalise a planning problem, using **Situation Calculus** to represent the world. Part 2 (worth 20%) is about implementing the model and verifying its correctness using a planner based on the **Golog syntax**. And part 3 (worth 35%) is about extending the model as well as its implementation in order to deal with additional aspects of the environment.

**Important:** Please attend the lecture on Friday 4th March at 3:10pm in Teviot Lecture Theatre Doorway 5, the Medical School, where Mihai Dobre will present this assignment together with hints and tips on how to complete it and how to program in Prolog. Please also attend the lab session on Friday 11am-1pm Forest Hill Room 1.B32 in weeks 8 and 9 (11th/18th March).

The files that you need are available from:

<http://www.inf.ed.ac.uk/teaching/courses/inf2d/coursework/Inf2D-Coursework2.tar.gz>

Please put your matriculation number in all files you submit (except `planner.pl` and `plan.sh`, which you should not edit). Note that you do not need to put your name in the files. The deadline for submission is **4pm on Monday the 21st of March 2016**.

The paper write-up will be done in the `answer.txt` file and for the implementation part you will need to copy and edit the `*-template.pl` files available in the coursework archive.

To submit your assignment, create a new archive file with the files: `answer.txt`, `domain-*.pl`, and `instance-*.pl` in it. You can do this using the following command in a DICE machine:

```
tar cvzf Inf2d-Assignment2-s<matriculation>.tar.gz <assignment-dir>
```

where `<matriculation>` is your matriculation number (e.g. 0978621), and `<assignment-dir>` is the directory in which you assignment files are stored.

**Submit** this archive file using the command:

```
submit inf2d 2 Inf2d-Assignment2-s<matriculation>.tar.gz
```

**Good Scholarly Practice:** Please remember the University requirement as regards all assessed work for credit. Details about this can be found at:

<http://www.ed.ac.uk/schools-departments/academic-services/students/undergraduate/discipline/academic-misconduct>

and at:

<http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct>

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

In this document, you will find two icons:



means that the following sentence describes how to answer the question/task.



means that the following sentence describes the question/task that should be answered/done.

## Part 1: Modelling the Planner (45%)

The first part of this assignment requires you to develop a model for a planning problem. You are initially asked to formalise the domain using the Situation Calculus, based on the same formalism as described in Russell & Norvig 3rd Edition, section 10.4.2. You will need to describe the state of the world using predicates and fluents for specifying the initial and goal states, and also some basic actions that affect the world. The actions will be available to the planner. You will then use the axioms you defined to infer a plan for an instance of the problem.

In the assignment archive, you will find a text file called `answer.txt`. Please fill in your answer in the relevant sections of this template.

### Problem Description

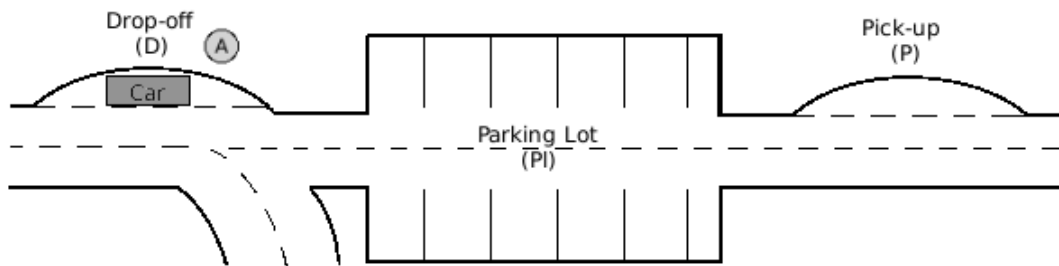


Figure 1: Map for the parking problem.

In this task, you need to model an automated parking agent for a restaurant which can plan and execute a sequence of actions to accomplish some goals. Figure 1 shows a map with the fields of the restaurant divided into 3 areas: the drop-off, the pick-up and the parking lot. For the sake of simplicity, we assume that the map is discrete. The drop-off (D) is connected to the parking lot (Pl), which is connected to the pick-up area (P) as shown. Customers leave their cars with the agent in the drop-off area. Once they have finished their meal, they wait for their car in the pick-up area. The agent (A) has the ability to move between two connected areas, drive the cars between areas, park the cars in the parking lot and deliver the car in the pick-up area. For now, you don't have to consider if there are any available spaces in the Parking Lot before parking a car.

### Knowledge Base (10%)

The first step in the creation of a model is the design of the knowledge base, i.e. the structures that will hold information about the environment that the planner can access when choosing an action as part of its plan. The initial model should include information about the three areas and their connections, the locations of the agent and the car as shown in the figure. It should also reflect the state of the car.

You should define a minimal set of predicates that can encode every state of the problem. Some of them will be *atemporal predicates*, which do not change as time progresses. Some will be *fluents*, i.e. predicates whose value depends on the current *situation*.



Fill in the first section in the `answers.txt` file:

<sup>0</sup>You may find a similar content in section 10.3 of Russell & Norvig 2nd Edition book.

- ⇒ [1.1 (2%)] Define the predicates you will use to describe the map in your knowledge-base (KB); specify which locations are connected.
- ⇒ [1.2 (2%)] Explain how you will keep track of the area in which the agent and the cars are at any particular moment.
- ⇒ [1.3 (2%)] Describe what can be used to express the state of the car: **parked** in the parking lot or **delivered** to the customers waiting in the pick-up area. Note that cars dropped-off by customers are not parked and not delivered.
- ⇒ [1.4 (4%)] Using the symbols you just defined, describe the *initial state* of the problem, as it is depicted in Figure 1.
- 👉 You can use **and** instead of  $\wedge$ .

## Actions (15%)

In the `answers.txt` file, formalize the following actions in terms of *possibility axioms* and *effect axioms*:

- ⇒ [1.5 (2%)] The agent can **move** between any two connected areas;
- ⇒ [1.6 (2%)] The agent can **park** a car if they are both in the parking lot; The effect of this action is that the car is parked;
- ⇒ [1.7 (3%)] The agent can **drive** a car between any two connected areas if it is in the same initial location as the car. In this case both the location of the car and the agent is changed and the car is no longer parked;
- ⇒ [1.8 (2%)] The agent can **deliver** the car, if they are both in the pick-up area; The effect is the car is delivered (and driven away).

👉 You can omit universal quantifiers, and use **and**, **or**, **not**,  $\Rightarrow$ , and  $\Leftrightarrow$  instead of  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\Rightarrow$ , and  $\Leftrightarrow$ .

Effect axioms alone are not sufficient: they describe how the new situation has been affected by the action executed, but they do not update information unrelated to the specific action, which may (or may not, if not updated) remain the same.

⇒ [1.9 (2%)] Briefly explain this well known problem which, in the Situation Calculus, has been solved by introducing *successor-state axioms*. How does another formalism for planning, i.e. STRIPS, solve the issue?

A successor-state axiom defines the state of a fluent, based on its value in the previous situation and the action that has just been executed. At a high level, it formalizes the idea that a fluent will be true if the most recent action makes it true, or if it was already true in the previous situation, and the most recent action has not changed its state.

- ⇒ [1.10 (4%)] Write the successor-state axioms for the fluents in your model.

## Resolution and Refutation (20%)

Using the definitions developed in this first part of the assignment, it should be possible to prove the feasibility of a plan or, given a goal, to find a plan that achieves it. One way of doing this is a *proof by refutation*. You will have to convert **all** the possibility and successor-state axioms to Conjunctive Normal Form (CNF), and then try to prove the *negation* of the goal. If there is a contradiction in the theory, the resolution process will find it, thus showing that the goal is indeed true. The sequence of axioms used shows which actions are needed to reach the goal.

Assume that the initial state is as depicted in Figure 1.

- ⇒ [1.11 (20%)] Prove by refutation that the following goal can be reached:

$$\exists s. \text{Parked}(\text{Car}, s)$$

(i.e., there is a situation in which the car is parked). Show the plan found.

## Part 2: Implementation (20%)

The second part of the assignment is centred on the implementation of the model that you developed in Part 1. Once we have translated the axioms into rules that a planner can understand, we can work on more complex instances of the planner. The conversion is fairly straightforward. It consists mostly of a translation of logical symbols into ASCII characters, as we shall see in this section.

### A planner and the Golog syntax for Situation Calculus

Golog (alGOrithms in LOGic) is a macro language which extends the Situation Calculus to include constructs usually found in imperative language, such as conditional branches, loops, and procedure. This assignment simply uses the syntax proposed by Golog to represent the axioms of the Situation Calculus. Since the Golog interpreter is written in Prolog, the syntax is similar, but no knowledge of Prolog is required for this assignment.

In the coursework archive file you will find a bash script named `plan.sh`. Running this script without parameters will print a short explanation of its command line. The script calls the planner (`planner.pl`), loads the chosen problem file, and runs the planning procedure.

In the same folder, you will find two examples implementing a simple blocks world, `sample-blocks.pl` and `sample-blocks-domain.pl`. You can run the planner for this example using the command:

```
./plan.sh sample-blocks.pl
```

Inside the examples you will find additional documentation on the format.

To show the differences and similarities between situation calculus and the language read by the planner, we can compare two versions of the blocks world example (simplified). What follows are the possibility and successor-state axioms for the *move* action.

Following the conventions on R & N, we have predicates starting with an uppercase letter and variables in lower-case, quantified:

$$\forall x, y, s. Clear(x, s) \wedge Clear(y, s) \Rightarrow Poss(Move(x, y), s) \quad (1)$$

In Golog:

- the opposite is true: predicates begin with lower-case letters and variables with capitals
- quantifiers are dropped
- logical connectivities change: the implication symbol is `:-` and it *points from right to left*. In other words, you should read `p :- q` as *if q then p*
- a comma `,` represents a conjunction
- disjunctions are marked by semi-colons `;`
- the end of a rule is marked by a dot `.`

For example, the above axiom 1 is written in Golog as follows:

```
poss(move(What, Where), S) :- clear(What, S), clear(Where, S).
```

In logic, a successor-state axiom is guarded by the predicate that verifies if the action is possible.

$$Poss(a, s) \Rightarrow on(x, y, Result(a, s)) \Leftrightarrow a = Move(x, y) \vee (on(x, y, s) \wedge a \neq Move(y, z)) \quad (2)$$

This is done automatically by the planner or Golog interpreter, and can be dropped. Moreover, we are interested in the **planning task**, so we only keep one direction of the *iff* in the formula above, the  $\Leftarrow$ . The resulting Golog successor-state axiom for axiom 2 is:


```


on(Block, Support, result(A, S)) :-
    A = move(Block, Support);
    on(Block, Support, S), not(A = move(Block, _)).

```

where the semicolon ; is a disjunction ( $\vee$ ), and the underscore  $\_$  is an anonymous variable that unifies with anything.


## Task 2.1: Translate axioms (10%)

After reading the sample files and the included documentation,  make a copy of the `domain-template.pl` file and rename it as `domain-task21.pl`.

 [2.1 (10%)] Translate the axioms of your model and save them in this file.

## Simple experiments

The following three exercises are minimal experiments to learn the language accepted by the planner, and for you to test the correctness of the model. Each task has at least one solution, and none of the plans exceed 15 actions in length: if the planner fails to find a plan, there might be something wrong in the model.

 For each task, make a new copy of the file `instance-template.pl` and rename it `instance-task<#>.pl` (replace `<#>` with task ID). Any comment or description can go inside the `.pl` source file.

## Task 2.2: Hello Golog (ID: 22)

In this instance of the problem:

- The agent is in the pick-up area;
- The **goal** is for the agent to be in the drop-off area.

 [2.2 (3%)] Implement and test.

## Task 2.3: Multiple Goals (ID: 23)

In this instance of the problem:


- The agent and the car are in the parking lot;
- The car is not parked;
- The **goal** is that the car is parked and the agent is in the drop-off area.

 [2.3 (3%)] Implement and test.

## Task 2.4: A More Complex Problem (ID: 24)

In this instance of the problem:



- A customer is waiting to pick up carA;
- carA is parked;
- Another customer just dropped-off carB;
- The agent is in the drop-off area;
- The **goal** is that carB is parked, carA is delivered and the agent is in the drop-off area.

 [2.4 (4%)] Implement and test.

Ideally, the planner should take less than a minute to come up with a plan, and certainly no more than 3 minutes.

## Part 3: Extending the Domain (35%)


In this section you will extend the original problem to include new actions and goals. Only the Golog implementation of the axioms are required, but:

-  Make sure the code is properly commented when defining new predicates or actions;
-  For each task, make a new copy of the file `domain-template.pl`, and rename it `domain-task<#>.pl` (replace `<#>` with the task's ID). Any comment or description should go inside the `.pl` source file.

To implement these extensions you can add new actions and predicates, as well as modify your set of axioms.

### Task 3.1: Multi-Storey Parking (ID: 31)


The owner decided to extend the restaurant and turn the parking lot into a multi-storey one. In the meantime, the parking facility could still be used, but any car parked gets dirty due to the work taking place upstairs. To make it up to the clients, the owner introduced a valet option free of charge. Therefore, the agent must clean the parked car before driving it to the pick-up area.

 [3.1 (10%)] Add a new action for cleaning a car, adapt the action for delivering a car so it can be performed only if the car is cleaned, and adapt the action of parking a car so an effect of this action is that the car is dirty. Test your code on the same problem as the one presented in Task 2.4. with the addition that `carA` is dirty.

### Task 3.2: Attempted robbery (ID: 32)

Until now, the agent has carried all the keys at all times (so we didn't need to model the keys). However, security footage has shown someone tried to steal the car keys from the agent. To discourage any future attempts, the owner added a utility box (U) with a very strong lock. Only the keys of the parked cars can be **stored** in the utility box and the agent is not allowed to carry more than one set of keys at any one time. As expected, the agent can't drive a car if it doesn't have the corresponding keys in its possession. When a car is delivered, the keys are left in ignition so the agent is no longer holding them.

**Hint:** You should introduce two new actions to handle the keys: **grab** and **store**, as well as a fluent to show that the agent is **holding** the keys. Update the previous actions for handling the car with the new constraints as well as make sure that the agent uses the correct set of keys with each car via a new predicate.

 [3.2 (15%)] test your code on the following problem:

- CarA is parked, dirty and keysA are stored in the parking lot;
- Customer A wants to collect his cars and keys;
- Another customer just dropped off carB;
- The agent is holding keysB and is in the drop-off area;
- The **goal** is that keysB are stored in the parking lot and carA is delivered.

### Task 3.3: Limited parking space (ID: 33)

The constructors require even more space to deposit the materials. Therefore, a large portion of the parking space was sacrificed and the parking lot now has a maximum of 4 spaces. Parking a car occupies a space, while driving a car away frees a space. A car cannot be parked if all the spaces are occupied but can still be in the parking lot.

👉 [3.3 (10%)] In the initial state of the following tasks, the agent is in the drop-off area and is not holding any keys as these are either stored if the car is parked or are in the same area as the dropped-off car. As before, already parked cars are dirty. Run the following experiments:

- The parking lot has 2 empty spaces and two new customers drop-off their cars (carA and carB with the corresponding sets of keys); The **goal** is that both sets of keys are stored;
- The parking lot is full and one new customers drops-off their car (carA with keysA), while a second customer wants to pick-up the parked carB and stored keysB; The **goal** is that carA is parked and carB is delivered;

**Hint:** The search space for this problem is very large so try to restrict on which objects the actions can be applied such that the agent will not face a large set of options at every step.

👉 For each of these two tasks make a copy of the instance file, giving them the names `instance-task33a.pl` and `instance-task33b.pl` respectively.

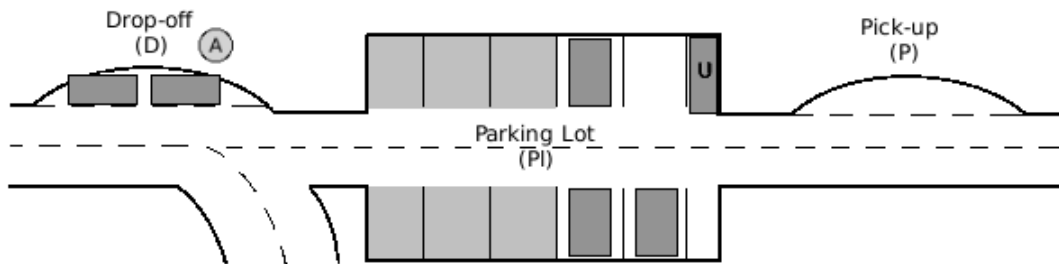


Figure 2: Example map for task 3.3.