FP6-004381-MACS

MACS

Multi-sensory Autonomous Cognitive Systems Interacting with Dynamic Environments for Perceiving and Using Affordances

Instrument: Specifically Targeted Research Project (STReP)

Thematic Priority: 2.3.2.4 Cognitive Systems

D4.3.3 A software prototype of the propositional MACS planning module (implementation)

Due date of deliverable: May 31, 2007
Actual submission date v1: July 11, 2007

Start date of project: September 1, 2004 Duration: 39 months

University of Osnabrück (UOS)

Revision: Version 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

EU Project MACS

Deliverable D4.3.3

A software prototype of the propositional MACS planning module (implementation)

Christopher Lörken, Andreas Bartel, Frank Meyer, Jens Poppenborg, Joachim Hertzberg

Number: **MACS/4/3.3**
WP: *2*
Status: *revision, version 1*
Created at: *June 1, 2007*
Revised at: *v1 – July 10, 2007*
Internal rev: *v3 – July 10, 2007*

FhG/AIS	Fraunhofer Institut für Intelligente Analyse- und Informationssysteme, Sankt Augustin, D
JR_DIB	Joanneum Research, Graz, A
LiU-IDA	Linköpings Universitet, Linköping, S
METU-KOVAN	Middle East Technical University, Ankara, T
OFAI	Österreichische Studiengesellschaft für Kybernetik, Vienna, A
UOS	Universität Osnabrück, Osnabrück, D

This research was partly funded by the European Commission's 6th Framework Programme IST Project MACS under contract/grant number FP6-004381. The Commission's support is gratefully acknowledged.

Corresponding author's address:

Christopher Lörken
Universität Osnabrück
Institut für Informatik
Albrechtstr. 28
D-49076 Osnabrück, Germany

Fraunhofer Institut für Intelligente Analyse- und Informationssysteme
Schloss Birlinghoven
D-53754 Sankt Augustin
Germany

Tel.: +49 (0) 2241 14-2683 (Co-ordinator)

Contact:
Dr.-Ing. Erich Rome

Joanneum Research
Institute of Digital Image Processing
Computational Perception (CAPE)
Wastiangasse 6
A-8010 Graz
Austria

Tel.: +43 (0) 316 876-1769

Contact:
Dr. Lucas Paletta

Linköpings Universitet
Dept. of Computer and Info. Science
Linköping 581 83
Sweden

Tel.: +46 13 24 26 28

Contact:
Prof. Dr. Patrick Doherty

Middle East Technical University
Dept. of Computer Engineering
Inonu Bulvari
TR-06531 Ankara
Turkey

Tel.: +90 312 210 5539

Contact:
Asst. Prof. Dr. Erol Şahin

Österreichische Studiengesellschaft für Kybernetik (ÖSGK)
Freyung 6
A-1010 Vienna
Austria

Tel.: +43 1 5336112 0

Contact:
Prof. Dr. Georg Dorffner

Universität Osnabrück
Institut für Informatik
Albrechtstr. 28
D-49076 Osnabrück
Germany

Tel.: +49 541 969 2622

Contact:
Prof. Dr. Joachim Hertzberg

Contents

1 Introduction

This document describes the current state of the implementation of version 1 of the MACS planning module according to the definitions of deliverable D2.3.2 [5]. The document is structured as follows:

Chapter 1 is this introduction.

Chapter 2 gives a short overview and recapitulation of the functionality and aim of the planning module in the context of an affordance-based architecture.

Chapter 3 describes the current state of the implementation and integration of the module and the structure and usage of the available files in the concurrent versioning system. It moreover contains a short tutorial of how to perform exemplary planning.

Chapter 5 will point out some future work aspects since this is again a living document.

2 Overview

The purpose of integrating a planner in a robot control architecture is to enrich the system with the ability of acting in a goal-directed fashion, to execute tasks of a complexity beyond the scope and ability of merely reactive systems.

In the context of the MACS project, the planning module faces the challenge of showing how the affordance concept can influence and aid the planning process; the resulting benefits and shortcomings need to be evaluated.

As described in deliverable D2.3.2 [5], we follow two different approaches for the planning module, namely, the domain description based planning and the cue-outcome based planning. In this first version of the document, we will so far solely focus on the first approach (section 3), leaving the cue-outcome based planning to future revisions.

3 The domain description based planning module

To briefly recall the aim and functionality of the domain description based planning approach (cf. [5]), it is to say that this approach needs a domain model which is completely provided by a human. Whilst this can be regarded as a drawback of the approach, it is to keep in mind that, when given a concept like in the MACS project, where the basic behaviors of the robot are preprogrammed as well and not learned from scratch, such a world model just reflects the design of the system the programmer has already chosen. So if we additionally provide a formal description of the tools or algorithms we have equipped the robot with, we do not limit the overall validity of the approach any further.

Instead, we have developed a way to model this knowledge on a level abstract enough to leave the choice of which test objects to interact with completely to the agent. This brings about the preponderant advantage of the suggested approach that lies in the usage of the affordance concept for operator grounding.

In other words, the cues and outcomes of an affordance representation triple (ART) are being detected and monitored in the world in order to implement, or ground, the operators delivered by the planning module. A sequence of the operators *lift, approach,*

stack, i.e., a *plan*, will thus be grounded on test objects that afford the specific actions; knowledge encoded in the learned ARTs.

In the line of classical planning approaches, we chose to model the world and its relations in the so-called planning domain definition language (PDDL, [6]).

The reminder of this section will thus sketch in 3.1 the PDDL descriptions used to actually model the domain and problems. 3.4 will present the applied planner and subsection 3.5 will eventually describe the structure of the MACS software repository and give an example of how to use the planner with the operator and problem files provided.

3.1 PDDL representation

The actual design of the world representation is described in detail in D2.3.2 [5] so we will only go into the structure and type of the resulting files here.

PDDL is a standardized language for formulating domain representations and planning problems. A domain representation contains a list of atomic formulas that can either be evaluated to be true or false and a list of operators that change the current truth value of such formulas (see section 3.2). In order to develop a plan, a so-called problem definition has to be provided that assigns initial values to the atomic formulas of the domain and defines a goal state with other truth values (section 3.3). A planner then uses the rules specified as operators in the domain description to develop a sequence of these operators for transforming the initial goal values of the problem definition to the desired goal state (section 3.4).

3.2 Domain description

Figure 1 shows as an example the domain description for a navigation task that allows the robot to change its current region, to trigger a switch in order to open the door as well as to change the rooms through the then available passage. The *predicates*-section contains the atomic formulas that in this case, first of all, describe the current location of the robot and encode the structure of the map; i.e., *robotAt(region1_left)* would describe the world fact that the robot is currently located at that particular region (see Figure 2). Moreover, the predicates-section contains a placeholder for an internal state description of the robot, namely a knowledge that it may have lifted an object appropriate for triggering the switch.[1] More interestingly, it also contains a symbolic representation of whether or not the abstract affordance type of a triggerable switch or a passage between two regions has been perceived in a particular region of the map. In which direct form it has been perceived, i.e., which actually learned affordance representation triple belonging to this abstract affordance type has led to the perception of the affordance, does not matter for the planner.

The domain description furthermore defines the three operators *approach-region*, *trigger-switch*, and *change-room* which describe in their *precondition*-parts which of the domain's predicates have to be instantiated with a true value in order for the action to be applicable. The *effect*-part describes the changes applied to the current truth values of the system.

For example, the *change-room* operator requires, firstly, the robot to be in a door region and, secondly, the perception of the affordance type *passable* from that door region

[1] Note that the process of lifting such an item is skipped here due to reasons of brevity. You may refer to appendices A.1 and B.1 for the current state of the definition.

```
(define (domain macs-navigation)
   (:requirements :strips :typing :equality )
   (:types doorRegion switchRegion - region
           switchRoom - room )
   (:predicates
        (robotAt  ?robotRegion - region )
        (inRoom   ?region - region ?room - room )
        (hasSwitchReleaserLifted )
        (switch-triggerable ?switchRegion -region )
        (passable ?startRegion - region ?targetRegion - region ))

   (:action approach-region
       :parameters (?startRegion  - region
                    ?targetRegion - region
                    ?room - room)
       :precondition
           (and (robotAt ?startRegion )
                (inRoom  ?startRegion  ?room )
                (inRoom  ?targetRegion ?room )
                (not (= ?startRegion ?targetRegion )))
       :effect
           (and (robotAt ?targetRegion )
                (not (robotAt ?startRegion ))))

   (:action trigger-switch
       :parameters (?doorRegion ?otherDoorRegion - doorRegion
                    ?switchRegion - switchRegion )
       :precondition
           (and (robotAt  ?switchRegion )
                (hasSwitchReleaserLifted )
                (switch-triggerable ?switchRegion )
                (not (= ?doorRegion ?otherDoorRegion )))
       :effect
           (and (passable ?doorRegion ?otherDoorRegion )
                (passable ?otherDoorRegion ?doorRegion )
                (not (switch-triggerable ?switchRegion ))
                (not (hasSwitchReleaserLifted ))))

   (:action change-room
       :parameters (?doorRegion       - doorRegion
                    ?targetDoorRegion - doorRegion )
       :precondition
           (and (robotAt  ?doorRegion)
                (not   (= ?doorRegion ?targetDoorRegion ))
                (passable ?doorRegion ?targetDoorRegion ))
       :effect
           (and (not (robotAt ?doorRegion ))
                (robotAt ?targetDoorRegion ))))
```

Figure 1: Domain description for a simple navigation example with the door to be opened by triggering the switch.

```
(define (problem macs-prob)
        (:domain macs-navigation)

        (:objects
           region1_left region2_left region1_right - region
           switchRegion - switchRegion
           doorRegionLeft doorRegionRight - doorRegion
           rightRoom    - room
           leftRoom     - switchRoom )

        (:init
           (inRoom  region1_left    leftRoom )
           (inRoom  region2_left    leftRoom )
           (inRoom  switchRegion    leftRoom )
           (inRoom  doorRegionLeft  leftRoom )
           (inRoom  region1_right   rightRoom )
           (inRoom  doorRegionRight rightRoom )

           (robotAt region1_left )
           (hasSwitchReleaserLifted )
           (switch-triggerable switchRegion ))

        (:goal
                (robotAt region1_right )))
```

Figure 2: Problem definition for the navigation domain.

to another one in order to 'move' the robot through that door (the result or effect being robotAt(?targetDoorRegion)).

Please keep in mind, that the operators and formulas specified here are just symbolic symbols for planning. It is not defined here how they are to be implemented by the Execution Module of the architecture.

3.3 Problem definition

An exemplary problem definition is given in Figure 2. The main part of this definition describes the map as it is shown in Figure 3 by specifying which (symbolic) objects of rooms and regions exist and in which relation they stand to each other.

The problem definition is furthermore a snapshot of the current situation as it is perceived by and/or known to the robot. In this example, the robot has localized itself in region1_left and has already lifted an item that it has perceived as being suitable to release, or to trigger, the switch.

The last part of the *init*-definition tells the planner, that the map contains the entry of a perceived 'switch-triggerable' affordance in the switch region of the map.

The goal description finally postulates, as the target of the planner, to develop a plan that has as a result the robot being located in region1_right, i.e., region1 in the other room.

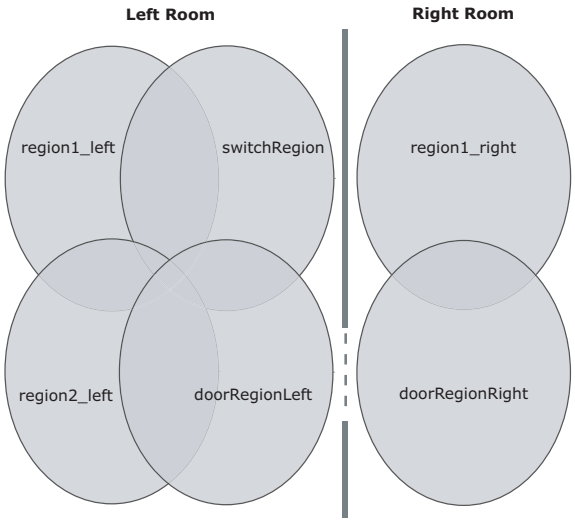

Figure 3: Topological Map - Regions are not sharply separated.

3.4 FF-planner

One of the benefits of using PDDL as input language is that there is a wide variety of available planning systems that can freely be used and that work without major problems on this description language.

We chose to employ the FF planner of Jörg Hoffmann [3]. We picked this planner as it has performed very well on international planning competitions, it is free to use and open source, which eases the integration process significantly. FF provides the right level of complexity and power in terms of supported PDDL features.

Providing the introduced domain description and problem definition as input to FF, the planner develops the plan depicted in Plan 1.

Plan 1 FF Generated Plan

0:	APPROACH-REGION	region1_left	switchregion	leftroom
1:	TRIGGER-SWITCH	doorregionleft	doorregionright	switchregion
2:	APPROACH-REGION	switchregion	doorregionleft	leftroom
3:	CHANGE-ROOM	doorregionleft	doorregionright	
4:	APPROACH-REGION	doorregionright	region1_right	rightroom

The planner thus recognizes that the robot can only change the rooms if the affordance of something passable is being perceived between the door regions of the two rooms. As the perception of this affordance is known to be the result when the robot releases the switch by implementing the trigger-switch operator, the planner will choose to put the currently lifted appropriate test object on the switch and will thus perceive the passable-affordance.

Note that this example is somewhat kept short to point out the necessary characteristics of the planner, and that a more extensive version is provided in appendices A.3 and B.3.

3.5 Installation and Usage

The current version of the planning module can be found in the MACS CVS system on the gibson server. It is located in the folder `UOS/planner-v1`. That folder has the following structure:

`pddl/domains` - The folder contains examples of the current domain description.

`pddl/problems` - The folder contains situation or problem definitions to be used with the domain descriptions.

`external` - The folder contains the external package of Jörg Hoffmann's FF-planner v2.3 obtained from [2].

As PDDL is only an input language to standard planners it does not require specific software or compilation.

To get a plan, the FF located in the `external` folder has to be extracted by running `tar xzvf FF-v2.3.tgz`.

When using the MACS reference control system of SuSE 9.3 it does not even have to be compiled since a working executable is already included in the archive.

To run the planner on one of the example scenarios, simply go to the `pddl` folder and execute:

```
../external/FF-v2.3/ff  -o domains/domain-highstack.pddl
                        -f problems/stacking-problem.pddl
```

for planning a stacking scenario. The flag `-o` specifies the so-called *operator file*, i.e. the domain description while `-f` specifies the *fact file* that is the current state of the system and the goal description or in other words the problem definition.

The second provided example can be instantiated by running:

```
../external/FF-v2.3/ff  -o domains/domain.pddl
                        -f problems/navigation-problem.pddl.
```

4 Integration & Additional Implementations

Integration with the other modules of the architecture, primarily with OFAI regarding the Affordance Representation Repository and METU-KOVAN regarding the Execution Module is still ongoing and will be presented here in later editions of this document.

The following sections will provide a short overview of the additional work done and the current state of the integration of that work in the complete system.

4.1 Crane

The MACS-crane has been integrated in the robot control code provided by FhG-IAIS as a step towards implementing the autonomous crane behaviors.

The crane has deeply been integrated in the C++ code on the level of the other previously available devices including the necessary connections to the CRobot class and

the CRobotState. The crane access was held in a way compatible to the IDL interfaces defined in D1.1.2 [7].

As a first version before the crane can be controlled autonomously, it was integrated in the behavior structure of the robot control software as a crane behavior that can be controlled by the robot or via the graphical user interface (see section 4.2).

4.2 Graphical User Interface

The affordance-based planning module will overall include three operator interfaces. The main window will represent the particle filter outputs (see section 4.3) such as the amount and the position of particles, the estimated position of the robot, and a static map of the surroundings. In addition, it is planned to incorporate a visualization of all detected affordance representation triples according to their region within the demonstrator. Each abstract triple will be represented in terms of a symbol or label that is visually distinguishable and that carries details about the triple's class and its descriptors.

In order to generate goals for the planning, it will be possible to relocate each labelled triple from its original location to a target location within the map of the main window by means of user interaction via mouse or keyboard input.

Upon relocation of an ART, the planner will be triggered and the outcome of the planning module will be provided within a second window. This window reports all details concerning the current plan in form of text, such that the current step of the plan is always highlighted.

The third window concerns the crane control. Within that interface it is possible to manipulate the horizontal and the vertical position of the magnet, the arm position, and all speed values. All values can be changed via spinbuttons and a mouse or by direct input via the keyboard. The displayed numbers always represent the current crane values.

A first version of this system has been realized using GTKmm. GTKmm was chosen because it fits well into the existing modules. It is a C++ wrapper for GTK+ which it written in C.

Extensions planned for this initial system nevertheless showed the need to include additional libraries that again would have dependencies violating the reference control system of SuSE 9.3. We therefore decided to port the system to Java. All further implementations and extensions of the GUI will henceforth rely on JDK6.

4.3 Map & Localization

As specified above, the planner uses a map to localize the robot and the system's percepts of affordance representation triples within regions. Here, the global localization of the robot will be accomplished by using a *particle filter* (cf. e.g. [1], [4]). Basically, the particle filter consists of three phases: the sampling step, the calculation of the importance weights and the resampling step. In the following paragraphs, these three phases will be described more closely. Finally, there will be some notes on the current implementation and integration of the particle filter.

Phase 1: Sampling

In the sampling step, the pose of all particles is updated based on the relative movement of the robot since the last update. The necessary data is gained from odometry mea-

surements. Should a particle intersect with an obstacle (e.g., a wall) the particle will be removed from the distribution.

Phase 2: Importance Weights

The importance weights of the remaining particles are calculated based on the current 2D laser scan of the robot's environment. For this purpose, the particle filter is given the distance measurements of the current laser scan which will then be compared to the predicted observations of the particles based on their position and orientation in the map. To reduce the computing time in this step, only a few of the available distance measurements are compared, e.g., every 10°. Finally, the importance weights of the particles are calculated based on the difference between expected distances and observed distances. Here, a shorter observed distance is less fatal than a longer observed distance since in a dynamic environment there is always a chance that a movable obstacle will block the path of a laser scan while, in the case of longer observed distances, it is unlikely that a wall has suddenly vanished.

Phase 3: Resampling

In this last step, the updated distribution is resampled based on the importance weights of the particles. This is done by applying the universal stochastic sampling algorithm which adds particles with a large importance weight more often to the resampled distribution than particles with a low importance weight which in fact, may not be added at all. Additionally, if the robots pose could not be determined sufficiently well, an additional number of new particles are randomly added to the distribution.

Implementation and Integration

Currently, the particle filter is working with a grid map of the MACS Demonstration Arena which is constructed from a line map upon initialization of the particle filter. Furthermore, the particle filter will be initialized with a uniform distribution of pre-defined number of particles which are distributed randomly in the map. Since pose updates and laser scan updates in the MACS Project can be made independent of each other, the sampling step and the calculation of the importance weights of the particle filter are also separate and will be called in the corresponding methods. In the now commencing testing phase, the integration of the particle filter will be solidified and the parameters will be adjusted to optimize for reliability and computing time.

5 Future Work

The upcoming work will primarily focus on integrating the above introduced planning module into the architecture. Therefore, the domain definition will be extended to deal with the different use cases and interfaces with the other components of the system will be defined and implemented. Chapter 3 will then incorporate the results of the currently ongoing discussion and integration work with METU-KOVAN (execution control) and OFAI (learning module).

Based on mainly trilateral discussions between IAIS, OFAI, and UOS, the second approach to the planning module, i.e., the cue-outcome based planning module, will eventually be evaluated again and, if it proves feasible, be implemented as an additional approach.

References

[1] Dieter Fox. Adapting the sample size in particle filters through kld-sampling. *I. J. Robotic Res.*, 22(12):985–1004, 2003.

[2] Jörg Hoffmann. Fast forward planner, 2001. http://members.deri.at/~joergh/ff.html.

[3] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[4] Cody C. T. Kwok, Dieter Fox, and Marina Meila. Real-time particle filters. In *Neural Information Processing Systems*, pages 1057–1064, 2002.

[5] C. Lörken and J. Hertzberg. A specification for a propositional planner and its interface to the MACS execution control module. Technical Report MACS/2/3.2 v1, University of Osnabrück, KBS Group, Osnabrück, Germany, March 2007.

[6] D. McDermott. PDDL - The planning domain definition language. Technical report, Yale University, 1998.

[7] Rainer Worst, Claus Hoffmann, Björn Wingman, Maya Cakmak, and Martin Hülse. Specification of module interfaces. Technical Report MACS/1/1.2 v3, Institute for Autonomous Intelligent Systems (FhG/AIS), 2004.

The appendix lists the two examples of navigation (appendix A), including lifting items that can trigger the switch, and high-stacking (appendix B), i.e. stacking test objects to a height of more than two test objects. Please note, that the latter example of the high-stacking scenario uses a newer and more sophisticated version of the domain specification. The files listed here can also be found in the CVS.

A Example: Navigation Task

A.1 Domain description

CVS file: `UOS/planner-v1/pddl/domains/domain.pddl`.

```
;; Domain MACS Example
;; PDDL definition of the MACS domain and the operators needed
;; for changing rooms and opening the door by triggering the
;; switch with appropriate test objects.
;; Assumption is that the operators are fully specified and that
;; a map representation holds the abstract affordance types
;; represented in regions.

(define (domain macs-example)


   (:requirements :strips :typing :equality )


   (:types doorRegion switchRegion - region
           switchRoom - room )


; ########### Predicates ################
   (:predicates

        ; AFFORDANCE PROPOSITIONS:
        ; affordance propositions get an assigend truth value
        ; determined from the snapshot of the world in the moment
        ; in which the planner is triggered.
        ; They are as well encoded in the effect part of an operator
        ; that results in this affordance to be available.
        ; E.g.:
        ;     - drop action follows liftable, or
        ;     - trigger switch follows passable.

        (non-releaser-liftable ?region - region )
        (switch-releaser-liftable ?region - region )

        ; open passage from one room to another.
        ; Implies knowledge of a map, and knowledge of the effect of
        ; opening doors, etc.
        (passable
                 ?startRegion - region
                 ?targetRegion - region )
        ; true if the switch is empty so one can place something on it
        (switch-triggerable ?region - switchRegion )
        ; true if something can be removed from the switch
        (affords-removing-from-switch ?region - switchRegion )
```

```
        (is-stacking-base ?region - region )
        (affords-stacking ?region - region )
        (is-liftable-stacking-base ?region - region )
        (affords-unstacking ?region - region )
        (affords-base-unstacking ?region - region )
        (affords-unstacking-from-liftable-base ?region - region )
        (affords-base-unstacking-from-liftable-base ?region - region )

        (hasStackingBaseLifted )
        (hasStackableLifted )

        ; NORMAL PREDICATES:

        ; The robot is in a certain region of a certain room.
        ; Note that we do not have to encode explicit entities as we
        ; only consider abstract affordances that have been perceived
        ; in the corresponding regions.
        (robotAt ?robotRegion - region )
        ; true if a region is in a room
        (inRoom ?region - region ?room - room )
        ; true if the robot has lifted something
        (hasLiftedSomething )
        ; true if robot has lifted an object that affords
        ; releasing the switch
        (hasSwitchReleaserLifted )


) ;; End predicates



; ############   Operators (Actions)  ################

(:action approach-region
    :parameters (?startRegion - region
                 ?targetRegion - region
                 ?room - room)
    :precondition
        (and
            (robotAt ?startRegion )
            (not (hasLiftedSomething ))
            ; make sure start and target region are in the same room
            (inRoom ?startRegion ?room )
            (inRoom ?targetRegion ?room )
            (not (= ?startRegion ?targetRegion )))
    :effect
        (and
            (robotAt ?targetRegion )
            (not (robotAt ?startRegion ))))

(:action carry
    :parameters (?startRegion - region
                 ?targetRegion - region
                 ?room - room )
    :precondition
        (and
```

```
            (robotAt ?startRegion )
            (hasLiftedSomething )
            ; make sure start and target region are in the same room
            (inRoom ?startRegion ?room )
            (inRoom ?targetRegion ?room )
            (not (= ?startRegion ?targetRegion )))
    :effect
        (and
            (robotAt ?targetRegion )
            (not (robotAt ?startRegion ))))


(:action lift-non-releaser
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            ; liftable perceived in that region?
            (non-releaser-liftable ?region )
            (not (hasLiftedSomething )))
    :effect
        (and
            (hasLiftedSomething )
            (not (non-releaser-liftable ?region ))))


(:action lift-switch-releaser
    :parameters (?region - region ?room - room )
    :precondition
        (and
            (robotAt ?region )
            (switch-releaser-liftable ?region )
            (not (hasLiftedSomething )))
    :effect
        (and
            (hasSwitchReleaserLifted )
            (hasLiftedSomething )
            ; negate the switch-releaser-liftable affordance for this
            ; location. If we would not do this, the
            ; planner would have an inexhaustible source
            ; of releaser liftable test objects here.
            (not (switch-releaser-liftable ?region ))))

(:action lift-stacking-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            ; affordance perceived in that region?
            (is-liftable-stacking-base ?region )
            (not (hasLiftedSomething )))
    :effect
        (and
            (hasLiftedSomething )
            (hasStackingBaseLifted )
            (not (is-liftable-stacking-base ?region ))))
```

```
(:action lift-stackable
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            ; stackable perceived in that region?
            (affords-stacking ?region )
            (not (hasLiftedSomething )))
    :effect
        (and
            (hasLiftedSomething )
            (hasStackableLifted )
            (not (affords-stacking ?region ))))


(:action drop-non-releaser
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (hasLiftedSomething )
            ; all types have own drop operators
            (not (hasSwitchReleaserLifted ))
            (not (hasStackingBaseLifted ))
            (not (hasStackableLifted )))
    :effect
        (and
            (not (hasLiftedSomething ))
            (non-releaser-liftable ?region )))


(:action drop-switch-releaser
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (hasSwitchReleaserLifted ))
    :effect
        (and
            (not (hasSwitchReleaserLifted ))
            (not (hasLiftedSomething ))
            ; switch-releaser liftable affordance will be perceivable
            ; in this region after dropping.
            (switch-releaser-liftable ?region )))


(:action drop-stacking-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (hasStackingBaseLifted ))
    :effect
        (and
            (not (hasLiftedSomething ))
            (not (hasStackingBaseLifted ))
            (is-liftable-stacking-base ?region )))
```

```
(:action drop-stackable
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (hasStackableLifted ))
    :effect
        (and
            (not (hasLiftedSomething ))
            (not (hasStackableLifted ))
            (affords-stacking ?region )))

(:action unstack-stackable
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (not (hasLiftedSomething ))
            (affords-unstacking ?region))
    :effect
        (and
            (hasLiftedSomething )
            (hasStackableLifted )
            (affords-stacking ?region )))

(:action unstack-stackable-from-liftable-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (not (hasLiftedSomething ))
            (affords-unstacking-from-liftable-base ?region))
    :effect
        (and
            (hasLiftedSomething )
            (hasStackableLifted )
            (not (affords-unstacking-from-liftable-base ?region ))
            (is-liftable-stacking-base ?region )))

(:action unstack-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (not (hasLiftedSomething ))
            (affords-base-unstacking ?region )
            ; on top of stack is a liftable base
            (is-liftable-stacking-base ?region ))
    :effect
        (and
            (hasLiftedSomething )
            (hasStackingBaseLifted )
            (not (affords-base-unstacking ?region ))
            (not (is-liftable-stacking-base ?region ))
            ; removed from non-liftable base:
```

```
            (is-stacking-base ?region )))


(:action unstack-base-from-liftable-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (not (hasLiftedSomething ))
            ; there is a liftable base beneath the top object
            (affords-base-unstacking-from-liftable-base ?region )
            ; on top of stack is a liftable base
            (is-liftable-stacking-base ?region ))
    :effect
        (and
            (hasLiftedSomething )
            (hasStackingBaseLifted )
            (not (affords-base-unstacking-from-liftable-base ?region ))))

(:action trigger-switch
    :parameters   (?doorRegion ?otherDoorRegion - doorRegion
                   ?switchRegion - switchRegion )
    :precondition
        (and
            (robotAt  ?switchRegion )
            (hasSwitchReleaserLifted )
            ; Does the switch afford to be triggered
            ; (i.e.: is it empty?)
            (switch-triggerable ?switchRegion )
            (not (= ?doorRegion ?otherDoorRegion )))
   :effect
        (and
            ; result is affordance perception of something passable
            (passable ?doorRegion ?otherDoorRegion )
            (passable ?otherDoorRegion ?doorRegion )
            ; the switch will not afford to be triggered again
            (not (switch-triggerable ?switchRegion ))
            ; it will afford to remove something from it
            (affords-removing-from-switch ?switchRegion )
            (not (hasSwitchReleaserLifted ))
            (not (hasLiftedSomething ))))


(:action remove-releaser-from-switch
    :parameters   (?doorRegion ?otherDoorRegion - doorRegion
                   ?switchRegion - switchRegion )
    :precondition
        (and
            (robotAt  ?switchRegion )
            (not (= ?doorRegion ?otherDoorRegion ))
            (not (hasLiftedSomething ))
            (affords-removing-from-switch ?switchRegion )
            ; remove heavy implies that the passage can be perceived.
            (passable ?doorRegion ?otherDoorRegion )
            (passable ?otherDoorRegion ?doorRegion ))
   :effect
        (and
```

```
            ; result is door == closed == not passable
            (not (passable ?doorRegion ?otherDoorRegion ))
            (not (passable ?otherDoorRegion ?doorRegion ))
            ; the switch will afford to be triggered
            (switch-triggerable ?switchRegion )
            ; it will not afford to remove something from it anymore
            (not (affords-removing-from-switch ?switchRegion ))
            ; the door was open --> we have removed something heavy
            (hasSwitchReleaserLifted )
            (hasLiftedSomething )))


(:action remove-non-releaser-from-switch
    :parameters    (?doorRegion ?otherDoorRegion - doorRegion
                    ?switchRegion - switchRegion )
    :precondition
        (and
            (robotAt  ?switchRegion )
            (not (hasLiftedSomething ))
            (not (= ?doorRegion ?otherDoorRegion ))
            (affords-removing-from-switch ?switchRegion )
            ; non-releaser implies that passage cannot be perceived
            (not (passable ?doorRegion ?otherDoorRegion ))
            (not (passable ?otherDoorRegion ?doorRegion )))
   :effect
        (and
            ; the switch will afford to be triggered
            (switch-triggerable ?switchRegion )
            ; it will not afford to remove something from it anymore
            (not (affords-removing-from-switch ?switchRegion ))
            ; Door was not open, item was no releaser
            (hasLiftedSomething )))

(:action stack-stackable
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (is-stacking-base ?region )
            (hasStackableLifted ))
    :effect
        (and
            (not (hasLiftedSomething ))
            (not (hasStackableLifted ))
            (not (is-stacking-base ?region ))
            (affords-unstacking ?region )))

(:action stack-stacking-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (is-stacking-base ?region )
            (hasStackingBaseLifted ))
    :effect
        (and
            (not (hasLiftedSomething ))
```

```
            (not (hasStackingBaseLifted ))
            (not (is-stacking-base ?region ))
            (affords-base-unstacking ?region )
            (is-liftable-stacking-base ?region )))

(:action stack-base-on-liftable-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (is-liftable-stacking-base ?region )
            (hasStackingBaseLifted ))
    :effect
        (and
            (not (hasLiftedSomething ))
            (not (hasStackingBaseLifted ))
            ;maintain the affordance of a liftable stacking base:
            (affords-base-unstacking-from-liftable-base ?region )))

(:action stack-stackable-on-liftable-base
    :parameters (?region - region )
    :precondition
        (and
            (robotAt ?region )
            (is-liftable-stacking-base ?region )
            (hasStackableLifted ))
    :effect
        (and
            (not (hasLiftedSomething ))
            (not (hasStackableLifted ))
            (not (is-liftable-stacking-base ?region ))
            (affords-unstacking-from-liftable-base ?region )))


(:action change-room
    :parameters (?doorRegion - doorRegion
                 ?targetDoorRegion - doorRegion )
    :precondition
        (and
            (robotAt ?doorRegion)
            ;disambiguate with carry-through
            (not (hasLiftedSomething ))
            (not (= ?doorRegion ?targetDoorRegion ))
            (passable ?doorRegion ?targetDoorRegion ))
    :effect
        (and
            (not (robotAt ?doorRegion ))
            (robotAt ?targetDoorRegion )))

(:action carry-through
    :parameters (?doorRegion - doorRegion
                 ?targetDoorRegion - doorRegion )
    :precondition
        (and
            (robotAt ?doorRegion)
            (hasLiftedSomething )
            (not (= ?doorRegion ?targetDoorRegion ))
```

```
        (passable ?doorRegion ?targetDoorRegion ))
    :effect
        (and
            (not (robotAt ?doorRegion ))
            (robotAt ?targetDoorRegion ))))
```

A.2 Problem definition

CVS file: `UOS/planner-v1/pddl/problems/navigation-problem.pddl`.

```
(define (problem macs-prob)

        (:domain macs-example)

        (:objects
           region1_left region2_left region1_right  - region
           switchRegion - switchRegion
           doorRegionLeft doorRegionRight - doorRegion
           rightRoom - room
           leftRoom - switchRoom )

        (:init
           (robotAt region1_left )
           (inRoom region1_left leftRoom )
           (inRoom region2_left leftRoom )
           (inRoom switchRegion leftRoom )
           (inRoom doorRegionLeft leftRoom )
           (inRoom region1_right rightRoom )
           (inRoom doorRegionRight rightRoom )

           (switch-releaser-liftable region2_left )
           (non-releaser-liftable region1_right )
           (affords-removing-from-switch switchRegion ))

        (:goal
            (and (non-releaser-liftable region1_left )
                 (non-releaser-liftable switchRegion )
                 (switch-triggerable switchRegion )
                 (not (hasLiftedSomething )))))
```

A.3 Navigation plan

Call

```
../external/FF-v2.3/ff  -o domains/domain.pddl
                        -f problems/navigation-problem.pddl.
```

in `UOS/planner-v1/pddl` to generate.

```
0:  APPROACH-REGION          region1_left      switchregion      leftroom
1:  REMOVE-NON-RELEASER-FROM-SWITCH
                             doorregionright   doorregionleft    switchregion
2:  CARRY                    switchregion      region1_left      leftroom
3:  DROP-NON-RELEASER        region1_left
4:  APPROACH-REGION          region1_left      region2_left      leftroom
```

5:	LIFT-SWITCH-RELEASER	region2_left	leftroom	
6:	CARRY	region2_left	switchregion	leftroom
7:	TRIGGER-SWITCH	doorregionright	doorregionleft	switchregion
8:	APPROACH-REGION	switchregion	doorregionleft	leftroom
9:	CHANGE-ROOM	doorregionleft	doorregionright	
10:	APPROACH-REGION	doorregionright	region1_right	rightroom
11:	LIFT-NON-RELEASER	region1_right		
12:	CARRY	region1_right	doorregionright	rightroom
13:	CHARRY-THROUGH	doorregionright	doorregionleft	
14:	CARRY	doorregionleft	switchregion	leftroom
15:	DROP-NON-RELEASER	switchregion		
16:	REMOVE-RELEASER-FROM-SWITCH	doorregionright	doorregionleft	switchregion
17:	DROP-SWITCH-RELEASER	switchregion		

B Example: Stacking Task

B.1 Domain description

CVS file: `UOS/planner-v1/pddl/domains/domain-highstack.pddl`.

```
(define (domain macs-stacking)


   (:requirements :adl :strips :typing :equality )

   (:types doorRegion switchRegion - region
           switchRoom - room
           objecttype )

   (:constants nothing base liftablebase stackable liftable trigger - objecttype )

   (:predicates

        (robotAt ?robotRegion - region )
        (inRoom ?region - region ?room - room )
        (isBase ?base - objecttype )
        (lifted ?objecttype - objecttype)

        (affords-to-be-base ?region - region )
        (affords-lifting ?region - region )
        (affords-to-be-liftable-base ?region - region )
        ;; Something liftable that can be placed on something else.
        (affords-to-be-stackable ?region - region )
        ;; Something liftable on top of something else
        (affords-to-be-unstackable ?region - region )
        ;; a liftable base on top of something
        (affords-high-stacking ?region - region )
        ;; Remove something from a high stack tower
        (affords-high-unstacking ?region - region )
   )

;; Action that either lifts a liftable, a liftable base or a stackable.
(:action lift
    :parameters (?liftable - objecttype
                 ?region - region )
    :precondition
        (and
            (robotAt ?region)
            (lifted nothing)
            (or
               (and (affords-lifting ?region )
                    (= ?liftable liftable ))
               (and (affords-to-be-liftable-base ?region )
                    (= ?liftable liftablebase ))
               (and (affords-to-be-stackable ?region )
                    (= ?liftable stackable ))))
    :effect (and
        (when (= ?liftable liftablebase )
           (and
               (not (lifted nothing ))
               (lifted ?liftable )
```

```
            (not (affords-to-be-liftable-base ?region ))))
        (when (= ?liftable stackable )
           (and
               (not (lifted nothing ))
               (lifted ?liftable )
               (not (affords-to-be-stackable ?region ))))
        (when (= ?liftable liftable )
           (and
               (not (lifted nothing ))
               (lifted ?liftable )
               (not (affords-lifting ?region ))))))

;; Drop anything that is lifted.
(:action drop
    :parameters (?liftedType - objecttype
                 ?region - region )
    :precondition
        (and
            (not (= ?liftedType nothing ))
            (lifted ?liftedType )
            (robotAt ?region )
        )
    :effect
        (and
            (when (= ?liftedType liftable )
                  (affords-lifting ?region ))
            (when (= ?liftedType stackable )
                  (affords-to-be-stackable ?region ))
            (when (= ?liftedType liftableBase )
                  (affords-to-be-liftable-base ?region ))
            (not (lifted ?liftedType ))
            (lifted nothing )))

;; Stackables and liftable bases are assumed to be stackable on something else.
(:action stack
    :parameters (?liftedType - objecttype
                 ?region - region )
    :precondition
        ; Stack either a stackable or a liftable base
        (or
            (and
                (= ?liftedType stackable )
                (lifted ?liftedType )
                (robotAt ?region )
                (affords-to-be-base ?region ))
            (and
                (= ?liftedType liftablebase )
                (lifted ?liftedType )
                (robotAt ?region )
                (affords-to-be-base ?region )))
    :effect
        (and
            (when (= ?liftedType stackable)
                  (and
                        (not (lifted ?liftedType ))
                        (lifted nothing )
                        (not (affords-to-be-base ?region ))
```

```
                    (affords-to-be-unstackable ?region)))
            (when (= ?liftedType liftablebase)
                  (and
                        (not (lifted ?liftedType ))
                        (lifted nothing )
                        (not (affords-to-be-base ?region ))
                        (affords-to-be-unstackable ?region)
                        (affords-high-stacking ?region )))))

;; Stackables and liftable bases are assumed to be unstackable
;; from something else.
(:action unstack
    :parameters (?unstackType - objecttype
                 ?region - region )
    :precondition
        (or
            (and
                (lifted nothing )
                (robotAt ?region )
                (= ?unstackType liftablebase )
                (affords-to-be-unstackable ?region )
                ;; This makes it a base
                (affords-high-stacking ?region ))
            (and
                (lifted nothing )
                (robotAt ?region )
                (= ?unstackType stackable )
                (affords-to-be-unstackable ?region )
                (not (affords-high-stacking ?region ))))
    :effect
        (and
            (when (= ?unstackType liftablebase )
                  (and
                        (not (lifted nothing ))
                        (lifted liftablebase )
                        (not (affords-high-stacking ?region ))
                        (not (affords-to-be-unstackable ?region ))
                        (affords-to-be-base ?region )))
            (when (= ?unstackType stackable)
                  (and
                        (not (lifted nothing ))
                        (lifted stackable )
                        (not (affords-to-be-unstackable ?region ))
                        (affords-to-be-base ?region )))))


;; When two items have been stacked and they still afford stacking
;; something on them, they afford high-stacking.
(:action highstack
    :parameters (?liftedType - objecttype
                 ?region - region )
    :precondition
        (or
            (and
                (= ?liftedType stackable )
                (lifted ?liftedType )
                (robotAt ?region )
```

```
                (affords-high-stacking ?region ))
            (and
                (= ?liftedType liftablebase )
                (lifted ?liftedType )
                (robotAt ?region )
                (affords-high-stacking ?region )))
    :effect
        (and
            (when (= ?liftedType stackable)
                (and
                    (not (lifted ?liftedType ))
                    (lifted nothing )
                    (not (affords-high-stacking ?region ))
                    (affords-high-unstacking ?region )))
            (when (= ?liftedType liftablebase )
                (and
                    (not (lifted ?liftedType ))
                    (lifted nothing )
                    (affords-high-unstacking ?region )))))

;; Unstack something that is on top of two other items.
(:action highunstack
    :parameters (?unstackType - objecttype
                 ?region - region )
    :precondition
        (or
            (and
                (lifted nothing )
                (robotAt ?region )
                (= ?unstackType liftablebase )
                (affords-high-unstacking ?region )
                ; == liftable base
                (affords-high-stacking ?region ))
            (and
                (lifted nothing )
                (robotAt ?region )
                (= ?unstackType stackable )
                (affords-high-unstacking ?region )
                ; == nur stackable
                (not (affords-high-stacking ?region ))))
    :effect
        (and
            ;; If highstacking was still afforded, the top element was a
            ;; liftable base. The high stacking affordance is still valid,
            ;; as it would be a normal unstack action otherwise.
            (when (= ?unstackType liftablebase )
                (and
                    (not (lifted nothing ))
                    (lifted liftablebase )
                    (not (affords-high-unstacking ?region ))
                    (affords-to-be-unstackable ?region )
                    (affords-high-stacking ?region )))
            (when (= ?unstackType stackable )
                (and
                    (not (lifted nothing ))
                    (lifted stackable )
                    (not (affords-high-unstacking ?region ))
```

```
                        (affords-to-be-unstackable ?region )
                        (affords-high-stacking ?region )))))

;; Approach a region in the same room.
(:action approach-region
    :parameters (?startRegion - region
                 ?targetRegion - region
                 ?room - room )
    :precondition
        (and
            (robotAt ?startRegion )
            (lifted nothing )
            ; make sure start and target region are in the same room
            (inRoom ?startRegion ?room )
            (inRoom ?targetRegion ?room )
            (not (= ?startRegion ?targetRegion )))
    :effect
        (and
            (robotAt ?targetRegion )
            (not (robotAt ?startRegion ))))

;; Carry something to a different region in the same room.
(:action carry
    :parameters (?objectType - objectType
                 ?startRegion - region
                 ?targetRegion - region
                 ?room - room )
    :precondition
        (and
            (robotAt ?startRegion )
            (not (lifted nothing))
            ; determine object type (not needed but better for output)
            (lifted ?objectType )
            ; make sure start and target region are in the same room
            (inRoom ?startRegion ?room )
            (inRoom ?targetRegion ?room )
            (not (= ?startRegion ?targetRegion )))
    :effect
        (and
            (robotAt ?targetRegion )
            (not (robotAt ?startRegion ))))

)
```

B.2 Problem definition

CVS file: `UOS/planner-v1/pddl/problems/stacking-problem.pddl`.

```
(define (problem macs-prob)

        (:domain macs-stacking)

        (:objects
           region1_left region2_left region1_right  - region
           switchRegion - switchRegion
           doorRegionLeft doorRegionRight - doorRegion
           rightRoom - room
```

```
    leftRoom - switchRoom )


  (:init
    (robotAt region1_left)
    (inRoom region1_left leftRoom )
    (inRoom region2_left leftRoom )
    (inRoom switchRegion leftRoom )
    (inRoom doorRegionLeft leftRoom )
    (inRoom region1_right rightRoom )
    (inRoom doorRegionRight rightRoom )

    (lifted nothing)
    (affords-high-unstacking doorRegionLeft ))

  (:goal (and
          (lifted nothing)
          (not (affords-high-stacking doorRegionLeft ))
          (not (affords-high-unstacking doorRegionLeft )))))
```

B.3 Stacking plan

Call

```
../external/FF-v2.3/ff  -o domains/domain-highstack.pddl
                        -f problems/stacking-problem.pddl.
```

in `UOS/planner-v1/pddl` to generate.

```
0: APPROACH-REGION   region1_left  doorregionleft leftroom
1: HIGHUNSTACK       stackable     doorregionleft
2: DROP              stackable     doorregionleft
3: UNSTACK           liftablebase  doorregionleft
4: DROP              liftablebase  doorregionleft
```