KomodoPlan – Project summary

Members:

Lior Zur-Lotan, 300197654, [llutan@cs.bgu.ac.il](mailto:llutan@cs.bgu.ac.il)

Maor Ashkenazi, 312498017, [maorash@cs.bgu.ac.il](mailto:maorash@cs.bgu.ac.il)

**Project Description**

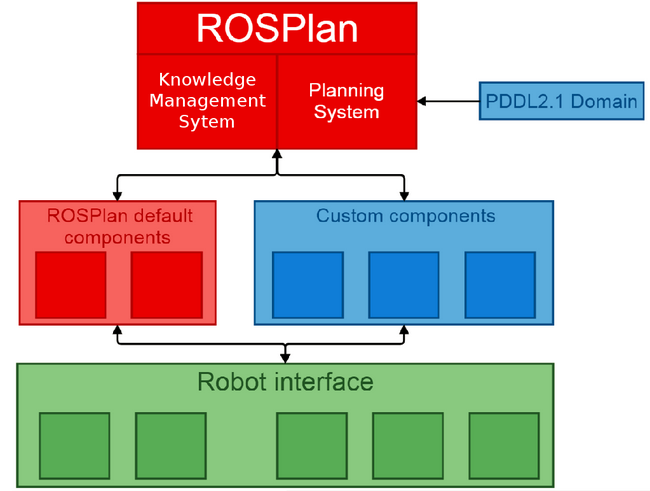
Our main objective was to lay the grounds for ROSPlan[[1]](#footnote-1) [[2]](#footnote-2) usage on the komodo robot. We wanted to create an implementation of ROSPlan’s abilities that would serve as a baseline for future projects in this area. For this we re-invented the BlocksWorld domain, in a setting suited for the Komodo robot with as much of the emphasis placed on the ROSPlan Component involved.

The main work done to achieve this goal:

1. ROSPlan framework: The ROSPlan framework is still very new, and as such it’s feature set, stability and documentation are still lacking. This meant that much of the work was focused on getting around and over the framework’s limitations
2. Arm Control: To keep the project focused on ROSPlan and the komodo, additional components (i.e. MoveIt) were not included. This meant that arm control was done manually. Given the complexity of motion planning, this meant that we had to put some restrictions on arm movements and subsequently our example domain.
3. Arm limitations: Given the limited arm control, and the physical limitations of the Komodo robot (underpowered and inaccurate arm joints, missing degrees of freedom, etc.), some work was needed to allow reliable actions in varied settings

**ROSPlan overview**

The ROSPlan frameworks architecture is the following



**Planning System** – the planning system is responsible for three main things: constructing a problem using ROSPlan’s UI, forming a plan and replaning (using an external planner) and dispatching the plan.

The default planner packed with ROSPlan is POPF. A severe limitation found in that planner is not supporting negative preconditions (making inequality very difficult to express). Replacing the planner currently requires modifying the Planning System’s code to handle a new planner’s output format. This work can be made simpler by creating an improved parser which uses some formal syntax (e.g. regular expressions) to describe the planner’s format, but is not in the scope of this project.

Using the feedback from components and its replaning abilities, the Planning System can (and has done so in our tests) even solve some simple POND problems using a classical planner without any modifications.

Note: The version of ROSPlan available for download while writing these lines contains a bug in its planning system. This bug will cause the Planning System to crash while executing a plan on a domain with predicates that have no parameters. The version of ROSPlan installed on the robot was fixed as a part of this project, and hopefully the fix will be merged into the main ROSPlan branch soon.

**Knowledge Management System** – the KMS is responsible for parsing the input domain PDDL, managing the system’s data and notifying the Planning System if the current plan has become invalid (i.e. an upcoming action has unsatisfied preconditions). The KMS is comprised of two parts:

* Model (domain) data – The state of the world as described by the modeling language, including predicates, objects, functions, plan and goals. This data is updated by both the Planning System and by the various components in the system.
* "Real data" – For the plan to be executed, more information is needed on top of the abstract modeling data. This can be coordinates of cities, shape of a block, power in a battery and anything that a component requires to achieve its goal. This data is managed in a database (using the ROS message\_store node) and is mostly manipulated by the components.

**Components** – A component is a ROS node which implements one (or more) PDDL actions. The component translate modeled abstract actions to real-life actions. Components are responsible for defining success, failure, duration and outcome of actions but updating the KMS and real data.

Currently, ROSPlan is oriented towards C++, and so no documentation or work is done to allow python nodes to work in the system, but thanks to ROS topics and services system the ROSPlan framework is still seamlessly Python compatible, and once the documentation was translated to python, Python components were quite simple to create

**Domain**

This is our version of the BlocksWorld domain (in PDDL). The domain was simplified as much as possible (so not to draw to much attention). It has a single object type, block\_t, and two actions, pick\_up and put\_down. There are psudo-special block\_t objects which define the table positions on which block towers can be stacked. These blocks are signified by not being “on” any other block (and so can’t be move given the preconditions of the pick\_up action), and do not correspond to any physical block.

(define (domain blocks-domain)

(:requirements :strips :typing :disjunctive-preconditions)

(:types block\_t)

(:predicates

(inhand ?block - block\_t)

(emptyhand)

(not\_emptyhand)

(on ?block ?on\_block - block\_t)

(clear ?block - block\_t)

)

(:action pick\_up

:parameters (?block ?from\_block - block\_t)

:precondition (and (emptyhand) (clear ?block) (on ?block ?from\_block))

:effect (and (inhand ?block) (clear ?from\_block) (not (emptyhand)) (not\_emptyhand) (not (on ?block ?from\_block)))

)

(:action put\_down

:parameters (?block ?on\_block - block\_t)

:precondition (and (not\_emptyhand) (clear ?on\_block) (inhand ?block))

:effect (and (on ?block ?on\_block) (emptyhand) (not (not\_emptyhand)) (not (inhand ?block)) (not (clear ?on\_block)))

)

)

Because the default planner can't handle negative preconditions as input, we added a predicate called (not\_emptyhand) in order to require that the hand won't be empty when trying to put down a block.

**Arm Component (arm\_component.py)**

The arm component is responsible for both pick\_up and put\_down actions. It listens to messages on the “/kcl\_rosplan/action\_dispatch” topic to handle those actions during plan dispatch. Each such message defines a single PDDL action dispatch.

The planning system publishes these messages to all nodes in the ROS system, with each component deciding on its own if it needs to perform work in the real world in order for the system to achieve its abstract action.

**Initializing and Parsing the Domain**

In order to understand what the real state of the world is, our component queries the KMS in order to analyze parts of the domain. The needed information is what are the table positions ("table blocks") and where are all the blocks initially.

Getting the table positions is done by querying for all the blocks and then filtering out those which are not in the robot’s grip and are not “on” any other block. Formally:

Next we find all the blocks’ positions relative to those table blocks.

Afterwards we parse the configuration file, which is simply a list of “key:value” pairs with ‘#’ indicating commented lines. All pairs are simply stored in the dictionary for later retrieval. Note that the keys are unstructured (except that they mustn’t contain spaces), and so are sensitive to type’os or other errors without providing any feedback.

Last, the arm is moved into a natural position so not to disturb the block accidently during the initial positioning during the first action.

**Operation**

When handling a “pick\_up” or “put\_down” action we first query the database for the blocks we got as parameters in order to get their real world position. This data will be used to position the arm and its gripper around the block.

By nature, the components are platform (robot) specific, and in many cases are also domain depended, in our Arm component this can be seen in the dictionary of arm constants (arm\_component.py:29-45) which is populated during initialization from a configuration file (or, if no file is provided, it uses the default values of our Komodo robot). These values are used to calculate the real world position of a block, given its abstract position in the domain (which block is it “on”).

During “pick\_up”, once the arm is positioned in place, the robot’s fingers grip the block in a such a fashion that the block tower is supported against any trembles that may be caused by the gripping action. This is done by first allowing one finger to touch the tower while the other moves in to clamp on the block.

During “put\_down” the block is “dropped” from a small height (several millimeters to a couple of centimeters), and then the tower is stabilized by gripping the newly placed block and the one below it. This method greatly improved stability as well as prevented most errors from accumulating, allowing for longer and more complex plans without the need for complex feedback from the cameras or other sensors.

Once the actions is complete, the real data needs to be updated. Real data for the arm component is fairly simple. It contains only the positions of all blocks, as pairs of discrete indices relative to the table positions and blocks’ level (how high is the block currently stacked above the table). Both of these values are initially computed by parsing the input PDDL file during initialization and are then maintained during the life of the component.

If a block was picked up, its position is set to an arbitrary value to allow the robot to know it’s carrying a block. If a block was put down, its position is simply the position of the block on which it was placed but at a higher level.

Last task is to update the KMS. Since ROSPlan handles non-deterministic domains, it’s the component’s responsibility to tell the system what was the actual effect. This is done using the KomodoPicknPlaceComp.apply\_effects\_to\_KMS method (arm\_component.py:506-603) which in turn uses the ROSPlan KMS services.

**Actions** **Feedback**

When an action dispatch message is received and the component decides it needs to handle this action, the component sends a feedback message to the Planning System (using the "/kcl\_rosplan/action\_feedback" topic) to tell it that the action is being performed - “action enabled” feedback message.

If during action execution something goes wrong (which will happen mainly if connection to the database or KMS is severed), another feedback message is sent – “action failed”.

On the other hand, if all went well, after the real data and KMS databases are updated, “action achieve” feedback is sent, telling the Planning system that the next action can be dispatched (not that in some ROSPlan configuration concurrent actions are possible, in which case consecutive actions might have already been dispatched, providing that their preconditions are met).

**Arm Motion**

The arm component uses all degrees of freedom provided by the Komodo arm, except for two – the elevator and the first elbow joint are locked in place in order to (greatly) reduce the geometric complexity of the motion - reducing the motion to 2 dimensions motion around a circular 3rd axis, while calculating only simple triangles.

Picking up a block is composed of the following arm commands:

* Open the wrists fingers
* Position the arm in the specific table position and height. This is done in the following order:
  + Locking elbow1 in a static (0 rads) position (in case it was moved between actions)
  + Position arm base in the desired angle (according to the target table position)
  + Positioning the wrist exactly parallel to the block so the grip is accurate (based on the arm base position)
  + Position elbow2 to be exactly orthogonal to the ground (according to the target block’s height level)
  + Wait for joints to complete most• of their motion
  + Lower the shoulder to the correct height (according to the target block’s height level) and wait for the shoulder to complete its motion.
* Close the fingers in order to grip the block. Use the target block’s height to calculate the gripping angle. The fingers are closed asynchronously, starting with the finger closer to the robot, to ensure the tower’s stability. The gripping angle and finger time difference are calculated using the target block’s height.

Putting down a block is composed of the following arm commands:

* Position the arm in the specific table position and use a elevate the arm slight higher than the target to avoid slamming into the tower due to shoulder motor imprecision
* Release the grip. As before, the fingers are opened with a time differential to improve stability
* Restabilize the tower:
  + Reposition the arm at the correct (non-elevated) height
  + Grip the block
  + Release the grip
* Rise arm

• To allow smoother motion, joints are allowed to reach within 0.1rad of their target   
position. At this delta there’s almost no risk of accidentally damaging a tower, and the   
benefits are significant.

**Summary**

We managed to integrate the ROSPlan framework into the komodo robot, providing a live (and cool) proof of concept. Using this work, future components can be written for the robot with relative ease.

The current arm component can be further extended by integrating it with MoveIt, to allow for smother and more robust arm control and allowing for even larger domains.

Further improvements can be achieved by integrating the arm component with the sensors on the robot (mainly the camera), to decouple it further from the domain and allow it to find the initial configuration of the domain on its own.

Future ROSPlan plans can (and should) include a better problem definition system. Currently, using the ROSPlan UI, problems need to be formulated with ease run of the ROSPlan system, without a real option for providing them from an external file.  
Another possible upgrade is to allow more flexibility in planner selection, as mentioned before.

1. <https://github.com/KCL-Planning/ROSPlan> [↑](#footnote-ref-1)
2. <https://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/download/10619/10379> [↑](#footnote-ref-2)