H2020-ICT 731761

IMAGINE

Robots Understanding Their Actions by Imagining Their Effects

Deliverable D6.3
Planner software prototype for generalization

Lead Beneficiary:	CSIC	
Dissemination Level:	Confidential	
Due:	Month 38	(2020-02-29)
Authors:	Alejandro Suárez-Hernández	CSIC
	Javier Segovia-Aguas	CSIC
	Guillem Alenyà	CSIC
	Carme Torras	CSIC

This report documents Deliverable D6.3: a *software prototype for generalizing the planning task* that builds upon prototypes for action selection and replanning. This software is based on the *Automated Programming Framework* (APF) [2] which is used to generate algorithm-like solutions that generalize over a set of planning problems. These solutions, named generalized plans[1], are programs with *sequential instructions* that execute planning actions, *conditional gotos* that control the program flow and *terminal instructions* that terminates program execution. Furthermore, APF encodes the programming and execution tasks with the *Planning Domain Definition Language* (PDDL) which ease the integration with our previous software prototypes. Thus, the planning system follows the next procedure: (1) if there is no generalized plan, it computes a new one given a set of previous experienced planning problems; (2) if no previous experiences or the generalized plan is not found, the software for action selection is triggered; otherwise there is a generalized plan but (3.A) the monitored action fails, in which case the new experience is included in the training set and replanning is applied to compute a new generalized plan; or (3.B) the monitored action succeeds and the execution of the generalized plan continues until the goal condition holds in the terminal instruction.

V1.0 2020-03-23

1 Specification

1.1 Main track

In this Deliverable D6.3 we use previous software prototypes of action selection and replanning preserving and integrating their functionality in the new architecture for generalization. Thus, the new planning system consists of 3 modules named (i) *Action Selector*,(ii) *Replanner* and (iii) *Generalizer*.

1.1.1 Action Selector

Action Selector was described in the first and updated in the second deliverable where action costs are calculated using a *beta distribution* Beta(α, β) with previous experiences. The probabilities of each action are inversely proportional to the costs, which are used to feed a planner for searching the sequence of actions that solve and optimize the planning problem.

1.1.2 Replanner

Replanner module, described in the second deliverable, deals with unexpected outcomes searching for a new plan whenever the current state differs from the expected one after executing an action. This module extends *Action Selector* by monitoring successful and failing action execution cases, and improving the confidence on the former while decreasing it on the later. Also, it triggers a classical planner to search for a new plan given the current situation.

1.1.3 Generalizer

Generalizer is an independent module, in that either it solves the current planning problem or it uses previous modules if a general plan is not found. A general plan is an algorithm-like solution that applied in previous experienced disassembly tasks can solve all of them, so that can be applied on new tasks. The general plan consists of 3 different kinds of instructions: (1) planning actions, (2) conditional gotos, and (3) terminal instructions. Algorithm 1 shows an example of a general plan to unscrew all screws from an HDD, where the first line of the program chooses a specific screw to be unscrewed from a component, the second line goes back to the first line while there are screws in that fix components, and the third line terminates the program while validating the goal condition holds, which in this case is true since last line can be reached only if all screws have been unscrewed.

In order to compute general plans such as the one in Algorithm 1, it needs

do
  unscrew(?c - component ?s - screw);
while ∃*(?c - component ?s - screw) (and (fixed-by ?c ?s))*;
return ∀(?c - component ?s - screw) (and (not (fixed-by ?c ?s)));
Algorithm 1: General plan to unscrew all screws from components

the planning domain and at least one planning problem where all screws are unscrewed from all components, an upper bound to represent the maximum number of program lines, and a set of features that can be used as conditions in the conditional gotos, e.g. ∃(?c - component ?s - screw) (and (fixed-by ?c ?s)).

Then, we follow a compilation-based approach, where all the input data is compiled into a new planning domain and a single planning problem [3]. In this new problem, the general plan is empty and a classical planner should fill it with new instructions, like the ones in Algorithm 1, and execute it on the new problem to get the sequence of actions. Disjunctions of planning actions are allowed, but conditional gotos and terminal instructions must be unique when programmed in specific lines (no disjunctions allowed for them). Figure 1 shows how a planner with the general plan generates the following sequence of actions given a problem with one component (c1) and 3 screws (s1, s2 and s3) that fix it. However, the only relevant actions to execute are the ones that correspond to the original planning domain, in other words the subsequence of planning actions, e.g. the subsequence ⟨ (unscrew c1 s1), (unscrew c1 s2), (unscrew c1 s3) ⟩.

While execution of planning actions succeed, this module follows with the next planning action computed by the general plan. However, if the previous planning action failed during its execution and unexpected outcomes occur, then replanning is applied to find a new general plan to solve both the previous tasks and the new problem, and action costs are updated with the *Action Selector* module. In case this module fails on searching for a general plan, the *Replanner* module is triggered as a contingency.

2 Implementation

The main tool is implemented as a C++ application. The software suite is dependent on the Fast Downward planning system[1], which is distributed together with our prototype. A Python interface is included as well in order to integrate our work with our previous deliverable. We include instructions to compile and run a demonstrator that focus on the generalizer module.

[1] `http://www.fast-downward.org/`

```
(unscrew  c1 s1 )
(eval-exist-fixed-components ) ; True
(goto-line-1 )
(unscrew c1 s2 )
(eval-exist-fixed-components ) ; True
(goto-line-1 )
(unscrew c1 s3 )
(eval-exist-fixed-components ) ; False
(end-execution )
```

Figure 1: Classical plan generated from a general plan that unscrew all screws from fixed components.

Our tool takes as input a PDDL domain and a set of instances which represent past experiences. These are used to generate a controller. This controller takes the form of a program, with certain decision points meant to be settled by the planning system. This program:

1. Provides intuition over the structure of the solutions for a class of disassembly problems. A human operator can study these programs and understand the reasoning process of the planner to come up with solutions for new problems. These reasoning is updated whenever the environment evolves in an unexpected way.

2. Guides the planner effectively towards a plan for a particular instance, since the program imposes a common solution structure for a wide range of problems.

3. Gives an effective mechanism to monitor and react to changes in the current task, since the program is followed sequentially without replanning at each step.

An example of such program is shown at Table 1. This controller is obtained from a scene in which the robot is presented with its first hard drive. This hard drive is presented open and facing upwards.

Line	**Instruction(s)**
0	*switch-tool*(...)
1	*perfom-task-from-current-side*(...) $\vee$ *peek*(...)
2	*goto(0, !all-tasks-completed)*
3	*end*

Table 1: Simple controller obtained after seeing one problem

Being this the first hard drive that the robot sees, the robot is not aware of the components at the bottom side of the device, and thus there are no instructions to operate on the other side. This program means:

- Line 0: switches the current tool for another one where *(...)* is the list of arguments. The action to apply from the original planning domain is *switch-tool(?old ?new - tool)* where *?old* and *?new* are decision points where the planner is able to choose.
- Line 1: performs one of the available affordable tasks OR peek a different side of the device. Whenever a disjunction of instructions is programmed in the same line, the planner must choose and execute one of them.
- Line 2: proceeds to next line if all tasks have been completed, otherwise goes back to line 0. The robot will be done when there are no remaining tasks, i.e. components to retrieve, and all the sides of the device have been seen at least once.
- Line 3: checks the goal is achieved and the terminates with the program execution.

As the robot progresses in the disassembly of the device, it may discover hidden components that could not be seen at the beginning. As this happens, our implementation tries to re-plan from the most recent state using its current controller. If no plan can be found, a new controller is generated. This happens when the PCB is discovered at the bottom side of the hard drive, yielding the controller seen in Table 2.

Line	**Instruction(s)**
0	*switch-tool(...)*
1	*perfom-task-from-current-side(...)* $\vee$ *peek(...)* $\vee$ *flip-from-peeking-position(...)*
2	*goto(0, !all-tasks-completed)*
3	*end*

Table 2: Simple controller obtained after seeing one problem

This controller is different from the one shown in Table 1 in Line 1, where another instruction is present in the disjunction. This new instruction is *flip-from-peeking-position*, and is required to flip the device upside down so the robot can manipulate the other side (which is different from merely peeking).

Interestingly, peeking only requires rotating the robot platform 180 degrees. This is useful to take a look at the other side or perform minor tasks like

unscrewing. However, more difficult tasks (such as levering the PCB) require that the entire device is flipped, or otherwise the platform will hinder the movement of the robot. However, to this end it is necessary to have a stable suction surface (like the lid) so the device can be suspended from the SCARA arm. This means that if such surface is eliminated prematurely, (e.g. the lid is removed before the PCB, and the PCB is loose so it cannot serve as an anchor point), a deadend may arise. Such event also triggers a new program generation.

More illustrative examples are provided in a Jupyter notebook[2].

3 Availability

The software prototype this deliverable references is available as a release in the IMAGINE Github organization (`https://github.com/IMAGINE-H2020/imagine_planning/releases/tag/D6.3`).

The release has been marked with the tag `D6.3` so it can be easily identified for this deliverable.

The release comes with a `README.md` file that explains how to set up the package, and how to use it. In addition, it comes with a proof of concept demonstrator. The documentation states how to set up these components and execute a minimal example.

References

[1] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. "A review of generalized planning". In: *The Knowledge Engineering Review* 34 (2019) (cit. on p. 1).

[2] Javier Segovia-Aguas. *Automated Programming Framework*. `https://github.com/aig-upf/automated-programming-framework`. Accessed: 2019-11-12. 2017 (cit. on p. 1).

[3] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. "Computing programs for generalized planning using a classical planner". In: *Artificial Intelligence* 272 (2019), pp. 52–85 (cit. on p. 3).

[2] `https://github.com/IMAGINE-H2020/imagine_planning/blob/ms3_devep/csic_imagine_ros/csic_imagine_planning/csic_imagine_planner/d6_3/d6_3_demonstrator.ipynb`