Master Thesis

Felix Kolbe

Goal Oriented Task Planning for Autonomous Service Robots

Felix Kolbe

Goal Oriented Task Planning for Autonomous Service Robots

im Studiengang Master Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Thomas Thiel-Clemen

Abgegeben am 15.11.2013

Felix Kolbe

Thema der Masterarbeit
Zielorientierte Aufgabenplanung für autonome Service-Roboter

Stichworte
dynamische Aufgabenplanung, task level, autonome Robotik, Service-Roboter, GOAP, ROS, SMACH, RGOAP

Zusammenfassung
Mobile autonome Service-Roboter agieren in hochgradig dynamischen Umgebungen und benötigen ein geeignetes Planungssystem um ihre Aufgaben zuverlässig zu erfüllen. In dieser Arbeit werden Aufgabenplanungskonzepte für autonome Service-Roboter diskutiert. Zur Aufgabenplanung in der Robotik wird RGOAP nach dem Konzept der zielorientierten Aufgabenplanung (GOAP) entwickelt. Die Implementierung der Python-Bibliothek erfolgt in drei Paketen: ein eigenständiges Kernpaket, ein Adapterpaket für das Robotik-Softwaresystem ROS und ein Adapterpaket für die Zustandsautomatenbibliothek SMACH. RGOAP wird schließlich erfolgreich genutzt um die Selbständigkeit des mobilen Service-Roboters Scitos zu verbessern.

Felix Kolbe

Title of the master thesis
Goal Oriented Task Planning for Autonomous Service Robots

Keywords
dynamic task planning, task level, autonomous robotics, service robot, GOAP, ROS, SMACH, RGOAP

Abstract
Mobile autonomous service robots operate in a highly dynamic environment, requiring a capable high-level planning system to achieve their tasks. In this paper the characteristics of task planning systems for autonomous service robots are discussed. RGOAP, a robotic planning system based on the concept of goal oriented action planning, is developed. It is implemented as a Python library with three packages: a third-party-independent core package, an adapter package for the robotic software framework ROS and an adapter package for the state machine library SMACH. Finally, RGOAP is used successfully to add an autonomous behaviour to the mobile service robot Scitos.

Contents

List of Figures

Nomenclature

A*	A path finding algorithm	18
FSM	Finite State Machine	10
GOAP	Goal Oriented Action Planning	12
Planner	The procedure that tries to find a plan for a given goal	23
RGOAP	Robotic GOAP	16
ROS	Robot Operating System, the software framework used on the robot Scitos	6
ROS network	The computation graph of interconnected ROS processes, called nodes	7
Scitos	The robot the developed planning system is used with	5
SMACH	A state machine implementation	10

1 Introduction

Robots are meant to support humans. Whether they are mobile or lined up in a factory, their tasks are either too dangerous, too repetitive or too difficult for humans. The classification of service robots usually marks a mobile robot which operates in the surroundings of humans. For years service robots are cleaning floors, guiding visitors, carrying loads in factories or guarding institutions at night.

In contrast to the measurable and controllable environment present to a conventional factory robot, human environments form a difficult task for robots. To master this environment, typical service robots use several different types of sensors to perceive their surroundings.

Their mechanical capabilities improve every year, as does the computing power that is available within the requirements of size, weight and energy consumption of a mobile system. Obviously this heightens the expectations on autonomous robots, for example to handle unknown and changing environments. The worldwide occurrence of mobile service robots spread continuously and the increase is expected to continue in the next years (IFR Statistical Department, 2013).

Nowadays robots are designed to manoeuvre and cooperate tightly with people, for example to transport meals and medicine in clinics (Hägele et al, 2011, p. 9). To push this cooperation to a high level and to simplify the commanding of such a robot, a very autonomous and intelligent behaviour is needed. Especially when the performance of a task is interrupted by unexpected difficulties, robots ought to continue that task without any additional user interaction.

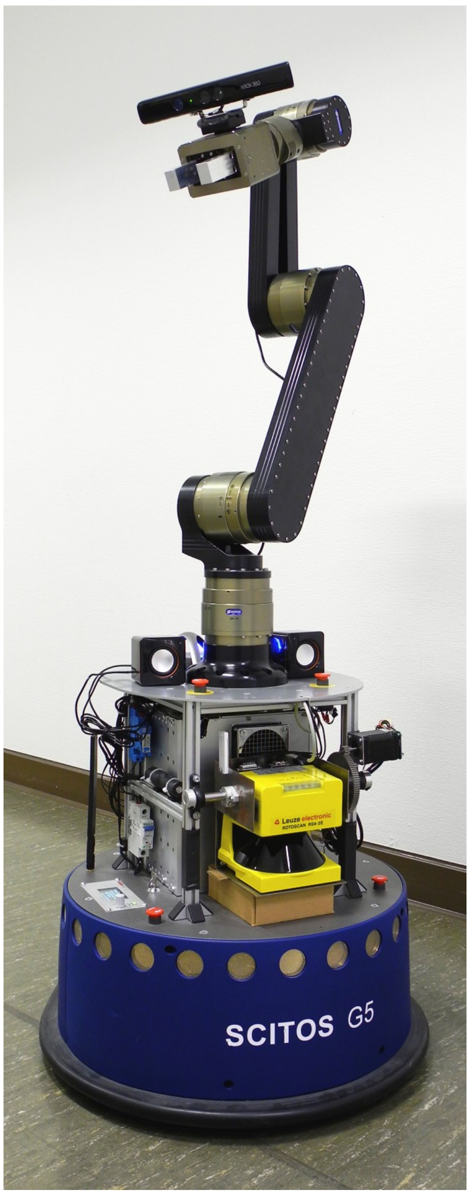


Figure 1.1: Mobile manipulation robot *Scitos*, Robot Vision Lab, UAS Hamburg

1.1 Development goals

The goal of this paper is the development and integration of a task planning component for service robots. The component should facilitate the service robot *Sctios* from the Robot Vision Lab at the University of Applied Sciences Hamburg (see figure 1.1 on the preceding page), that has no dynamic planning functionality yet, to achieve an autonomous behaviour.

Therefore tasks are needed that are suitable for the robot, given its capabilites and constraints. The robot should complete a given task even if unseen problems occur. Examplary scenarios are:

- If the current navigation path is suddenly blocked, a new path has to be found.
- If a human runs into the robot and the bumper is triggered, locking the drive unit, the robot should release its bumper and continue its plan or replan to reach the task's goal.
- If the manipulator has to be contracted into the robot's footprint to avoid collisions with the environment, or if directing the manipulator-mounted camera towards an obstacle would help in planning, the robot should be able to detect this and perform its actions accordingly.

1.2 Thesis structure

This paper's general challenge is described in the previous section. Hereafter the environment for the needed task planning system is introduced, including the robot and its capabilities (Chapter 2: Background). Next, existing task planning technics are explained (Chapter 3: Concurrent work). In the subsequent chapter requirements for a robotic task planning system are listed and classified (Chapter 4: Analysis & Design). After that the implementation of the developed planning system is detailed (Chapter 5: Development), including its usage with the robot Scitos (Chapter 6: Use case: the robot). Afterwards the compliance of the developed system to the requirements listed before is evaluated (Chapter 7: Evaluation). Finally the summarized and further improvements are given (Chapter 8: Conclusion & Outlook).

2 Background

In this chapter background information about planning systems in general is given. Also it further introduces the environment and the used robot for which a task planning system is to be developed.

2.1 Planning architectures

A program for a robotic system is usually split into different levels of capabilities, arranged in a hierarchical structure (Siciliano et al, 2009, p. 233).

For the task planning of mobile robots at least a *hybrid architecture* is used which combines a reactive with a deliberative layer (Uhl et al, 2007, p. 104). Here the low-level reactive layer controls the robot's actuators, obeying sensor input directly, for example to navigate around unexpected obstacles. The deliberative layer plans to achieve a higher-level goal and therefore instructs and monitors the reactive layer.

Often robot control programs are split into a *three level architecture* (Kortenkamp and Simmons, 2008): The highest one, the *planning* or *task* level, preserves the available tasks as well as the overall knowledge of the world and the robot. It selects one task to be followed and instructs the according part of the *executive* level to follow that task (Siciliano et al, 2009, p. 235). The executive level knows how to split that task into actions, including their ordering and timing. Accordingly, the *behavioural control* level controls the actuators to perform the actions given by the executive level.

A fourth *servo* level can additionally be defined as the lowest level (Siciliano et al, 2009, p. 234), though these low-level controllers are usually integrated in the hardware.

2.2 The robot Scitos

The use case of the task planner developed in this paper is the service robot *Scitos* (see figure 1.1) from the *Robot Vision Lab* at the *University of Applied Sciences Hamburg*. It is used as a development robot to examine and improve robotics, especially for service applications at home. The planning system has to function in this environment.

2.2.1 Current capabilities of the robot

The robot Scitos consists of a moving platform with several sensors and a manipulator on top. These give the robot the following capabilites:

- The platform has a differential drive allowing for two-dimensional movements and in-place rotation.
- A two-dimensional laser scanner and the drive's odometry provide data for environment mapping and self localisation. The software used is able to navigate the robot to a goal while avoiding obstacles.
- The manipulator has five joints and two fingers for pick and place operations. It can be controlled using kinematics and self collision detection.
- A Kinect[1] camera provides both color and depth images. It is mounted to the manipulator's gripper for a direct view over the objects to be manipulated. Panning the manipulator facilitates a three-dimensional perception of the robot's surroundings.

2.2.2 Current control architecture of the robot

For the robot algorithms on the lowest *behavioural control* level are available and functioning, like for instance trajectory following. The mid level *executive* can easily be built using the state machine framework SMACH (introduced in section 3.1.1). For the *task planning* level also a state machine could be used, though a state machine itself is constructed statically and does not support dynamic task planning.

2.3 Software & ROS

For programming the service robot Scitos and connecting all its software components the ROS framework (Quigley et al, 2009) is used. ROS is an open-source meta-operating system for any robot (ROS Wiki, 2013c) and began as a collaboration between the STAIR[2] project of the Artificial Intelligence Laboratory at Standford University and the *Personal Robots Program* at Willow Garage[3]. It features both various robot control libraries and many tools for package maintenance and process infrastructure analysis.

[1]Kinect is a color and depth camera device by Microsoft: http://www.xbox.com/kinect
[2]STAIR: STanford Artificial Intelligence Robot – http://stair.stanford.edu
[3]Willow Garage: a robotics research and development centrum – http://www.willowgarage.com/

2.3.1 ROS communication concepts

For a decent modularity every hardware and software component of the robot is integrated into the ROS Computation Graph[4] and accessed via standardised ROS message types and topics (channels), as illustrated in figure 2.1. The planning system will have to use these ROS interfaces, for instance to inquire the current position or to instruct a planning component.

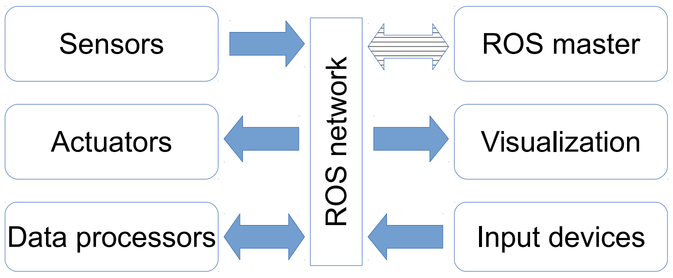


Figure 2.1: ROS Computation Graph, a peer-to-peer network with one central coordinator

In addition to stateless *message topics*, ROS provides *services*[5], which combine a request with a reply message (see figure 2.2). Furthermore the *actionlib* package provides *actions*[6] that combine a goal message with an interim feedback message and a result message. Therefore services and actions can be used for efficient and reliable control of subroutines, whereas the success of commands sent via messages has to be verified by listening on a separate message topic, if available.

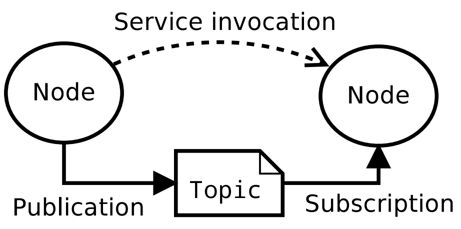


Figure 2.2: Basic ROS communication concepts *message topic* and *service* (ROS Wiki, 2013b)

2.3.2 Available interfaces on used robot

All component interfaces for the service robot are listed as follows, divided into sensors, data processors, actuators and visualization/input devices. These can be used to define actions

[4]ROS Computation Graph: peer-to-peer network of ROS processes – http://wiki.ros.org/ROS/Concepts
[5]ROS services: RPC-like request-reply-interactions – http://wiki.ros.org/Services
[6]ROS actionlib stack: standardized interface for interfacing with preemptable tasks – http://wiki.ros.org/actionlib

and conditions in the planning system. The more of them the planning system can perceive or control, the better it should perform.

2.3.2.1 Sensors

Odometry: Position via message topic and via coordinate frame infrastructure

Bumper status: Pushed and locked states via message topic

Laserscanner: Range data from front and rear laserscanner via message topic

Battery status: Charge state and charging state via *diagnostics* message topic

Camera/Kinect: Image via message topic, point cloud via message topic

Manipulator: Joint positions via message topic, pose via coordinate frame infrastructure

2.3.2.2 Data processors

Planar map: Online from laserscanner generated map via message topic and service

Spacial map: Online from kinect point cloud generated OctoMap[7] via message topic

2.3.2.3 Actuators

Bumper reset: Commandable via message topic

Platform movement:

- Directly commandable by means of translational and rotational velocity via message topic
- Controlled by means of target position in map frame via action

Manipulator movement:

- Directly by means of joint velocities, accelerations and positions via message topics
- Delegated by means of joint-wise target positions via service
- High-level controlled by means of target gripper pose and avoidance of self or environment collision via action

[7] OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees - http://octomap.github.io

2.3.2.4 Visualization and interaction devices

Computer: Visualization of robot data via RViz[8]

Game controller: Haptic, analog user input

Loudspeaker: Commandable via message topic

Speech synthesis: Commandable via message topic

Tablet: Visualization and user input

[8]RViz: 3D visualization tool for ROS – http://www.ros.org/wiki/rviz

3 Concurrent work

In the research and the development of robots and virtual agents several task planning and behaviour specifications have been created. This chapter gives an overview of a subset of existing methods related to robotics.

3.1 Finite State Machines

The historical common way of behaviour control are finite state machines (FSM). Using conditions a FSM switches between statically linked states. This can be used on any higher or lower level of control. Respective to the level the states can represent algorithm steps, actions or tasks.

There are a number of tools and libraries available for the development of state machines. One example is SMACH, which is integrated into the software framework used with the Scitos robot and described in the next section.

3.1.1 State Machine implementation: SMACH

SMACH[9] is a library for state machines written in Python by Bohren and Cousins (2010). In addition to being available as package `smach` in ROS, it has been extended with ROS-specific states in the package `smach_ros` (ROS Wiki, 2013a). Developed in 2010, it is now used in a wide number of projects (Meeussen, 2012).

Though SMACH is defined as a state machine, its states do not directly describe a full state of the environment, but rather a directed graph of interconnected activities. When the machine switches to a state, that state's activity code is executed. The activity code dictates the state's outcome, which leads to the next state, according to the statically linked states graph.

[9]SMACH is a contraction derived from StateMACHine and pronounced like "smash" (Bohren and Cousins, 2010).

Experience with SMACH

SMACH has been used for the robot to set up simple task sequences. For each kind of subtask, e.g. waiting for a message, calling an action service or waiting for user input dedicated `smach.State` subclasses exist or were written. The complex ones are parameterized when they are used. To prevent redundant parameterization all over the various task definitions, nearly every action got its own state class. Examplary actions are waiting for a goal message, opening or closing the gripper, navigating to a goal or moving the arm in a look-around pattern.

Composing a complex task based on such subtasks then simply means to add their states to a `smach.StateMachine` and choose which state should follow another state's outcome. Access to the state machine's `UserData` structure can be defined for each state to enable data handover where needed.

Example state

As SMACH will be important for a later part of this paper, this section gives a short look at how a custom `smach.State` is used. In the following example code a state is constructed that forwards a given `(x, y, yaw)` tuple via the ROS network to the action server that is responsible for navigating the robot platform to a given position.

Therefore the predefined `smach_ros.SimpleActionState` is extended to combine the necessary action message generation in one state:

```python
class MoveBaseState(smach_ros.SimpleActionState):
    """Calls a move_base action server with the goal (x, y, yaw)
        from userdata"""
    def __init__(self, frame='/map'):
        SimpleActionState.__init__(self, 'move_base', MoveBaseAction,
            input_keys=['x', 'y', 'yaw'], goal_cb=self._goal_cb)
        self.frame = frame

    def _goal_cb(self, userdata, old_goal):
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = self.frame
        goal.target_pose.header.stamp = rospy.Time.now()
        quat = tf.transformations.quaternion_from_euler(0, 0,
            userdata.yaw)
        goal.target_pose.pose.orientation = Quaternion(*quat)
        goal.target_pose.pose.position = Point(userdata.x,
            userdata.y, 0)
        return goal
```

Example state machine

The state machine graph 3.1 on page 15 shows a simple `smach.StateMachine` used to let the robot patrol between two given goal poses. Among others it uses the `MoveBaseState` explained before.

Though SMACH itself is functional and comfortable, it cannot be used for dynamic behaviour planning as the states representing actions are linked statically and cannot be freely ordered at runtime.

3.2 Task execution environment for robotics

The *task execution environment for robotics (teer)* is a library written in Python that uses tasks in the form of co-routines to implement a high-level executive (ROS Wiki, 2012). Teer is meant to be an alternative to SMACH. Accordingly, while facilitating the operation of parallel tasks, teer belongs to the executive layer and does not feature any high-level task planning itself.

3.3 Goal oriented action planning

The *goal oriented action planning (GOAP)* pattern was designed to improve the behaviour of non-player characters in computer games (Orkin, 2003) and originates in the *Stanford Research Institute Problem Solver (STRIPS)* (Fikes and Nilsson, 1971).

Though being designed for software agents, it can also be used for hardware agents, providing “benefits at both development and runtime” (Orkin, 2003, p. 4). For example, subtle behaviour improvements come for free when planning in real time (Orkin, 2005, p. 5). In unsuitable situations GOAP might not be able to determine a plan. But a plan returned by the planner is assured to be valid, in contrast to hand coded processes (Orkin, 2003, p. 5).

Different agents can simply use the same GOAP system by having different subsets of all actions available (Orkin, 2003, p. 5). This can likewise be used for different robots, cases of application or levels of behaviour complexity.

3.3.1 Details of GOAP

Goals define a desired change in the representational attributes of the agent or its environment (Orkin, 2003, p. 8).

The agent’s abilities are each defined as an action. *Actions* combine a functionality with additional information needed by the planner. These are preconditions, effects and cost metrics

(Orkin, 2003, p. 2). *Preconditions* must be valid before the action can be executed. *Effects* are changes to the world introduced by the action. *Cost metrics* allow the planner to find the optimal plan, i.e. the plan with least total cost of all possible plans.

A *plan*, which is determined at runtime by the planner, specifies a list of actions that change all involved attributes from their current state to the state defined in the goal. An example is shown in figure 3.2, where a plan leading from start to goal was found. Attributes not mentioned by the goal are ignored unless they get involved by actions.

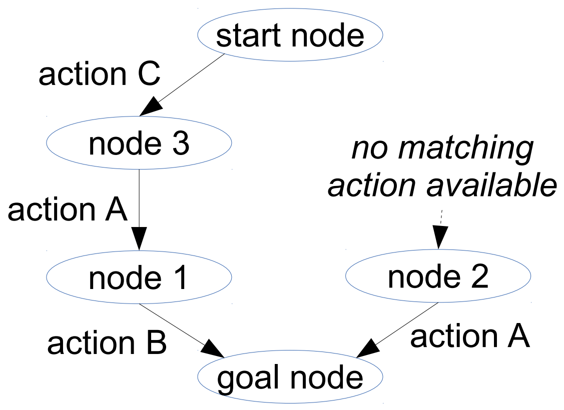


Figure 3.2: Simple example of a node graph containing one path between start and goal node

The functionality of an action can be any kind of code, but of course should target the actions declared effects. Whether those effects are assured or merely promised to be valid after performing the action depends on the implementation details. For example, in a virtual, abstract world where the action's effects could be totally reliable, specific functionality code would be obsolete. But in a real world, where any unforeseen event might disturb the functionality's outcome, it might be necessary to check whether the post action state matches the effects.

3.3.2 Existing GOAP implementations

There are at least two reference projects that implement a GOAP mechanism:

Thesis: Emotional GOAP Klingenberg (2010) used to simulate emotional agents. The development is based on GOAP by Orkin, extending it with emotional aspects. It is written in Java for an agent simulation environment.

Game library: pyGOAP Theden (2010) implemented GOAP in the context of Pygame[10], a game oriented Python library collection. Named *pyGOAP*, the module “tries to let virtual characters come alive through planning” (Theden, 2010).

[10]Pygame: a free set of Python modules designed for writing games - http://www.pygame.org/

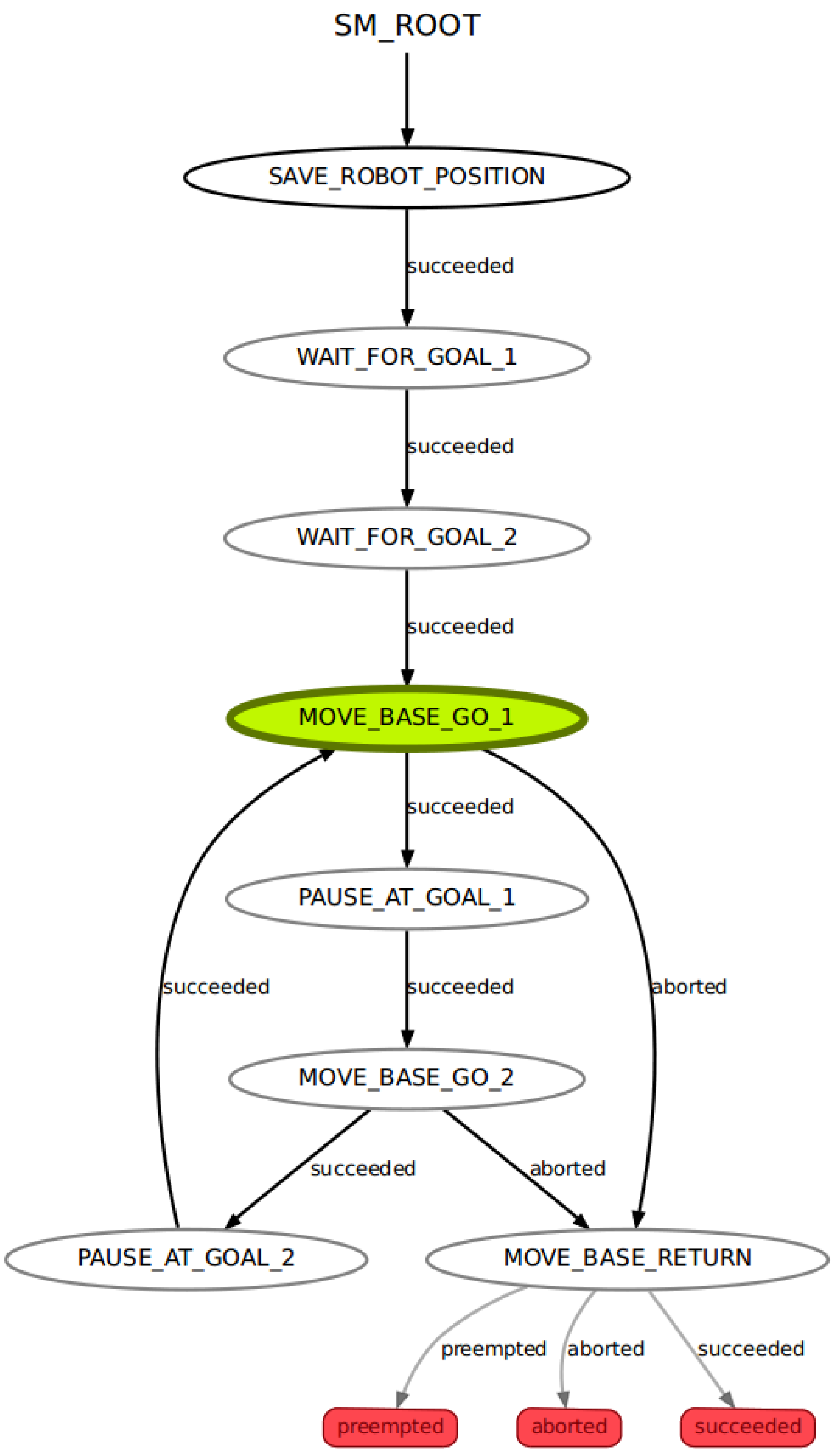


Figure 3.1: SMACH state machine example: patrolling between two goal poses – displayed using SMACH's own introspection viewer

4 Analysis & Design

In this chapter the functional and nonfunctional requirements for a robotic task planning system are explained.

The task planning system is developed using the *Goal Oriented Action Planning (GOAP)* concept. GOAP has been considered usable in various use cases (Orkin, 2005; Klingenberg, 2010; Theden, 2010). Providing an effective method to accomplish task planning, GOAP can be specifically designed to the problem.

For each requirement a design decision is conluded for the implementation of the Robotic Goal Oriented Action Planning system named **RGOAP**.

4.1 Functional requirements

There are a lot of mandatory criteria that a GOAP system must fulfill as well as optional variations and improvements. Which of these are useful and applicable for the intended operation purpose is determined in this section.

Orkin (2003, 2005) describes several criteria for a GOAP system in computer games. Most of them can be adapted to a real robot. Dini et al (2006) name a number of criteria a successful planning system should fulfill. They are intended for simulated agents in virtual worlds like a computer game, but most of them are applicable to a real robot, too.

The following sections enumerate criteria while classifying them as required, reasonable, optional or irrelevant: *Required* criteria are essential for a GOAP system to work. *Reasonable* criteria are not mandatory for a GOAP system, but seem to be of avail for a robotic GOAP system. Therefore they are expected to be more interesting than the *optional* criteria. Criteria rated as *irrelevant* are not applicable to this planning system's robotic use case or explicitly ignored in this development.

Note that the fundamental functioning of GOAP as described in the section 3.3 on page 12 is not explicitly repeated in these criteria but to be followed nonetheless.

4.1.1 Required criteria

The required criteria are split into the sections action definitions, precondition types, planning algorithm and cost metrics.

4.1.1.1 Action definitions

Variable actions Actions can be implemented in a discrete and a generic manner (Orkin, 2003, p. 10). For example, one action that can move the robot to any position is more useful than individual actions for each position in the environment. As a consequence that action's preconditions and effects need to be defined not only by discrete values but variables. This feature must be implemented as a robot must be able to handle the endless possibilities of the continuous, real world.

Context effect function Orkin (2005, p. 4) proposed an optional *context effect function* "which runs arbitrary code after the action completes execution". It is unclear whether this means that every action's effect is blindly applied symbolically when processing the action. In the real world it is uncertain whether the robot or its environment will end exactly as the action's effects promise. Therefore a method for arbitrary code is necessary and the list of effects might only be used for planning, but not blindly while executing a plan.

4.1.1.2 Precondition types

The preconditions, that are required for an action to be valid for executing, should be implemented as two different types of preconditions: *symbolic planner preconditions* and *freeform context preconditions* (Orkin, 2003, p. 10).

Symbolic planner preconditions These preconditions are the important ones for the GOAP planner, as they can be satisfied by other actions' (symbolic) effects. Both the symbolic planner preconditions and the effects have to match with their symbolic condition to the list of conditions concerned by the planner.

Freeform context preconditions These preconditions must also be satisfied for an action to work, but no other action could satisfy those purposively, so the planner cannot make use of them except for checking. Freeform context preconditions can be "any piece of code that results in a Boolean value" (Orkin, 2003, p. 10).

Though it would be better to have actions available to satisfy every kind of precondition, this feature is considered to be likely necessary. For example, in the distributed ROS network the connection to an action server can be checked to know if the action can actually be performed. Though that does not imply that an alternative solution would be available, separating context preconditions from symbolic preconditions seems valuable for future diagnostics.

4.1.1.3 Planning algorithm

Ignored conditions Those conditions that are semantically not relevant for the goal do not have to be included in the goal's list of preconditions. While planning for a goal, these irrelvant conditions should be ignored until they are involved in a helpful action.

Graph algorithm GOAP converts the problem of finding a list of actions to satisfy the goal to a graph theory problem. In that graph the optimal path, i.e. with minimum total cost between start node and end node, has to be found. For path finding many algorithms exist, among them *A**, which is widely used (Siciliano et al, 2009, p. 607) and usually faster than others (Matthews, 2002, p. 105). Orkin (2005), Theden (2010) and Klingenberg (2010) use A* to implement a GOAP system. Because the planning system – with its high-level nature – indeed has to be optimal, and the fact that A* is widely and successfully used, this development uses A* for the prototype.

Search direction An A* planner can be built to search progressively or regressively. For the GOAP system's graph "a regressive search is more efficient and intuitive" (Orkin, 2003, p. 7). Although this means that the robot cannot execute the first action before the full path is calculated, this is negligible as the task planning is expected to require much less time than the robot's movements.

4.1.1.4 Cost metrics

To find the optimal path in the planning graph the cost of a path must be quantifiable. The path can be split into the known part, i.e. the found path from the goal node to the current node (planning regressively), and the unknown part, i.e. the to-be-planned gap from the current node to the start node (Orkin, 2003, p. 7). The known part can be summed up from all nodes belonging to the found path leading to the current node. The unknown part must be estimated using some heuristics.

Node cost A node's cost can simply be transcribed from the node's action. While this needs every action to define a cost value, a default action cost can be used. Also, cost metrics can be used "to force A* to consider more specific actions before more general ones" (Orkin, 2005, p. 4).

Heuristics The heuristics estimate the cost between two nodes, specifically two world states. A very simple solution would be to take the number of their unsatisfied conditions (Orkin, 2005, p. 4) as the estimate.

Heuristics upper bound As one important criteria the heuristics must not overestimate the minimum path (Siciliano et al, 2009, p. 607). If they do, the algorithm might not yield the optimal path: a different path might be chosen which in total has higher costs than the overestimated path, but has lower costs than the estimated costs of that overestimated path.

4.1.2 Reasonable criteria

The reasonable criteria are split into the sections world representation, planning extensions and failure recovery.

4.1.2.1 World representation

Information attributes In Orkin (2005, p. 2) a *WorkingMemoryFact* contains various attributes aside the fact value, for example position, direction, stimulus, object and desire. The *WorkingMemory* provides query functions to search for facts that match certain attributes. Which information is needed for a robotic GOAP system has to be evaluated.

Confidence In addition, each knowledge fact can contain a confidence value (Orkin, 2005, p. 3) to improve planning with vague inputs. Dini et al (2006, p. 2) entitle this problem *partial observability* and refer to Ghallab et al (2004), resolving this with probability information.

4.1.2.2 Planning extensions

Action duration As the robot itself takes some time to complete actions involving robot movement, it might be helpful to integrate an estimated action duration into the cost metrics.

Action ordering The planner could be modified "to enforce orderings of e.g. two specific sequenced actions" (Orkin, 2005, p. 5). Although it might help in certain cases this seems to be inessential by now. But both action and goal definitions should not need to express orderings of specific actions.

Multiple action results Non-determinism of actions can be handled by naming more than one possible result, each with different probability (Dini et al, 2006, p. 2).

4.1.2.3 Failure recovery

Recovery planning If the robot's world is sufficiently complex and uncertain, a plan can become invalid during execution, which "motivates having an execution and monitoring system put on top of a simple planner" (Dini et al, 2006, p. 2).

The simplest method of recoverying from action failure is to simple replan with an updated world representation. Optionally a dynamic replanning could be implemented for a faster switch to a fallback/alternative plan.

Precomputed recovery policies In case of plan execution failure replanning is inevitable. This can simply be done when needed, accepted the pause while replanning. As an alternative, especially when dealing with highly dynamic environments, Dini et al (2006, p. 4) suggests to use precomputed policies for every possible post-action world state.

4.1.3 Optional criteria

The optional criteria are split into timing constraints and multiple goals.

4.1.3.1 Timing constraints

Real-time planning The planning system contemplated for the robot does not need to support any real time criteria. Planning will only happen on a high level, the actions which move the robot or its manipulator will resort to existing controllers that handle the movement from start to end.

Asynchronous precondition calculation For the same reason as for *real-time planning* it is not necessary to compute any expensive preconditions asynchronously as suggested in Orkin (2005, p. 2).

4.1.3.2 Multiple goals

For a complex robot with many capabilites it might not be possible to define one goal that fits every task. So a user has to activate the currently preferred goal from a list of concurrent goals. To automate this task and lessen the need for user interaction an arbitrator needs to select one of the available goals.

Goal generation To make the goals reactive to the current environment, “a sensor may generate a list of potential tactical positions” (Orkin, 2005, p. 1), for which goals can be created dynamically. Gordon and Logan (2004) control a game agent using dynamically generated goals with flexible goal prioritization depending on the agent’s situation.

Goal relevancy When choosing between multiple goals *relevancy* attributes can improve the decision towards the desired behaviour. This is also named *desirability* of goals (Dini et al, 2006, p. 3). That relevancy can be calculated by the goal itself (Orkin, 2003, p. 1).

Goal categories To handle different levels of autonomous behaviour goals can be categorized, for example as relaxed, investigative or aggressive (Orkin, 2003, p. 1). This might help the autonomous goal activator to choose the best category according to the current amount of user input.

4.1.4 Irrelevant criteria

Teamwork "Distributed plans and teamwork" (Dini et al, 2006, p. 2) are not relevant because in the robot's major use cases no other robot or planning agent is involved.

Authorability Dini et al (2006, p. 5) declare the *authorability* as whether those who will have to use the planning system to e.g. express a story in a computer game are able to use the system for their purpose. This paper's planning system does not concern any special usability and is, for now, targeted to be used by robot programmers, not anyone that might use a specific or simplified language.

4.2 Nonfunctional requirements

The developed planning system has to meet the following nonfunctional requirements.

4.2.1 Autonomous behaviour

In the end the task planning should enable the robot to behave more autonomous in choosing his actions and more robust to disturbances as described in section 1.1 Development goals.

4.2.2 For use within ROS

Because it will primarily be used by the robot running ROS, the planning system should follow the ROS developer guide and make use of various ROS infrastructure aspects like packages, communication, logging and parameters to ease the planning systems usage and maintenance.

4.2.3 For students' use

The resulting system has to be practically usable for future students of the lab working on hardware or software components for the robot. The students should be able to integrate those into the robot's behaviour.

4.2.4 Programming language

The ROS framework can be used within various languages[11], whereas only C++ and Python are fully supported and widely used within ROS.

The planning system should be implemented using Python[12]. Python is a common language for simple and fast but versatile programming. Also SMACH (introduced on page 10) is written in Python. Using the same language allows for easier code reuse or interconnection of RGOAP and SMACH if needed. The type flexibility and therefore compatibility of Python had also been a reason for SMACH to be developed in Python (Bohren and Cousins, 2010, p. 20).

[11] ROS client libraries: http://wiki.ros.org/Client Libraries

[12] Python: Programming language – http://www.python.org/

5 Development

This chapter explains the implementation of the *robotic goal oriented action planning* system RGOAP, following the requirements from the previous chapter. It is divided into three sections to explain the data classes, the control flow classes and the SMACH adaptor classes separately.

The following list gives an overview of the used terminology:

GOAP The concept of a goal oriented action planning

RGOAP The *robotic goal oriented action planning* developed in this paper

planner The procedure that tries to find a plan for a given goal

condition A fact in the environment relevant for task planning

world state A snapshot of all known or relevant conditions

planning graph The data structure constructed while planning, containing the simulated effects of every considered action. The graph structure is a directed, rooted tree.

nodes Every node in the planning graph contains the action considered in that step and the world state representing that action's effects

current node The node in the planning graph currently inspected by the planner

start node The node representing the actual environment

goal_node The node representing a world state concluded calculated from the goal definition

plan The found path, i.e. a list of actions, that converts the start world state to the goal world state

5.1 The need for an own implementation

The reference GOAP implementations (listed in 3.3.2 on page 13) do not match the defined criteria. The *Emotional GOAP* is written in Java, which is not the preferred choice (see requirement *programming language*). The library *pyGOAP* is built too game centric to be used in the robot's environment.

5.2 Package overview

The developed RGOAP library is split into packages. Those and other involved Python packages are:

rgoap The core of the developed planning system, independent of ROS, SMACH or any other third-party library

rgoap_ros ROS-related specializations of RGOAP classes

rgoap_smach Adapter classes to bridge RGOAP and SMACH

smach The core of SMACH, ROS independent

smach_ros Modules and specializations that interface between SMACH and ROS

package-less Classes listed without a package are exemplary, use case specific and meant to reside in a user-code package instead of a library package like those above

5.3 Data classes

The class diagram 5.1 shows the RGOAP classes representing the data structure.

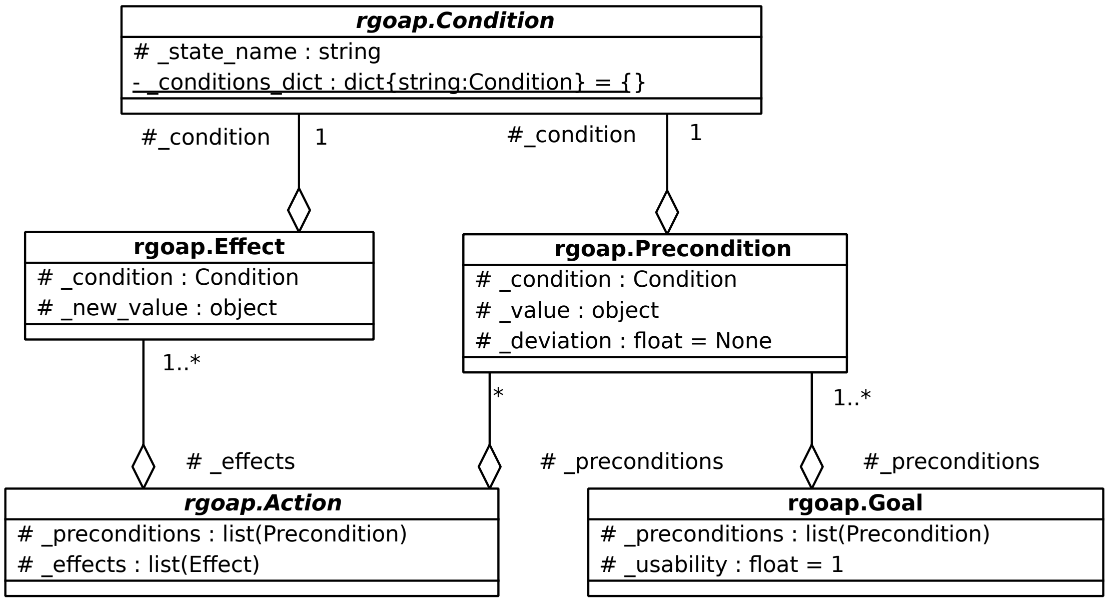
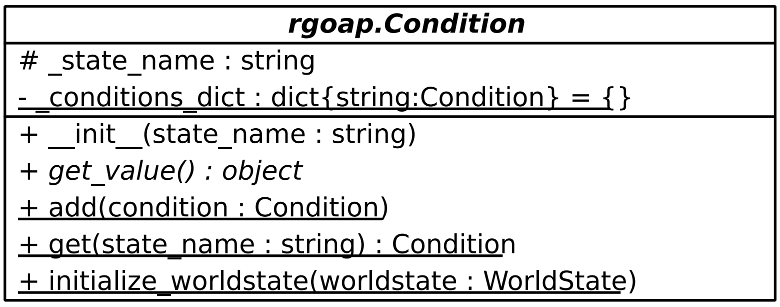


Figure 5.1: Class diagram of RGOAP classes representing data (methods not shown)

5.3.1 Conditions

Every aspect of the real world that the RGOAP planner has to consider is stored using a `rgoap.Condition` object. These conditions, known as predicate symbols in planning, are referred to when actions declare their preconditions and effects (see criterion SYMBOLIC PLANNER PRECONDITIONS). Each condition has a unique identifier name and a value.


rgoap.Condition
# _state_name : string
- _conditions_dict : dict{string:Condition} = {}
+ __init__(state_name : string)
+ *get_value() : object*
+ add(condition : Condition)
+ get(state_name : string) : Condition
+ initialize_worldstate(worldstate : WorldState)

For each condition identifier, for example `'robot.pose'`, exactly one `Condition` object may exist. A class scoped dictionary, mapping identifier strings to `Condition` instance objects, is used when preconditions and effects reference involved conditions. Conditions are retrieved through the syntax `Condition.get('robot.pose')`.

The value of a condition is not restricted to any specific type. Therefore setting and reading the value of a condition across preconditions, effects and actions has to follow a certain data type scheme. As a benefit it is possible to use ROS messages directly as the condition's value. This is convenient as the value is probably received via a ROS message, and later forwarded to another ROS node, e.g. via an action call. In such cases it would not help to unwrap and rewrap complex message types. For example, in this prototype implementation the pose of the robot is stored as a `geometry_msgs.Pose` object, rather than a `(x, y, yaw)` tuple.

This class has to be subclassed to implement the `get_value()` method. Generic subclasses are used to ease the integration of similar world aspects, depending on its source of information. For example, the class `rgoap_ros.ROSTopicCondition` easily reflects a field of a ROS message as its own value. Sample instantiations of `rgoap_ros.ROSTopicCondition` are:

```
# parameters: state_name, topic, topic_class, field
ROSTopicCondition('robot.pose', '/odom', Odometry, '/pose/pose')
ROSTopicCondition('robot.bumpered', '/bumper', Bumper, '/motorstop')
```

5.3.1.1 Data update

Condition objects are meant to update their internal value automatically, at the latest when `get_value()` is called. For example, the `ROSTopicCondition` updates with every message received and returns data from the last received message on `get_value()`.

5.3.2 World state

The global state of all known conditions is stored in the class `rgoap.WorldState`. It contains a private dictionary for every condition and their values, a getter and setter for that dictionary as well as helping methods for planning.

rgoap.WorldState
_condition_values : dict{Condition:object} = {}
+ __init__(worldstate : WorldState)
+ get_condition_value(condition : Condition) : object
+ set_condition_value(condition : Condition, value : object)
+ matches(start_worldstate : WorldState) : bool
+ get_unsatisfied_conditions(worldstate : WorldState)

5.3.3 Effects

Each effect must refer to a condition. There are two classes available to create effects: `rgoap.Effect` and `rgoap.VariableEffect`.

5.3.3.1 Static effects

Instances of `rgoap.Effect` hold a *value*. Triggering a static effect would set the referred condition to this value. For example the `ResetBumperAction` declares the following effect:

```
Effect(Condition.get('robot.bumpered'), False)
```

This action has the effect to set the condition `'robot.bumpered'` to false. The planner uses this information directly: If this condition is true in the start world state, but false in the goal worldstate, this effect would state that particular action as helpful, depending on further effects and preconditions.

rgoap.Effect
_condition : Condition
_new_value : object
+ __init__(condition : Condition, new_value : object)
+ matches_condition(worldstate : WorldState, start_worldstate : WorldState) : bool

5.3.3.2 Variable effects

The planning system must be able to handle the continuous nature of the robot's environment, see criterion VARIABLE ACTIONS. There are the following possible types of variable actions:

- The action can change a condition from any value to any other value (e.g. the robot's position)
- The action can change a condition from specific values to any other value (e.g. the start value must be somehow valid)
- The action can change a condition from any value to specific values (e.g. a device could have multiple (error) states, but only two of them can be activated intentionally)
- The action can change a condition from specific values to unrelated specific values (e.g. from every even to every odd value)
- The action can change a condition from specific values to related specific values (e.g. multiplying a condition's value)

Note that with the last type of variable action the effect is sensitive to the combination of both values, so they cannot be defined independently.

rgoap.VariableEffect
_condition : Condition
+ __init__(condition : Condition) + matches_condition(worldstate : WorldState, start_worldstate : WorldState) : bool # _is_reachable(value : object, start_value : object) : bool

Therefore instances of `rgoap.VariableEffect` do not hold a static value. Instead, the effect's applicability can be specified in two ways, depending on the use case:

Reachability of a variable effect When subclassing VariableEffect and overriding the following method, a variable effect can dictate whether it can reach a specific value from a given start_value:

```
VariableEffect._is_reachable(value, start_value)
```

For example, the `CheckForPathVarEffect` used by `MoveBaseAction` forwards both parameters `value` and `start_value` to the navigation algorithm server to check if there is a path available. If not overridden this method defaults to true, easily allowing for the any-to-any variable effect.

Dynamically created preconditions Every action declaring a variable effect has to override the following abstract method, which can be seen as the precondition-counterpart for variable effects.

```
Action._generate_variable_preconditions(
          var_effects, worldstate, start_worldstate)
```

There are two reasons this method is needed:

1. An action could have multiple variable effects, which valid values somehow depend on each other. In this method such actions can calculate combined valid start values for those effects as needed.
2. An action being that flexible cannot name preconditions statically because of endless possible valid values. In this method the action has to create all needed preconditions according to this specific use between the given worldstates. The number of preconditions is independent of the number of variable effects the action declares and could differ between situations.

The generated list of preconditions returned by the subclass is used by the Action base class in the same way as static preconditions declared by static actions. If an action uses a subclassed variable effect that already specifies its *reachability*, the method `_generate_variable_preconditions()` can create a precondition for the conditon's value from `start_worldstate`, as the variable effect already confirmed to be valid in this situation.

5.3.4 Preconditions

The class `rgoap.Precondition` represents the symbolic preconditions, implementing the criterion SYMBOLIC PLANNER PRECONDITIONS. A precondition refers to a condition, holds a value and optionally an allowed deviation factor. It is valid to a given worldstate, if the referred condition equals the value with respect to the deviation (if given).

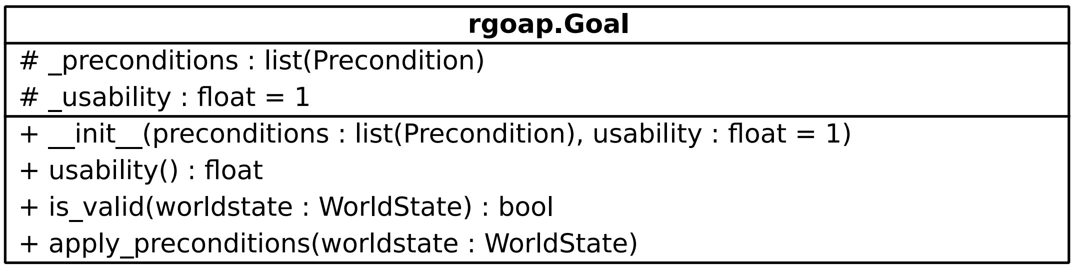
rgoap.Precondition
_condition : Condition
_value : object
_deviation : float = None
+ __init__(condition : Condition, value : object, deviation : float = None)
+ is_valid(worldstate : WorldState) : bool
+ apply(worldstate : WorldState)

For example the action class `MoveBaseAction` defines one precondition as:

```
Precondition(Condition.get('robot.arm_folded'), True)
```

5.3.5 Goals

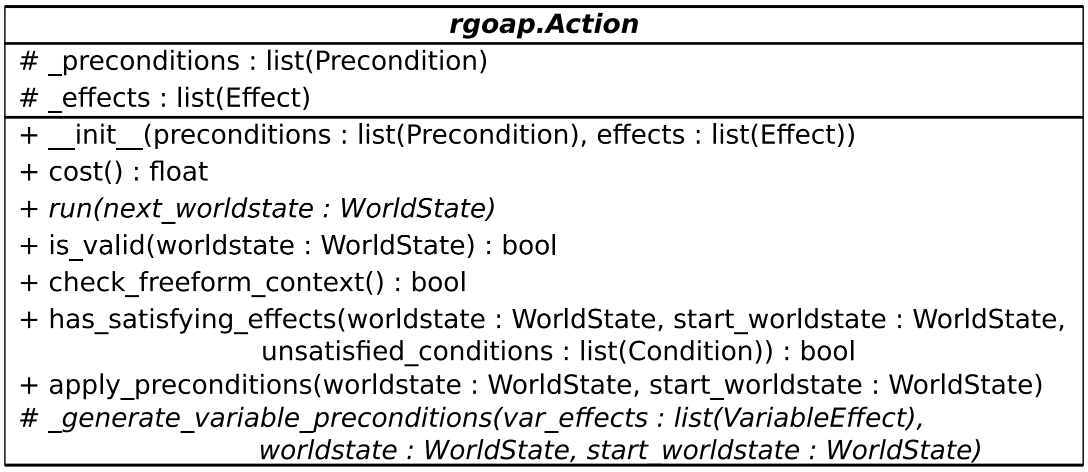
The class `rgoap.Goal` stores a list of preconditions that have to be satisfied for the goal to become valid. Also defined is a numerical value representing the usability of a goal (see criterion GOAL RELEVANCY).



```
rgoap.Goal
# _preconditions : list(Precondition)
# _usability : float = 1
+ __init__(preconditions : list(Precondition), usability : float = 1)
+ usability() : float
+ is_valid(worldstate : WorldState) : bool
+ apply_preconditions(worldstate : WorldState)
```

5.3.6 Actions

For every functionality of the robot that is relevant in an autonomous behaviour a `rgoap.Action` object is needed.



```
rgoap.Action
# _preconditions : list(Precondition)
# _effects : list(Effect)
+ __init__(preconditions : list(Precondition), effects : list(Effect))
+ cost() : float
+ run(next_worldstate : WorldState)
+ is_valid(worldstate : WorldState) : bool
+ check_freeform_context() : bool
+ has_satisfying_effects(worldstate : WorldState, start_worldstate : WorldState,
                         unsatisfied_conditions : list(Condition)) : bool
+ apply_preconditions(worldstate : WorldState, start_worldstate : WorldState)
# _generate_variable_preconditions(var_effects : list(VariableEffect),
                         worldstate : WorldState, start_worldstate : WorldState)
```

Each action stores its preconditions and effects. The class is abstract and has to be subclassed for every type of functionality.

The functionality (see criterion CONTEXT EFFECT FUNCTIONS) is implemented by overriding the method `run(next_worldstate)`. To add freeform context preconditions (criterion FREEFORM CONTEXT PRECONDITIONS) the method `check_freeform_context()`, which defaults to being valid (returning true), can be overridden. An action can change its cost value from the default cost by overriding the `cost()` method (see criterion ACTION DURATION), with respect to the restrictions detailed in section 5.4.2 on page 33. The functionality of the method `_generate_variable_preconditions`, which is only needed for variable actions, was described along *variable effects* in section 5.3.3.2.

5.3.7 Condition.set_value() vs Action.run()

The value of a condtion is retrieved via the condition itself, specifically by calling `Condition.get_value()`. In contrast, setting the value of a condition is done by the *context effect function* `Action.run()` which alters the environment directly (see criterion CONTEXT EFFECT FUNCTION).

An alternative approach would be to give conditions a setter method like `Condition.set_value(value)`. Instead of executing an action's context effect function, the action's effect would let the condition itself attain the new value.

Both alternatives have different pros and cons: If there is only one mechanism to set a condition, it would make sense to implement `Condition.set_value(value)`. In this case multiple actions having an effect on the same condition do not need to repeat that mechanism in their respective context effect function.

But if various mechanisms are used to set a condition (e.g. command a pose for the manipulator via trajectory action or a simple joint angle message), this cannot be handled by `Condition.set_value(value)`, but by implementing a separate action (with its own context effect function) for each of those mechanisms. This is why RGOAP uses a context effect function for actions rather than a value setter for conditions.

5.4 Control flow classes

The communication diagram 5.2 on the following page shows how the RGOAP classes `Planner`, `Introspector` and `PlanExecutor` are called by the central class `Runner`, which in turn is called by user code. In the following sections these classes are explained in detail. The *Tasker* shown is an example for any instance calling the *Runner* and not part of any RGOAP package (see section 6.4 on page 42).

5.4.1 Planner

The planner uses a regressive A* algorithm (see critera GRAPH ALGORITHM and SEARCH DIRECTION). As noted with the latter criterion, Orkin (2003, p. 7) states that a GOAP planner benefits from a regressive planner over a forwards stepping one. This was confirmed by implementing both type of planners until a certain point in development. As the forward planning was either

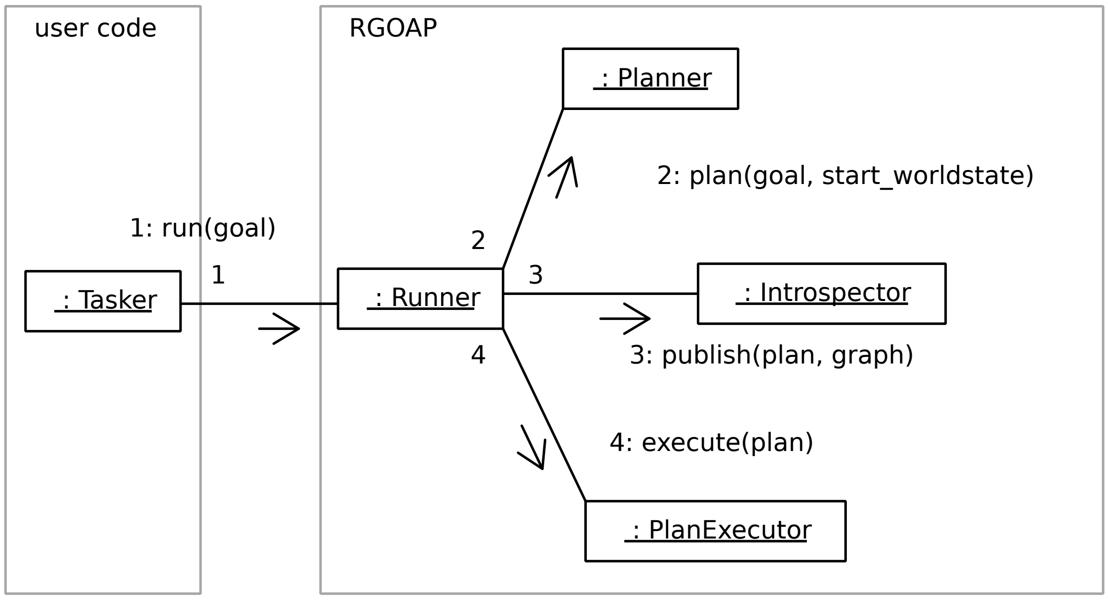


Figure 5.2: Communication diagram of RGOAP classes responsible for control flow

not successfull or its calculated graph of nodes was more complex and less goal oriented, all further development concentrated on the regressive planner.

The planner's inputs are the start world state, the goal (i.e. a list of preconditions) and a list of available actions. From the given goal a world state is derived. The basic idea of the planner is to find actions that match the difference between the start and goal world states. With each considered action a new world state is computed, which results from simulating the action's effects. Any condition contained in a world state is ignored until involved through a precondition or effect (see criterion IGNORED CONDITIONS).

The planner's output, if a plan was found, is a `Node` object that matches the start node. As every node contains a list of its parent nodes, the start node contains the list of action leading from the start world state to the goal world state. An exemplary plan is shown in figure 5.3 on the following page, in which the goal had one precondition that demands a change in the condition `'robot.pose'`. Every node is named after its action, its internal id and its cost values.

5.4.2 Nodes

The class `rgoap.Node` is used to represent the nodes in the computed A* tree. In figure 5.4 on page 33 a sample tree is shown which contains the path displayed in figure 5.3 on the following page). Because the planning is regressive the tree's root lies in the goal node which is constructed from the goal world state.

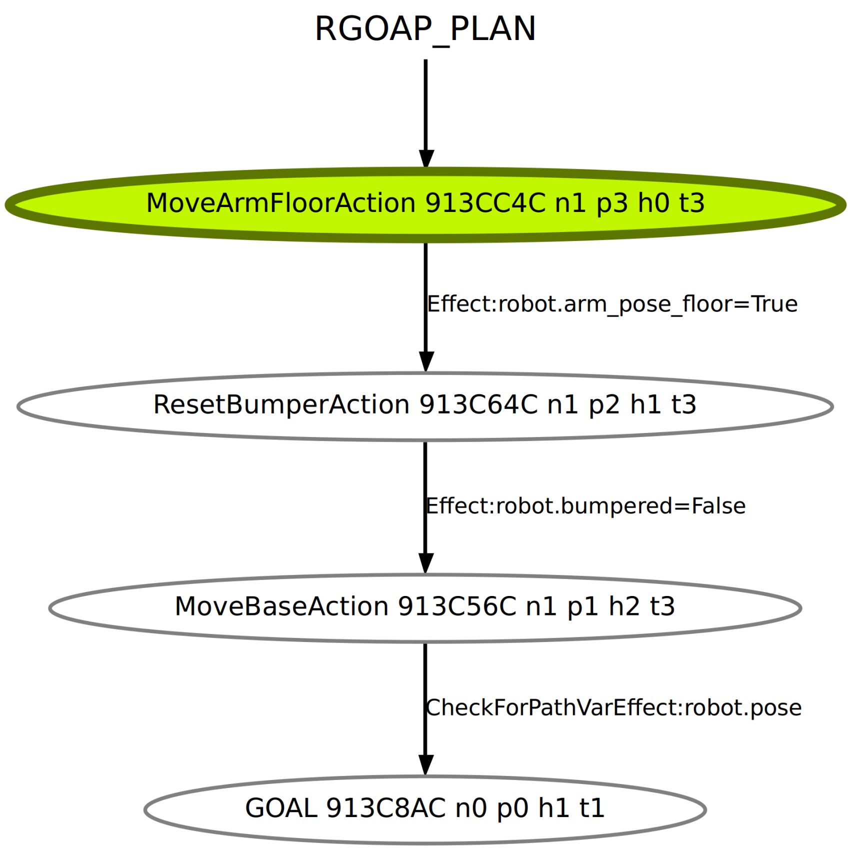

RGOAP_PLAN
MoveArmFloorAction 913CC4C n1 p3 h0 t3
Effect:robot.arm_pose_floor=True
ResetBumperAction 913C64C n1 p2 h1 t3
Effect:robot.bumpered=False
MoveBaseAction 913C56C n1 p1 h2 t3
CheckForPathVarEffect:robot.pose
GOAL 913C8AC n0 p0 h1 t1

Figure 5.3: RGOAP plan example – *n: node's own cost; p: path cost, summed from node to goal; h: heuristic between node and start node; t: total cost: path plus heuristic*

Each node stores the world state that the planner computed for that node, as well as the action that would change this node's world state to the world state of the parent node in the tree. Except for the goal node which – being the root node – has neither an action nor a parent node.

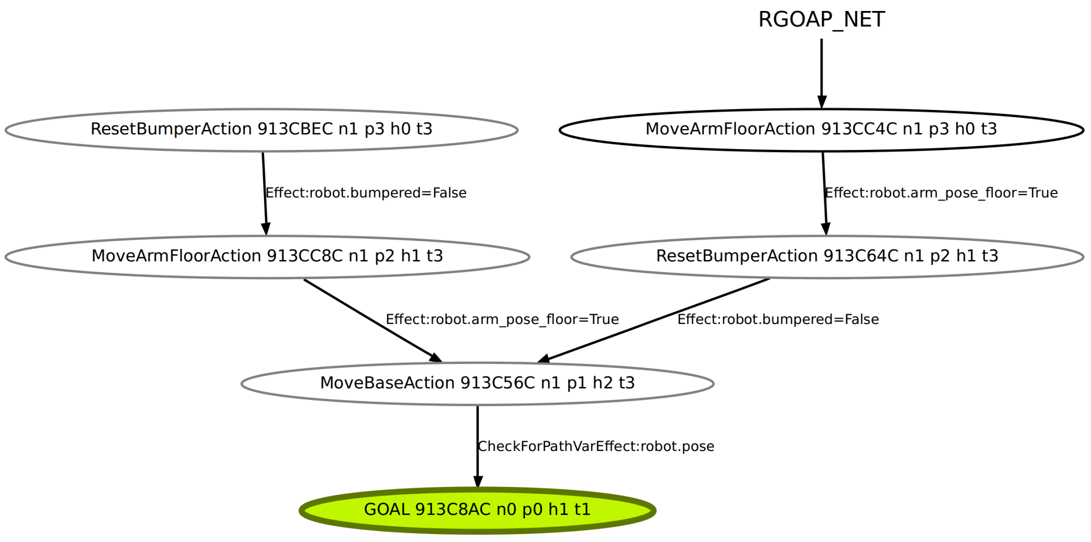

RGOAP_NET
ResetBumperAction 913CBEC n1 p3 h0 t3
MoveArmFloorAction 913CC4C n1 p3 h0 t3
Effect:robot.bumpered=False
Effect:robot.arm_pose_floor=True
MoveArmFloorAction 913CC8C n1 p2 h1 t3
ResetBumperAction 913C64C n1 p2 h1 t3
Effect:robot.arm_pose_floor=True
Effect:robot.bumpered=False
MoveBaseAction 913C56C n1 p1 h2 t3
CheckForPathVarEffect:robot.pose
GOAL 913C8AC n0 p0 h1 t1

Figure 5.4: RGOAP planning graph example – *n: node's own cost; p: path cost, summed from node to goal; h: heuristic between node and start node; t: total cost: path plus heuristic*

rgoap.Node
+ worldstate : WorldState + action : Action + possible_previous_nodes : list(Node) = [] - parent_nodes_path_list : list(Node) - heuristic_distance : float = None
+ __init__(worldstate : WorldState, action : Action, parent_nodes_path_list : list(Node)) + is_goal() : bool + parent_node() : Node + cost() : float + path_cost() : float + total_cost() : float - _calc_heuristic_distance_for_node(start_worldstate : WorldState)

Node costs and planning heuristics

The cost value of a node is derived from that node's action (see criterion NODE COST). The heuristic (see criterion HEURISTICS) is used by the planner to estimate the costs between two (unconnected) nodes. It compares those nodes' world states. The heuristic cost defaults to the number of unsatisfied conditions between those world states as the cost estimate.

As the heuristic must not overestimate the unknown cost (see criterion HEURISTICS UPPER BOUND), the cost of an action must be equal or greater than the number of conditions the action can change. Accordingly the method `Action.cost()` by default returns the number of declared effects, as every effect changes exactly one condition. Actions are allowed to override this method, but only to return a cost higher than the default.

5.4.3 Introspection

SMACH provides a comfortable introspection mechanism within ROS, named *smach viewer*. It displays the structure of any SMACH state machine graphically and updates in real-time with information about the currently active state and its data structure. This is shown in figure 3.1 on page 15.

As both the planning net and the found plan calculated by RGOAP are graph structures, it is possible to reuse the smach viewer to display them using the SMACH introspection viewer. An exemplary RGOAP plan is shown in figure 5.3 on page 32, an exemplary RGOAP planning net in figure 5.4 on the previous page.

5.4.4 Runner

The class `rgoap.Runner` simplifies the use of the RGOAP library. It holds all conditions and actions defined for the robot. The runner's methods receive a goal or a list of goals and update the worldstate, run the planner, publish the planning results via introspection, execute the plan and repeat the process until the goal is achieved (see criterion RECOVERY PLANNING). This manner is shown in sequence diagram 5.5 on the following page.

Multiple Goals

To create a continuous autonomous behaviour for the robot the class `rgoap.Runner` accepts a list of goals (see MULTIPLE GOALS criteria). First of all, the list is sorted by the goals' usability (see criterion GOAL RELEVANCY). If a plan is found for the most usable goal, that plan is executed. If no plan is found, or the execution was not successful (returning *aborted*), the second most usable goal is tested.

This is repeated until either no goals are left or a plan could be executed successfully. Only when the plan execution returns with *preempted* as outcome, the loop is terminated to forward the *preempted* state to the upper caller. As the preemption (see section 5.5.2 on page 37) is purposively done on user input it is important to observe this signal.

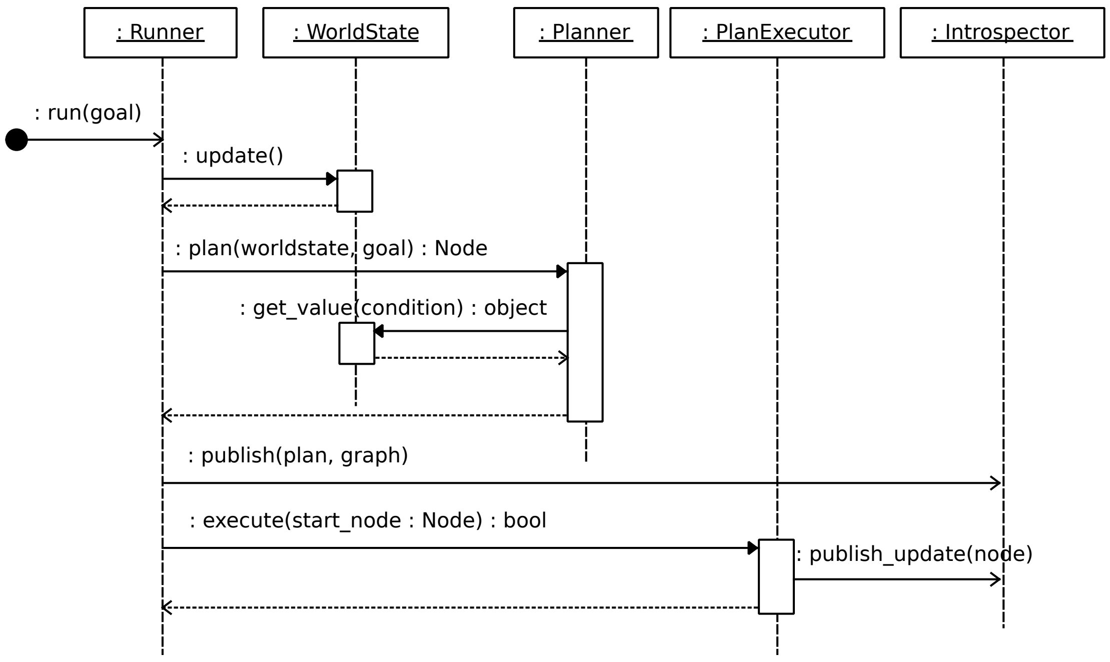


Figure 5.5: Sequence diagram of the *Runner* class planning and executing a goal

5.5 Connecting RGOAP and SMACH

The interface between RGOAP and SMACH consists of three parts:

- Integrating SMACH states in RGOAP, to reuse functionality written in states within RGOAP
- Executing RGOAP plans in SMACH, to use SMACH's execution behaviour with RGOAP plans
- Invoking the RGOAP planner from any regular SMACH state machine

They can be used independently as well as in combination.

5.5.1 Integrating SMACH states in RGOAP

Some functionality for the robot had previously been implemented using SMACH states and state machines. To avoid reimplementing existing functionality and code redundancy, functionality written in SMACH structures should be reusable within RGOAP. That is why the following subproblems have to be solved:

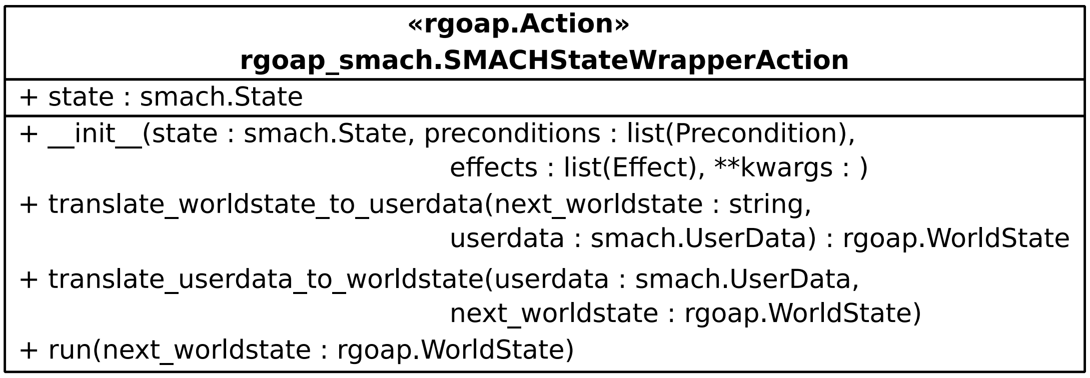
1. Wrap each SMACH state to a RGOAP action

2. Provide action details like preconditions, effects, context checks and a cost value
3. Convert the data between a `rgoap.WorldState` and a `smach.UserData`, for states that rely on data input to parametrize their functionality

The planner needs each action to have requirements and effects defined. These cannot be concluded automatically from the SMACH state's parameters or even its code. Therefore it is not possible to design a generic wrapper that can handle every SMACH state.

5.5.1.1 SMACHStateWrapperAction

The wrapper class `rgoap_smach.SMACHStateWrapperAction` inherits from `rgoap.Action` for being used in the RGOAP planning and wraps a `smach.State` that represents the actual functionality:



«rgoap.Action»
rgoap_smach.SMACHStateWrapperAction

+ state : smach.State

+ __init__(state : smach.State, preconditions : list(Precondition), effects : list(Effect), **kwargs :)
+ translate_worldstate_to_userdata(next_worldstate : string, userdata : smach.UserData) : rgoap.WorldState
+ translate_userdata_to_worldstate(userdata : smach.UserData, next_worldstate : rgoap.WorldState)
+ run(next_worldstate : rgoap.WorldState)

Wrapped in a `rgoap_smach.SMACHStateWrapperAction` any SMACH state can be considered by the RGOAP planner.

Simple use case: instantiation If the wrapped state does not need any data from the world state, `SMACHStateWrapperAction` can be used without subclassing. Only the state, effects and preconditions are required as shown in the following example:

```
fold_arm_action = SMACHStateWrapperAction(
        get_move_arm_to_joint_positions_state(ARM_POSE_FOLDED),
        [Precondition(Condition.get('arm_can_move'), True),
         Precondition(Condition.get('robot.arm_folded'), False)],
        [Effect(Condition.get('robot.arm_folded'), True)]
        )
```

The method `SMACHStateWrapperAction.run()` overrides the abstract method `Action.run()` to forward the call to the wrapped state's `execute()` method.

Complex use case: subclassing For more complex or communicative states the class `SMACHStateWrapperAction` has to be subclassed. A full example is listed in the appendix A.2 on page 55.

Subclassing allows to override the method `translate_worldstate_to_userdata()`. This is responsible for the conversion of data from the `next_worldstate`, which is the `rgoap.WorldState` the `SMACHStateWrapperAction` is planned to achieve, to the `smach.UserData` structure, which the state will use for data input. Accordingly via `translate_userdata_to_worldstate()` the state's output is translated back to the world state.

A dictionary, mapping condition identifiers to userdata fields, does not suffice as an alternative to those translation methods, because the datatypes used on both sides often do not match.

5.5.2 Executing RGOAP plans as SMACH container

Besides providing a comfortable visualization (see section 5.4.3 Introspection), the SMACH execution mechanism has one major advantage over the plan executor: While the latter one first checks, if every action in the path is valid to be executed and then executes it, SMACH also provides an ***preemption mechanism***:

From outside the executed state machine a preemption request can be expressed. Every state of that machine can, when active, check whether a preemption is requested. Usually only states whose *execute* method contains looping or waiting code check for the request. The request can be ignored or accepted. When accepted, the state returns *preempted* as its outcome, in contrast to *succeeded*, *aborted* or any other outcome declared by the state. If every surrounding SMACH state container passes the *preempted* outcome to the surrounding one, the state container on the root level returns with *preempted*.

To let SMACH execute a RGOAP plan the following subproblems must be solved:

1. Convert the plan to a structure executable by SMACH
2. Add every node in the path to that structure

Executable structure

The SMACH executor can operate on `smach.Container` implementations like `smach.StateMachine` or `smach.Sequence`. A `StateMachine` with the outcomes *succeeded*, *preempted* or *aborted* is used to represent the RGOAP plan. Figure 5.6 on the next page shows the SMACH container that is generated from the RGOAP plan shown in figure 5.3 on page 32.

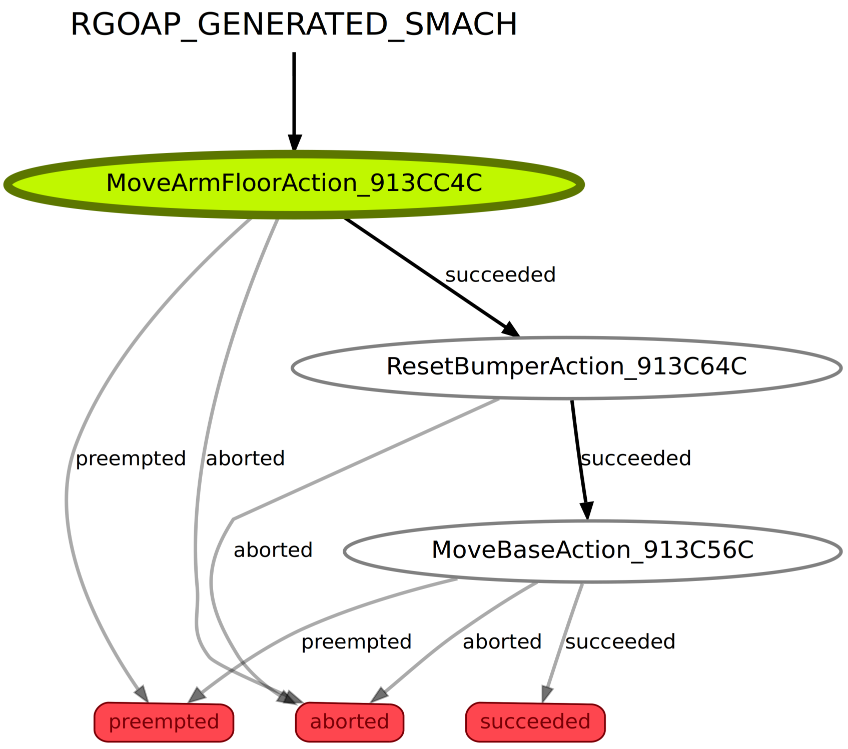
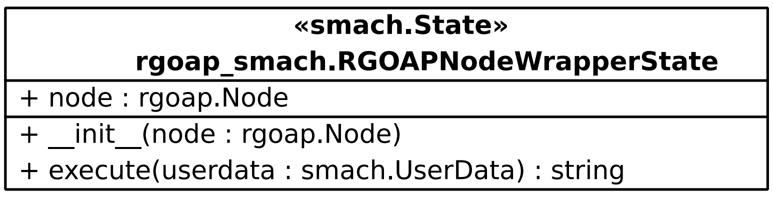


Figure 5.6: RGOAP generated SMACH state machine example

Add nodes to structure

For every node in the path that node's action can be of type `SMACHStateWrapperAction` (see section 5.5.1.1 on page 36), which means this action already wraps a SMACH state. This state is directly added to the state machine. In every other case the node has to be converted to a `smach.State` using the wrapper `rgoap_smach.RGOAPNodeWrapperState`.

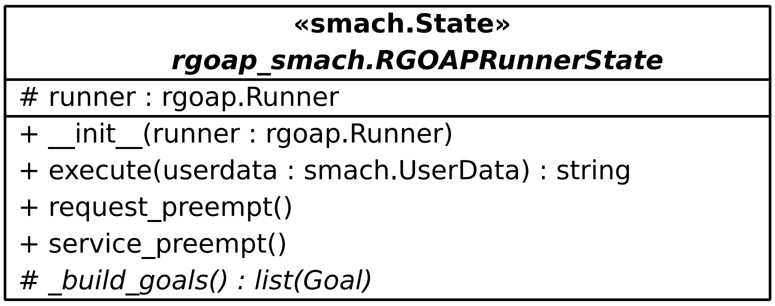

«smach.State»
rgoap_smach.RGOAPNodeWrapperState
+ node : rgoap.Node
+ __init__(node : rgoap.Node)
+ execute(userdata : smach.UserData) : string

The class `RGOAPNodeWrapperState` overrides the method `State.execute()` to forward the call to the `run()` method of the wrapped node's action.

5.5.3 Invoking RGOAP from SMACH

Being able to execute RGOAP plans using smach is essentially useful when combined with the ROS actionlib interface[13]. The `smach_ros.ActionServerWrapper` makes any given SMACH state container available to the ROS network via an action server. The action server can be commanded to execute its configured state machine by any ROS node. In addition, any ROS node can cancel the currently active action request. This preemption request (see section 5.5.2) is handled by the state machine, and eventually returned. An action server can return either *succeeded*, *preempted* or *aborted*, the outcome triple used by many action-centric SMACH states and state containers.

The class `rgoap_smach.RGOAPRunnerState` provides the missing link between this state machine and the RGOAP planning system. Using this wrapper the runner can be invoked in form of a SMACH state.


«smach.State»
rgoap_smach.RGOAPRunnerState
# runner : rgoap.Runner
+ __init__(runner : rgoap.Runner)
+ execute(userdata : smach.UserData) : string
+ request_preempt()
+ service_preempt()
# _build_goals() : list(Goal)

The abstract method `_build_goals()` has to be implemented in a subclass. Examples are listed in the next chapter. The methods `request_preempt()` and `service_preempt()` forward the preemption request from the enclosing state machine to the inner state machine that is planned and executed by the runner.

An interaction of the `RGOAPRunnerState` is visualized in the overview diagram 6.1 on page 43.

[13]ROS actionlib stack: standardized interface for interfacing with preemptable tasks – http://wiki.ros.org/actionlib

6 Use case: the robot

The developed planning system RGOAP has been validated using abstract tests. In this chapter RGOAP is used to improve the behaviour of the real robot *Scitos*. To connect the robot's interfaces (listed in section 2.3.2 on page 7) with the RGOAP library the following subclasses and instances have been composed.

6.1 Defined conditions

The following condition symbols have been used in the robot's use case.

6.1.1 MemoryConditions

A `rgoap.MemoryCondition` represents the value of a memory variable, which can be used for local, virtual conditions. In addition it can be used to mock a normal condition, which is convenient when writing tests. If no other element will change that condition's variable in memory, the condition returns the given value forever.

arm_can_move Provisionally defined condition mock. The arm-moving actions already depend on this being true.

awareness Provisionally defined condition mock. The variable is used as an abstract metric of the robot's awareness of its surroundings.

6.1.2 ROSTopicConditions

These conditions are created using `rgoap_ros.ROSTopicCondition` (see section 5.3.1 on page 25):

robot.pose representing the robot's pose in the map, which is the global base coordinate frame

robot.bumpered representing the boolean bumper state

robot.arm_folded represents whether the arm's pose reflects a certain folded pose

robot.arm_pose_floor represents whether the arm's pose reflects a certain pose pointing to the floor

6.2 Defined actions

The following actions have been used in the robot's use case:

6.2.1 Pure RGOAP actions

These actions directly subclass `rgoap.Action`.

ResetBumperAction Reset the bumper to reactivate the platform's drive.

6.2.2 Actions wrapping SMACH states

These actions subclass `SMACHStateWrapperAction` (see section 5.5.1.1 on page 36):

MoveBaseAction Command the base navigation to move the robot to a goal

LookAroundAction Pan the arm with the wrist-mounted camera around to perceive the surroundings (wrapping a `smach.StateMachine` that encloses a `smach.State` for each arm pose), increasing the `awareness` variable by one.

FoldArmAction Move the arm to a pose which is inside the robot's footprint for safe driving, placing the wrist-mounted camera at eye height.

MoveArmFloorAction Move the arm to a pose (inside the robot's footprint) that points the wrist-mounted camera to the floor in front of the robot for obstacle detection while driving.

6.3 Defined goals

The following goals have been used in the robot's use case:

6.3.1 Static goals

LocalAwarenessGoal Requires the `awareness` condition to increase. Used to trigger the `LookAroundAction`.

6.3.2 Generated goals

See criterion GOAL GENERATION for more information.

TaskPosesGoalGenerator Generates a `MoveToPoseGoal` for every goal pose given via user input. The usability of these goals decreases the older the user input is.

HectorExplorationGoalGenerator Sends a path planning request to an other process running in the ROS network, namely `hector_exploration_node`[14] which aims for exploring unknown parts in the map. If a plan is returned, a `MoveToPoseGoal` for the target pose of that plan is created.

RandomGoalGenerator Generates a `MoveToPoseGoal` for a given number of randomly computed poses around the robot's current position.

6.4 Tasker

The *tasker* is the central task control program for the robot Scitos. It is includes various tasks in an `smach_ros.ActionServerWrapper` to be controllable via any other ROS process. The tasks are SMACH states, whereas the ones of type `rgoap_smach.RGOAPRunnerState` activate the RGOAP planning system. One task, the *Autonomous RGOAP Loop*, features the goal generation described in the previous section.

An interaction overview of the RGOAPRunnerState is depicted in diagram 6.1 on the following page.

[14]hector_exploration_node: node providing exploration plans to unknown environments – http://wiki.ros.org/hector_exploration_node

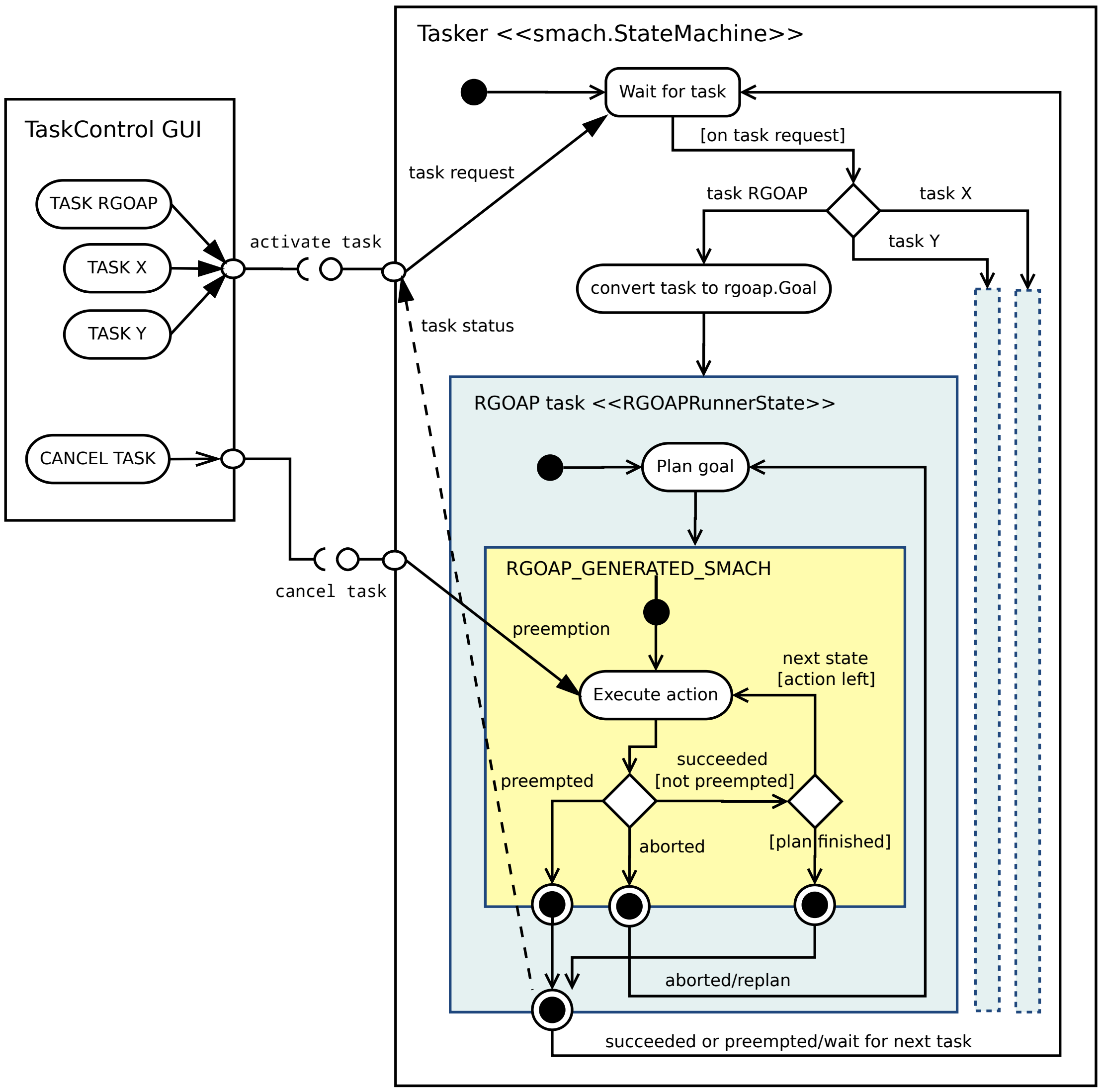


Figure 6.1: Interaction overview diagram of RGOAP

7 Evaluation

In this chapter the compliance of the developed planning system RGOAP to the requirements listed in chapter 4 is evaluated. In the analysis section several criteria for a robotic goal oriented action planning system have been described and classified. The following sections list in detail which of those have been implemented within RGOAP.

7.1 Functional requirements

The functional requirements have been met as follows:

7.1.1 Required criteria

All of the criteria classified as required have been implemented.

7.1.1.1 Action definitions

Variable actions Implemented

Implemented using variable effects and generated preconditions.

Context effect function Implemented

Implemented in method `rgoap.Action.run()`.

7.1.1.2 Precondition types

Symbolic planner preconditions Implemented

Implemented as instance of class `rgoap.Precondition`.

Freeform context preconditions Implemented

Implemented in method `rgoap.Action.check_freeform_context()`.

7.1.1.3 Planning algorithm

Ignored conditions Implemented

The planner ignores any condition not included by the goal's or action's preconditions or effects.

Graph algorithm Implemented

The RGOAP planner is realized as an A* algorithm.

Search direction Implemented

The implemented A* search proceeds regressively.

7.1.1.4 Cost metrics

Node cost Implemented

A RGOAP node takes its cost from the wrapped action.

Heuristics Implemented

A heuristic is calculated using the number of unsatisfied conditions.

Heuristics upper bound Implemented

The applied cost metric and heuristic calculation complies with this criterion.

7.1.2 Reasonable criteria

Some of the criteria classified as reasonable have been implemented.

7.1.2.1 World representation

Information attributes Not implemented in library

The implementation does not support corresponding attributes for worldstate data yet. But using appropriate data structures for the data type might suffice. For example, instead of using `geometry_msgs.Pose` objects for the condition `'robot.pose'`, the type `geometry_msgs.PoseStamped` could be used, which adds a time stamp and a coordinate frame identifier to the mere pose.

Confidence Not implemented

This feature has been omitted.

7.1.2.2 Planning extensions

Action duration Not implemented

RGOAP itself has no support for action durations. Actions defined by the user code (see chapter 6) can use any information when calculating an action's cost, but currently no duration is considered.

Action ordering Not implemented

This feature has been omitted.

Multiple action results Not implemented

This feature has been omitted.

7.1.2.3 Failure recovery

Recovery planning Implemented rudimentarily

The `Runner` class supports a try-again-loop, starting the planner with an updated world-state and the same goal. The planner itself does not assist in replanning.

Precomputed recovery policies Not implemented

This feature has been omitted.

7.1.3 Optional criteria

Some of the criteria classified as optional have been implemented.

7.1.3.1 Timing constraints

Real-time planning Not implemented

This feature has been omitted as the robotic use case (chapter 6) showed no need for this feature.

Asynchronous precondition calculation Not implemented

This feature was omitted for the same reason.

7.1.3.2 Planning extension: multiple goals

Goal generation Implemented in user code

The dynamic generation of goal objects is implemented not as part of the RGOAP library but as part of the user code.

Goal relevancy Implemented

For goal objects a usability has been implemented. It is used to weight generated goals.

Goal categories Not implemented

RGOAP has no support for goal categories itself. The user code (see chapter 6) can adjust the goals' usability attribute to prioritize different goal types.

7.1.4 Irrelevant criteria

None of the criteria classified as irrelevant have been implemented.

Teamwork Not implemented

This feature has been omitted.

Authorability Not implemented

This feature has been omitted.

7.2 Nonfunctional requirements

The nonfunctional requirements have been met as follows:

7.2.1 Autonomous behaviour

The robot Scitos gained an autonomous behaviour. It can master the exemplary scenarios described in section 1.1 Development goals. Using the multiple goals feature (see section 5.4.4) the robot is programmed to create, check and achieve new goals autonomously and independent from user input.

7.2.2 For use within ROS

The RGOAP Python packages are each put into a ROS package to streamline their usage inside ROS. The core package `rgoap` uses bare console prints for logging. When the ROS-specific package `rgoap_ros` is loaded, it overwrites the logging methods of rgoap with those from `rospy` to use the ROS logging infrastructure.

7.2.3 For students' use

RGOAP has not been used by anyone else yet.

7.2.4 Programming language

Python proved to be suitable for the RGOAP implementation.

8 Conclusion & Outlook

In this chapter known issues and possible further improvements of RGOAP are listed, subsequently this paper is recapitulated.

8.1 Known issues

The following issues have been discovered while using RGOAP:

8.1.1 Improve deviation handling

Preconditions in RGOAP can not only be valid to discrete values but also feature an optional deviation (see section 5.3.4 on page 28). This deviation is only considered when checking if a goal is valid (satisfied) or an action is valid to be executed in a world state. But the deviation is not considered while planning regressively when applying preconditions to a world state. To achieve this, a world state needs to hold not only a value for each condition, but also the summed up deviation factor.

8.1.2 Precondition-effect-symmetry

An action that defines a static effect, also has to define a precondition on the same condition. For example the `ResetBumperAction` declares to set the condition `'robot.bumpered'` to false, but only if the condition currently evaluates to true:

```python
class ResetBumperAction(Action):
    def __init__(self):
        Action.__init__(self,
                [Precondition(Condition.get('robot.bumpered'), True)],
                [Effect(Condition.get('robot.bumpered'), False)])
    # ...
```

Though the action does not care about the previous state of that condition, the planner needs the precondition to make the currently inspected world state approach the start world state. Instead of defining such an effect-precondition pair, the action indeed can declare a variable effect (see 5.3.3.2 on page 27), which suits the from-any-value-to-false effect in this case better, as the condition can be **True**, **False** or **None**. But then the action has to implement `Action._generate_variable_preconditions()`, only to copy the current value.

This has to be considered to make actions work. RGOAP should be modified to simplify usage and reduce error-proneness. Effects without a belonging precondition should be handled automatically.

8.1.3 Actions can change conditions accidentally

Actions can change everything in the environment from within their context effect function (see 5.3.6 on page 29). But changing something that is integrated into the RGOAP planner as a condition is problematic. For example, the `LookAroundAction` moves the robot's arm around, but does not specify at all in which pose it will reside afterwards:

```python
class LookAroundAction(SMACHStateWrapperAction):
   def __init__(self):
      SMACHStateWrapperAction.__init__(self,
            get_lookaround_smach(glimpse=True),
            [Precondition(Condition.get('arm_can_move'), True)],
            [VariableEffect(Condition.get('awareness'))])
   # ...
```

The plan executor checks each action that is about to be executed if it is valid in the current environment and aborts the execution when invalid. Though the runner (see 5.4.4 on page 34) will then replan for the same goal, which eventually will be achieved, it would be better if the planner could create a correct plan in the first place.

8.2 Further improvements

The following aspects are expected to improve RGOAP, but have not been implemented yet.

8.2.1 Use common data structure in RGOAP and SMACH

The RGOAP system could be built upon SMACH and reuse its data and control structures to avoid the additional data translation between those components. The structure `smach.UserData` is actually a dictionary made thread-safe, and has no restrictions on the values' types (Bohren and Cousins, 2010, p. 20). It could be similar enough to advocate for reusing this structure. To retain RGOAP's independence of any other third-party library an own but compatible structure could be used which eases data transfer.

8.2.2 Alternative planning algorithms

The *dynamic A* (D*)* algorithm extends A* with an efficient replanning (Stentz, 1995). When an executed action fails, the A* algorithm has to plan again for the new situation. D* can be more efficient when replanning from a newly void plan (Dudek and Jenkin, 2010, p. 206).

D* Lite by Koenig and Likhachev (2002) is an algorithmically different approach to solve replanning. It yields the same paths as D* but “appears to be even slightly more efficient”.

8.3 Ending

The goal of this paper was to develop a robotic task planning system for an autonomous service robot. A planning system that can be used with the service robot *Scitos* from the Robot Vision Lab and integrated into the robot’s current software environment was needed.

The requirements were listed and analysed, leading to design decisions that were observed in the implementation of RGOAP. This paper demonstrated the development of RGOAP as a GOAP-based task planning system which is implemented as a third-party-independent library in Python.

The supplementary package `rgoap_ros` provides ROS-specific subclasses to connect the planning system to messages and services from the ROS network. The classes from the additional `rgoap_smach` package facilitate the combination of RGOAP and SMACH, allowing to integrate RGOAP as part of existing SMACH user code, as well as the availability of RGOAP as a ROS-typical action server.

RGOAP has also been successfully used to give the service robot *Scitos* a more autonomous behaviour, whereby the robot continually searches for possible goals and tries to achieve them. Goals created from user input are prioritized over self-generated ones. Though RGOAP works well currently it can be further improved in functionality and usability.

Bibliography

[Bohren and Cousins 2010] BOHREN, J. ; COUSINS, S.: The SMACH High-Level Executive [ROS News]. In: *IEEE Robotics Automation Magazine* 17 (2010), Nr. 4, p. 18–20. – ISSN 1070-9932

[Dini et al 2006] DINI, Don M. ; VAN LENT, Michael ; CARPENTER, Paul ; IYER, Kumar: *Building robust planning and execution systems for virtual worlds*. Defense Technical Information Center, 2006. – URL `http://www.aaai.org/Papers/AIIDE/2006/AIIDE06-009.pdf`. – date visited: 2013-03-13

[Dudek and Jenkin 2010] DUDEK, Gregory ; JENKIN, Michael: *Computational principles of mobile robotics*. 2. ed. Cambridge [u.a.] : Cambridge Univ. Press, 2010. – ISBN 0-521-69212-1, 978-052-169-212-0, 978-052-187-157-0

[Fikes and Nilsson 1971] FIKES, Richard E. ; NILSSON, Nils J.: Strips: A new approach to the application of theorem proving to problem solving. In: *Artificial Intelligence* 2 (1971), Nr. 3–4, p. 189–208. – URL `http://www.sciencedirect.com/science/article/pii/0004370271900105`. – date visited: 2013-02-28. – ISSN 0004-3702

[Ghallab et al 2004] GHALLAB, Malik ; NAU, Dana ; TRAVERSO, Paolo: *Automated planning: theory and practice*. Amsterdam u.a. : Elsevier, Kaufmann, 2004 (The Morgan Kaufmann Series in Artificial Intelligence). – ISBN 1-55860-856-7

[Gordon and Logan 2004] GORDON, Elizabeth ; LOGAN, Brian: Game over: You have been beaten by a GRUE. In: *Challenges in Game Artificial Intelligence: Papers from the 2004 AAAI Workshop, AAAI Press, Menlo Park, CA*, URL `http://www.aaai.org/Papers/Workshops/2004/WS-04-04/WS04-04-004.pdf`. – date visited: 2013-10-29, 2004, p. 16–21

[Hägele et al 2011] HÄGELE, Martin ; BLÜMLEIN, Nikolaus ; KLEINE, Oliver: Wirtschaftlichkeitsanalysen neuartiger Servicerobotik-Anwendungen und ihre Bedeutung für die Robotik-Entwicklung. In: *Eine Analyse der Fraunhofer Institute IPA und ISI im Auftrag des BMBF, Fraunhofer Gesellschaft* (2011). – URL `http://www.autonomik.de/documents/EFFIROB_2011_07_21_72dpi_oI.pdf`. – date visited: 2013-10-09

[IFR Statistical Department 2013] IFR STATISTICAL DEPARTMENT: World Robotics 2013 - Executive Summary / IFR Statistical Department. URL `http://www.worldrobotics.org/uploads/media/Executive_Summary_WR_2013.pdf`. – date visited: 2013-10-26, 2013. – Research Report

[Klingenberg 2010] KLINGENBERG, Arne: *Prototypische Entwicklung eines emotionalen Agenten auf der Basis des Goal Oriented Action Plannings*, HAW Hamburg, Ph.D. thesis, february 2010. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2010/964/. – date visited: 2013-10-18

[Koenig and Likhachev 2002] Koenig, S. ; Likhachev, M.: Improved fast replanning for robot navigation in unknown terrain. In: *IEEE International Conference on Robotics and Automation, 2002. Proceedings. ICRA '02* Volume 1, 2002, p. 968–975 vol.1

[Kortenkamp and Simmons 2008] Kortenkamp, David ; Simmons, Reid: Robotic Systems Architectures and Programming. In: Prof, Bruno S. (Editor) ; Prof, Oussama K. (Editor): *Springer Handbook of Robotics*. Springer Berlin Heidelberg, january 2008, p. 187–206. – URL http://link.springer.com/referenceworkentry/10.1007/978-3-540-30301-5_9. – date visited: 2013-10-29. – ISBN 978-3-540-23957-4, 978-3-540-30301-5

[Matthews 2002] Matthews, James: Basic A* pathfinding made simple. In: *AI Game Programming Wisdom* (2002), p. 105–113

[Meeussen 2012] Meeussen, Wim: *Re: Current state of SMACH in ROS*. february 2012. – URL http://ros-users.122217.n3.nabble.com/Current-state-of-SMACH-in-ROS-tp3749748p3751229.html. – date visited: 2013-11-05

[Orkin 2003] Orkin, Jeff: Applying goal-oriented action planning to games. In: *AI Game Programming Wisdom* 2 (2003), Nr. 1, p. 217–227

[Orkin 2005] Orkin, Jeff: Agent architecture considerations for real-time planning in games. In: *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment* (2005). – URL http://www.aaai.org/Papers/AIIDE/2005/AIIDE05-018.pdf. – date visited: 2013-03-01

[Quigley et al 2009] Quigley, Morgan ; Conley, Ken ; Gerkey, Brian ; Faust, Josh ; Foote, Tully ; Leibs, Jeremy ; Wheeler, Rob ; Ng, Andrew Y.: ROS: an open-source Robot Operating System. In: *ICRA workshop on open source software* Volume 3, URL http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf. – date visited: 2013-06-04, 2009

[ROS Wiki 2012] ROS Wiki: *executive_teer - ROS Wiki*. march 2012. – URL http://wiki.ros.org/executive_teer. – date visited: 2013-03-15

[ROS Wiki 2013a] ROS Wiki: *executive_smach - ROS Wiki*. january 2013. – URL http://wiki.ros.org/executive_smach. – date visited: 2013-11-09

[ROS Wiki 2013b] ROS Wiki: *ROS/Concepts - ROS Wiki*. october 2013. – URL http://wiki.ros.org/ROS/Concepts. – date visited: 2013-11-13

[ROS Wiki 2013c] ROS Wiki: *ROS/Introduction - ROS Wiki*. october 2013. – URL http://wiki.ros.org/ROS/Introduction. – date visited: 2013-11-06

[Siciliano et al 2009] Siciliano, Bruno ; Sciavicco, Lorenzo ; Villani, Luigi ; Oriolo, Giuseppe: *Robotics: modelling, planning and control*. London : Springer, 2009 (Advanced textbooks in control and signal processing). – ISBN 978-1-8462-8641-4

[Stentz 1995] STENTZ, Anthony: The Focussed D* Algorithm for Real-Time Replanning. In: *Proceedings of the International Joint Conference on Artificial Intelligence* Volume 95, URL `http://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/s/stentz_D2.pdf`. – date visited: 2013-11-09, august 1995, p. 1652–1659

[Theden 2010] THEDEN, Leif: *pyGOAP - pygame - python game development*. february 2010. – URL `http://www.pygame.org/project-pyGOAP-1408-.html`. – date visited: 2013-10-18

[Uhl et al 2007] UHL, K. ; ZIEGENMEYER, M. ; GASSMANN, B. ; ZÖLLNER, J. M. ; DILLMANN, R.: Entwurf einer semantischen Missionssteuerung für autonome Serviceroboter. In: BERNS, Karsten (Editor) ; LUKSCH, Tobias (Editor): *Autonome Mobile Systeme 2007*. Springer Berlin Heidelberg, january 2007 (Informatik aktuell), p. 103–109. – URL `http://link.springer.com/chapter/10.1007/978-3-540-74764-2_16`. – date visited: 2013-10-29. – ISBN 978-3-540-74763-5, 978-3-540-74764-2

A Code examples

A.1 Subclassing VariableEffect

The class `CheckForPathVarEffect` from the following listing subclasses `VariableEffect` to override `_is_reachable()`, in which `move_base`, the ROS node responsible for two-dimensional navigation, is queried if a path is available.

```
class CheckForPathVarEffect(VariableEffect):
   def __init__(self, condition):
      VariableEffect.__init__(self, condition)
      self.service_topic = '/move_base/make_plan'
      self._service_proxy = rospy.ServiceProxy(self.service_topic,
         GetPlan)
      self._planned_paths_pub =
         rospy.Publisher('/task_planning/goal_paths', Path)

   def _is_reachable(self, value, start_value):
      request = GetPlanRequest()
      request.start.header.frame_id = '/map'
      request.start.pose = start_value
      request.goal.header.frame_id = '/map'
      request.goal.pose = value
      request.tolerance = 0.2 # meters in x/y
      response = self._service_proxy(request)
      self._planned_paths_pub.publish(response.plan)
      return len(response.plan.poses) > 0
```

A.2 Subclass of SMACHStateWrapperAction

The class `MoveBaseAction` from the following listing extends from `SMACHStateWrapperAction` to reuse the functionality of an existing SMACH state. This wrapped state is of type `MoveBaseState`. This action uses one variable effect `CheckForPathVarEffect` to potentially reach every possible pose. The pose data is translated to the `smach.UserData` structure, which is passed to the wrapped state. In its `check_freeform_context()` method the connection to the navigation node is checked.

```python
class MoveBaseAction(SMACHStateWrapperAction):

   class CheckForPathVarEffect(VariableEffect):
      def __init__(self, condition):
         VariableEffect.__init__(self, condition)
         self.service_topic = '/move_base/make_plan'
         self._service_proxy =
            rospy.ServiceProxy(self.service_topic, GetPlan)
         self._planned_paths_pub =
            rospy.Publisher('/task_planning/goal_paths', Path)

      def _is_reachable(self, value, start_value):
         request = GetPlanRequest()
         request.start.header.frame_id = '/map'
         request.start.pose = start_value
         request.goal.header.frame_id = '/map'
         request.goal.pose = value
         request.tolerance = 0.2 # meters in x/y
         response = self._service_proxy(request)
         self._planned_paths_pub.publish(response.plan)
         return len(response.plan.poses) > 0

   def __init__(self):
      self._condition = Condition.get('robot.pose')
      self._check_path_vareffect =
         MoveBaseAction.CheckForPathVarEffect(self._condition)
      SMACHStateWrapperAction.__init__(self, MoveBaseState(),
                  [Precondition(Condition.get('robot.bumpered'),
                     False),
                   Precondition(Condition.get('robot.arm_pose_floor'),
                      True)],
                  [self._check_path_vareffect])

   def check_freeform_context(self):
      if not
         self.state._action_client.wait_for_server(rospy.Duration(1)):
         rospy.logwarn("%s context check: cannot access move_base
            action server"
                   % self.__class__.__name__)
         return False
      try:
         self._check_path_vareffect._service_proxy.wait_for_service(1)
      except rospy.exceptions.ROSException:
         rospy.logwarn("%s context check: cannot access %s service
            server"
                   % (self.__class__.__name__,
                        self._check_path_vareffect.service_topic))
            return False
        return True


    def _generate_variable_preconditions(self, var_effects,
        worldstate, start_worldstate):
        effect = var_effects.pop() # this action has one variable
            effect
        assert effect._condition is self._condition
        precond_value =
            start_worldstate.get_condition_value(Condition.get('robot.pose'))
        return [Precondition(effect._condition, precond_value, None)]

    def translate_worldstate_to_userdata(self, next_worldstate,
        userdata):
        goal_pose =
            next_worldstate.get_condition_value(Condition.get('robot.pose'))
        (_roll, _pitch, yaw) =
            tf.transformations.euler_from_quaternion(
                    pose_orientation_to_quaternion(goal_pose.orientation))
        userdata.x = goal_pose.position.x
        userdata.y = goal_pose.position.y
        userdata.yaw = yaw
```

Versicherung über Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15.11.2013 ____________________________