Sustainable and reliable robotics for part handling in manufacturing

Project no.:	**610917**
Project full title:	**Sustainable and reliable robotics for part handling in manufacturing**
Project Acronym:	**STAMINA**
Deliverable no.:	**D3.3**
Title of the deliverable:	**Report on integration of robot skills**

Contractual Date of Delivery to the CEC:	**31.10.2016**
Actual Date of Delivery to the CEC:	**31.10.2016**
Organisation name of lead contractor for this deliverable:	**AAU**
Author(s):	**Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios S. Polydoros, Bjarne Großmann, Ronald P. A. Petrick, Volker Krüger**
Participants(s):	**P01, P05, P07**
Work package contributing to the deliverable:	**WP3**
Nature:	**R**
Version:	**0.6**
Total number of pages:	**42**
Start date of project:	**01.10.2013**
Duration:	**42 months – 31.03.2017**

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 610917

Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

We will show how SkiROS offers the possibility to integrate different functionalities in form of skills or "Apps" and how SkiROS offers services for integrating these skills into a consistent workspace. Furthermore, we will show how these skills can be automatically executed based on autonomous, goal-directed task planning. SkiROS helps the developers to program and port their high-level code over a heterogeneous range of robots, meanwhile the minimal Graphical User Interface (GUI) allows non-expert users to start and supervise the execution. As an application example, we present how SkiROS was used to vertically integrate a robot into the manufacturing system of PSA Peugeot-Citröen. We will discuss the characteristics of the SkiROS architecture which makes it not limited to the automotive industry but flexible enough to be used in other application areas as well. SkiROS has been developed on Ubuntu 14.04 LTS and ROS indigo and it can be downloaded at https://github.com/frovida/skiros. A demonstration video is also available at https://youtu.be/mo7UbwXW5W0.

This mainmatter of this document has been accepted for publication as a book chapter in *Robot Operating System (ROS) The Complete Reference (Volume 2)* under the title ***SkiROS – A skill-based robot control architecture on top of ROS.***

Keyword list: SkiROS, Robot Skills

Document History

Version	Date	Author (Unit)	Description
0.1	Sep 15, 2016	Volker Krüger	Draft for review
1.0	Oct 25, 2016	Volker Krüger	Final version approved

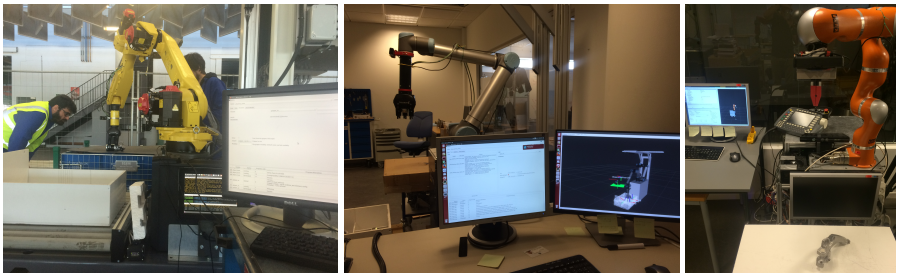


Fig. 1. SkiROS and the kitting pipeline ported on 3 heterogeneous industrial mobile manipulators.

1 Introduction

In robotics the ever increasing level of system complexity and autonomy is naturally demanding a more powerful system architecture to relieve developers from reoccurring integration issues and to increase the robot's reasoning capabilities. Nowadays, several middleware-based component platforms, such as ROS, are available to support the composition of different control structures. Nevertheless, these middlewares are not sufficient, by themselves, to support the software organization of a full-featured autonomous robot.

First, the presence of a knowledge-integration framework is necessary to support logic programming and increase software composability and reusability. In traditional robotic systems, knowledge is usually hidden or implicitly described in terms of if-then statements. With logic programming, the knowledge is integrated in a shared semantic database and the programming is based on queries over the database. This facilitates further software compositions since the robot's control program does not need to be changed, and the extended knowledge will automatically introduce more solutions. Also, reusability is improved because knowledge that has been described once, can now be used multiple times for recognizing objects, inferring facts, or parametrizing actions.

Second, the complex design process and integration of different robot's behaviors requires the support of a well-defined framework. The framework is not only necessary to simplify the software integration, but it is also fundamental to extend scripted behaviors with autonomous task planning based on context awareness. In fact, task planning in robotics is still not largely used due to the complexity of defining a planning domain and keeping it constantly updated with the robot's available capabilities and sensors readings.

In the course of a larger project on kitting using mobile manipulators [1–3], we have developed a particularly efficient pipeline for automated grasping of parts from pallets and bins, and a pipeline for placing into kits. To integrate these pipelines, together and with other ones, into different robot platforms, the Skill-based platform for ROS (*SkiROS*) was developed. The proposal for implementing such a programming platform defines *tasks* as sequences of *skills*, where skills are identified as the re-occurring actions that are needed to execute stan-

dard operating procedures in a factory (e.g., operations like pick 'object' or place at 'location'). Embedded within the skill definitions are the sensing and motor operations, or *primitives*, that accomplish the goals of the skill, as well as a set of condition checks that are made before and after execution, to ensure robustness. This methodology provides a process to sequence the skills automatically using a modular task planner based on a standard domain description, namely the Planning Domain Definition Language (PDDL) [4]. The planning domain is automatically inferred from the robot's available skill set and therefore does not require to be stated explicitly from a domain expert.

In this research chapter we present a complete in-depth description of the platform, how it is implemented in ROS, and how it can be used to implement perception and manipulation pipelines on the example of mobile robot depalletizing, bin picking and placing in kits. The chapter is structured as follows. Sec. 2, discusses related work in general with a focus on the existing ROS applications. Sec. 3, discusses the software architecture theoretical background. Sec. 4, holds a tutorial on the graphical user interface. Sec. 5, holds a tutorial on the plug-ins development. Sec. 6, discusses the task planner theoretical background and tutorial on planner plug-in development. Sec. 7, presents an application on a real industrial kitting task. Sec. 8, discusses relevant conclusions.

1.1 Environment configuration

SkiROS consist of a core packages set that can be extended during the development process with plug-ins. The SkiROS package, and some initial plug-ins sets can be downloaded from, respectively:

- `https://github.com/frovida/skiros`, core package
- `https://github.com/frovida/skiros_std_lib`, extension with task planner, drive-pick-place skills and spatial reasoner
- `https://github.com/frovida/skiros_simple_uav`, extension with drive-pick-place for UAVs, plus takeoff and landing skills

SkiROS has been developed and tested on ubuntu 14.04 with ROS indigo and the compilation is not guaranteed to work within a different setup.

Dependencies Skiros requires the oracle database and the redland library installed on the system. These are necessary for the world model activity. To install all dependencies is possible to use the script included in the SkiROS repository `skiros/scripts/install_dependencies.sh`. Other dependencies necessary for the planner can be installed running the script `skiros_std_lib/scripts/install_dependencies.sh`. After these steps, SkiROS can be compiled with the standard "catkin_make" command. For a guide on how to launch the system after compilation, refer to Sec. 3.1.

2 Related work

During the last three decades, three main approaches to robot control have dominated the research community: reactive, deliberative, and hybrid control [5]. Reactive systems rely on a set of concurrently running modules, called behaviours, which directly connect input sensors to particular output actuators [6, 7]. In contrast, deliberative systems employ a sense-plan-act paradigm, where reasoning plays a key role in an explicit planning process. Deliberative systems can work with longer timeframes and goal-directed behaviour, while reactive systems respond to more immediate changes in the world. Hybrid systems attempt to exploit the best of both worlds, through mixed architectures with a deliberative high level, a reactive low level, and a synchronisation mechanism in the middle that mediates between the two [8]. Most modern autonomous robots follow a hybrid approach [9–12], with researchers focused on finding appropriate interfaces between the declarative descriptions needed for high-level reasoning and the procedural ones needed for low-level control.
In the ROS ecosystem, we find *ROSco* [1], *Smach* [2] and pi_trees [3] which are architectures for rapidly creating complex robot behaviors, under the form of Hierarchical Finite State Machine (HFSM) or Behavior Trees (BT). These softwares are useful to model small concatenations of primitives with a fair reactive behavior. The approach can be used successfully up to a level comparable to our skills' executive, but doesn't scale up for high dynamic contexts. In fact the architectures allow only static composition of behaviors and cannot adapt those to new situations during execution. At time being, and at the best of author knowledge, we find in the ROS ecosystem only one maintained package for automated planning: *rosplan* [4] (Trex is no longer maintaned). In rosplan, the planning domain has to be defined manually from a domain expert. With our approach, the planning domain is automatically inferred at run-time from the available skill set, which results in a higher flexibility and usability.
Knowledge representation plays a fundamental role in cognitive robotic systems [13], especially with respect to defining world models. The most relevant approach for our work is the cognitivist approach, which highlights the importance of symbolic computation: symbolic representations are produced by a human designer and formalised in an ontology. Several modern approaches for real use-cases rely on semantic databases [14–16] for logic programming. It allows robotic system to remain flexible at run-time and easy to re-program. A prominent example of knowledge processing in ROS is the KnowRob system [17], which combines knowledge representation and reasoning methods for acquiring and grounding knowledge in physical systems. KnowRob uses a semantic library which facilitates loading and accessing ontologies represented in the Web Ontology Language (OWL). Despite its advanced technology, KnowRob is a framework with

[1] http://pwp.gatech.edu/hrl/ros-commander-rosco-behavior-creation-for-home-robots/
[2] http://wiki.ros.org/smach
[3] http://wiki.ros.org/pi_trees
[4] https://github.com/KCL-Planning/ROSPlan

a bulky knowledge base and a strict dependency with the 'Prolog' language. For our project, a simpler and minimal implementation has been preferred, still compliant with the widely used OWL standard. Coupled with KnowRob there is CRAM (Cognitive Robot Abstract Machine)[18]. Like SkiROS, CRAM is a software toolbox for the design, the implementation, and the deployment of cognition-enabled autonomous robots, that do not require the whole planning domain to be stated explicitly. The CRAM kernel consists of the CPL plan language, based on Lisp, and the KnowRob knowledge processing system. SkiROS presents a similar theoretical background with CRAM, but differs in several implementations choices. For example, SkiROS doesn't provide a domain specific language such as CPL to support low-level behavior design, but relies on straight C++ code and the planner in CRAM is proprietary, meanwhile in SkiROS is modular and compatible with every PDDL planner.

3 Conceptual Overview

The Skill-based platform for ROS (*SkiROS*) [19] helps to design and execute the high-level skills of an hybrid behavior-deliberative architecture, commonly referred with the name of *3-tiered architecture* [9–12]. As such, it manages the executive and deliberative layers, that are expected to control a behavior layer implemented using ROS nodes. While the theory regarding 3-tiered architectures is well know, it is still an open question how to build a general and scalable platform with well-defined interfaces. In this sense, the development of the SkiROS platform has been carried out taking into consideration the needs of two key stakeholders: the developer and the end-user. This approach derives from the field of the interaction design, where the human's needs are placed as focal point of the research process. It is also included in the ISO standard [ISO9241-210;ISO16982].
Briefly, SkiROS provide: (i) a workspace to support the development process and software integration between different sources and (ii) an intuitive interface to instruct missions to the robot. The main idea is that the developers can equip the robots with *skills*, defined as the fundamental software building blocks operating a modification on the world state. The end-user can configure the robot by defining a scene and a goal and, given this information, the robot is able to plan and execute autonomously the sequence of skills necessary to reach the required goal state. The possibility of specifying complex states is tightly coupled with the amount of skills that the robot is equipped with and the richness of the robot's world representation. Nevertheless, developing and maintaining a large skill set and a rich knowledge base can be an overwhelming task, even for big research groups. Modular and shareable skills are mandatory to take advantage of the *network effect* - a phenomenon occurring when the number of developers of the platform grows. When developers start to share skills and knowledge bases, there is possible to develop a robot able to understand and reach highly articulated goals. This is particularly achievable for the industrial case, where the skills necessary to fulfill most of use-cases have been identified from different researchers as a very compact set [20, 21].
The ROS software development approach is great to develop a large variety of different control systems, but lacks support to the reuse of effective solutions to recurrent architectural design problems. Consequently, we opted for a software development based on App-like plug-ins, that limits the developer to program a specific set of functionalities, specifically: primitives, skills, conditions, task planners and discrete reasoners. This approach partially locks the architecture of the system, but ensure a straightforward re-usability of all the modules. On the other side, we also modularized the core parts of the system into ROS nodes, so that the platform itself doesn't become a black box w.r.t. to ROS and can be re-used in some of its parts, like e.g. the world model.
Several iterative processes of trial and refinement has been necessary in order to identify:

- how to structure the system

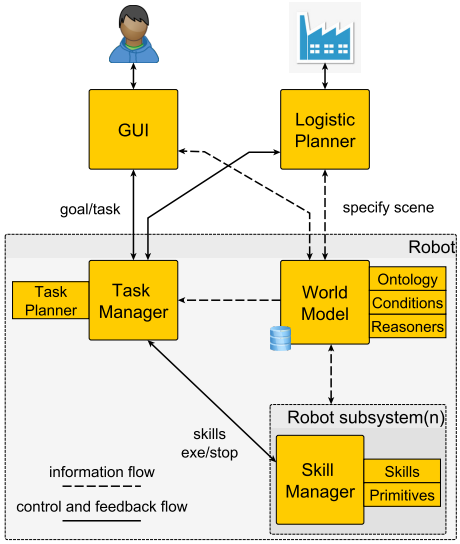


Fig. 2. An overview of the SkiROS architecture, with squares representing ROS nodes and rectangles representing plug-ins. The robot presents an external interface to specify the scene and receive a goal, that can be accessed by the GUI or directly from a factory system. Internally, the task manager dispatches the generated plans to the skill managers in each subsystem of the robot. A skill manager is the coordinator of a subset of capabilities, keeping the world model updated with its available hardware, skills and primitives. The world model is the focal point of the system: all knowledge is collected and shared through it.

- the part of the system that needs to be easily editable or replaced by the developer
- the interface required from the user in order to control and monitor the system, without the necessity of becoming an expert on all its parts

The application on a real use-case has been fundamental to apply these iterations.

3.1 Packages structure

SkiROS is a collection of ROS packages that implements a layered architecture. Each layer is a stand-alone package, which shares few dependencies with other layers. The packages in the SkiROS core repository are:

- **skiros** - the skiros meta-package contains ROS launch files, logs, ontologies, saved istances and scripts to install system dependencies
- **skiros_resource, skiros_primitive** - these packages are still highly experimental and are not taken into consideration in this paper. The primitives are currently managed together with skills, in the skill layer.

- **skiros_skill** - contains the skill manager node and the base class for the skills and the primitives
- **skiros_world_model** - contain the world model node, C++ interfaces to the ROS services, utilities to treat ontologies and the base class for conditions and reasoners
- **skiros_common** - shared utilities
- **skiros_msgs** - shared ROS actions, services and messages
- **skiros_config** - contains definition of URIs, ROS topic names and other reconfigurable parameters
- **skiros_task** - the higher SkiROS layer, contain the task manager node and the base class for task planner plug-in
- **skiros_rqt_gui** - the Graphical User Interface, a plug-in for the ROS rqt package

Each layer implements core functionalities with plug-ins, using the ROS 'pluginlib' system. The plug-ins are the basic building blocks available to the developer to design the robot behavior and tailor the system to his specific needs. This methodology ensure a complete inter-independence of the modules at compile time. Every node has clear ROS interfaces with others so that, if necessary, any implementation can be replaced. The system is also based on two standards: the Web Ontology Language (OWL) standard [22] for the knowledge base and the Planning Domain Definition Language (PDDL) standard [4] for the planner. The platform architecture is visualized in Fig. 2. The complete platform consist of three ROS nodes - task manager, skill manager and world model - plus a Graphical User Interface (GUI). It can be executed using the command:

```
roslaunch skiros skiros_system.launch robot_name:=my_robot
```

Where `my_robot` should be replaced with the desired semantic robot description in the knowledge base (see Sec. 5.1). The default robot model loaded is `aau_stamina_robot`. In the `skiros_std_lib` repository there is an example of the STAMINA use-case specific launch file:

```
roslaunch skiros_std_lib skiros_system_fake_skills.launch
```

This launch file runs the SkiROS system with two skill managers: one for the mobile base, loading the drive skill, and one for the arm, loading pick and place skills.

3.2 World model

Generally speaking, it is possible to subdivide the robot knowledge into three main domains: continuous, discrete and semantic. Continuous data is extracted directly from sensors. Discrete data are relevant features that are computed from the continuous data and are sufficient to describe a certain aspect of the environment. Semantic data is abstract data, that qualitatively describes a certain aspect of the environment. Our world model stores semantic data. It works as a

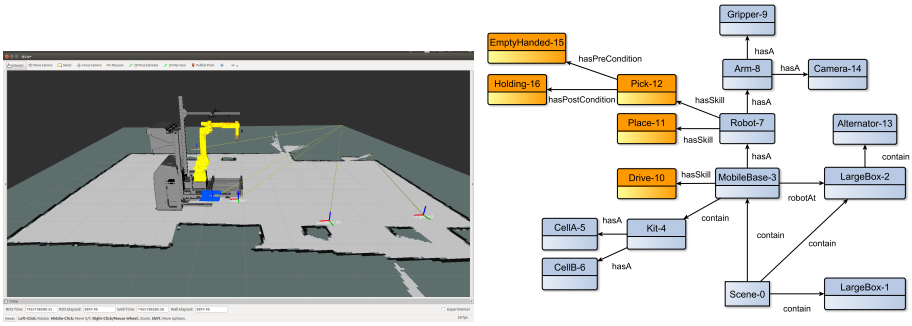


Fig. 3. An example of a possible scene, with the robot visualized on rviz (left) and the corresponding semantic representation (right). The scene includes both physical objects (blue boxes) and abstract objects (orange boxes).

knowledge integration framework and supports the other subsystems' logic reasoning by providing knowledge on any relevant topic. In particular, the robot's knowledge is organised into an ontology that can be easily embedded, edited and extracted from the system. It is defined in the Web Ontology Language (OWL) standard which ensures greater portability and maintainability. The OWL ontology files have usually a .owl extension, and are based on XML syntax. An ontology consists of a set of definitions of basic categories (objects, relations, properties) which describe the elements of the domain of interest, their properties, and the relations they maintain with each other [23]. Ontologies are defined in Description Logic (DL), a specialisation of first-order logic, which is designed to simplify the description of definitions and properties of categories. The knowledge base consists of a terminological component (T-Box), that contains the description of the relevant concepts in the domain and their relations, and an assertional component (A-Box) that stores concept instances (and assertions about those instances).
The SkiROS core ontology `skiros/owl/stamina.owl` gives a structure to organize the knowledge of 3 fundamental categories:

- the objects in the world
- the robot hardware
- the robot available capabilities (skills and primitives)

The knowledge base can be extended from the developer at will. It is possible to modify the default OWL loading path `skiros/owl`, by specifying the parameter `skiros/owl_workspace`. All the OWL files found in the specified path are automatically loaded from the world model at boot and merged to the SkiROS knowledge core - that is always loaded first. The world model node can be executed individually with the command:

```
rosrun skiros_world_model world_model_node
```

At run-time, the world model allows all the modules to maintain a shared working memory in a world instance, or scene, which forms a database complementary to the ontology database. The scenes are managed in the path specified in the `skiros/scene_workspace` parameter (default: skiros/scene). It is possible to start the world model with a predefined scene, by specifing the `skiros/scene_name` parameter. It is also possible to load and save the scene using the ROS service or the SkiROS GUI. An example of the scene tree structure is showed in Fig. 3. The ontology can be extended automatically by the modules, to learn new concepts in a long-term memory (e.g. to learn a new grasping pose). The modules can modify the A-Box but not the T-Box and It is possible to interface with the world model using the following ROS services and topics:

- */skiros_wm/lock_unlock* - shared mutex for exclusive access (service)
- */skiros_wm/query_ontology* - query the world model with SPARQL syntax (service)
- */skiros_wm/modify_ontology* - add and remove statements in the ontology. New statements are saved in the file `learned_concepts.owl` in the owl workspace path. The imported ontologies are never modified (service)
- */skiros_wm/element_get* - get one or more elements from the scene (service)
- */skiros_wm/element_modify* - modify one or more elements in the scene (service)
- */skiros_wm/set_relation* - set relations in the scene (service)
- */skiros_wm/query_model* - query relations in the scene (service)
- */skiros_wm/scene_load_and_save* - save or load a scene from file (service)
- */skiros_wm/monitor* - publish any change done to the world model, both ontology and scene (topic)

It is also available a C++ interface class `skiros_world_model/world_model_interface.h` that wraps the ROS interface and can be included in every C++ program. This interface is natively available for all the skills and primitives plugins (see Sec. 5.2).

3.3 Skill manager

The skill manager is a ROS node that collects the capabilities of a specific robot's subsystem. A skill manager is launched with the command:

```
rosrun skiros_skill skill_manager_node __name:=my_robot
```

Where `my_robot` has to be replaced with the identifier of the robot in the world model ontology. Since many of the skill managers' operations are based on the information stored in the world model, it requires the world model node to be running. Each skill manager in the system is responsible to instantiate in world scene its subsystem information: hardware, available primitives and available skills. Similarly, each primitive and skill can extend the scene information with the results of robot operation or sensing. To see how to create a new robot

definition refer to Sec. 5.1. A skill manager, by default, tries to load all the skills and primitives that are been defined in the pluginlib system[5]. It is also possible to load only a specific set by defining the parameters: `skill_list` and `module_list`. For example:

```
<node name="my_robot" pkg="skiros_skill" type="skill_manager_node">
  <param name="skill_list" type="string" value="pick place"/>
  <param name="module_list" type="string" value="arm_motion locate"/>
</node>
```

In this case the robot `my_robot` will try to load pick and place skill, and the arm_motion and locate primitives. If the modules are loaded correctly, they will appear on the world model, associated to the robot name. It is possible to interface with the skill manager using the following ROS services and topics:

- */my_robot/module_command* - command execution or stop of a primitive (service)
- */my_robot/module_list_query* - get the primitive list (service)
- */my_robot/skill_command* - command execution or stop of a skill (service)
- */my_robot/skill_list_query* - get the skill list (service)
- */my_robot/monitor* - publish execution feedback (topic)

It is also available a C++ interface class `skiros_skill/skill_manager_interface.h` that wraps the ROS interface and can be included in every C++ program and an high-level interface class `skiros_skill/skill_layer_interface.h` to handle multiple skill managers. Note that, on every skill manager, the same module can be executed once at a time, but different modules can be executed concurrently.

3.4 Task manager

The task manager acts as the general robot coordinator. It monitors the presence of robot's subsystems via the world model and use this information to connect to the associated skill manager. The task manager is the interface for external systems, designed to be controlled by a GUI or the manufacturing execution system (MES) of a factory. The task manager is launched individually with the command:

```
rosrun skiros_task task_manager_node
```

It is possible to interface with the task manager using the following ROS services and topics:

- */skiros_task_manager/task_modify* - add or remove a skill from the list (service)
- */skiros_task_manager/task_plan* - send a goal to plan a skill sequence (service)
- */skiros_task_manager/task_query* - get the skill sequence (service)
- */skiros_task_manager/task_exe* - start or stop a task execution (topic)
- */skiros_task_manager/monitor* - publish execution feedback (topic)

[5] http://wiki.ros.org/pluginlib

3.5 Plugins

The plug-ins are C++ classes, derived from an abstract base class. Several plug-ins can derive from the same abstract class. For example, any skill derives from the abstract class skill base. The following system parts have been identified as modules:

- **skill** - an action with pre- and postconditions that can be concatenated to form a complete task
- **primitive** - a simple action without pre- and postconditions, that is concatenated manually from a expert programmer inside a skill. The primitives support hierachical composition
- **condition** - a desired world state. It is expressed as a boolean variable (true/false) applied on a property of an element (property condition) or a relation between two elements (relation condition). The plug-in can wrap methods to evaluate the condition using sensors.
- **discrete reasoner** - an helper class necessary to link the semantic object definition to discrete data necessary for the robot operation
- **task planner** - a plug-in to plan the sequence of skills given a goal state. Any planner compatible with PDDL and satisfing the requirements described in Sec. 6 can be used

These software pieces are developed by programmers during the development phase and are inserted as plug-in into the system.

3.6 Multiple robots control

SkiROS can be used in multi-robot system in two ways. In the first solution, each skill manager is used to represent a robot in itself, and the task manager is used to plan and dispatch plans to each one of them. The solution is simple to implement and the robots will have a straightforward way to share the information via the single shared world model. The main limitation is that the skill execution is at the moment strictly sequential. Therefore, the task manager will move the robots one at a time. The second solution consist in implementing an high-level mission planner, and use this to dispatch goals for the SkiROS system running on each one of the robots. The latter solution is the one currently used for the integration in the PSA factory system [24].

4 User interface

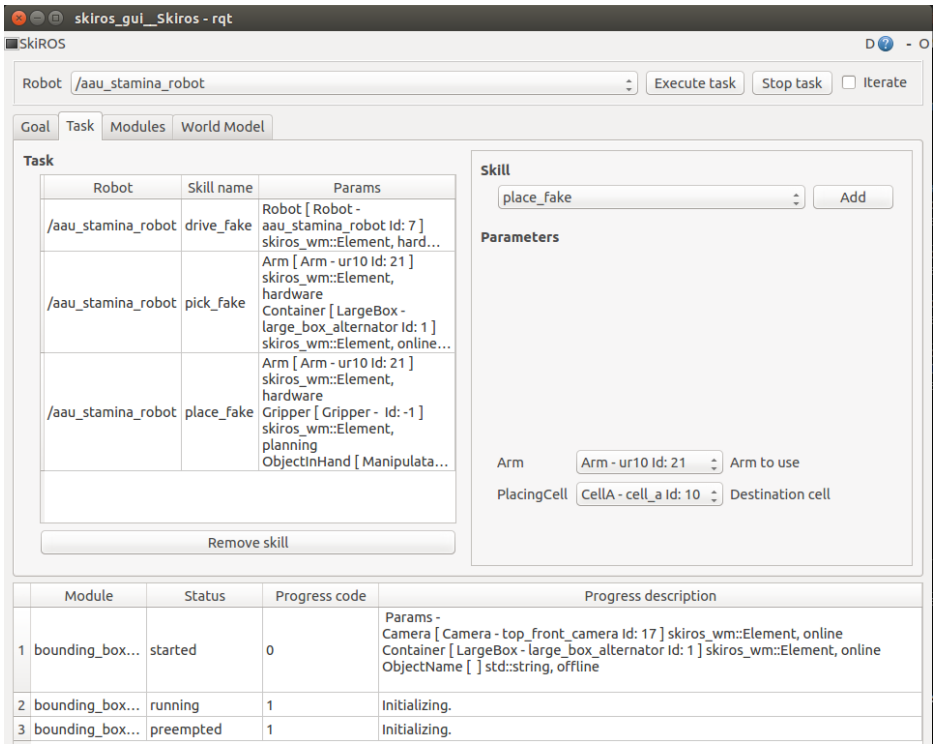


Fig. 4. The full GUI showing the task tab.

To allow any kind of user to be able to run and monitor the execution of the autonomous robot, the support of a clean and easy-to-use UI is necessary. At the moment, the interaction between human and robot is based on a Graphical UI, which in the future can be extended with more advanced and intuitive ways of interaction, like voice or motion capture. The full GUI is presented in Fig. 4. It consists of 4 tabs:

- **Goal** - from this tab is possible to specify the desired goal state and trigger an action planning
- **Task** - this tab visualize the planned skill sequence and allows to edit it
- **Module** - this tab allows to run modules (skills and primitives). It is principally used for testing purposes
- **World model** - from this tab is possible to load, edit and save the world scene

The GUI is structured for different level of user skill. The most basic user is going to use the Goal tab and the World model tab. First of all, he can build up

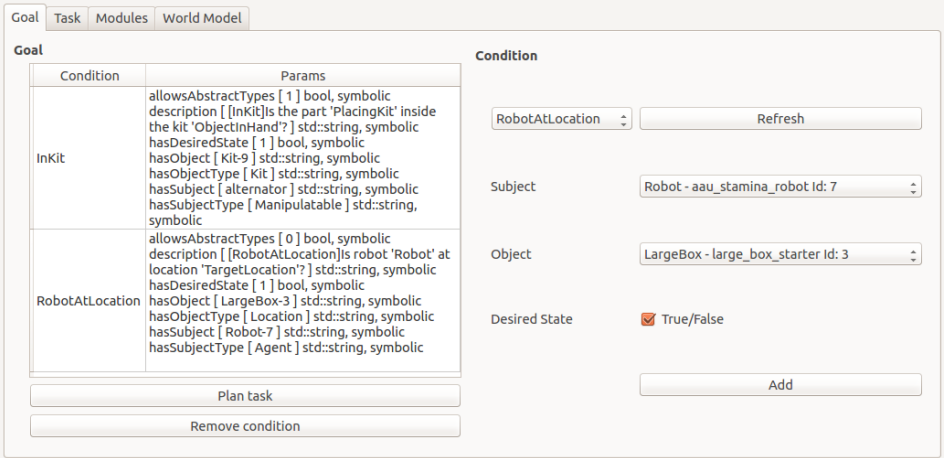


Fig. 5. The goal tab.

a scene, then can specify the goals, plan a task and run or stop the execution. More advanced user can edit the planned task or build it by themselves from the Task tab. System tester can use the Module tab for module testing.

4.1 Edit, execute and monitor the task

From the task tab presented in Fig. 4 is possible to edit a planned task or create a new one from scratch. The menu on the left allows to add a skill. First, the right robot must be selected from the top bar (e.g. `/aau_stamina_robot`). After this, a skill can be selected from the menu (e.g. `place_fake`). The skill must be parametrized appropriately and then can be added to the task list clicking the 'Add' button. The user can select each skill on the task list and remove it with the 'Remove skill' button. On the top bar there are two buttons to execute and stop the task execution and the 'Iterate' check box, that can be selected to repeat the task execution in loop (useful for testing a particular sequence). At the bottom is visualized the execution output of all the modules, with the fields:

- **Module** - the module name
- **Status** - this can be: *started*, *running*, *preempted*, *terminated* or *error*
- **Progress code** - a positive number related to the progress of the module execution. A negative number indicates an error
- **Progress description** - a string describing the progress

4.2 Plan a task

From the goal tab it is possible to specify the desired goal state and generate automatically a skill sequence, that will be then available in the task tab. The goal is expressed as a set of conditions required to be fulfilled. These can be

chosen between the set of available ones. The available conditions set is calculated at run-time depending on the robots' skill set and can be updated using the 'Refresh' button. It is possible to specify as goals only conditions that the robot can fulfill with its skills, or in other words, conditions that appear in at least one of the skills. In the example in Fig. 5, we require an abstract `alternator` to be in `Kit-9` and we require the robot to be at `LargeBox-3`. By *abstract* we refer to individuals that are defined in the ontology, but not instantiated in the scene. Specify an abstract object means specify any object which match the generic description. The `InKit` condition allows abstract types, meanwhile `RobotAtLocation` can be applied only on instantiated objects (objects in the scene). For more details about conditions the reader can refer to Sec. 5.3.

4.3 Module testing

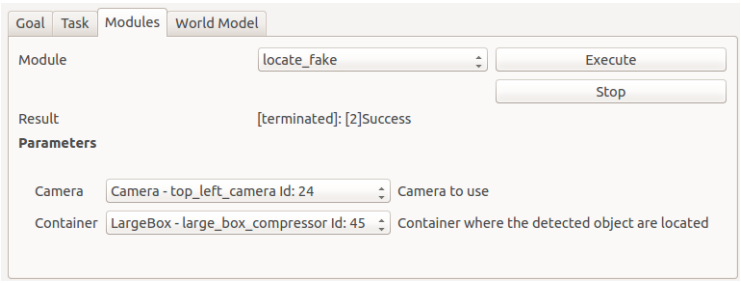


Fig. 6. The Modules tab.

From the Modules tab is possible to execute primitives. The procedure is exactly the same presented previously for the skills, except that the primitives are executed singularly. The module tab becomes handy to test the modules singularly or to setup the robot, e.g. to teach a new grasping pose or to move the arm back to home position.

4.4 Edit the scene

From the world model tab Fig. 7 is possible to visualize and edit the world scene. On the left the scene is visualized in a tree structure. The limit of the tree structure doesn't allow to visualize the whole semantic graph, which can count several relations between the objects. We opted to limit the visualization to a scene graph, that is a general data structure commonly used in modern computer games to arrange the logical and spatial representation of a graphical scene. Therefore, only the spatial relations (contain and hasA) are visible, starting from the scene root node. On the right there are buttons to add, modify and remove objects in the scene. When an element in the tree is selected, its properties are

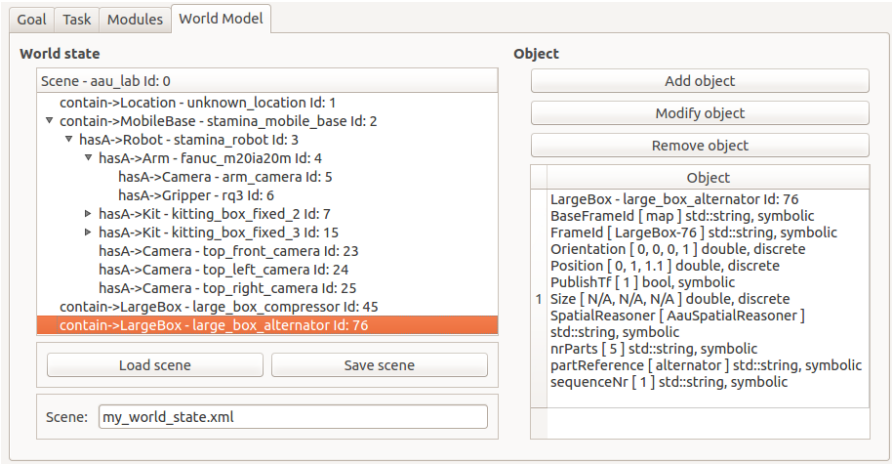


Fig. 7. The World Model tab.

displayed in the box on the right. In the figure, for example, we are displaying the properties of the LargeBox 76. It is possible to edit its property by clicking on the 'Modify object' button. This opens a pop-up window where properties can be changed one by one, removed (by leaving the field blank) or added with the '+' button on bottom. It is also possible to add an object by clicking on the 'Add object' button. When an object is added to the scene, it becomes child of the selected element in the tree. The properties and objects are limited to the set specified in the ontology. This not only helps to avoid input mistakes, but also to give the user an intuitive feedback on what it possible to put in the scene. Once the scene is defined, the interface on the bottom left allows to save the scene for future use.

5 Development

In this section we discuss how to develop an action context for the robot planning, using as an example a simplified version of the kitting planning context. The development process consist of two steps: specify the domain knowledge in the ontology and develop the plug-ins specified in Sec. 3.5. Some plug-ins and templates are available together with the SkiROS core package (see Sec. 1.1).

5.1 Edit the ontology

Before starting to program, an ontology must be defined to describe the objects in the domain of interest. This regards in particular:

- **Data** - define which kind of properties can be related to elements
- **Concepts** - define the set of types expected to find using a taxonomy, a hierarchical tree where is expressed the notion about types and subtypes

Fig. 8. The taxonomy of spatial things for the kitting application.

- **Relations** - define a set of relations between elements
- **Individuals** - some predefined instances with associated data. E.g, a specific device or a specific robot

The world model is compliant with the OWL w3c standard that counts several tools for managing ontologies. A well-known open-source program is Protege (http://protege.stanford.edu/). To install and use it, the reader can refer to one of the several guides available on internet[6]. As introduced in Sec. 3.2, it is possible to have several custom ontologies in the defined OWL path and these get automatically loaded and merged with the SkiROS knowledge core at boot. The reader can refer to the `uav.owl` file in the `skiros_simple_uav` repository to see a practical example on how to create an ontology extension. The launch file in the same package provide an example on how to load it. Note in particular the importance of providing the right ontology *prefix* in the launch file and in general when referencing entities in the ontology.
The entities defined in the ontology are going to constraint the development of skill and primitives. For example, a place skill will require as input only objects that are subtypes of `Container`. To avoid the use of strings in the code, that are impossible to track down, an utility has been implemented in the `skiros_world_model` package to generate an enum directly from the ontology. It is possible to run this utility with the command:

[6] e.g. http://protegewiki.stanford.edu/wiki/Protege4GettingStarted

```
rosrun skiros_world_model uri_header_generator
```

This utility updates the `skiros_config/declared_uri.h`, automatically included in all modules. Using the generated enum for the logic queries allows to get an error at compile time, if the name changes or is missing.

Create a new robot definition The semantic robot structure is the necessary information for the skill manager to manage the available hardware. E.g. if the robot has a camera mount on the arm, it can move the camera to look better at an object. The robot and its devices must be described in detail with all the information that the developer wants to have stated explicitly. The user should use protege to create an ontology with an *individual* for each device and an *individual* for the robot, collecting the devices using the *hasA* relation. To give a concrete example, lets consider the `aau_stamina_robot`, the smaller stamina prototype used for laboratory test:

- **NamedIndividual**: aau_stamina_robot
- **Type**: Robot
- *LinkedToFrameId*: base_link
- *hasA*− >top_front_camera
- *hasA*− >top_left_camera
- *hasA*− >top_right_camera
- *hasStartLocation*− >unknown_location
- *hasA*− >ur10

The robot has a `LinkedToFrameId` property, related to the `AauSpatialReasoner`, and a start location, used for the drive skill. For more information about this properties, refer to the plug-ins description. The robot hardware consist of 3 cameras and a robotic arm (ur10). If we expand the ur10 description we find:

- **NamedIndividual**: ur10
- **Type**: Arm
- *MoveItGroup*: arm
- *DriverAddress*: /ur10
- *MotionPlanner*: planner/plan_action
- *MotionExe*: /arm_controller/follow_joint_trajectory
- *hasA*− >arm_camera
- *hasA*− >rq3

The arm has an additional camera and a end-effector rq3. Moreover, it has useful properties for the configuration of MoveIt. Thanks also to the simple parametrization of MoveIt, we have been able to port the *same* skills on 3 heterogeneous arms (ur, kuka and fanuc) by changing only the arm description as presented here.

5.2 Create a primitive

A SkiROS module is a C++ software class based on the standard ROS plug-in system. In particular, those who are experienced in programming ROS nodelets [7] will probably find it straightforward to program SkiROS modules. Developing a module consists of programming a C++ class derived from an abstract template. The basic module, or primitive, usually implements an atomic functionality like opening a gripper, locating an object with a camera, etc. These functionalities can be reused in other primitives or skills or executed individually from the module tab. A primitive inherits from the template defined in `skiros_skill/module_base.h`. It requires to specify the following virtual functions:

```
//! \brief personalized initialization routine
virtual bool onInit() = 0;
//! \brief module main
virtual int execute() = 0;
//! \brief specialized pre-preempt routine
virtual void onPreempt();
```

The `onInit()` function is called when the skill manager is started to initialize the primitive. The `execute()` function is called when the primitive is executed from the GUI or called by another module. The `onPreempt()` function is called when the primitive is stopped, e.g. from the GUI. All primitives have protected access to a standard set of interfaces:

```
//! \brief Interface with the parameter set
boost::shared_ptr<ParamHandler> getParamHandler();
//! \brief Interface with the skiros world model
boost::shared_ptr<WorldModelInterfaceS> getWorldHandler();
//! \brief Interface to modules
boost::shared_ptr<SkillManagerInterface> getModulesHandler();
//! \brief Interface with the ROS network
boost::shared_ptr<NodeHandle> getNodeHandler();
```

The `ParamHandler` allows to define and retrieve parameters.
The `WorldModelInterface` allows to interact with the world model. The primitive's parameter set must be defined in the `classconstructor` and never modified afterwards.
The `SkillManagerInterface` allows the primitive to interact with other modules.

World model interface Modules' operations apply over an abstract world model, which has to been constantly matched to the real world. There is no space in this chapter to describe in detail the world model interface. Nevertheless, it is

[7] http://wiki.ros.org/nodelet

important to present the atomic world model's data type, defined as *element*. In fact, the element is the most common input parameter for a module. An element structure is the following:

```
//Unique Identifier of the element in the DB
int id;
//Individual identifier in the ontology
std::string label;
//Category identifier in the ontology
std::string type;
//Last update time stamp
ros::Time last_update;
//A list of properties (color, pose, size, etc..)
std::map<std::string, skiros_common::Param> properties;
```

The first 3 fields are necessary to relate the element in the ontology (label and type) and the scene database (id). The properties list contains all relevant information associated to the object.

Parameters Every module relies on a dynamic set of parameters to configure the execution. The parameters are divided in the following categories:

- *online* - parameters that must always be specified
- *offline* - usually are configuration parameters with a default value, such as the desired movement speed, grasp force, or stiffness of the manipulator
- *optional* - like an offline parameter, but can be left unspecified
- *hardware* - indicate a robot's device the module need to access. This can be changed at every module call (e.g. to locate with different cameras)
- *config* - like an hardware parameter, but it is specified when the module loads and *cannot* be changed afterwards. (e.g. an arm motion module bounded to a particular arm)
- *planning* - a parameter necessary for pre and post condition check in a skill. This is set automatically and doesn't appear on the UI.

To understand the concept, we present a code example. First, we show how to insert a parameter:

```
getParamHandler()->addParam<my::Type>("myKey", "My description", skiros_common::online, 3);
```

Here we are adding a parameter definition, specifying in the order: the key, a brief description, the parameter type, and the vector length. The template argument has to be specified explicitly too. In the above example we define an online parameter as a vector of 3 doubles. The key `myKey` can be used at execution time to access the parameter value:

```
std::vector<double> myValue = getParamHandler()->↩
    getParamValues<double>("myKey");
```

The parameter state is defined as initialized until its value get specified. After this the state changes to specified. A module cannot run until all unspecified parameters are specified. It is also possible to define parameters with a default value:

```
getParamHandler()->addParamWithDefaultValue("myKey", true, "↩
    My description", skiros_common::offline, 1);
```

In this case, the parameter will be initialized to true. When an input parameter is a world's element, it is possible to apply a special rule to limit the input range. In fact sometimes a module requires a precise type of element as input. For example, a pick skill can pick up only elements of type `Manipulatable`. In this case, it is possible to use a partial definition. For example:

```
'/skiros_std_lib/skiros_lib_dummy_skills/src/pick.cpp' L.34:
getParamHandler()->addParamWithDefaultValue("object",↩
    skiros_wm::Element("Manipulatable"), "Object to pick up")↩
    ;
```

In this example, only subtypes of `Manipulatable` will be valid as input for the `object` parameter. Every module can have a customized amount of parameters. The parameters support any data type that can be serialized in a ROS message. This means all the standard data types and all the ROS messages. ROS messages requires a quick but non-trivial procedure to be included in the system, that is excluded from the chapter for space reasons.

Invoke modules Each module can recursively invoke other modules' execution. For example, the pick skill invoke the locate module with the following:

```
1 '/skiros_std_lib/skiros_lib_dummy_skills/src/pick.cpp' L.155:
2 skiros::Module locate(getModulesHandler(), "locate_fake", ↩
      this->moduleType());
3 locate.setParam("Camera", camera_up_);
4 locate.setParam("Container", container_);
5 locate.exe();
6 locate.waitResult();
7 v = getWorldHandle()->getChildElements(container_,"", ↩
      objObject.type());
```

Here, line 2 instantiate a proxy class for the module named `locate_fake`. Line 3 and 4 set the parameters and line 5 request the execution. The execution is non blocking, so that is possible to call several modules in parallel. In this case, we wait for the execution end and then we retrieve the list of located object.

5.3 Create a skill

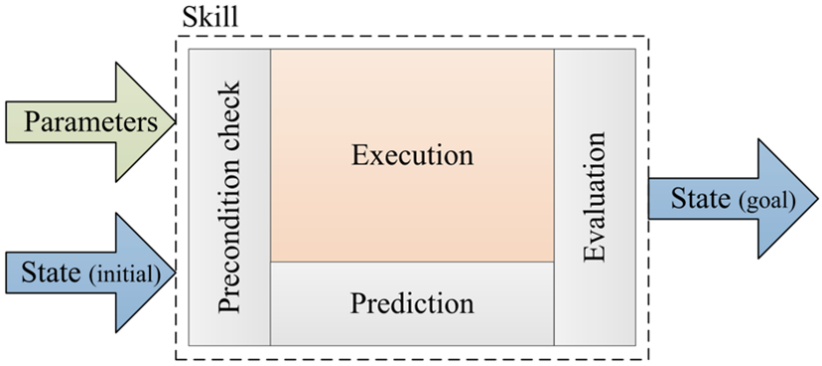


Fig. 9. The model of a robot skill [21]. Execution is based on the input parameters and the world state, and the skill effectuates a change in the world state. The checking procedures before and after execution verify that the necessary conditions are satisfied.

A skill is a complex type of module, which inherits from the template defined in `skiros_skill/skill_base.h`. A conceptual model of a complete robot skill is shown in Fig. 9. A skill extends the basic module definition with the presence of pre- and postcondition checks. By implementing pre- and postcondition checking procedures the skills themselves verify their applicability and outcome. This enables the skill-equipped robot to alert an operator or task-level planner if a skill cannot be executed (precondition failures) or if it did not execute correctly (postcondition failures). A formal definition of pre- and postconditions is not only useful for robustness, but also task planning, which utilizes the preconditions and prediction to determine the state transitions for a skill, and can thus select appropriate skills to be concatenated for achieving a desired goal state. A skill adds also two more virtual functions, `preSense()` and `postSense()`, where sensing routines can be called before evaluating pre- and postconditions.

Create a condition Preconditions and postconditions are based on sensing operations and expected changes to the world model from executing the skill. The user defines the pre- and postconditions in the skill `onInit()` function, after the parameter definitions. The conditions can be applied only on world model's *elements* input parameters. While some ready-to-use conditions are available in the system, it's also possible to create a new condition by deriving it from the condition templates `skiros_wm/condition.h`. There are two base templates: `ConditionRelation` and `ConditionProperty`. The first put a condition on a relation between two individuals. The second put a condition on a property of an individual. When implementing a new condition, two virtual functions have to be implemented:

- **void init()** - here define the property (or the relation) on which the condition is applied
- **bool evaluate()** - a function that returns true if the condition is satisfied or false otherwise

Once the condition is defined, every skill can add it to its own list of pre or postconditions in the `onInit()` function. For example:

```
'/skiros_std_lib/skiros_lib_dummy_skills/src/pick.cpp' L.62:
addPrecondition(newCondition("RobotAtLocation", true, "Robot", "Container"));
```

This command instantiate a new condition `RobotAtLocation` between `Robot` and `Container` and add it to the list of preconditions. Note that `Robot` and `Container` refer to the key defined in the input parameters. In this case, the skill requires the `Robot` parameter to have a specific relation with the `Container` parameter. If this relation doesn't hold, the skill will return a failure without being executed.

5.4 Create a discrete reasoner

The world's *elements* are agnostic placeholders where any kind of data can be stored and retrieved. Their structure is general and flexible, but this flexibility requires that no data-related methods are implemented. The methods are therefore implemented in another code structure, called *discrete reasoner*, that is imported in the SkiROS system as a plug-in. Any reasoner inherits from the base class `skiros_world_model/discrete_reasoner.h`. The standardized interface allow to use the reasoners as utilities to (i) store/retrieve data to/from elements and (ii) to reason about the data to compare and classify elements at a semantic level.

Spatial reasoner A fundamental reasoner for manipulation is the spatial reasoner, developed specifically to manage position and orientation properties. The `AauSpatialReasoner`, an implementation based on the standard 'tf' library of ROS, is included in the `skiros_std_lib/skiros_lib_reasoner` package. An example of the reasoner use is in the following:

```
container_ = getParamHandler()->getParamValue<skiros_wm::Element>("Container");
skiros_wm::Element object;
object.type()= concept::Str[concept::Compressor];
object.storeData(tf::Vector3(0.5,0.0,0.0),data::Position, "AauSpatialReasoner");
tf::Quaternion q;
q.setRPY(0.0,0.0,0.0);
object.storeData(q, data::Orientation);
```

```
8  object.storeData(string("map"), data::BaseFrameId);
9  tf::Pose pose= object.getData<tf::Pose>(data::Pose);
10 std::set<std::string> relations = object.getRelationsWrt(↩
       container_);
```

In this example, we first get the container variable from the input parameters. Then we create an new object instance and we use the `AauSpatialReasoner` reasoner to store a position, an orientation and the reference frame. Note that it necessary to specify the reasoner only on the first call. At line 8, we get back the object pose (a combination of position and orientation). At line 9 we use the reasoner to calculate semantic relations between the object itself and the container. The relations will contain predicates like front/back, left/right, under/over, etc. It is also possible to get relations with associated literal values for more advanced reasoning. It is up to the developer to define the supported data structures in I/O and which relevant semantic relations are extracted.

Example To give an example of some of the concepts presented in the section, lets consider the code of a skill to start the flying of an UAV (note: the code is slightly simplified w.r.t. the real file):

```
1  '/skiros_simple_uav/simple_uav_skills/src/flyToAltitude.cpp':
2  class FlyAltitude : public SkillBase
3  {
4  public:
5      FlyAltitude()
6      {
7          this->setSkillType("Drive");
8          this->setVersion("0.0.1");
9          getParamHandle()->addParamWithDefaultValue("Robot", ↩
               skiros_wm::Element(concept::Str[concept::Robot]),↩
                "Robot to control", skiros_common::online);
10         getParamHandle()->addParamWithDefaultValue("Altitude"↩
               , 1.0, "Altitude to reach (meters)", ↩
               skiros_common::offline);
11     }
12     bool onInit()
13     {
14         addPrecondition(newCondition("uav:LowBattery", false,↩
                "Robot"));
15         addPostcondition("NotLanded", newCondition("uav:↩
               Landed", false, "Robot"));
16         return true;
17     }
18     int preSense()
19     {
20         skiros::Module monitor(getModulesHandler(),"↩
               monitor_battery", this->skillType());
```

```
21          monitor.setParam("Robot", getParamHandle()->↩
                getParamValue<Element>("Robot"));
22          monitor.setParam("f", 10.0);
23          monitor.exe();
24          return 1;
25      }
26      int execute()
27      {
28          double altitude = getParamHandle()->getParamValue<↩
                double>("Altitude");
29          this->setProgress("Going to altitude " + std::↩
                to_string(altitude));
30          ros::Duration(2.0).sleep(); //Fake an execution time
31          setAllPostConditions();
32          return 1;
33      }
34  };
35  //Export
36  PLUGINLIB_EXPORT_CLASS(FlyAltitude, skiros_skill::SkillBase)
```

Lets go through the code line by line:

- **Constructor** - line 7-8 define constants to describe the module itself. Line 9-10 define the required parameters.
- **onInit()** - line 14 add the precondition of having a charged battery, line 15 add a postcondition of having the robot no more on the ground. Note the use of the prefix 'uav:' to the condition names. This because the conditions are defined in the `uav.owl` ontology.
- **preSense()** - invoke the `monitor_battery` module, to update the condition of the battery.
- **execute()** - at line 28 the parameter `Altitute` is retrieved. Line 29 print out a progress message. Line 30 and 31 are in place of a real implementation. In particular, the `setAllPostConditions()` command set all postconditions true, in order to simulate the execution at a high-level. Line 32 return a positive value, to signal that the skill terminated correctly.

At the very end, line 36 exports the plug-in definition.

6 Task Planner

The Task Planner's function is to provide a sequence of instantiated skills that, when carried out successfully, will lead to a desired goal state that has been specified by the user or some other automated process. For example, the goal state may be that a certain object has been placed in a kit that is being carried by the robot, and the returned sequence of skills (the plan) may be to *drive* to the location where the object can be found, to *pick* the object, and then to *place* the object in the kit. Of course, real goals and plans can be much more complicated than this, only limited by the relations that exist in the world model and the skills that have been defined.

This section discusses the general translation algorithm to the Planning Domain Definition Language (PDDL) aswell as the additions tailored for the STAMINA use-case. PDDL is a well-known and popular language for writing automated planning problems and, as such, is supported by a large array of of-the-shelf planners. The PDDL files created with the basic Task Planner options only use the types requirement of the original PDDL 1.2 version and are therefore suitable for use with almost all existing planners. Some extra features of the Task Planner (see Section 6.3) can be employed which introduce *fluents* and *durative actions*, and therefore must be used with a PDDL 2.1 compatible temporal planner. The Task Planner is built as a plug-in to SkiROS that generate generic PDDL so that the developer can insert whichever external planning algorithm they wish to use.

6.1 Overview and Usage

The Task Planner exists as a plug-in for SkiROS (`skiros_std_lib/task_planners/skiros_task_planner_plugin`) that creates a PDDL problem using the interface provided in `skiros_task/pddl.h`. The Task Planner contains two const's; `robotParameterName`, and `robotTypeName` which default to `Robot` and `Agent` respectively and must be consistent with the robot names used in the skill and world model definitions. Additionally, pddl.h contains a const bool `STAMINA` that toggles the use of specific extra features (explained in Section 6.3). Operation of the Task Planner is split into five main functions, as shown in Figure 10 and described below:

- **initDomain** - This function causes the Task Planner to translate the skill information in the world model into the planning actions and predicates found in the planning domain. While this translation is based on the preconditions and postcondition defined in the skill, it is not direct, and details of the necessary modifications are given below.
- **setGoal** - This function sets the goal, or set of goal predicates, to be planned for. These can be provided as SkiROS *elements* or as PDDL strings.
- **initProblem** - This function takes no arguments and tells the Task Planner to query the world model to determine the initial state of the planning problem. This must be called after the previous two functions as it relies

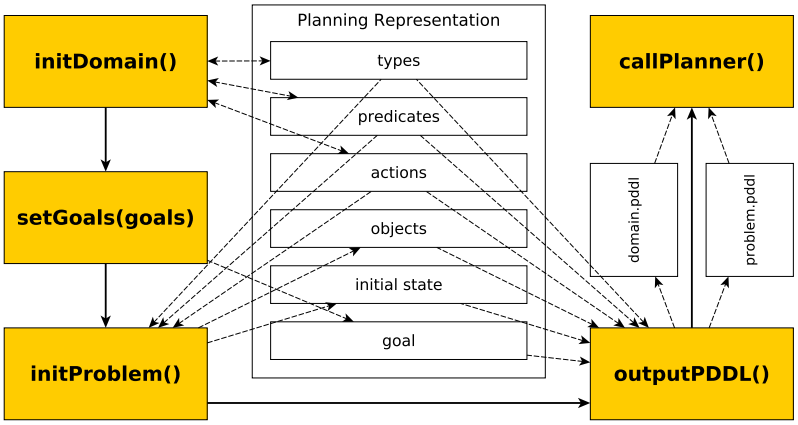


Fig. 10. An overview of the Task Planning pipeline. The inputs are the set of goals and the world scene. The output is a sequence of instantiated skills. The solid arrows represent execution flow. The dashed arrows pointing towards the data structures represent writing or modification while the dashed arrows pointing away from the data structures represent read access.

on them to work out which parts of the world model are relevant to the planning problem.
- **outputPDDL** - This function prints out a pddl domain file *domain.pddl* and problem file *p01.pddl* in the task planner directory.
- **callPlanner** - This function invokes an external planner that will take the previously output PDDL files as input and return a plan. This must be implemented by the user for whichever external planner they wish to use. The planner has been tested with Fast Downward[8] for the general case and Temporal Fast Downward[9] for the STAMINA use-case. Any plan found must then be converted to a vector of parameterised skills, so any extra parameters created for internal use by the planner must be removed at this point.

The user only needs to specify the external call to the planner in the callPlanner function. The setGoal function is the only one that takes arguments and requires a goalset comprised of SkiROS *elements* or PDDL strings. The other functions interface directly with the world model and require no arguments.

6.2 From skills to PDDL

The design of the SkiROS skills system facilitates the translation to a searchable A.I. planning format. However, a direct translation from skills to planning actions is not possible, or even desirable. The preconditions and postconditions

[8] `http://www.fast-downward.org/`
[9] `gki.informatik.uni-freiburg.de/tools/tfd/`

of skills should be definable by a non-planning expert based on the precondition and postcondition checks required for safe execution of the skill along with any expected changes to the world model. Therefore, the translation algorithm is required to do some work to generate a semantically correct planning problem. We will briefly discuss two important points; how it deals with heterogeneous robots, and implicitly defined transformations.

Heterogenous Robots There may be multiple robots in the world, each with different skill sets. For example, in the STAMINA use-case, the mobile platform is defined as a separate robot to the robotic arm. The mobile platform has the *drive* skill while the gripper has the *pick* and *place* skills.

To ensure that each skill 's' can only be performed by the relevant robot, a 'can_s' predicate is added as a precondition to each action, so that the action can only be performed if 'can_s(r)' is true for robot 'r'. 'can_s(r)' is then added to the initial state of the problem for each robot 'r' that has skill 's'. This way the planner can plan for multiple robots at a time.

Implicit Transformations The skill definitions may include implicit transformation assumptions that need to be made explicit for the planner. For example, the STAMINA drive skill is implemented with the following condition:

```
addPostcondition("AtTarget", newCondition("RobotAtLocation", true, "Robot", "TargetLocation"));
```

That is, only a single postcondition check, that the robot is at the location it was meant to drive to. For updating the SkiROS world model, setting the location of the robot to the 'TargetLocation' will automatically remove it from its previous location. However, the planner needs to explicitly encode the deletion of the previous location otherwise the robot will end up in two places at once in its representation.

Of course, it is possible to add the relevant conditions to the skill definition. However, this is not ideal as it would mean that the robot performs verification that it is not at the previous location at the end of the drive skill. This prohibits the robot from driving from its location to the same location as the new postcondition check would fail. Whether this is a problem or not, allowing the Task Planner to automatically include explicit updates reduces the pressure on the skill writer to produce both a skill definition that is correct in terms of both the robot and the internal planning representation.

The transformation is performed in a general manner that works for all spatial relations. The skills in the planning library are iterated over and checked against the spatial relations defined in SkiROS. If spatial relations are found to be missing, in either the preconditions or delete effects of the action (i.e., no predicate with matching relation and subject as in the case of the drive skill), then a new predicate of the same spatial relation and the same object, but a new subject variable, is created and added to the preconditions and delete effects of

the action. If a related spatial relation exists in just one of the preconditions and delete effects then it is added (with the same subject) to the other. More details of the planning transformation algorithm can be found in [25].

6.3 Additional Features

The task planner contains additional features used in the STAMINA problem instance that can either be enabled or disabled depending on user preference. These extra features include sequence numbers for ordering navigation through the warehouse, for which the planner employs numeric fluents and temporal actions, and abstract objects for which the planner must add internal parameters to ensure correct execution.

Sequence Numbers In the usecase for STAMINA, the robot navigates around a warehouse following a strict path. This is enforced following the previous setup to ensure that human workers will always exit in the same order they entered, therefore preserving the output order of the kits they are creating. The locations that it is possible to navigate to are given a sequence number (by a human operator) and these numbers are used to determine the shortest path based on the particular parts in the current order.

Abstract Objects In the world model for STAMINA, parts are not instantiated until they are actually picked up. The pick skill is called on an abstract object because it is not known before execution which of the possibly numerous objects in a container will be picked up. On the other hand, the place skill is often called for an instantiated object, as, at the time of execution, it is a particular instance of an object that is in the gripper.

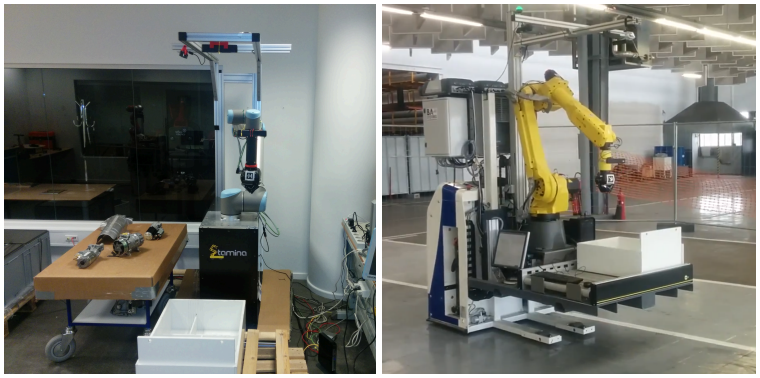


Fig. 11. The two different hardware setups that have been used for evaluation.

7 Application Example

7.1 Overview

As an example application, we focus on a logistic operation and specifically consider the automation of an industrial kitting operation. Such operation is common in a variety of manufacturing processes since it involves the navigation of a mobile platform to various containers from which the objects are picked and placed in their corresponding compartments of a kitting box. Thus, in order to achieve this task we have developed three skills, namely the **drive**, **pick** and **place** which consist of a combination of primitives such:

- **locate** - roughly localize an object on a flat surface using a camera
- **object registration** - precisely localize an object using a camera
- **arm motion** - move the arm to a desired joint state or end-effector pose
- **kitting box registration** - localization of the kitting-box using a camera
- **gripper ctrl** - open and close the gripper

Ontology The ontology that represents our specific kitting domain has been defined starting from the general ontology presented in Fig. 8. We extended the ontology with the types of manipulatable objects (starter, alternator, compressor, etc.) and boxes (pallet, box, etc.). Second, the following set of conditions has been defined: FitsIn, EmptyHanded, LocationEmpty, ObjectAtLocation, Carrying, Holding, RobotAtLocation.

Learning primitives Learning primitives are needed to extend the robot's knowledge base with some important information about the environment. The learning primitives, in our case, are:

- **object train** - record a snapshot and associate it to an object's type specified from the user
- **grasping pose learn** - learn a grasping pose w.r.t. an object's snapshot
- **placing pose learn** - learn a placing pose w.r.t. a container
- **driving pose learn** - learn a driving pose w.r.t. a container

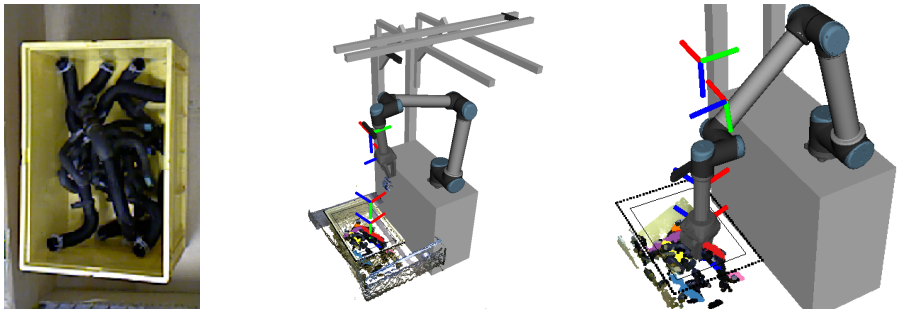
The primitives are executed from the GUI module tab during an initial setup phase of the robot. The snapshot and the poses taught during this phase are then used for all subsequent skills' execution.

7.2 Skills

*The **Drive Skill*** is the simplest skill. It is based on the standard ROS navigation interface, therefore the execution consist of an action call based on the 'move_base_msgs::MoveBaseGoal' message. The details about navigation's implementation are out of the scope of this chapter, but more implementation details can be found in [26].

*The **Picking Skill*** pipeline is organized in several stages, it 1. detects the container (pallet or box) using one of the workspace cameras, 2. moves the wrist camera over the detected container to detect and localize parts, and 3. picks up a part using predefined grasps. Low cycle times of roughly 40 s to 50 s are achieved by using particularly efficient perception components and pre-computing paths between common paths to save motion planning time. Fig. 15 shows examples of part picking using different mobile manipulators in different environments. The picking skill distinguishes two types of storage containers: pallets in which parts are well separated and boxes in which parts are stored in unorganized piles. Internally, the two cases are handled by two different pipelines which, however, follow the same three-step procedure as mentioned above. In case of pallets, we first detect and locate the horizontal support surface of the pallet and then segment and approach the objects on top of the pallet for further object recognition, localization and grasping [3]. For boxes, we first locate the top rectangular edges of the box and then approach an observation pose above the box center to take a closer look inside and to localize and grasp the objects in the box [27]. In the following, we will provide further details about these two variants of the picking pipeline and how the involved components are implemented as a set of primitives in the SkiROS framework. An example of this three-step procedure for grasping a part from a transport box is shown in Fig. 12.

*The **Placing Skill*** is responsible for reliable and accurate kitting of industrial parts in confined compartments [28]. It consists of two main modules the *arm motion* and the *kit locate*. The first is responsible for reliable planning and execution of collision-free trajectories subject to task-depended constrains, while the *kit locate* is responsible for the derivation of the kitting-box pose. The high precision and reliability of both is crucial for a successful manipulation of the objects in the confined compartments of the kitting box.



(a) Scene (b) Approaching the observation Pose (c) Grasping the part

Fig. 12. Example of grasping a tube connector from a transport box (a): after detecting the box and approaching the observation pose (b), the part is successfully grasped (c).

7.3 Primitives

*The **locate** primitive* is one of the perception components in the picking pipeline and locates the horizontal support surfaces of pallets. In addition it segments the objects on top of this support surface, selects the object to grasp (object candidate being closest to the pallet center) and computes an observation pose to take a closer look at the object for recognition and localization. The detection of the horizontal support surface is based on very fast methods for computing local surface normals, extracting points on horizontal surfaces and fitting planes to the extracted points [3]. In order to find potential object candidates, we then select the most dominant support plane, compute both convex hull and minimum area bounding box, and select all RGB-D measurements lying within these polygons and above the extracted support plane. We slightly shrink the limiting polygons in order to neglect measurements caused by the exterior walls of the pallet. The selected points are clustered (to obtain object candidates), and the cluster being closest to the center of the pallet is selected to be approached first.

After approaching the selected object candidate with the end effector, the same procedure is repeated with the wrist camera in order to separate potential objects from the support surface. Using the centroid of the extracted cluster as well as the main axes (as derived from principal component analysis), we obtain a rough initial guess of the object pose. With the subsequent registration stage, it does not matter when objects are not well segmented (connected in a single cluster) or when the initial pose estimate is inaccurate.

*The **registration** primitive* accurately localized the part and verifies whether the found object is the correct part or not. The initial part detection only provides a rough estimate of the position of the object candidate. In order to accurately determine both position and orientation of the part, we apply a dense registration of the extracted object cluster against a pre-trained model of the

part. We use multi-resolution surfel maps (MRSMAPs) as a concise dense representation of the RGB-D measurements on an object [29]. In a training phase, we collect one to several views on the object whose view poses can be optimized using pose graph optimization techniques. The final pose refinement approach is then based on a soft-assignment surfel registration. Instead of considering each point individually, we map the RGB-D image acquired by the wrist camera into an MRSMAP and match surfels. This needs several orders of magnitudes less map elements for registration. Optimization of the surfel matches (and the underlying joint data-likelihood) yields the rigid 6 degree-of-freedom (DoF) transformation from scene to model, i.e., the pose of the object in the coordinate frame of the camera.

After pose refinement, we verify that the observed segment fits to the object model for the estimated pose. We can thus find wrong registration results, e.g., if the observed object and the known object model do not match or if a wrong object has been placed on the pallet. In such cases the robot stops immediately and reports to the operator (a special requirement of the end-user). For the actual verification, we establish surfel associations between segment and object model map, and determine the observation likelihood similar as in the object pose refinement. In addition to the surfel observation likelihood, we also consider occlusions by model surfels of the observed RGB-D image as highly unlikely. Such occlusions can be efficiently determined by projecting model surfels into the RGB-D image given the estimated alignment pose and determining the difference in depth at the projected pixel position. The resulting segment observation likelihood is compared with a baseline likelihood of observing the model MRSMAP by itself. We determine a detection confidence from the re-scaled ratio of both log likelihoods thresholded between 0 and 1.

The ***arm motion primitive*** exploits many capabilities of MoveIt software[10] such as the Open Motion Planning Library (OMPL), a voxel representation of the planning scene and interfaces with the move-group node. In order to achieve motions that are able to successfully place an object in compartments which in many cases are very confined, we have developed a planning pipeline by introducing two deterministic features which could be anticipated as planning reflexes.

The need for that addition arises due to the stochasticity of the OMPL planners which makes them unstable as presented in preliminary benchmarking tests [28]. The Probabilistic Road-maps (PRM), Expansive-Spaces Tree (EST) and Rapidly exploring Random Tree (RRT) algorithms were evaluated. Based on the results PRM performs better on the kitting task. However, its success rate is not desirable for industrial applications.

Another deterrent factor on motion planning is the Inverse Kinematics (IK) solutions that derive from MoveIt. Although that there exist multiple IK solutions for a given pose, MoveIt functions provide only one which is not always the optimal i.e. the closest to the initial joint configuration. We deal with this prob-

[10] `http://moveit.ros.org`

lem by sampling multiple solutions from the IK solver with different seeds. The IK solution whose joint configuration is closer to the starting joint configuration of the trajectory is used for planning.

The developed planning pipeline achieves repeatable and precise planning by introducing two planning reflexes, the joint and operational space linear interpolations. The first ensures that the robot's joints will rotate as less as possible and can be anticipated as an energy minimization planner. This happens by linearly interpolating, in the joint space of the robot, between the starting and the final configurations. Additionally, the operational space interpolation results to a linear motion of the end-effector in its operational space. This is achieved by performing a linear interpolation between the starting and final poses. Furthermore, the spherical linear interpolation (slerp) is used for interpolating between orientations. This linear motion is desirable for going in the narrow compartments of the kitting box.

The path that is created from the two reflexes is evaluated for collisions, constraint violations and singularities. If any of those happen then the pipeline employs the more sophisticated PRM algorithm for solving the planning problem.

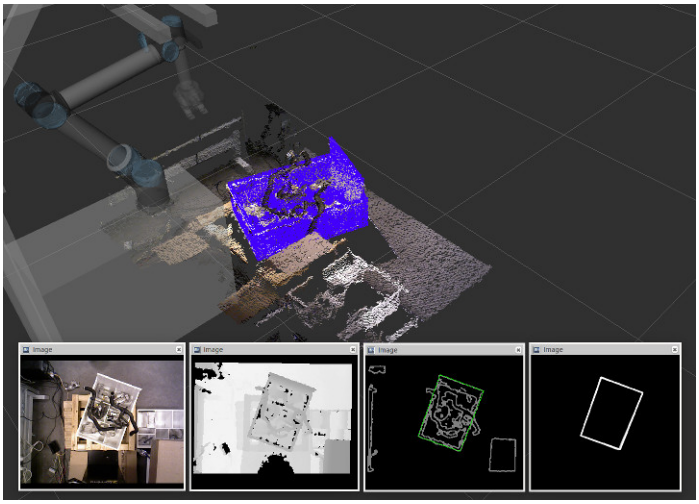


Fig. 13. The 3D scene shows a highly cluttered kitting box with the final registered kitting box (blue) overlaid. The images in the bottom (left to right) show different steps of the registration process: a) raw RGB image b) normalized depth image c) after edge extraction with box candidates overlaid in green d) final voting space for box pixels

The ***kitting box registration*** *primitive* is responsible to locate the kitting box in which compartments the grasped objects have to be placed. Usually, the pose estimation of the kitting box has to be executed whenever a new kitting

box arrives in the placing area of the robot, however, due to slight changes in its position or orientation occurring during the placing task, the kitting box registration can be used as part of any skill, e.g. in the beginning of the placing skill.

For the pose estimation of the box, an additional workspace camera with a top view on the kitting area is used. Due to the pose of the camera and objects which are already placed in the kitting box, most of it is not visible except for the top edges. Additionally, parts of the box edges are distorted or missing in the 3D data caused by the noise of the camera. The kitting box registration therefore implements an approach based on a 2D edge detection on a cleaned and denoised depth image. It applies a pixelwise temporal voting scheme to generate potential points belonging to the kitting box. After mapping these back to 3D space, a standard ICP algorithm is utilized to find the kitting box pose (cmp. Fig. 13). A more detailed description of the algorithm and a performance evaluation is presented in [28].

7.4 Results

We have evaluated the presented architecture with two robotic platforms that operate on different environments. A stationery Universal Robotics UR10 robot that operates within a lab environment and a Fanuc M-20iA which is mounted on a mobile platform and operates within an industrial environment. Both robotic manipulators are equipped with a Robotiq 3-Finger Adaptive Gripper and a set of RGB-D sensors. The drivers that have been used for control of both robotic manipulators and the gripper are available from the ROS-Industrial project[11]

Kitting Operation with UR10 We evaluate the whole kitting task on the UR10 robot with various manipulated objects. The task was planned using the Graphical User Interface presented in Section 4 and is a concatenation of the pick and place skills. Fig. 14 illustrates the key steps of the kitting taks. Detailed results on the performance of the kiting operation can be found in [2, 28]

Kitting Operation with Fanuc on mobile platform Additionally to the UR10 test case, where the robot is stationery and operates in a lab environment, we have applied the presented architecture on a Fanuc robot which is mounted on a mobile base. In this case the kitting operation consists of three skills, the drive, pick and place. Thus, the robot navigates at the location of the requested object, picks it and then places it in the kitting box. This sequence is illustrated in Fig. 15. Using the presented architecture the mobile manipulator is able to perform kitting operations with multiple objects that are located in various spots.

[11] `http://rosindustrial.org/`

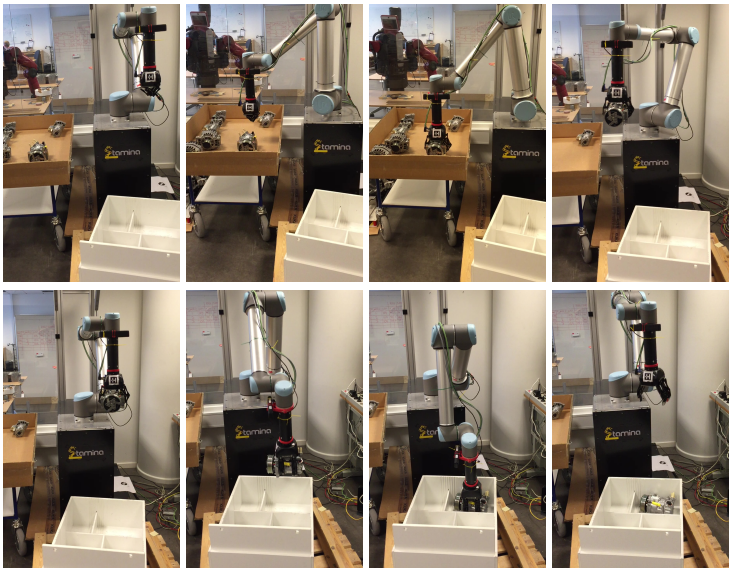


Fig. 14. Sequence of a kiting operation in the lab environment. The top row illustrates the execution of the pick skill and the bottom row the execution of placing skill.

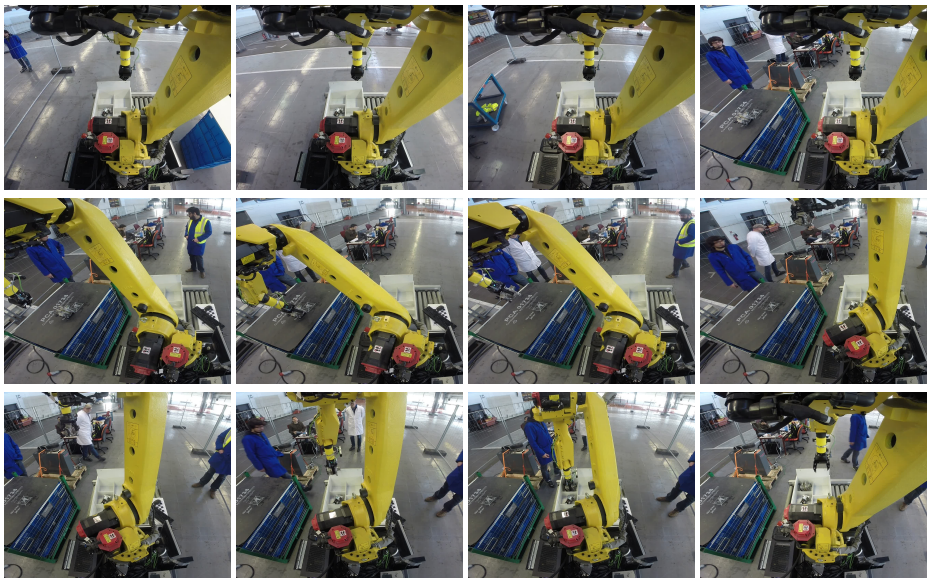


Fig. 15. Sequence of a kiting operation in an industrial environment. The top row illustrates the drive skill, the middle row the picking skill and the bottom row the placing skill.

8 Conclusions

In this research chapter we presented SkiROS, a skill-based platform to develop and deploy software for autonomous robots, and an application example on a real industrial use-case. The platform eases the software integration and increases the robot reasoning capabilities with the support of a knowledge integration framework. The developer can split complex procedures into 'skills', that get composed automatically at run-time to solve goal oriented missions. We presented an application example where a mobile manipulator navigated in the warehouse, picked parts from pallets and boxes, and placed them in kitting boxes. Experiments conducted in two laboratory environments, and at the industrial end-user site gave a proof-of-concept of our approach. ROS joined with SkiROS allowed for porting the pipelines on several heterogeneous mobile manipulator platforms. We believe that using SkiROS, the pipelines can integrate easily with other skills for other use-cases, e.g. for assembly operations. The code released with the chapter allows any ROS user to try out the platform and plan with a fake version of the drive, pick and place skills. The user can then add their own skill definitions and pipelines to the platform and use SkiROS to help implement and manage their own robotics systems.

9 Author Biographies

Francesco Rovida is a Ph.D. student at the Robotics, Vision and Machine Intelligence Lab (RVMI), Aalborg University Copenhagen, Denmark. He holds a Bachelor's degree in Computer Science Engineering (2011), and a Master's degree in Robotic Engineering (2013) from the University of Genoa (Italy). He did his Master's thesis at the Istituto Italiano di Tecnologia (IIT, Genoa, Italy) on the development of an active head with motion compensation for the HyQ robot. His research interests include knowledge representation and software integration for the development of autonomous robots.

Matthew Crosby is a Postdoctoral Research Associate currently working at Heriot Watt University on high-level planning for robotics on the EU STAMINA project. His background is in multiagent planning (PHD, Edinburgh) and Mathematics and Philosophy (MSci, Bristol). More details can be found at mdcrosby.com.

Dirk Holz received a diploma in Computer Science from the University of Applied Sciences Cologne in 2006 and a M.Sc. degree in Autonomous Systems from the University of Applied Sciences Bonn-Rhein-Sieg in 2009. He is currently pursuing the Ph.D. degree at the University of Bonn. His research interests include perceiving, extracting and modeling semantic information using 3D sensors as well as simultaneous localization and mapping (SLAM).

Athanasios S. Polydoros received a Diploma in Production Engineering from Democritus University of Thrace in Greece and a M.Sc. degree with Distinction in Artificial Intelligence from the University of Edinburgh, Scotland. He is currently a Ph.D. student at the Robotics, Vision and Machine Intelligence (RVMI) Lab, Aalborg University Copenhagen, Denmark. His research interests are focused on machine learning for robot control and cognition and model learning.

Bjarne Großmann graduated with a dual M.Sc. degree in Computer Science in Media from the University of Applied Sciences Wedel (Germany) in collaboration with the Aalborg University Copenhagen (Denmark) in 2012. He is currently working as a Ph.D. student at the Robotics, Vision and Machine Intelligence (RVMI) Lab in the Aalborg University Copenhagen. The main focus of his work is related to Robot Perception - from Human-Robot-Interaction over 3D object recognition and pose estimation to camera calibration techniques.

Ronald Petrick is a Research Fellow in the School of Informatics at the University of Edinburgh. He received an MMath degree in Computer Science from the University of Waterloo and a PhD in Computer Science from the University of Toronto. His research interests include planning with incomplete information and sensing, cognitive robotics, knowledge representation and reasoning, and applications of planning to human-robot interaction. His recent work has focused on the application of automated planning to task-based action and social interaction on robot platforms deployed in real-world environments. Dr. Petrick has participated in a number of EU-funded research projects under FP6 (PACOPLUS) and FP7 (XPERIENCE and STAMINA). He was also the Scientific Coordinator of the FP7 JAMES project.

Volker Krüger is a Professor at Aalborg University, Denmark where he has worked since 2002. His teaching and research interests are in the area of cognitive robotics for manufacturing and industrial automation. Since 2007, he has headed the Robotics, Vision and Machine Intelligence group (RVMI). Dr. Krueger has participated in a number of EU-funded research projects under FP4, FP5, and FP6, coordinated the FP7 project GISA (under ECHORD), and participated in the EU projects TAPAS, PACO-PLUS, and CARLoS. He is presently coordinating the FP7 project STAMINA. Dr. Krueger has recently completed an executive education at Harvard Business School related to academic industrial collaborations and knowledge-exchange.

References

1. Pedersen, M.R., Nalpantidis, L., Andersen, R.S., Schou, C., Bøgh, S., Krüger, V., Madsen, O.: Robot skills for manufacturing: From concept to industrial deployment. Robotics and Computer-Integrated Manufacturing (2015) Available online.
2. Holz, D., Topalidou-Kyniazopoulou, A., Rovida, F., Pedersen, M.R., Krüger, V., Behnke, S.: A skill-based system for object perception and manipulation for automating kitting tasks. In: Prof. of the IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA). (2015)
3. Holz, D., Topalidou-Kyniazopoulou, A., StÃ¼ckler, J., Behnke, S.: Real-time object detection, localization and verification for fast robotic depalletizing. In: Proc. of the IEEE/RSJ Int. Conference on Intelligent Robots and Systems (IROS), Hamburg, Germany (October 2015) 1459–1466
4. McDermott, D.: The 1998 ai planning systems competition. Artifical Intelligence Magazine (2000) 21(2):35–55
5. Kortenkamp, D., Simmons, R.: Robotic systems architectures and programming. In Siciliano, B., Khatib, O., eds.: Springer Handbook of Robotics. Springer (2008) 187–206
6. Arkin, R.C.: Behavior-based Robotics. 1st edn. MIT Press, Cambridge, MA, USA (1998)
7. Brooks, R.A.: A robust layered control system for a mobile robot. Journal of Robotics and Automation **2**(1) (1986) 14–23
8. Firby, R.J.: Adaptive Execution in Complex Dynamic Worlds. PhD thesis, Yale University, USA (1989)
9. Gat, E.: On three-layer architectures. In: Artificial Intelligence and Mobile Robots, MIT Press (1998)
10. Ferrein, A., Lakemeyer, G.: Logic-based robot control in highly dynamic domains. Robotics and Autonomous Systems **56**(11) (2008)
11. Bensalem, S., Gallien, M.: Toward a more dependable software architecture for autonomous robots. IEEE Robotics and Automation Magazine (2009) 1–11
12. Magnenat, S.: Software integration in mobile robotics, a science to scale up machine intelligence. PhD thesis, École polytechnique fédérale de Lausanne, Switzerland (2010)
13. Vernon, D., von Hofsten, C., Fadiga, L.: A Roadmap for Cognitive Development in Humanoid Robots. Springer (2010)
14. Balakirsky, S., Kootbally, Z., Kramer, T., Pietromartire, A., Schlenoff, C., Gupta, S.: Knowledge driven robotics for kitting applications. Volume 61., Elsevier B.V. (2013) 1205–1214
15. Björkelund, A., Malec, J., Nilsson, K., Nugues, P., Bruyninckx, H.: Knowledge for Intelligent Industrial Robots. In: AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI. (2012)
16. Stenmark, M., Malec, J.: Knowledge-based industrial robotics. Scandinavian Conference on Artificial Intelligence (2013)
17. Tenorth, M., Beetz, M.: KnowRob: A knowledge processing infrastructure for cognition-enabled robots. The International Journal of Robotics Research **32**(5) (may 2013) 566–590
18. Beetz, M., Mösenlechner, L., Tenorth, M.: CRAM - A Cognitive Robot Abstract Machine for everyday manipulation in human environments. IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings (2010) 1012–1017

19. Rovida, F., Krüger, V.: Design and development of a software architecture for autonomous mobile manipulators in industrial environments. In: 2015 IEEE International Conference on Industrial Technology (ICIT). (2015)
20. Huckaby, J.: Knowledge Transfer in Robot Manipulation Tasks. PhD thesis, Georgia Institute of Technology, USA (2014)
21. Bøgh, S., Nielsen, O.S., Pedersen, M.R., Krüger, V., Madsen, O.: Does your robot have skills? In: The 43rd Intl. Symposium of Robotics (ISR). (2012)
22. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language reference (10 February 2004) Available at `http://www.w3.org/TR/owl-ref/`.
23. Lortal, G., Dhouib, S., Gérard, S.: Integrating ontological domain knowledge into a robotic DSL. Models in Software Engineering (2011) 401–414
24. Krüger, V., Chazoule, A., Crosby, M., Lasnier, A., Pedersen, M.R., Rovida, F., Nalpantidis, L., Petrick, R.P.A., Toscano, C., Veiga, G.: A vertical and cyber-physical integration of cognitive robots in manufacturing. Proceedings of the IEEE **104**(5) (2016) 1114–1127
25. Crosby, M., Rovida, F., Pedersen, M., Petrick, R., Krueger, V.: Planning for robots with skills. In: Planning and Robotics (PlanRob) workshop at the International Conference on Automated Planning and Scheduling (ICAPS). (2016)
26. Sprunk, C., Rowekamper, J., Parent, G., Spinello, L., Tipaldi, G.D., Burgard, W., Jalobeanu, M.: An experimental protocol for benchmarking robotic indoor navigation. In: ISER. (2014)
27. Holz, D., Behnke, S.: Fast edge-based detection and localization of transport boxes and pallets in rgb-d images for mobile robot bin picking. In: Proceedings of the 47th International Symposium on Robotics (ISR), Munich, Germany (2016)
28. Polydoros, A.S., Großmann, B., Rovida, F., Nalpantidis, L., Krüger, V.: Accurate and versatile automation of industrial kitting operations with skiros. In: 17th Conference Towards Autonomous Robotic Systems (TAROS), (Sheffield, UK), 2016
29. Stückler, J., Behnke, S.: Multi-resolution surfel maps for efficient dense 3D modeling and tracking. Journal of Visual Communication and Image Representation **25**(1) (2014) 137–147