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.:	**D3.2**
Title of the deliverable:	**Revised report on skill architecture**

Contractual Date of Delivery to the CEC:	**31.05.2015**
Actual Date of Delivery to the CEC:	**28.05.2015**
.2015	
Organisation name of lead contractor for this deliverable:	**Aalborg University (AAU)**
Author(s):	**Francesco Rovida, Mikkel Rath Pedersen**
Participants(s):	**P01**
Work package contributing to the deliverable:	**WP3**
Nature:	**R**
Version:	**1.0**
Total number of pages:	**28**
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:

This deliverable presents the revised and final architecture of the skill based system SkiROS, one of the core components in the STAMINA project. The deliverable explains the skills, architecture, and integration on the robot and the link to the higher level planning system.

Keyword list: skill based system, software integration, task planning

Document History

Version	Date	Author (Unit)	Description
0.5	May 18, 2015	M.R. Pedersen	Started document based on D3.1
0.6	May 26, 2015	F. Rovida	Revised document
1.0	May 27, 2015	M.R. Pedersen	Finalized document

Table of Contents

1 Introduction

The purpose of this deliverable is three-fold. Firstly, it clarifies the concept of skill-based programming in order to define precisely what this programming paradigm requires and is able to realize. Secondly, it describes the final state of *SkiROS* (Skill Based System for ROS), the architecture designed and developed at AAU for the software integration in the STAMINA project. Finally, it provides some insights on how to program and integrate robot skills into the architecture, that can be executed on the STAMINA robot, and that can function within the skill-based programming paradigm.

Comparing to *D3.1 – Preliminary report on skill architecture*, this document contains updated information about the skill model, world model, infrastructure between the components and SkiROS, integration considerations, and overall presents SkiROS as it will be used for the remainder of the STAMINA project.

In Section 2 we introduce the theory behind the skill-based programming.
In Section 3 we present an overview of the system.
In Section 4 we go in details on how to program skills.

1.1 Integration goals

The software integration goal, in general, is to link together different computing systems and algorithms providing different functionalities, so that they act as a coordinated whole. But this isn't the only goal for the integration software. As it is the linker between algorithms, its utility is also evaluated in terms of how it can support and simplify the coding of new algorithms and increase the code reuse. The import of external algorithms not specifically coded for the system should be straightforward and flexible. Finally, it should support the writing of sustainable software.

In this document, the functionalities to coordinate are called *skills*. Skills are potentially running on board of several robots, and must be made available to a task-level planner. The skills presented to the planner should be at a level of abstraction where the planning algorithms can be almost immediately applied, to concatenate the skills to form a task. Error handling should be done as much as possible from the integration software.

Finally, the code from other STAMINA partners, which may be implemented independently from our system, should be convertible into a skill with little effort, but without shortcuts that could compromise the reliability of the system.

The overall goal of the integration software in the STAMINA project is to ensure that robots are equipped with an adequate set of skills, which allow the planner to concatenate them to reach a given goal from an *a-priori* specified use-case domain. More precisely, the skill framework will:

- create a level of abstraction to both simplify the shop-floor workers programming and allow autonomous planning based on a semantic model of the world,
- load and present the robot skills to the user/planner and coordinate their execution,
- allow the user/planner to see and command the skills execution on different robots in the environment,
- handle skill failures and maximize the skills execution reliability,

- simplify the development and integration of new skills.

2 Skill theory

In this section we clarify the concept and the structure of a skill. We then present what is a world model and how the skills relate to it.

2.1 What are skills

The core idea of robot skills is that they are fundamental software building blocks, concatenated to form complex tasks [2]. The concept is comparable to the manipulation primitive nets, described in [5] or the Object Action Complexes (OAC), described in [4].

The set of skills are found by analyzing standard operating procedures from the factory where the robot is supposed to work. This has two important consequences:

1. The choices for skills can be aligned with human intuition, with the robot actions associated with human language, so that we can have skills called "pick", "place" and "move".
2. We can be sure that the space of possible robot goals is *skill-complete*[2,3].

On this level of abstraction it is much easier for a shop-floor worker to build up a task for a specific goal.

In order for the skills to be useful for task-level programming, they must both change the world state through sensing or action, and be self-sustained, so they can be used in any task. Self-sustainability implies that each skill should be [2]:

- parametric in its execution, so it will perform the same basic operation regardless of input parameter,
- able to identify if a skill can be executed in a specific state of the world, and
- able to verify whether or not it was executed successfully

The skills are "object-centric" in the sense that physical objects are provided as input parameters rather than coordinates (e.g. the object location and pose). Thus, the robot is able to use the same skill on different kinds of objects in different scenarios. Intuitively, the *pick* skill should work equally on all objects known to the scenario, and the skill has sufficiently capable sensing, prior knowledge and control to identify the object and the necessary grasp and to execute the pick.

This makes the skill implementation slightly more complex than usual programming. Eventually, a skill will probably not consist of one single approach that covers the variety of the whole scenario. Instead it will consist of a collection of approaches that are reliable on sub aspects of the scenario, where the exact approach to use is specified on an object-basis, or ideally determined at run-time.

2.2 Skill model

A conceptual model of a complete robot skill is shown in Fig. 1. This conceptual model is the basis on which the skill infrastructure in SkiROS is developed and integrated.

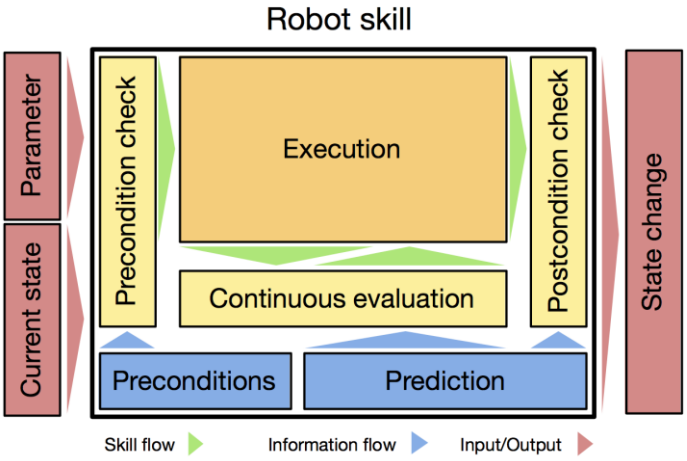

Fig. 1: Complete conceptual model of a robot skill [6]. The preconditions and prediction (blue blocks) are informative aspects of the skill. The checking procedures (yellow) verify the informative aspects before, during and after execution (orange). Execution is based on the input parameter and the world state (red), and the skill effectuates a change in the world state (red).

The *execution* block of the skill is not unlike a traditional robot program. However, the execution needs to be sufficiently general to be applicable within all given use-cases.

The parameters for the skill are two-fold. As skills are applied on objects, only a certain object as a *parameter* is needed, e.g., a "*surface*" or a "*box*" to place an "*object*" on, or an "*object*" to pick up. This is the single parameter that is necessary to supply at task programming time (even if it is possible to specify other parameters). Everything else is handled within the skill. Upon skill execution, the necessary calculations are made in order to successfully execute the skill. It should be noted that running the skill with the same input parameter could result in a different skill behavior, which depends also on the specific scenario constraints. For example, a place skill takes has input parameter only the location, but the shape of the object in the gripper influence the skill result movements.

Since the skills are meant to be the building blocks of programs, they must include *preconditions* and *predictions* that assure proper concatenation. By implementing *pre- and postcondition checking procedures* for these, 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 (pre-condition failures) or if it did not execute correctly (post-condition failures). For instance, the “place” skill would have to verify if the location to place the object is reachable and free. After presumably correct execution, it is verified that the object now occupies the desired placing location. This is determined based on the *prediction* of the skill. A formal definition of pre- and post-conditions 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. This also means that the task planner will be able to replan in the case of skill errors.

2.3 Sensing vs. world-model

The objects that skills operate on may not always be in the field of view of the robots sensors. Therefore, the robot must have some notion of the current state of the world. This can be implemented in multiple ways. Unfortunately, modelling and maintaining a world model (WM) is still an open problem without a standard solution.

A good world model is important for a good skill layer and for task-level planning. It provides an abstraction of the complex environment and contains only the information strictly necessary for completing the desired task goals.

The WM we will use in the STAMINA project is a database of previous detections of objects and locations, stored in a graph structure, and labelled with a precise ID. This allows the robot to search for a specific object or object type. The robot system is also capable of modifying this information, when manipulating an object or detecting that an object is no longer present at a certain location.

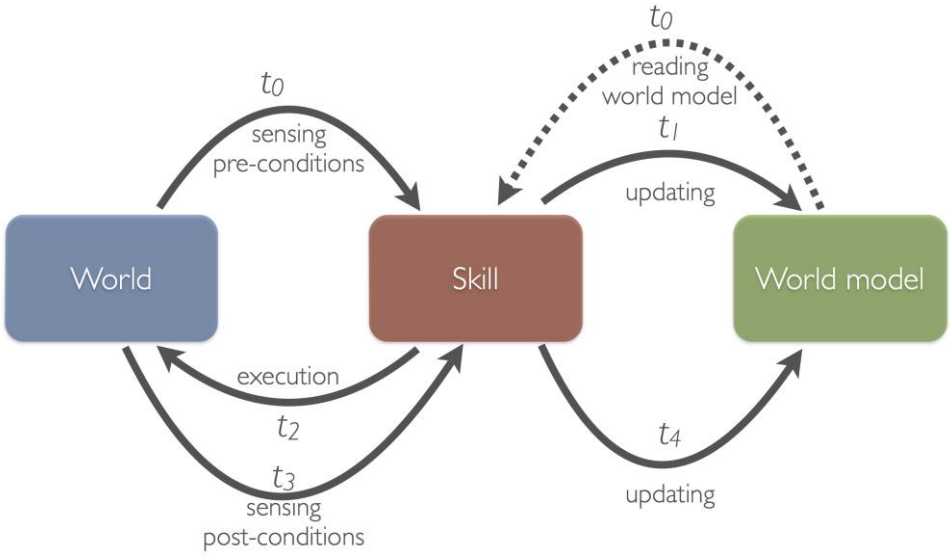

Fig. 2: Information flow during skill execution. First the skill evaluates the pre-conditions by sensing the world (*t0*) and updating the WM (*t1*). It then, executes the action (*t2*) and again senses the world to evaluate the post-conditions (*t3*). Finally, the skill updates the world model with the action effects (*t4*).

From a planning point of view, the skill descriptions and the WM offer a potentially simple mapping into standard planning representations (e.g., PDDL), allowing these structures to be used almost directly with standard planning techniques. E.g., the WM represents the current (initial) world state and the skills are the fundamental actions (rules) available to modify it. Moreover, actions transform the world model from its present configuration to a new state, depending on the input parameters of the skill and the execution context. A planner can use the pre-conditions and post-conditions of the skills to generate an appropriate sequence of actions that can reach a given goal state from an initial state.

Checking pre- and post-conditions is an important task for the robustness of the system, nevertheless sometimes it could result in slowing down too much the skills execution. In special cases, it could be necessary to find a balance between the sensing problem against just assuming the skill succeeds, for example with some lightweight but incomplete checks, or considering the time of last update of the WM data.

3 SkiROS architecture overview

In this section, we present the implementation of the Skill Based System for ROS (*SkiROS)*. We will give an overview of how the software system is structured, and how a skill interfaces with the lower level components of the robot system. Furthermore, we will give details on how SkiROS utilizes a world model as explained in the previous section, and present implementation details regarding the world model associated with SkiROS.

3.1 The integration layers

Integration software can work on different levels of abstraction, going from the low-level, specialized and hard real-time layers to more abstract, user friendly and soft real-time layers. Our software is developed on different levels of abstractions, which are presented in Fig. 3. The core idea is that a complex functionality in one layer is wrapped into fundamental software building blocks, which is then presented to the layer directly above it.

As illustrated in Fig. 3, SkiROS is divided into 4 layers. The purpose of these modular abstraction layers is the possibility to re-use previously developed code, as well as to allow autonomous run-time selection of the best module to use for the problem at hand. The abstraction layers of SkiROS are:

1) *Device layer:* realizes the hardware abstraction and presents an interface to the devices that make up the system.
2) *Primitive layer:* embeds and coordinates *a)* motion primitives, i.e. software blocks that realize a movement controlled with a multi-sensor feedback, and *b)* services, i.e. software blocks that realize a generic computation. The common characteristic of the blocks in this layer is that they do not influence the world state, but only the intrinsic robot state. The current, complete robot state cannot be fully detected from the task layer. This should be expected, as the world model is an advantageous simplification of the physical world.
3) *Skill layer:* embeds and coordinates skills. An integral part of this layer is the *skill managers* that run on the individual robot systems and coordinate the local skill execution.
4) *Task layer:* embeds task-level programming and planning capabilities, and can thus be used in two ways: from the end user or with a planner. In the former, an end-user manually concatenates the skills, in the latter a planner automatically finds a skill sequence for a given goal state of the world model.

In this document two layers are of main importance: the skill layer and the task layer. The task layer is the level where skills are concatenated into complex robot behaviors, e.g. through shop-floor worker programming or through the use of a task-level planner. The task layer interfaces with the skill layer, where skill execution is coordinated. The skill layer allows partners to integrate their own skills easily, without dependencies on low-level *SkiROS* layers.

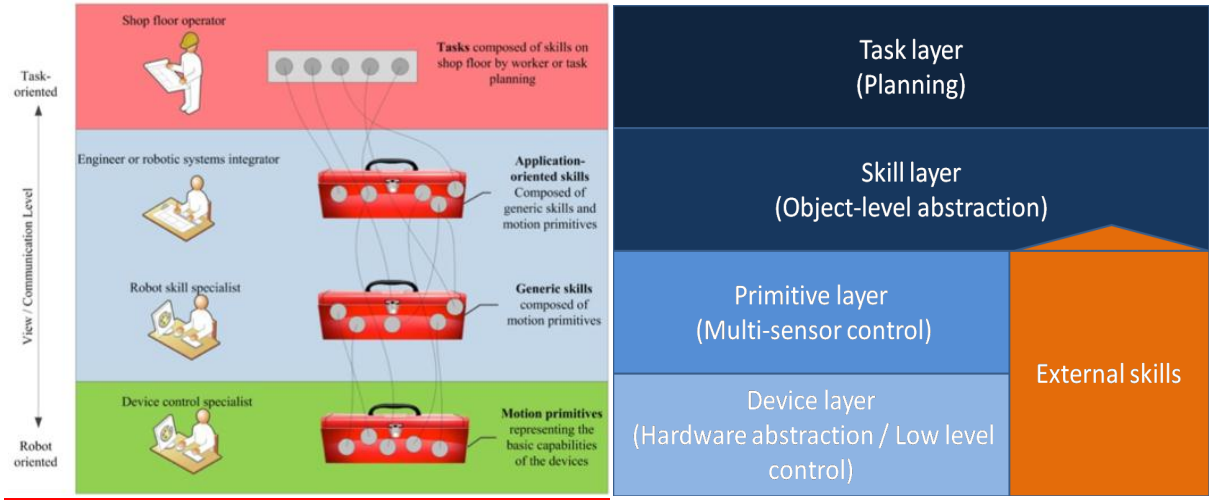

Fig. 3: The figure shows the different layers within *SkiROS*. The Device Layer realizes the hardware abstraction; the Primitive Layer merges the hardware control with sensors feedback. The Skill Layer realizes a concatenation of primitive device properties, based only on semantic information. Finally, the Task Layer plans the skill sequence to achieve the final goal. The orange arrow shows that it is also possible to insert skills that internally don't make use of the primitive layer to control the robot. It should be noted that the green motion primitive layer (left) corresponds to the primitive layer (right). The device layer (right) is not shown in the left image

3.2 ROS implementation structure

SkiROS is a collection of ROS packages implemented in C++, that implements the layered architecture presented in Section 3.1. Each layer is a stand-alone package, which shares few dependencies with other layers.

The packages contained in SkiROS are:

- **SkiROS**
  - **skiros:** the skiros meta-package contains ROS launch files, logs, ontologies, saved istances and scripts to install system dependencies
  - **skiros_resource, skiros_primitive, skiros_skill**: These packages define the structure of the layer corresponding to the package name. Each layer contains a manager and a template for the skill/primitive/device plugins
  - **skiros_world_model**: the world model imports the ontology definition from standard RDF/OWL files and provides and maintains a shared world instance. The robots can store in the world instance the data necessary for operations, and learn new concepts in a teaching phase or during operations by modifying the ontology
  - **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 task layer is the higher SkiROS layer. The task manager node concatenate into sequences the skills provided by different skill managers and coordinates the execution of these. It provides a command line UI to concatenate

skills manually or by mean of a planner. Finally, it provides the interfaces to the WP4 logistic planner.

Note that every layer is associated to a library of plugins separated from the system core. The plugins for every layer are the components that actually implement the real functionalities. The system itself is only a structure created to configure and coordinate these plugins. The plugins are implemented as C++ classes, derived from a single abstract class.

There is only one base class for all skills. Skills (pick, place, etc.) on a single robot are imported into the skill manager as plugins, using the ROS pluginlib package [6].This allows us to implement them in a private ROS package. Since the plugins are imported into the system as shared libraries it is even possible to update and reload them without shutting down the manager. To keep our selection of plugins tidy, they are integrated in a "library" repository, separated from the SkiROS system, named 'skiros_lib'.

- **SkiROS_lib**
  - **skiros_lib_proxies:** plugins for the device layer
  - **skiros_lib_prim**: plugins for the primitive layer
  - **skiros_lib_skills**: plugins for the skill layer
  - **skiros_lib_params**: non-standard types definition for skill's parameters and world elements properties
  - **skiros_lib_reasoners**: utilities to store, process and retrieve data from\to world elements
  - **skiros_lib_planners**: planners for the task layer
  - **skiros_lib_tools**:other utilities

3.3 Skill manager, skills and communication

All available skills are coordinated through a "skill manager". The skill manager is a ROS node running on board the robot, which is in charge of loading the available skills, checking the resources necessary for the execution and maintain communication with the task manager and the world model.

It is in principle possible to have several skill managers within the network, e.g. one for every component (manipulator, navigation platform etc.) of the STAMINA robot, as presented in Fig. 4.

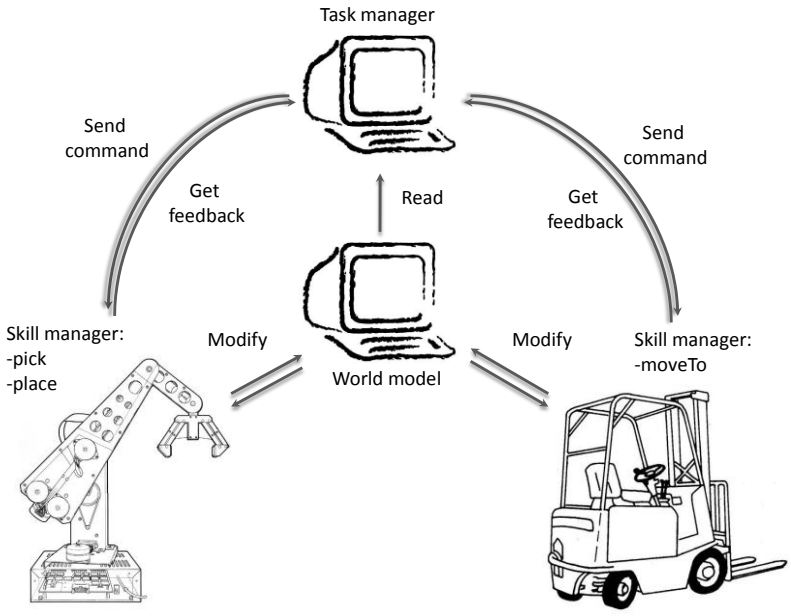

Fig. 4: A graphical representation of the skill managers and the interaction between high-level layers. The planner can communicate with several managers over the ROS network

Every skill manager node in the ROS network will offer the following API:

- a ROS service to query the current list of available skills, and their semantics
- a ROS service to activate a skill's execution block

Similar to a function, a ROS service can receive input and send back an output. At start-up, the skill manager updates the world model with the presence of a new robot with skills. In the robot description it will specify the ROS address to contact it.

A planner, independently from where it is executed, can see the skill managers in the world model, get their ROS addresses from the description, query each skill manager as to which skills it has available, and command a skill to be executed.

Error handling of a skill's execution will be split between the skill manager and the planner. In particular, we foresee the skill manager being in charge of the "quick" error handling and safety checks, while the planner deals with problems that arise due to divergences from expected states in planned action sequences. An advantage of the skill model shown in Sec. 2.1, incorporating predictions and checks of pre- and post-conditions, is the possibility of implementing error handling on the task-level. For example, in the case of skill execution errors (post-condition failures) or failures to actually execute a skill (pre-conditions failures), the task-level planner can be directed to find an alternative sequence of skills that will potentially lead to the desired goal state, possibly involving backtracking on the part of the robot.

3.4 World model

The world model plays a fundamental role in every advanced knowledge-driven system. In fact, all planning systems rely on a world model to organize the knowledge about the objects in the world, their properties and their relations. To build a semantic world model the knowledge to be defined is about:

- *Data*: which kind of properties can be related to elements

- *Elements:* elements set defined into a taxonomy (a hierarchical tree where is expressed the notion about types and subtypes)
- *Relations*: a set of qualitative spatial relations between elements
- *Constraints*: constraints over the world relations and element properties. This should include also tolerances on the constraints.

This set of information is generally referred to as an ontology. In the SkiROS world model, the ontology is defined in the Web Ontology Language (OWL) [7] standard. The OWL file is created using any available editor (e.g. Protègè), then it is parsed and imported into the software. The main advantage of this approach is the fact that a domain expert can modify the world knowledge using a graphical editor with no need to touch the code. Nevertheless, the system designer must develop an interpreter to extract the data from the OWL file and maintain a world model instance. This functionality is already contained in SkiROS, in the **skiros_world_model** package.

An example of a SkiROS world model instance is presented in Fig. 5.This world model (WM) represents only the implementation relevant to SkiROS, and is motivated by the scene-graphs used in computer graphics. The goal of the SkiROS world model is to contain the information necessary for the skills to function, and not all information needed for the complete STAMINA scenario. There is a separate world model used for vertical integration that is used in the Logistic Planner of WP4, of which SkiROS utilizes only a subset of the data, when a task is assigned to the robot.

The SkiROS world model is defined as a Directed Acyclic Graph (DAG), where nodes are world elements and edges define relations (*contain/hasA),* which evaluates to true/false depending on the relation between the *parent* element and a *child* element.

All the information related to a specific object (e.g., the color, position and orientation, or size) is contained into a list inside the node itself. This is discussed more in detail in Section 4.

As the world model will be a shared resource between all skill managers and to some degree the planner, it is available on the network with a static address.

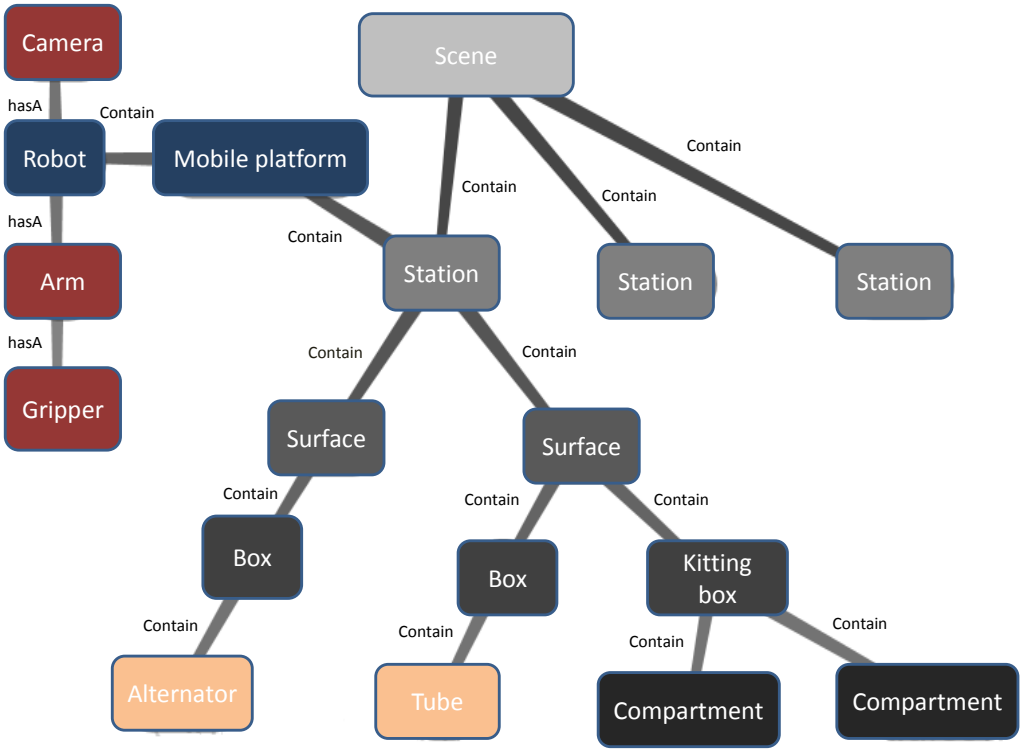

Fig. 5: An example of a world model instance as a Directed Acyclic Graph. Nodes are world elements, containing relevant data, and edges define relations between parent and child nodes.

3.4.1 Prior knowledge

One aspect of the approach, which will influence the set-up time of a new scenario, is the amount of prior knowledge that must be inserted into the model. While some information can be left to the robot to be sensed, other pieces of information must be defined beforehand. This will be defined as part of the vertical integration, and specifically the Logistic Planner in WP4, since the information should ideally come from the Manufacturing execution System (MES). This information includes:

- **Station locations**: Every station must be mapped to a precise position in the map from the navigation system. This might require the robot to move around the environment and set all the stations one by one where necessary. Every station forms a workspace where object poses are saved relative to the station.
- **Surface pose**: At each station, the relative pose and size of the surfaces, like shelves or pallet layers, is necessary for knowing where to point the camera to start segmentation and for checking if the robot is segmenting the right surface. This information can be acquired just after the registration of a new station.
- **Rough location of boxes and objects**: Boxes and objects should be detected from the vision system of the robot. However, rough knowledge about the surface or the box in which they are located is necessary for planning, in the same way it is necessary for a human being. When instructing a person to "take the alternator in the box on shelf 4, at station 1," the rough location of the involved object is already included. This is exactly the information that the robot also needs.

- **CAD models, grasping positions, etc.:** any other relevant information about objects, that is necessary for skill execution should be stored and shared over the network.

4 Skill programming

In this last section we explore what steps are necessary to convert an executable code into a skill. Efforts have been made to design a structure that is easy to integrate new skills, and that minimizes the overhead for the partners in the converting their code. We expect this section will be subject to heavy revisions, and will be frequently updated as long as the system is under development.

4.1 Skill template

As presented in Sec 3.2, the plugin structure of *SkiROS* allows to import skills from every package into the system with little effort. The drawback is that we must code each skill in C++, as the plugin is a class derived from an abstract C++ class.

Every skill must be derived from the base skill template (Appendix A), which fits the model presented in Section 2. In its simplest form, standard code must be divided into three main chunks: *pre-condition-checking*, *execution* and *post-condition-checking*. In detail, the following functions must be implemented:

- *constructor*, specifies the input parameter for the skill
- *onInit(),* defines formal pre-, post- conditions and initialize resources
- *preSense()*
    - check the skill pre-conditions using sensing
    - updates the world model with the newest values from the world
    - upon failure in the pre-conditions, exit the skill and send message to the task-level
- *execution(),*execute the movements
- *postSense()*
    - check the skill post-conditions by using sensing capabilities to sample the relevant parameters in the world
    - updates the world model with the newest values from the world
    - upon failure in the post-conditions, send message to the task-level.

Internally, every skill can use the following interfaces, initialized by the skill manager:

- *getNodeHandle(),*interface to the ROS network
- *getParamHandle(),*interface to define/retrieve parameters
- *getWorldHandle(),*interface to the world model node

A skill, like all most common executables in a complex robotic system, is going to use several other external resources. These resources should report back to the skill in a reasonable period, in order for the skill to update the execution state, check if any interrupt has occurred from the skill manager and,in the pre- and post- condition chunks, to update the WM after any relevant measurement has been done.

In other words: every skill should ideally be in complete control of its own execution and act as a coordinator of resources.

The least invasive solution targeted to integrate partner's code during the project has been found in the *actionlib* package, provided by ROS. This package realizes a client\server coordination system that takes the name of *action*. In this system, the action client (executed by the skill) connects to the action server (provided from an external source) and can request to execute the action, giving a goal. The client can monitor the action progress from a constant feedback message and it is possible to preempt the action execution at any moment. This gives the skill a reasonable control over the external code.

4.2 World model interface

One of the skill's key aspects is that they operate on an abstract world model, which has to been constantly matched to the real world. During the pre- and post-condition check, the skill updates the world model (WM) whenever it receives new information about the environment (see Fig. in Section 2.3). For this reason, the skill needs a read/write interface to the WM. The interface is presented in Appendix B.

In our proposal, there are two main forms of interaction with the world model: element centric and relation centric.

Element centric interaction include addElement, removeElement, getElement and resolveElement functions that are used when the properties associated to an element are needed.

Relation centric interaction works with the IDs of elements and includes functions to modify and query the relations between elements.

The basic data type in the world model is an *element*, a class defined as the following:

```
//Unique Identifier of the element, given when inserted in the world model
intid;
//Defines the element category
std::string type;
//A string identifier (optional)
std::string label;
//The time stamp of the last update
ros::Time last_update;
//A list of properties related to the object (color, pose, size, etc..)
std::map<std::string, skiros_common::Param>properties;
```

The first 3 fields are necessary to retrieve the element in the ontology and the instance database. The last update time stamp can be used to speed up the pre- post-condition checking. The properties list contains all relevant information associated to the object, given a priori or sensed.

It is possible to get a default element, defined a-priori in the ontology, with the command:

```
skiros_common::Element e  =getWorldHandle()->getDefaultElement(individual::my_individual);
```

This will return an element with type and properties set as defined in the ontology. Note that, in the OWL standard, what we define element is referred to as individual.

The id is set to '-1' as long as the element is not inserted in the world instance. The following command add the element to the world instance, and relates it to the parent element with the relation *contain*:

```
e.id()  =getWorldHandle()->addElement(e, parent, relation::contain);
```

As an example of relation centric interaction, consider the world model in Section 3.2, and the place skill. The place skill gets as input a kitting box. Supposing that this is the first time we try to place something in the box, the exact position is not specified in the object description. Using the command:

```
skiros_common::Element surface = getWorldHandle()->getParentElement(kitting_box);
```

we get back the surface element, parent of the kitting box. Suppose now that the position of the surface is specified. This gives us information on where to point the camera. We can now segment the surface and locate the box position.

Once we get the information on box position we update the world model, with the command:

```
kitting_box.properties("Position").setValue(sensed_position);
getWorldHandle()->updateElement(kitting_box);
```

After that the place skill drops the object in the box, and we change the relation between the box and the object, with the following command:

```
getWorldHandle()->setRelation(kitting_box, relation::contain, object);
```

The skiros world model package embeds a tool to generate the enum of types, individuals and relations into the header 'default_uri.h'. This header should be imported and used in all skills in order to detect and remove any name misspelling at compile time.

4.3 Skill's parameters

Every skill requires one or more physical objects to operate on. E.g., a pick skill requires a physical object that can be picked, a place skill requires a box or a surface where the object in the gripper can be placed, etc. Moreover, it is useful to define some parameters for customizing and optimizing skill execution.

The parameter handler is the manager of all parameters required from a skill. The parameters have a *parameter type*, that currently divides in the following types:

- *online*, the main skill's parameters. These parameters must always been set when executing the skill. Since skills are executed on objects, online parameters are usually related to world model elements.
- *offline*, usually are configuration parameters with a default value, such as the desired movement speed, grasp force, or stiffness of the manipulator, and should be taken into account only when strictly necessary

- *hardware*, define the hardware the skill need to access. Same as the online parameters, this information is usually passed to the skill as a world model element, that in the properties should have all the information to access and control the hardware

Every skill can define a customized amount of parameters, which can be a data type or a class.

Online parameters are provided either a) during run-time, b) by the shop-floor worker during the skill-based programming, or c) by a planner during plan generation and execution.

To better understand these concepts, we present a short code example. First, we show how to insert a parameter:

```
getParamHandler()->addParam("myKey", "My name", typeid(myType), skiros_common::online, 3);
```

Here the first line provides a parameter *definition*, where the function arguments specify, in the order: the unique key, a description, the data type, a parameter type, and the number of variables. The second line pushes the parameter into the parameter handler.

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, e.g.,

```
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 skill cannot run until all unspecified parameters are specified.

It is also possible to define parameters with a default value with the following:

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

In this case, the parameter state is set immediately to *specified* and the value will be set to *true*, unless differently indicated.

There is a special case where the skill need the user to specify online a parameter, but wants to limit the decision range. In fact sometimes a skill requires a precise type of element as input. For example, a pick skill cannot pick up an element type *'station'*. In this case, it is possible to use a partial definition. For example:

```
skiros_common::Element e;
e.type = my_type;
getParamHandler()->addParamWithDefaultValue("myKey", "My name", e, skiros_common::online, 1);
```

In this example, only elements of the type (and subtypes) of *'my_type'* will be provided to the skill.

4.4 Pre- and post-conditions

In Section 4.1 we introduced a simple way to implement pre- and post-conditions. A more formal way also existed to define them is presented in this section. The user defines the pre- and post-conditions in the skill constructor, after the parameter definitions. 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 template (Appendix D). In the last case, two functions have to be implemented:

- the constructor, where the description of the condition is defined, together with the necessary, specific, inputs
- the evaluation function, a function that returns a positive value, if the condition is satisfied , zero or negative if not.

Once the condition is defined, it is sufficient to push it inside the list where it will be evaluated automatically before skill execution.

The same considerations apply for post-conditions.

4.5 Discrete reasoners

The world elements are agnostic placeholders where any kind of data can be stored and retrieved within a skill. Their structure is very 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 plugin. Any reasoner is derived from the base class presented in Appendix .

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.

An example of discrete reasoner use is in the following:

```
skiros_wm::ReasonerPtrTypereasoner =
skiros_wm::getDiscreteReasoner("skiros_reasoner::AauSpatialReasoner");
skiros_wm::Element skiros_surface;
skiros_surface.type() = concept::Str[concept::Surface];
tf::Vector3 pos;
pos.setValue(0.5,0.0,0.0);
reasoner->storeData(skiros_surface, pos, "Position");
tf::Quaternion q;
q.setRPY(0.0,0.0,0.0);
reasoner->storeData(skiros_surface, q, "Orientation");
tf::Pose pose = reasoner->getData<tf::Pose>(objObject, "Pose");
```

In this example, we first instantiate the AauSpatialReasoner, developed specifically to reason about position and orientation. It makes use of the 'tf' library, provided in the tf package of ROS. In this simple example, we use the reasoner to store a position and an orientation and to finally get back a pose (a combination of position and orientation).

Appendix A - Skill template

```
namespaceskiros_skill
{

structSkillProgress
{
SkillProgress(): id(0), description(""){}
SkillProgress(intid_in, std::string description_in): id(id_in), description(description_in){}
intid;
std::string description;
};

/*!
/brief Base class for all skills.

   The skills are the atomic software blocks that operate a modification on the world model.
   The skills are derived from this base class and imported in SkiROS using the pluginlib package system.
   Programming inside a skill is not much different from any other C++ program, except for few rules.
   Each skill must define its constructor and overwrite the abstract methods: onInit, preSense, execute and postSense.
   In the constructor and onInit methods the skill should define:
      -A pre- and post-condition set
      -A set of input parameters
   To interact with the rest of the system, the skills are provided with:
      -The ROS node handle, advertise/submit to ROS topics, services and actions
      -The world model interface, offers I\O with the SkiROS world model
      -The parameters handle, to manage parameters
   Internally, the skills can connect to all other ROS nodes as usual.
*/
classSkillBase : publicskiros::ModuleCore
{
protected:
//--------- Polymorphic methods --------------
//These methods must be specialized inside the specific skill
//! Initialize the skill
/*!
     Here should be defined:
      -pre-/post- conditions
      -persistents ROS listeners/advertiser
    */
virtualboolonInit() = 0;
//! Skill's main execution routine
virtualintexecute() = 0;
//! Execute a sensing routine before the pre-condition check
virtualintpreSense() {return 1;}
//! Execute a sensing routine before the post-condition check
virtualintpostSense() {return 1;}

public:
virtual~SkillBase() {}

//--------- SkiROS system methods --------------
//! Check that the parameters are all defined, check pre conditions, set the state to running, then calls the preSense, execute and postSense
functions
voidstart();

//------------ GET -----------------
//! Get the skill progress
SkillProgressprogress()
   {
boost::mutex::scoped_lock lock(state_mux_);
state_changed_ = false;
returnskill_progress_;
   }
protected:
//! Set the skill progress, automatically increase the id
inlinevoidsetProgress(std::string description){ setProgress(SkillProgress(++progress_id_, description)); }
//! Set the skill progress
inlinevoidsetProgress(int id, std::string description)     { progress_id_=id; setProgress(SkillProgress(progress_id_, description)); }
//! Set the skill progress
voidsetProgress(SkillProgress progress)
```

```
  {
boost::mutex::scoped_lock lock(state_mux_);
state_changed_ = true;
skill_progress_ = progress;
state_cond_.notify_all();
  }
inlinevoidaddPrecondition(skiros_wm::Condition* condition) { pre_conditions_.push_back(condition); }
inlinevoidaddPostcondition(skiros_wm::Condition* condition) { post_conditions_.push_back(condition); }
public:
SkillBase() : progress_id_(0) {}
//!< Used to keep track of the skill execution progress. Used also to continue the execution after a skill pause() command
SkillProgressskill_progress_;
intprogress_id_;
private:
//! Check pre-conditions
boolcheckPreConditions();
//! Check post-conditions
boolcheckPostConditions();
//!< Vectors store Pre and Post conditions
std::vector<skiros_wm::Condition*>pre_conditions_;
std::vector<skiros_wm::Condition*>post_conditions_;
};

} //namespaceskiros_skill
```

Appendix B - World model interface

```
namespaceskiros_wm
{
typedefstd::vector<std::pair<int, double>>IdLkhoodListType;
typedefstd::vector<std::pair<std::string, double>>ClassLkhoodListType;
/*! \brief The WorldModelInterface class provides methods to access the world model
 *
 *  This interface is shared between all skills
 */
classWorldModelInterface
{
public:
WorldModelInterface(ros::NodeHandlenh);

//--------- Ontology query methods  ----------
//! Prototype
std::string queryOntology(std::string query_string);
//--------- Relation methods  ----------
//! Modify relations between elements
intsetRelation(intsubject_id, std::string predicate, intobject_id, bool set = true);
//! Query relations between elements
RelationsVectorqueryRelation(int subject, std::string predicate, int object);
//--------- Element methods  ----------
//! Input: element ID. Return: Element description. Error return: Element(unknown) if element matching ID  is not found.
skiros_wm::Element getElement(int id);

//! Input: element Return: a vector of elements matching the given description.
std::vector<skiros_wm::Element>resolveElement(skiros_wm::Element e);

//! Remove the element. Return true if succeed
boolremoveElement(int id);

intaddElement(skiros_wm::Element e, intparent_id, std::string predicate)
  {
RelationsVector relations;
relations.push_back(RelationType(parent_id, predicate, -1));
returnaddElement(e, relations);
  }
/*!
   * \briefaddElement adds an to the world model. The ID will be overwritten with a new one
   * \param e - element to add
   * \param relations - list of relations
   * \return the ID assigned to the object
   */
```

```
intaddElement(skiros_wm::Element e, RelationsVector relations);

intaddBranch(skiros_wm::Element object, intparent_id, std::string relation);

//! Update an element description. The ID must be >0. All properties not specified are not overwritten.
intupdateElement(skiros_wm::Element e);

//! Get an element with the default properties of a specific standard individual
skiros_wm::Element getDefaultElement(std::string individual);

//--------- Identification methods  -------------
/*!
   * \brief classify
   * \param e - element to classify
   * \param threshold - between 0 and 1. Filters the matches below the filter
   * \return a list of pairs -> type-likelihood
   */
ClassLkhoodListTypeclassify(skiros_wm::Element e, float threshold = 0);
/*!
   * \brief identify
   * \param e - element to identify
   * \paramparent_id - limit the comparison to the branch of the parent
   * \param threshold - between 0 and 1. Filters the matches below the filter
   * \return a list of pairs -> id-likelihood
   */
IdLkhoodListTypeidentify(skiros_wm::Element e, intparent_id, float threshold = 0);

//--------- Advanced methods  -------------
/*!
   * \briefgetParentElement
   * \param V1) parent element V2) parent id
   * \param relation
   * \return parent element or element with id -1 if not found
   */
skiros_wm::Element getParentElement(skiros_wm::Element e, std::string relation=""){returngetParentElement(e.id(), relation);}
skiros_wm::Element getParentElement(int id, std::string relation="");
/*!
   * \briefgetChildElements
   * \param V1) parent element V2) parent id
   * \param relation
   * \return vector of child elements
   */
inlinestd::vector<skiros_wm::Element>getChildElements(skiros_wm::Element e, std::string relation="")
{returngetChildElements(e.id(), relation);}
std::vector<skiros_wm::Element>getChildElements(int id, std::string relation="");
/*!
   * \briefremoveBranch remove an entire branch of the world model
   * \param id is the root node id
   */
voidremoveBranch(int id);
inlinevoidremoveBranch(skiros_wm::Element e){removeBranch(e.id());}

//--------- Robot registration methods  -------------
skiros_wm::Element getRobot();
skiros_wm::Element getRobotLocation();
inlinestd::vector<skiros_wm::Element>getRobotHardware()
  {
if(robot_registered_id_>0) returngetChildElements(robot_registered_id_);
elsereturnstd::vector<skiros_wm::Element>();
  }
intregisterRobot(skiros_wm::Element location_id, skiros_wm::Element robot, std::vector<skiros_wm::Element> devices);

voidunregisterRobot();

//--------- Callbacks& utility  -------------
boolisConnected();
voidwmMonitorCB(constskiros_msgs::WmMonitor&msg);
private:
//--------- Private methods  -------------
voidconnectionFailed(std::string service_name);
//--------- Private variables  -------------
                        introbot_registered_id_;
boolconnected_;
```

```
ros::NodeHandlenh_;
ros::ServiceClientquery_ontology_;
ros::ServiceClientquery_model_;
ros::ServiceClientset_relation_;
ros::ServiceClientelement_get_;
ros::ServiceClientelement_modify_;
ros::ServiceClientelement_identify_;
ros::ServiceClientelement_classify_;
ros::Subscriber wm_monitor_sub_;

//------ Tf -------
public:
voidstartTfLissener();
voidstopTfLissener();
boolwaitForTransform(std::string target_frame, std::string  source_frame, ros::Time t, ros::Duration d = ros::Duration(3.0));
voidlookupTransform(std::string target_frame, std::string source_frame, ros::Time t, tf::StampedTransform& transform);

private:
boost::shared_ptr<tf::TransformListener>tf_listener_;
};

}
```

Appendix C - Parameter handler

```
namespaceskiros_common
{
classParamHandler
{
public:
                                        ParamHandler(void) {}
                                        ~ParamHandler() {}

                                        //------- Methods used inside skills -------------
                                        // Add new parameter:
                                        booladdParam(skiros_common::Param p);

                                        // Get parameter value:
                                        template<classT>std::vector<T>getParamValues(std::string key);

                                        //------- Methods used from manager -------------

                                        // Specify value for a parameter (that should already exist)
                                        boolspecify(std::string key, std::vector<boost::any> values);
                                        // Specify value for a parameter exploiting the boost::lexical_cast. Usable only with a limited range
                                        of types
                                        boolspecify(std::string key, std::vector<std::string> values);

                                        // Get parameter:
                                        boolgetParam(std::string key, Param&p);

                                        // Get the parameters map
                                        std::map<std::string,skiros_common::Param>getParamMap();

                                        // Get the map of a specific type of parameters
                                        std::map<std::string,skiros_common::Param>getParamMapFiltered(ParamSpecType type);

protected:
                                        std::map<std::string, skiros_common::Param>params_;

};
}
```

Appendix D - Condition template

```
namespaceskiros_common
{
classCondition
{
```

```
public:

        Condition(WorldModelInterface *wm) : wm_(wm), description_("Undefined") {}

        ~Condition(){}

        virtualboolevaluate()
        {
                returntrue;
        }

        std::string description()
        {
                returndescription_;
        }

protected:
        std::string description_;
        WorldModelInterface *wm_;
};
}
```

Appendix E - Discrete reasoner template

```
namespaceskiros_wm
{
//! /brief This virtual class is the template for all discrete reasoners.
classDiscreteReasoner
 {
public:
virtual~DiscreteReasoner(){}
//! Compare two elements to determine the coefficient of istance similarity (identification)
virtualfloatcomputeIstanceSimilarity(skiros_wm::Element &lhs, skiros_wm::Element &rhs) = 0;
//! Compare two elements to determine the coefficient of class similarity (classification)
virtualfloatcomputeClassSimilarity(skiros_wm::Element &lhs, skiros_wm::Element &rhs) = 0;
//! Convert user data to reasoner data and store it into given element
virtualboolstoreData(skiros_wm::Element & e, boost::any any, std::string  set_code="") = 0;
//! Templatedversion of storeData
template<classT>
boolstoreData(skiros_wm::Element & e, T any, std::string set_code="")
  {
returnstoreData(e, boost::any(any), set_code);
  }
//! Extract data from given element  and return it inside a boost::any
virtual boost::any getData(skiros_wm::Element & e, std::string  get_code="") = 0;
//! Templated version of getData
template<classT>
TgetData(skiros_wm::Element & e, std::string get_code="")
  {
return boost::any_cast<T>(getData(e, get_code));
  }
//! Initialize the element with default properties
virtualvoidaddProperties(skiros_wm::Element& e) = 0;
protected:
DiscreteReasoner(){}
skiros_common::ParamMapparams_;
 };
}
```

Appendix F - Skill example

```
//---------- Mandatory skill's include ----------
#include<pluginlib/class_list_macros.h>//Plugin export library
#include<ros/ros.h>
#include"skiros_skill/skill_base.h"//Base template for all skills
#include"skiros_common/logger_sys.h"//Skiros logging system
#include"skiros_world_model/condition_types.h"//Conditions
#include"skiros_config/param_types.h"//Default parameters
```

```
#include"skiros_config/declared_uri.h"//Default world model URIs (Unified Resource Identifier) -> generated by
skiros_world_model/generate_uri
//---------- Mandatory skill's include end ----------
#include<actionlib/client/simple_action_client.h>
#include<boost/concept_check.hpp>
#include<boost/bind/bind.hpp>
#include<stamina_task_control/TaskGoal.h>
#include<stamina_task_control/TaskAction.h>
#include"skiros_world_model/reasoners_loading_func.h"


namespaceskiros_skill
{
usingnamespaceskiros_config::owl;
usingnamespaceskiros_wm;

classPick : publicskiros_skill::SkillBase
{

private:
boolreceived;
voidfeedbackCb(conststamina_task_control::TaskFeedbackConstPtr&msg)
{
                        this->feedback.state = msg->state;
                        this->feedback.object = msg->object;
                        this->feedback.pose = msg->pose;
this->feedback.chosen_grasp = msg->chosen_grasp;
received = true;
}

voiddoneCb(constactionlib::SimpleClientGoalState& state, conststamina_task_control::TaskResultConstPtr& result)
{
}
voidactiveCb()
{
}
public:
stamina_task_control::TaskFeedbackfeedback;

/*!
    Here should be defined:
     -name, description, version
     -skill's parameters
    */
Pick()
  {
////////////////////////////////////////////////////////
// Specify the skill description constants
////////////////////////////////////////////////////////
this->setSkillType("pick");
this->setDescription("Grasp a manipulable element with a selected gripper");
this->setVersion("0.0.5");

getParamHandle().addParamWithDefaultValue("fakeExe", false, "Fake execution", skiros_common::offline);
////////////////////////////////////////////////////////
// Parameters
////////////////////////////////////////////////////////
getParamHandle().addParamWithDefaultValue("container",skiros_wm::Element(concept::Str[concept::Container]), "Container to get the
object from");

getParamHandle().addParamWithDefaultValue("CameraUp", skiros_wm::Element(concept::Str[concept::Camera]), "Upper workspace
camera", skiros_common::hardware);

////////////////////////////////////////////////////////
// Required Hardware
////////////////////////////////////////////////////////
getParamHandle().addParamWithDefaultValue("gripper", skiros_wm::Element(concept::Str[concept::Gripper]), "Gripper to use",
skiros_common::hardware);
getParamHandle().addParamWithDefaultValue("arm", skiros_wm::Element(concept::Str[concept::Arm]), "arm to use",
skiros_common::hardware);
  }
~Pick() {this->taskClient.reset();}
```

```
/*!
    Here should be defined:
     -pre-/post- conditions
     -persistents ROS listeners/advertiser
    */
boolonInit()
  {
////////////////////////////////////////////////////
// Define the pre conditions
////////////////////////////////////////////////////
// 1. Gripper should be empty
addPrecondition(newskiros_config::LocationEmpty(getWorldHandle(), getParamHandle(), "gripper"));

////////////////////////////////////////////////////
// Define the post conditions
////////////////////////////////////////////////////

// 1. Object should be in gripper
addPostcondition(newskiros_config::LocationFull(getWorldHandle(), getParamHandle(), "gripper"));

////////////////////////////////////////////////////
// Advertise ROS communications
////////////////////////////////////////////////////

this->taskClient.reset(newactionlib::SimpleActionClient<stamina_task_control::TaskAction>(*getNodeHandle(), "/task" ));
                                        this->received= false;
returntrue;
  }

//! Execute a sensing routine before the pre-condition check
intpreSense()
  {
////////////////////////////////////////////////////
// Get parameters
////////////////////////////////////////////////////

//
container_ = this->getParamHandle().getParamValue<skiros_wm::Element>("container");
// Get object to grasp from parameter handler
std::vector<skiros_wm::Element> v = getWorldHandle()->getChildElements(container_);
if(v.size()<=0)return -1;
std::stringObjType = container_.properties(data::Str[data::partReference]).getValue<std::string>();
for(int i=0; i<v.size(); i++)
{
if(v[i].label()==ObjType)
        {
objObject = v[i];
break;
        }
     }
// Get the object place from world model
objSurface = this->getWorldHandle()->getParentElement(objObject.id());
// Get gripper to use from parameter handler
objGripper = this->getParamHandle().getParamValue<skiros_wm::Element>("gripper");

camera_up_ = this->getParamHandle().getParamValue<skiros_wm::Element>("CameraUp");
return 1;
  }

//! Skill's main execution routine
intexecute()
  {
if(getParamHandle().getParamValue<bool>("fakeExe")) returnexecute_fake();
elsereturnexecute_bonn();
  }

//! Execute a sensing routine before the post-condition check
intpostSense()
  {
return 1;
  }

private:
```

```
//Simulated grasping routine
intexecute_fake()
  {
setProgress("Attach object to gripper in world model.");
getWorldHandle()->setRelation(objGripper.id(), relation::Str[relation::contain], objObject.id());
return 1;
  }

//Bonn grasping routine
intexecute_bonn()
  {
//Check the server presence
if(!taskClient->waitForServer(ros::Duration(1.0)))
    {
this->setProgress(SkillProgress(-1, "Task Server seems down. Abort."));
return -1;
    }
std::stringcamera_address = camera_up_.properties("DriverAddress").getValue<std::string>();
                    //Prepare the goal message
stamina_task_control::TaskGoal goal;
ROS_INFO("Object type %s", objObject.label().c_str());
goal.object = objObject.label();
goal.left_side = (camera_address=="top_left_camera");
skiros_wm::ReasonerPtrTypereasoner = skiros_wm::getDiscreteReasoner("skiros_reasoner::AauSpatialReasoner");
tf::Posepose = reasoner->getData<tf::Pose>(objObject, "Pose");
tf::poseTFToMsg(pose, goal.pose);
goal.repeat = false;
goal.task.task = stamina_task_control::TaskID::PICK_OBJECT;

//Send the goal message to task server
setProgress(1, "Sending goal to task server.");
taskClient->sendGoal(goal,  boost::bind(&Pick::doneCb, this, _1, _2), boost::bind(&Pick::activeCb, this)
    , boost::bind(&Pick::feedbackCb, this, _1));
if(!taskClient->waitForResult(ros::Duration(180.0)))
    {
this->setProgress(SkillProgress(-2, "Failed to Contact the Task Server."));
return -2;
    }

//Wait for answer
while(!this->received)sleep(1.0);

//Parse the answer
setProgress(2, "Parsing action answer.");
intreturn_code;
if(taskClient->getState()==actionlib::SimpleClientGoalState::SUCCEEDED)
    {
//Success
tf::Posepose;
tf::poseMsgToTF(feedback.chosen_grasp.pose, pose);
reasoner->storeData(objObject, pose, "Pose");
getWorldHandle()->updateElement(objObject);
this->getWorldHandle()->setRelation(objGripper.id(), relation::Str[relation::contain], objObject.id());
setProgress(3, "Success.");
return_code = 1;
    }
else
    {
//Failure
if(this->feedback.state.value==stamina_task_control::TaskFinishState::UKNOWN_OBJECT)
      {
this->setProgress(SkillProgress(-2, "The requested object has no grasp definition."));
return_code = -2;
      }
this->setProgress(SkillProgress(-2, "The pick skill failed due to unknown reasons."));
return_code = -2;
    }
//Clean
feedback.state.value=stamina_task_control::TaskFinishState::FAILURE;
this->received = false;
//Finish
returnreturn_code;
```

```
  }

skiros_wm::Elementcontainer_;
skiros_wm::ElementobjObject;
skiros_wm::ElementobjSurface;
skiros_wm::ElementobjGripper, camera_up_;
boost::shared_ptr<actionlib::SimpleActionClient<stamina_task_control::TaskAction>>taskClient;
};// class
} // namespace

//Export
PLUGINLIB_EXPORT_CLASS(skiros_skill::Pick, skiros_skill::SkillBase)
```

Bibliography

[1] M. R. Pedersen, L. Nalpantidis, A. Bobick, and V. Krüger, “On the Integration of Hardware-Abstracted Robot Skills for use in Industrial Scenarios,” in *2nd International IROS Workshop on Cognitive Robotics Systems: Replicating Human Actions and Activities*, 2013.

[2] M.R. Pedersen, L. Nalpantidis, R.S. Andersen, C. Schou, S. Bøgh, V. Krüger, O. Madsen, "Robot skills for Manufacturing: From Concept to Industrial Deployment," *Robotics and Computer-Integrated Manufacturing*, 2015

[3] S. Bøgh, M. Hvilshøj, M. Kristiansen, and O. Madsen, “Identifying and evaluating suitable tasks for autonomous industrial mobile manipulators (AIMM),” *Int. J. Adv. Manuf. Technol.*, vol. 61, no. 5–8, pp. 713–726, Nov. 2011.

[4] N. Krüger, C. Geib, J. Piater, R. Petrick, M. Steedman, F. Wörgötter, A. Ude, T. Asfour, D. Kraft, D. Omrčen, A. Agostini, and R. Dillmann, “Object–Action Complexes: Grounded abstractions of sensory–motor processes,” *Rob. Auton. Syst.*, vol. 59, no. 10, pp. 740–757, Oct. 2011.

[5] T. Kröger, B. Finkemeyer, and F. Wahl, “Manipulation primitives—A universal interface between sensor-based motion control and robot programming,” *Robot. Syst. Handl. Assem.*, 2011.

[6] Plugin-lib, http://wiki.ros.org/pluginlib

[7] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, “OWL Web Ontology Language reference,” W3C Recommendation, 10 February 2004, available at http://www.w3.org/TR/owl-ref/

Abbreviations

SkiROS = SkillBased System for ROS

WM = World Model