Project no.: **610917**

Project full title: **Sustainable and reliable robotics for part handling in manufacturing**

Project Acronym: **STAMINA**

Deliverable no.: **D3.1**

Title of the deliverable: **Preliminary report on skill architecture**

Contractual Date of Delivery to the CEC:	**31.07.2014**
Actual Date of Delivery to the CEC:	**30.07.2014**
Organisation name of lead contractor for this deliverable:	**Aalborg University**
Author(s):	**Francesco Rovida**
Participants(s):	**P01**
Work package contributing to the deliverable:	**WP3**
Nature:	**PU**
Version:	**1.0**
Total number of pages:	**22**
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 preliminary architecture of the skill based system, that is, the integration software of the robots functionalities and link to the planning system.

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

Document History

Version	Date	Author (Unit)	Description
0.1	Jun 14, 2014	F. Rovida	Started document
0.2	Jun 29, 2014	V. Kruger	Revision
0.3	Jul 28, 2014	R. Petrick	Revision
1.0	Jul 29, 2014	F. Rovida	Final document

Table of Contents

1 Introduction

The purpose of this paper 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 present state of *SkiROS* (Skill Based System for ROS), the implementation designed at AAU for the software integration in the STAMINA project. Finally, it provides some preliminary insights on how to program robot skills that can be executed on the STAMINA robot, and that can function within the skill-based programming paradigm. In many sections this document presents ideas that should open discussions between the partners to allow revision and improvement. The results of these discussions will be presented in deliverable D3.2 in M20.

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 that may be written independently from our system should be convertible into a skill with little effort, but without short-cuts 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 should:

- Create a level of abstraction to 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 [1]. 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 analysing 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* [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 kind 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, the 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.

2.2 Skill model

Our model of a complete robot skill is shown in Fig. 1.

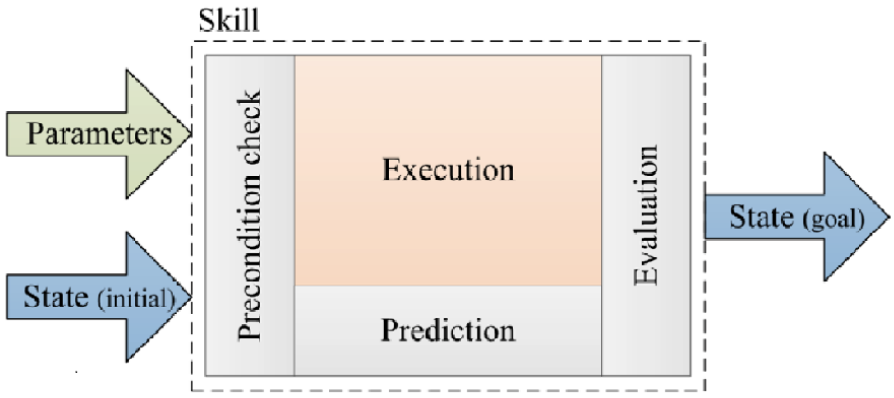

Fig. 1, Model of a robot skill

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. Should be noted that running the skill with the same input parameter can result in a different skill behaviour that 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 *pre-* and *post-conditions* that assure proper concatenation. By implementing a checking procedure for these pre- and post-conditions, 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, e.g. have to verify if the location to place the object is reachable and free. After presumably correct execution, it is verified that the desired location is now occupied by the object. A formal definition of pre- and post-conditions is not only useful for robustness, but also task planning. It is easy to imagine a robot that would replan a task, if either pre-condition or post-condition were not satisfied.

Finally, the *prediction* formally specifies the expected effects of executing a skill. Pre-conditions and predications can thus be used by a task-level planner for selecting appropriate skills to be concatenated for achieving a desired goal state.

2.3 Sensing vs. world-model

The objects that skills operate over 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 (WM) 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 plan to use in the STAMINA project is a database of previous detections of objects and locations, saved in a graph or similar 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.

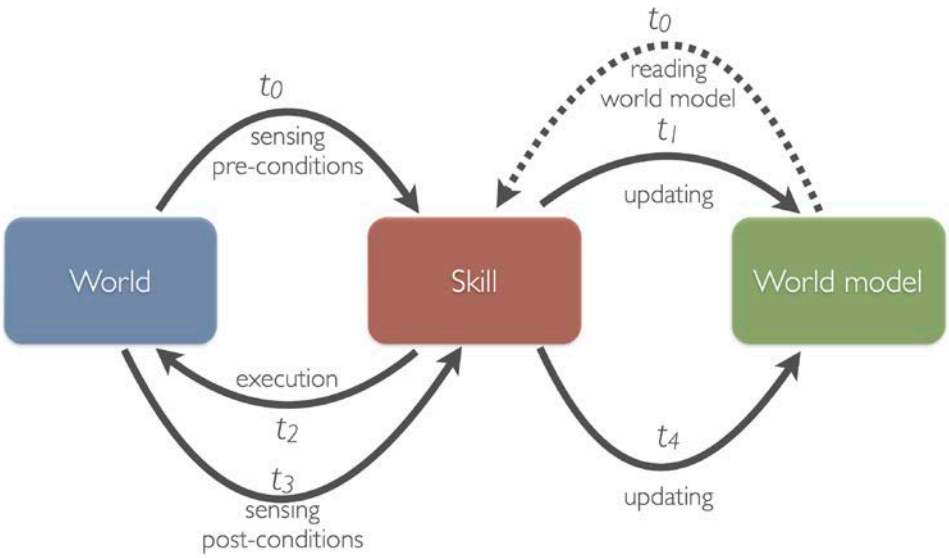

Fig. 2, Information flow during skill execution. First the skill evaluates the pre-conditions by sensing the world (*t0*) and updating the WM (*t1*). It then executes the action (*t2*) and again senses the world to evaluate the post-conditions (*t3*). Finally, the skill updates the world model with the 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 preliminary 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 system. We expect this section will be subject to heavy revisions and will be continually updated as long as the system is being developed. In particular, this section outlines our roadmap for system development and offers the STAMINA partners the possibility to contribute and participate.

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. 1. 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.

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 behaviours, 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.

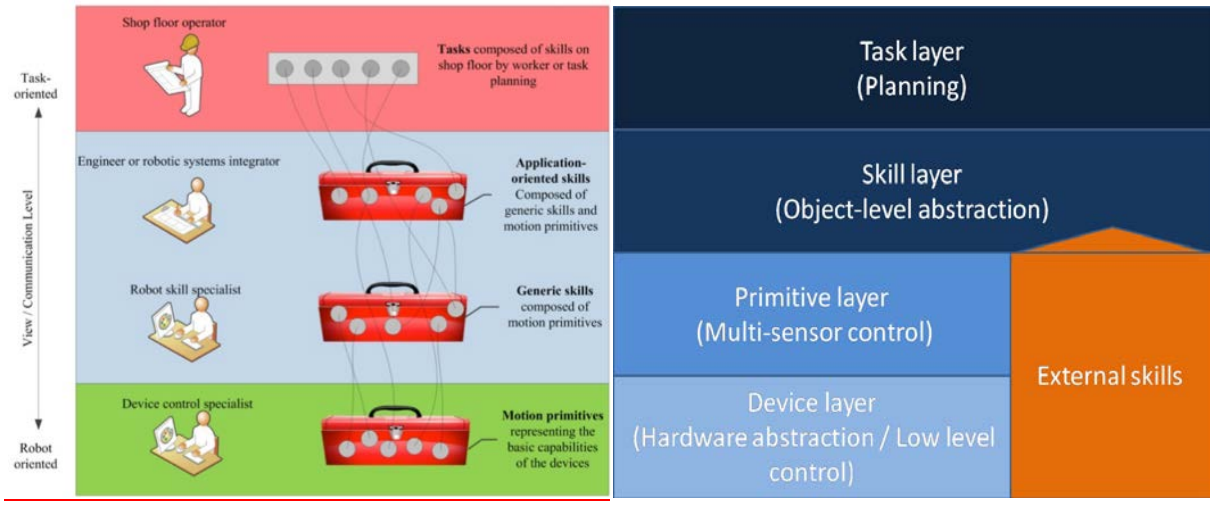

Fig. 3, The figure shows the different layers within *SkiROS*. The Device Layer realizes the hardware abstraction, the Primitive Layer merges the hardware control with sensors feedback. The Skill Layer realizes a concatenation of primitive device properties, based only on semantic information. Finally, 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 The repository structure

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

The package contained in the repository is:

- **SkiROS**
  - **skiros_device, 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 plugins
  - **skiros_world_model**: preliminary world model implementation
  - **skiros_common**: shared classes. Parameters, Parameters Handler, world element, world model interface, condition template
  - **skiros_msgs**: shared ROS actions, services and messages
  - **skiros_config**: contains standard parameters, standard elements and standard conditions definition.
- **SkiROS lib**
  - **skiros_lib_proxy:** plugins for the device layer
  - **skiros_lib_prim**: plugins for the primitive layer
  - **skiros_lib_skill**: plugins for the skill layer

We can see that every layer has 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, with the possibility of informing the skill manager about the plugin. 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, we created a "library" repository, named 'skiros_lib'.

We now describe the skill layer in more detail.

We note that the terms “skill” and “plugin” may sometimes be used interchangeably, since a skill is always imported into the system as a plugin.

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 if they have the resources necessary for execution and communicating the information over the ROS network.

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.

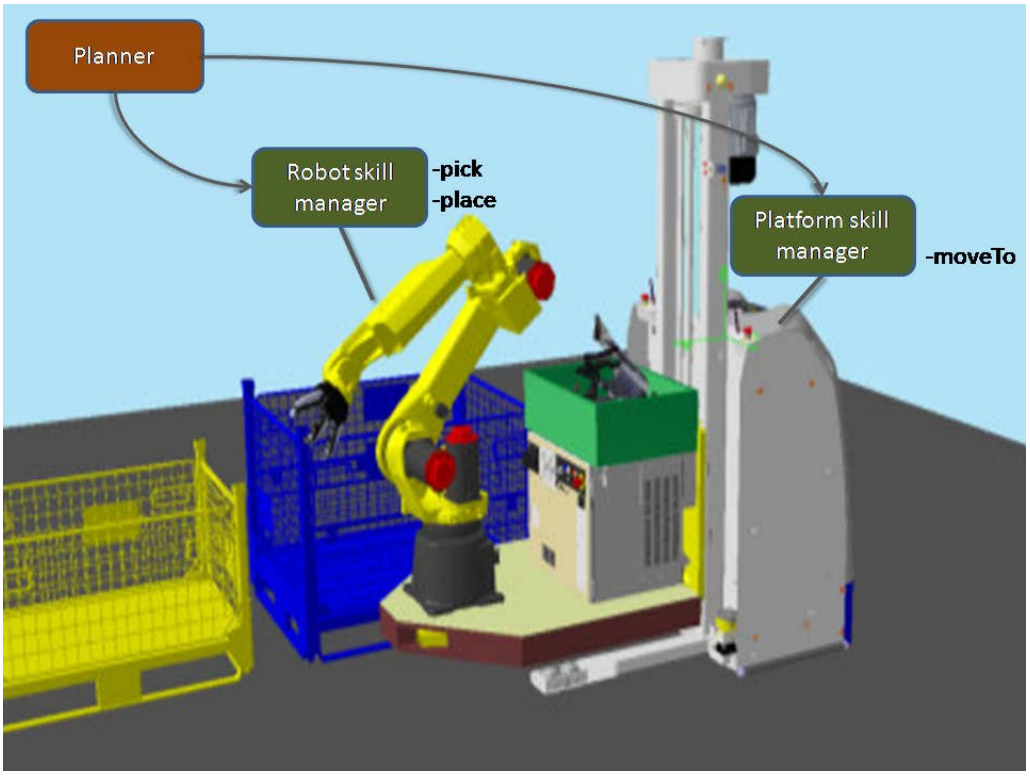

Fig. 4, A graphical representation of the skill managers. The planner can communicate with several managers over the ROS network.

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

A preliminary world model that was used in the first test sprint is presented in Fig. 5. Please note that this world model (WM) represent only a first, ad-hoc, implementation which was motivated by the scene-graphs used in computer graphics. All of the details we present in this section are simply suggestions that are open to discussion and change in the future. In our present implementation, the WM is defined as a graph, where nodes are world elements and edges define a relation *contain*, which evaluates to true/false depending on whether a parent element contains the child element.

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

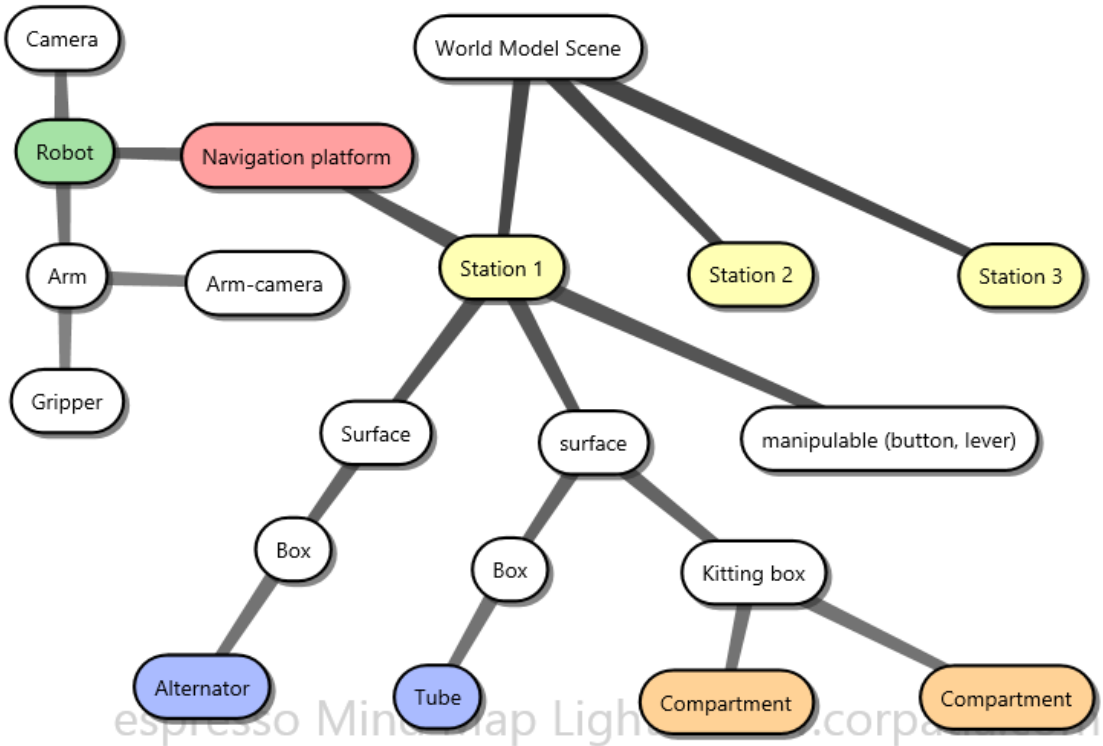

Fig. 5, An example of a WM scene. The nodes are elements and the edges are *contain* relations

The most suitable world model must be discussed between the STAMINA partners.

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

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. A preliminary list 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 robot base frame.

- **Surface pose**: At each station, the precise relative pose and size of the surfaces, like shelves, 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.
- **Objects and boxes rough position**: 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 an object someone would probably say a sentence like "take the alternator in the box on shelve 4, at station 1".This is exactly the information that the robot also needs.
- **CAD models, grasping positions, etc.**: any other relevant information about objects should be stored and shared over the network

4 Skill programming

In this last section we explore what steps are necessary for converting a standard piece of 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 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 us 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
- **preConditions()**
  - 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
- **postConditions()**
  - 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.

Technically, every function can simply be implemented as a call to a ROS service of a ROS node which contains the real code. However, we strongly discourage this kind of implementation, since it doesn't allow the skill manager to keep track of the execution state, and to stop the skill execution in case of problems.

A skill can also make use of external resources, but these resources should report back to the skill in a reasonable period, in order to update the execution state and check if any interrupt has occurred from the skill manager. Moreover, in the pre- and post- condition chunks, the skill has the important task of updating the WM after any relevant measurement has been done.

In other words: every skill should ideally be in complete control of its own execution.

The code structure presented here is simple and meant to allow easy conversion of new code without major changes. However, in terms of code reusability, there also exists the possibility of using skill primitives, and a formal way of implementing pre- and post-condition checks. These ideas are analysed briefly in the following sections.

4.2 Parameters and elements

Every skill requires one or more physical objects that it operates 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 holds all the parameters that the skill needs as input from the user/planner. These parameters are divided into offline and online parameters. **Every skill can define a customized amount of parameters, which can be a data type or a class**.

Offline parameters are parameters that configure the general skill execution, such as the desired movement speed, grasp force, or stiffness of the manipulator. These parameters are usually defined a priori with standard values, but can be replaced with more optimal values by an expert programmer at a later time.

Online parameters are mainly related to elements in the world model. For example, a place skill will usually require an object in the gripper and a location to put the object. "Gripper", "Location" and "Object" are all defined as *elements* of the world model.

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 demonstrate how to insert an offline parameter:

```
skiros_common::Param p("myKey", "My name", typeid(double), skiros_common::offline, 3);
ph_.addParam(p); //ph_ is the parameter handler (private variable defined in the skill template)
```

Here the first line provides a parameter *definition*, where the function arguments specify, in order: a key for later access, a name, a 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 offline 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 = ph_.getParamValues<double>("myKey");
```

Inside the skill, the implementation knows exactly which type of variables we stored in the parameter handler, so it is not a problem to cast them back, e.g., as a double in the above example. (From the skill manager's point of view the situation is potentially more complex; however, this issue should not affect the skill developer.)

The parameter state is defined as "*initialized*" until its value is specified by the user. After this is done, the state changes to "*specified*". A skill cannot run until all parameters are specified. This property is particularly important for the element's properties, as we will see.

The definition of an online parameter is the same as an offline one except for the parameter type (online/offline).

Our preliminary definition of an element of the world model is a class with 4 variables:

```
//Unique Identifier of the element, given when inserted to world model
int id;
//Type of the element. Types defined into configuration file
std::string type;
//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 properties list contains all the relevant information about the object that can be given a priori or be sensed. The time stamp of the last update can be used to speed up the pre- post-condition checking. We also define a preliminary list of elements types, many of which can be seen in the world model presented in Section 3.2. Each element type has an associate property set. For example, in our definition, an *object* has the following standard list of properties (name->data type):

- Size: tf::Vector3
- Position: tf::Vector3
- Orientation: tf::Quaternion

The definition of the standard parameters and standard elements is contained in "skiros_config" package. In particular, this definition is deliberately designed to be flexible and can be used to implement a standard list of elements and a standard set of properties related to every element, once such a list is agreed upon by STAMINA partners.

As above, we note that an element can have a standard property in its description (e.g., “Position”), but this parameter will not immediately be in a "*specified*" state. Instead, the parameter will remain "*initialized*" until a value is specified by a user.

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 define the type of element the skill requires as input as part of its skill definition. For example:

```
skiros_config::ElementTypes elem_types = skiros_config::ElementTypes::getInstance();
skiros_common::Param p ("Obj", "Object", typeid(skiros_common::Element),
                                                    skiros_common::online, 1);
p.setValue(elem_types.getDefault("Object"));
ph_.addParam(p);
```

In this example, only elements of type "object" can be provided to the skill: the first line gets an instance of the singleton "skiros_config::ElementTypes", which contains the definitions of standard elements; the second line defines a single online parameter, of type " skiros_common::Element"; the third line set the value of the parameter; while the last line pushes the value into the parameter handler.

4.3 World model interface

During the pre- and post-condition check, the skill must update 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. A preliminary proposal for such an interface is presented in Appendix B; however, this interface is subject to change as the world model structure becomes further defined.

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

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

Relation centric queries works with the IDs of elements and are used to modify and query the relations between elements.

As an example, 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 = world_model_.getParentElement(kitting_box.id);
```

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);
world_model_.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:

```
world_model_.setRelation(kitting_box, skiros_common::contain, object);
```

4.4 Pre- and post-conditions

In Section 4.1 we introduced a simple way to implement Pre- and post-conditions, and we said that a more formal way also existed to define them. In this second case, 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. The error codes also have to be defined.

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 Primitives

It is possible, but not absolutely mandatory, to use primitives inside the skill code that call low-level functionalities provided by the *SkiROS* primitive layer. In this case, the required primitives should be identified in the primitive list. Access to primitives is still under development and will be presented in a successive deliverable, if such functionality is required on STAMINA.

Appendix A - Skill template

```
namespace skiros_skill
{
class SkillBase : public SkillCore
{
protected:
        //--------- Polymorphic methods ----------
        // Execute the skill
        virtual int execute() = 0;
public:
        virtual ~SkillBase() {}

        //Pass to the skill the ROS node handler and the interface to the world model.
        virtual bool init(ros::NodeHandle * nh, skiros_common::WorldModelInterface * world_model);

        //Optional way of defining pre-, post-conditions
        virtual int preconditions() {}

        virtual int postconditions() {}


        //--------- Non Polymorphic methods ----------
        //Set the parameters (it is mandatory to set the parameters before execution). The specification
        process is handled by the skill manager
        bool setParams(skiros_common::ParamHandler ph);
        //Get the parameter handler of the skill
        skiros_common::ParamHandler getParamHandler();
        //This check the parameters are all defined, check pre conditions, set the state to running, then calls the execute
        function
        int start();

        bool checkPreConditions();

        bool checkPostConditions();

protected:
        SkillBase() : param_are_specified_(false) {}

        //Pointers to interface with the world model and ROS system
        skiros_common::WorldModelInterface * world_model_;
        ros::NodeHandle *  nh_;

        //Holds the set of parameters necessary for the skill
        skiros_common::ParamHandler ph_;

        //Used to keep track of the skill execution progress.
        int skill_progress_;
        int skill_max_progress_;

        //Set to true when all parameters have been specified
        bool param_are_specified_;

        //To set only when SkiROS Primitive are used inside the skill
        std::vector<std::string> mandatory_primitives_;
        std::vector<std::string> optional_primitives_;

        //Vectors to store Pre and Post conditions
        std::vector<skiros_common::Condition*> pre_conditions_;
        std::vector<skiros_common::Condition*> post_conditions_;
};
}
```

Appendix B - World model interface

```
namespace skiros_common
{
class WorldModelInterface
{
public:
        WorldModelInterface();


        //--------- Basic ID focus methods  ----------
        //Modify relations between elements
        int setRelation(int subject_id, Predicate predicate, int object_id, bool remove = false);
        //Query relations between elements
        std::vector<int> query(int subject, Predicate predicate, int object);

        //--------- Basic element focus methods  ----------
        //Input: element ID. Return: Element description. Error return: Element(unknown),if element matching ID  is not found.
        Element getElement(int id);
        // Input: element Return: a vector of IDs of elements matching the given description.
        std::vector<Element> resolveElement(Element e);
        // Remove the element. Return true if succeed
        bool removeElement(int id);
        //THe object ID is assigned when the object is added to the world model and returned to the caller. If the provided
        object already has an ID the system update the existing description
        int addElement(Element e, int parent_id);
        //Update element description
        int updateElement(Element e);

        //--------- Hybrid methods  ----------
        //Input: element ID. Return: Parent element description. Error return: Element(unknown), if element matching ID is not
        found.
        Element getParentElement(int id);

};
}
```

Appendix C - Parameter handler

```
namespace skiros_common
{
class ParamHandler
{
public:
                ParamHandler(void) {}
                ~ParamHandler() {}

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

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

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

                // Specify value for a parameter (that should already exist)
                bool specify(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
                bool specify(std::string key, std::vector<std::string> values);

                // Get parameter:
                bool getParam(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

```
namespace skiros_common
{
class Condition
{
public:

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

        ~Condition(){}

        virtual bool evaluate()
        {
                return true;
        }

        std::string description()
        {
                return description_;
        }

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

Appendix E - Skill example

```
#include <pluginlib/class_list_macros.h>
#include <ros/ros.h>
#include <boost/any.hpp>
#include "skiros_skill/skill_base.h"
#include "skiros_config/element_types.h"
#include "skiros_config/condition_types.h"
#include "skiros_common/logger_sys.h"

namespace skiros_skill_test
{
class Test : public skiros_skill::SkillBase
{
public:
Test()
{
        //-------------------------------------------------------
        //Specify the skill description constants
        //-------------------------------------------------------
        this->name_ = "Test skill";
        this->description_ = "This skill is just a showcase of implementation";
        this->version_ = "0.0.2";
        //-------------------------------------------------------
        //Define the parameters needed from the skill
        //-------------------------------------------------------
        skiros_common::Param p("a0", "A Double", typeid(double), skiros_common::offline, 2);
        ph_.addParam(p);

        p.reset("a1", "An Integer", typeid(int), skiros_common::offline, 1);
        ph_.addParam(p);
```

```
                p.reset("Loc", "Location", typeid(skiros_common::Element), skiros_common::online, 1);
                ph_.addParam(p);
                //Getting a standard element, example
                //Note: this pre-definition of the object limits the user to choose the input element within a  constrained set of
                elements in the world model
                skiros_config::ElementTypes elem_types = skiros_config::ElementTypes::getInstance();
                p.reset("Obj", "Object", typeid(skiros_common::Element), skiros_common::online, 1);
                p.setValue(elem_types.getDefault("Object"));
                ph_.addParam(p);
                p.reset("Grip", "Gripper", typeid(skiros_common::Element), skiros_common::online, 1);
                p.setValue(elem_types.getDefault("Gripper"));
                ph_.addParam(p);
                //-------------------------------------------------------
                //Define the pre/post conditions
                //-------------------------------------------------------
                //Define as precodintion that the object is in a certain location
                skiros_config::ObjectAtLocation *pre_cond1 = new skiros_config::ObjectAtLocation(world_model_, ph_, "Obj", "Loc");
                pre_conditions_.push_back(pre_cond1);
                //Define as postcondition that the object is in the gripper
                skiros_config::ObjectAtLocation *post_cond1 = new skiros_config::ObjectAtLocation(world_model_, ph_, "Obj", "Grip");
                post_conditions_.push_back(post_cond1);
}
~Test() {}
//Pass to the skill the ROS node handler and the interface to the world model
//NOTE: is not necessary to declare this function, however if you need any persistent listener to ROS topics you can define it here
/*bool  SkillBase::init(ros::NodeHandle & nh, skiros_common::WorldModelInterface * world_model)
{
                nh_ = nh;
                world_model_ = world_model;
                return true;
}*/
private:
int execute()
{
                //-------------------------------------------------------
                //Getting back parameters, example
                //-------------------------------------------------------

                std::vector<double> val0 = ph_.getParamValues<double>("a0");
                if(val0.size()>0) {FINFO("Got a0");}
                else FINFO("a0 not found");
                std::vector<int> val1 = ph_.getParamValues<int>("a1");
                if(val1.size()>0) {FINFO("Got a1");}
                else FINFO("a1 not found");

                //-------------------------------------------------------
                //Getting back objects, example
                //-------------------------------------------------------

                std::vector<skiros_common::Element> location = ph_.getParamValues<skiros_common::Element>("Loc");
                if(location.size()>0) {FINFO("Got location");}
                else { FINFO("No location"); return -1; }

                std::vector<skiros_common::Element> object = ph_.getParamValues<skiros_common::Element>("Obj");
                if(object.size()>0) {FINFO("Got object");}
                else { FINFO("No object"); return -1; }

                std::vector<skiros_common::Element> gripper = ph_.getParamValues<skiros_common::Element>("Grip");
                if(gripper.size()>0) {FINFO("Got Gripper");}
                else { FINFO("No Gripper"); return -1; }

                //Modify the World model, example
                FINFO("Pick up object");
                if(world_model_->modify(gripper[0].id, skiros_common::contain ,object[0].id)) {FINFO("Object picked");}
                else FINFO("Picking failed");

                return 1;
}
};
};
//Export
PLUGINLIB_EXPORT_CLASS(skiros_skill_test::Test, 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] S. Bøgh and O. Nielsen, “Does your robot have skills?,” *43rd Intl. Symp.*, 2012.

[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

Abbreviations

SkiROS = Skill Based System for ROS

WM = World Model