Design and development of a software architecture for autonomous mobile manipulators in industrial environments

Francesco Rovida, Volker Krüger
Robotics, Vision and Machine Intelligence Lab,
Aalborg University of Copenhagen
AC Meyers Vaenge 15  DK-2450 Copenhagen
Email: {francesco, volker}@m-tech.aau.dk

Abstract—During the last decades, methods for intuitive task level robot programming have become a fundamental point of interest for industrial applications. A robot programming framework is needed to facilitate task-level programming of mobile manipulators, e.g. by providing the robot with a set of movement primitives and skills. Robot skills have already been used and tested successfully within the FP7 project TAPAS for this purpose, and are presently used in several new FP7 projects (e.g. CARLOS, STAMINA, ACAT). To meet the goal of programming mobile manipulators on-the-fly inside industries we need to bring together the actual skills and primitives with other aspects such as task planning, an extended knowledge integration framework and a control infrastructure for the skill execution. In this paper we want to discuss an architecture that organizes clearly the software processes on the different abstraction levels and propose a world model that ground abstract reasoning to concrete skills. Our aim is to develop a software framework to get towards the implementation of flexible and highly modular cognitive robot tasks.

Keywords: Autonomous mobile manipulator, Skill-based system, Industrial environment, World model, Knowledge integration framework

I. INTRODUCTION

Task-level programming is a way of programming a robot on an abstract level. In this programming paradigm the human programmer is required to specify what the robot should do in terms of actions on the objects involved in the task and not necessarily to the full extent of how it should be achieved. This promises to decrease the amount of competences necessary to deal with complex robots and it is a powerful way to speed up the programming process and the code reuse. It is realized by creating an abstraction which hides the complexity of the lower layers and allows to focus on the task itself. In earlier papers [1], [2] we defined tasks as a sequence of skills where skills are identified as those re-occurring actions that are needed to execute standard operation procedures. The skills are comparable to the manipulation primitive nets, described best in [3]. These nets are sequences of manipulation primitives, with simple decisions based on the outcome of each single manipulation primitive in the net. In these nets each manipulation primitive needs its own explicitly specified parameters, e.g. a specific force or velocity in a specific direction. This, however, makes this particular implementation unsuitable for robotics novices in a factory hall. Instead, our skills propose a method for specifying only high-level parameters, while low-level parameter for the lower level primitives are mostly inferred through autonomous reasoning by the robot. This also brings the skill names more close to human intuition, e.g., *pick* and *place*.

Even though the skills have an important application to facilitate robot programming, they are not limited to this use. In more recent work [4] we show how skills can also be used successfully in conjunction with a planner. A task-planner is briefly an algorithm that reasons on an abstract model of the world. Knowing the initial world state and a declared set of feasible actions, the algorithm is able to plan the sequence of actions necessary to bring the world model from the initial state to the goal state. The skills proven to be a good structure to ground the planner abstract reasoning to a real robot movement. While the use of robot skills is not new, it is still an open question of how to program a robot so that it uses the skill paradigm to its full potential. This means, the robot software need to have an infrastructure to link the abstract domain of the planner to a real, controlled and predictable execution of the skills, that organize the world knowledge appropriately, that allows skills scalability and portability across different robots and that opens to robot cooperation.

We started to develop a reference implementation under the name *SkiROS* (Skill-based system for ROS), a robot software architecture tailored for the specific needs of industrial projects, which implements the skill paradigm within a modular and scalable structure. As also the name suggest, the implementation is built on top and is tightly coupled with the Robotic Operating System (ROS) [5] middleware, from which we exploit the inter-process communication structure and other features. Many design characteristic have been chosen in order to make *SkiROS* a flexible platform, integrating the state of the art in the task-level programming. These characteristics are result of numerous experiments within our previous EU project TAPAS. The software presented here is not just a prototype of an infrastructure to run skills, but the conclusion from all our previous skill-framework implementations. Relevant characteristics of the system are (but are not limited to):

- modularization into different abstraction layers
- plug-in structure
- presence of a knowledge integration framework
open to multi-robot cooperation.

The software framework is organized in the layers of device, primitive, skill and task. The number of layers, discussed in section III-A, is justified from our previous research and also from other work presented in section II, where the researcher seems to agree on a division similar to the one proposed in the paper.

One focus aspect of our system is the knowledge integration framework and the world model. The world model of our first software has been sufficient to describe environments with objects of the same shape and size. Nevertheless, the increasing complexity required to manipulate different kind of objects requires a more sophisticated management of the knowledge associated to objects. In this paper we will present how we addressed the problem, showing 1) the world model we adopted, based on our earlier experience with industrial logistic tasks and 2) the implementations developed.

The contribution claimed from this paper is the following:

- Definition of design aspects for a modular software architecture for task-level robot programming
- Definition of a world model for industrial use and comparison of three possible implementations: with an ad-hoc structure, using DSL tools or using the Web Ontology language (OWL) standard
- Methodology to ground the semantic world model to sensor data and vice-versa

II. RELATED WORK

Our software framework is organized in the layers of device, primitives, skills and task (more details are presented in sec III-A). In this section, whenever it is possible, we present the related work associated with respect to these layers. The research on tiered robotic architectures started during the '80 [7], and substantially substituted the previous sense-plan-act programming paradigm, even if current systems could be considered as hybrid. Since the '80, many solutions have been presented, but still no standard exists. Numerous middlewares have been proposed to support the implementation of robotics applications, such as (YARP [8], ROS [5], Player/Stage [9], OROCOS [10]). These middlewares hide the complexity of the operating system from the developer and highly simplify the development of a distributed software. Robotic software frameworks nowadays are usually build on top of these middlewares. There are programming paradigms, like Component-Based Software Engineering (BRICS [11]) and Model-Driven Engineering ([12], [13]), which rely on general-purpose notations such as UML to model domain-specific concerns. By using general-purpose and established notations, these approaches leverage existing knowledge from developers. Even though this approach theoretically help the user to develop and organize his software, the tools available nowadays to generate code from a model are quite often complex to set-up and use. SkiROS is designed following many concepts from Model-Driven Engineering, but unfortunately none of the available tools have found a practical use in our project.

SkiROS is build on top of the ROS middleware where there are some software frameworks going in the similar direction. Some examples are moveIT [14], ROSco [15] and Smach [16]. The former is focused on industrial manipulators and embeds many good mechanism, like collision-detection, arm motion planning and trajectory execution, etc., suitable for our primitive layer. ROSco and Smach work on an higher level of abstraction with respect to moveIT, comparable to our skill layer. These softwares are architectures for rapidly creating complex robot behaviors, under the form of Hierarchical Finite State Machine (HFSM). Both architectures allow only static composition of behaviors and cannot adapt those to new situations during execution.

At a level of primitives we can relate the work of Kröger et al. [3], [17] where they define a primitive as a triple composed of hybrid motion, a tool command and a stop condition. In their work they realized a multi-sensor control using an adaptive selection matrix to switch the control feedback. In [18] we get the definition of Task-Frame Formalism (TFF), which is a powerful approach to model a motion constraint and to specify the desired forces and motions compatible with this constraint. This approach is perfect for a primitive layer and was in fact embedded later in the iTaSc framework [19], [20], developed on top of the OROCOS middleware. In iTasc they define a constraint-based task specification methodology (iTaSC) to simplify the integration of multiple sensors in movement control. In more recent developments, the framework added a high-level Finite State Machine (FSM) system to coordinate several iTasc. This is a system similar to ROSco and Smach and comparable to our skill layer.

At a skill level, besides the FSMs presented before, we can cite the work of [21] that gives the definition of Object Action Complexes (OAC) and the manipulation primitive nets [3]. Other approaches try to integrate the programming aspect with learning capabilities, like the work of [22] or Riano et al. [23]. In the latter, they start from an initial set of predefined primitives, and generate artificial neural-networks to connect them.

At the task level the research is focused on the planning aspects. One of the earliest developments in autonomous robot, Shakey, uses symbolic representation of tasks and the STRIPS planner to reason about which actions may lead to accomplishing a goal [24]. Dozen of other algorithms have followed, based on the same ideas, in which they act on a set of actions that alter the current world state. Some of the most recent are [25]–[27].

The problem with modeling the world state, and maintaining this model is a common problem for all planning algorithms. All these algorithms require a description of the world state on which they can reason about. In this context, a first issue is to define the rules on which the world model instance is based on. A widely used approach to create this knowledge is to formalize it under the form of an ontology. The KnowRob framework [28], [29] is a prominent example of that way of thinking in the area of service robotics. It is based on SWI-Prolog and its Semantic Web library which serves for loading and accessing ontologies represented in the Web Ontology Language (OWL). A similar approach, which differs in the implementation of the semantic database, based in this case on openRDF, is the one presented in [30]–[32] for the Rosetta project. Both KnowRob and Rosetta use the semantic web standard, and exploit the ontologies not only to define the world model, but also to
formalize the robot actions, the devices and the robot self-description. Another solution is to define a world model using a Domain Specific Language (DSL) definition tool, like it has been done by [33]. However, as analyzed by Lortal et al. in [34] and Nilsson [35] and as we will also see further in this paper, it is possible to exploit an ontology, defined for example in OWL, to automatically generate code constraints which, to a certain extend, guide the programming exactly as a DSL is able to do, but with less effort for a developer not confident with DSL tools.

At very last, to underline how the integration plays nowadays a fundamental role in the research, we cite some examples of recent software frameworks used to develop knowledge-driven robotic systems in a wide range of applications [36]–[38].

III. OVERALL ARCHITECTURE

In this section we present an overview of the system architecture. As this paper is focused on the high-level aspect of the task-programming, after a description of the global layered structure, we describe in detail our skill’s model and we discuss the high-level layers, that have a direct interaction with the world model: the task and the skill layer.

A. Abstraction layers

A main design aspect to realize a flexible task-level programming software is to organize it on modular abstraction layers. This allow to 1) re-use the code 2) allow the robot to select the best module in order to optimize its behavior at run time depending on the context and available resources and 3) simplify the workspace at high-level. As illustrated in Figure 1, SkiROS is divided into 4 layers:

- **Device layer** - realize the hardware abstraction and present an interface to the devices.
- **Primitive layer** - embed and coordinate motion primitives, software blocks that realize a movement controlled with a multi-sensor feedback, and services, 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. This means that not all robot movements can be detected from the task layer. This should be expected as the world model is an advantageous simplification of the real-world.
- **Skill layer** - embed and coordinate skills, which are software blocks that operate a modification in the world state. These are analyzed more in detail in the following section.
- **Task layer** - the task layer can be used in two ways: from the end user or with a planner. In the first case the user concatenates manually the skills, in the latter the planner automatically finds a skill sequence for a given goal state on the world model.

The number of layers is justified from our previous research, where we identified these layers as the most suitable to organize the software. Compared with what was presented in [2], we added the device layer to increase the skills portability on different robots, we extended the definition of primitives to include services and we defined precisely the position of the world model.

Our system exploits two important features provided by the ROS middleware: 1) the communication system (topics, services and actions) and 2) the ‘pluginlib’ package. The ROS communication system realizes an abstract API (Application Programming Interface) between layers. The APIs are strictly decoupled from their implementation meaning that, if necessary, any layer implementation can be replaced with a more appropriate one without changing the rest of the system. On the other side, the ‘pluginlib’ package allows each layer to switch the implementation easily using plug-ins. The device layer imports device descriptions, the primitive layer imports primitives, the skill layer imports skills, the task layers is meant to import planners. All important functionality of the system are imported from the extern as plug-ins, that are compiled independently from the system and let the user free to program in a safe region.

The plug-ins are actually C++ classes, derived from an abstract base class. Several plug-ins can derive from the same abstract class. For example, any skill derives from the abstract class skill_base. This particular mechanism bind to use C++ code, at least to realize the plugins.

B. Skill model

The core idea of our robot skills is that they are fundamental software building blocks operating a modification on the world state [1]. They have a direct correspondence with the planners action set, and get as input objects that are present in the world model. Our set of skills was found by analyzing standard operating procedures from factories where the robot is supposed to work [39]. This has two important consequences: 1) the choices for skills is aligned with human intuition, as the robot actions are associated with human language, so that we can have skills called “pick”, “place” and “move” and 2) we can be sure that the space of possible robot goals in that particular factory hall is skill-complete, under the assumption that the
robot skills are able to reproduce the human skills. The model of a complete robot skill is shown in Fig. 2. 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 a variety of situations. Eventually, the skill will 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. However, the internal structure of a skill goes beyond the scope of this paper. 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" to place a "product" on or a "product" to pick up. Everything else is handled within the skill. Upon skill execution, the necessary calculations are made in order to successfully execute the skill. Running the skill with the same input parameter can result in a different skill behavior, that depends also on the specific scenario constraints. For example, a place skill takes one input parameter, the placing location, but the shape of the object in the gripper influence the skill's 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) and allows to re-plan a task without side-effects, if either pre-condition or post-condition are not satisfied. 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 veried that the desired location is now occupied by the object.

C. Task-Skill layer interaction

In our architecture, every layer can run one to several managers, that are registered as nodes on the ROS network. The task layer makes an exception to this rule, in fact, there can be only one task manager, as it is the overall controller of the various skill managers. The managers import the defined plug-ins and present the plug-ins set to other layers over the ROS network. As the managers can communicate over the Ethernet using the ROS communication system they can be executed on different machines, creating a flexible distributed network. In this way, we exploit the natural characteristic of ROS to create a distributed system, but 1) we create a meaningful structure to organize the nodes, and 2) we reduce the impact of ROS network communications, because a big part of the communication stream remains between the threaded plug-ins inside managers’ processes. The final structure extend like a tree starting from the task manager, the main ‘brain’, going to skill managers, that are associated to every robot, and ends up in the primitives and device managers. These low-level layers are associated to every sub-system of a single robot, in the very common cases where the robot is not a single system, but a system of systems. However, the details on lower layers go beyond the scope of this paper.

By executing a skill manager for every robot in the working domain, it is possible to realize a multi-agent planning system. In fig. 3 it is presented an example of how this is practically realized. The task manager is connected in read-only access to the world model, where it monitors the state of the world and the robots. The task manager can acquire the description of the available skills from the skill’s managers. To each skill is associated a description of pre-conditions and prediction. Using this information, the planner can use general planning techniques to generate a sequence of skills to reach a goal state. Consequently, it contacts the skill managers over the ROS network and schedule the skills execution. The skills check the environment with real sensing before executing the action, and in case of error, return to the task manager that can then re-plan a skills sequence.
IV. 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. Based on this information, the planner can apply logic rules systematically to discover a correct sequence of actions, in order to reach a goal state from an initial state. Constructing a knowledge base for robot application is a challenging task since robotic applications have very specific demands. Generally, knowledge is hard-coded in the task planner in a suitable way. This solution is a problem when there is necessity to modify the knowledge and propagate it over different software modules. We analyze in this section what knowledge (the meta-model) is required to define a world model, we present solutions to ground this model to reality and finally we present possible implementations. The world model structure considered in this paper is a graph asymmetric and not-weighted.

A. Spatial knowledge meta-model

To build a semantic world model the knowledge required to be defined is about:

- **Data** - define which kind of properties can be related to elements
- **Elements** - define the set of elements expected to find using a taxonomy (a hierarchical tree where is expressed the notion about types and subtypes)
- **Relations** - define a set of qualitative spatial relations between elements
- **Constraints** - define constraints over the world relations and element properties. This should include also tolerances on the constraints.

This meta-model must be instantiated to generate a graph where the nodes are the physical elements and the edges are spatial relations. The spatial relations are qualitative, in the sense they are not precise metric positions, on the contrary they give a semantic specification of the position in accordance to human knowledge (e.g. onTop, under). This is usually sufficient information for a planning algorithm. It is important nevertheless to create a mapping between precise metric position and qualitative definitions, and this leads to the argument of the following section.

B. Grounding

It is common sense in robotic to divide the knowledge in three main domains: continuous, discrete and semantic. Continuous data is extracted directly from sensors. Discrete data are relevant features that are computed from the continuous data and are sufficient to describe a certain aspect of an object. Semantic data is abstract data, that describe qualitatively a certain aspect of an object. The semantic world model, as the name suggest, manage semantic knowledge. Nevertheless the data related to elements lies in a gray area between the discrete and semantic domain, and there is need of a methodology to link the data back and forward. Task and skill layer share the same world model, but the definition of objects in the two domains are slightly different: the planner at task-level refers to objects with semantic information, on the contrary the skills at skill-level require concrete, discrete information about the objects.

In our world model we partially manage to bridge this gap using a specific design solution where we split the data from the methods that reason on it. The data is stored in a class called `Element`, characterized by: type, *id*, *label* and a flexible *list of attributes*. The methods are contained in the *discrete reasoners*. We call these reasoners discrete because we want to distinguish from the semantic reasoners often used in advanced inference systems. The role of *discrete reasoners* is two fold: 1) convert the sensor data to discrete data and 2) offer abstract API to the semantic level in order to reason about discrete data. This is summarized in the schema in fig. 4. Once an element is filled up with some data from a reasoner it is marked and associated to that particular reasoner, in order to inform other skills and task-manager which reasoners can be used on that particular element. To give an example: at a semantic level, we can define as constraint that an element *e1* can be *inside* an element *e2* only when *e1.size < e2.size*. This particular constraint cannot be evaluated only by the semantic graph, but it need to compare an internal property (the size) stored in the *list of attributes* of the elements. In this case, the task-layer can contact the *discrete reasoner* associated to size, and request to compare the two elements. The presence of *discrete reasoners* is important to keep in one place all the methods regarding some aspect of an element, without affecting the elements structure. This allow skills and the task-layer to share paradigms of reasoning. Moreover, we can change radically the way the implementation about, e.g., the size reasoning, without affecting the rest of the system.

C. Implementation

It is not our aim to go too much in detail of implementation aspects, which are of no interest for a reader. In this section, it is our wish to analyze three possible solutions and compare
their pros and cons. All the implementations have in common
the advantage to not put any constraint on the internal imple-
mentation of skills, keeping decoupled the abstract reasoning
domain from the concrete actions.

1) Ad-hoc: Indeed, an hard-coded ad-hoc solution is the
most quick and simple to realize. In many cases it can result
the most appropriate solution, when 1) semantic domain is well
defined and 2) only the programmer have to apply modifications
to the model.

The data that is necessary to encode, derived from the meta-
model plus the mechanism for grounding semantic knowledge, are:

- a data domain set
- an elements taxonomy definition
- a relations set
- a constraints handler, which should keep separated 1)
  constraints tracking from 2) constraints maintenance
  policy
- a reasoners handler
- a database to store the world model instance, plus a
database to store the world knowledge

Most of these modules should be implemented to be easily
editable. In particular, the reasoners and the data domain can be
subject to many modification even when the semantic domain is
well known. In fact, they are influenced by the low-level
implementation of skills and primitives. Once implemented,
this solution have a very limited flexibility of the semantic
model.

2) DSL tools: This implementation has not been practically
realized from us, but indeed DSLs tools such as Xtext [40] can
be a powerful solution to define a world model. In a nutshell,
with DSL specification tools it is possible to specify some
primitives, relations, constraints and tolerances, that build-up a
language specific to describe a certain domain. This information
is similar to what presented in section IV-A. The software
development get constrained in the defined domain, making 1)
the code more readable and compact and 2) limiting the syntax
events. This solution has a clear advantage in the way it guides
the programming. The main drawback is that the DSL must be
translated into code in a process that must be set-up. Moreover,
it is complicate to modify the semantic model from users who
didn’t implement the DSL.

3) Web Ontology Language: The OWL standard is a most
used standard to define ontologies. To introduce the term
ontology we use the definition of Wikpedia: "An ontology is a
formal definition of the types, properties, and interrelationships
of the entities that exist for a particular domain of discourse".
Looking again at the meta-model in section IV-A, it is possible
to see that it is actually an ontology. In fact we need to
define types (elements), properties (data) and interrelationships
(relations). Consequently OWL can be a solution to define it.
In this paper we refer in particular on a subfamily of OWL, the
OWL-DL. This family is grounded on Description logics (DLs):
a set of logics that are decidable fragments of first-order logic
with interesting computational properties. OWL-DL combines
a syntax for describing and exchanging ontologies, and formal
semantics that gives them meaning.

An ontology defined in a OWL file cannot be used directly by
software, but it must be decomposed into Resource Description
Framework (RDF) [41] triples, that are triples composed of
a subject, a predicate and an object, representing the atomic
block of knowledge of an ontology. In order to use OWL into
a software it is needed to: 1) create the OWL file using an
editor 2) use a tool to parse the file into an RDF-triple store
and manages queries 3) develop an interpreter to translates
RDF nodes back and forward to concrete code structures. With
respect to the hard-coded implementation, all data is defined
in a OWL file.

The advantages to use the OWL standard are that:

- Graphic editors are available (ex. Protégé [42] )
- It’s sharable and auto-documenting
- It ensure maintainability
- Can be used to express other type of knowledge, besides spatial knowledge
- Its formalism allows inference of new concepts

In a final analysis, between the presented solutions the OWL
standard result as the most suitable to bridge the gap from code
experts to domain experts. Using a graphical editor, a domain
expert can modify the world knowledge 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. Note also that OWL files are not meant
to store big clusters of binary data.

D. An example of model for the factory domain

![Fig. 5. An instance of the world model.](image-url)

In Fig. 5 is presented an example of a world model instance.
In this example, the instance is generated from the following
set of knowledge:

- Data: [ size, position, orientation, SpatialReasoner-Name ]
- Elements: [ scene, location, station, surface, container, box, kitting box, compartment, product,
Alternator, tube, agent, robot, gripper, arm, navigation platform, camera

- Relations: [contain, hasA]

- Constraints: [\(\forall e \in \text{Elements with } e \neq \text{scene}, \exists \text{at least 1 triple } (x, y, z) \text{ where } z = e, y \in \text{Relations and } x \neq e]\]

We can see we have only two relation: contain and hasA and one constraint, that requires all elements, except the scene root, to be passively related to at least another element. This example is quite basic, nevertheless allow to reason about objects localization and can straightforward represent a key concept like full/empty or a product displacement from a surface to a kitting-box, this by simply changing one relation.

V. CONCLUSION AND FUTURE WORK

In this paper we presented the architecture of a complete software framework to program autonomous mobile manipulators in an industrial environment, result of the experience acquired in our earlier research. The architecture allows to plan and control different skill-equipped robots in the same scene, where the robots and the task manager share knowledge by mean of a semantic world model. We analyzed the kind of knowledge that must be defined to generate a useful world model for the planner and a methodology to ground the model to concrete data for the skill execution. We also discussed three possible solutions to define this knowledge, where we concluded that the OWL standard is the most suitable for a number of reasons, like the maintainability and possibility of using inference techniques.

Compared to other architectures present in the literature, our solution offers a methodology to ground the abstract reasoning to the robot skills, by mean of discrete reasoners. Moreover, it provides a meaningful structure to organize the ROS nodes. This structure is designed to let the system to distribute over different machines meanwhile minimizing the network communication. At the time of writing, the world model, the skill layer and the task layer have been implemented and we are now testing the system to plan and execute a sequence of skills.

In a future work we plan to focus on lower layers. First, increase the knowledge contained in the world model with the robot self-description and develop the device layer in order to decouple the upper layers from the hardware. We then plan to focus on the primitive layer’s design. As the system is going to be used in industrial environment, the characteristics we aim to achieve are: system simplicity (to decrease error probability), robustness, reliability and predictability.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Unions seventh framework program (FP7/2007-2013) under grant agreement n° 610917, STAMINA.

REFERENCES


