ICT Call 7
ROBOHOW.COG
FP7-ICT-288533

Deliverable D7.7:

Report on benchmarking set-ups with preliminary acquisitions and representations

January 31st, 2015

Project acronym:	ROBOHOW.COG
Project full title:	Web-enabled and Experience-based Cognitive Robots that Learn Complex Everyday Manipulation Tasks
Work Package:	WP 7
Document number:	D7.7
Document title:	Report on benchmarking set-ups with preliminary acquisitions and representations
Version:	1.0
Delivery date:	January 31st, 2015
Nature:	Report
Dissemination level:	Public
Authors:	Pierre Gergondet (CNRS) Kevin Chappellet (CNRS) Gianni Borghesan (KUL) Nadia Figueroa (EPFL)

The research leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement n^{o}288533 ROBOHOW.COG.

Contents

Cited Works

[Gergondet et al., 2014] Gergondet, P., Petit, D., Meilland, M., Kheddar, A., Comport, A. I., and Cherubini, A. (2014). Combining 3D SLAM and Visual Tracking to Reach and Retrieve Objects in Daily-Life Indoor Environments. In *International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*.

[Pais et al., 2014] Pais, A. L., Umezawa, K., Nakamura, Y., and Billard, A. (2014). Task parametrization using continuous constraints extracted from human demonstrations. *Submitted*.

Chapter 1
Introduction

This document summarizes the current progress on the benchmark set-ups. The role of these benchmarking examples is to illustrate the advances and the results achieved throughout the project on different robotic platforms.
At this point of the project, the scenarios of these benchmark set-ups have been decided and realized on specific platforms. An integration effort is currently being made to perform those scenarios on all the robotic platforms at two levels:

1. Task-oriented controllers, namely the expressiongraph-based Task Controller, (eTC, that supersedes the former framework iTaSC, see D3.3) and the Stack-of-Tasks (SoT) controllers, are being integrated onto the low-level controllers of the robotic platforms. This has already been achieved on most platforms.

2. Allowing those controllers to receive their tasks from the high-level semantic queries. This has been a continuous effort of the project since its beginning and thus only the latest development of the partners have yet to be integrated.

In this document, we report the progress that has been made on each of the benchmark scenario. In each section, we describe the scenario that has been adopted, the demonstrations that have been achieved by the partners relatively to those scenarios and finally the perspectives for the integration on the other robotic platforms.

Chapter 2

Benchmark scenario 1: Everyday manipulation for meal preparation

2.1 Introduction

The first benchmark is an autonomous mobile manipulation platform that is to perform simple meal preparation tasks based on instructions and observed activity demonstrations starting with tasks that involve mainly pick and place capabilities such as setting the table. The range of tasks will be carefully selected with respect to the perception and manipulation capabilities of the robot. The robot does not have to do all manipulation tasks by itself but it has to have enough knowledge about its capabilities to know what it cannot and ask for help if necessary.

2.2 Scenario description

In this scenario, the robot should prepare a simple meal, specifically, it should prepare a pizza. Therefore, it first has to roll out a piece of dough. At the end of the preparation the dough should have a circular form. To accomplish its objective the robot should be able to handle a rolling pin and to visualize the dough form in order to monitor the task progress.
During this scenario, the robot will perform a rolling sequence. This sequence is composed of 3 actions. Each action is a decomposition of the movements that a human performs to roll out a dough. The first is to reach the dough with the rolling pin, the second is to roll out the dough and the third is to go back to the initial position. These actions are stored into Knowrob knowledge. The robot will repeat this sequence until the dough has a circular form.
The dough form is controlled by the vision module. Between each rolling sequence, the robot acquires data from its sensor to determine the next movement. The actual dough form is obtained through visual inspection and the initial position of the next roll out action is calculated, this includes the rotation of the rolling pin if necessary.

2.3 Achievements

The initial steps necessary to achieve this demonstration have been undertaken on the Boxy robot [Pais et al., 2014]. This work is enlightened in the deliverables D5.3 in the part concerning

the Task 5.3 Learning adaptive stiffness control that has the desired effects. This part is the Chapter 2 named Constraints extraction for sequences of tasks.

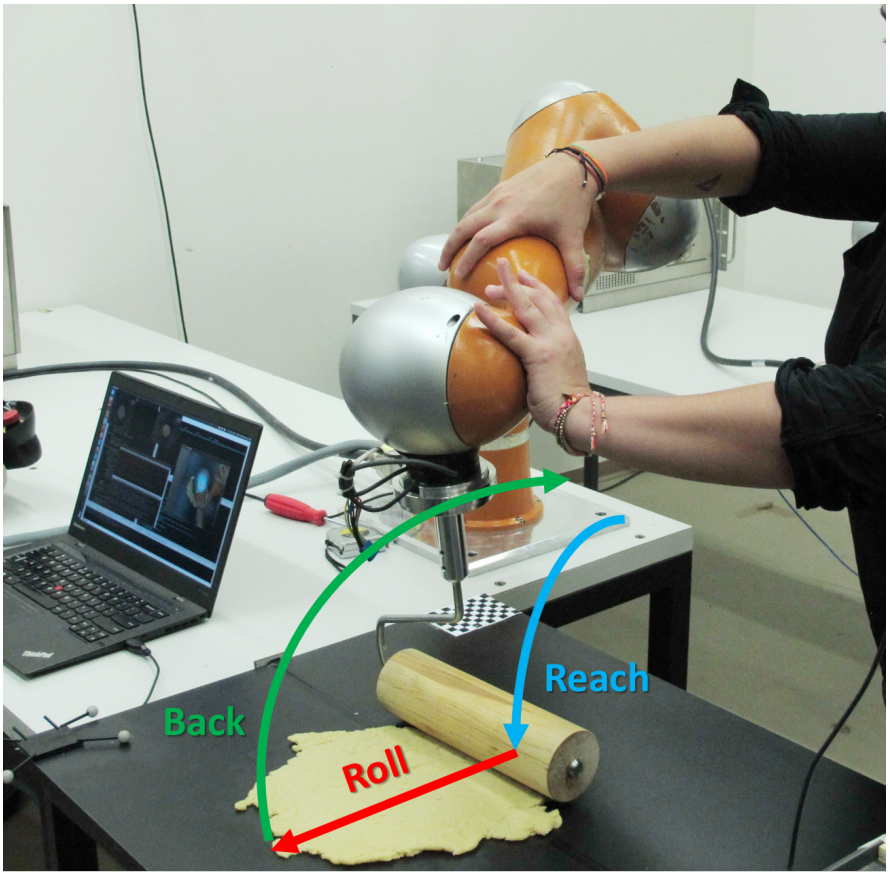


Figure 2.1: Illustration of the rolling sequence. The 3 actions, in order, are reach the dough, roll it and go back to the initial position.

2.4 Perspectives

The current software architecture that is used to control the boxy robot in this demonstration is shown in Figure 2.2.
The current focus of integration is to replace the "Joint Space Transformer" entity, highlighted in red, by the Stack-of-Tasks controller. Indeed, this entity role is to transform the desired end effector position, desired stiffness and desired contact force given by the high-level "Motion Controller" into a velocity command for the Boxy robot. This could and should be achieved by the SoT controller.
Once this integration will be achieved, we will be able to perform this scenario on all the robotic platforms used in the project since the Stack-of-Tasks controller is now able to control them all. This scenario will thus properly achieve its role of benchmarking.

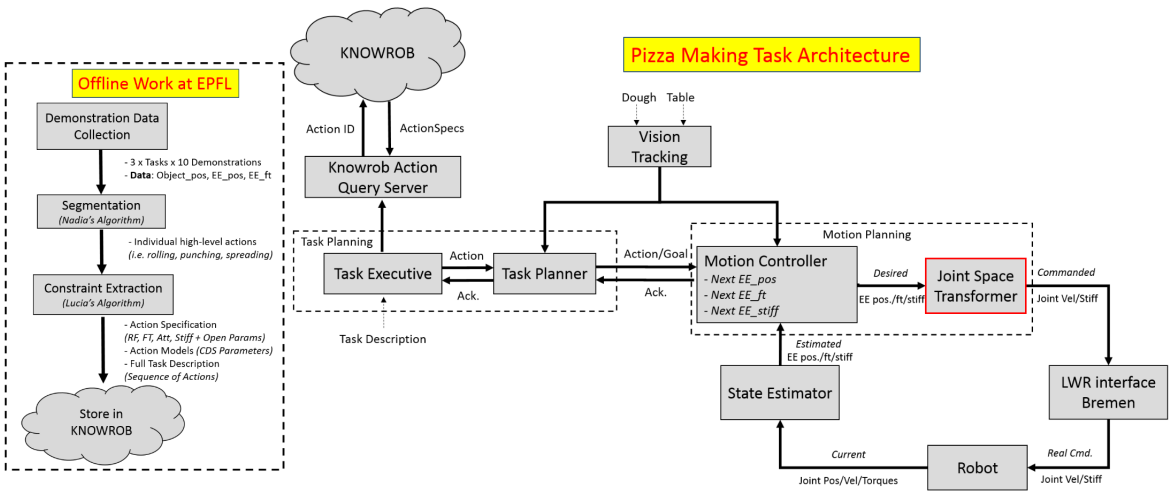


Figure 2.2: Current architecture of the dough demonstration involving the boxy robot.

Chapter 3

Benchmark scenario 2: Humanoid robot providing daily office services

3.1 Introduction

The second benchmark involves a humanoid robot and emphasizes on knowledge acquisition and human monitoring to highlight improvements in whole body manipulation (e.g. coordination of locomotion and manipulation). The anthropomorphic properties of a humanoid robot may benefit from human monitoring to improve in achieving the assigned tasks and also its own behavior (WP2-4), namely those that are achieved in close interaction/cooperation with a human. By M36, we expect to have a humanoid robot (HRP series: HRP-2 or HRP-4 and Romeo) already embedded with full capabilities of web-representation and exploitation, such as mapping complete tasks plans (WP1) from the web. In addition, the humanoid robot will be embedded with capabilities of learning and tasks extraction from human monitoring (WP5) and with tuning mechanisms that allow the robot to progressively adjusted its behavior from human monitoring (WP2).

3.2 Scenario description

In this scenario the humanoid robot goal is to operate a printer. This is decomposed in two phase. The first is the localisation and navigation towards the printer in an office environment. The second is the actual operation of the printer which has been reached.

During the localisation phase, the robot searches the office until it locates the printer. Once the robot detected the printer, it navigates towards the printer until it is close enough to operate it and then stops. The localisation of the printer requires the robot to know the printer 3D model and texture information. This data should be generated before the experiment and stored somewhere the robot can retrieve it such as the KnowRob knowledge base.

In the operation phase, the robot should be able to perform three different tasks: (i) open the lower part of the printer, (ii) open the upper part of the printer, and (iii) push the power button. The location of these elements relatively to the printer should be known in order to perform the tasks. Based on this information, the robot should be able to properly detect the position of the different elements and perform the chosen task based on this information.

3.3 Achievements

The execution of the task on the target object requires the robot to first locate and then navigate towards this object. This was achieved by integrating well known and efficient recognition and navigation techniques from computer vision applied to robotics [Gergondet et al., 2014].

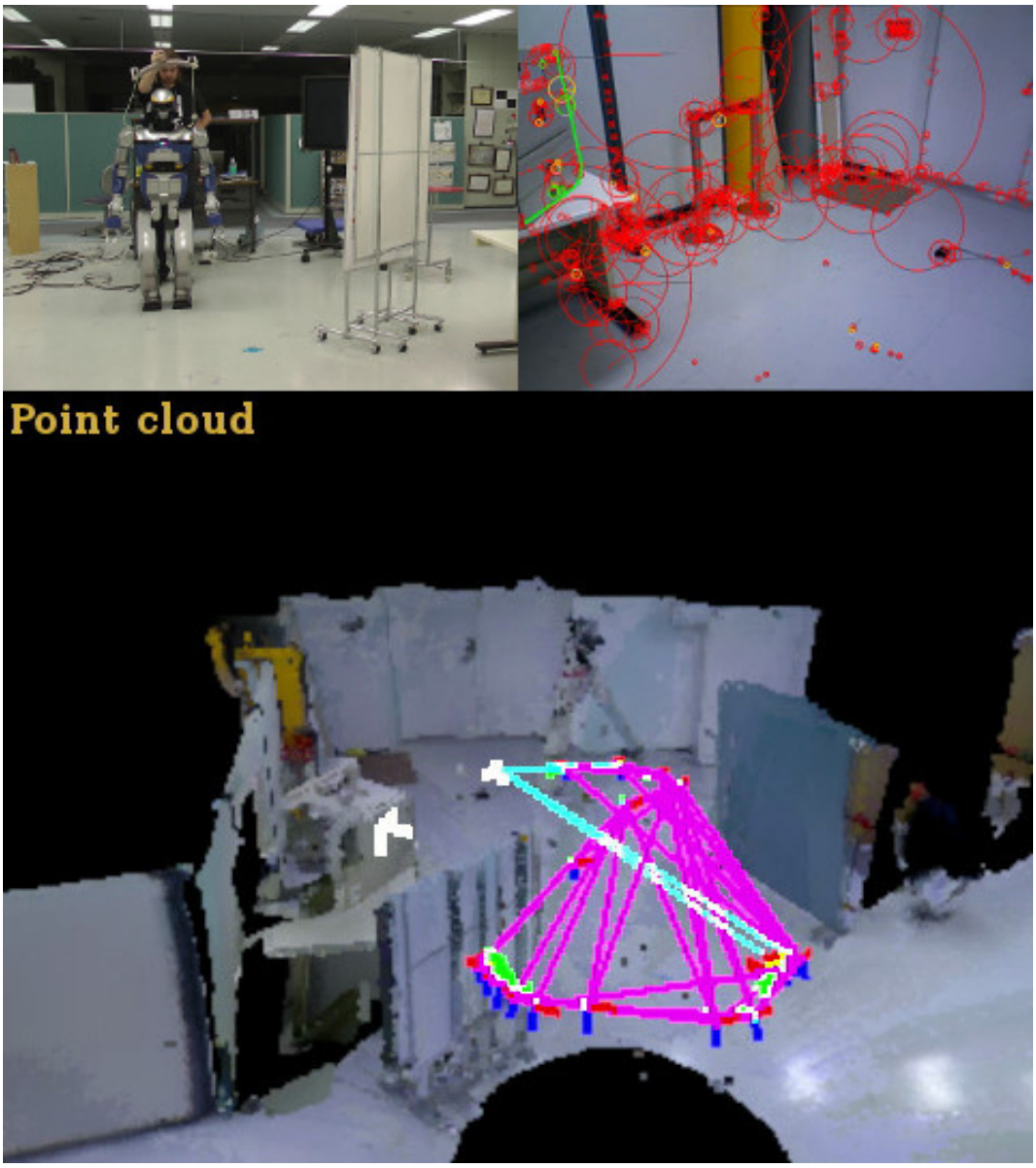


Figure 3.1: In the top-left corner, the robot is seen in its environment looking for the object of interested. In the top-right corner, the recognition software, BLORT, found the object of interest. Finally, in the bottom, a view of the environment map generated by the D6DSLAM software and the robot's localization within this map as the object has been reached. The nodes of the purple graph represents key-frames while the blue line indicates the current dislocation of the robot compared to the key-frame that is used as a reference for localization.

In the second phase of the benchmark scenario, the robot should operate the printer that it just reached. This was achieved on the HRP-4 humanoid robot as seen in Figure 3.2. This work was also done on the PR2 robot but only in simulation. More technical details can be found in the corresponding publications, that have been added in the Appendix. Moreover this simulation refers to the work that was done for the deliverable D3.3.

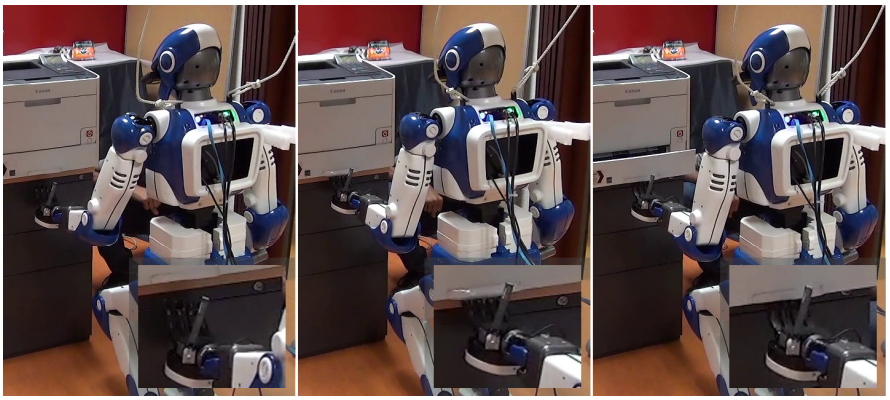


Figure 3.2: An extract of the task where the HRP-4 opens the printer drawer. In this case several constraints are exploited to ensure stable contacts on feet and hands, to shape the hand so that the handle can be grasped, and to maintain the equilibrium while exerting pulling forces.

3.4 Perspectives

The current integration effort is twofold:

1. Integrate the two phases of the scenario into a single experiment. That is currently being done on HRP-4.

2. Try out this demonstration on the other humanoid robot of the project (HRP-2 and Romeo). As those are already controlled by the Stack-of-Tasks, this should only require actual trial and no further software integration, thus achieving the benchmark role of this demonstration.

3. The experiment described in the journal paper in appendix will be run on the PR2.

Finally, as progress is being made in parallel in other work packages, this scenario is well suited to incorporate additional complexity regarding the tasks that are being achieved in the "office" environment given to the robot.

List of articles

Noname manuscript No.
(will be inserted by the editor)

Introducing geometric constraint expressions into robot constrained motion specification and control

Gianni Borghesan · Enea Scioni · Abderrahmane Kheddar · Herman Bruyninckx

the date of receipt and acceptance should be inserted later

Abstract The problem of robotic task definition and execution was pioneered by Mason (Mason (1981)), which defined *setpoint constraints* where the position, velocity, and/or forces are expressed in one particular *task frame* for a 6-DOF robot. Later extensions generalized this approach to *i) multiple* frames, *ii) redundant* robots, *iii)* constraints in *other sensor spaces* such as cameras, and *iv)* tracking *trajectory* constraints. Our work extends tasks definition to *i) implicit expressions* of constraints between geometric entities (orthogonality, parallelism, distance, angle...) in place of *explicit setpoint constraints*, *ii)* a systematic *composition* of constraints, *iii) runtime monitoring* of all constraints (that allows for *runtime sequencing* of constraint sets via, for example, a *Finite State Machine* (FSM)), and *iv) formal* task descriptions, that can be used by *symbolic reasoners* to plan and analyses tasks. This means that tasks are seen as ordered groups of constraints to be achieved by the robot's motion controller, possibly with different set of geometric expressions to measure outputs which are not controlled, but are relevant to assess the task evolution. Those monitored expressions may result in events that trigger associated FSM to make a decision to eventually switch to another ordered group of constraints to execute and monitor.

For all these task specifications, formal language definitions are introduced, with the aim of providing clear abstractions from the concrete capabilities (hardware as well as control behaviour) of the executing robot platforms. The influence of each specific platform can be added, also in the form of a set of (not task-related) constraints (such as kinematic and actuator limits), to satisfy by the robot's task controller.

When both task and platform constraints are expressed in formal languages with grounded semantics, it will become possible to reason about particular robot-task combinations and evaluate the feasibility of a particular task before trying to execute it.

Keywords Task-based control · Constraint-Based control · Domain Specific Language

1 Introduction

The robotics research community is, since a long time already, trying to solve the technical challenge to make robots execute tasks by only specifying *what* one wants a robot to do, instead of having to code manually *how* it has to do it.

One particular approach that is under active development uses the paradigm of *constrained optimization system*: the instantaneous motion of the robot's joints is computed as the solution to a constrained optimization problem in which:

- the *objective function* is a combination (weighted and/or prioritized) of cost functions (i.e., moving the robot requires effort) and/or utility functions (i.e., its motion creates progress in the execution of the task).
- the *constraints* are functions of a subset of the task and robot "state variables", that must be satisfied during the whole task execution. In general, such satisfaction requirements involve *equality* as well as *inequality* constraints. The functions can be *implicit*

The authors gratefully acknowledge the support from the European Commission's FP7 project "RoboHow" (FP7-ICT-288533), and KU Leuven's *Geconcerteerde Onderzoeks-Actie* "Global real-time optimal control of autonomous robots and mechatronic systems".

Gianni Borghesan, Enea Scioni and Herman Bruyninckx
KU Leuven, Department of Mechanical Engineering, Heverlee, Belgium. E-mail: name.surname@mech.kuleuven.be

Enea. Scioni
with Engineering Department (ENDIF), University of Ferrara, Ferrara, Italy.

Abderrahmane Kheddar
CNRS-AIST Joint Robotics Laboratory (JRL), UMI3218/CRT, Tsukuba, Japan, and CNRS-Université Montpellier 2, LIRMM, Interactive Digital Human group, Montpellier, France

Herman Bruyninckx
Department of Mechanical Engineering, Eindhoven University of Technology, the Netherlands.

Fig. 1 An extract of the task where the HRP-4 open the printer drawer. In this case several constraints are exploited to ensure stable contacts on feet and hands, to shape the hand so that the handle can be grasped, and to maintain the equilibrium while exerting pulling forces.

(*e.g.*, $g(y,a,b,c) \leq 0$) or *explicit* (*e.g.* $y = f(a,b,c) = 0$).
- the *switching* between different sets of objective functions and constraints must happen on the basis of monitoring the extent to which some functions (not necessary constraint functions) assume given values during the task execution.

Partial solutions exist already in this approach, *e.g.*, *Stack of Tasks* ("SoT", Mansard et al (2009)), *instantaneous Task Specification and Control* ("iTaSC", De Schutter et al (2007)), *Whole Body Control* ("WBC", Sentis and Khatib (2006)), or a revised versions of the *Task Frame Formalism*, ("TFF", Bruyninckx and De Schutter (1996); Kröger et al (2011)).

With these formalisms, a robot can be driven to achieve complex tasks, like the one already described in literature, *e.g.* climbing a ladder (Vaillant et al (2014)), manipulate objects, *etc.*). As example, we bring the task depicted in Fig. 1, where the the HRP-4 opens a printer paper tray (will consider a similar task opening a drawer as exemplification in Sec. 7). This task, as well as the above mentioned, requires the definition of many constraints in different spaces and in different reference frames. However, while the control aspects of practically all these frameworks are mature, it is questionable whether the same can be asserted regarding the way tasks are designed and formalized.

Summing up, the state of the practice is that:

- current approaches do not follow the rule of the *separation of concerns*: these software frameworks do not separate the configuration of the hardware platform, communication infrastructure, *etc.*, from the task definition. Therefore, users are confronted with non-trivial learning curves, as they need to master all these aspects while they should focus only on the task specification.
- the overlaps between the approaches have not yet been fully investigated, and refactored integrations are still partial, and in a premature state, is therefore difficult to execute the same task with different combinations of robotic platforms and control approach, and compare the outcome.
- a formal semantics is still missing, reducing the opportunities to use reasoning and task planning at the symbolic level to automatically generate tasks.

Looking at the above consideration about the state-of-art, we reach the belief that a fundamental step to ease the employment of constraint-based systems, and in general, task-based programming, is to separate the task description from its implementation.

In the next sections we will propose an approach to achieve this result. In fact, we considered the characteristics of the above cited constraint-based approach, and we considered, based on our programming experiences, which are the more common constructs that we implicitly or explicitly refer in the task design process. We gathered these informations and coded in a language definition (sections 3 to 6), that can be interpreted on different platforms. Lastly, in Sec. 7, a task example is provided.

2 Tasks as sets of constraints: fundamentals

Our goal is to describe a task constraint with a *a minimal set* (of properties, relations, equations...), in such a way the *specification* is abstracted (as much as possible) by the *implementation* of how *to solve* for the constraint during the *execution* of the task. We start considering the simplest task description formalism, the *Task Frame Formalism*, Mason (1981); Bruyninckx and De Schutter (1996), where each constraint is defined employing a minimal set of specifications, namely: *i)* the controlled and task frames, *ii)* selection matrix $\mathbb{S}$ that expresses which, among the six components of the Cartesian space, are controlled in position and which are controlled in force, and *iii)* the reference values. Thus, the description of a task (in the Task Frame Formalism) consists of at most six non-conflicting constraints, that are applied in the controlled frame (that normally corresponds to the robot end effector) and expressed in a frame where position and orientation are chosen to represents elements of interest for the task itself. Employing the Task Frame Formalism approach, Bruyninckx and De Schutter, Bruyninckx and De Schutter (1996), already proposed a generic way of defining a tasks as a set of constraints (an example is given in Listing 1).

Listing 1 Example of a guarded-motion task definition from Bruyninckx and De Schutter (1996)

```
move compliantly {
  with task frame directions
  xt: velocity 0 mm/sec
  yt: velocity 0 mm/sec
  zt: velocity v_des mm/sec
  axt: velocity 0 rad/sec
  ayt: velocity 0 rad/sec
  azt: velocity 0 rad/sec
} until zt force <- f_max N
```

Other works followed this line, either focusing on the specification (*e.g.* Kröger et al (2004)), or on the flex-

keyword:	*meaning:*
name	mnemonic name,
postcheck	function that checks data consistency,
type	class described,
sealed	specify if 'dict', 'vect', or both can have additional entries,
dict	dictionary (collection) of types, identified by a name,
vect	vector of a given type,
optional	list of optional entries in the dictionary.

Table 1 DSL keywords used to define tasks. Refer to Klotzbücher (2013) for a more in-depth explanation.

ibility of constraint definition (*e.g.* De Schutter et al (2007)).

The Task Frame Formalism separates task description from its implementation: indeed, Listing 1 fully defines a task, with no explicit dependancies to *robot information*, such as: *i)* the kinematics (redundancy as well as mechanical and actuator constraints), *ii)* the control laws employed in the motion or force control, and *iii)* the perception capabilities (estimation, observation...). Similar considerations can be applied to more recent frameworks (*e.g.* SoT, Mansard et al (2009), and WBF, Sentis and Khatib (2006)), that, while posing the emphasis on different aspects, maintain the same approach in task definition.

Our extension to this task description approach has the following characteristics:

- larger freedom in the choice of the constrained variables, that, instead of being related to the Cartesian space representation (not always suited to easily describe a generic relationship between two bodies), will be represented as (one of a enumerable set of functions between) *geometric primitives*. In this regard, Sec. 3 focuses on the description of the *geometric expressions*, and on the geometric primitives used by these geometric expressions,
- substitution of the type of *control* (*e.g.*, force and position) with the type of *behaviour* that we want to achieve; which includes the description of the type of control *and* the type of constraint, Sec. 4,
- rules *to compose* constraints into tasks, Sec. 6.1, while specifying how to resolve conflicts in over-constrained robots, and
- specification of *measurement expressions* that are employed *to monitor* the task execution state, and to trigger changes in the robot behaviour.

In order to uniquely define the expressions, we refer to snippets of code, that implement a *Domain Specific Language* for tasks. The formal definition of primitives, expressions, and all the other elements are given by means of code listings, that implements the domain specific language of tasks. Such code is based on uMF, a modeling framework for the Lua language. This DSL tool employs few keywords, that are described in table 1.

3 From geometric entities to expressions

This Section introduces the concepts of geometric entities, geometric primitives and geometric expressions, which will be employed in the rest of the discussion.

The definition of an expression is fairly simple: it can be either a *joint expression* or a *geometric expression*. **Expressions** (both geometric and joints) **define spaces**: these spaces can be used either for computing values relative to (generalised) positions, forces, or velocities (Sec. 5), that in turn can be used to enforce constraints, or to monitor such quantities (Sec. 6.2). In case of position, the value of the expression itself represents the desired value of a forward kinematics. When velocity or force are sought, the partial derivative of the expression (with respect to joint angles) represents the differential map that relates joint velocities to generalised velocities, or joint torques to generalised forces.

Joint expressions (*e.g.* limits on joint positions) typically represent *robot* intrinsic constraints, while most of the *task-centric* constraints come in via the geometric expressions; their constitutive elements –geometric expressions and primitives– are introduced hereafter.

3.1 Geometric entities and primitives

The most elementary types of *entities* are the *point* and the *versor*; both of them are represented by a triple (x, y, z); additionally, the versor must comply with unitary norm constraint.

By combination of these entities, we can have the other two common geometric entities, that are the line and the plane. Both are combinations of a point and a versor for a total of five free parameters for each description. The formal description of entities are given in Listing 2.

Listing 2 Geometric entities definition

```
point_spec = umf.ObjectSpec{
  name='point',
  type=Point,
  sealed='both',
  dict={x=umf.NumberSpec{},y=umf.NumberSpec{},z=umf.NumberSpec{}}
}
versor_spec = umf.ObjectSpec{
  name='versor',
  postcheck=versor_unitary_mod_check,
  type=Versor,
  sealed='both',
  dict={x=umf.NumberSpec{},y=umf.NumberSpec{},z=umf.NumberSpec{}}
}
line_spec = umf.ObjectSpec{
  name='line',
  type=Line,
  sealed='both',
  dict={ origin=point_spec,direction=versor_spec},
}
plane_spec = umf.ObjectSpec{
  name='plane',
  type=Plane,
```

Geometric primitive:	*symbol*:	*entity composed by*:
point expr. in $\{w\}$	$\boldsymbol{p}_{\{w\}}$	scalars x, y, z
versor expr. in $\{w\}$	$\widehat{\boldsymbol{n}}_{\{w\}}$	scalars x, y, z s.t. $\|\cdot\| = 1$
line expr. in $\{w\}$	$\widetilde{\boldsymbol{n}}_{\{w\}}$	point *origin*, versor *direction*
plane expr. in $\{w\}$	$\mathfrak{P}_{\{w\}}$	point *origin*, versor *normal*

Table 2 Summary of geometric primitives.

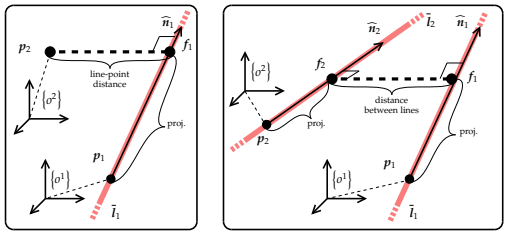
	point	line
point	point-point distance	
line	line-point distance projection of point on line	distance btw lines projection o1-f1 projection o2-f2
plane	point-plane distance	

(a) Geometric expressions on distances.

	versor	plane
versor	angle btw versors	
plane	incident angle	angle btw planes

(b) Geometric expressions on angles.

Table 3 Summary of geometric expressions between primitives.



(a) Line 1 is expressed in $\{o^1\}$, Point 2 in $\{o^2\}$.

(b) Lines 1 and 2 are expressed in $\{o^1\}$ and $\{o^2\}$, respectively.

Fig. 2 Graphical representations of the 5 possible relations between a point and line (Fig. 2a) and two lines (Fig. 2b).

```
    sealed='both',
    dict={origin=point_spec, normal=↩
        versor_spec},
}
```

Entities needs to be grounded on a frame in which they are represented. As a consequence we introduce the geometric primitive, that associates a geometric entity with an *object frame*. The four possible geometric primitives are reported in table 2, along with the associated mathematical symbol.

3.2 Geometric expressions

While the list of primitives provided in table 2 is certainly not exhaustive, it suffices to build-up most of the scalar expressions that describe positioning between (the feature of) two objects. Table 3 summarizes the distances and angles that can be measured (and thus imposed) between the entities of two primitives.

Since most of the entries in table 3 are self-explaining, we focus only on the definitions given by of line-point and line-line distances, providing a clearer explanation for each of the expressions.

The line-point entry: It has two expressions, *i)* the *line-point distance*, that is the distance between the point $\boldsymbol{p}_2$ and the point of shortest distance $\boldsymbol{f}_1$, and *ii)* the *projection of point on line*, that is is the (signed) distance between the points $\boldsymbol{p}_1$ and $\boldsymbol{f}_1$.

line-line entry: Following the same line, it is possible to define three expression distances in the line-line entry: *i)* the *distance from lines*, distance d between the two lines, *ii)* the signed distance between the point $\boldsymbol{p}_1$ and the minimum distance point $\boldsymbol{f}_1$, and *iii)* the signed distance between the origin of the line $\boldsymbol{p}_1$ and the minimum distance point $\boldsymbol{f}_2$, as shown in Fig. 2b.

3.3 Expressions in the joint space

In many cases, the need to express constraints in the joint space arises: the most obvious cases are limits in force, position, or velocity. These kind of constraints are not task-specific since they dependent on the robot in use. In some cases, these tasks are used to maximise some runtime characteristic of the robot, such as the manipulability index, Ögren et al (2012).

In general, the expression on joint space is a generic scalar function of joints:

$$y = f(\boldsymbol{q}). \tag{1}$$

We can identify several prototype functions, however we will consider only selection functions:

$$y = f(\boldsymbol{q}, i) \triangleq [0, \cdots, 0, 1, 0, \cdots, 0]\boldsymbol{q} = q_i, \tag{2}$$

which allows to impose for example, joint limits and joint configurations, deferring other possibilities to future work.

3.4 Non-scalar expressions

Without loss of generality, we decided to focus on scalar functions, omitting multi-dimensional expressions, as relative poses or (3D) rotations, as these can be achieved as a composition of *projection of point on line* and *angle between versors* expressions.

3.5 Sensor-space expressions

In some applications constraints are directly expressed in the a sensor space, as for example in visual servoing. In this case, in addition to the robot kinematics, an additional relation that rules how a change in pose of the sensor affects the sensor space is necessary. As example, let's consider the case of face-following with a camera: in this case we will need to define a camera (camera model, mounting frame), and *sensor-space*

expression (*e.g.* a distance) that relates a measured feature (*e.g.* the barycenter of the tracked face) w.r.t a "virtual feature" (*e.g.* the center of the sensor space). This kind of expression can be again expressed as a generic (scalar) function:

$$y = f(\boldsymbol{q}, \boldsymbol{\chi_u})$$

where $\boldsymbol{\chi_u}$ represents the state of measured objects.

Incorporating these specifications does not present technical impediments, but addressing the problem in details goes beyond the goal of this work.

3.6 Virtual Fixtures and Mechanisms

Virtual fixtures are computer generated geometric features pioneered by Rosenberg (1992) to overlay real sensory feedback in telerobotics. They can be seen as a sort of a guide for a better perception. Later on Sayers (1999) extended virtual fixtures to generate control primitives in robot teleprogramming of remote robots in the presence of delay. Recently, they are largely exploited in teleoperation and shared control scenarios, especially in the robotic medical field (see Abbott et al (2007) and cited works). In this case, more complex primitives are employed, as often the goal of the task is to constraint the end effector to move in a subspace defined on top of generic curves, and leaving few free direction of motion. These motion primitives are simple and their combination for complex tasks is not always easy or even possible. The concept of *virtual mechanism* proposed in Joly and Andriot (1995) and also in Kosuge et al (1995) allows a more systematic description of the task and embed a task controller with desired properties (*e.g.* passivity). Virtual mechanisms can be expressed in the task or joint spaces but are very often task specifically designed. While adding these kind of geometric primitives and expressions can easily fit in our proposed framework (*e.g.* we can generalise the concept of line to curves), most of the common virtual mechanisms that relies on linear, rotational, or spherical degrees of freedoms can be obtained as a combination of constraints defined in the output spaces generated by geometric expression: for example, the task of grasp the handle of a drawer can be achieved by means of geometric primitives-based constraints (see Sec. 7.2), or could have been achieved with constraints defined on top of a virtual linear mechanism, and a purposely designed virtual mechanism that regulates the gripper orientation in such a way its fingers points toward the handle.

4 The behaviour

Once the output space is described by a joint or a geometric expression, we characterize the constraint to be applied along such space with the desired *behaviour*. The identified behaviours are the following:

- **Positioning**, used in position regulation problems.
- **Move toward**, used when we are interested in specifying the direction and rate of motion, rather than the desired final position.
- **Physical Interaction** used when we want to control force exchanged in physical contact.
- **Compliant positioning** used when we want to control the position of the system, while allowing for physical compliance in order to cope with unexpected or partially modelled contacts.
- **Limits** used for reducing the space of feasible solutions, in order to prevent the robot to go in undesired positions or to exert excessive forces.

In table 4, we match the behaviours with the needed characteristics of the system, described in terms of: *i)* the type of control, *ii)* the type of set-point (position, velocity , forces, either constant or provided by a trajectory generator), *iii)* and the type of constraint.

These functionalities (that must be provided by the system) are introduced here to exemplify the meaning of each behaviour. In particular: *i)* in the first three cases, the goal is to have a zero steady state error in either position, velocity, or force reference respectively, *ii)* in the compliance mode we want to regulate the position, but allowing deviations proportional to force, and *iii)* with limits, we want to specify the bounds of the system.

In addition, as shown in table 4, the first four behaviours need a sole parameter, that indicates *i)* the time constant of the error, in the first three cases, or *ii)* the relation between angular or linear displacement (along the output direction) w.r.t. the disturbance (force or torques respectively).

With the first three laws, we seek an asymptotically zero converging error in position, velocity, or force, respectively, and the error dynamic should be "similar" to

$$\dot{y}_d^\circ = \boldsymbol{K_p}(y_d - y). \tag{3}$$

where the gain $\boldsymbol{K_p}$ is the *specification*, and its dimension is [1/s], regardless of the constrained variable.

Using the compliant motion behaviour, instead, we expects to achieve a given displacement from a rest position as response of disturbance (*e.g.* an external force). Henceforth, the tuning parameter represents desired apparent stiffness:

$$\delta y = \mathbf{K}\delta f. \tag{4}$$

In this specification, for sake of brevity, we omit the specification of apparent damping and inertia, that could be as well incorporated in the specification. Lastly, in case of limiting-type behaviours, no parameter is defined.

In our opinion, the described behaviours should suffice to cover most of the tasks proposed in literature; however, if the need arises, it is possible to enrich the language with additional behaviours (or add additional parameters to the proposed ones), provided that

behaviour	*needs*: controller	setpoints (one of)					Force measurement	constraints		*specification*
		y_d	$\dot{y}_d$	$y_d, \dot{y}_d$	λ_d	$\lambda_d, \dot{\lambda}_d$		=	<	
Positioning	Position	✓		✓				✓		dominant pole [1/s]
Move toward	Velocity		✓					✓		dominant pole [1/s]
Physical interaction	Force				✓	✓	✓	✓		dominant pole [1/s]
Compliant motion	Impedance	✓					✓	✓		Stiffness [N/m] or [Nm/rad]
Position Limit	Position	✓							✓	
Velocity Limit	Velocity		✓						✓	
Force limit	Force				✓		✓		✓	

Table 4 List of behaviours: each behaviour is related the type of control and its specification, the type of set-point, the needed measurements (position measurement is always needed), and the related constraint (either equality or inequality).

the underling control framework and robotic systems is able to execute them.

5 Constraints definition

The elements given until now are enough to describe a constraint, and implicitly hints to the decision process that the control designer undergoes in drafting the constraints and the tasks, as shown in Sec. 6.

At this point, we introduce a more formal declarative description of the constraint, and of all the elements referenced there. The constraint, (Listing 3) is defined by an expression, a behaviour, and optionally a trajectory generator (when not specified, null references are assumed), and a specification (that can be either a bandwidth or a stiffness, see table 4) that can defaults to platform-dependent values.

Listing 3 Constraint definition.
a constraint is the collection of an expression (Listing 4) and a behaviour (table 4) .

```
Constraint=umf.class("Constraint")
constraint_spec = umf.ObjectSpec{
  name='constraint_spec',
  type=Constraint,
  sealed='both',
  dict={
    output_expression=ExpressionSpec{},-- ↩
        either geometric_expression_spec or↩
         joint_expression_spec
    behaviour=behaviour_type_spec,
    tr_gen=trj_gen_id_spec,
    specification=umf.NumberSpec{}},
  optional={"tr_gen","specification"}
}
```

Each expression is either a geometric or a joint expressions. In the geometric expression specification a check function is used, since not all the combinations of entities and expressions are legal (see table 3):

Listing 4 Geometric Expression definition.
A Geometric Expression is defined by two geometric primitives (Listing 5) and an expression (table 3).

```
GeometricExpression=umf.class("↩
    GeometricExpression")
geometric_expression_spec = umf.ObjectSpec{
  name='geometric_expression',
  postcheck=↩
      geometric_entity_vs_expression_check,
  type=GeometricExpression,
  sealed='both',
  dict={
    p1=primitive_spec,
    p2=primitive_spec,
    expression=↩
        geometric_expression_type_spec}
}
```

In geometric expressions are requested two primitives, each one composed by an entity and a frame, that is the reference frame upon which such entity is expressed:

Listing 5 Primitive specification.
A primitive aggregates an object frame and the attached entity.

```
Primitive=umf.class("Primitive")
primitive_spec = umf.ObjectSpec{
  name='primitive',
  type=Primitive,
  sealed='both',
  dict={
  entity=EntitySpec{},--either point_spec, ↩
      versor_spec, line_spec, or plane_spec
  object_frame=object_frame_spec}
}
```

The next step is the composition of constraints in a task, as shown in Sec. 6.3.

6 Tasks

Set of constraints are combined into *tasks*, and when one or more constraint space intersects, conflict can arise: in Sec. 6.1 we describe how these conflicts can be ruled out.

Moreover, tasks should be associated with ways to evaluate their status. For this reason each task has one or more measurement expressions, that follow the same rule for expressions given in Listing 4. These expressions, once compared with a reference value, provide the user or a supervisor system with helpful informations. We named this mechanism Monitoring and will give an overview in Sec. 6.2.

6.1 Hierarchical constraint composition

In all but the simplest applications, several constraints are enforced together. Since the initial conditions, the environment, and other aspects can be unknown at the time of defining the application, or varying between

executions, tasks could require objectives that cannot be achieved simultaneously, due to conflicts. Such conflicts can easily be individuated and avoided in the design phase only for trivial cases, but they are very difficult to foresee beforehand in a general scenario, where constraints are expressed in different spaces.

For this reason, it is necessary to express explicitly in which way conflicts should be handled at run time. Designing the behaviour of the system when constraints are conflicting cannot be done in an exhaustive way without peering to the underlying implementation and the possible options that it provides. To the best of the authors' knowledge, two methods are used: *i)* weighting (in an objective function) of "deflection" of constraints from their nominal values, employed in velocity- and acceleration- resolved schemes: it allows to specify the relative weight of the velocity or acceleration desired value, when computing the joint velocity of acceleration to be commanded to the robot, respectively, and *ii)* prioritization of constraints: achieved mostly with null space projectors.

In our experience, we found out that three levels of hierarchy suffice, in most applications, to lay out the constraints in such a way the behaviour is met. We named these three levels as follows:

1. Safety constraints: constraints that are necessary for the the safeguard of the system, (*e.g.* hardware limitations, non-desired collision avoidance, sustain balance in humanoid robots, etc.). These constraints are often formulated as inequalities, thus reducing the space of feasible solutions in which lower level constraints can be fulfilled. Constraints executed in this level should no be conflicting.
2. Primary constraints: the actual constraints to be executed. In this level it could be possible, depending by the solver capabilities to schedule constraints, that can conflict between them. In this case, relative weights will rule out the system behaviour.
3. Auxiliary constraints: constraints that facilitate the execution of primary constraints; an example is the optimization of the robot pose configuration with respect to the manipulability index, Ögren et al (2012); Borghesan et al (2014), another one is gazing, etc.

6.2 Monitoring

Monitoring mechanism is inspired by Bruyninckx and De Schutter (1996) and many other analogous approaches, and driven by the necessity to change the system behaviour in front of the fulfilling of some conditions. Monitors observe a variable and implement a logic predicate, raising an event once such event is fulfilled. A monitor can either observe:

1. a controllable variable (for example a variable that is used as constraint), or
2. a measured quantity that is influenced by the robot actions, but cannot be directly controlled. This quantity can be measured in terms of: *i)* a variable that lives in the space described by an expression, or *ii)* an external monitored value.

Examples of the latter case are distance to the nearest obstacle provided by a range finder, interaction (estimated) forces in compliant controlled system (where constraints are defined in terms of position), or a guard that dictates a time-out for task. For the cases 1) and 2*i*), a monitor is defined as : *i)* an expression (that specifies the space where the variable lives), *ii)* an event name (*e.g.* `finished, failed`), *iii)* the (monitored) variable type (`POSITION`, `VELOCITY`, `FORCE`), that is expressed in the space defined by the expression, *iv)* a comparison type ($<, >, \in, \notin$), *v)* reference value(s). The formal specification of the monitors is given in Listing 6.

Listing 6 Expression-based Monitor specification.
This monitor is defined by a the monitored expression (Listing 4), the type of comparison, and the event risen.

```
monitored_variable_type_spec = umf.EnumSpec↩
    {"POS","FORCE","VEL"}
comparison_type_spec = umf.EnumSpec{"EQUAL"↩
    ,"LESS","IN_INTERVAL","OUT_INTERVAL"}

ExpMonitor=umf.class("ExpMonitor")
exp_monitor_spec = umf.ObjectSpec{
  name='exp_monitor_spec',
  postcheck=comparison_type_check,
  type=ExpMonitor,
  sealed='both',
  dict={
  monitor_expression=ExpressionSpec{},
  event_risen=umf.StringSpec{},
  monitored_variable_type=↩
      monitored_variable_type_spec,
  comparison_type=comparison_type_spec,
  lower_bound=umf.NumberSpec{},
  upper_bound=umf.NumberSpec{},
  },
  optional={"lower_bound","upper_bound"}
}
```

where `comparison_type_check` is a function that checks the consistency between bounds and and the comparison type.

In the case of the external monitors the expression and the variable type are substituted by an identifier (`monitored_variable_name`), that points out to an externally computed value, as shown in Listing 7.

Listing 7 External Monitor specification.
These monitors relies on externally computed values.

```
ExtMonitor=umf.class("ExtMonitor")
ext_monitor_spec = umf.ObjectSpec{
  name='ext_monitor_spec',
  postcheck=comparison_type_check,
  type=ExtMonitor,
  sealed='both',
  dict={
  event_risen=umf.StringSpec{},
  monitored_variable_name=umf.StringSpec{},
  comparison_type=comparison_type_spec,
  lower_bound=umf.NumberSpec{},
  upper_bound=umf.NumberSpec{},},
  optional={"lower_bound","upper_bound"}
}
```

6.3 Task definition

Finally, we can introduce the task definition, in Listing 8. In the task specification, there is only one mandatory field, that is the primary constraints vector (that must include at least one element). Thus, the minimum specification of the task corresponds to a constraint: an expression, a behaviour that rules which kind of control must be used, and an objective given by a trajectory generator.

Listing 8 Task specification.
A task is a collection of constraints (Listing 3) scheduled at different priorities, and a collection of monitors (Listings 6 and 7).

```
Task=umf.class("Task")
task_spec = umf.ObjectSpec{
  name='task_spec',
  type=Task,
  sealed='both',
  dict={
    safety_constraints=↩
        constraint_array_spec,
    primary_constraints=↩
        constraint_array_spec,
    auxiliary_constraints=↩
        constraint_array_spec,
    monitors=monitor_array_spec},
  optional={"safety_constraints","↩
     auxiliary_constraints","monitors"}
}
```

Listing 8 concludes the description of the language; in the remainder of section we will focus on the problem of determining whether a task is feasible or not with a given system.

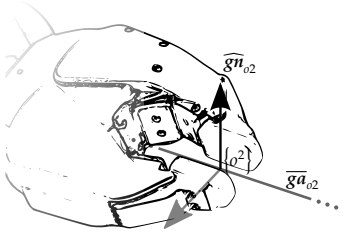
6.4 Task Execution

In order to be executed, the task description must be translated toward one the underling framework that provide to real motion control; however, not all the frameworks are able to execute all the possible options, and is therefore necessary to take such limitation at task design time. At this end, we are working towards a semantic standardization of the symbolic expression to refer to the task execution capabilities of robot platforms, where we refer as robot platform the system that comprises the robot and the control framework.

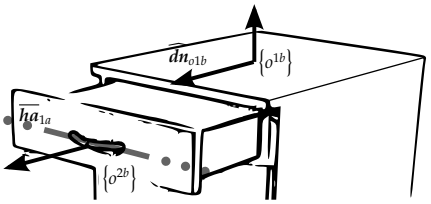
The focal points that have to be analysed are: *i)* the capability to implement a given behaviour in motion control strategy, and *ii)* the capability to solve (and derive) a given expression. We will not focus on such aspects and refer the reader to the vast literature in the state-of-the-art.

7 Task specification example

This section illustrates the proposed approach by means of an open-a-drawer operation composed of the following tasks: *i) Approach to handle of drawer*, *ii) Grasp the handle*, and *iii) Open the drawer*. We consider that the robot at hand is able to perform *Positioning* behaviour only.



(a) Gripper: the line **ga** represents the direction of approach of grasping, while versor **gn** must be parallel to the handle axis direction **ha**.



(b) Drawer: axis of the handle and direction of opening are the two main features.

Fig. 3 Some of the geometric entities involved in the task definition.

In the following, we briefly describe the tasks (starting from the identification of geometric primitives), and we present the result of a simulation, where a state machine is used in order to switch active task, in reaction of events triggered by monitors.

7.1 Geometric Primitives

The object frames that are involved in the Geometric Primitives are the following:

- $\{o^{1a}\}$, attached to the handle center, with the z axis along the handle itself.
- $\{o^{1b}\}$, attached to the chest of drawers (fixed in the world).
- $\{o^2\}$, the grasp frame of the robotic hand, *i.e.* the frame that is the center of the grasp once the hand closes.

On such frames, the following Geometric primitives are defined:

- $\boldsymbol{hp}_{o1a}$ (`handle_position`): *point* in origin of $\{o^{1a}\}$.
- $\boldsymbol{hip}_{o1b}$ (`handle_initial_position`): *point* in $\{o^{1b}\}$; it coincides with the previous when the drawer is closed.
- $\boldsymbol{gp}_{o2}$ (`grasp_position`): *point* in origin of $\{o^2\}$.
- $\widehat{\boldsymbol{gn}}_{o2}$ (`grasp_normal_direction`): *versor* aligned with the $x-$axis of $\{o^2\}$.

- $\widehat{\boldsymbol{ha}}_{o1a}$ (`handle_axis_direction`): *versor* aligned with the $z-$axis of $\left\{o^{1a}\right\}$.
- $\overline{\boldsymbol{hx}}_{o1a}, \overline{\boldsymbol{hy}}_{o1a}, \overline{\boldsymbol{hz}}_{o1a}$ `handle_axis_(x-y-z)`: three *lines* with origin in $\left\{o^{1a}\right\}$, and aligned with the $x-$, $y-$, or $z-$axis, respectively.
- $\overline{\boldsymbol{ga}}_{o2}$ `grasping_axis`: *line* with origin in $\left\{o^{2}\right\}$, and aligned with its $z-$axis.
- $\overline{\boldsymbol{oa}}_{o2}$ `opening_axis`: *line* with origin in $\left\{o^{1b}\right\}$, and aligned with the opening direction.

7.2 Task descriptions

The example application is ruled by a simple three-states state machine, whose state transitions are executed in response of the event raised by monitors (described in Sec. 7.3). The states (*approach the tray*, *grasp the handle*, and *open the drawer*) are ordered sequentially, and for each state a different set of *primary constraints* is enforced. In addition, an underling *auxiliary task* described in joint space suggests the robot to keep its arms in a central position w.r.t. the joint limits (*Positioning behaviour*), while a *safety constraint* limits the controller to joint ranges (*Position Limit behaviour*).

Approach the tray (S.1): In this phase, the robot should bring its end effector in such a position that can conveniently grasps the handle. To do so, the robotic hand should be: *i)* oriented toward the object, *ii)* rotated along its z- (approach-) axis in the "correct" way, and then *iii)* bring the distance to between the grasping point and the handle center to zero. This description translates in a set of Positioning behaviours, ruled by the following expressions:

a) *line-point distance* between $\overline{\boldsymbol{ga}}_{o2}$ and $\boldsymbol{hp}_{o1a}$ should be zero,
b) *angle btw versors* $\widehat{\boldsymbol{gn}}_{o2}$ and $\widehat{\boldsymbol{h}}_{o1a}$ should be zero,
c) while *line-point projections* between $\overline{\boldsymbol{hx}}_{o1a}, \overline{\boldsymbol{hy}}_{o1a}, \overline{\boldsymbol{hz}}_{o1a}$ and $\boldsymbol{hp}_{o1a}$ must go to zero should go to zero.

These set of constraints allow for a one full degree of freedom (angle of approach to the handle around its normal). Note that expressions c) dictate that the distance between the origins of the frames $\left\{o^{1a}\right\}$ and $\left\{o^{2}\right\}$ be zero. However, we express three constraints in place of a *point-point distance* as the derivative of such expression is ill-defined at the desired value, thus causing local instability around such point.

Grasp the handle (S.2): While the positioning constraints are then held in compliant motion mode, we close the gripper; in this way, we can comply with inaccuracies in pose estimation, etc.

Open the drawer (S.3): At this point, the drawer is opened, so we command to increase the distance along the line direction of opening, using a compliant motions that increase the point-line projection between $\overline{\boldsymbol{oa}}_{o2}$ and $\boldsymbol{gp}_{o2}$. As we do not want the handle grasp to be broken, we continue to enforce previous constrains.

For sake of brevity, we focused on only one gripper; however, nothing prevents to express constraints on other frames attached to other robot links *e.g.* use the other gripper or forearm to keep firm the chest of drawer.

7.3 Monitors and trajectory generator

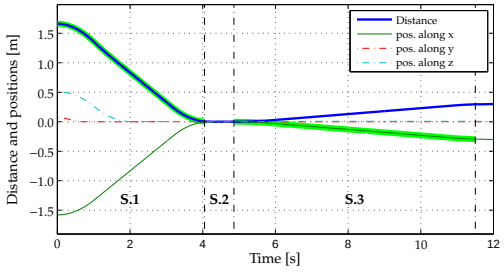
For the example at hand, we will limit a sketch of possible monitors that can dictate the success of the action, and leave the reader to figure out how failure can be detected.

a) Approach the tray: the final goal is to reach the center of the handle. In this case, we use the *point-point distance* between the origins of frames $\left\{o^{1a}\right\}$ and $\left\{o^{2}\right\}$. In this case, the monitored expression is different to the controlled one.
b) Grasp the handle: success can be achieved by force direct/indirect measurement in the grasp space. However, will rely on position informations (the distance between fingertips).
c) Open the drawer: success can be acknowledged by monitoring when the distance between the gripper position and the initial drawer position is above a given threshold.

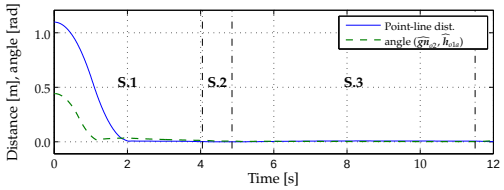
All the constraints that are described can be simply commanded to the final value; however, this would lead to a jerky behaviour. In many cases a simple set point generator, as a trapezoidal velocity one, will generate smother movements, and will allow to rule out race conditions. We delegate further investigations to future works.

7.4 Simulation Results

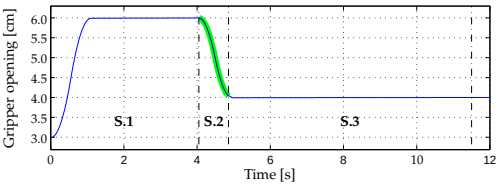
We implemented the above example in a kinematic simulation, taking advantage of the a module that allows the automatic derivation form this task DSL to the eTaSL language Aertbeliën and De Schutter (2014), that can be executed (either in simulation or on a real system), in realtime. At the current state-of-the-art, eTaSL supports for velocity resolved systems, and takes advantage of the quadratic programming solver described in Ferreau et al (2014) to treat the optimization problem that computes, at each time instant, the desired joint velocity. In our case, we interpolate all the trajectories by means of trapezoidal velocity profile, and we solve racing conditions (that could arise in the first task, where several constants are enforced concurrently) by imposing a slower trajectory on some directions, *e.g.* on the approach direction, in comparison to the alignment constraints that should be fulfilled before the task is near its conclusion. The results can be appreciated in Fig. 4, where the time evolution



(a) x, y, z positions of gripper w.r.t. the handle frames, and total distance between frame origins.



(b) Point-line distance between the gripper grasping direction and the handle frame origin, and misalignment between the handle axis and normal grasping directions.



(c) Distance between gripper fingers.

Fig. 4 Values of constrained and monitored expressions during simulation. Vertical lines shows the transitions between states (labelled from **S.1** to **S.3**, each one corresponding to a set of *primary constraints*). In the first state, the gripper is opened (Fig. 4c), aligned (Fig. 4b), and the brought to the the handler (Fig. 4a). State transition is triggered when total distance (blue flat line in Fig. 4a) decreases under a given threshold. In second phase the gripper is closed (Fig. 4c), and, lastly, the gripper is commanded to move back to $x = -0.3\,\mathrm{m}$ (thin solid line in Fig. 4a). The monitored expressions in each task are highlighted with a green underling line.

of all expressions are reported. In these figures we reported also which is the monitored expression, and the transitions between tasks.

8 Conclusion

We proposed a way to represent tasks with a limited set of mnemonic elements (and very few numerical values) with the aim to provide a way to unify under the hood different approaches. The benefit of this process is manifold: first of all, striving for standardization and normalization of systems allows for comparison between similar system, portability and reusability of applications, and easiness of use. On the other side, by defining constraints by means of a limited number of elements it will be possible to design reasoning algorithms that can sort out the best plan for a given action, *e.g.* Beetz et al (2010).

Naturally, standardization and simplification comes at the cost of pruning some of the possibilities offered by each framework, thus limiting the user to the provided functionalities, *e.g.* discarding the explicit use of virtual kinematic chains (that can be used, instead, in the implementation of the geometric expressions) limits the flexibility of defining output spaces, or fine-tuning of weights and gains in the control function. Nonetheless, the set of geometric primitives (and relations) can be extended, e.g. segments, planar polygons, solids, oriented curves to provide virtual guidance *etc.*. While most of the cases can be achieved as combination of primitive, native implementation of complex relations is beneficial for the clarity of language and computational load.

On the other hand, the language can be extended toward non-geometric constraints (*e.g.* manipulability indexes), or in order to provide black-boxed constraints that are platform dependent, (*e.g.* "maintain the equilibrium" in a humanoid robot platform).

References

Abbott J, Marayong P, Okamura A (2007) Haptic virtual fixtures for robot-assisted manipulation. In: Robotics Research, Springer Tracts in Advanced Robotics, Springer, pp 49–64

Aertbeliën E, De Schutter J (2014) etasl/etc: A constraint-based task specification language and robot controller using expression graphs. In: IEEE/RSJ Proc. of International Conference on Intelligent Robots

Beetz M, Mosenlechner L, Tenorth M (2010) Cram — a cognitive robot abstract machine for everyday manipulation in human environments. In: IEEE/RSJ Proc. of International Conference on Intelligent Robots, pp 1012–1017

Borghesan G, Aertbeliën E, De Schutter J (2014) Constraint- and synergy-based specification of manipulation tasks. In: IEEE Proc. of the Int. Conf. on Robotics and Automation

Bruyninckx H, De Schutter J (1996) Specification of force-controlled actions in the "task frame formalism" - a synthesis. IEEE Transactions on Robotics and Automation 12(4):581–589, DOI 10.1109/70.508440

De Schutter J, De Laet T, Rutgeerts J, Decre W, Smits R, Aertbelin E, Claes K, Bruyninckx H (2007) Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. International Journal of Robotic Research 26(5):433–455

Ferreau H, Kirches C, Potschka A, Bock H, Diehl M (2014) qpOASES: A parametric active-set algorithm for quadratic programming. Mathematical Programming Computation (in print)

Joly LD, Andriot C (1995) Imposing motion constraints to a force reflecting telerobot through real-time simulation of a virtual mechanism. In: IEEE Proc. of International Conference on Robotics and Automation, Nagoya, Japan, vol 1, pp 357–362

Klotzbücher M (2013) micro Modelling Framework (uMF). URL `github.com/kmarkus/uMF`

Kosuge K, Itoh T, Fukuda T, Otsuka M (1995) Tele-manipulation system based on task-oriented virtual tool. In: IEEE Proc. of International Conference on Robotics and Automation, Nagoya, Japan, vol 1, pp 351–356

Kröger T, Finkemeyer B, Wahl FM (2004) A task frame formalism for practical implementations. In: IEEE Proc. of In-

ternational Conference on Robotics and Automation, New Orleans, LA, USA, pp 5218–5223

Kröger T, Finkemeyer B, Wahl FM (2011) Manipulation primitives - a universal interface between sensor-based motion control and robot programming. In: Robotic Systems for Handling and Assembly, Springer Tracts in Advanced Robotics, vol 67, Springer Berlin Heidelberg, pp 293–313

Mansard N, Khatib O, Kheddar A (2009) A unified approach to integrate unilateral constraints in the stack of tasks. IEEE Transactions on Robotics 25(3):670–685

Mason MT (1981) Compliance and force control for computer controlled manipulators. Systems, Man and Cybernetics, IEEE Transactions on 11(6):418–432, DOI 10.1109/TSMC.1981.4308708

Ögren P, Smith C, Karayiannidis Y, Kragic D (2012) A multi objective control approach to online dual arm manipulation. In: Proc. of 10th IFAC Symposium on Robot Control, Dubrovnik, Croatia, pp 820–825

Rosenberg LB (1992) The use of virtual fixtures as perceptual overlays to enhance operator performance in remote environments. Technical Report AL-TR-1992-XXX, Wright Patterson Air Force Base, OH: U.S.A.F Armstrong Laboratory

Sayers C (1999) Remote control robotics. Springer Verlag

Sentis L, Khatib O (2006) A Whole-Body Control Framework for humanoids operating in human environments. In: IEEE Proc. of the Int. Conf. on Robotics and Automation

Vaillant J, Kheddar A, Audren H, Keith F, Brossette S, Kaneko K, Morisawa M, Yoshida E, Kanehiro F (2014) Vertical ladders climbing by the HRP-2 humanoid robot. In: IEEE/RAS International Conference on Humanoid Robots, Madrid, Spain