eTaSL/eTC: A constraint-based Task Specification Language and Robot Controller using Expression Graphs

Erwin Aertbeliën and Joris De Schutter

***Abstract*— This paper presents a new framework for constraint-based task specification of robot controllers. A task specification language (eTaSL) is defined as well as a corresponding implementation of a controller (eTC). This new framework is based on feature variables and a new concept referred to as expression graphs. It avoids some of the common pitfalls in previous frameworks, and provides a flexible and composable way to define robot control tasks. An architecture for a robot controller is proposed, as well as an implementation that can execute tasks described in the new specification language. Typical usage patterns for the new framework are explained on an example consisting of a kinematically redundant, bi-manual task on a PR2 robot. A comparison with existing frameworks shows the advantages of the new approach.**

I. INTRODUCTION

This paper presents a new framework for constraint-based task specification of robot controllers. A task specification language (*e*xpressiongraph-based *Ta*sk *S*pecification *L*anguage, eTaSL) is defined as well as a corresponding implementation of a controller (*e*xpressiongraph-based *Ta*sk *C*ontroller, eTC).

Distinction is made between discrete control tasks, where the focus lies on sequencing and scheduling different discrete actions (e.g. rFSM [1], SMACH [2], RosCo [3]), and continuous control tasks, where a set of objectives is continuously achieved. These two types of tasks are considered two independent problems in task specification. Care is taken to obtain a clear separation with a well-defined interface between the corresponding control tasks. The focus of this paper lies on continuous task specification.

Of course, execution of a real-life robot application includes other aspects such as planning. However, the proposed framework focuses on *continuous reactive control* that also can deal with on-line sensor measurements.

One of the recurring themes in this paper is *the separation of concerns*. On the implementation side, the separation of concerns focuses on the 5C's [4], i.e. the separation of *c*omputation, *c*onfiguration, *c*oordination, and *c*ommunication, while taking into account *c*omposability. On the specification side, it focuses on separating robot-related, task-related and environment-related aspects of a robot application. Using traditional, trajectory-based task specification such a separation is difficult to achieve, especially when dealing with robots with a larger number of joints, such as the PR2 robot. A common approach to this problem is to start from a constraint-based approach, where the task is specified in terms of constraints [5][6][7]. This approach originates from the research on redundant robots. When it only concerns positioning tasks, it is sometimes referred to as Generalized Inverted Kinematics [8].

The contribution of this paper is to describe a new task specification language eTaSL and a corresponding implementation of a controller eTC, to explain a series of usage patterns, and to compare eTaSL with existing frameworks. A key component in this language is an expression graph, a data representation for geometric operations. A dual arm task is explained and demonstrated, but this example serves only as an illustration of the usage patterns.

Section II starts with defining the terminology and detailing the architecture. Section III describes the basic elements and principles of the task specification language. Section IV explains how this task specification is translated into a running task controller. In section V, a (non-exhaustive) list of usage patterns explains the effective use of the framework. These usage patterns are demonstrated on a dual arm manipulation task of a PR2 robot. Section VI compares eTaSL with the *i*nstantaneous *Ta*sk *S*pecification using *C*onstraints (iTaSC) framework [6] and *S*tack *o*f *T*asks (SoT) framework [8].

II. IMPLEMENTATION AND ARCHITECTURE

The implementation of the controller (eTC) is a library that is split up into three layers, as is shown in fig. 1. In the *specification layer*, a specification is built up using a task specification language (eTaSL) that defines *constraints* that have to be satisfied while executing the task at hand. In the *solver* layer, this specification is translated into a numerical optimization problem, typically a quadratic programming (QP) problem. The *numerical solver* layer solves the optimization problem. The framework handles the continuous part of the control task. The discrete part of the control task is handled by defining *monitors* that can generate events, and by allowing activating/deactivating groups of constraints while the controller is running. In this way, the discrete part of the task specification can be handled outside eTaSL, e.g. using a state space specification such as rFSM [1], SMACH [2], or [3]. The framework is implemented as a library, and is independent of the execution environment (e.g. Orocos [9] or ROS [10]). An example ROS component is provided to demonstrate integration within ROS. This architectural approach fits nicely to the separation of concerns principle

All authors are with the KU Leuven, Department of Mechanical Engineering, Belgium. All authors gratefully acknowledge the financial support by the European Community's Seventh Framework Programme projects RoboHow(FP7-ICT-288533) and Factory-in-a-day(FP7-609206). Corresponding author: erwin.aertbelien@kuleuven.be

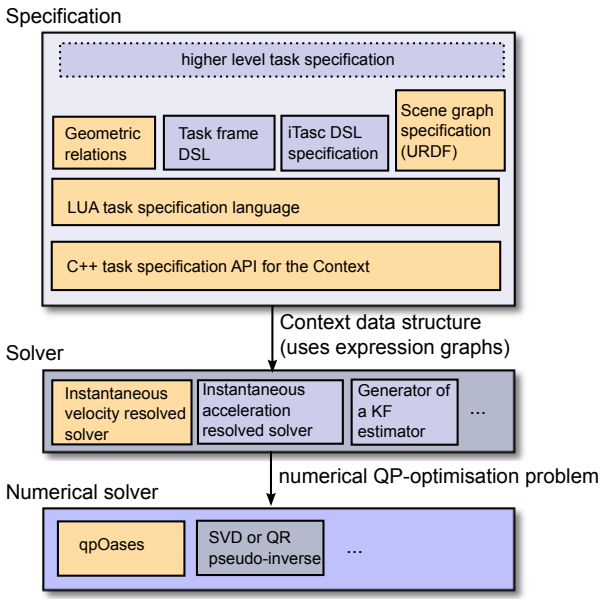

Fig. 1. The task specification language and implementation fits into a 3-layered architecture. The light-colored boxes are currently implemented.

of the 5C's [4] separating computation, coordination, configuration, composition and communication.

Fig. 1 shows how this architecture is used by the current implementation. The numerical problem is solved using the qpOASES toolbox [11]. A solver is implemented that translates the specification into a velocity-resolved constraint-based task controller. The *context* is a data structure that contains the complete definition of the task. The specification layer provides facilities that build up this context. A C++ API is provided to build up the context. Desired tasks can be completely specified using this API, but typically, the task specification language eTaSL is used. This language is based on Lua [12], an extendable and embeddable scripting language, and provides an easy way to specify the task. Constraint programming languages from the domain of artificial intelligence are not applicable, since we are dealing with a non-linear *control* problem, and not a pure constraint satisfaction problem. The choice for Lua is pragmatic and is based on its suitability for embedded systems and its common usage in the Orocos environment [1] and in the iTaSC DSL language[7]. On top of this task specification language, facilities are provided to handle common geometric relations (such as distances and angles between different types of entities). An extension to the language is provided to generate the appropriate expressions and constraints for scene graphs and robot models from an URDF-specification.

The task specification language allows for easy separation between the robot related aspects, task related aspects and the environmental aspects (see section V).

III. SPECIFICATION LANGUAGE

The task specification language is based on Lua and provides easy-to-use primitives that refer to the underlying C++ API. This section explains the concepts and syntax of these primitives.

The *context* is the data structure to be sent to the solver containing all information of the user-level specification of the task. It is also an entity in the task specification language. A task specification is always added to a given context. Multiple contexts can be used, e.g. in the case of multiple independently controlled robots.

Variables are explicitly defined and named (`name`) in the specification and are always specific for a given context (`context`). They are managed by the framework and are always accessed by name. This is a key factor providing composability and abstraction. There are different types of variables (`vartype`): *Robot joint variables* are actuated and the task controller needs to provide setpoint velocities or accelerations for these variables. There is an explicit *time variable*. This has as a consequence that trajectories can be (but does not need to be) defined inside the task specification. *Feature variables* are auxiliary variables used to specify free movement. They do not correspond to an actuated joint and do not need to correspond to a physical entity. Section V explains the use of feature variables. Variables can have a weight (`weight`) that is interpreted by the specific solver that is used (see section IV). The following gives an example of a variable definition:

```
Variable {
  context    = ctx,
  name       = 'along_path',
  vartype    = 'feature',
  weight     = 1.0
}
```

Expressions can be stored in an *expression variable*. These expressions are internally represented as *Expression graphs*. They consist of a tree-like data structure that represents a function. The expression graphs used in this framework consist of multiple value types to support expression of geometric relations between rigid bodies: scalar values, three-dimensional vector values, twists, rotation matrices and transformation matrices. This symbolic data structure supports different operations such as: serialization (i.e. translation into a storable format; for storage and transmission), evaluation of the value of the expression; evaluation of (possibly higher-order) partial derivatives and Jacobians using automatic differentiation [13]; and inspection to determine the variable dependencies. Evaluation is performed efficiently and on demand: parts of the expressions that did not change are not recalculated. The solver can eliminate unused variables by keeping track of variable dependencies. Consequently, the specification can freely define variables and expressions; only when they are effectively used, they contribute to the computation time of the controller.

A *constraint* can be at the position-level or at the velocity level. A constraint can be an equality constraint (`target`), or an inequality constraint (`target_lower` / `target_upper`). A position-level constraint expresses the desire that a given expression (`expr`) evolves towards and follows a given target. The dynamics by which this expression evolves is specified by a value `K` whose semantics are defined by the solver (see section IV). A set of

constraints can be conflicting, i.e. they cannot be satisfied simultaneously. Constraints with a higher `priority` are always satisfied before lower priority constraints. To ensure safety, the highest priority constraints cannot be conflicting. Conflicting constraints of the same priority can be given a `weight` to indicate their importance with respect to each other. Semantically, the constraints specify a Lagrange condition (i.e. enforced along the whole time interval the task controller is active); they do not specify a Mayer condition (that only needs to be achieved at the end of the task). The following gives an example of a constraint definition:

```
Constraint{
 context  =  ctx,
 name     = 'point_to_point_distance',
 expr     = origin(arm)-origin(trajectory),
 target   = {0.0, 0.0, 0.0},
 K        = 4,
 weight   = 1.0,
 priority = 2
}
```

The right hand side of `expr` gives an example of an expression graph that expresses the vector difference between the origin of a frame at the arm end effector (`arm`) and a trajectory point (`trajectory`).

Monitors specify the conditions under which actions (`actionname`) are invoked. A typical action is an event that is sent towards a state machine. The monitors are decoupled from the underlying execution system. Monitors observe the value of an expression and are edge triggered: when the value of the expression exceeds the bounds, an event is fired only once. There can be an upper bound (`upper`) and/or a lower bound (`lower`). An event is also fired when the value is outside the interval when the monitoring starts up. The following gives an example of a monitoring definition:`arm` is an expression variable containing an expression dependent on a series of robot joint variables. `trajectory` is an expression variable dependent on the time variable.

```
Monitor {
   context = ctx,
   name    = "goal_reached"
   expr    = norm( origin(arm)-origin(goal) ),
   lower   = 1E-4,
   actionname     = "event",
   argument       = "e_goal_reached"
}
```

In the example above an event with label `e_goal_reached` is sent out when the condition is triggered (i.e. distance between).

Besides the previously defined concepts, the specification also contains mechanisms to define inputs and outputs of the constraint controller. This allows us to take in sensor data from e.g. a force sensor or a camera. There is also a mechanism to group the constraints in a specification, such that they can be selectively activated or deactivated.

IV. SOLVER

The solver translates a given specification (see section III) into a numerical optimization problem representing the control strategy (see fig. 1), in this case, a velocity-resolved constraint controller.

At each time step a quadratic optimization problem of the following form is solved in order to compute the joint velocity input towards the robot actuators:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{x}^T\mathbf{H}\mathbf{x} && \text{(1a)} \\ \text{subject to} \quad & \mathbf{L}_A \leq \mathbf{A}\mathbf{x} \leq \mathbf{U}_A && \text{(1b)} \\ & \mathbf{L} \leq \mathbf{x} \leq \mathbf{U} && \text{(1c)} \end{aligned}$$

$\mathbf{H}$ corresponds to the Hessian of the optimization problem, $\mathbf{L_A}$ and $\mathbf{U_A}$ corresponds to the lower and upper bounds of the (linear) constraints described by the matrix $\mathbf{A}$. The box optimization constraints (1c), with bounds $\mathbf{L}$ and $\mathbf{U}$ are commonly occurring as velocity limit task constraints. Additionally, they are more efficiently solved than the more general optimization constraints (1b) (as e.g. in the implementation of qpOASES [11]). The optimization variable $\mathbf{x}$ corresponds to $[\dot{\mathbf{q}}^T \quad \dot{\boldsymbol{\chi}}_f^T \quad \boldsymbol{\varepsilon}^T]^T$. $\dot{\mathbf{q}}$ corresponds to the n_r robot joint velocities, $\dot{\boldsymbol{\chi}}_f$ corresponds to the n_f feature variable velocities and $\boldsymbol{\varepsilon}$ corresponds to the n_s slack variables that will be used to introduce task constraints with a lower priority.

The remainder of this section explains how each task constraint of section III is translated into a part of the above optimization problem using a methodology inspired on Samson's task functions [5], extended with an explicit time dependency, and with task functions that have the rotation group $SO(3)$ as range. These task functions are constraints at the position level that have to be obeyed according to a specified dynamic behavior.

For a scalar task constraint i, a task function $e_i(\mathbf{q}, t)$ is defined. This task constraint, and hence the corresponding task function, can have explicit time dependencies. The following equation imposes an evolution of this task function as a first order system with time constant K^{-1}:

$$\frac{d}{dt} e_i(\mathbf{q}, \boldsymbol{\chi}_f, t) = -\mathbf{K} e_i(\mathbf{q}, \boldsymbol{\chi}_f, t). \tag{2}$$

By expanding the total derivative in the above equation into partial derivatives using:

$$\frac{de_i}{dt} = \frac{\partial e_i}{\partial t} + \sum_{j=1}^{n_r} \frac{\partial e_i}{\partial q_j}\dot{q}_j + \sum_{k=1}^{n_f} \frac{\partial e_i}{\partial \chi_{f,k}}\dot{\chi}_{f,k}, \tag{3}$$

rewriting the partial derivatives as a Jacobian row $\boldsymbol{J}_i$ and introducing an additional slack variable ε_i, the equation becomes:

$$\mathbf{J}_i \begin{bmatrix} \dot{\mathbf{q}} \\ \dot{\boldsymbol{\chi}}_f \end{bmatrix} = -\mathbf{K} e_i - \frac{\partial}{\partial t} e_i + \varepsilon_i. \tag{4}$$

This equation defines a row in constraint (1b) of the above optimization problem, making sure that the robot moves in such a way that the desired evolution of the task function is achieved. In the case of hard constraints (i.e. high priority constraints) the term ε_i is left out, in the case of soft constraints (i.e. lower priority constraints), the term ε_i is added and an additional term of $w_i\varepsilon_i^2$ is added to the optimization criterion (1a). The introduction of the slack variables allows for conflicting constraints and the resulting controller will

optimally approximate the desired specifications, given the specified weights.

For a scalar inequality constraint, (2) becomes:

$$\frac{d}{dt}e_i\left(\mathbf{q}, \boldsymbol{\chi}_f, t\right) \leq -\mathbf{K}e_i\left(\mathbf{q}, \boldsymbol{\chi}_f, t\right), \tag{5}$$

in other words, the task function should not move faster towards its limits than a first order linear system with a given time constant K^{-1}. This way, there is no sudden application of the inequality, and undesirable dynamic effects can be avoided.

Joint velocity constraints can be easily translated into rows of the optimization constraint (1c). In addition, the joint position constraints can be translated into the form of (1c) in a similar way as the previous more general constraints.

The Hessian $\boldsymbol{H}$ in (1a) has the following form:

$$\boldsymbol{H} = \begin{bmatrix} \mu\boldsymbol{W}_r & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mu\boldsymbol{W}_f & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mu\boldsymbol{I} + \boldsymbol{W}_s \end{bmatrix} \tag{6}$$

$\boldsymbol{W}_r$ corresponds to the robot joint space weights, $\boldsymbol{W}_f$ correspond the feature space weights, and $\boldsymbol{W}_s$ consists of the previously introduced constraint weights w_i for the soft constraints. μ is a small numerical value. On the one hand, μ should be large enough to ensure that the optimization problem (1) remains positive definite and sufficiently regularizes the QP problem. On the other hand, μ should be small enough to ensure that the joint space and feature space velocities have a negligible influence on the soft constraints. μ can be considered mostly independent from the specific task specification at hand. The effects of μ are comparable to the effects of the damped least squares method for solving kinematic redundancies and task constraints [14]. The weights $\boldsymbol{W}_r$ and $\boldsymbol{W}_f$ only have a significant influence on the resulting control when the task specification is redundant. In such a case, these weights influence the null space motion of the robot system.

The current implementation uses the qpOASES solver [11] to solve the resulting numerical optimization problem. At each time sample, the constraint controller measures the robot joint positions, composes and solves a QP problem corresponding to the task specification, and applies the obtained robot joint velocities $\dot{\mathbf{q}}$ to the low-level robot controller. The computed feature variable velocities $\dot{\boldsymbol{\chi}}_f$ are integrated numerically.

When the constraint controller starts to execute a given task specification, $\mathbf{q}$ is initialized from the measured robot joint positions. In order to initialize the feature variables $\boldsymbol{\chi}_f$, a QP problem needs to be solved similar to (1). However, in this QP problem, the robot joint velocities $\dot{\boldsymbol{q}}$ become a parameter and are equal to $\mathbf{0}$ and the optimization variable $\boldsymbol{x}'$ corresponds to $[\dot{\boldsymbol{\chi}}_f^T \quad \boldsymbol{\varepsilon}^T]^T$.

V. USAGE PATTERNS AND DEMONSTRATOR

This section explains a series of typical usage patterns within the proposed framework. These usage patterns are illustrated on a demonstrator example given in fig. 2. Both the left and right manipulator arms of the robot follow a circular trajectory. The trajectories overlap with each other. The right manipulator follows its trajectory with a constant velocity; the left manipulator adapts its velocity in order to avoid collision between the manipulators. All joint position and velocity limits should be respected. At the same time, the head of the PR2 robot follows the tip of the right hand manipulator. The specification is translated into a reactive control strategy. In contrast to planning approaches, this control strategy could also immediately incorporate sensor feedback.

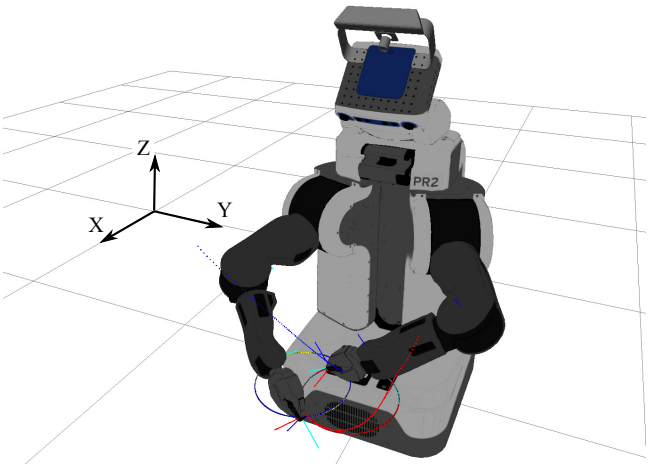

Fig. 2. A task on a PR2 robot to demonstrate a few of the capabilities of the task specification language.

A. Separation of robot, environment and task

To achieve reusable task specifications, it is important to separate the specification of the robot, the environment and task. The proposed task specification framework is a relatively low-level specification that is very suited to combine constraints from different sources. At the level of the task specification language, there is no essential distinction between robot, task or environmental constraints. All are made up of variables, expression graphs and constraints. By using the standard programming language facilities of LUA, such as Lua subroutine definitions and Lua tables, it is easy to write subroutines that define the variables, expression graphs and constraints belonging to a robot, environment or task. For example, collision avoidance is implemented as a library inside the task specification language.

The higher-level robot model can be imported into the library. The user can specify a URDF-model of a robot and a series of links of interest. Expression graphs are generated for the transformations between these links, as well as constraints that enforce the joint position and velocity limits.

For example, in the demonstrator application, an expression variable `rightarm` is created that contains an expression graph for the pose of the right end effector to the world frame (the same for `head` and `leftarm`).

B. Background constraints

Many tasks impose fewer constraints than there are degrees of freedom. To avoid that these degrees of freedom

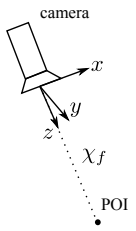

Fig. 3. The use of a feature variable χ_f to simplify the definition of the *head following* constraint.

will take on arbitrary, unpredictable, values, one typically defines *background constraints*: constraints with a very low weight that will only dominate in a certain direction when there is no constraint working in that direction.

The background constraints in the demonstrator example enforce each joint to a nominal value within its range with a very low weight. Often, these background constraints depend on the environment, e.g. for tasks on a kitchen countertop (or on a conveyor belt) specific background constraints are chosen.

C. The use of feature variables

The use of feature variables *facilitates the formulation of complex constraints*. An illustration of this is the iTaSC formalism [6] where, using a strict methodological procedure, the task is defined as a six-degrees-of-freedom virtual kinematic chain (VKC).

The head following constraint in the demonstrator is also an example of the use of feature variables. The point of interest (POI) is defined in a local frame at the head, using one additional feature variable (χ_f), and transformed to world coordinates. A soft constraint is set such that this POI coincides with the origin of the frame at the end effector of the right arm (see fig. 3):

```
chi_f=Variable {
  context=ctx, name='distance', vartype='feature'
}
poi = head*vector(constant(0),constant(0),chi_f)
Constraint{
 context  =  ctx,  name = 'head_following',
 expr     = origin(rightarm)-poi,
 target   = 0.0,
 K        = 4,    weight = 1.0,  priority = 2
}
```

The feature variable in the above example indicates that the corresponding distance can freely vary.

Another example of a feature constraint in the demonstrator is the following. To ensure that the velocity of the left arm trajectory can be adapted, the trajectory is defined in function of a feature variable (path variable) pv, and a soft constraint is set such that this feature variable follows `0.2*time` (a velocity of $0.2m/s$):

```
pv = Variable {
   context=ctx, name='along_circle',
   vartype='feature'
}
Constraint{
   contex  =  ctx,  name      = 'follow',
   expr    = pv-constant(0.2)*time,
   target  = 0.0,
   K       = 0,  weight = 0.05,  priority = 2
}
```

TABLE I

IMPLEMENTED DISTANCE AND ANGULAR RELATIONSHIPS

	Point	Line	Segment	Plane
Point	D/†			
Line	D/†	D/A		
Segment	D/†	D/A	D/A	
Plane	D/†	†/A	D/A	†/A

D:distance relationship is implemented, A:angular relationship is implemented †:not relevant. Redundant relationships are not indicated.

The control constant K is zero, such that the path variable does not catch up with lost time. The weight is low, such that the system prefers to vary the velocity along the path rather than deviating from the path.

The previous demonstrates that feature variables facilitate the formulation of task constraints. They can be used to specify *additional freedom*, in contrast to constraints alone that can only specify *restrictions*.

D. Explicit functions of time

Trajectories can be defined inside the task specification due to the availability of an explicit time variable, as is shown in the demonstrator example. Using the solver of section IV, this will not lead to tracking errors, because of the $\partial e_i/\partial t$ term in (4).

Alternatively, the task controller can take trajectory information from a separate trajectory generator outside the controller. To avoid tracking errors in such a case, this external trajectory generator should also provide time derivatives for its signals (see also the velocity feedforward in [6]).

E. Distances and collision constraints

Geometric relations are implemented in a higher-level library (see fig. 1) using the task specification language itself. This library can return expressions for the distance and angular relationships between the following entities: points, lines, line segments, and planes (see table I). In the demonstrator example that will follow, the links of the robot are modeled as line segments. Collisions between the manipulator links are avoided by imposing a constraint on the distance between these line segments: The (complicated) expression for this distance is directly formulated in the task specification language and used in a constraint. A more detailed model of the robot geometry could be implemented using spherically extended convex hulls [15], in such a case, the computation of the distances would be implemented (in e.g. C++ using an external library) as a new type of expression graph node to be used in the expression graphs.

F. Simulation results for the demonstrator

The eTaSL specification for the demonstrator is 104 lines long, and leads to a controller with 18 robot variables and 2 feature variables. There are 68 constraints. The examples of section V cover parts of the task specification of this

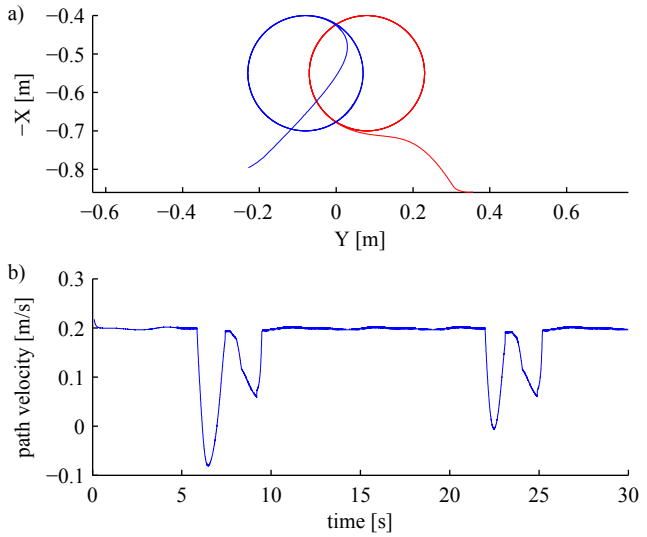

Fig. 4. (a) motion of the two robot end effectors in the horizontal plane, together with the nominal path. (b) The path velocity along the nominal trajectory of the left end effector is plotted in function of time.

TABLE II
COMPARISON BETWEEN FRAMEWORKS

	eTaSL/eTC	iTaSC	SoT
Separation of specification and execution	yes	yes	+/-
Formal model of the specification available	no[1]	yes	no
Implementation separated from the underlying execution environment	yes	no	no
Named variables	yes	no	no
Dependency computations	yes	no	no
On demand evaluations	yes	no	yes
Provides a strict methodology for task specification	no[2]	yes	no
Feature variables	yes	yes	no
Explicit loop closure	yes	no	no
Explicit time variable	yes	no	yes
Maturity	new	high	high
Built-in inequalities	yes	+/-	yes
Priority levels	≤ 2	$\nearrow$	$\nearrow$

1: no model is available but it is feasible to provide a formal model.
2: the iTaSC procedure can still be followed in eTaSL, it is however not enforced.

demonstrator. Execution time for one sample period is around 5 ms on a standard laptop. The included video shows simulation results for the demonstrator. Fig. 4a shows both the nominal trajectory and the trajectory of the robot end effectors, for both the left and the right manipulator. The X- and Y-axes correspond to fig. 2. The left and right end effectors start form outside the trajectories. There is a negligible tracking error after the end effectors have reached the trajectories. Fig. 4b shows the path velocity for the trajectory of the left end effector. The path velocity is 0.2 m/s, except when there is a danger for collision, at which time the path velocity decreases. Around time 6s, the left end effector is even pushed back along its trajectory in order to avoid the right end effector.

VI. DISCUSSION

Table II summarizes the main differences between the specification language and implementation eTaSL/eTC, and the existing frameworks iTaSC [6] and SoT[8].

Both eTaSL and iTaSC provide a separation of specification and execution. On the other hand, SoT uses a graph of entities to specify tasks, but this model is intermixed with execution aspects. SoT has its own execution environment that defines a way to define blocks and their interaction. The controller implementation of iTaSC is thoroughly intermixed with the Orocos environment [9] eTaSL carefully separates its functionality from the underlying execution environment. All frameworks can be used inside a ROS environment.

The iTaSC DSL [7] provides a formal model of a task (using UML 2.0/Ecore). This DSL can be translated into a corresponding implementation. Both the iTaSC DSL and the iTaSC implementation follow a strict methodology that describes a constraint using a six-degree-of-freedom VKC. iTaSC's DSL explicitly models robots, VKCs and objects in the environment. eTaSL starts from a lower level: the language itself does not know about robots, objects and other elements in the task. However, as is also shown in the demonstrator, models such as URDF [16] and COLLADA [17], can easily be imported into the task specification language and translated into lower-level expression graphs and constraints. In combination with Lua language constructs such as subroutines and tables, libraries with higher-level constructs are defined inside eTaSL (see also fig. 1). In contrast to iTaSC DSL, these descriptions also contain constraints on the joint limits and information for collision avoidance. More flexible and more composable descriptions of tasks and robots are achieved: tasks, robots, and other concepts can be abstracted and parameterized in values and expression graphs. There can also be libraries that only provide expressions for use in constraints (cf. table I). Other data structures can be passed around, such as a list of robot links to be used as parameters for a collision avoidance constraint.

In eTaSL, monitors, constraints and variables are always accessed by name. At execution time, variable dependencies will be tracked and only the necessary variables will be involved in the optimization problem. Consequently, a task programmer does not need to be bothered with the internal details of a robot or task description. In eTaSL, it is easy to define new types of variables. This will make it easier to develop extensions such as defining specifications for the automatic generation of estimators. Both eTaSL and SoT use on-demand evaluation to avoid unnecessary computation when certain parts in the evaluation are not changed.

As is shown in the demonstrator example, feature variables can make the formulation of complex constraints easier. They allow specifying additional freedom, in contrast to constraints that by themself only specify limitations. Both eTaSL and iTaSC offer feature variables. It is possible to follow the iTaSC methodology in eTaSL: a VKC can be defined in function of feature variables and an explicit loop closure equation can be defined. Multiple VKCs can be used

to describe a complete task.

However, iTaSC always uses a VKC with (exactly) six feature variables and one loop closure constraint to formulate tasks. Such a VKC will always show singularities for certain values of the feature variables. Furthermore, in a non-singular position, there are up to 16 different solutions (configurations) for the VKCs[18]. Because of these artificially introduced singularities and configurations, it can become more difficult to obtain a robust and encapsulated task specification. eTaSL simplifies these problems because there are no implicitly defined loop closure equations and no need to use always six feature variables. For example, the use case in fig. 3 also has a singular configuration ($\chi_f = 0$), and two different solutions to its "inverse kinematics". This can however be easily avoided by imposing an additional constraint $\chi_f > 0$. In this case, such an additional constraint will not limit the usability of the task. iTaSC can impose similar additional constraints. However, in iTaSC, one is *forced* to use six feature variables. For robust tasks, additional constraints are always necessary and sometimes these additional constraints will limit the usability of the task.

The current numerical solver of eTaSL is based on a mature solver for QP-optimization problems [11]. It can only handle two levels of priority. Both iTaSC and SoT are not limited in the number of priority levels. SoT has a mature solver that is specifically designed to deal with a higher number of priority levels in a numerical accurate way. All frameworks can handle inequalities; however, inequality constraints are only approximately handled in iTaSC by activating equality constraints gradually in the neighborhood of the limits.

The expression graphs in eTaSL allow a decomposition with a finer granularity than is possible in iTaSC and SoT, without imposing an additional burden on the task programmer. In SoT, it is necessary to implement the computation of Jacobians for each new type of task; in eTaSL, this is handled using automatic differentiation.

VII. CONCLUSION AND FUTURE RESEARCH

This paper discussed a new framework for the specification of robot task controllers. This framework is built according to an architecture that carefully separates the specification of a task (eTaSL), the translations of this specification into an optimization problem (*solver*), the computation of the solution for this optimization problem (*numerical solver*), and the underlying execution environment.

A powerful task specification language is defined based on Lua. This language contains a series of concepts that facilitate the definition of complex tasks in a flexible and composable way, such as expression graphs, feature variables, monitors, constraints, explicit naming of variables and an explicit time variable.

Typical usage patterns of the framework are explained, such as the separation of robot, task and environment, background constraints, and dealing with geometric relations. This is demonstrated on an example of task for a kinematically redundant, bi-manual, task on a PR2 robot.

A comparison with the existing frameworks iTaSC and SoT reveals the benefits of the new framework concerning the separation of specification and execution, abstraction, composability, efficiency, and the ability to specify complex tasks.

Future research will focus on extending the framework with a specification language for the automatic generation of estimators. Inspired by [6], this combination of task specification and estimation leads to self-calibrating tasks and learnable skills that combine model-based task specification with learning from examples.

The software implementation will be available at [19].

REFERENCES

[1] M. Klotzbučher and H. Bruyninckx, "Coordinating robotic tasks and systems with rfsm statecharts," *JOSER: Journal of Software Engineering for Robotics*, vol. 3, pp. 28–56, 2010.

[2] J. Bohren and S. Cousins, "The smach high-level executive," *IEEE Robot. Automat. Mag.*, vol. 17, pp. 18–20, 2010.

[3] H. Nguyen, M. Ciocarlie, J. Hsiao, and C. C. Kemp, "Ros commander (rosco): Behavior creation for home robots," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[4] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Trends in Distributed Systems. CORBA and Beyond.* Springer-Verlag, 1996, pp. 162–176.

[5] C. Samson, M. Le Borgne, and B. Espiau, *Robot Control, the Task Function Approach*, ser. Combinatorial Scientific Computing. Clarendon Press, 1991.

[6] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *The International Journal of Robotics Research*, vol. 26, no. 5, pp. 433–455, 2007.

[7] D. Vanthienen, M. Klotzbuucher, J. De Schutter, T. De Laet, and H. Bruyninckx, "Rapid application development of constrained-based task modelling and execution using domain specific languages," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013, pp. 1860–1866.

[8] N. Mansard, O. Stasse, P. Evrard, and A. Kheddar, "A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The stack of tasks," in *Proceedings of the 2009 International Conference on Advanced Robotics*, Munich, Germany, 2009.

[9] H. Bruyninckx and P. Soetens. (2001) Open RObot COntrol Software (Orocos). Last visited February 2014. [Online]. Available: http://www.orocos.org/

[10] ROS. Last visited February 2014. [Online]. Available: http://ros.org

[11] H. Ferreau, "qpoases - an open source implementation of the online active set strategy for fast model predictive control." in *Proceedings of the Workshop on Nonlinear Model Based Control Software and Applications*, Loughborough, 2007.

[12] Lua. Last visited February 2014. [Online]. Available: http://lua.org

[13] L. B. Rall, *Automatic Differentiation: Techniques and Applications*, ser. Lecture Notes in Computer Science. Springer, 1981, no. 120.

[14] C. Wampler, "Manipulator inverse kinematic solutions based on vector formulations and damped least squares methods," *IEEE Trans. Syst., Man, Cybern.*, pp. 93–101, 1986.

[15] A. Dietrich, T. Wimbock, H. Taubig, A. Albu-Schaffer, and G. Hirzinger, "Extensions to reactive self-collision avoidance for torque and position controlled humanoids," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011, pp. 3455–3462.

[16] Willow Garage, "Universal Robot Description Format (URDF)," http://www.ros.org/wiki/urdf, 2009.

[17] M. Barnes and E. L. Finch, "COLLADA—Digital Asset Schema Release 1.5.0," http://www.collada.org, 2008, last visited August 2013.

[18] H.-Y. Lee and C.-G. Liang, "A new vector theory for the analysis of spatial mechanisms," *Mechanism and Machine Theory*, vol. 23, no. 3, pp. 209–217, 1988.

[19] eTaSL. Last visited June 2014. [Online]. Available: http://people.mech.kuleuven.be/ eaertbel/etasl/