International Journal of Humanoid Robotics
Vol. 13 (2015) 1550040 (57 pages)

DOI: 10.1142/S0219843615500401

World Scientific
www.worldscientific.com

ControlIt! — A Software Framework for Whole-Body Operational Space Control

Chien-Liang Fok[*,†,§], Gwendolyn Johnson[*,¶], Luis Sentis[*,‖], Aloysius Mok[‡,**] and John D. Yamokoski[†,††]

[*]*Human Centered Robotics Lab, Mechanical Engineering, University of Texas at Austin, 204 East Dean Keeton Street Austin, TX 78712, USA*

[†]*NASA Johnson Space Center 2101 NASA Road 1 Houston, TX 77058, USA*

[‡]*UT Real-Time Systems Group, Computer Science University of Texas at Austin, 2317 Speedway Stop D9500 Austin, TX 78712, USA*

[§]*liangfok@gmail.com*
[¶]*gwendolynbrook@gmail.com*
[‖]*lsentis@austin.utexas.edu*
[**]*mok@cs.utexas.edu*
[††]*john.d.yamokoski@nasa.gov*

Received 19 January 2015
Accepted 30 July 2015
Published 6 October 2015

Whole Body Operational Space Control (WBOSC) enables floating-base highly redundant robots to achieve unified motion/force control of one or more operational space objectives while adhering to physical constraints. It is a pioneering algorithm in the field of human-centered Whole-Body Control (WBC). Although there are extensive studies on the algorithms and theory behind WBOSC, limited studies exist on the software architecture and APIs that enable WBOSC to perform and be integrated into a larger system. In this paper, we address this by presenting ControlIt!, a new open-source software framework for WBOSC. Unlike previous implementations, ControlIt! is multi-threaded to increase maximum servo frequencies using standard PC hardware. A new parameter binding mechanism enables tight integration between ControlIt! and external processes via an extensible set of transport protocols. To support a new robot, only two plugins and a URDF model is needed — the rest of ControlIt! remains unchanged. New WBC primitives can be added by writing `Task` or `Constraint` plugins. ControlIt!'s capabilities are demonstrated on Dreamer, a 16-DOF torque controlled humanoid upper body robot containing both series elastic and co-actuated joints, and using it to perform a product disassembly task. Using this testbed, we show that ControlIt! can achieve average servo latencies of about 0.5 ms when configured with two Cartesian position tasks, two orientation tasks, and a lower priority posture task. This is 10 times faster than the 5 ms that was achieved using UTA-WBC, the prototype implementation of WBOSC that is both application and platform-specific. Variations in the product's position is handled by updating the goal of the Cartesian position task. ControlIt!'s source code is released under LGPL and we hope it will be adopted and maintained by the WBC community for the long term as a platform for WBC development and integration.

Keywords: Software framework; whole-body control; whole-body operational space control; upperbody humanoid robot.

1. Introduction

Whole-Body Control (WBC) takes a holistic view of multi-branched highly redundant robots like humanoids to achieve general coordinated behaviors. One of the first WBC algorithms is Whole Body Operational Space Control (WBOSC),[1–4] which provides the theoretical foundations for achieving operational space inverse dynamics, task prioritization, free floating degrees of freedom, contact constraints, and internal forces. There is now a growing community of researchers in this field as exemplified by the recent formation of an IEEE technical committee on WBC.[5] While the foundational theory and algorithms behind WBC have made great strides, less progress exists in software support limiting the use of WBC today. In this paper, we remedy this problem by presenting ControlIt!,[a] an open source[b] software framework for WBOSC.

In this paper, we introduce ControlIt!, a software framework that enables WBOSC controllers to be instantiated and is designed for systems integration, extensibility, high performance, and use by both WBC researchers and the general public. Instantiating a WBOSC controller consists of defining a prioritized compound task that specifies the operational space objectives and underlying goal postures that the controller should achieve, and a constraint set that specifies the natural physical constraints of the robot. Systems integration is achieved through a parameter binding mechanism that enables external processes to access WBOSC parameters through various transport protocols, and a set of introspection tools for gaining insight into the controller's state at runtime. ControlIt! is extensible through plugins that enable the addition of new WBC primitives and support for new robot platforms. High performance is achieved by using state-of-the-art software libraries and multiple threads that enable ControlIt! to offer higher servo frequencies relative to previous WBOSC implementations. By making ControlIt! open source and maintaining a centralized website (`https://robotcontrolit.com`) with detailed documentation, installation instructions, and tutorials, ControlIt! can be used to evaluate new WBC ideas and supported long term.

The key contributions of this paper are as follows:

(1) We design a software architecture for supporting general use of WBOSC and its integration within a larger system via parameter binding and events.
(2) We introduce the first API based on WBOSC principles for use across general applications and robots.

[a] ControlIt! should not be associated with MoveIt!.[6] ControlIt! is focused on whole body feedback control whereas MoveIt! is focused on motion planning. Thus, MoveIt! and ControlIt! typically reside at different levels of the software stack. The default feedback controller used by MoveIt! is `ros_control`.[7] However, MoveIt! could be configured to work with ControlIt!.

[b] ControlIt!'s source code is available under a LGPLv2.1 license. Download and usage instructions are available at `https://robotcontrolit.com`.

(3) We provide an open-source software implementation.
(4) We design and implement a high performance multi-threaded architecture that increases the achievable servo frequency by 10X relative to previous implementations of WBOSC.
(5) We reduce the number of components that need to be modified to develop a new behavior to the set of `RobotInterface`, `Clock`, `CompoundTask`, and `ConstraintSet` and decouple these changes from core ControlIt! code via plugins.
(6) We demonstrate ControlIt!'s utility and performance using a humanoid robot executing a product disassembly task.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of WBOSC's mathematical foundations. Section 4 presents ControlIt!'s software architecture and APIs. Section 5 presents how ControlIt! was integrated with Dreamer and used to develop a product disassembly application. Section 6 contains a discussion on other experiences using ControlIt! and future research directions. The paper ends with conclusions in Sec. 7.

2. Related Work

As a relatively new field in robotics, WBC is rapidly evolving. Most WBC algorithms issue torque commands[8–26] or position commands.[27] They differ in whether they are centralized[28,29] or distributed,[30,31] focus on manipulation,[32] locomotion,[33–35] or behavior sequencing,[36,37] the underlying control models used,[38–40], and whether they have been evaluated in simulation or on hardware.[41–69] These efforts demonstrate the behaviors enabled by WBC such as the use of compliance, multi-contact postures, robot dynamics, and joint redundancy to balance multiple competing objectives. ControlIt! is currently focused on supporting general use of WBOSC and its capabilities, but may be enhanced to include ideas and capabilities from these recent WBC developments.

An implementation of WBOSC called Stanford-WBC[70] was released in 2011. Stanford-WBC includes mechanisms for parameter reflection, data logging, and script-based configuration, but was a limited implementation of WBOSC that did not support branched robots, mobile robots, or contact constraints. It was used to make Dreamer's right arm wave and shake hands. More recently, UTA-WBC extended Stanford-WBC to support the full WBOSC algorithm, which includes branched robots, free floating degrees of freedom, contact constraints, and a more accurate robot model that includes rotor inertias.[71] UTA-WBC was used to make a wheeled version of Dreamer containing 13 DOFs maintain balance on rough terrain. While this demonstrated the feasibility of WBOSC using a real humanoid robot, UTA-WBC was a research prototype targeted for a specific robot and specific behavior, i.e., balancing.[29] The implementation was not designed to work as part of a larger system for general applications. Instead, ControlIt! is a complete software redesign and re-implementation of the WBOSC algorithm with a focus on the software constructs and APIs that facilitate the integration of WBOSC into larger systems.

Table 1. A comparison between UTA-WBC and ControlIt!.

Property	UTA-WBC	ControlIt!
OS	Ubuntu 10.04	Ubuntu 12.04 and 14.04
ROS Integration	ROS Fuerte	ROS Hydro and Indigo
Linear Algebra Library	Eigen 2	Eigen 3
Model Library	Tao	RBDL 2.3.2
Model Description Format	Proprietary XML	URDF
Integration (higher levels)	N/A	Parameter binding
Integration (lower levels)	Proprietary	`RobotInterface` and `Clock` plugins
Controller Introspection	Parameter reflection	Parameter reflection and ROS services
WBC Initial Configuration	YAML	YAML and ROS parameter server
WBC Reconfiguration	N/A	Enable/disable tasks and constraints, update task priority levels
Key Abstractions	Task, constraint, skill	Compound task, constraint set
Task/Constraint Libraries	Statically coded	Dynamically loadable via ROS pluginlib
Number of threads	1	3
Simulator	Proprietary	Gazebo 6.1
Website	`https://github.com/lsentis/uta-wbc-dreamer`	`https://robotcontrolit.com`

The differences between UTA-WBC and ControlIt! are shown in Table 1. Compared with UTA-WBC and Stanford-WBC, ControlIt! is a complete re-implementation that replaces the previous implementation. Specifically, ControlIt! contains new and more expressive software abstractions that enable arbitrarily complex WBOSC controllers to be configured, works with newer software libraries, middleware, and simulators, supports extensibility through a plugin-based architecture, is multi-threaded, and is designed to easily integrate with external processes through parameter binding and controller introspection mechanisms.

The ability to integrate with external processes is important because applications of branched highly redundant robots of the type targeted by WBC are typically very sophisticated involving many layers of software both above and below the whole body controller. To handle such complexity, a distributed component-based software architecture is typically used where the application consists of numerous independently running software processes or threads that communicate over both synchronous and asynchronous channels.[72,73] The importance of distributed component-based software for advanced robotics is illustrated by the number of recently developed middleware frameworks that provide it. They include OpenHRP,[74,75] RT-Middleware,[76] Orocos Toolchain,[77] YARP,[78] ROS,[79,80] CLARAty,[81,82] aRD,[83] Microblx,[84,85] OpenRDK,[86–88] and ERSP.[89] Among these, ControlIt! is currently integrated with ROS and is a ROS node within a ROS network, though usually as a real-time process potentially within another component-based framework (i.e., ControlIt!'s servo thread was an Orocos real-time task during the DRC Trials, and is

a RTAI[90] real-time process in the Dreamer experiments discussed in this paper). In general, ControlIt! can be modified to be a component within any of the other aforementioned component-based robot middleware frameworks.

ControlIt! is designed to interact with components both below (i.e., closer to the hardware) and above (i.e., closer to the end user or application) it within a robotic system. Components below ControlIt! include robot hardware drivers or resource allocators like `ros_control`[7,91] and Conman[92] that manage how a robot's joints are distributed among multiple controllers within the system. This is necessary since multiple WBC controllers may coexist and a manager is needed to ensure only one is active at a time. In addition, joints in a robots' extremity like those in an end effector usually have separate dedicated controllers. Components that may reside above ControlIt! include task specification frameworks like iTaSC,[93–95] planners like MoveIt!,[6] management tools like Rock,[96] MARCO,[97] and G$^{\text{en}}$oM,[98] behavior sequencing frameworks like Ecto[99] and Robot Task Commander (RTC),[100] and other frameworks for achieving machine autonomy[101–109] or the coordination of multiple humanoids.[110] Clearly, the set of components that ControlIt! interacts with is large, dynamic, and application-dependent. This is possible since component-based architectures provide sufficient decoupling to allow these external components to change without requiring ControlIt! to be modified.

3. Overview of WBOSC

This section provides a brief overview of WBOSC. Details are provided in previous publications.[2–4,29] Let n_{joints} be the number of actual DOFs in the robot. The robot's joint state is represented by the vector q_{actual} as shown by the following equation.

$$q_{\text{actual}} = \langle q_1 \dots q_{n_{\text{joints}}} \rangle. \tag{1}$$

The robot's global pose is represented by a six-dimensional floating virtual joint that connects the robot's base link to the world, i.e., three rotational and three prismatic virtual joints. It is denoted by vector $q_{\text{base}} \in \mathbb{R}^6$. The two partial state vectors, q_{actual} and q_{base}, are concatenated into a single state vector $q_{\text{full}} = q_{\text{actual}} \cup q_{\text{base}}$. This combination of real and virtual joints into a single vector is called the *generalized* joint state vector. Let n_{dofs} be the number real and virtual DOFs in the model that is used by WBOSC. Thus, $q_{\text{full}} \in \mathbb{R}^{6+n_{\text{joints}}} = \mathbb{R}^{n_{\text{dofs}}}$. The total state that is provided to the whole body controller consists of the full joint position vector q_{full} and the full joint velocity vector $\dot{q}_{\text{full}}$.

The underactuation matrix $U \in \mathbb{R}^{n_{\text{joints}} \times n_{\text{dofs}}}$ defines the relationship between the actuated joint vector and the full joint state vector as shown by the following equation.

$$q_{\text{actual}} = U q_{\text{full}}. \tag{2}$$

Let A be the robot's generalized joint space inertia matrix, B be the generalized joint space Coriolis and centrifugal force vector, G be the generalized joint space gravity force vector, J_c be the contact Jacobian matrix that maps from generalized

joint velocity to the velocity of the constraint space dimensions, λ_c be the co-state of the constraint space reaction forces, and τ_{command} be the desired force/torque joint command vector that is sent to the robot's joint-level controllers. The robot dynamics can be described by a single linear second-order differential equation shown by the following equation.

$$A\begin{pmatrix} \ddot{q}_{\text{base}} \\ \ddot{q}_{\text{actual}} \end{pmatrix} + B + G + J_c^{\text{T}}\lambda_c = \begin{pmatrix} 0_{6\times 1} \\ \tau_{\text{command}} \end{pmatrix}. \tag{3}$$

Constraints are formulated as follows. Let $\dot{p}_c$ be the velocity of the constrained dimensions, which we approximate as being completely rigid and therefore yielding zero velocity at the contact points, as shown by the following equation.

$$\dot{p}_c = J_c\begin{pmatrix} \dot{q}_{\text{base}} \\ \dot{q}_{\text{actual}} \end{pmatrix} \triangleq 0. \tag{4}$$

Tasks are formulated as follows. Let $\dot{p}_t$ be the desired velocity of the task, J_t be the Jacobian matrix of task t that maps from generalized joint velocity to the velocity of the task space dimensions, and N_c be the generalized null-space of the constraint set. Furthermore, let J_t^* be the contact consistent reduced Jacobian matrix[3] of task t, i.e., it is consistent with U and N_c. The definition of $\dot{p}_t$ is given by the following equation where operator $\overline{arg}$ is the dynamically consistent generalized inverse of arg.[3]

$$\begin{aligned} \dot{p}_t &= J_t\begin{pmatrix} \dot{q}_{\text{base}} \\ \dot{q}_{\text{actual}} \end{pmatrix} = J_t\overline{UN_c}\,\dot{q}_{\text{actual}} \\ &= J_t^*\,\dot{q}_{\text{actual}} \end{aligned} \tag{5}$$

Let Λ_t^* be the contact-consistent prioritized task-space inertia matrix[3] for task t, $\ddot{p}_{t,\text{ref}}$ be the reference, i.e., desired, task-space acceleration for task t, β_t^* be the contact-consistent task-space Coriolis and centrifugal force vector for task t, and γ_t^* be the contact-consistent task space gravity force vector for task t. The force/torque command of task t, denoted F_t, is given by the following equation.

$$F_t = \Lambda_t^*\,\ddot{p}_{t,\text{ref}} + \beta_t^* + \gamma_t^*. \tag{6}$$

To achieve multi-priority control, let $J_{t|\,\text{prev}}^*$ be the Jacobian matrix of task t that is consistent with U, N_c, and all higher priority tasks. The equation for τ_{command} is the sum of all of the individual task commands multiplied by the corresponding $J_{t|\,\text{prev}}^*$ matrix as shown by the following equation.

$$\tau_{\text{command}} = \sum_t J_{t|\,\text{prev}}^{*\text{T}} F_t. \tag{7}$$

Finally, when a robot has more than one point of contact with the environment, there are internal tensions within the robot. By definition, these "internal forces" are orthogonal to joint accelerations since they result in no net movement of the robot.

The control structures like the multicontact/grasp matrix that are used to control these internal forces are documented in previous publications.[4] Let L^* be the null-space of (UN_c) and τ_{internal} be the reference (i.e., desired) internal forces vector. The contribution of the internal forces can thus be added to Eq. (7) as shown by the following equation.

$$\tau_{\text{command}} = \sum_t (J^{*\text{T}}_{t|\text{prev}} F_t) + L^{*\text{T}} \tau_{\text{internal}}. \tag{8}$$

τ_{command} is sent to the joint-level controllers concluding one cycle of the servo loop. This concludes the mathematical discussion of WBOSC. The next section focuses on the software architecture and abstractions for implementing WBOSC.

4. ControlIt! Software Architecture

There are six guiding principles behind ControlIt!'s development: (i) separate concerns into interface definitions, implementations, and configuration, (ii) support extensibility and platform-independence through dynamically loadable plugins, (iii) encourage code reuse through plugin libraries, (iv) support systems integration through parameter binding, events, data introspection services, and compatibility with a modern software ecosystem, (v) be cognizant of performance and real-time considerations, and (vi) support two types of end users: developers who use ControlIt! and researchers who modify ControlIt!.

Section 4.1 contains a discussion of ControlIt!'s software architecture, which describes the software components within ControlIt's core. Many of these components either instantiate plugins or are implemented by plugins based on ROS pluginlib.[111] The use of plugins enables ControlIt! to support different robots and applications. Section 4.2 discusses mechanisms for configuring and integrating ControlIt! into a larger system. This includes the parameter reflection, binding, and event signaling mechanisms, and YAML specification files. Finally, a description of ControlIt!'s multi-threaded architecture is discussed in Sec. 4.3.

4.1. *Software architecture*

The software abstractions that enable ControlIt! to instantiate and integrate general WBOSC controllers are shown in Fig. 1. The abstractions that are extensible via plugins are colored gray. They include tasks, constraints, the whole body controller, the clock, and the robot interface. Nonextensible components include the compound task, robot model, constraint set, and coordinator. The coordinator implements the servo loop and uses all of the other abstractions except for the clock, which implements the servo thread and controls when the coordinator executes the next cycle of the servo loop. The software abstractions can be divided into three general categories: configuration, WBC, and hardware abstraction.

Configuration. Configuration software abstractions include the robot model, compound task, and constraint set. Their APIs and attributes are shown in Fig. 2.

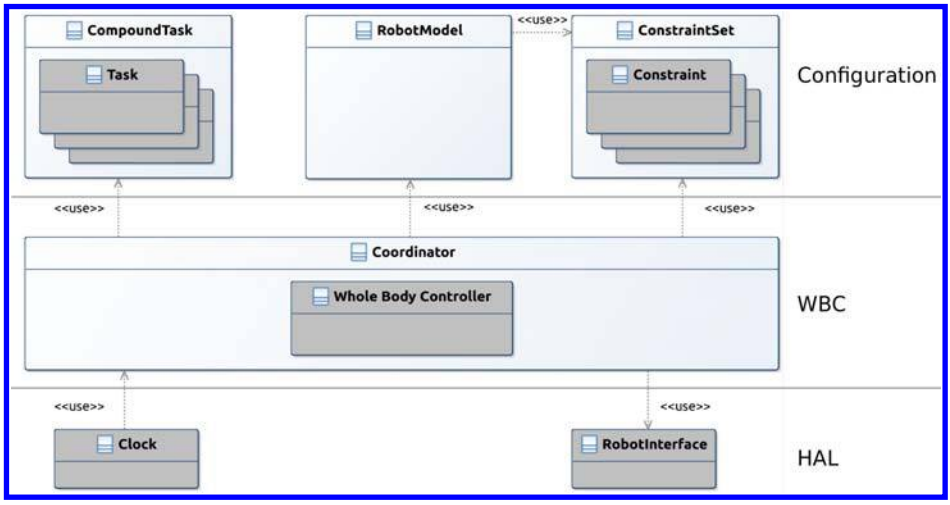


Fig. 1. The primary software abstractions within ControlIt! fall into three categories: configuration, WBC, and Hardware Abstraction Layer (HAL). Configuration components consist of a compound task, constraint set, and robot model. The compound task contains a set of prioritized tasks. Tasks specify operational space or postural objectives and contain task-space controllers; multiple tasks may have the same priority level. Constraints specify natural physical constraints that must be satisfied at all times and are effectively higher priority than the tasks. The robot model computes kinematic and dynamic properties of the robot based on the current joint states. WBC components include a coordinator that ties all of the other components together and implements the actual servo loop and the whole body controller itself. HAL components include a clock and robot interface. They enable ControlIt! to work on many platforms. Arrows indicate usage relationships between the software abstractions. The abstractions that are extensible via plugins are colored gray.

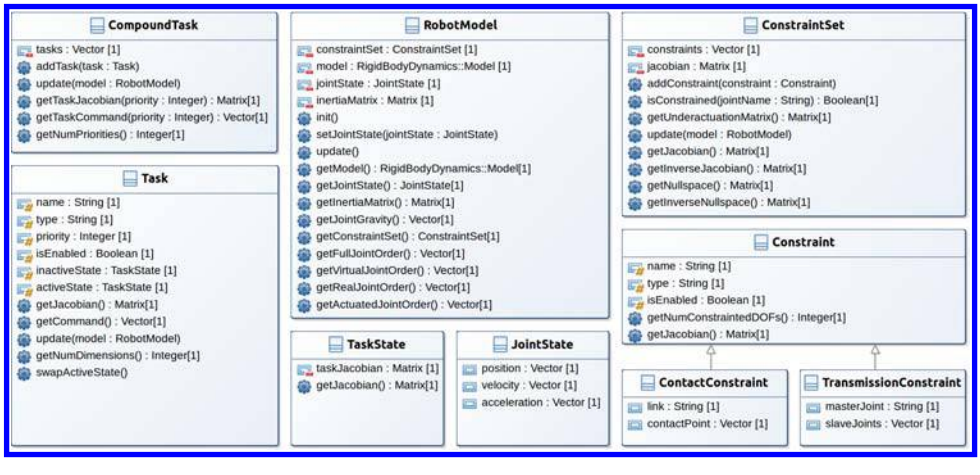


Fig. 2. This UML diagram specifies the APIs of ControlIt!'s configuration software abstractions. They are used to specify the objectives and constraints of the whole body controller.

The robot model determines the kinematic and dynamic properties of the robot and builds upon the model provided by the Rigid Body Dynamics Library (RBDL),[112] which includes algorithms for computing forward and inverse kinematics and dynamics and frame transformations. The kinematic and dynamic values provided by the model are only estimates and may be incorrect, necessitating the use of a whole body feedback controller. The robot model API includes methods for saving and obtaining the joint state, q_{full}, and getting properties of the robot like the joint space inertia matrix, Coriolis and centrifugal force vector, and gravity compensation vector, which are variables A, B, and G in Eq. (3). There are also methods for obtaining the joint order within the whole body controller. A reference to the constraint set is kept within the robot model to determine which joints are virtual (i.e., the 6-DOF free floating joints that specify a mobile robot's position and orientation within the world frame), real, and actuated.

The compound task and constraint set contain lists of tasks and constraints, respectively. Tasks and constraints are abstract; concrete implementations are added to ControlIt! through plugins. Both have names and types for easy identification and can be enabled or disabled based on context. A task represents an operational or postural objective for the whole body controller to achieve. Concrete task implementations contain goal parameters that, in combination with the robot model, produce an error. The error is used by a controller inside the task to generate a task-space effort command,[c] which is accessible through the `getCommand()` method and may be in units of force or torque. In addition to the command, a task also provides a Jacobian that maps from task space to joint space. The compound task combines the commands and Jacobians of the enabled tasks and relays this information to the whole body controller. Specifically, for each priority level, the compound task vertically concatenates the Jacobians and commands belonging to the tasks at the priority level. The WBOSC algorithm uses these concatenated Jacobian and command matrices to support task prioritization and multiple tasks at the same priority level, as defined by Eq. (7).

Task Library. To encourage code reuse and enable support for basic applications, ControlIt! comes with a task library containing commonly used tasks. The tasks within this library are shown in Fig. 3. There are currently six tasks in the library: joint position, 2D/3D Orientation, Center of Mass (COM), Cartesian position, and Center of Pressure (COP). In the future, more tasks can be added to the library by introducing additional plugins. Of these, the joint position, orientation, and Cartesian position tasks have been successfully tested in hardware. The rest have only been tested in simulation. Note that all of the tasks make use of a PIDController. This feedback controller generates the task-space command based on the current error and gains. Alternative types of controllers like sliding mode control may be provided in the future.

The joint position task directly specifies the goal positions, velocities, and accelerations of every joint in the robot. It typically defines the desired "posture" of the

[c]We use the word "effort" to denote generalized force, i.e., force or torque.

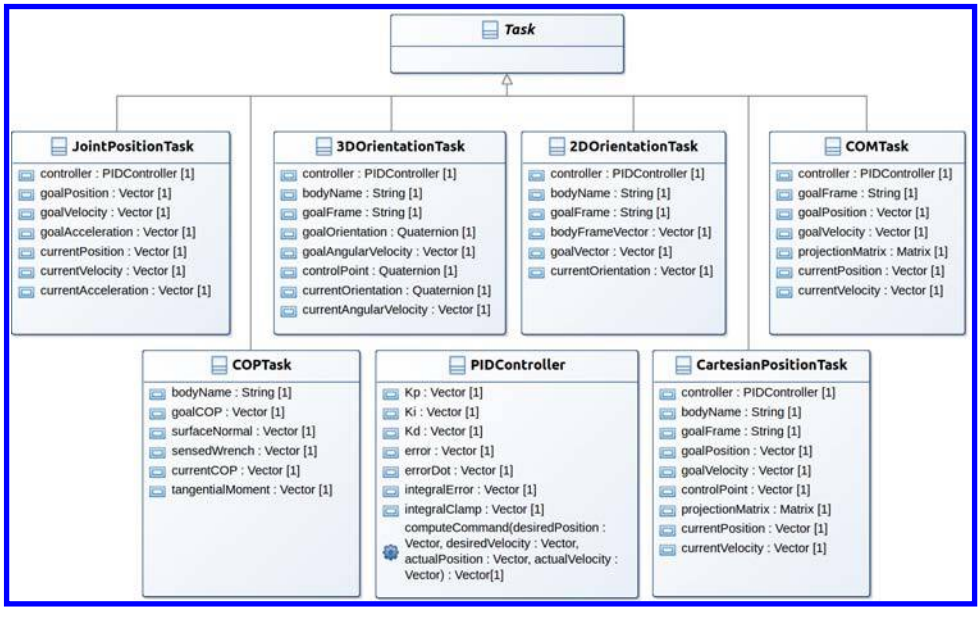


Fig. 3. This UML class diagram shows the tasks in ControlIt!'s task library and the PID controller that they use. Combinations of these tasks specify the operational space and postural objectives of the whole body controller and collectively form the compound task. Concrete tasks are implemented as dynamically loadable plugins. ControlIt! can be easily extended with new tasks via the plugin mechanism.

robot, which is not an operational objective but accounts for situations where there is sufficient redundancy within the robot to result in non-deterministic behavior when no posture is defined. Specifically, a posture task is necessary when the null space of all higher priority tasks and constraints is not nil, and the best practice is to always include one as the lowest priority task in the compound task. The joint position task has an input parameter called `goalAcceleration` to enable smooth transitions between joint positions. The goal acceleration is a desired acceleration that is added as a feedforward command to the control law. The `currentAcceleration` output parameter is a copy of the `goalAcceleration` parameter and is used for debugging purposes.

The 2D and 3D orientation tasks are used to control the orientation of a link on the robot. They differ in terms of how the orientations are specified. Whereas the 2D orientation is specified by a vector in the frame of the body being oriented, the 3D orientation is specified using a quaternion. The purpose of providing a 2D orientation task even though a 3D orientation could be used is to reduce computational overhead when only two degrees of orientation control is required. For example, a 2D orientation task is used to control the heading of Trikey, a three wheeled holonomic mobile robot, as shown in Fig. C.1, whereas a 3D orientation task is used to control the orientation of Dreamer's end effectors, as shown in Fig. C.2(b). Visualizations of these two task-level controllers are given in Appendix C. The 2D orientation task does not include a `goalAngularVelocity` input parameter because its current

implementation assumes the goal velocity is always zero. This assumption can be easily removed in the future by modifying the control law to include a non-zero goal velocity.

The COM task controls the location of the robot's COM, which is derived from the robot model. It is useful when balancing since it can ensure that the robot's configuration always results in the COM being above the convex polygon surrounding the supports holding up the robot. The COP task controls the center of pressure of a link that is in contact with the ground. It is particularly useful for biped robots containing feet since it can help ensure that the COP of a foot remains within the boundaries of the foot thereby preventing the foot from rolling. The Cartesian position task controls the operational space location of a point on the robot. Typically, this means the location of an end effector in a frame that is specified by the user and is by default the world frame. For example, it is used to position Dreamer's end effectors in front of Dreamer as shown in Fig. C.2. As indicated by the figure, multiple Cartesian position tasks may exist within a compound task, as long as they control different points on the robot.

As previously mentioned, the aforementioned tasks are those that are currently included with ControlIt!. They are implemented as plugins that are dynamically loaded on-demand during the controller configuration process. Additional tasks may be added in the future. For example, an external force task may be added that controls a robot to assert a certain amount of force against an external obstacle. In addition, an internal force task may be added to control the internal tensions between multiple contact points. A prototype of such a task was successfully used during NASA Johnson Space Center (JSC) DARPA Robotics Challenge (DRC) critical design review[d] to make Valkyrie walk in simulation, but is not included in the current task library due to the need for additional testing and refinement. For the walking behavior, ControlIt!'s compound task included a COM Task, internal tensions task, posture task, and, for each foot, a COP, Cartesian position, and orientation task.

Constraints. A constraint specifies natural physical limits of the robot. There are two types of constraints: `ContactConstraint` and `TransmissionConstraint`. Contact constraints specify places where a robot touches the environment. Transmission constraints specify dependences between joints, like when they are co-actuated. The parent `Constraint` class includes methods for obtaining the number of DOFs that are constrained and the Jacobian of the constraint, J_c, as used in Eqs. (3) and (4). Contact constraints have a `getJoint()` method that specifies the parent joint of the constrained link. Transmission constraints have a master joint that is actuated and a set of slave joints that are co-actuated with the master joint. Unlike tasks, constraints do not have commands since they simply specify the

[d]As a Track A DRC team, NASA JSC was required to undergo a critical design review by DARPA officials in June 2013, which was in the middle of the period leading up to the DRC Trials in December 2013. The results of the review determined whether the team would continue to receive funding and proceed to compete in the DRC Trials as a Track A team. NASA JSC was one of six Track A teams to pass this critical design review.

nullspace within which all tasks must operate. Like the compound task, the constraint set computes a Jacobian that is the vertical concatenation of all the constraint Jacobians. In addition, it provides an update method that computes both the null space projector and $\overline{UN_c}$ (defined in Eq. (5)), accessors for these matrices, and methods for determining whether a particular joint is constrained. The whole body controller uses this information to ensure all of the constraints are met. While it is true that contact constraints are mathematically similar to tasks without an error term, we wanted to distinguish between the two since they serve significantly different purposes: tasks denote a user's control objectives while constraints denote a robot's physical limits. We do not want to confuse the API by using the same software abstraction for both purposes. Furthermore, by separating tasks and constraints, the API will be easier to extend to support optimization-based controllers with inequality constraints.

Constraint Library. Constraints included in ControlIt!'s constraint library are shown in Fig. 4. Contact constraints include the flat contact constraint, omni wheel contact constraint, and point contact constraint. The flat contact constraint restricts both link translation and rotation. The omni wheel contact constraint restricts one rotational DOF and one translational DOF based on the current orientation of the wheel. Point contact constraint restricts just link translation. One transmission constraint called `CoactuationConstraint` is provided that enables ControlIt! to handle robots with two co-actuated joints, like Dreamer's torso pitch joints. It includes a transmission ratio specification to handle situations where the relationship between the master joint and slave joint is not one-to-one. Currently only the two-joint co-actuation case is supported, though a more generalized constraint that supports more than two co-actuated joints could be trivially added in the future. Specifically, another child class of `TransmissionConstraint` can be added as a plugin to support the co-actuation of more than two joints by adding more rows to

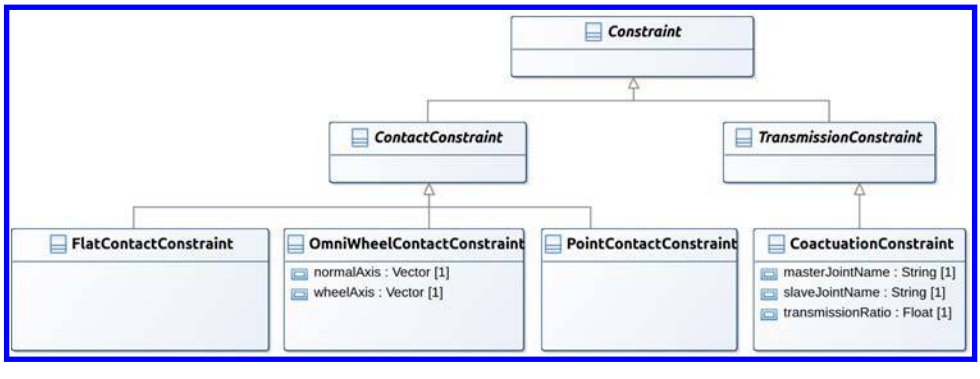


Fig. 4. This UML class diagram shows the constraints in ControlIt!'s constraint library. Combinations of these constraints specify natural physical limits of the robot and constitute the constraint set. Concrete constraints are implemented as dynamically loadable plugins. Additional constraints can be easily added via the plugin mechanism.

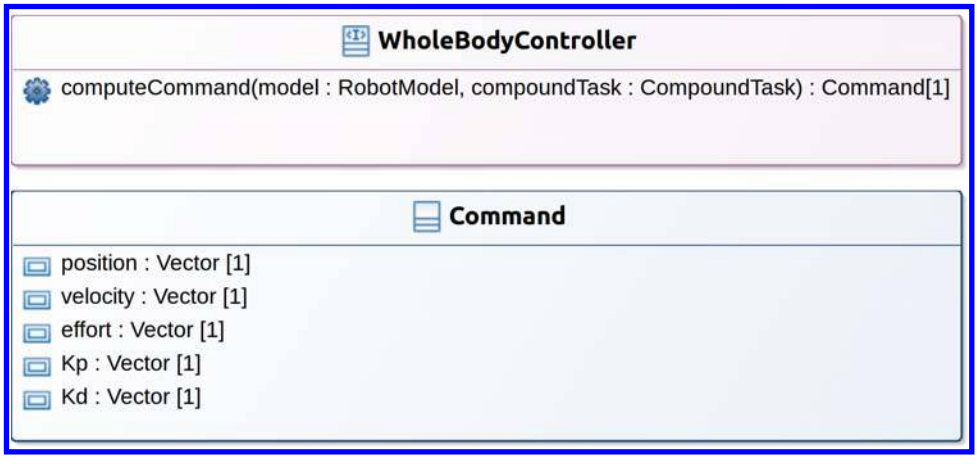


Fig. 5. The WBC software abstractions within ControlIt! consist of an interface called `WBC` and a class called `Command`. The `WBC` interface defines a single method called `computeCommand` that takes two input parameters, the robot model, which includes the constraint set, and the compound task. It returns a `Command` object. The command includes position, velocity, effort, and position controller gains. Depending on the type of joint controller used, one or more of the member variables inside the command may not be used. For example, a force- or torque-controlled robot will only use the effort specification within the command.

the constraint's Jacobian. Like the task library, the constraint library can be easily extended with new constraints via plugins.

Whole body control. The class diagrams for the WBC software abstractions are shown in Fig. 5. There are two classes: `WholeBodyController` and `Command`. `WholeBodyController` is an interface that contains a single method, `computeCommand()`. This method takes as input the robot model, which includes the constraint set, and the compound task. It performs the WBC computations that generate a command for each joint under its control and returns it within a `Command` object. The `Command` object specifies the desired position, velocity, effort, and feedback gains. Note that sometimes not all of the fields within a command are used. For example, a robot that is effort controlled will only use the effort command. The optional fields within the command are included to support robots with joints that are impedance/position controlled.

The whole body controller within ControlIt! is dynamically loaded as a plugin. Two plugins are currently available as shown in Fig. 6. They include WBOSC and WBOSC_Impedance. The WBOSC plugin implements the WBOSC algorithm. It computes the nullspace of the constraint set and projects the task commands through this nullspace. Task commands are iteratively included into the final command based on priority. The commands of tasks at a particular priority level are projected through the nullspaces of all higher priority tasks and the constraint set. This ensures that all constraints are met and that higher priority tasks override lower priority tasks. The output of `WBOSC` is an effort command that can be sent to effort

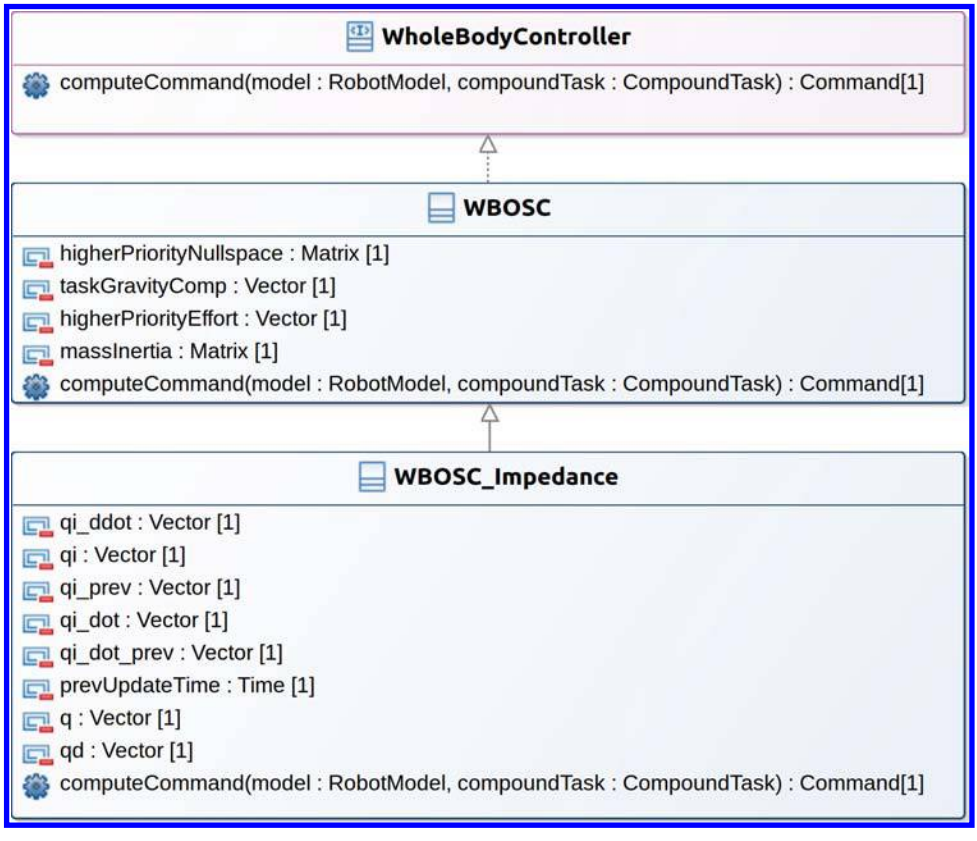


Fig. 6. ControlIt! currently includes two plugins in its WBC plugin library. They consist of `WBOSC` and `WBOSC_Impedance`. `WBOSC` implements the actual `WBOSC` algorithm that takes a holistic view of the robot and achieves multiple prioritized task objectives using nullspace projection. It outputs an effort command and is used with effort-controlled robots like Dreamer. The second plugin, `WBOSC_Impedance`, extends `WBOSC` with an internal robot model that is used to derive the desired joint positions and velocities based on the torque commands generated by `WBOSC`. This is useful to support robots with joint position/impedance controllers like NASA JSC's Valkryie.

controlled robots like Dreamer. The member variables within the WBOSC plugin ensure that memory is pre-allocated, which reduces execution time jitter and thus increases real-time predictability.

To support impedance/position-controlled robots, ControlIt! also comes with the `WBOSC_Impedance` plugin. Unlike effort-controlled robots, impedance-controlled robots take desired position and velocity commands, and position and velocity feedback gains. The benefit of using impedance control is the ability to attain higher levels of stiffness since the position and velocity feedback control loop can be closed by the embedded joint controller, which typically has a higher servo frequency and lower communication latency than the central WBC controller. The `WBOSC_Impedance` plugin extends the WBOSC plugin with an internal model that

converts the effort commands generated by the WBOSC algorithm into expected joint positions and velocities. The member variables within the `WBOSC_Impedance` plugin that start with "qi_" hold the internal model's joint states. The `prevUpdateTime` member variable records when this internal model was last updated. Each time `computeCommand` is called, `WBOSC_Impedance` computes the desired effort command using WBOSC. It then uses this effort command along with the robot model to determine the desired accelerations of each joint. `WBOSC_Impedance` then updates the internal model based on these acceleration values, the time since the last update, the previous state of the internal model, and the actual position and velocity of the joints. The derived joint positions, velocities, and efforts are saved within a `Command` object, which is returned. As previously mentioned, this control strategy was used on the upper body of NASA JSC's Valkyrie robot to perform several DRC manipulation tasks.

Hardware abstraction. To enable support for a wide variety of robot platforms, ControlIt! includes a HAL consisting of two abstract classes, the `RobotInterface` and the `Clock`, as shown in Fig. 7. Concrete implementations are provided through dynamically loadable plugins. `RobotInterface` is responsible for obtaining the robot's joint state and sending the command from the whole body controller to the robot. For diagnostic purposes, it also publishes the state and command information onto ROS topics using a real-time ROS topic publisher, which uses a thread-pool to offload the publishing process from the servo thread. `Clock` instantiates the servo thread and contains a reference to a Controller, which is implemented by the Coordinator. `Clock` is responsible for initializing the controller by calling `servoInit()` and then periodically executing the servo loop by calling the

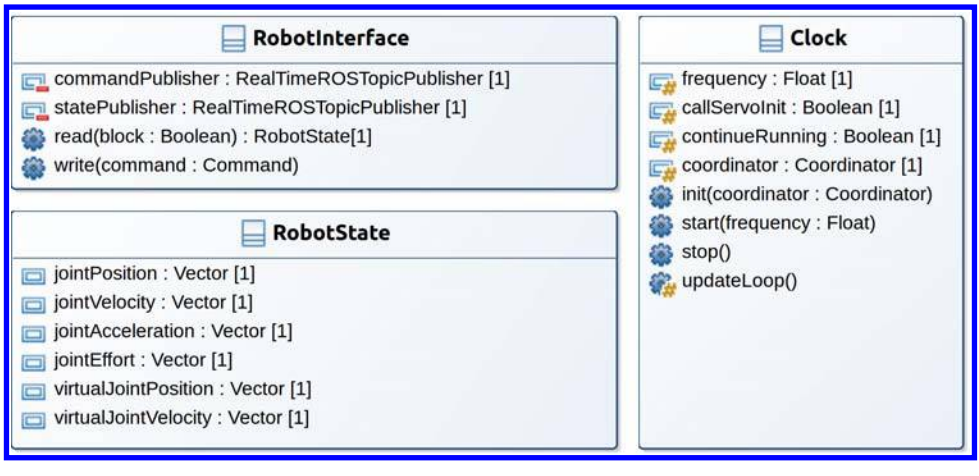


Fig. 7. ControlIt! employs a HAL that consists of a `RobotInterface` and a `Clock`. The `RobotInterface` has two methods: `read` and `write`. The `read` method returns a `RobotState` object that includes the robot's joint positions, velocities, accelerations, and efforts. The `write` method takes as input a `Command` object and issues the command to the robot joints.

`servoUpdate()` method. Initialization using the actual servo thread is needed to handle situations where certain initialization tasks can only be done by the servo thread. This occurs, for example, when the servo thread is part of a real-time context meaning only it can initialize certain real-time resources.

ControlIt! includes libraries of `RobotInterface` and `Clock` plugins as shown in Fig. 8. RobotInterface plugins include general ones that communicate with a robot via three different transport layers: ROS topics (RobotInterfaceROSTopic), UDP datagrams (RobotIntefaceUDP), and shared memory (RobotInterfaceSM). These are meant for general use – ControlIt! includes generic Gazebo plugins and abstract classes that facilitate the creation of software adapters for allowing simulated and real robots to communicate with ControlIt! using these three transport layers. Among the three transport layers, shared memory has the lowest latency and is most reliable in terms of message loss. It uses the ROS shared memory interface package,[113] which is based on boost's interprocess communication library.

In addition to general `RobotInterface` plugins, ControlIt! also includes two robot-specific plugins, one for Dreamer (`RobotInterfaceDreamer`), and one for Valkyrie (`RobotInterfaceValkyrie`). `RobotInterfaceDreamer` interfaces with a RTAI real-time shared memory segment that is created by the robot's software platform called the M3 Server. It also implements separate PID controllers for robot joints that are not controlled by WBC. They include the finger joints in the right hand, the left gripper joint, the neck joints, and the head joints. In the current implementation, these joints are fixed from WBC's perspective. `RobotInterfaceValkyrie` interfaces with the shared memory segment is created by Valkyrie's software platform. This involves integration with a controller manager provided by ros_control[7] to gain access to robot resources.

ControlIt! includes several `Clock` plugins to enable flexibility in the way the servo thread is instantiated and configured to be periodic. The current `Clock` plugin library includes plugins for supporting servo threads based on a ROS timer, a C++`std::chrono` timer, or an RTAI timer. Support for additional methods can be included in the future as additional plugins.

4.2. *Configuration and integration*

Support for configuration and integration is important because, as a software framework, ControlIt! is expected to be (1) used in many different applications and hardware platforms that require different whole body controllers and (2) just one component in a complex application consisting of many components. In addition, ControlIt!'s configuration and integration capabilities directly impacts the software's usability, which must be high to achieve widespread use. ControlIt! supports integration through four mechanisms: (1) parameter reflection, which exposes controller parameters to other objects within ControlIt! and is used by the other two mechanisms, (2) parameter binding, which enables the parameters to be connected to external processes through an extensible set of transport layers, (3) events, which

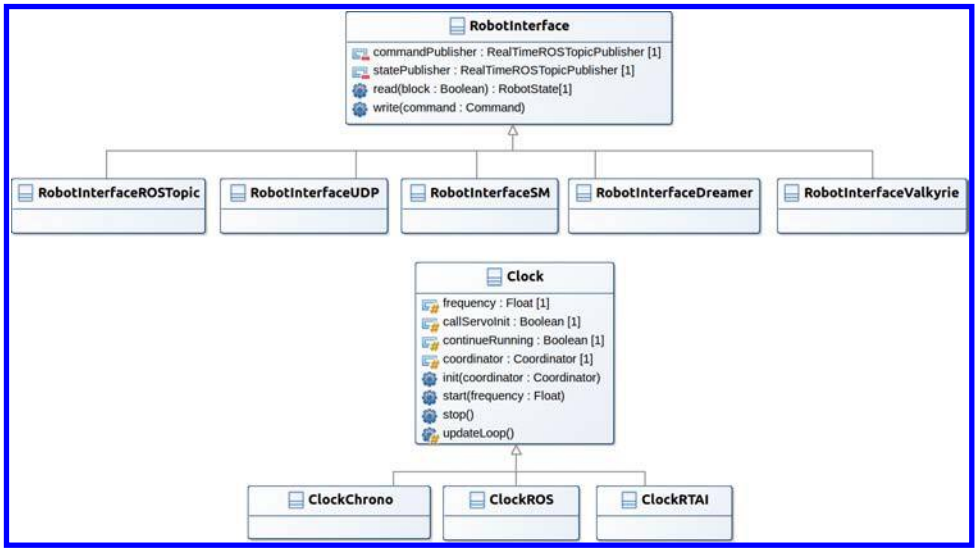


Fig. 8. The robot interface plugins that are currently available include support for the following transport protocols: ROS Topic, UDP, and shared memory. There are also specialized robot interfaces for Dreamer and Valkyrie. The clocks provided include support for `std::chrono`, ROS time, and RTAI time.

enable parameter changes to trigger the execution of external processes without the use of polling, and (4) services, which enable external processes to query information about the controller. ControlIt! supports configuration through scripts that enable users to specify the structure of the compound task and constraint set, the type of whole body controller and hardware interface to use, the initial values of the parameters, the parameter bindings, and the events. These scripts are interpreted during ControlIt!'s initialization to automatically instantiate the desired whole body controller and integrate it into the rest of the system. Details of ControlIt!'s support for configuration and integration are now discussed.

Parameter Reflection. Parameter reflection was originally introduced in Stanford-WBC. It defines a `ParameterReflection` parent class through which child class member variables can be exposed to other objects within ControlIt!. The API and class hierarchy of `ParameterReflection` is shown in Fig. 9(a). Parameter reflection enables internal controller parameters to be exposed to other objects within ControlIt!. It does this by specifying methods for declaring and looking up parameters. When a parameter is declared, it is encapsulated within a `Parameter` object, which contains a name, pointer to the actual variable, a list of bindings, and a method to set the parameter's value. Subclasses of `ParameterReflection` can declare their member variables as parameters and thus make them compatible with ControlIt's parameter binding and event mechanisms, which are now discussed.

Parameter Binding. Parameter binding enables the integration of ControlIt! with other processes in the system by connecting parameters to an extensible set of transport layers. Its API and class hierarchy is shown in Fig. 9(b). The classes that

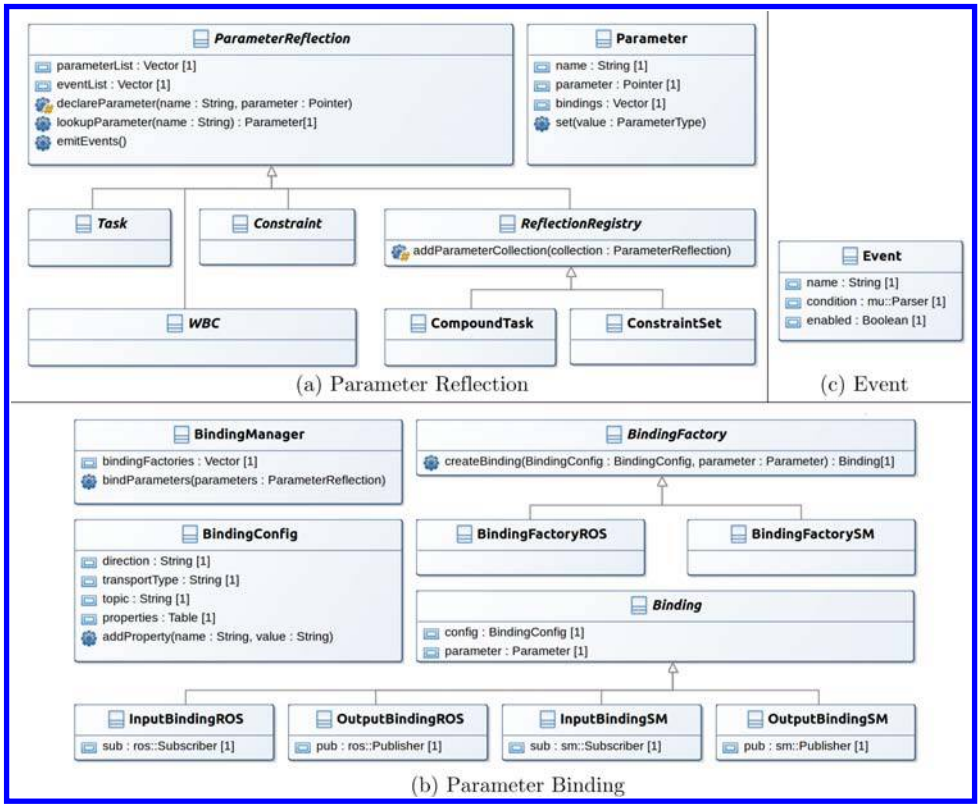


Fig. 9. ControlIt! includes three mechanisms for integration: parameter reflection, parameter binding, and events. Sub-figure (a) shows the parameter reflection mechanism that enables parameters to be exposed to other objects within ControlIt! including the parameter binding and event mechanisms. Sub-figure (b) shows the parameter binding mechanism that enables parameters to be bound to an extensible set of transport layers, which enables them to be accessed by external processes. Sub-figure (c) shows an event definition. Events are stored within Parameter Reflection objects and are emitted at the end of the servo loop. They enable external processes to be notified when a logical expression over a set of parameters transitions from being false to true and eliminates the need for external processes to poll for state changes within ControlIt!.

constitute the parameter binding mechanism consist of a `BindingManager` that maintains a set of `BindingFactory` objects that actually create the bindings, and a `BindingConfig` object that specifies properties of a binding. The required properties include the binding direction (either input or output), the transport type, which is a string that must match the name of a `Binding` provided by a `BindingFactory` plugin, and a topic to which the parameter is bound. The `BindingConfig` also contains an extensible list of name value properties that is transport protocol specific. For example, transport-specific parameters for ROS topic output bindings include the publish rate, the queue size, and whether the latest value published should be latched.

During the initialization process, `BindingConfig` objects are stored as parameters within a ParameterReflection object, which is passed to the `BindingManager`. The `BindingManager` searches through its `BindingFactory` objects, which are dynamically loaded via plugins, for factories that are able to create the desired binding. The current bindings in ControlIt!'s binding library include input and output bindings for ROS topics and shared memory topics. More can be easily added in the future via plugins. The newly created `Binding` objects are stored in the parameter's `Parameter` object. When a parameter's value is set via `Parameter.set()`, the new value is transmitted through output bindings to which the parameter is bound. This enables changes in ControlIt! parameters to be published onto various transport layers notifying external processes of the latest values of the parameters. Similarly, when an external process publishes a value onto a transport layer to which a parameter is bound via an input binding, the parameter's value is updated to be the published value. This enables, for example, external processes to dynamically change a task's reference parameters or controller gains, which is necessary for integration.

Events. Events contain a logical expression over parameters that are interpreted via muParser,[114] an open-source math parser library. Its API is shown in Fig. 9(c). Events are stored in the `ParameterReflection` parent class. The servo thread calls `ParameterReflection.emitEvents()` at the end of every servo cycle. The names of events whose expressions evaluate to true are published on ROS topic `/[controller name]/events`. Events contain a Boolean variable called "`enabled`" that is used to prevent an event from continuously firing when the condition expression remains true since this would likely flood the events ROS topic. Instead, events maintain a fire-once semantic meaning they only fire when the condition expression changes from false to true.

Service-based controller introspection capabilities. To further assist ControlIt! integration into a larger system, ControlIt! also includes a set of service-based introspection capabilities. Unlike ROS topics, which are asynchronous and unidirectional, ROS services are bi-directional and synchronous. ControlIt! uses this capability to enable external processes to query certain controller properties as it is running. For example, two often-used services include `/[controller name]/diagnostics/getTaskParameters`, which returns a list of all tasks in the compound task, their parameters, and their parameter values, and `/[controller name]/diagnostics/getRealJointIndices`, which returns the ordering of all real joints in the robot. This is useful to determine the joint order when updating the reference positions of a posture task or interpreting the meaning of the posture task's error vector. A full list of ControlIt!'s service-based controller introspection capabilities is provided in Appendix C.

Script-based configuration and initialization. As previously mentioned, ControlIt! supports script-based configuration specification and initialization enabling integration into different applications and platforms without being recompiled. This is necessary given the plethora of properties that must be defined and the wide range of anticipated applications and hardware platforms. To instantiate a

whole body controller using ControlIt!, the user must specify many things including the compound task, constraint set, whole body controller, robot interface, clock, initial parameter values, parameter bindings, and events. In addition, there are numerous controller parameters as defined in Appendix B. ControlIt! enables users to define the primary WBC configuration and integration abstractions including tasks, constraints, compound tasks, constraint set, parameter bindings, and events via a YAML file whose syntax is given in Appendix D. The remaining parameters are defined through the ROS parameter server, which can also be initialized via another YAML file that is loaded via a ROS launch file.[115] ROS launch is a powerful tool for loading parameters and instantiating processes. ControlIt! leverages this capability to enable users to initialize and execute a whole body controller and all its surrounding processes using a single command.

4.3. *Multi-threaded architecture*

Higher servo frequencies can be achieved by decreasing the amount of computation in the servo loop. The amount of computation can be reduced because robots typically move little during one servo period, which is usually ≤ 1 ms. Thus, state that depends on the robot configuration like the robot model and task Jacobians often do not need to be updated every servo cycle. ControlIt! takes advantage of this possibility by offloading the updating of the robot model and the task states, which include the task Jacobians, into child threads. Specifically, ControlIt! uses three threads as shown in Fig. 10. They include (1) a `Servo` thread that executes the servo loop, (2) a `ModelUpdater` thread that updates the robot model, which includes the kinematics, inertia matrix, gravity compensation vector, the constraint set, and the virtual linkage model, and (3) a `TaskUpdater` thread that updates the states of each task in the compound task, which includes the task Jacobians. The `Servo` thread is instantiated by the `Clock` and can thus be real time when, for example, `ClockRTAI` is used. `ModelUpdater` and `TaskUpdater` are child threads that do not operate in a real time manner. From a high-level perspective, `Servo` provides `ModelUpdater` with the latest joint states. The `ModelUpdater` uses this information to update the robot model in parallel with the `Servo` thread, and provides the updated robot model to the `Servo` thread when complete. Whenever the robot model is updated, the `Servo` thread provides the updated model to the `TaskUpdater` thread, which updates the task states. These updated task states are then provided to the `Servo` thread. Details on how this process is achieved in a non-blocking and safe manner are now discussed.

Two key requirements of the multi-threaded architecture are (1) the servo thread must not block and (2) there must not be any race conditions between threads. The first requirement implies that the servo thread cannot call the blocking `lock()` method on the mutexes protecting the shared states between it and the child threads. Instead, it can only call the non-blocking `try_lock()` method, which returns immediately if the lock is not obtainable. ControlIt!'s multi-threaded architecture is thus structured to only require calls to `try_lock()` by the `Servo` thread. To prevent

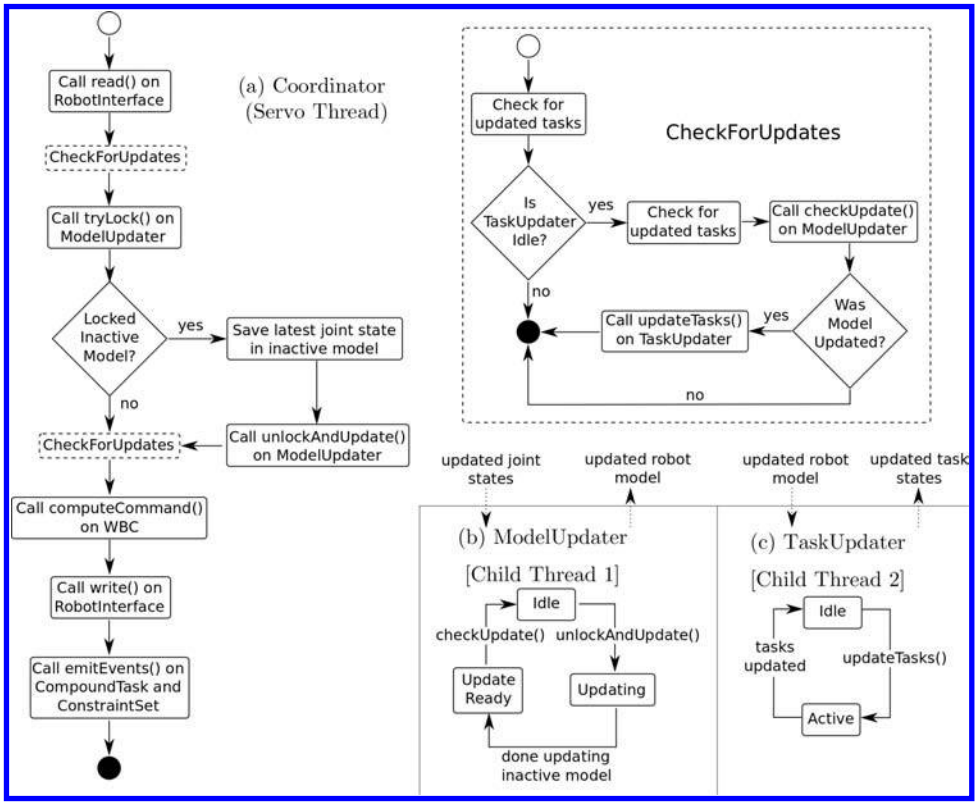


Fig. 10. To achieve higher servo frequencies, ControlIt! employs a multi-threaded architecture consisting of three threads: (a) `Servo`, (b) `ModelUpdater`, and (c) `TaskUpdater`. `Servo` is a real-time thread whereas `ModelUpdater` and `TaskUpdater` are non-real-time threads. This figure shows the behavior and interactions of these threads. At a high level, `Servo` gives `ModelUpdater` the latest joint states and receives an updated `RobotModel`. It also gives `TaskUpdater` an updated `RobotModel` and receives updated state for each task, which includes the task Jacobians. To prevent the `Servo` thread from blocking, which is necessary for real-time operation, ControlIt! maintains two copies of the `RobotModel` and two copies of the state for each task — an "active" one and an "inactive" one. Active versions are used solely by `Servo`. Inactive versions are updated by the child threads. To get updates from the child threads, the `Servo` thread swaps the active and inactive versions when it can be done in a non-blocking and safe manner. It does this by calling the non-blocking `tryLock()` operation on the mutex protecting the inactive version of the `RobotModel` and only performing the swap when it successfully obtains the lock. The swapping of task state is kept non-blocking and safe through FSM design — a task will only indicate it has updated state after the `TaskUpdater` thread is done updating it. To prevent contention between the child threads, the inactive and active robot models can only be swapped when `TaskUpdater` is idle. To further reduce unnecessary computations, `TaskUpdater` only executes after the `RobotModel` is swapped.

race conditions between threads, two copies of the robot model and task state are maintained: an "active" copy that is used by the `Servo` thread, and an "inactive" one that is updated by the other threads. Updates from the child threads are provided to the `Servo` thread by swapping the active and inactive states. This swapping is done by the `Servo` thread in a non-blocking and opportunistic manner.

Figures 10(a) and 10(b) show how the `Servo` thread passes the latest joint state to the `ModelUpdater` thread and trigger it to execute. After obtaining the latest joint states by calling `RobotInterface.read()` and checking for updates from the child threads by executing the `CheckForUpdates` finite state machine, the servo thread attempts to obtain the lock on the mutex protecting the inactive `RobotModel` by calling `ModelUpdater.tryLock()`. If it obtains the lock on the mutex, it saves the latest joint states in the inactive `RobotModel` and then triggers the `ModelUpdater` thread to execute by calling `ModelUpdater.unlockAndUpdate()`. As the name implies, the `Servo` thread releases the lock on the inactive `RobotModel` thereby allowing the `ModelUpdater` thread to access and update the inactive `RobotModel`. If the `Servo` thread fails to obtain the lock on the inactive `RobotModel`, the `Model Updater` thread must be busy updating it. In this situation, the `Servo` thread continues without updating the inactive `RobotModel`.

To prevent race conditions between the `Servo` thread and the child threads, updates from child threads are opportunistically pulled by the `Servo` thread. This is because the child threads operate on inactive versions of the `RobotModel` and task states, and only the `Servo` thread can swap the active and inactive versions. There are two points in the servo loop where the `Servo` thread obtains updates from the child threads. This is shown by the two "`CheckForUpdates`" states in left side of Fig. 10(a). They occur immediately after obtaining the latest joint states by calling `RobotInterface.read()`, and immediately after triggering the `ModelUpdater` thread to run or failing to obtain the lock on the inactive robot model. More checks for updates could be interspersed throughout the servo loop but we found these two placements to be sufficient.

The operations of the `CheckForUpdates` state are shown in the upper-right corner in Fig. 10. The `Servo` thread first obtains task state updates and then checks whether the `TaskUpdater` thread is idle. If it is idle, the `Servo` thread checks for updated task states again. This is to account for the following degenerate thread interleaving during the first check for updated task states that could result in loss of updated task state:

(1) The `Servo` thread begins to check some of the tasks for updated states.
(2) `TaskUpdater` updates all of the tasks including those that were just checked by the `Servo` thread and returns to idle state. Note that this is possible even if the `Servo` thread is real time and has higher priority since `TaskUpdater` may execute on a different CPU core.
(3) The `Servo` thread completes checking the remainder of the tasks for updates.

In the above scenario, the tasks that were checked in step 1 would have updated states that would be lost without the `Servo` thread re-checking for them after it confirms that the `TaskUpdater` is idle. In a worst-case scenario, the `TaskUpdater` thread may update all of the tasks after the servo thread checks for updates but before it checks whether the `TaskUpdater` is idle, resulting in the loss of updated

state from every task. The loss of updated task state is not acceptable despite the presence of future update rounds since it is theoretically possible for the updated states of the same tasks to be continuously lost during every update round. While improbable, this "task update starvation" problem was actually observed and thus discovered while testing ControlIt! on Valkyrie.

After verifying that the `TaskUpdater` thread is idle and ensuring all of the updated task states were obtained, the `Servo` thread checks for an updated `RobotModel` by calling `ModelUpdater.checkUpdate()`. This method switches to the updated `RobotModel` if one is available. If the model was updated, the `Servo` thread then calls `TaskUpdater.updateTasks()` passing it the updated `RobotModel`. This method is non-blocking since the `TaskUpdater` thread must be idle. It triggers the `TaskUpdater` thread to update the states of each task in the compound task. Note that if the `RobotModel` was not updated, the `Servo` thread does not call `TaskUpdater.updateTasks()` since task state updates are based on changes in the `RobotModel`.

The current implementation does not consider the possibility that the active `RobotModel` or task states become excessively stale. This can occur if the robot moves so quickly that the model changes significantly since the last time it was updated. ControlIt's multi-threaded architecture can be easily modified to monitor difference between the current robot state and the robot state that was used to update the active `RobotModel` and task states. If the difference exceeds a certain threshold, the `Servo` thread can update the active `RobotModel` itself to prevent excessive staleness. We currently do not implement this because our evaluations did not indicate the need for it.

Sometimes a multi-threaded architecture is not necessary when the robot has a limited number of joints, the control computer is particularly fast, and the compound task is structured to reduce computational complexity (e.g., by using simpler tasks or limiting the number of tasks that share the same priority level). In this case, ControlIt!'s multi-threaded architecture can be disabled by setting two ROS parameters, `single_threaded_model` and `single_threaded_tasks`, to be true prior to starting ControlIt!. Details of these parameters are given in Table B.2 in Appendix B. When these parameters are set to true, the `Servo` thread updates the `RobotModel` and task states every cycle of the servo loop.

Regardless of whether a multi-threaded architecture is used, the servo loop must be executed in a real-time manner. To help facilitate this, no dynamic memory allocation can occur once the servo loop starts. The initialization process consists of instantiating all objects using their constructors and then calling `init()` methods on all of the objects. All necessary memory is allocated during either the construction or initialization phases. To ensure no memory is being dynamically allocated in the linear algebra operations that are extensively used in WBOSC, we tested the code by defining the `EIGEN_RUNTIME_NO_MALLOC` preprocessor macro prior to including the Eigen headers.

5. Evaluation

We integrate ControlIt! with Dreamer, a dual-arm humanoid upper body robot made by Meka Robotics, which was purchased by Google in December 2013. Dreamer's arms and torso contains series elastic actuators and high fidelity torque control. The robot is modeled as a $(16 + 6 = 22)$ DOF robot where 16 are physical joints and the remaining 6 represent the floating DOFs.[e]

5.1. *Product disassembly application*

Using ControlIt!, we developed an application that makes Dreamer disassemble a product as shown in Fig. 11. The task is to take apart an assembly consisting of a metal pipe with a rubber valve installed at one end. To remove the valve, Dreamer is programed to grab and hold the metal pipe with her right hand while using her left gripper to detach the valve. Once separated, Dreamer places the two pieces into separate storage containers.

Two compound task configurations were used:

(1) single priority level containing a joint position task,
(2) dual priority level containing two higher priority Cartesian position tasks and two 2D orientation tasks (one for each wrist) and a lower priority posture task.

The benefits of the second configuration are shown by demonstrating how changing just three controller parameters, i.e., the Cartesian position of the product, enables the controller to adapt to changes in the product's location while continuously minimizing the squared error of the posture task. This is in the spirit of WBC where changes in a low-dimensional space (three Cartesian dimensions) results in desirable changes in a larger dimensional space (e.g., the number of DOFs in the robot).

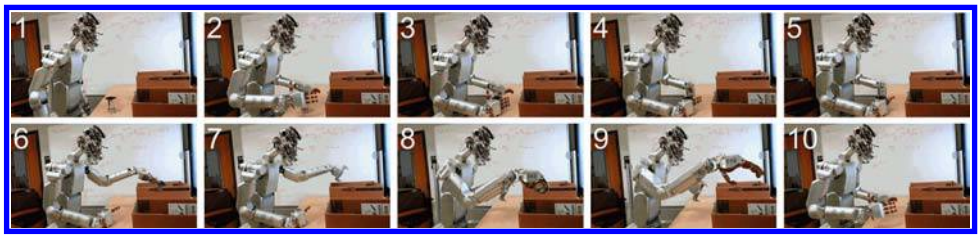


Fig. 11. This sequence of snapshots show the movements of Dreamer performing a product disassembly task. Initially a metal pipe with a rubber valve is in front of Dreamer. To disassemble the product, Dreamer grabs the pipe with her right hand while using her left gripper to remove the valve. The pipe and valve are then placed into separate containers for storage. This demonstrates the integration of ControlIt! with a robot and an application. It shows that the task and constraint libraries are sufficiently expressive to accomplish this application.

[e]WBOSC always assumes a floating base. When the robot is fixed in place, the fixture is represented in WBOSC using a constraint. This enables ControlIt! to support both mobile and fixed robots.

Developing the product disassembly application required writing new `RobotInterface` and `ServoClock` plugins that enable ControlIt! to work with Dreamer. This is because Dreamer comes with the M3 software that is designed specifically for robots built by Meka Robotics. The M3 software includes the M3 Server, which instantiates an RTAI shared memory region through which ControlIt! can transmit torque commands and receive joint state information. In addition, the M3 Server also implements the transmissions that translate between joint space and actuator space and protocols for setting the modes and gains of the joint controllers on the robot's DSPs. Other useful tools provided by the M3 software include applications for tuning and calibrating individual joints. The ControlIt! robot interface we developed for Dreamer is called `RobotInterfaceDreamer`. It uses the shared memory region created by the M3 Server to connect the WBOSC controller to the robot, and implements separate simpler controllers for the joints that are not controlled by WBOSC. These joints include the finger joints in the right hand, the left gripper joint, the neck joints, and the head joints (eyes and ears). In the current implementation, these joints are fixed in place from WBOSC's perspective. While this is not true, they are located at the robot's extremities and are attached to relatively small masses; the feedback portion of the WBOSC controller is able to sufficiently account for these inaccuracies as demonstrated by the successful execution of the application.

Because Dreamer's M3 software is designed to work with RTAI we created an RTAI-enabled servo clock called `ServoClockRTAI`, which instantiates a RTAI real-time thread for executing the servo loop within ControlIt!. Whereas `RobotInterfaceDreamer` is specific to Dreamer, `ServoClockRTAI` can be re-used on any robot that is RTAI-compatible to get real-time execution semantics.

Since Dreamer contains a 2-DOF torso and two 7-DOF arms, we use a compound task containing a Cartesian position and orientation task for each of the two end effectors, and a lower priority joint position task for defining the desired posture. The constraint set contains two constraints: a `FlatContactConstraint` for fixing the robot's base to the world and a `CoactuationConstraint` for the upper torso pitch joint that is mechanically connected to the lower torso pitch joint by a 1:1 transmission. This results in the positions and velocities of the two joints to always be the same. The Jacobian of the `CoactuationConstraint` consists of one row and a column for each DOF in the robot's model. The column representing the slave joint contains a 1 and the column representing the master joint contains the negative of the transmission ratio. Details of these types of constraints were discussed in Ref. 29.

Finally, the goal state and error of every task in the compound task are bound to ROS topics so they can be accessed by the application. A data logger based on ROSBag[116] is used to record experimental data. Figure 12 shows how the various components are connected. Kinesthetic teaching is used to obtain the trajectories for performing the task, which consists of manually moving the robot along the desired trajectories while taking snapshots of the robot's configuration. Cubic spline is used to interpolate intermediate points between snapshots. Note that the application is

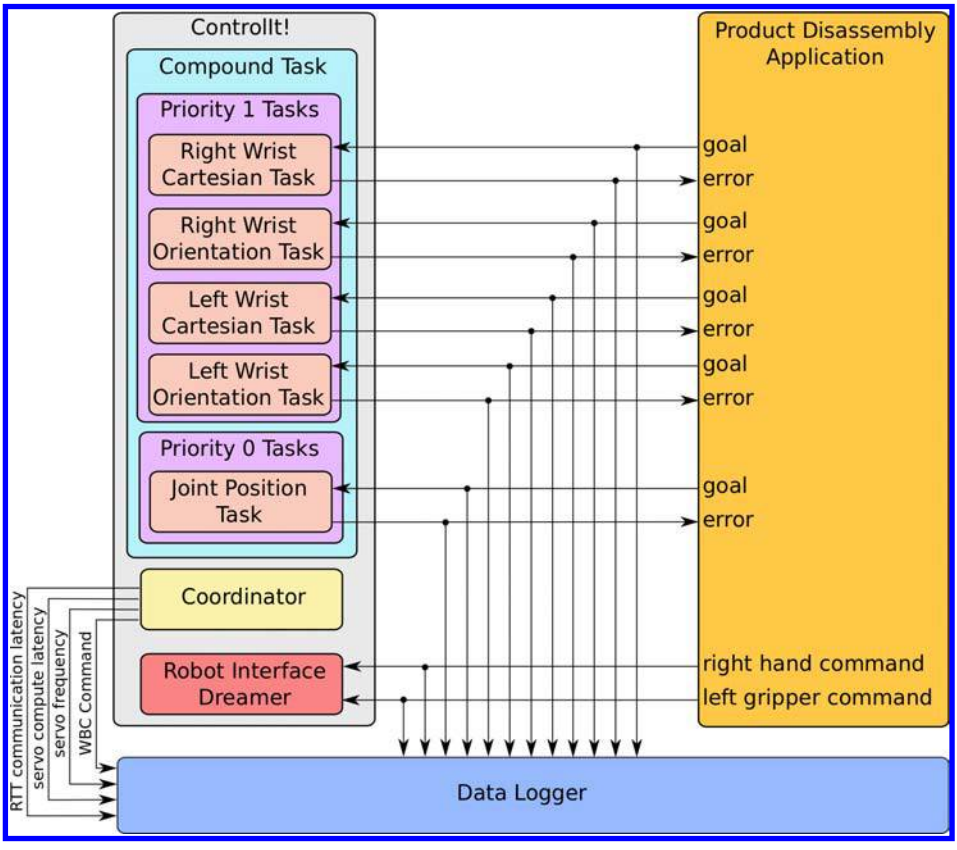
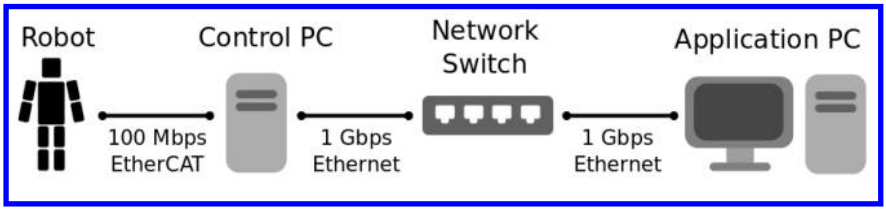


Fig. 12. ControlIt! is integrated into a larger system consisting of three major components: ControlIt!, the application, and a data logger. Each of these components run as a separate process but communicate over ROS topics, which are represented by the arrows. The ROS topics are bound the variables within ControlIt!. The WBOSC configuration consists of two priority levels within the compound task. Higher priority numbers correspond to higher priority tasks. The other components within ControlIt! are not shown since they do not have any bound parameters in this application.

open-loop in that the robot does not sense where the metal pipe and valve assembly is located. We manually reposition the assembly at approximately the same location prior to executing the application.

Before the application can be successfully executed, calibration and gain tuning must be done for every joint and controller in the system. We calibrated and tuned one joint at a time starting from those in the robot's extremities (e.g., wrist yaw joints) and moving inward to joints with increasing numbers of child joints. Once all of the joints were calibrated and torque controller gains tuned, we proceeded to tune the task-level gains in the following order: joint position task, Cartesian position tasks, and finally orientation tasks. The gains used are given in Appendix E. Note that these gains are dependent on ControlIt's servo frequency, which we set to be



(a)

Property	Control PC	Application PC
CPU	Intel Core i7-4771 @ 3.56GHz	Intel Core i7-4771 @ 3.56GHz
Motherboard	Zotac H87	JetWay JNF9J-Q87
OS	Ubuntu 12.04 server, 32-bit, kernel 2.6.32.20, RTAI 3.9, EtherCAT 1.5.1	Ubuntu 14.04 desktop, 64-bit, Kernel 3.13.0-44
Middleware and Applications	ROS Hydro, ControlIt!, M3 Server	ROS Indigo, demo applications, Gazebo

(b)

Fig. 13. The system consists of a humanoid robot that is connected to a control PC over a 100Mbps EtherCAT network. The control PC runs ControlIt! and is connected to an application PC over a two-hop 1Gbps Ethernet network. The application PC runs the application, which remotely interacts with ControlIt! via ROS topics. Details of the hardware and software on the control and application PCs are given in the table. Note that the control PC runs an older operating system and older middleware than the application PC despite having similar hardware. This is because configuring the control PC for real-time operation is time-intensive and thus cannot be repeated every time the operating system or middleware is updated. Allowing applications to run on a separate PC enables them to operate in a more up-to-date software environment and reduces the likelihood of interference between the applications and the controller.

1kHz, and the end-to-end communication latency between the whole body controller and the joint torque controllers, which is about 7 ms.

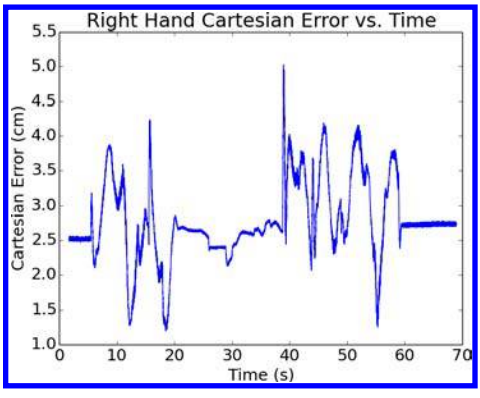
The system architecture is shown in Fig. 13. It consists of the robot, the control PC, and the application PC. The robot communicates with the control PC over a 100 Mbps EtherCAT link. The control PC communicates with an application PC via a 2-hop 1Gbps Ethernet network. The control PC runs ControlIt! on an older but real-time patched version of Linux relative to the application PC. This is because upgrading the operating system on the control PC while maintaining compatibility with RTAI and necessary drivers like EtherCAT and ensuring acceptable real-time performance is a difficult and time-consuming process that requires extensive testing. The product disassembly application could run directly on the Control PC, but we chose to run on a different application PC to emphasize the ability to integrate ControlIt! with remote processes and to allow the application to make use of a newer operating system, middleware, and libraries. In addition, running the application on a separate PC reduces the likelihood that the application would interfere with the whole body controller especially if the application includes complex GPU-accelerated operations.

The application PC includes the dynamics simulator Gazebo.[117] When developing the product disassembly application, we always tested the application in simulation prior to evaluation on real hardware, reducing the number of potentially catastrophic problems encountered on hardware. For example, on the real hardware, if the application crashes while the arms are above the table, the arms may slam into the table and damage both the robot and the table. Testing the application in simulation enabled us to evaluate application stability. We implemented the application in Python (see Appendix F for an example code fragment), which further increases the importance of simulation testing since there is no compilation stage to identify potential problems. Note that the application could have been written in any programming language supported by ROS.[118] Because ControlIt! has a HAL consisting of a RobotInterface plugin and a ServoClock plugin, switching between testing the application in simulation versus on the real hardware is simple and does not require any changes to the code.

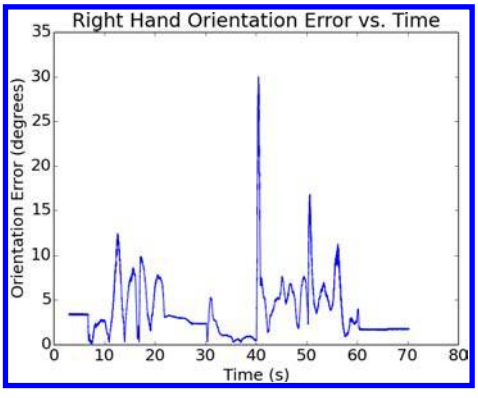
After tuning the controllers, we were able to repeatedly execute the application in a reliable manner. Figure 14 shows performance data collected from one of the many executions of the application. The data was collected from ROS topics to which internal controller parameters were bound. Average statistics are given in Table 2. The results show average servo computational latencies of about 0.5 ms, which is the amount of time the servo thread takes to compute one cycle of the servo loop and is an order of magnitude faster than the 5ms achieved by UTA-WBC. Table 3 shows the results of an experiment that obtains a detailed breakdown of the latencies within the servo loop by instrumenting the servo loop with timers. The values are the average over 1000 executions of the servo loop. The vast majority of the servo loop's computational latency is from executing the WBOSC algorithm to get the next command. Multi-threading significantly decreases the latency of updating the model and slightly decreases the latency of computing the command. The slightly higher average total latency in the multi-threaded case in Table 3 relative to the servo computational latency in Table 2 is most likely due to the additional instrumentation that was added to the servo loop to obtain the detailed latency breakdown information.

The results in Table 2 also show Cartesian positioning errors of up to 5 cm and orientation errors of up to 30°, though the errors are much less on average. Note that the Cartesian position and orientation errors are both model-based meaning they are derived from the joint states and the robot model and not from external sensors like a motion capture system. Thus, the accuracy of these error values depend on the accuracy of the robot's model and should not be considered absolute. However, they do represent the errors that the whole body controller sees and attempts to eliminate but cannot because the feedback gains cannot be made sufficiently high to remove the errors.

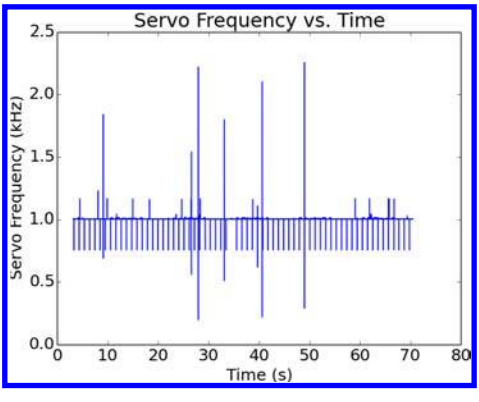
Figures 14(c) and 14(f) indicate a problem with achieving real-time semantics on the control PC since the servo frequency and computational latency occasionally suffers excessively low and high spikes. The lowest servo frequency measured in



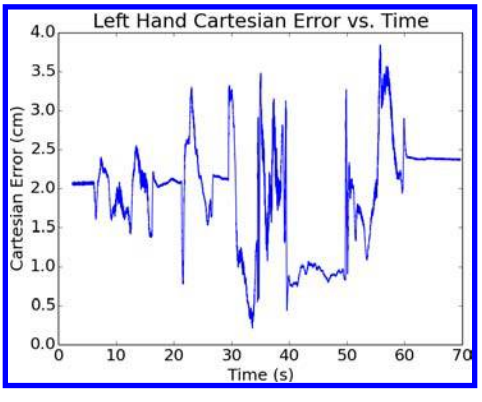
(a)



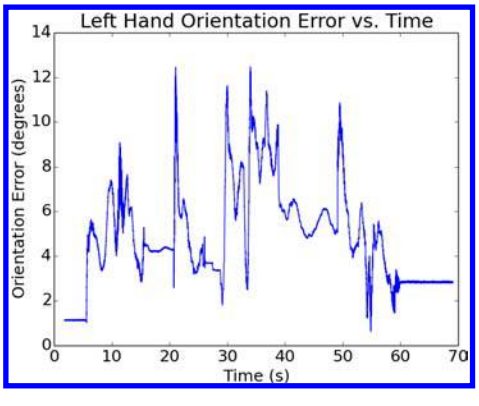
(b)



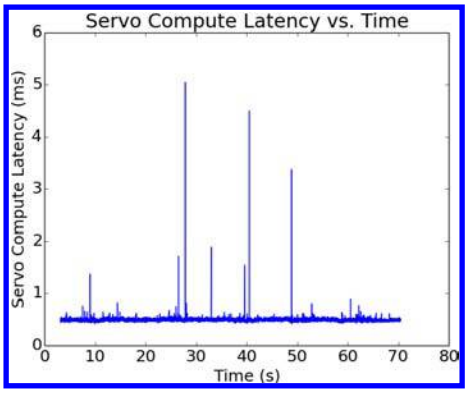
(c)



(d)



(e)



(f)

Fig. 14. Performance data collected from one execution of the product disassembly application.

Table 2. Average statistics of the performance data from one execution of the product disassembly task using the 22-DOF Dreamer model. The average range is the standard deviation of the data set. The results indicate that average Cartesian position error of the end effectors are about 2–3 cm and average orientation is about 3–5°. The servo frequency is slightly above the desired 1 kHz and there is jitter despite running within an RTAI real-time context. The servo compute latency indicates that on average it only takes about 0.5 ms to perform all computations in one cycle of the servo loop, which is significantly faster than the 5 ms required by UTA-WBC.

Statistic	Sample size	Average	Units
Right hand cartesian error	49,137	2.79 ± 0.56	cm
Right hand orientation error	55,735	3.72 ± 3.12	degrees
Left hand cartesian error	43,026	1.91 ± 0.67	cm
Left hand orientation error	50,381	4.86 ± 2.23	degrees
Servo frequency	67,225	1005.43 ± 15.68	Hz
Servo compute latency	64,118	0.487 ± 0.0335	ms

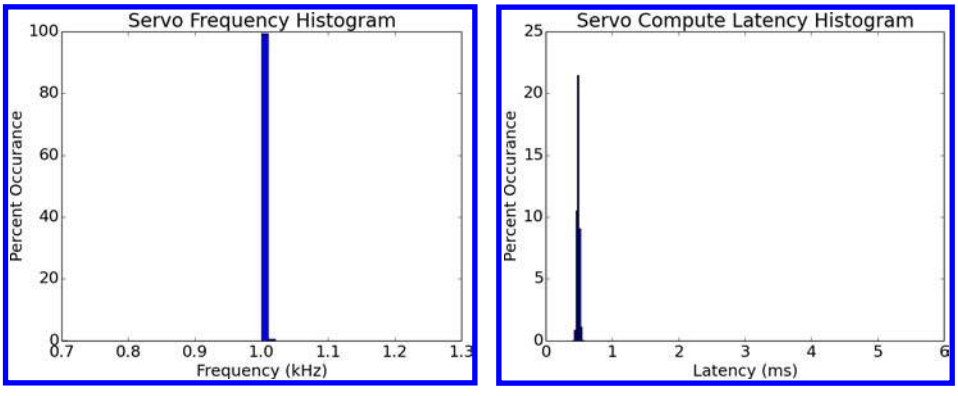
this sample is only 195.3 Hz, the maximum is 2.254 kHz, and the average is 1.01 ± 0.016 kHz. Coincident with the large spikes in the servo frequency are large spikes in the servo compute latency. This indicates that something in the operating system or underlying hardware occasionally prevented ControlIt!'s real-time servo thread from executing as expected. Despite the violations in real-time semantics and errors in Cartesian position and orientation, the ControlIt! is still able to make Dreamer reliably perform the task. This is probably because the spikes are rare as shown by the histograms of the same data as shown in Fig. 15.

5.2. *Latency benchmarks*

The results in Table 2 indicate that the servo loop spends about 0.487 ± 0.0335 ms computing the next command. This is for a specific compound task with two priority levels and 2D orientation tasks and with multi-threading enabled. We now vary the compound task configuration in terms of both number of priority levels (which

Table 3. A breakdown of the latencies incurred within one cycle of the servo loop for both the single- and multi-threaded scenarios using a 22-DOF robot model. All values are in milliseconds and are the average and standard deviation over 1000 samples. Most of the latency is spent computing the command, which includes executing the WBOSC algorithm. The benefits of multi-threading are apparent in the latency of updating the model.

Step in servo loop	Multi-threaded latency	Single-threaded latency
Read joint state	0.020 ± 0.0020	0.020 ± 0.0026
Publish odometry	0.014 ± 0.0041	0.0147 ± 0.00526
Update model	0.0075 ± 0.00256	0.272 ± 0.00235
Compute command	0.470 ± 0.0128	0.497 ± 0.0120
Emit events	0.0036 ± 0.00028	0.0041 ± 0.00027
Write	0.0116 ± 0.00075	0.0125 ± 0.00119
Total	$\mathbf{0.528 \pm 0.0144}$	$\mathbf{0.820 \pm 0.0145}$



(a) (b)

Fig. 15. Histograms of the servo frequency and computational latency measured during one execution of the product disassembly application. The vast majority of the measurements were at the desired 1 kHz frequency and expected 0.5 ms computational latency.

affects the number of tasks per priority level) and types of orientation task used (2D versus 3D). We also evaluate both multi-threaded and single-threaded execution of ControlIt!.

All tests involve five tasks: a Cartesian position task for each of the two end effectors, an orientation task for each of the two end effectors, and a posture task. Two types of orientation tasks are used: 2D and 3D. When 2D orientation tasks are used, only 5 DOFs of each end effector are controlled by the orientation and position tasks; the sixth DOF is controlled by a lower priority posture task. When 3D orientation tasks are used, all 6 DOFs of each end effector are controlled by the orientation and position tasks.

Three configurations of the compound task are evaluated. The first configuration uses two priority levels and assigns all four Cartesian position and orientation tasks to be at the higher priority level. The posture task is located at the lower priority level. The second configuration uses three priority levels and assigns the Cartesian position tasks to be at the highest priority level and the orientation tasks to be in the middle priority level. This is possible since the orientation tasks operate within the nullspace of the Cartesian position tasks. Like the first configuration, the posture task is located at the lowest priority level. The third configuration uses five priority levels. The two Cartesian position tasks are placed in the top two priority levels. The two orientation tasks are placed in the next two priority levels. Finally, the posture task is located in the lowest priority level.

The results are shown in Table 4. The use of multi-threading significantly decreases computational latency by about 0.2–0.3 ms. Interestingly, distributing the tasks across more priority levels decreases computational latency. In this case, placing the orientation tasks and Cartesian position tasks at different priority levels results in a significant decrease in servo computational latency. This is because the

Table 4. The servo loop's computational latency when configured with several different compound tasks and running in both multi-threaded and single-threaded mode using a 22-DOF model. All latencies are the average over 1000 consecutive measurements and the intervals are the standard deviations. The results show that the servo loop's computational latency can be significantly decreased by using multi-threading and placing fewer tasks at each priority level.

Priority levels/task allocation	Orientation task	Threading	Latency (ms)
2 priority levels	2D	Multi	0.528 ± 0.0144
4 tasks at higher priority		Single	0.820 ± 0.0145
1 task at lower priority	3D	Multi	0.999 ± 0.0261
		Single	1.289 ± 0.0218
3 priority levels	2D	Multi	0.494 ± 0.0161
2 tasks at highest priority		Single	0.764 ± 0.0217
2 tasks at middle priority	3D	Multi	0.788 ± 0.0212
1 task at lowest priority		Single	1.068 ± 0.0207
5 priority levels	2D	Multi	0.477 ± 0.0155
1 task at each level		Single	0.744 ± 0.0386
	3D	Multi	0.603 ± 0.0166
		Single	0.882 ± 0.0168

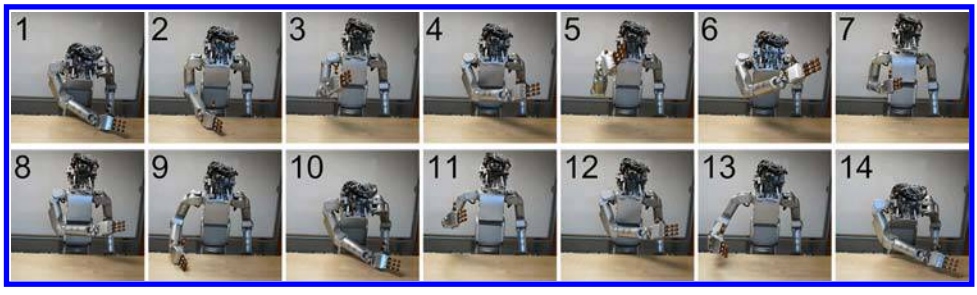
Jacobians and commands of all tasks within the same priority level are concatenated into a large matrix and, in this case, performing operations on large matrices takes more time than performing a larger number of operations and nullspace projections using smaller matrices.

Note that ControlIt! can maintain a 1 kHz servo frequency in many of the compound task configurations even when running in single-threaded mode. Specifically, when 2D orientation tasks are used, 1 kHz servo frequencies are achieved in all compound task configurations. When 3D orientation tasks are used, 1 kHz servo frequencies can be achieved when the five tasks are spread across five priority levels. The 0.882 ± 0.0168 ms that is achieved in this case is similar to the 0.9 ± 0.045 ms that is achieved using an optimized quadratic programming WBC algorithm.[63]

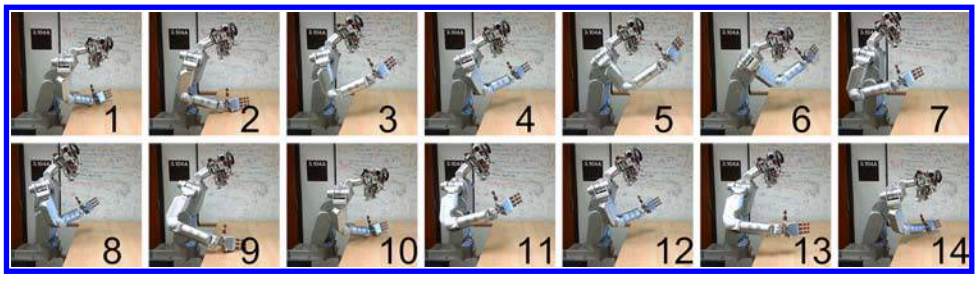
5.3. *Flexible end effector repositioning*

As previously mentioned, the product disassembly application operates open-loop and requires the product to be placed at approximately the same location at the beginning of each execution of the application. For the application to be more robust, additional sensors need to be integrated that can determine the actual location of the product and communicate this information to the application. Such a sensor could be easily integrated since the application is a ROS node meaning it can simply subscribe to the ROS topic onto which the sensor publishes the actual location of the product. Once the application knows where the product is located, it can generate the Cartesian space trajectories that make the end effectors disassemble the product.

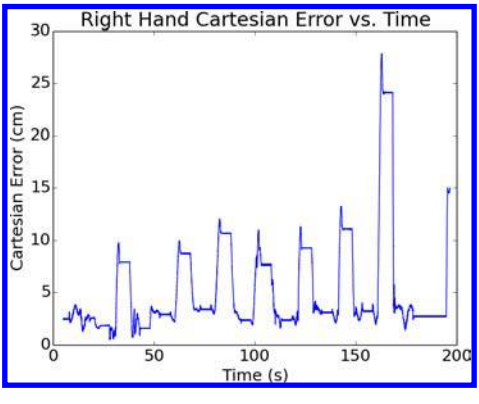
To demonstrate the ability for ControlIt! to make Dreamer follow different Cartesian space trajectories based on a sensed Cartesian goal coordinate, we created an application that makes Dreamer's right hand move to random Cartesian positions



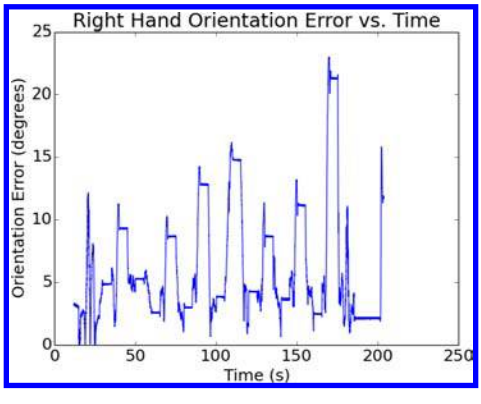
(a)



(b)



(c)



(d)

Fig. 16. This figure shows two different perspectives of the same execution of Dreamer changing the Cartesian position of her right hand while keeping the lower priority joint position task unchanged. It demonstrates WBOSC's ability to handle changes in the goal Cartesian position while predictably handling joint redundancies. The error plots show periodically elevated errors when the goal Cartesian position is moved beyond the robot's workspace. The errors are square-shaped because of a 5-s pause inserted between successive Cartesian trajectories. The controller remains stable despite this problem.

while keeping the lower priority joint position task unchanged. The results are shown in Fig. 16. Note that the right hand is able to move into a wide range of Cartesian positions and that the whole body of the robot moves to help achieve the goal of the right hand's Cartesian position task. The elevated error values that periodically appear in Figs. 16(c) and 16(d) are due to the goal Cartesian position being moved beyond the robot's workspace. Note that the controller remains stable despite this problem. This demonstrates ControlIt!'s ability to be integrated into different applications and WBOSC's ability to handle joint redundancies in a predictable and reliable manner.

6. Discussion

In this section, we provide a brief history of ControlIt's development followed by future research directions.

6.1. *History of ControlIt!'s development*

Prior to integration with Dreamer, ControlIt! was initially developed for NASA JSC's Valkyrie humanoid robot (also called R5).[119] Software and hardware development commenced simultaneously in October 2012. Since hardware development took nearly a year, the first year of developing and testing ControlIt! involved using a simulated version of Valkyrie in Gazebo.[117] During this phase, ControlIt! was initially used to control individual parts of the robot, e.g., each individual limb, the lower body, the upper body, and finally the whole robot. By the summer of 2013, ControlIt! was used to control 32-DOFs of Valkyrie in simulation (6 DOFs per leg, 7 DOFs per arm, 3 DOFs in the waist, and 3 DOFs in the neck). Compound tasks consisting of up to 15 tasks were employed. They include Cartesian position and orientation tasks for the wrists, feet, and the head, an orientation task for the chest, a COM task and posture task for the whole robot, and COP tasks for the feet. Contact constraints for the hands and feet were configured, though not always enabled, depending on whether contact with the environment was being made. Management of all of these tasks and constraints were done using a higher level application called RTC,[100] which provided a graphical user interface for operators to instantiate and configure controllers based on ControlIt!, integrate these controllers with planners and other processes via ROS topics (locomotion was done using a phase space planner[120]), and sequence their execution within a finite state machine. Integration of ControlIt! with Valkyrie in simulation was successful. We were able to do most of the DRC tasks including valve turning, door opening, power tool manipulation, ladder and stair climbing, water hose manipulation, and vehicle ingress. This enabled us to pass the DRC critical design review in June 2013 and continue to participate in the DRC Trials as a Track A team.

By the end of Summer 2013, Valkyrie's hardware development was nearing completion. At this point we began integrating ControlIt! with actual Valkyrie hardware. After using ControlIt! to control parts of the robot individually, we

attempted to control all 32 DOFs but ran into problems where feedback gains could not be increased high enough to sufficiently reduce errors due to modeling inaccuracies. The robot could stand under joint position control but it was not sufficiently stiff to locomote and certain joints like the knees and ankles would frequently overheat. We later hypothesized that one problem was likely due to high communication latencies between ControlIt! and the joint-level controllers. We have since developed a strategy called embedded damping to help maintain stability despite the high communication latency.[121] Since we could not control all 32 DOFs in time for the DRC Trials in December 2013, we resorted to use ControlIt! on Valkyrie's upper body to perform several DARPA Robotics Challenge tasks including opening a door, using a power tool, manipulating a hose, and turning a valve. Laboratory tests of ControlIt! being used to make Valkyrie turn a valve and integrated with the RTC-based operator interface is shown in Fig. 17.

It is important to note that the currently demonstrable capabilities of WBOSC on real hardware is a subset of the capabilities achieved in simulation. For example, while preparing for the DRC critical design review in June 2013, ControlIt! was used to make a simulated Valkyrie walk using a phase-space locomotion planner and a compound task that controls the COP of the feet, the COM location, and the internal tensions between the feet. We will continue to strive to demonstrate these capabilities using ControlIt! on real hardware. Recent results showing an

Fig. 17. This figure shows Valkyrie's upper body being controlled by an early version of ControlIt!. Using a compound task consisting of Cartesian position and orientation tasks for each hand, and a flat contact constraint for the torso, a human operator uses Valkyrie to turn an industrial valve. Parameter binding is used to integrate ControlIt! with the operator's command and visualization applications.

application-specific implementation of WBOSC controlling Hume, a point-foot biped, and making it walk in two dimensions is promising.[120]

6.2. *Future research directions*

As an open-source framework that supports whole body controllers, we hope that ControlIt! will be adopted by the research community and serve as a common platform for developing, testing, and comparing whole body controllers. As a standalone system that works in both simulation and on real hardware, ControlIt! opens numerous avenues of research. For example, ControlIt! currently allows tasks and constraints to be enabled and disabled and to change priority levels at runtime. We tested this on hardware by using a joint position task to get the robot into a ready state and then switching on higher priority Cartesian position and orientation tasks to perform a manipulation operation. The transition resulted in a discontinuity in the torque signal going to the robot, which is not a problem for an upper body manipulation task, but will likely be a problem for legged locomotion.

We are currently considering ways to enable smooth WBOSC configuration changes. For example, one method we are considering is to compute the difference between the current and new compound tasks' torque commands and adjusting for the difference in a feed-forward manner. This feed-forward adjustment can be gradually eliminated to ensure a smooth transition between compound task configurations. Specifically, let $\tau_{\text{command\_old}}$ be the old compound task's command, $\tau_{\text{diff}} = \tau_{\text{command\_old}} - \tau_{\text{command}}$, and $\alpha_\tau \in [0, 1]$. A smooth transition can be achieved by ramping α_τ from 1 to 0 and modifying τ_{command} to be as follows:

$$\tau_{\text{command}} = \tau_{\text{command}} + \alpha_\tau \tau_{\text{diff}}. \tag{9}$$

We recently used this technique on Hume, a biped robot, for smooth transition between contact and non-contact states of the feet.[120]

While ControlIt! is designed to support multiple WBC algorithms via plugins, we currently only have two WBC plugins and both are based on WBOSC. Other successful WBC algorithms incorporate quadratic programming.[26,59,63,122] Unlike WBOSC that analytically solves the WBC problem, quadratic programming is an optimization method that more naturally supports inequality constraints. While quadratic programming is computationally intensive, recent progress on methods to simplify quadratic programming-based whole body controllers have enabled them to execute in less than 1ms on robots with two fewer joints than Dreamer.[63] As future work, it would be interesting to determine (1) whether quadratic programming-based whole body controllers could be implemented as a plugin within ControlIt!'s architecture and (2) the pros and cons of WBOSC relative to quadratic programming-based whole body controllers. Note that others have developed formulations similar to WBOSC that include support for inequality constraints and solve them using quadratic programming.[18,123] The integration of on-line optimization techniques to allow the incorporation of

inequality constraints is an area of future work and may require modifying the current constraint API to include a specification of whether the constraint is negative or positive.

To the best of our knowledge, there are no other multi-threaded open source implementations of WBOSC or other forms of whole body controllers. We are currently unable to prove that our multi-threaded design consisting of a real-time servo thread with two child threads is optimal. Other choices certainly exist. For example, the two child threads could be combined into a single child thread that update both the model and the tasks. Going in the opposite direction, a separate child thread could be instantiated for each task where there is one thread per task. Performing a more detailed analysis on the ideal multi-threaded architecture is a future research direction.

One consequence of adopting a multi-threaded strategy in the robot model is no longer updated synchronously with the servo thread and thus can become stale. We currently do not use any metric to determine when the model has become excessively stale. A child thread simply updates the model as quickly as possible. For our product disassembly task, the child thread was able to update the model fast enough to enable WBOSC to reliably complete the task. A future research direction is to investigate the correlation between model staleness and robot performance.

A given constraint can have an infinite number of null space projectors. The one we use in ControlIt! is the Dynamically Consistent Null Space Projector.[124] The nullspace projector is currently derived within the constraint set. Given the existence of alternative null space projectors, a potential improvement to ControlIt! would be to make the constraint set extensible via plugins. The default plugin will use the current Dynamically Consistent Null Space Projector. However, the user can easily override this by providing a plugin that provides another null space projector.

The results in Sec. 5.1 show that the control PC is unable to maintain hard real-time semantics. There are occasional latency spikes that violate the desired servo frequency. Learning why the latency spikes occur is useful since eliminating them will likely increase system performance. However, we have yet to notice the latency spikes causing any problem during our extensive use of Dreamer. It is worth noting that Dreamer is a COTS robot and its control PC was configured by the robot's manufacturer. Given that the control PC was pre-configured for us, from our perspective, it is a "black box". If the need arises (i.e., the latency spikes actually prevent us from executing a particular task), we will investigate the latency spikes using a two-pronged approach. First, we will instrument the Linux kernel with debug messages that help track down when the latency spikes occur. Second, we will remove all unnecessary kernel modules and disable all unnecessary hardware until the latency spikes no longer occur. We will then slowly add hardware and software modules re-testing for latency spikes after each addition. Once the latency spikes return, we know which hardware or software module caused it and can investigate it further.

In this paper, we did not explicitly account for singularities but they did not pose a problem in our tests even when the arms are fully stretched out as described in Sec. 5.3. This is probably due to our choice of the tolerances for computing pseudo-inverses within the controller. However, we have not performed a detailed study on adequate tolerances nor on handling singularity thus far.

Other future research areas include how to add adaptive control capabilities that continuously improve the robot model based on observed robot behavior, which should enable the resulting WBOSC commands to have an increasingly high feed-forward component and lower feedback component, and the integration of ControlIt! with a network of sensors[125] to enable, for example, visual servoing[126] and integration with higher level sensor-based whole body affordance planning[127] and learning frameworks.[128,129]

7. Conclusions

With the increasing availability of sophisticated multi-branched highly redundant robots targeted for general applications, whole body controllers will likely become an essential component in advanced human-centered robotics. ControlIt! is an open-source software framework that defines a software architecture and set of APIs for instantiating and configuring whole body controllers, integrating them into larger systems and different robot platforms, and enabling high performance via multi-threading. While it is currently focused on facilitating the integration of controllers based on WBOSC, the software architecture is highly extensible to support additional WBC algorithms and control primitives.

This paper provided a software framework that enables the quick instantiation and configuration of WBOSC behaviors for practical applications such as a product disassembly task using a 22-DOF humanoid upper body robot. The experiments demonstrated high performance with servo computational latencies of about 0.5 ms.

In summary, WBC is a rich and vibrant though fragmented research area today with numerous algorithms and implementations that are not cross-compatible and thus difficult to compare in hardware. We present ControlIt! as a software framework for supporting the development and study of whole body operational space controllers and their integration into useful advanced robotic applications.

Acknowledgements

We would like to thank the entire 2013 NASA Johnson Space Center DARPA Robotics Challenge team for helping with the integration, testing, and usage of ControlIt! on Valkyrie. This work is funded in part by NASA grant #NNX12AQ99G, NSF NRI grant #NNX12AM03G, ONR grant #N000141210663, Texas Emerging Technology Fund, and an anonymous donor. We would also like to thank Nicholas Paine for helping with some figures in the evaluation section.

Appendix A. ControlIt! Dependencies

Table A.1. ControlIt! dependencies.

Dependency	Version	Purpose
g++	4.8.2 or 4.6.3	Compiler for C++11 programming language
Eigen	3.0.5	Linear algebra operations
RBDL	2.3.2	Robot modeling, forward and inverse kinematics and dynamics
URDF	1.11.6	Parsing robot model descriptions
ROS	Hydro or Indigo	Component-based software architecture, useful libraries like pluginlib, runtime support like a parameter server and roslaunch bootstrapping capabilities
RTAI	3.9	Real-time execution semantics (only required when using Dreamer or other RTAI-compatible robots)
Gazebo	6.1.0	Test controller in simulation prior to on real hardware

Appendix B. ControlIt! Parameters

Tables B.1 and B.2 contains additional ControlIt! parameters that can be loaded onto the ROS parameter server. They must be namespaced by the controller's name.

Table B.1. ControlIt! parameters (1 of 2).

Name	Description
coupled_joint_groups	Specifies which groups of joints should be coupled. Effectively modifies the model to decouple group of joints from each other. This is useful for debugging purposes or to account for modeling inaccuracies. It is an array of strings.
enforce_effort_limits	Whether to enforce joint effort limits. These limits are specified in the robot description. If true, effort commands exceeding the limits will be truncated at the limit and a warning message will be produced. It is an array of Boolean values.
enforce_position_limits	Whether to enforce joint position limits. These limits are specified in the robot description. If true, position commands exceeding the limits will be truncated at the limit and a warning message will be produced. It is an array of Boolean values.
enforce_velocity_limits	Whether to enforce joint velocity limits. These limits are specified in the robot description. If true, velocity commands exceeding the limits will be truncated at the limit and a warning message will be produced. It is an array of Boolean values.
gravity_compensation_mask	Specifies which joints should not be gravity compensated. This is useful when certain joints have so much friction that gravity compensation is not necessary. It is an array of joint name strings.
log_level	The log level, which can be DEBUG, INFO, WARN, ERROR, or FATAL. This controls how much log information is generated during runtime. It is a string value.

Table B.2. ControlIt! parameters (2 of 2).

Name	Description
log_fields	Specifies the optional fields that are in a log message's prefix. Possible values include: package - the ROS package containing the message file - file containing the message line - the line number of the message function - the method producing the message pid - the process ID of the thread producing the message It is an array of strings.
max_effort_command	Specifies the maximum effort that should be commanded for each joint. A warning is produced if this is violated. It is an array of integers.
parameter_binding_factories	The names of the plugins containing the parameter binding factories to use. It is an array of strings.
robot_description	Contains the URDF description of the robot. This is used to initialize ControlIt's floating model. It is a string value.
robot_interface_type	The name of the robot interface plugin to use. It is a string.
servo_clock_type	The name of the servo clock plugin to use. It is a string value.
servo_frequency	The desired servo loop frequency in Hz. Warnings will be published if this frequency is not achieved. It is an integer value.
single_threaded_model	Whether to use the servo thread to update the model. It is a Boolean value.
single_threaded_tasks	Whether to use the servo thread to update the task states. It is a Boolean value.
whole_body_controller_type	The name of the WBC plugin to use. It is a string value.
world_gravity	Specifies the gravity acceleration along the X-, Y-, and Z-axis of the world frame. Defaults to $\langle 0, 0, -9.81 \rangle$. This is useful for debugging or when working in worlds where the gravity does not pull in negative Z-axis direction. It is an integer array.

Appendix C. ControlIt! Introspection Capabilities

This appendix describes ControlIt!'s introspection capabilities, which enable users to gain insight into the internal states of the controller.

Task-based introspection capabilities. Tasks can be configured to publish ROS `visualization_msgs/MarkerArray` and `visualization_msgs/InteractiveMarkerUpdate` messages onto ROS topics that show the current and goal states of the controller. These messages can be visualized in RViz to understand what the task-level controller is trying to achieve. For example, Fig. C.1 shows the marker array messages published by a 2D orientation task. The green arrow shows the goal heading whereas the blue arrow shows the current heading. Figure C.2 shows visualizations of 2D and 3D orientation tasks and Cartesian position tasks.

ROS service-based introspection capabilities. Table C.1 lists the various service-based controller introspection capabilities that are provided by ControlIt!. These services can be called by external processes and are useful for integrating ControlIt! into a larger system. All services are namespaced by the controller's name enabling multiple instances of ControlIt! to simultaneously exist.

ROS topic-based introspection capabilities. Table C.2 lists the various topic-based controller introspection capabilities that are provided by ControlIt!.

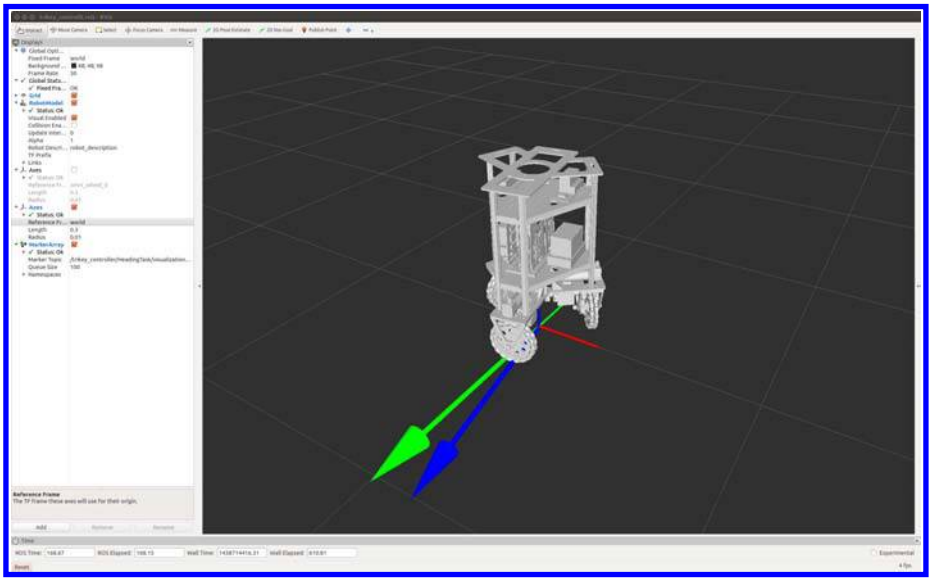


Fig. C.1. When integrated with Trikey, ControlIt! can be configured to publish ROS `visualization_msgs/MarkerArray` messages containing the current and goal headings of the robot. These marker messages can be visualized in RViz as arrows. In this screenshot, ControlIt! is in the process of rotating Trikey clockwise when viewed from above.

These topics can be subscribed to by external processes and are useful for integrating ControlIt! into a larger system. All topics are namespaced by the controller's name enabling multiple instances of ControlIt! to simultaneously exist.

Appendix D. ControlIt! Configuration File

ControlIt! enables users to specify the controller configuration using a YAML file. The syntax of this file is shown below. By enabling YAML-based configuration, ControlIt! can be made to work with a wide variety of applications without modifying the source code and recompiling.

Task specification:

```
tasks:
 - name: [task name] # user defined
   type: [task type] # must match plugin name
   ... # task-specific parameters and their values
 ... # additional tasks
```

Constraint specification:

```
constraints:
 - name: [constraint name] # user defined
   type: [constraint type] # must match plugin name
```

```
   ... # constraint-specific parameters and their values
 ... # additional constraints
```

Compound task specification:

```
compound_task:
 - name: [task name]
   priority: [priority level]
   operational_state: [enable, disable, or sense]
 ... # additional tasks
```

Constraint set specification:

```
constraint_set:
 - name: [constraint name]
   type: [constraint type]
   operational_state: [enable or disable]
 ... # additional constraints
```

Binding Specification:

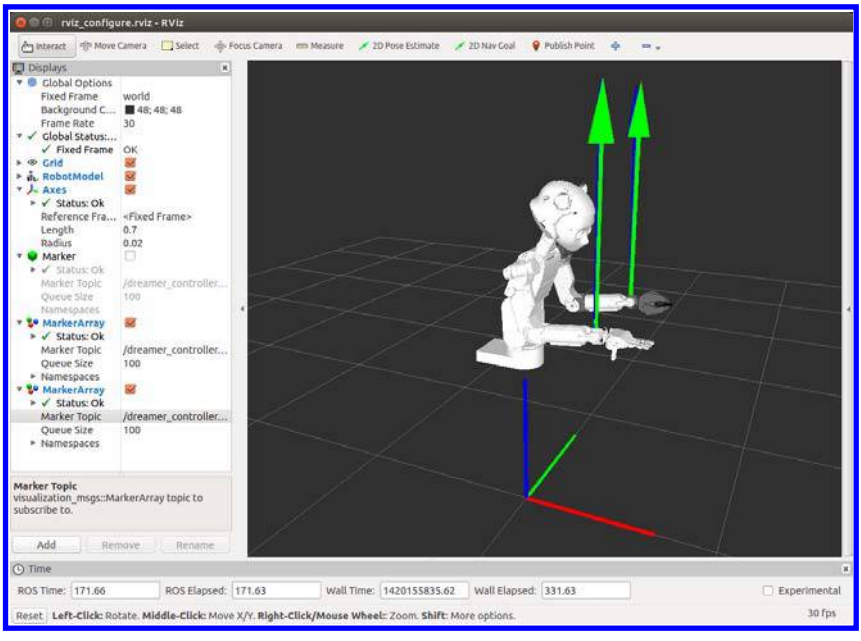
```
bindings:
 - parameter: [parameter name] # must match real parameter name
   direction: [input or output]
   topic: [topic name]
   transport_type: [transport type] # must match plugin name
   properties:
    - [transport-specific property]
 ... # additional transport-specific properties ...
 # additional bindings
```

Event Specification:

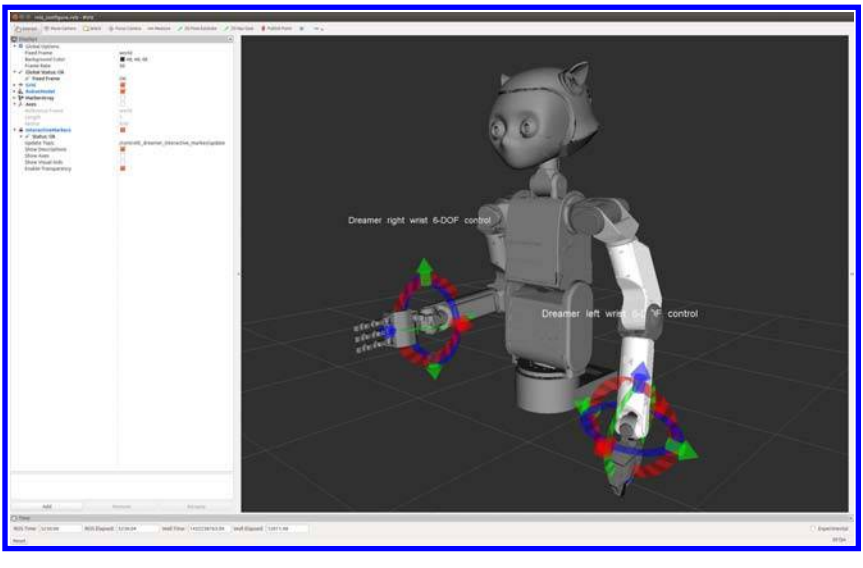
```
events:
 - name: [event name] # user defined
   expression: [logical expression over parameters]
 ... # additional events
```

Appendix E. Controller Gains

The following tables provide the gains used by the various controllers in the product disassembly application using Dreamer. The negative joint position controller gains are strange but were configured as such by Meka Robotics, the robot's manufacturer (Meka Robotics has since been bought by Google). We do not know



(a)



(b)

Fig. C.2. Two Cartesian position tasks and two orientation tasks are used to position and orient Dreamer's end effectors in the world. The orientation and Cartesian position tasks have higher priority than a joint position task that defines the robot's posture. (a) Shows the current and goal 2-DOF orentations. (b) Shows how ROS 6-DOF interactive markers denote the current position and orientation of the wrists. The interactive markers can be dynamically and visually changed by the user to update the goal positions and orientaions of the robot's wrists.

Table C.1. ControlIt!'s ROS service-based controller introspection capabilities.

Service	Description
diagnostics/getActuableJointIndices	Provides the order of every actuable joint in the robot model (omits joints that are real but not actuable)
diagnostics/getCmdJointIndices	Provides the order of the joints in the command issued by ControlIt! to the robot.
diagnostics/getConstraintJacobianMatrices	Provides the Jacobian matrices belonging to the constraints in the constraint set.
diagnostics/getConstraintParameters	Provides a list of every constraint parameter and its current value.
diagnostics/getControlItParameters	Provides the current values of the ControlIt! parameters defined in Appendix A.2.
diagnostics/getControllerConfiguration	Provides the current state of the compound task and constraint set.
diagnostics/getRealJointIndices	Provides the order of every real joint in the robot model.
diagnostics/getTaskParameters	Provides a list of every task parameter is its current value.

Table C.2. ControlIt!'s ROS topic-based controller introspection capabilities.

Service	Description
diagnostics/RTTCommLatency	Publishes the latest round-trip communication time between ControlIt! and the joint-level controllers. This is done by transmitting sequence numbers to the joint-level controllers, which are reflected back through the joint state data. ControlIt! monitors the time between transmitting a particular sequence number and receiving it back.
diagnostics/command	Publishes the latest command issued by ControlIt! to the robot.
diagnostics/errors	Publishes any runtime errors that are encountered. An example error is when the command includes NaN values.
diagnostics/gravityVector	Publishes the current gravity compensation vector.
diagnostics/jointState	Publishes the latest joint state information.
diagnostics/modelLatency	Publishes the staleness of the currently active model. The model latency is the time since the model was last updated.
diagnostics/servoComputeLatency	Publishes the amount of time it took to execute the computations within one cycle of the servo loop.
diagnostics/servoFrequency	Publishes the instantaneous servo frequency.
diagnostics/warnings	Publishes any runtime warnings that are encountered. An example warning is when the joint position or velocity exceeds expected limits.

for sure why some gains are negative since we are unable to access the details of the joint-level controllers. It is possible that the direction of the encoder is opposite of the motor resulting in the need for negative gains. Regardless, these were the functioning settings used in the development and testing of ControlIt! on Dreamer.

The reason why the left and right arms have different gains is because the left arm is about three years newer than the right arm and internally the mechatronics of the left arm are significantly different from that of the right arm.

Table E.1. Dreamer joint torque controller gains.

Controller	Kp	Ki	Kd
torso_lower_pitch	−3	0	0
left_shoulder_extensor	10	1	0
left_shoulder_abductor	10	1	0
left_shoulder_rotator	10	1	0
left_elbow	10	1	0
left_wrist_rotator	50	0	0
left_wrist_pitch	15	0	1
left_wrist_yaw	15	0	1
right_shoulder_extensor	7	0	0
right_shoulder_abductor	6	0	0
right_shoulder_rotator	5	0	0
right_elbow	5	0	0
right_wrist_rotator	−3	0	1
right_wrist_pitch	−15	0	−1
right_wrist_yaw	−15	0	−1

Table E.2. ControlIt! Task-level controller gains used to control dreamer.

Task	Kp	Ki	Kd
Joint position task	60	0	3
Left hand orientation	60	0	3
Right hand orientation	60	0	3
Left hand position	64	0	3
Right hand position	64	0	3

Appendix F. Example Application Code

Figure F.1 contains an example code fragment from the product disassembly. The application is written in the Python programming language, though any programming language supported by ROS could be used including C++. The code fragment shows how the Cartesian position trajectory is generated for moving the right hand into a position where it can grab the metal tube. Lines 548-552 specify the Cartesian (x, y, z) waypoints that the hand is expected to traverse. For brevity, only one waypoint is shown. Line 555 creates a cubic-spline interpolator, which is used on line 559 to generate the intermediate points between the waypoints. The while loop starting on line 564 obtains the current goal Cartesian position based on the elapsed time (line 572) and transmits this goal via a ROS topic (line 576). The goal parameter of the right hand Cartesian position task within ControlIt! is bound to this ROS topic enabling ControlIt! to follow the desired Cartesian trajectory. The trajectory is transmitted at 100 Hz, based on line 579. Once the trajectory is done, line 583 issues a command to close the fingers in the right hand is issued via another bound ROS topic.

```
def grabMetalTube(self):

  # define the right hand Cartesian position trajectory waypoints
  rjCartWP = []
  rjCartWP.append([0.25822435038901964, -0.1895604971725577,
    1.0461857180093073])
  # insert more waypoints here

  # create a cubic-spline-based trajectory generator
  rhCartTG = TrajectoryGeneratorCubicSpline
    .TrajectoryGeneratorCubicSpline(rjCartWP)

  # generate the trajectory over a certain time interval
  TOTAL_TRAVEL_TIME = 5.0 # seconds
  rhCartTG.generateTrajectory(TOTAL_TRAVEL_TIME)

  # execute the trajectory
  done = False
  while not done and not rospy.is_shutdown():

    # compute elapsed time
    deltaTime = self.getTimeSeconds() - startTime
    if deltaTime >= TOTAL_TRAVEL_TIME:
      done = true

    # get the current Cartesian position along trajectory
    goal = rhCartTG.getPoint(deltaTime)

    # save Cartesian goal into ROS message and publish it
    self.rhCartGoalMsg.data = goal
    self.rhCartTaskGoalPublisher.publish(self.rhCartGoalMsg)

    if not done:
      rospy.sleep(0.01) # 100Hz

  # do power grasp
  self.rightHandCmdMsg.data = True
  self.rightHandCmdPublisher.publish(self.rightHandCmdMsg)
}
```

Fig. F.1. Code fragment from product disassembly application.

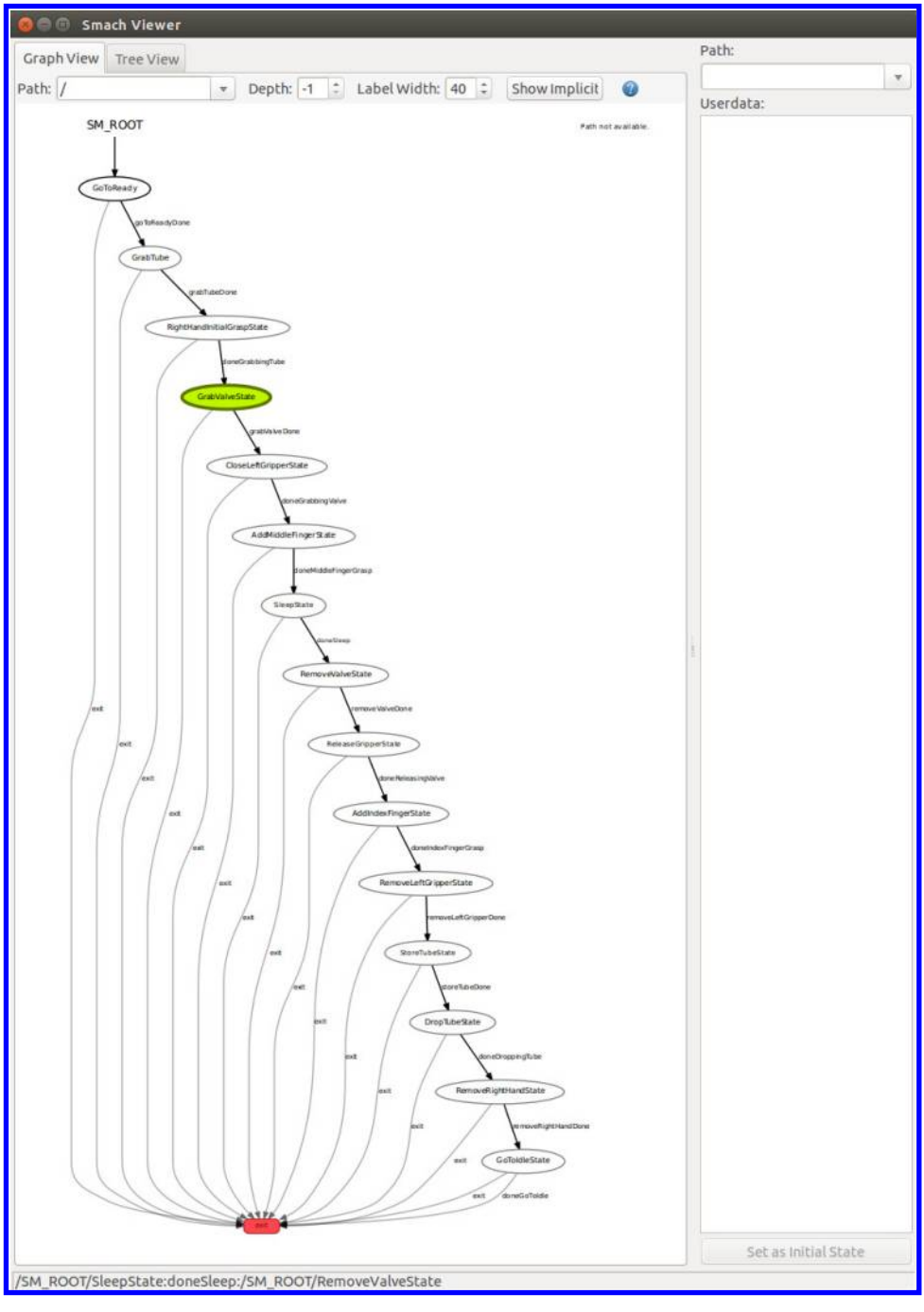


Fig. G.1. This figure shows a visualization of the FSM used by the product disassembly application. The ROS package SMACH is used to both implement the FSM logic and visualize its execution. The highlighted state is the current state of the demo.

Appendix G. ControlIt! SMACH FSM Integration

The following screenshot is a visualization of the product disassembly finite state machine provided by ROS SMACH Visualizer. It is updated in real time as the application executes. This particular screenshot shows that Dreamer is in the "GrabValveState" which is when her left gripper is being positioned to grab the valve.

References

1. O. Khatib, L. Sentis, J. Park and J. Warren, Whole-body dynamic behavior and control of human-like robots, *Int. J. Humanoid Robot.* **1**(1) (2004) 29–43.
2. L. Sentis and O. Khatib, Synthesis of whole-body behaviors through hierarchical control of behavioral primitives, *Int. J. Humanoid Robot.* **2** (2005) 505–518.
3. L. Sentis, Synthesis and control of whole-body behaviors in humanoid systems, Ph.D. dissertation, Stanford University, Supervised by Oussama Khatib (2007).
4. L. Sentis, J. Park and O. Khatib, Compliant control of multicontact and center-of-mass behaviors in humanoid robots, *IEEE Trans. Robot.* **26**(4) (2010) 483–501.
5. IEEE Robotics and Automation Society, Whole body control technical committee (2015), Available at http://www.ieee-ras.org/whole-body-control (accessed on 13 February 2015).
6. I. A. Sucan and S. Chitta, Moveit! (2015), Available at http://moveit.ros.org/ (accessed on 13 February 2015).
7. Robot Operating System, Ros control (2015), Available at http://wiki.ros.org/ros_control (accessed on 13 February 2015).
8. F. Aghili, A unified approach for inverse and direct dynamics of constrained multibody systems based on linear projection operator: Applications to control and simulation, *IEEE Trans. Robot.* **21**(5) (2005) 834–849.
9. S.-H. Hyon, J. G. Hale and G. Cheng, Full-body compliant human — Humanoid interaction: Balancing in the presence of unknown external forces, *IEEE Trans. Robot.* **23**(5) (2007) 884–898.
10. J. Nakanishi, M. Mistry and S. Schaal, Inverse dynamics control with floating base and constraints, *2007 IEEE Int. Conf. Robotics and Automation*, April 2007, pp. 1942–1947.
11. M. Mistry, J. Buchli and S. Schaal, Inverse dynamics control of floating base systems using orthogonal decomposition, *2010 IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2010, pp. 3406–3412.
12. K. Nagasaka, Y. Kawanami, S. Shimizu, T. Kito, T. Tsuboi, A. Miyamoto, T. Fukushima and H. Shimomura, Whole-body cooperative force control for a two-armed and two-wheeled mobile robot using generalized inverse dynamics and idealized joint units, *2010 IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2010, pp. 3377–3383.
13. M. Mistry and L. Righetti, Operational space control of constrained and underactuated systems, in *Proc. Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011, pp. 1–8.
14. L. Righetti, J. Buchli, M. Mistry and S. Schaal, Inverse dynamics control of floating-base robots with external constraints: A unified view, *2011 IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2011, pp. 1085–1090.
15. L. Righetti and S. Schaal, Quadratic programming for inverse dynamics with optimal distribution of contact forces, *2012 12th IEEE-RAS Int. Conf. Humanoid Robots (Humanoids)*, November 2012, pp. 538–543.

16. K. Wakita, J. Huang, P. Di, K. Sekiyama and T. Fukuda, Human-walking-intention-based motion control of an omnidirectional-type cane robot, *IEEE/ASME Trans. Mechatronics* **18**(1) (2013) 285–296.
17. S.-H. Lee and A. Goswami, A momentum-based balance controller for humanoid robots on non-level and non-stationary ground, *Auton. Robots* **33**(4) (2012) 399–414.
18. J. Salini, Dynamic control for the task/posture coordination of humanoids: Towards synthesis of complex activities, Ph.D. dissertation, Universit Pierre et Marie Curie (2013).
19. F. L. Moro, M. Gienger, A. Goswami, N. G. Tsagarakis and D. G. Caldwell, An attractor-based whole-body motion control (wbmc) system for humanoid robots, *2013 13th IEEE-RAS Int. Conf. Humanoid Robots (Humanoids)* (2013), pp. 42–49.
20. L. Righetti, J. Buchli, M. Mistry, M. Kalakrishnan and S. Schaal, Optimal distribution of contact forces with inverse-dynamics control, *Int. J. Robot. Res.* **32**(3) (2013) 280–298.
21. L. Saab, O. Ramos, F. Keith, N. Mansard, P. Soueres and J. Fourquet, Dynamic whole-body motion generation under rigid contacts and other unilateral constraints, *IEEE Trans. Robot.* **29**(2) (2013) 346–362.
22. S. Lengagne, J. Vaillant, E. Yoshida and A. Kheddar, Generation of whole-body optimal dynamic multi-contact motions, *Int. J. Robot. Res.* **32**(9–10) (2013) 1104–1119.
23. T. Koolen, Force control for a humanoid robot using momentum and instantaneous capture point dynamics, *2013 IEEE Int. Conf. Robotics and Automation (ICRA)* (2013). Available at: http://robots.ihmc.us/s/ICRA2013Momentum.pdf
24. B. Henze, C. Ott and M. Roa, Posture and balance control for humanoid robots in multi-contact scenarios based on model predictive control, *2014 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS 2014)*, September 2014, pp. 3253–3258.
25. L. Righetti, M. Kalakrishnan, P. Pastor, J. Binney, J. Kelly, R. Voorhies, G. Sukhatme and S. Schaal, An autonomous manipulation system based on force control and optimization, *Auton. Robots* **36**(1–2) (2014) 11–30.
26. A. Escande, N. Mansard and P.-B. Wieber, Hierarchical quadratic programming: Fast online humanoid-robot motion generation, *Int. J. Robot. Res.* **33**(7) (2014) 1006–1028.
27. K. Nishiwaki, M. Kuga, S. Kagami, M. Inaba and H. Inoue, Whole-body cooperative balanced motion generator for reaching, *Int. J. Humanoid Robot.* **2**(4) (2005) 437–457.
28. S. Hyon, A motor control strategy with virtual musculoskeletal systems for compliant anthropomorphic robots, *IEEE/ASME Trans. Mechatronics* **14**(6) (2009) 677–688.
29. L. Sentis, J. Peterson and R. Philippsen, Implementation and stability analysis of prioritized whole-body compliant controllers on a wheeled humanoid robot in uneven terrains, *Auton. Robots* **35**(4) (2013) 301–319.
30. I. Mizuuchi, Y. Nakanishi, Y. Sodeyama, Y. Namiki, T. Nishino, N. Muramatsu, J. Urata, K. Hongo, T. Yoshikai and M. Inaba, An advanced musculoskeletal humanoid kojiro, *2007 7th IEEE-RAS Int. Conf. Humanoid Robots*, November 2007, pp. 294–299.
31. Y. Nakanishi, S. Ohta, T. Shirai, Y. Asano, T. Kozuki, Y. Kakehashi, H. Mizoguchi, T. Kurotobi, Y. Motegi, K. Sasabuchi, J. Urata, K. Okada, I. Mizuuchi and M. Inaba, Design approach of biologically-inspired musculoskeletal humanoids, *Int. J. Adv. Robot. Syst.* **10**(216) (2013) 1–13.
32. A. Dietrich, C. Ott and A. Albu-Schaffer, Multi-objective compliance control of redundant manipulators: Hierarchy, control, and stability, *2013 IEEE/RSJ International Conf. Intelligent Robots and Systems (IROS)*, November 2013, pp. 3043–3050.
33. C. Ott, M. Roa and G. Hirzinger, Posture and balance control for biped robots based on contact force optimization, *2011 11th IEEE-RAS Int. Conf. Humanoid Robots (Humanoids)*, October 2011, pp. 26–33.

34. J. Englsberger, C. Ott and A. Albu-Schaffer, Three-dimensional bipedal walking control using divergent component of motion, *2013 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, November 2013, pp. 2600–2607.
35. F. Moro, N. Tsagarakis and D. Caldwell, Walking in the resonance with the COMAN robot with trajectories based on human kinematic motion primitives (kMPs), *Auton. Robots* **36**(4) (2014) 331–347.
36. E. Whitman and C. Atkeson, Control of instantaneously coupled systems applied to humanoid walking, in *2010 10th IEEE-RAS Int. Conf. Humanoid Robots (Humanoids)*, December 2010, pp. 210–217.
37. M. Hutter, M. Bloesch, J. Buchli, C. Semini, S. Bazeille, L. Righetti and J. Bohg, AGILITY — Dynamic full body locomotion and manipulation with autonomous legged robots, *2013 IEEE Int. Symp. Safety, Security, and Rescue Robotics (SSRR)*, October 2013, pp. 1–4.
38. K. Hirai, M. Hirose, Y. Haikawa and T. Takenaka, The development of honda humanoid robot, *1998 IEEE Int. Conf. Robotics and Automation, 1998 Proc.*, Vol. 2, May 1998, pp. 1321–1326.
39. S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi and H. Hirukawa, Resolved momentum control: Humanoid motion planning based on the linear and angular momentum, *Proc. 2003 IEEE/RSJ Int. Conf. Intelligent Robots and Systems, 2003. (IROS 2003)*, Vol. 2, October 2003, pp. 1644–1650.
40. K. Bouyarmane and A. Kheddar, On the dynamics modeling of free-floating-base articulated mechanisms and applications to humanoid whole-body dynamics and control, *2012 12th IEEE-RAS Int. Conf. Humanoid Robots (Humanoids)*, November 2012, pp. 36–42.
41. T. Ohmichi, S. Hosaka, M. Nishihara, T. Ibe, A. Okino, J. Nakayama, T. Miida and M. Ishida, Development of the multi-function robot for the containment vessel of the nuclear plant, *Int. Conf. Advanced Robotics*, Vol. 19, No. 20 (1985).
42. S. Hirose, A. Morishima, S. Tukagosi, T. Tsumaki and H. Monobe, Design of practical snake vehicle: articulated body mobile robot KR-II, *Fifth Int. Conf. Advanced Robotics, 1991. 'Robots in Unstructured Environments', 91 ICAR.*, Vol. 1, June 1991, pp. 833–838.
43. N. Eiji and N. Sei, Leg-wheel robot: A futuristic mobile platform for forestry industry, in *1993 IEEE/Tsukuba Int. Workshop on Advanced Robotics, 1993. Can Robots Contribute to Preventing Environmental Deterioration? Proc.*, November 1993, pp. 109–112.
44. O. Matsumoto, S. Kajita, K. Tani and M. Oooto, A four-wheeled robot to pass over steps by changing running control modes, in *1995 IEEE Int. Conf. Robotics and Automation, 1995. Proc.*, Vol. 2, May 1995, pp. 1700–1706.
45. T. Asfour, K. Berns and R. Dillmann, The humanoid robot ARMAR: Design and control, *IN IEEE/APS INTL. Conf. Humanoid Robots* (2000), pp. 7–8.
46. D. Katz, E. Horrell, Y. Yang, B. Burns, T. Buckley, A. Grishkan, V. Zhylkovskyy, O. Brock and E. Learned-Miller, The umass mobile manipulator uman: An experimental platform for autonomous mobile manipulation, in *Workshop on Manipulation in Human Environments at Robotics: Science and Systems.*, 2006.
47. C. Loughlin, A. AlbuSchffer, S. Haddadin, C. Ott, A. Stemmer, T. Wimbck and G. Hirzinger, The DLR lightweight robot: Design and control concepts for robots in human environments, *Indus. Robot: An Int. J.* **34**(5) (2007) 376–385.
48. C. Borst, C. Ott, T. Wimbock, B. Brunner, F. Zacharias, B. Bauml, U. Hillenbrand, S. Haddadin, A. Albu-Schaffer and G. Hirzinger, A humanoid upper body system for two-handed manipulation, in *2007 IEEE Int. Conf. Robotics and Automation*, April 2007, pp. 2766–2767.

49. D. Theobold, J. Ornstein, J. G. Nichol and S. E. Kullberg, Mobile robot platform google patents, US Patent, 7,348,747 (2008).
50. G. Freitas, F. Lizarralde, L. Hsu and N. Reis, Kinematic reconfigurability of mobile robots on irregular terrains, in *IEEE Int. Conf. Robotics and Automation, 2009. ICRA'09*, May 2009, pp. 1340–1345.
51. M. Beetz, L. Mosenlechner and M. Tenorth, CRAM — A cognitive robot abstract machine for everyday manipulation in human environments, *2010 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, October 2010, pp. 1012–1017.
52. H. Iwata and S. Sugano, Design of human symbiotic robot TWENDY-ONE, *IEEE Int. Conf. Robotics and Automation, 2009. ICRA '09*, May 2009, pp. 580–586.
53. U. Reiser, C. Connette, J. Fischer, J. Kubacki, A. Bubeck, F. Weisshardt, T. Jacobs, C. Parlitz, M. Hagele and A. Verl, Care-o-bot 3 — Creating a product vision for service robot applications by integrating design and technology, *IEEE/RSJ Int. Conf. Intelligent Robots and Systems, 2009. IROS 2009*, October 2009, pp. 1992–1998.
54. C.-H. King, T. L. Chen, A. Jain and C. C. Kemp, Towards an assistive robot that autonomously performs bed baths for patient hygiene, *IROS* (IEEE, 2010), pp. 319–324.
55. B. Stephens and C. Atkeson, Dynamic balance force control for compliant humanoid robots, *2010 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, October 2010, pp. 1248–1255.
56. M. Stilman, J. Olson and W. Gloss, Golem krang: Dynamically stable humanoid robot for mobile manipulation, *2010 IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2010, pp. 3304–3309.
57. W. Meeussen, M. Wise, S. Glaser, S. Chitta, C. McGann, P. Mihelich, E. Marder-Eppstein, M. Muja, V. Eruhimov, T. Foote, J. Hsu, R. Rusu, B. Marthi, G. Bradski, K. Konolige, B. Gerkey and E. Berger, Autonomous door opening and plugging in with a personal robot, in *2010 IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2010, pp. 729–736.
58. S. Hart, J. Yamokoski and M. Diftler, Robonaut 2: A new platform for human-centered robot learning, *Robot. Sci. Syst.* Workshop on Mobile Manipulation: Learning to Manipulate (2011).
59. B. J. Stephens, Push recovery control for force-controlled humanoid robots, Ph.D. dissertation, Carnegie Mellon University (2011).
60. F. Moro, N. Tsagarakis and D. Caldwell, A human-like walking for the COmpliant huMANoid COMAN based on CoM trajectory reconstruction from kinematic motion primitives, *2011 11th IEEE-RAS Int. Conf. Humanoid Robots (Humanoids)*, October 2011, pp. 364–370.
61. N. Tsagarakis, Z. Li, J. Saglia and D. Caldwell, The design of the lower body of the compliant humanoid robot "cCub", *2011 IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2011, pp. 2035–2040.
62. J. Smith, S. Bertrand, P. Neuhaus and J. Pratt, Momentum-based control framework: Application to the humanoid robot atlas and valkryie, IROS 2014 Workshop on Whole-Body control for Robots in the Real World (2014) Chicago, IL, pp. 1–59.
63. A. Herzog, L. Righetti, F. Grimminger, P. Pastor and S. Schaal, Balancing experiments on a torque-controlled humanoid with hierarchical inverse dynamics, in *Proc. 2014 IEEE/RSJ Int. Conf. Intelligent Robots and Systems* Chicago, IL (2014), pp. 981–988.
64. A. Herzog, L. Righetti, F. Grimminger, P. Pastor and S. Schaal, Momentum-based balance control for torque-controlled humanoids, *CoRR*, Vol. abs/1305.2042 (2013), Available at http://arxiv.org/abs/1305.2042.
65. M. Hutter, C. Gehring, M. Bloesch, M. A. Hoepflinger, C. D. Remy and R. Siegwart, StarlETH: A compliant quadrupedal robot for fast, efficient, and versatile locomotion,

15th Int. Conf. Climbing and Walking Robot — CLAWAR 2012, John Hopkins University, Baltimore, Mayland, (2012), pp. 483–491.
66. M. Hutter, H. Sommer, C. Gehring, M. Hoepflinger, M. Bloesch and R. Siegwart, Quadrupedal locomotion using hierarchical operational space control, *Int. J. Robot. Res.* **33**(8) (2014) 1047–1062.
67. M. Fuchs, C. Borst, P. Giordano, A. Baumann, E. Kraemer, J. Langwald, R. Gruber, N. Seitz, G. Plank, K. Kunze, R. Burger, F. Schmidt, T. Wimboeck and G. Hirzinger, Rollin' justin — Design considerations and realization of a mobile platform for a humanoid upper body, *IEEE Int. Conf. Robotics and Automation, 2009. ICRA'09*, May 2009, pp. 4131–4137.
68. C. Semini, N. G. Tsagarakis, E. Guglielmino, M. Focchi, F. Cannella and D. G. Caldwell, Design of HyQ — A hydraulically and electrically actuated quadruped robot, *Proc. Inst. Mech. Eng. I: J. Syst. Control Eng.* **225**(6) (2011) 831–849.
69. R. Tellez, F. Ferro, S. Garcia, E. Gomez, E. Jorge, D. Mora, D. Pinyol, J. Oliver, O. Torres, J. Velazquez and D. Faconti, Reem-B: An autonomous lightweight human-size humanoid robot, *8th IEEE-RAS Int. Conf. Humanoid Robots, 2008. Humanoids 2008*, December 2008, pp. 462–468.
70. R. Philippsen, L. Sentis and O. Khatib, An open source extensible software package to create whole-body compliant skills in personal mobile manipulators, in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)* (San Francisco, CA, 2011), pp. 1036–1041.
71. Human Centered Robotics Laboratory at the University of Texas at Austin, Uta-wbc (2015), Available at https://github.com/lsentis/uta-wbc-dreamer (accessed on 13 February 2015).
72. G. T. Heineman and W. T. Councill (eds.), *Component-Based Software Engineering: Putting the Pieces Together* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001).
73. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd edn. (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002).
74. F. Kanehiro, K. Fujiwara, S. Kajita, K. Yokoi, K. Kaneko, H. Hirukawa, Y. Nakamura and K. Yamane, Open architecture humanoid robotics platform, *IEEE Int. Conf. Robotics and Automation, 2002. Proc. ICRA'02*, Vol. 1 (2002), pp. 24–30.
75. H. Hirukawa, F. Kanehiro, K. Kaneko, S. Kajita, K. Fujiwara, Y. Kawai, F. Tomita, S. Hirai, K. Tanie, T. Isozumi, K. Akachi, T. Kawasaki, S. Ota, K. Yokoyama, H. Handa, Y. Fukase, J. Ichiro Maeda, Y. Nakamura, S. Tachi and H. Inoue, Humanoid robotics platforms developed in {HRP}, *Robot. Auton. Syst.* **48**(4) (2004) 165–175.
76. N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku and W.-K. Yoon, RT-middleware: Distributed component middleware for rt (robot technology), *2005 IEEE/RSJ Int. Conf. Intelligent Robots and Systems, 2005. (IROS 2005)*, August 2005, pp. 3933–3938.
77. Orocos, Orocos toolchain (2015), Available at http://www.orocos.org/toolchain (accessed on 13 February 2015).
78. G. Metta, P. Fitzpatrick and L. Natale, YARP: Yet another robot platform, *Int. J. Adv. Robot. Syst.* **3**(1) (2006) 43–48.
79. Robot Operating System, Ros (2015), Available at http://www.ros.org/ (accessed on 13 February 2015).
80. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler and A. Y. Ng, ROS: An open-source robot operating system, *ICRA Workshop on Open Source Software*, Vol. 3, No. 3.2 (2009) p. 5.

81. I. A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Dias-Caldron, T. Estlin, R. Madison, J. Guineau, M. McHenry, I. Shu, and D. Apfelbaum, CLARAty: Challenges and steps toward reusable robotic software, *Int. J. Adv. Robot. Syst.* **3**(1) (2006) 23–30.
82. I. A. Nesnas, CLARAty: A collaborative software for advancing robotic technologies, in *Proc. NASA Science and Technology Conf.*, June 2007. University of Maryland, College Park, Maryland, pp. 1–7.
83. G. Hirzinger and B. Bauml, Agile robot development (aRD): A pragmatic approach to robotic software, *2006 IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, October 2006, pp. 3741–3748.
84. Microblx, Microblx — A lightweight, dynamic, reflective, hard real-time safe function block framework (2014), Available at http://www.microblx.org/ (accessed on 13 February 2015).
85. M. Klotzbuecher and H. Bruyninckx, Microblx: A reflective, real-time safe, embedded function block framework, in *15th Real Time Linux Workshop*, Scuola Universitaria Professionale della Suizzera Italiana in Lugano-Manno, Switzerland, pp. 1–1. October 2013.
86. RoCoCo Laboratory, Open robot development kit (2015), Available at http://openrdk.sourceforge.net/ (accessed on 13 February 2015).
87. D. Calisi, A. Censi, L. Iocchi and D. Nardi, OpenRDK: A modular framework for robotic software development, *IEEE/RSJ Int. Conf. Intelligent Robots and Systems, 2008. IROS 2008*, September 2008, pp. 1872–1877.
88. D. Calisi, A. Censi, L. Locchi and D. Nardi, Design choices for modular and flexible robotic software development: The OpenRDK viewpoint, *J. Softw. Eng. Robot.* **3**(1) (2012) 13–27.
89. M. Munich, J. Ostrowski and P. Pirjanian, ERSP: A software platform and architecture for the service robotics industry, *2005 IEEE/RSJ Int. Conf. Intelligent Robots and Systems, 2005 (IROS 2005)*, August 2005, pp. 460–467.
90. Dipartimento Di Scienze e Tecnologie Aerospaziali del Politecnico di Milano, Real-time application interface (2015), Available at https://www.rtai.org/ (accessed on 13 February 2015).
91. A. R. Tsouroukdissian, ros-control: An overview, *ROSCon 2014*, Chicago, IL September 2014, pp. 1–123.
92. J. Bohren, Conman — A robot state estimator and controller manager for use in orocos rtt and ros (2015), Available at https://github.com/jbohren/conman (accessed on 13 February 2015).
93. J. De Schutter, T. De Laet, J. Rutgeerts, W. Decr, R. Smits, E. Aertbelin, K. Claes and H. Bruyninckx, Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty, *Int. J. Robot. Res.* **26**(5) (2007) 433–455.
94. W. Decre, R. Smits, H. Bruyninckx and J. De Schutter, Extending iTaSC to support inequality constraints and non-instantaneous task specification, *IEEE Int. Conf. Robotics and Automation, 2009. ICRA'09*, May 2009, pp. 964–971.
95. W. Decre, H. Bruyninckx and J. De Schutter, Extending the iTaSC constraint-based robot task specification framework to time-independent trajectories and user-configurable task horizons, *2013 IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2013, pp. 1941–1948.
96. Rock Robotics, Rock — The robot construction kit (2015), Available at http://rock-robotics.org/stable/ (accessed on 13 February 2015).
97. B. Brunner, K. Landzettel, G. Schreiber, B. Stinmetz and G. Girzinger, A universal task level ground control and programming system for space robot applications — the

MARCO concept and its application to the ets vii project, in *Proc. 5th iSAIRAS Int. Symp. Artifical Intelligence, Robotics, and Automation in Space*, Noordwijk, The Netherlands (1999), pp. 507–514.
98. S. Fleury, M. Herrb and R. Chatila, GenoM: A tool for the specification and the implementation of operating modules in a distributed robot architecture, in *Proc. 1997 IEEE/RSJ Int. Conf. Intelligent Robots and Systems, 1997. IROS'97*, Vol. 2, September 1997, pp. 842–849.
99. Willow Garage, Ecto — a c++/python computation graph framework (2015), Available at http://plasmodic.github.io/ecto/ (accessed on 13 February 2015).
100. S. Hart, P. Dinh, J. Yamokoski, B. Wightman and N. Radford, Robot task commander: A framework and IDE for robot application development, *2014 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS 2014)*, September 2014, pp. 1547–1554.
101. R. Arkin and R. Murphy, Autonomous navigation in a manufacturing environment, *IEEE Trans. Robot. Autom.* **6**(4) (1990) 445–454.
102. R. Alami, R. Chatila, S. Fleury, M. Ghallab and F. Ingrand, An architecture for autonomy, *Int. J. Robot. Res.* **17** (1998) 315–337.
103. O. C. Jenkins and M. J. Matari, Performance-derived behavior vocabularies: Data-driven acquisition of skills from motion, *Int. J. Humanoid Robot.* **1**(2) (2004) 237–288.
104. K. Kawamura, R. A. Peters II, R. E. Bodenheimer, N. Sarkar, J. Park, C. A. Clitton, A. W. Spratley and K. A. Hambuchen, A parallel distributed cognitive control system for a humanoid robot, *Int. J. Humanoid Robot.* **1**(1) (2004) 65–93.
105. P. Pastor, H. Hoffmann, T. Asfour and S. Schaal, Learning and generalization of motor skills by learning from demonstration, *IEEE Int. Conf. Robotics and Automation, 2009. ICRA'09*, May 2009, pp. 763–768.
106. K. Kim, J.-Y. Lee, D. Choi, J.-M. Park and B.-J. You, Autonomous task execution of a humanoid robot using a cognitive model, *2010 IEEE Int. Conf. Robotics and Biomimetics (ROBIO)*, December 2010, pp. 405–410.
107. C. Ott, B. Henze and D. Lee, Kinesthetic teaching of humanoid motion based on whole-body compliance control with interaction-aware balancing, *2013 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, November 2013, pp. 4615–4621.
108. R. Simmons and D. Apfelbaum, A task description language for robot control, in *1998 IEEE/RSJ Int. Conf. Intelligent Robots and Systems, 1998. Proc.*, Vol. 3, October 1998, pp. 1931–1937.
109. D. Kortenkamp, R. Burridge, R. P. Bonasso, D. Schreckenghost and M. B. Hudson, An intelligent software architecture for semiautonomous robot control, *Autonomy Control Software Workshop, Autonomous Agents 99* (1999), pp. 36–43.
110. H. Lim, Y. Kang, Y. Lee, J. Kim and B.-J. You, Software architecture and task definition of a multiple humanoid cooperative control system, *Int. J. Humanoid Robot.* **6**(2) (2009) 173–203.
111. Robot Operating System, Ros pluginlib (2015), Available at http://wiki.ros.org/pluginlib (accessed on 13 February 2015).
112. Martin Felis, Rigid body dynamics library (2015), Available at http://rbdl.bitbucket.org/ (accessed on 13 February 2015).
113. J. James, Ros shared memory interface (2015), Available at https://bitbucket.org/jraipxg/ros_shared_memory_interface (accessed on 13 February 2015).
114. Ingo Berg, Muparser — A fast math library (2015), Available at http://muparser.beltoforion.de/ (accessed on 13 February 2015).
115. Robot Operating System, Ros launch (2014), Available at http://wiki.ros.org/roslaunch (accessed on 13 February 2015).

116. Robot Operating System, Ros bag (2015), Available at http://wiki.ros.org/rosbag (accessed on 13 February 2015).
117. Open Source Robotics Foundation, Gazebo simulator website (2015), Available at http://gazebosim.org/ (accessed on 13 February 2015).
118. Robot Operating System, Ros client libraries (2015), Available at http://wiki.ros.org/Client%20Libraries (accessed on 13 February 2015).
119. N. A. Radford, P. Strawser, K. Hambuchen, J. S. Mehling, W. K. Verdeyen, S. Donnan, J. Holley, J. Sanchez, V. Nguyen, L. Bridgwater, R. Berka, R. Ambrose, C. McQuin, J. D. Yamokoski, S. Hart, R. Guo, A. Parsons, B. Wightman, P. Dinh, B. Ames, C. Blakely, C. Edmonson, B. Sommers, R. Rea, C. Tobler, H. Bibby, B. Howard, L. Nui, A. Lee, M. Conover, L. Truong, D. Chesney, R. P. Jr., G. Johnson, C.-L. Fok, N. Paine, L. Sentis, E. Cousineau, R. Sinnet, J. Lack, M. Powell, B. Morris and A. Ames, Valkyrie: NASA's first bipedal humanoid robot, *J. Field Robot.* **10** (2014) 397–419.
120. D. Kim, Y. Zhao, G. Thomas and L. Sentis, Accessing whole-body operational space control in a point-foot series elastic biped: Balance on split terrain and undirected walking, preprint (2015), Available at http://arxiv.org/abs/1501.02855.
121. Y. Zhao, N. Paine, K. Kim and L. Sentis, Stability and performance limits of latency-prone distributed feedback controllers, *IEEE Trans. Ind. Electron.* (2015), Available at http://arxiv.org/pdf/1501.02854v1.pdf.
122. M. Johnson, B. Shrewsbury, S. Bertrand, T. Wu, D. Duran, M. Floyd, P. Abeles, D. Stephen, N. Mertins, A. Lesman, J. Carff, W. Rifenburgh, P. Kaveti, W. Straatman, J. Smith, M. Griffioen, B. Layton, T. de Boer, T. Koolen, P. Neuhaus and J. Pratt, Team IHMC's lessons learned from the DARPA robotics challenge trials, *J. Field Robot.* **32**(2) (2015) 192–208.
123. M. Nikoli, B. A. Borovac, M. Rakovi and S. Savi, A futher generatlization of task-oriented control through task prioritization, *Int. J. Humanoid Robot.* **10**(3) (2013) 1350012.
124. O. Khatib, A unified approach for motion and force control of robot manipulators: The operational space formulation, *IEEE J. Robot. Autom.* **RA-3**(1) (1987) 43–53.
125. Y. Liu, J. Yang and Z. Wu, Ubiquitous and cooperative network robot system within a service framework, *Int. J. Humanoid Robot.* **8**(1) (2011) 147–167.
126. C. Gaskett, A. Ude and G. Cheng, Hand-eye coordination through endpoint closed-loop and learned endpoint open-loop visual servo control, *Int. J. Humanoid Robot.* **2**(2) (2005) 203–224.
127. P. Kaiser, N. Vahrenkamp, F. Schltje, J. Borrs and T. Asfour, Extraction of whole-body affordances for loco-manipulation tasks, *Int. J. Humanoid Robot.* (2015) 1550031.
128. O. Brock, A. Fagg, R. Grupen, R. Platt, M. Rosenstein and J. Sweeney, A framework for learning and control in intelilgent humanoid robots, *Int. J. Humanoid Robot.* **2**(3) (2005) 301–336.
129. M. Gonzlez-Fierro, C. Balaguer, N. Swann and T. Nanayakkara, Full-body postural control of a humanoid robot with both imitation learning and skill innovation, *Int. J. Humanoid Robot.* **11**(2) (2014) 1450012.

Chien-Liang Fok received two B.S. degrees in computer engineering and computer science and a Ph.D. degree in computer science all from Washington University in St. Louis. His Ph.D. studies focused on highly adaptive middleware for wireless sensor networks. From 2010–2012, he was a postoctoral fellow at UT Austin where he managed a mobile swarm robotics laboratory. From 2013–2015, he was a research fellow in the Human Centered Robotics Lab at UT Austin where he participated in the DARPA Robotics Challenge with NASA JSC. His research interests include advanced software architectures for human-centered robotics, cloud robotics, robot decision-making software, and real-time control software.

Gwendolyn Johnson received M.S. and Ph.D. degrees in Aeronautics from the California Institute of Technology, where her focus was on developing simulation techniques and optimal controllers for under-actuated systems to take advantage of contact and impact mechanics. The primary application of this work was to self-assembling large space apertures. As a research fellow at UT Austin's Human Centered Robotics Laboratory and a member of the NASA-JSC team for the DARPA robotics challenge, she developed whole-body compliant controllers for humanoid robots subject to changing contact constraints. She has since moved from optimizing robots to optimizing athletes as an engineer at Under Armour Connected Fitness.

John D. Yamokoski received his M.S. and Ph.D. degrees in mechanical engineering from University of Florida. From 2010–2012, he worked for Oceaneering Space Systems where he served as controls lead for NASA's Robonaut 2 (R2) project. From 2012 to 2014, he was Chief Software Architect at the Florida Institute for Human Machine Cognition (IHMC) and a Research Scientist at NASA Johnson Space Center (JSC) where he lead the software development effort for NASA JSC's entry into the DARPA Robotics Challenge. He is currently Director of Research and Development at Houston Mechantronics, Inc.

Aloysius K. Mok received his B.S. degree in electrical engineering, M.S. degree in computer science, and a Ph.D. degree is computer science, all from the Massachusetts Institute of Technology. Since 1983, he has been on the faculty of the computer science department at the University of Texas at Austin where he is currently the Quincy Lee Centennial Professor of computer science. Mok has done extensive research on the technical problems of hard-real-time system design, and has consulted for both government and industry on software engineering issues of large embedded systems. His current interests include real-time and embedded systems, fault-tolerant and secure system design, network-centric computing, and cyber-physical systems. He is past chair of the Technical Committee on Real-Time Systems of the Institute of Electrical and Electronic Engineers.

Luis Sentis received his M.S. and Ph.D. degrees in electrical engineering from Stanford University, where he formulated leading work in theoretical and computational methods for the compliant control of humanoid robots. Prior to that, he worked in Silicon Valley in the area of clean room automation. He is currently an Assistant Professor at The University of Texas at Austin, where he directs the Human Centered Robotics Laboratory. He was the UT Austin's Lead for DARPA's Robotics Challenge entry with NASA Johnson Space Center in 2013. His research focuses on foundations for the compliant control of humanoid robots, algorithms to generate extreme dynamic locomotion, and building humanoid robots for human–robot interactions.